POLITECNICO DI TORINO

Master degree course in Computer Engineering

Tesi di Laurea Magistrale

Audio Conferencing Over HTTP3 Benchmark

Supervisor
Prof. Antonio Servetti

Candidate
Toni Saliba
Matricola: 275112

ACADEMIC YEAR 2021 – 2022

To my parents, Jana, Pierre and lifelong

friends who supported me during these years.

Summary

We are witnessing significant adoption of web-based audio/video communication platforms that run in a browser and can be easily integrated with the web environment. Most of these solutions are video conferencing software that follows a peer-2-peer architecture using webRTC. This is based on the fact that there are some hurdles and blockers when building a client-server videoconferencing solution that limits the software from achieving real-time, low-latency communication. However, the addition of the new protocols HTTP/3 and WebTransport has changed the way the internet works. They claim to solve the latency issues that older protocols were facing. So, it is interesting to see how well those new protocols will perform in an audio/video communication context. This master thesis focuses on analyzing the performance of HTTP/3 and WebTransport in a client-server audioconferencing context and discusses the possible advantages of adopting these protocols.

Audio is extracted from a client's mic, compressed and encoded, and then sent through the internet as a client request using the WebTransport Datagram API over HTTP/3, which ignores reliability. That audio is received by an HTTP/3 server, decoded, and processed in a circular buffer that holds all audio data for streaming purposes. Finally, audio is encoded back as a response to the client, who will decode the audio and play the corresponding sounds. This solution is being compared with another solution that implements the same architecture and the same design but uses a different communication protocol, HTTP/1.

The comparison between the two approaches is carried out by considering different configurations of a server-side delay that's being added to ensure good audio quality for clients,

measuring the round-trip latency each implementation leads to, and measuring the sound latency that each configuration and protocol lead to.

The measurements showed promising results in favor of HTTP/3 and WebTransport by showing that the HTTP/3 server-side delay needed for good audio quality is much smaller than the one for HTTP/1, also by showing that the round-trip latency for HTTP/1 is significantly smaller than the one for HTTP/1. Finally, the sound-latency measurements showed that HTTP/3 is much more efficient than HTTP/1 when building real-time low-latency audioconferencing solutions. It also showed we can now adopt client-server architectures in real-time low-latency contexts due to HTTP/3 showing low sound latency results that were lower than the low-latency communication threshold of 300 milliseconds.

So, we can consider that HTTP/3 and WebTransport will be a revelation for low-latency solutions, and they will allow the adoption of client-server architectures in low-latency contexts.

Acknowledgments

First and foremost, I would like to thank my supervisor, Professor Antonio Servetti. He has guided me from the very beginning of this project with a great deal of patience and an astonishing amount of knowledge and expertise. Being able to discuss and share ideas with Professor Servetti on a regular basis has not only benefited my thesis but also had a positive impact on how I view things and approach matters academically.

I would like to thank Politecnico di Torino and everyone involved for giving me this opportunity to pursue my master's degree as a double-degree international student. I am also grateful for my former professors and mentors at the Lebanese University who forged my way forward.

A special thanks goes out to my parents, my sisters, my dear friend Pierre and each and every one who stood beside me during these five years. It is through their dedication and unwavering support that I was able to achieve all that I have.

I am also deeply grateful to my girlfriend Jana, for all her love and support. She’s the one that encouraged me to make it till the end.

Contents

List of Figures

List of Tables

"Good communication is the bridge between confusion and clarity."
- Nat Turner.

Chapter 1

Introduction

Technological advancements have revolutionized how and where we conduct business. Remote working, the globalization of business, and instant on-demand communication are significant examples.

Communication with partners, internal teams, suppliers, and investors is critical to business growth. That is where video conferencing comes into play. Not only does it boost productivity and save time and money, but it also promotes overall collaboration without the need to travel and have face-to-face communication.

Video conferencing is an ongoing challenge in how digital human interactions are produced, consumed, and distributed via the network. From a technical point of view, It is a real-time data web application challenge.

There are a couple of different ways to architect and design the infrastructure of a solution that can help us solve this real-time challenge. We will narrow our discussion to a peer-to-peer approach and a client-server approach. Those two approaches represent telecommunication

networks that transfer information from the source to the destination without minimal transmission loss.

Peer-to-peer networks follow a distributed application architecture that partitions tasks and workloads between peers. Peers will act as servers and clients, which poses challenges from a computer security perspective. This architecture works best for simple real-time data web applications with two to four concurrent participants. No server is needed, so the costs for setting up and maintaining that infrastructure are significantly lower than its competition.

Client-server networks follow a distributed application architecture that partitions tasks or workloads between servers and clients. In this network, clients and servers exchange messages in a request-response pattern. The client will send a request, and the server will return a response. Due to its centralized design, it is easier to manage security, and it is easier to scale. This architecture will allow you to build real-time data web applications that can scale to a large number of concurrent participants. There is no limit to the number of participants this architecture can handle. The more you would like to scale, the more you would need to invest in your servers. However, the costs needed to set up this architecture are much higher than peer-to-peer.

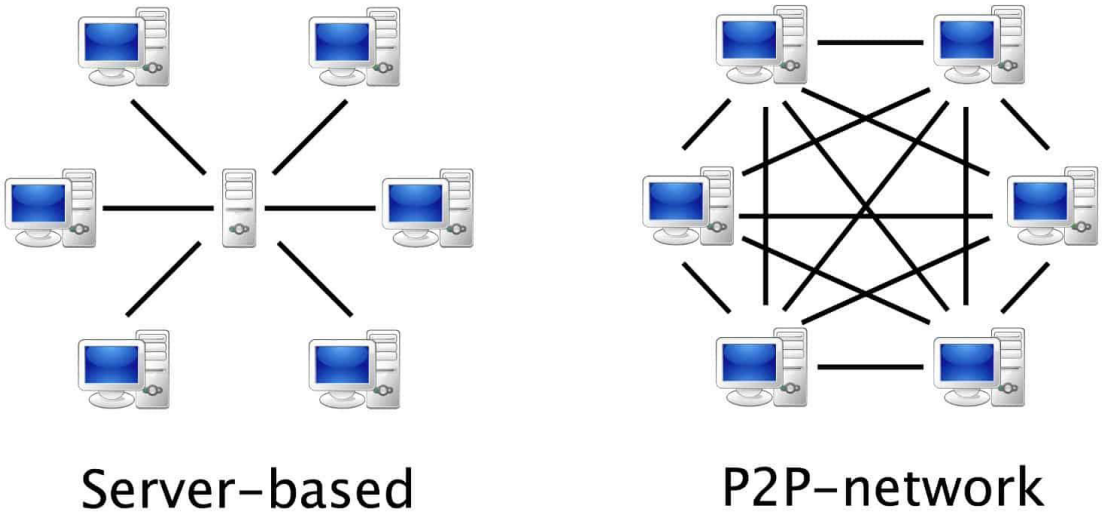

Figure 1.1. Client-Server vs. P2P Architecture

We have protocols on the web that can help us build the communication layer for the models described above. We have HTTP1.X/2 (1) (2)and HTTP3 (3), excellent protocols that have been in place for over two decades now, and they have scaled tremendously, but they are primarily file-based. You either get an object, or you post an object that can be used to handle sequential flows. For interactive sequential flows, we have WebSockets (4). HTTP1.x/2 and WebSockets are suitable protocols, but they exhibit head-of-line blocking (5) and lack low latency (real-time) data transport due to their TCP-based (6) nature. For those sequential flows of real-time data, we have WebRTC. We can consider it a collection or a stack of many different protocols that work together to provide the capabilities of peer-to-peer-based communication. WebRTC (7) is a tightly-bound stack of peer-to-peer session establishment, control & delivery protocols, codecs, and logic. One essential feature WebRTC does not cover sending Secure streams of real-time data. Hence a new real-time transport for the web is needed. So, this is where WebTransport (8) comes in.

WebTransport solves the real-time data problem for the internet. It is a transport protocol (specified by IETF (9)) and an easy-to-use Web API (specified by the W3C) that enables clients operating under the Web security model to communicate with a remote server using a secure, multiplexed, real-time transport.

WebTransport provides multiple uni-directional and bi-directional streams of reliable and ordered data, it also provides an unreliable flow of UDP-like (10) datagrams, and it operates over HTTP/3 with fallback over HTTP/2. It is essential to mention that WebTransport is not a UDP Socket API. While WebTransport uses HTTP/3, which uses UDP "under the hood," WebTransport has encryption and congestion control requirements that make it more than a basic UDP socket API.

In some cases, WebTransport can be considered a replacement for WebSockets. WebSockets communications are modeled around a single ordered stream of reliable messages, a type of communication covered by WebTransport's streams APIs that provide reliable and ordered data transfer but more efficiently. It is more efficient because multiple WebTransport streams are analogous to establishing multiple TCP connections. However, HTTP/3 uses the lighter-weight QUIC (11) protocol under the hood so that they can be opened and closed without much overhead.

In comparison, WebTransport datagram APIs provide low-latency delivery without guarantees about reliability or ordering, so WebTransport cannot be considered a direct replacement for WebSockets.

WebTransport is also considered an alternative to WebRTC data channels for client-server connections. WebTransport shares many properties as WebRTC data channels, although the

underlying protocols differ. Generally, running an HTTP3/-compatible server requires less setup and configuration than maintaining a WebRTC server, which involves understanding multiple protocols like ICE, DTLS, and SCTP (12) to get a working transport. WebRTC entails many moving pieces that could lead to failed client/server negotiations. The WebTransport API was designed with web developer use cases in mind and should feel more like writing modern web platform code than webRTC/s data channel interfaces. Unlike WebRTC, WebTransport is supported inside Web Workers, allowing you to perform client-server communications independent of a certain HTML page. And since WebTransport exposes a Streams-compliant interface, it supports optimizations around backpressure.

Additionally, WebTransport supports sending data unreliably via its datagram APIs. WebTransport Datagrams are ideal for sending and receiving data that do not need reliability. Individual packets of data are limited in size by the maximum transmission unit (MTU) (13) of the underlying connection and may or may not be transmitted successfully. If those packets of data were transferred, they may arrive in any order. Having said that, this makes the datagram APIs ideal for low-latency, best-effort data transmission. You can consider datagrams as user datagram protocol (UDP) messages, but encrypted and congestion-controlled.

In this project, we will tackle the challenges that real-time data web applications face, and we will find, discuss and analyze different solutions and new technologies.

We have developed two real-time data web applications to provide a solution for audio conferencing software. The difference between the two is the communication layer, the main one is using WebTransport over HTTP/3, and the other used for comparison purposes is using HTTP/1.

In the following chapters, we are going to discuss the architecture that we followed in order to build these solutions. We are going to discuss how we solved the real-time data challenge that was introduced earlier. We are going to discuss and compare the test results that were produced from both web applications, and finally, we are going to conclude and reason all the arguments that were discussed during this report.

Chapter 2

Application Technical Design

This application follows a client-server design where the client nodes represent the users of this app, and the server is a centralized application that connects the clients together.

For the HTTP/3 implementation, the application's communication layer is based on WebTransport over HTTP/3 with the use of its Datagram API in order to implement a secure low, latency real-time data exchange.

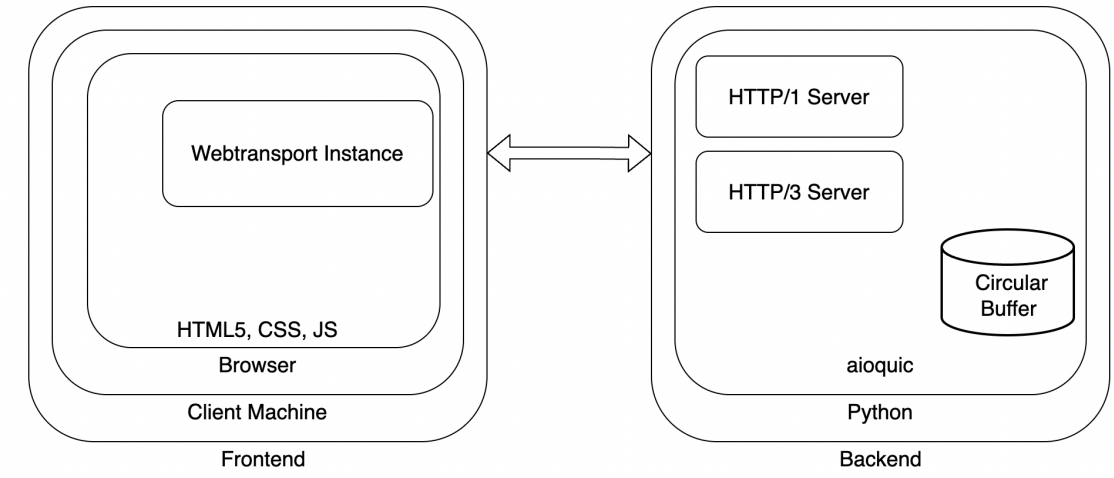

Figure 2.1. Web Application Architecture

The front end was implemented using HTML5, CSS, and Javascript, and it's being served over an HTTP Python server. The backend is an HTTP/3 python server built using the aioquic[1] library. Aioquic is a library for the QUIC network protocol. It features a minimal TLS 1.3 (14) implementation and an HTTP/3 stack. You can find the code used to implement the HTTP/3 python server in annex 1.4. You can also find that code snippet in `server_wrapper.py`.

We also have an HTTP/1 server next to the HTTP/3 server, which is only used for configuration. The clients use that server to fetch configuration values like `server_sample_rate`.

We are also using the OPUS (15) library for compressing audio samples because OPUS is a lossy audio coding format designed to efficiently code speech and general audio in a single format while remaining low-latency enough for real-time interactive communication and low-complexity enough for low-end embedded processors.

2.1 2.1 - Client and Server Connection and Startup

When the client node starts, it will need to connect to the server. It does that by creating a WebTransport instance and pointing it to the server by using the correct URI and port. The server, listening for CONNECT methods with WebTransport protocols on a certain port, will initiate a WebTransport handshake after receiving a connect request. It's important to mention

[1] You can find the aioquic library in this repo: https://github.com/aiortc/aioquic

that since the server is an HTTP/3 server and the HTTP/3 protocol always operates using TLS, running WebTransport over an HTTP/3 server requires a valid TLS certificate. We generated a certificate and a private key using OpenSSL (16) and RSA (17) as a cryptographic algorithms.

After a successful handshake, the client node will get metadata from the server. The most important things fetched now are the `server_clock` and the `server_sample_rate`. The `server_clock` and the `server_sample_rate` are fetched using the `query_server_clock(target_url)` function which is referenced in Annex 1.5. That function will make an HTTP/1 GET request and will return the `server_clock` and `server_sample_rate` in its response Headers, and the client will do some processing to estimate the correct `server_clock`.

The client node will use this information to set up its inner state clock and configure its write clock. It's important for all clients to have a synchronized `write_clock`. We will discuss this more in the following sections.

2.2 Client-Side Request Process

After successfully starting up the client and the server and connecting them together, and adjusting the `server_clock` estimation and the states, it's time to start the low-latency audio-conferencing functionality. For that to happen, the client will start reading audio from the mic. To do that, the client will add the `audio-worklet.js` bundle to the audio context. This module implements some functions that will assist us in handling audio output and input. The

main function that we're using is to handle sending and playing audio processes `(inputs, outputs)`[2], where `inputs` represent the input audio that comes from the mic, and `outputs` represent the output audio that should be played by the speakers.

We should also mention that we're handling stereo output by cloning mono output.

```
for (var chan = 1; chan < outputs[0].length; chan++) {
  outputs[0][chan].set(outputs[0][0]);
}
```

Listing 2.1. Code Snippet for Cloning Mono Output to Produce Stereo Output

After getting the input audio from the mic, we encode and compress them using OPUS by using the function `encode_chunk(chunk)`[3], and then we call the `samples_to_h3_server`[4] functions to encode the audio samples to 8-bit unsigned integers and send them as a datagram using the WebTransport datagram API.

```
let encoder = new TextEncoder("utf-8");
const writer = transport.datagrams.writable.getWriter();
writer.write(encoder.encode(totalOutputArray));
```

Listing 2.2. Code Snippet for Opening WebTransport Datagram Writer

The data that's sent in the datagram contains the audio samples with some client metadata like `user_name`, `write_clock` and number of samples. The `write_clock` is used by the client to give directions to the server on what index to use when writing and reading audio from his in-memory audio store, which is a circular buffer. And finally, the number of samples is used for indexing and ease of programming.

[2] You can find that function in audio-worklet.js
[3] You can find that function in app.js, line 178
[4] You can find that function in net.js line 456

After the audio sample has been sent by the client, the client will now wait for a response from the server. And this is what we mean by a request-response model. For every request, there must be a response, and with every request, we are posting audio samples and metadata data to the server. With every response from the server, the client will receive new metadata and new audio samples that are the same amount of samples as the audio samples that were originally sent.

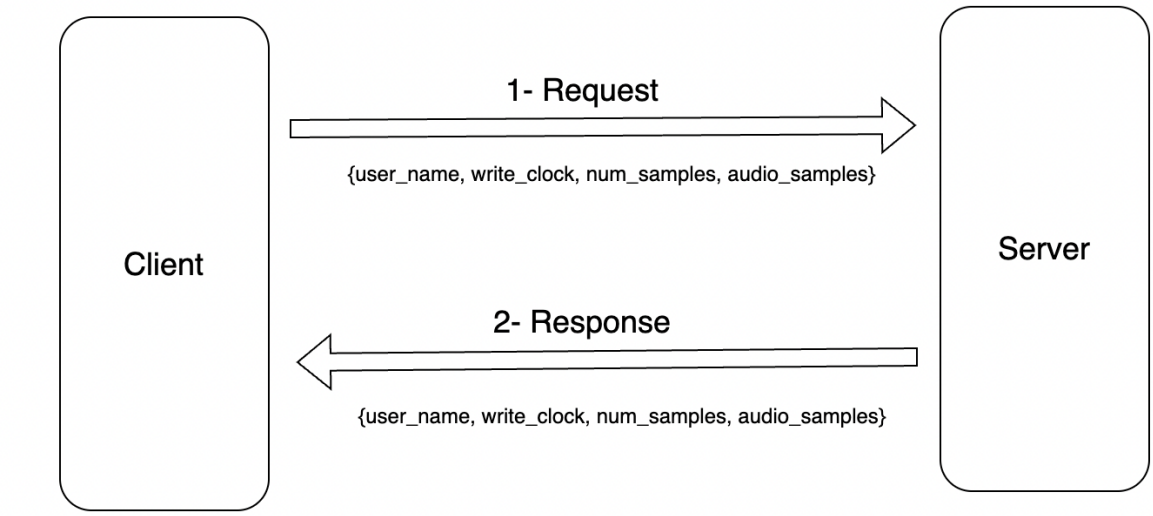

Figure 2.2. Schema Showing Request and Response Metadata

2.3 Server-Side Audio Sample Handling and Processing

Once the client sends a request to the server with the audio samples and the metadata in its body, the datagram listener on the server will catch the client's request, extract its metadata, use the number of samples argument to correctly extract the audio samples, and then decode them (Since they were 8 bit encoded) and then will decode them again using audio-decoders.

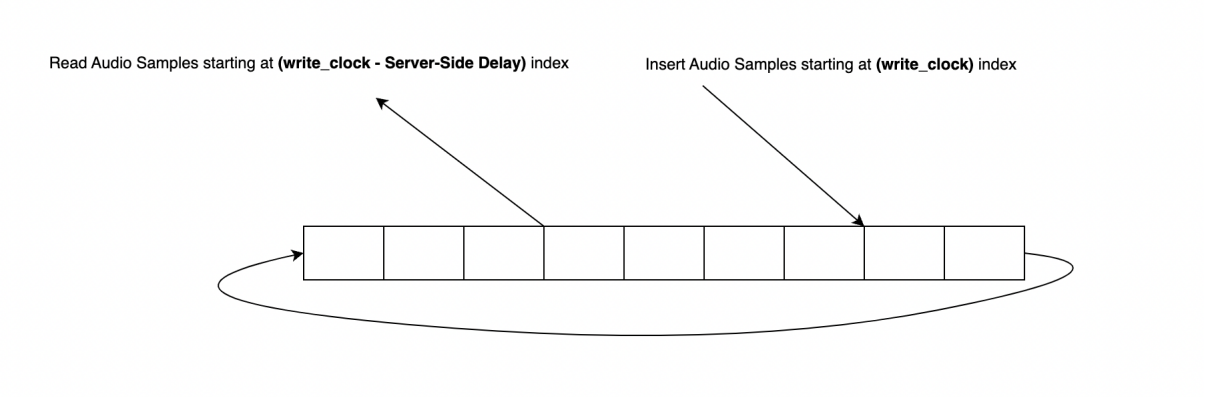

Figure 2.3. Reading and writing audio into the server circular buffer

After that, the server will use the `write_clock` that was extracted from the request's metadata and will write the audio samples inside a circular buffer using the `write_clock` as an index. After that, the server will extract audio samples from the circular buffer using `write_clock` as an index with a certain server-side delay[5] as an offset, encode the data, and then 8-bit encode them alongside some new metadata to send them in the datagram channel. The newly sent metadata will contain important information.

When the server is reading from the circular buffer, it is adding a server-side delay, which means when the client requests data at a certain clock (which is specified by `write_clock`), the server subtracts a constant amount of units and then using the total result to read data from the circular buffer.

```
read_clock = write_clock - server_side_delay,
```

where `read_clock` represents the index we use to read from the circular buffer.

[5] Server-side delay has been discussed with more details in section 2.2.5

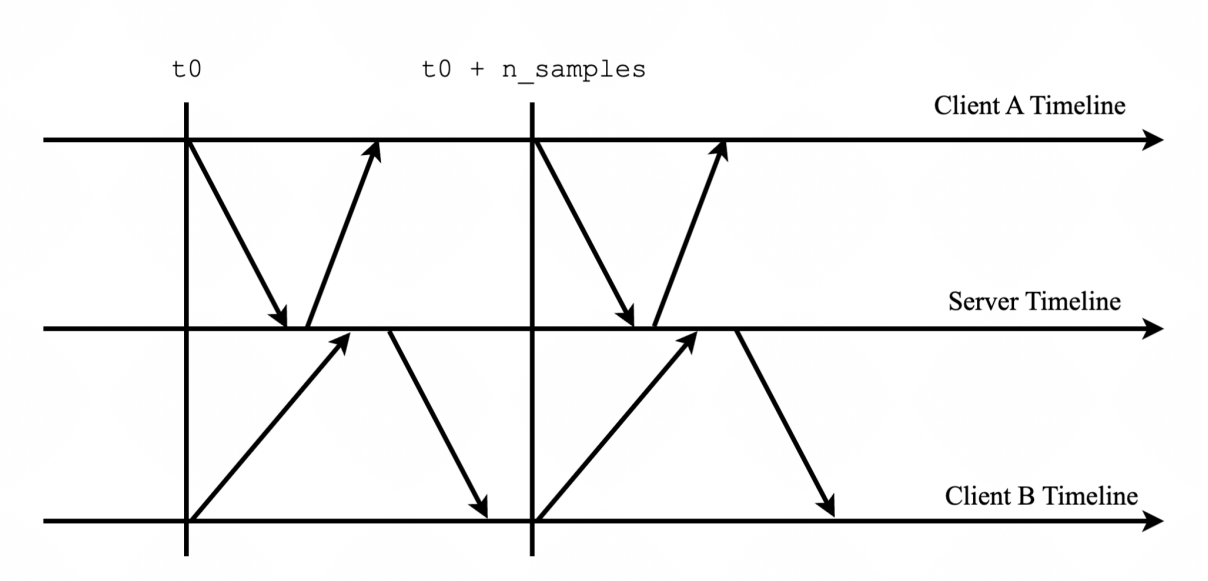

Figure 2.4 Timlelines of Audiorequests of Client A and B according to the Server Timleline (Client A write_clock=Client B write_clock)

We are doing this because when two clients have an audio-conferencing session, those two clients might not be able to completely hear each other. Some audio samples coming from client A might not get heard by client B. The reason behind this is that the clients are using `write_clock` in order to indicate at what index to write audio samples and write audio samples. In case client A sends a request with `write_clock = t0` and client B sends another request with `write_clock = t0` as well, and in case the request from client A makes it to the server before client B, client A won't be reading the audio samples sent from client B because those audio samples did not arrive yet. But client B will be able to read the audio samples coming from client A because they were there at the time the request made it to the server. And since the next `write_clock` for both clients will be the same `write_clock = t0 + n_samples,` then the audio that was missed on the first request won't be included in the second request.

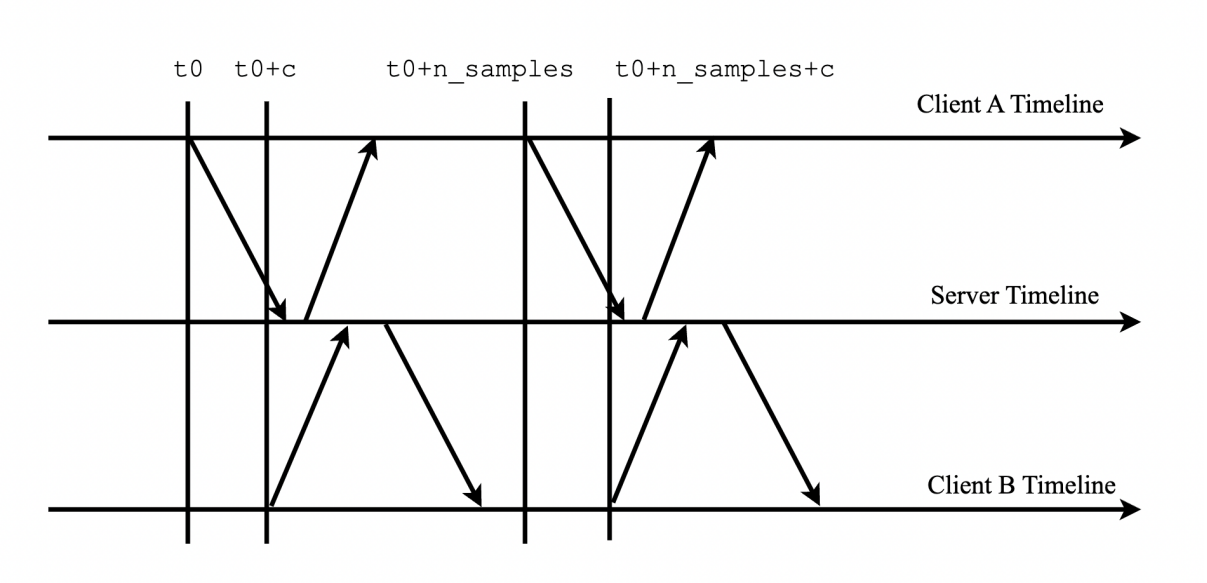

Figure 2.5 Timelines of Audiorequests of client A and B according to the Server Timeline (Client A write_clock and client B write_clock have a constant offset c)

Since having two identical `write_clocks` is an extreme case that is highly unlikely to happen, we're going to talk about another scenario where client A has made a request with `write_clock = t0` and client B has made a request with `write_clock = t0 + c` where `c` is a constant that's smaller than `n_samples`, that represents the difference in clocks between clients. Let's suppose client A's request makes it first to the server, which means client A will pull audio from the server, and then client B's request arrives. On client A's second request, he will have a `write_clock = t0 + n_samples`. But client B's first request had audio samples ranging between `[t0 + c; t0 + c + n_samples]`. And client A is going to pull audio data that's ranging between `[t0 + n_samples; t0 + 2*n_samples]`. So, in that case, client A is going to hear client B's audio data that ranges between `[t0 + n_samples; t0 + c + n_samples]`, which means client A will miss `n_samples - c` audio samples. That is why we need to add a server-side delay that should be equal to `n_samples - c`. Let's suppose we picked a server-side delay equal to `n_samples`, then on client A's second request, he will be pulling data that ranges between

`[t0; t0 + n_samples]`, which means client A will only miss the last `c` samples, but those samples are going to be fetched in Client A's next request.

It's important to mention that we are not using the actual `write_clock` as an index when accessing the circular queue. Since it's a circular queue, it has its own mechanism for handling and translating indices. The function that handles reading audio samples from the circular queue is `wrap_get(queue, start, len_vals)`, and the function that handles writing audio samples in the circular queue is `wrap_assign(queue, start, vals)`. You can find both functions respectively in annex 1.1 and 1.2, and you can also find them in `server.py` lines 423 and 439, respectively. In `server.py`, we are calling `wrap_get(audio_queue, write_clock - server_side_delay, n_samples)` and `wrap_assign(audio_queue, write_clock, audio)`. We should also mention that when the server is writing new audio in the queue, it's actually summing the old audio with the new audio. The code that describes this behavior can be found in annex 1.3

2.4 Client-Side Response Handling Process

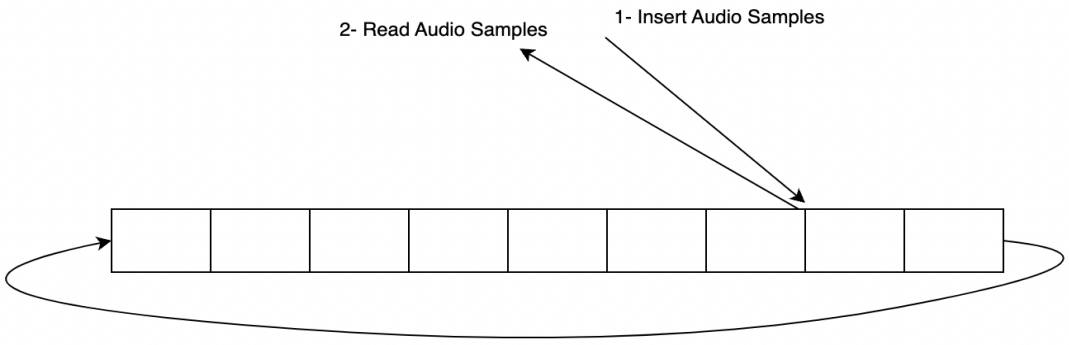

Figure 2.6. Reading and Writing Audio into the Client Circular Buffer

After waiting for a datagram response, the client will now receive a datagram from the server that will then be decoded (Since it was 8-bit encoded), and then the audio sample will be extracted and decoded again using the audio_decoders. This audio sample will be stored in a circular buffer that was created using the `ClockedRingBuffer`[6] class. And then the circular buffers will be polled for audio, and then the sounds will be played on the speaker.

2.5 Clock Calibration on The Client Side

Since a client is using a `write_clock` to indicate the index where he wants to read or write audio from the server's circular buffer, this means that all clients should operate under the same `write_clock`. To do that, all clients are going to calibrate their clocks according to the server's clock. This calibration will happen one time at the start of the conferencing session. The client will fetch the `server_clock` and will use it as a reference.

It's also important for all clients to stay synchronized with the server because we don't want a client to read audio samples far from the future in that case, that client will not be getting any audio from other well-calibrated clients. We don't want a client to write in the future because the other clients will be hearing that client with an unnecessary and faulty delay. We don't want a client to read audio that is far behind in the past because, in that case, that client will hear delayed audio, and we don't want a client to write in the past because the other clients won't be able to hear him. As mentioned before, we want to retain and preserve the feel of real-time, low-latency social interaction during a call session. However, we don't need to worry about the occurrence of such things because the clients will always stay well-calibrated, and the difference in clocks will always remain too little, so it's not worth handling.

[6] You can find the code for that class in audio-worklet.js line 168

2.6 HTTP/1 vs. HTTP/3 Implementation

As mentioned in the introduction section, we have developed two audio conferencing solutions, one using HTTP/3 and the other using HTTP/1.

2.6.1 Main Differences in The Implementation

The HTTP/1 implementation only differs in the communication layer. We are using HTTP/1, which means the client is sending POST requests with all the audio samples and the metadata in its body, then waiting for a server response that will contain the audio samples and some metadata. That means we are using TCP as a transmission layer.

While in the HTTP/3 implementation, the client is sending datagrams that contain all the audio samples and the metadata using the WebTransport datagrams API, and then the client will wait for a datagram to be sent by the server, which will act as a response to the previously sent datagram and that will contain audio samples and some metadata. And that means that the HTTP/3 implementation uses QUIC as a transmission layer.

2.6.2 Performance and Design Limitations

One important thing we noticed when implementing those two solutions is that in order to make those two solutions work under the same environment and configuration, we had to find a threshold that was appealing to both. The threshold parameters include the sample rate, which is the number of samples that a client should send to the server in a second, and the size of a batch in a request in milliseconds, which we can use to calculate the number of samples that were sent in a request using the Sample rate. The main limitations have to do with size and rate.

The HTTP/3 solution was highly performant, but it was limited in the size of the request that was being sent to the server. That is because, in the HTTP/3 solution, we are using the datagrams

API, which means that we are sending one datagram per request, so we need to respect the Datagram's MTU (Maximum Transmission Unit) size.

To get the Datagrams MTU size, we ran the PING (18) command on a local terminal toward our test server. In this PING command, we used different values for the datagram size to find the correct MTU size. The MTU size in our environment was 1472 bytes.

```
>> ping -D -s 1472 13.38.214.141
```

If you try to set the datagram size to 1473, the PING command will fail and will return a `Message too long` error.

So we couldn't use big batch sizes, we had to use a batch size that was equal to 20 milliseconds under a sample rate set to 48000 Hz, which means that a request had (48000 sample/second * 20 ms) / 1000 ms = 960 samples, and that means that we were sending 960 samples every 20 milliseconds.

The HTTP/1 solution was highly flexible when it came to the size of the packets that we are sending, but it was limited in performance. When we tried using a batch size of 20 ms with 48000 Hz, the software was crashing. And that's because HTTP/1 wasn't able to support and handle the high rate of requests that was happening, which was 1 request that had 960 samples every 20 milliseconds. This configuration was really performance demanding and the HTTP/1 protocol which uses TCP as a Transmission layer, a transport protocol that suffers a lot from delays due to head-of-line blocking, wasn't able to deliver.

WebTransport was able to support sending requests with 960 samples every 20 milliseconds, and that's because it is an API that uses HTTP/3, which is built using QUIC as a transmission layer

instead of TCP. QUIC is kind of similar to UDP, so it will not face the head of line blocking problem. So that means that HTTP/3 will face smaller delays than HTTP/1.

So by the end, in order to find a configuration that suits both protocols, we lowered the Server Sample rate to 8000 samples per second, and we used a batch size of 80 milliseconds. Which means every request will have `(8000 sample/second * 80 milliseconds) / 1000 milliseconds = 640 samples.`

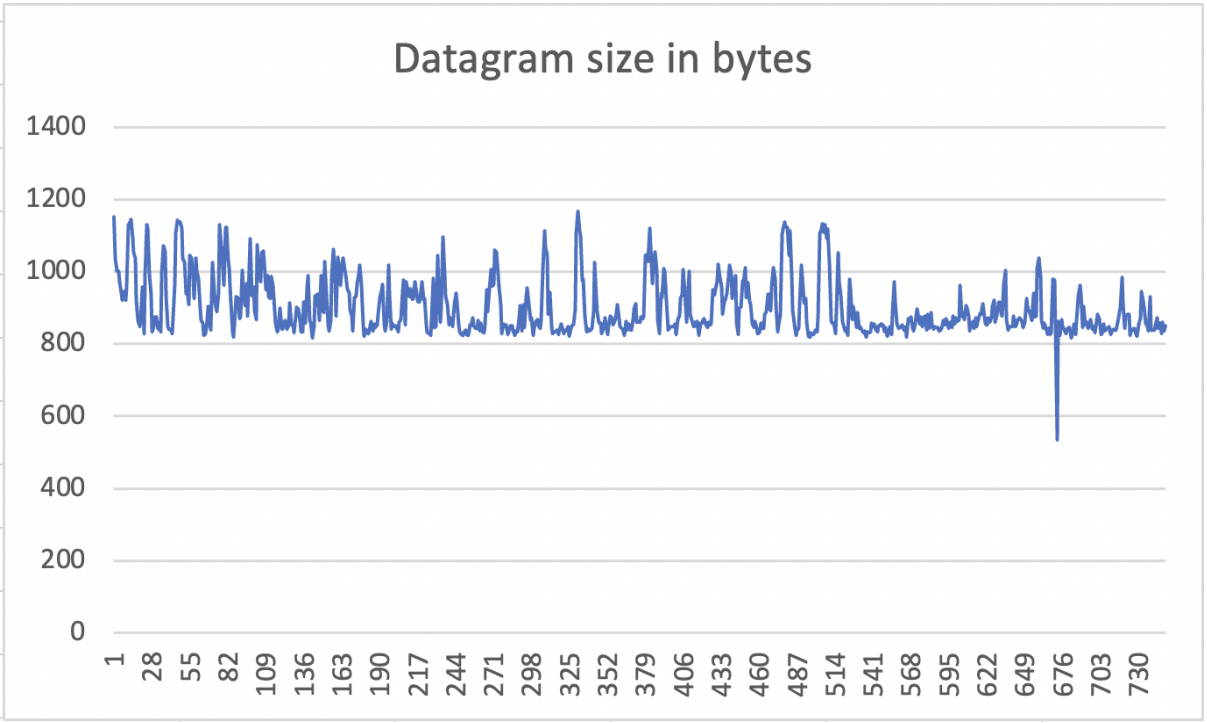

Figure 2.7. Graph plotting the size of the datagrams sent by the client

Datagram size stats		
AVG (bytes)	MIN (bytes)	MAX (bytes)
901.74	534	1167

Table 2.1. Datagram Size Stats

In figure 2.7, we have plotted the size of the datagram that the client sent to the server under a sample rate set to 8000 Hz, and a batch size equal to 80 ms. In the x-axis, you will see the request number and in the y-axis, you will see the size in bytes. We also gathered the stats and showed them in table 2.3. Our Maximum datagram size during our tests was 1167 bytes which are smaller than the MTU size. So by picking those configuration values, we were able to keep our datagrams under the MTU size.

Chapter 3

WebTransport Protocol vs. HTTP/1

WebTransport is a web API that uses the HTTP/3 protocol as a bidirectional transport. It is intended for two-way communications between a web client and an HTTP/3 server. So this thesis tackles the difference between HTTP/1 and HTTP/3. Although we have mentioned WebTransport several times and will mention it in all our tests, the main technical difference between our two applications lies in the protocol differences between HTTP/1 and HTTP/3.

Furthermore, to go deeper into this comparison, HTTP/3 is another binding of HTTP over a particular transport layer. HTTP semantics are consistent across protocol versions, so HTTP/1 and HTTP/3 have the same request methods, Status codes, and message fields. The real difference lies in mapping these HTTP semantics to the underlying transports. HTTP/1 uses TCP as its transport layer, while HTTP/3 uses QUIC as its transport layer. So again, we have narrowed down our technical comparison, and that new scope is the transport layer, TCP VS QUIC.

When sending data over TCP, what happens is that we are giving the data to TCP. This data will get divided into several blocks of data, which will then be transformed into network packets by adding metadata like sequence number and checksum. Those network packets are going to be reliably transmitted to the destination. Since it sends data reliably, it does not tolerate any missing data or reordering of data. When either problem occurs, TCP will order the sender to resend the lost or damaged network packets using an automatic repeat request, ARQ (19).

In case of an error on a specific connection, TCP will consider this event a blocking operation. TCP will block all future transfers until this error is resolved, or this connection will be considered failed. Moreover, since HTTP/1 uses one single connection to send multiple streams of data, one error on one stream will block the other streams. Furthermore, that is the head-of-line blocking issue TCP suffers from, which can add much latency during data transport.

On top of blocking all the streams on that one connection where the error was found, in most cases, considerable additional data may be received before TCP even notices that there is something wrong, so all this data will be blocked or flushed. At the same time, TCP works on fixing the error in that connection.

QUIC was initially developed to be similar to a TCP connection but with reduced latency. One of the most critical TCP problems that QUIC solved was the head-of-line blocking problem we discussed in the previous paragraph. QIUC tackled this issue by relying on understanding the behavior of HTTP traffic and fixing it in the HTTP context. It fixed that problem using UDP, which does not include loss recovery as its basis instead of TCP. QUIC exploits this property and layers a flexible stream multiplexing on top, in which only the content of each stream is ordered. So each QUIC stream is separately flow controlled, and lost data is retransmitted at the level of

QUIC, not UDP. This means that if there is an error in one stream, other streams inside that same connection will keep operating normally. The data coming from the other streams can generally be processed while the stream with errors is fixed. Moreover, this fix solves many of the latency problems that TCP has been facing.

To make sure that this is what's happening on our software, we monitored network activity on the client side using netstat (20), to check how many connections we're opening and to check if we're facing head-of-line blocking in different scenarios. First, we ran the netstat command on our HTTP/1 audio conferencing software.

```
>> netstat -an |  awk '$5 ~ /^13.38.214.141/ && $6 ~ /ESTABLISHED/'
```

Listing 3.1. netstat command for monitoring the network on a certain machine and filters by IP and state of connection

We noticed that we are opening multiple connections, which means that for every new request, we are opening a new connection (which is the expected behavior), and the most important thing that we noticed is that sometimes the Send-Q (20) is high. A high Send-Q means that the data is put on a TCP/IP buffer, but it is not sent or it is sent but not ACKed due to head-of-line blocking.

After that, we ran that same netsat command when testing our HTTP/3 audio conferencing software, and we noticed that the software is using one connection only for all requests, and the Send-Q is always set to 0. This means that there is no congestion or head-of-line blocking.

One other thing that might introduce latency to TCP transmission is including encryption in the communication. Since TCP was developed to act like a « data pipe » or « stream,» it has little understanding of the data it is transporting. So if that data has additional requirements like encryption using TLS, It should be handled by the TCP system. The TCP system should initiate a TCP connection between the client and the receiver. Then packets should be sent back and forth

to negotiate the security protocol that will be used during this transmission. Moreover, since encryption using TLS is a feature in high demand, that added latency needs to be fixed.

QUIC aims to reduce this connection setup overhead significantly. It fixed this problem by making the negotiation of security protocol part of the initial handshake process. Furthermore, when we say security protocol negotiation, we mean the exchange of setup keys and supported protocols. So now, when a client opens a connection, the response packet includes the data needed for future packets to use encryption. This eliminates the need to set up the TCP connection and negotiate the security protocol via additional packets.

Chapter 4

Results

In this chapter we will present several results. We will focus on the analysis of the communication and all the factors which may influence its performance.

We have developed the same software using the two different protocols HTTP/1 and HTTP/3 for the sake of proving that HTTP/3 is more efficient than HTTP/1 and that it's the way forward for low latency real-time applications.

We used WebTransport over HTTP/3 because we wanted to scope this to client/server architectures. And we picked HTTP/1 because we can follow the client/server architecture using this protocol and it's the most commonly used protocol on the internet for the time being.

In the following sections, we'll be talking and discussing how we implemented and performed our tests, then we'll talk about how we were able to find the best value for the sound delay that was added to the `write_clock` on the server side when the server was pulling audio samples from the circular buffer and finally we'll talk about two sets of results. We are going to compare the round-trip performance that both these protocols present, and we are going to compare the performance of both software implementations.

4.1 Implementing Tests

To start us off, we are going to talk about how and where we performed our tests.

In order to properly test the HTTP/1 and HTTP/3 protocols, we had to upload our server to the public internet. That way, we'll be able to include a test case that covers the head of line blocking, and we'll be able to truly understand the difference of performance between the two protocols. So for that reason, we uploaded our server to the cloud using AWS. We created an EC2 instance in AWS, and we uploaded our servers there. We made sure that we can access the IP of the EC2 instance publicly. We also assigned a domain name for the EC2 public IP, that way we can use the domain name instead of the IP address.

We ran the client UI and all the tests on the same MAC M1 machine (Memory: 16GB; Chip: Apple M1 Pro; Model:2021) from the Netherlands, Amsterdam. The Server was uploaded to AWS Paris Servers (Canonical, Ubuntu, 22.04 LTS, amd64 jammy image build on 2022-09-12; 1 vCPU and 1 GIB Memory).

We used Chrome web browser (Version 106.0.5249.91; Official Build; arm64) when running the client front end. We were limited to Google Chrome, because it's the only browser that had a stable implementation for webTransport, although it was hidden behind a feature flag.

In our implementation, we did not separate the circular buffers that stored the audio samples on the server. If we wanted to implement an audio conferencing software that would be used by others to have audio conferences, we would have implemented separate circular buffers for every user that's using the software. That way every user can store his audio samples in his own circular buffer, and when a client requests audio samples from the server, the server will pull audio samples from all the circular buffers except for this certain user's buffer. That way no user will hear his own voice but will be able to hear all other users. We did not do that

implementation in our current software because we wanted for one user to hear himself, that way it would make testing easier.

During every test session, when generating results, we were gathering network stats in order to know how stable the network was when generating results, and how reliable those results truly are. To generate those network stats, we ran the PING command using the IP address of our server that's on the AWS cloud as an argument. The PING command calculates round-trip times and packet loss statistics and displays a brief summary on completion. The statistics that are presented by the PING command are the maximum, minimum, average, and standard deviation for the round-trip duration. And they include the percentage of lost packets.

We are also interested in knowing the bandwidth. So we ran an iPerf (21) test when gathering results. We ran this command on the server side, which is the EC2 instance.

```
>> iperf -s
```

Listing 4.1. iPerf command used on the server to gather bandwidth results

And we ran this command on the client.

```
>> iperf -c 13.38.214.141
```

Listing 4.2. iPerf command used on the client to gather bandwidth results

And what this does is give us the bandwidth between the client and the server.

To perform our tests, we picked a Sample Rate equal to 8000 (which means that we will be sending 8000 samples per second), a batch size of 80 milliseconds (which means that we will be sending 80 milliseconds worth of audio in every request, and that means that every request will have 640 samples, and we will be sending a request every 80 milliseconds) and an OPUS frame

size equal to 10 milliseconds (which means that every batch will contain 8 OPUS frames, and every frame will contain 80 audio samples).

4.2 Network Stats

In this section, we're going to show the network stats that we were able to gather before every test. As mentioned before, we gathered those stats using the PING command and the iPerf command.

PING Stats during the HTTP/1 test				
MIN (milliseconds)	AVG (milliseconds)	MAX (milliseconds)	STDDEV (milliseconds)	PACKET LOSS (%)
23.916	25.851	38.851	2.138	0

Table 4.1. the PING State during the HTTP/1 test

PING Stats during the HTTP/3 test				
MIN (milliseconds)	AVG (milliseconds)	MAX (milliseconds)	STDDEV (milliseconds)	PACKET LOSS (%)
24.391	27.448	86.342	6.489	0

Table 4.2. PING State during the HTTP/3 test

iPerf Stats during HTTP/1 tests and HTTP/3 tests	
Protocol	Bandwidth
HTTP/1	7.83 MBits/sec
HTTP/3	7.60 Mbits/sec

Table 4.3. Bandwidth results gathered from the iPerf command during both HTTP/1 and HTTP/3 tests

4.3 Server-Side Delay Best Value

As mentioned in the previous sections, we need to add some delay on the server side in order to allow a user to hear most of the audio that's coming from others. This was discussed in section 2.3.

Finding the best value for that delay can be tricky if we wanted to find it manually by trial and error. So we followed an analytical approach to find the best value. What we did was gather some data that will help us in finding the percentage of the audio that clients are receiving from other clients in the function of the server-side delay that we're introducing when clients are trying to read audio samples from the circular buffer that's sitting inside the server.

We performed the tests for both the HTTP/1 implementation and the HTTP/3 implementation, and in both cases, we had two clients connected to the audio-conferencing software and having a conversation together. Those two clients are `Client A` and `Client B`. So, for both implementations, we found the percentage of the audio that Client A heard from Client B and the percentage of the audio that Client B heard from Client B for different server-side delays.

To gather the data, we added some code in the server that will push to an array some metadata whenever a client tries to read from the circular buffer that's inside the server.

```
testResults.append({"username": user.name, "client_write_clock":
client_write_clock})
```

Listing 4.3. Code snippet: the metadata used to gather data for calculating the percentage of heard audio

We're registering the username of the user because we want to get results separately for Client A and Client B, and we're registering the client_write_clock because that's how we'll know if one client heard the other client.

This is the code that we used in order to produce the stats:

```
const getStats = (factor) => {
 const data = [];


 for (let i = 0; i < jsonData.length; i++) {
   if (jsonData[i].username === target) {
     let total = 0;
     //check if audio samples were heard
     for (let j = i + 1; j < jsonData.length; j++) {
       if (jsonData[j].username !== target) {
         total += intersects(
           jsonData[i].client_write_clock, //write audio
           jsonData[i].client_write_clock + nSamples,
           jsonData[j].client_write_clock - factor * READ_DELAY, // read audio
           jsonData[j].client_write_clock + nSamples - factor * READ_DELAY
         );
       }
     }
     data.push(total);
   }
 }
 return data;
};
```

Listing 4.4. Code snippet: Calculating the percentage of heard audio

```
const intersects = (targetLower, targetUpper, lower, upper) => {
 if (targetUpper < lower) return 0;
 if (includes(targetUpper, lower, upper)) return targetUpper - lower;
 if (includes(targetLower, lower, upper)) return upper - targetLower;
 if (targetLower > upper) return 0;
};
```

Listing 4.5. Code snippet: the intersect function used by the getStats(factor) function

4.3.1 Server-Side Delay best value for the HTTP/1 Implementation

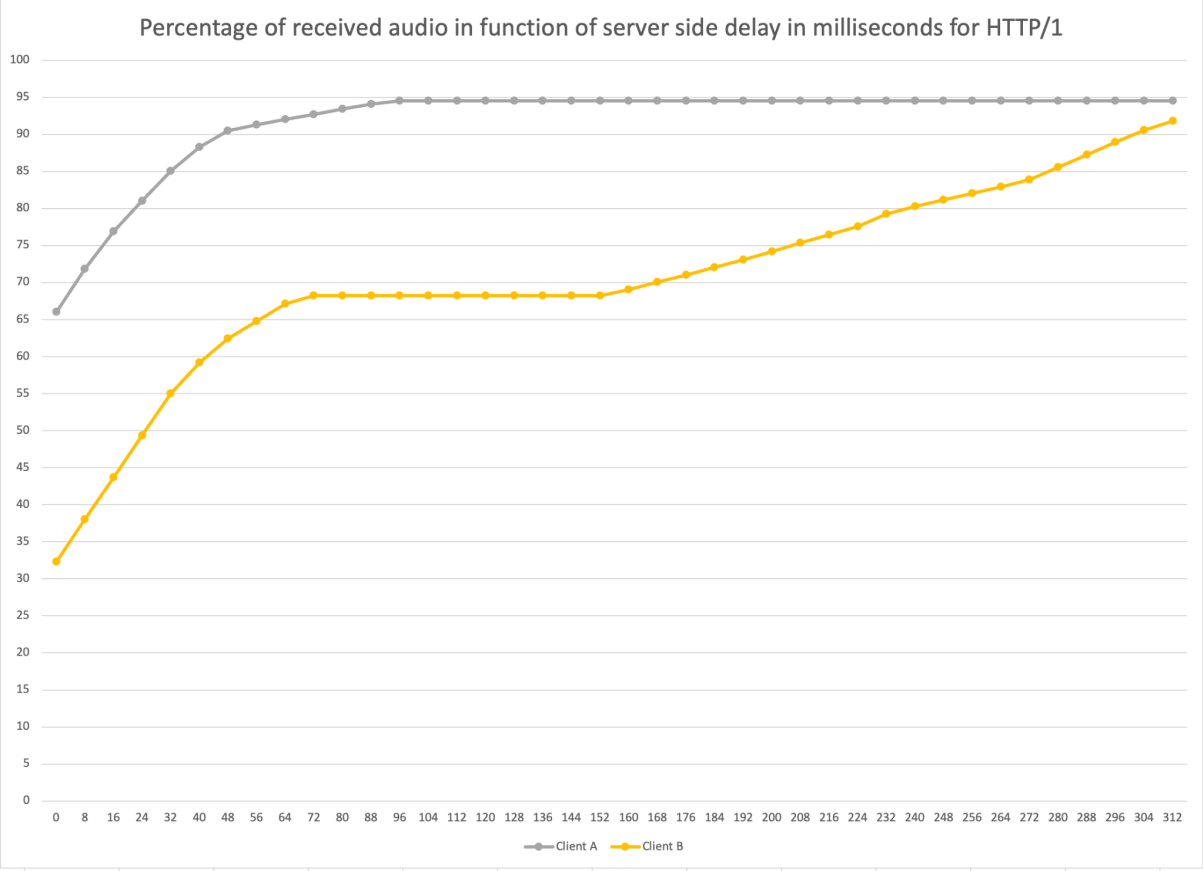

Figure 4.1. Received audio (%) in function of the server-side delay (ms) for Client A and Client B (HTTP/1)

In this graph, we have plotted the results of the HTTP/1 implementation. The x-axis represents the value of the server-side delay in milliseconds. The y-axis represents the percentage of the audio that was heard.
The plot in yellow represents the percentage of the audio that client B was able to hear from client A in the function of the server-side delay and the plot in gray represents the percentage of the audio that client A was able to hear from client B.

According to the results, when we had 0 milliseconds server-side delay, the percentage of the audio that was heard by Client A was equal to 66%, while Client B was able to get 32% of the audio. When we start increasing the server-side delay from 0 milliseconds to 72 milliseconds, both plots seem to be increasing at the same rate. At this point, Client A is able to hear 92% of the total audio, and Client B was able to get 68% of the audio. After this point, Client A will increase slowly to a maximum of 94%. However, client A stays idle for server-side delays between 72 milliseconds and 152 milliseconds around 68% and it starts increasing pretty well after that 152 milliseconds server-side delay till it gets to a maximum of 91% at 312 milliseconds.

We are mainly interested in giving the possibility to the clients to hear more than 90% of audio so that audio quality is considered to be good quality. We also want to have the smallest server-side delay value since that will add more latency to the session.

Having said that, the best server-side delay that fits our KPIs is at 304 milliseconds which has Client A getting 94% of audio and Client B getting 90% of audio.

4.3.2 Server-Side Delay best value for the HTTP/3 implementation

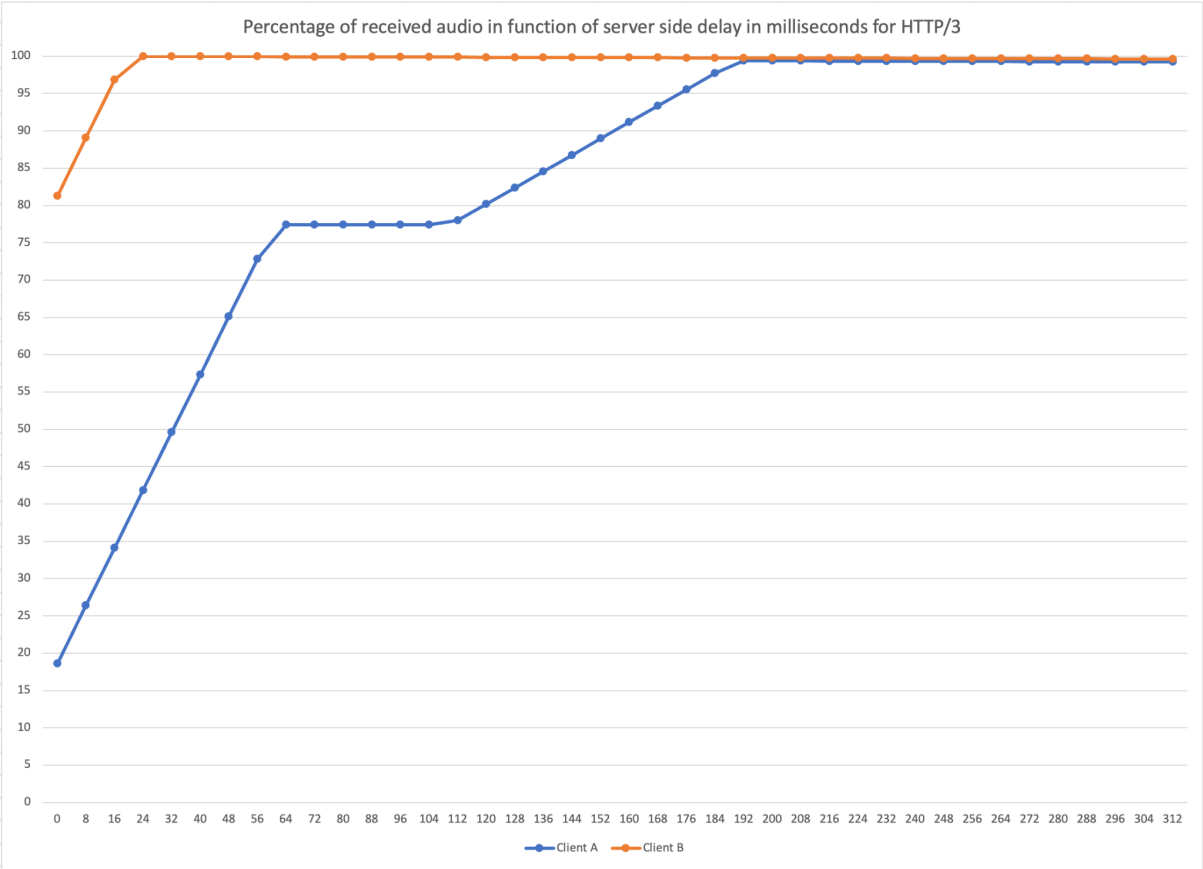

Figure 4.2. Received audio (%) in function of the server-side delay (ms) for Client A and Client B (HTTP/3)

In this graph, we have plotted the results for the HTTP/3 implementation. The x-axis represents the value of the server-side delay in milliseconds. The y-axis represents the percentage of the audio that was heard.

The plot in orange represents the percentage of the audio that client B was able to hear from client A in the function of the server-side delay and the plot in blue represents the percentage of the audio that client A was able to hear from client B.

According to the results, when we had 0 milliseconds server-side delay, the percentage of the audio that was heard by Client A was equal to 18%, while Client B was able to

get 81% of the audio. When we start increasing the server-side delay from 0 milliseconds to 24 milliseconds, both plots seem to be increasing at the same rate. At this point, Client A is able to hear 41% of the total audio, and Client B was able to get almost 100% of the audio. After this point, Client B will remain idle at a delay that's almost 100%. However, client A keeps on increasing server-side delays ranging from 24 milliseconds to 64 milliseconds which brings it to 77%. After that, it stays idle for server-side delays between 64 milliseconds and 112 milliseconds around 77% and it starts increasing pretty well after those 112 milliseconds till it almost gets to 100% at 192 milliseconds.

As mentioned, we are mainly interested in giving the possibility for the clients to hear more than 90% of audio while keeping the smallest server-side delay value since that will add more latency to the session.

So according to the plots, the best server-side delay that fits our KPIs is at 160 milliseconds which has Client A getting 91% of the audio and Client B getting almost 100% of the audio.

4.3.3 HTTP/3 vs HTTP/1 Server-Side Delay

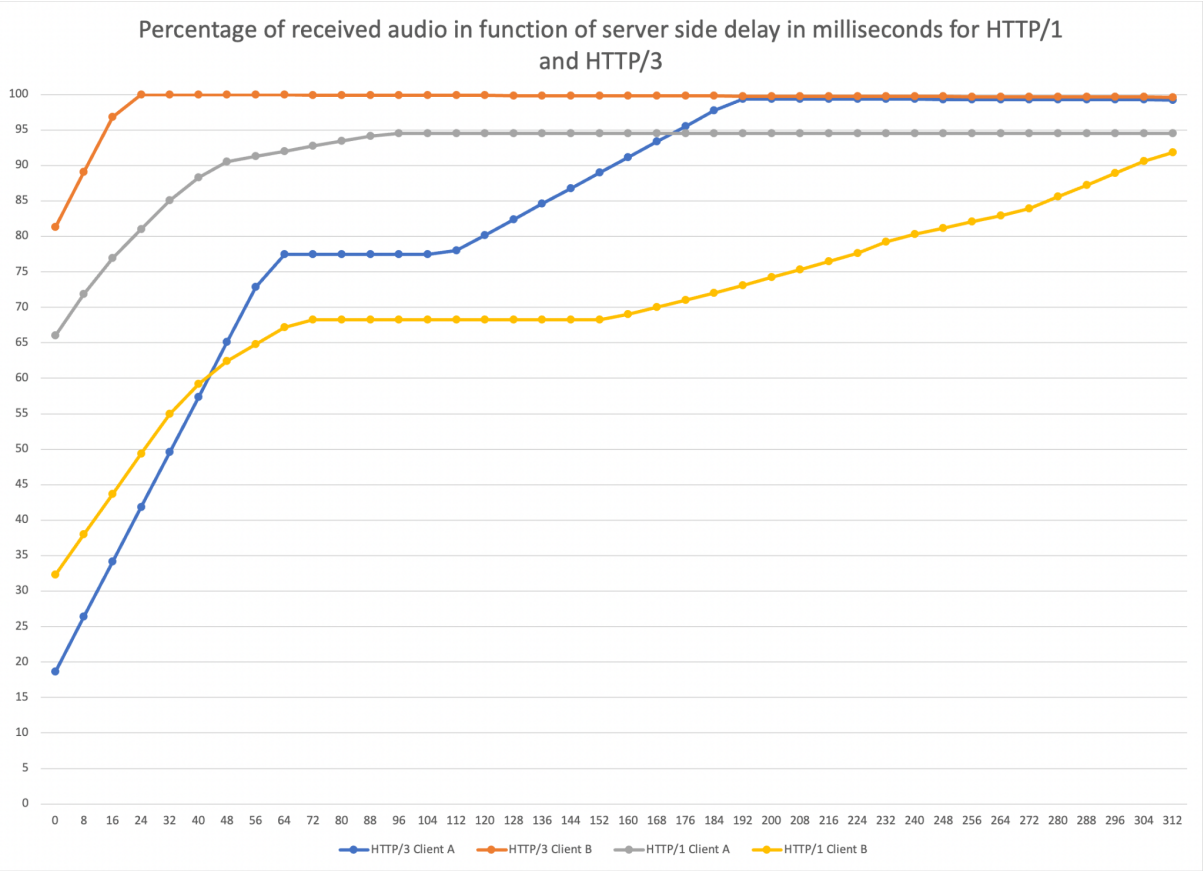

Figure 4.3. Received audio in function of the server-side delay in milliseconds for Client A and Client B (HTTP/1 vs. HTTP/3)

In this graph we plotted all the previously mentioned server-side delay plots all together. The plots in yellow and gray are the ones for HTTP/1 and the plots that are in blue and orange represent the HTTP/3 implementation.

4.4 Round Trip Latency

Let's start discussing the time it takes for a client to send a request to the server and to get the response from the server, which we can also call round-trip latency.

Those tests are important because our application is involved in sending requests and receiving responses, so we want to know what kind of delay that includes, and we want to get a sense of the performance differences between the two protocols. It's also important we do these tests because, in our other tests, there is lots of this request/response action going on. So it makes sense to check out where the delay difference is coming from, that should help us in making a proper analysis.

What we did here, is that in both implementations (HTTP/1 and HTTP/3 implementations), we added some additional metadata to the request that is being sent from the client to the server and we also included those same metadata values in the response that was sent from the server to the client for this certain request. Those metadata include important information for this test like UID, which is a unique identifier for requests that we can use in order to keep track of requests. Metadata also includes `sent_time` and `received_time`. And those are respectively the time when this request was sent and the time when the response was received. In the case of HTTP/1, we set the value for `sent_time` when the POST request was called from the client, and we set the `received_time` when the POST response arrives at the client. In the case of HTTP/3, we set the `sent_time` when the datagram gets sent from the client and we set the `received_time` when the response datagram was received. How do we know that this response datagram was sent as a response to our certain request datagram? It's done using the Unique Identifier that we were sending in the metadata.

We are using the `performance.now()` function in order to get the current time.

To calculate the round-trip latency, we are subtracting `sent_time` from `received_time`. The result is in milliseconds.

We are storing all that data inside a hashmap that we are going to use to generate the results, the stats, and the plots.

4.4.1 HTTP/1 vs HTTP/3 round-trip latency

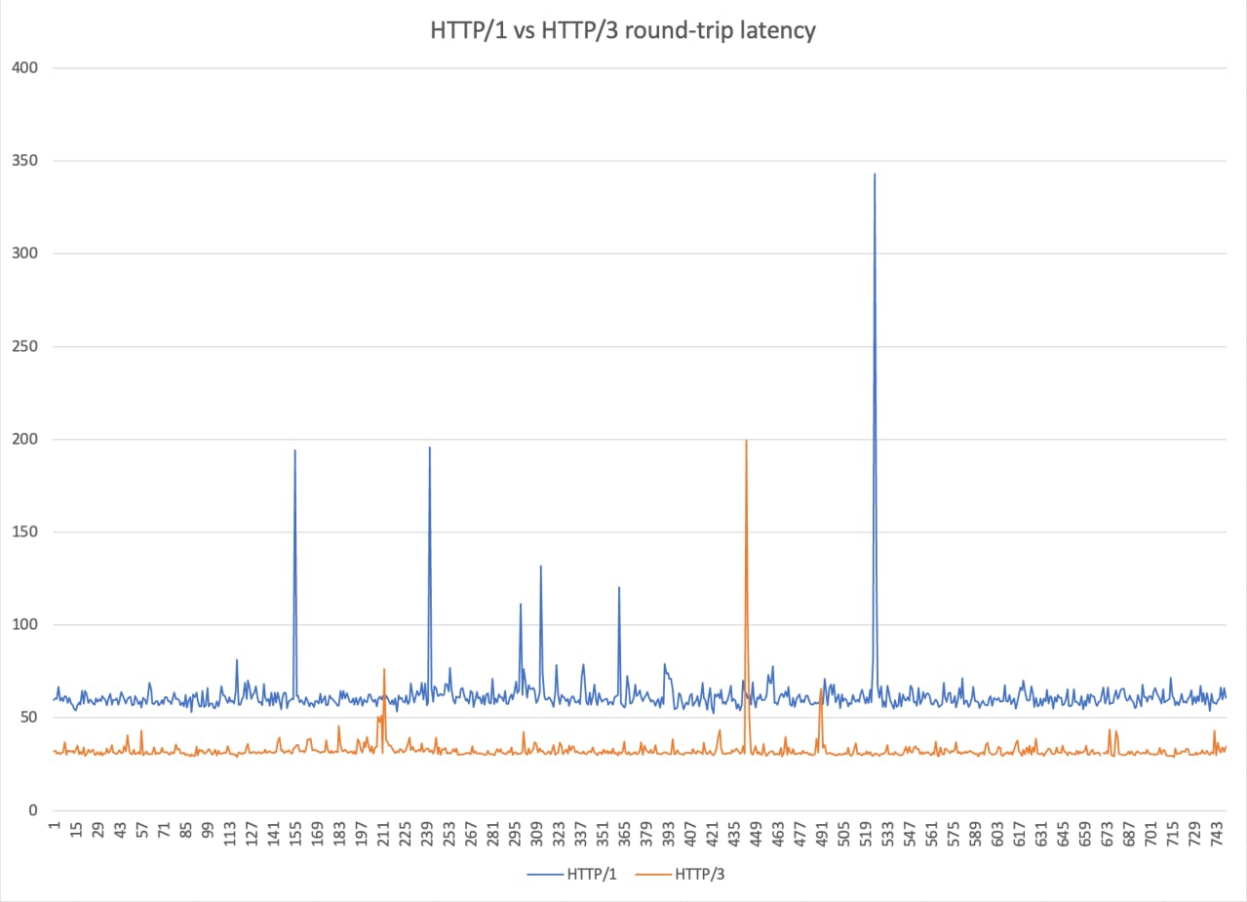

Figure 4.4. Graph plotting the round-trip latency results of HTTP1 vs. HTTP/3

In this figure, we have plotted our round-trip latency results. The plot in blue represents the round-trip latency results that were generated by the solution that implements the HTTP/1 protocol, and the plot in orange represents the results that were generated by the solution that implements the HTTP/3 protocol. The y-axis represents the latency in milliseconds, which in this context is the duration of time that it took for a client to send a request to the server and then receive a response from the server. It's obvious that latency includes the time it took for packets

to go from the client to the server, the overhead that's being introduced by the server which should be negligible because the server is only going to pull audio samples from its circular buffer, and finally, the time it took for packets to go from the server to the client. The x-axis represents which packet we're plotting the results for.

We plotted the results for both protocols in the same graph in order to do a clear analysis of the results and to clearly see the differences between the two.

According to the graph, it's clear that the latency introduced by HTTP/1 is higher than the latency that's introduced by HTTP/3. The difference in latency between the two protocols is around 30 milliseconds. This shows that HTTP/3 is faster than HTTP/1.

We can also see that the latency for the HTTP/3 protocol is more stable than the HTTP/1 protocol. In HTTP/1, the latency keeps on going up and down more frequently than the latency in HTTP/3, and that is because HTTP/1 suffers from the head of line blocking which means that some delay is going to be added to requests and responses when the network faces some hiccups.

For the HTTP/3 request that's around requests number 435 and 449, we can see that that's where HTTP/3 hit its maximal latency of 200 milliseconds. This latency is due to a hiccup in the network that made that datagram take time to complete its round-trip journey.

For the HTTP/1 request that's around requests number 519 and 533, we can see that that's where HTTP/1 hit its maximal latency of 344 milliseconds. This latency is due to a hiccup in the network that caused the packets to either get lost or arrive at their destination in an unordered manner during their round-trip journey which means that head-of-line blocking delays were introduced.

Those tests were not generated at the same time. They were generated separately. We did get the PING stats in order to make sure that those results are reliable.

4.4.2 Round-Trip Latency Distribution

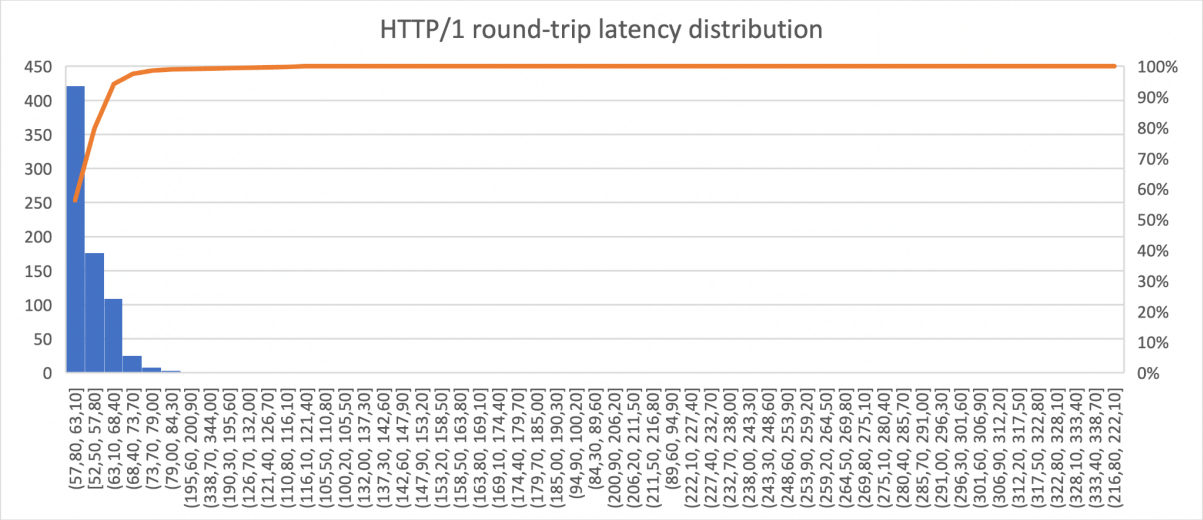

Figure 4.5. Graph plotting the HTTP/1 round-trip delay distribution with a cumulative plot

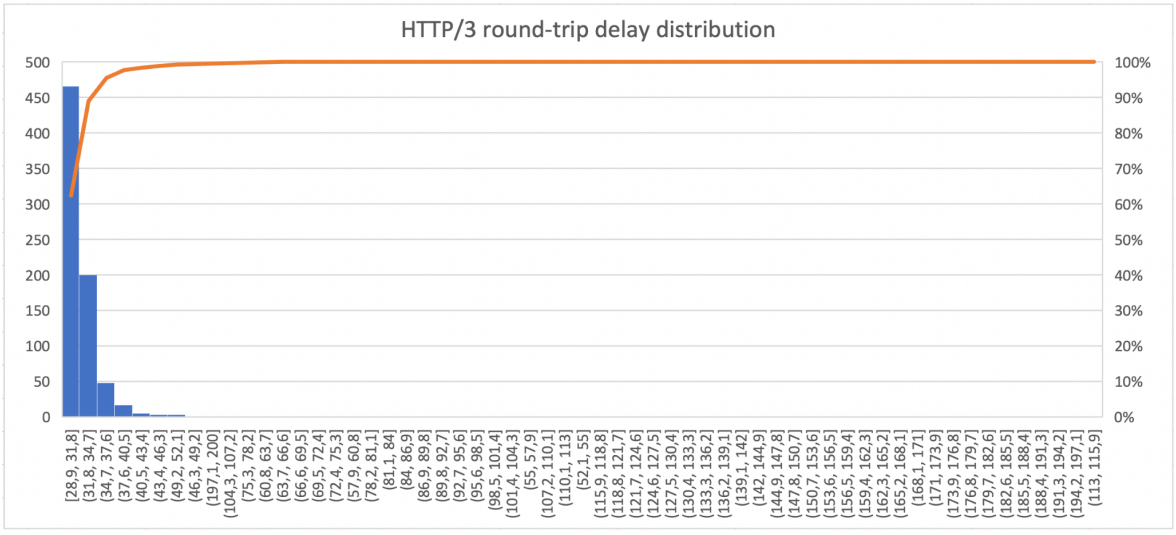

Figure 4.6. Graph plotting the HTTP/3 round-trip delay distribution with a cumulative plot

In those two figures, we showed the distribution of the round-trip delay for HTTP/1 and HTTP/3 in milliseconds. In the horizontal axis, you will find the range of delays. In the vertical axis, you will find the amount of requests/responses that had a round-trip delay that was included in a certain round-trip delay range.

According to these two histograms, we can see that most of HTTP/1 requests and responses had a round-trip delay in the range of (57.8, 63.1] milliseconds. While most of HTTP/3 datagrams had a round-trip delay in the range of [28.9, 31.8] milliseconds. So most of HTTP/3 datagrams were faster than most HTTP/1 requests and responses by (28.9, 31.3] milliseconds. That is almost 50% less.

We can also see that the minimum delay for HTTP/3 was 28.9 milliseconds, while the minimum delay for HTTP/1 was 52.5 milliseconds. The maximum delay for HTTP/3 was 200 milliseconds, while the maximum delay for HTTP/1 was 344 milliseconds. We will show more insights for the results in the following table.

HTTP/3 round-trip latency stats		
AVG (milliseconds)	MIN (milliseconds)	MAX (milliseconds)
32.51	28.9	199.4

Table 4.4. HTTP/3 round-trip latency stats

HTTP/1 round-trip latency stats		
AVG (milliseconds)	MIN (milliseconds)	MAX (milliseconds)
61.77	52.5	343.1

Table 4.5. HTTP/1 round-trip latency stats

4.5 Sound Latency

In this third section, we want to discuss and compare the performance that was brought by the application that was implemented using HTTP/1 as a communication protocol and by the one that was implemented using WebTransport over HTTP/3 as a communication protocol.

And in this section, the performance is judged by the amount of latency that was added to the audio. This audio latency is actually the sum of

1. The latency is introduced by the process of polling audio samples from our audio input buffer
2. The latency introduced when sending the audio samples from the client to the server (which is packet latency in the case of HTTP/1 and datagram latency in the case of WebTransport)
3. The latency introduced by the server overhead (which is the time spent processing, writing, and reading the audio samples for and from clients)
4. The latency introduced by the server-side delay was discussed in section 4.1
5. The latency introduced when sending the audio samples from the server to the client

For the first point, that latency is equal to the batch size value that we configured before testing, which is 80 milliseconds.

The latency that was introduced by the server overhead should be negligent in comparison to the communication latency.

Most of the latency should be coming from the server-side delay that was introduced in order to ensure clients maintain good audio quality

We did not try to calculate the mouth-to-ear latency because it includes the mic and speaker latency, and that should be considered out of scope from this thesis since we are more interested in the latency that's introduced by the different communication protocols and how they affect the stability of our application, which can be inspected from within the application without the need for mouth to ear latency calculations.

To calculate and get those test results, we simply added the server-side delay results that were discussed in section 4.1 to the round-trip results that were discussed previously in section 4.2.

4.5.1 HTTP/1 vs. HTTP/3 sound latency

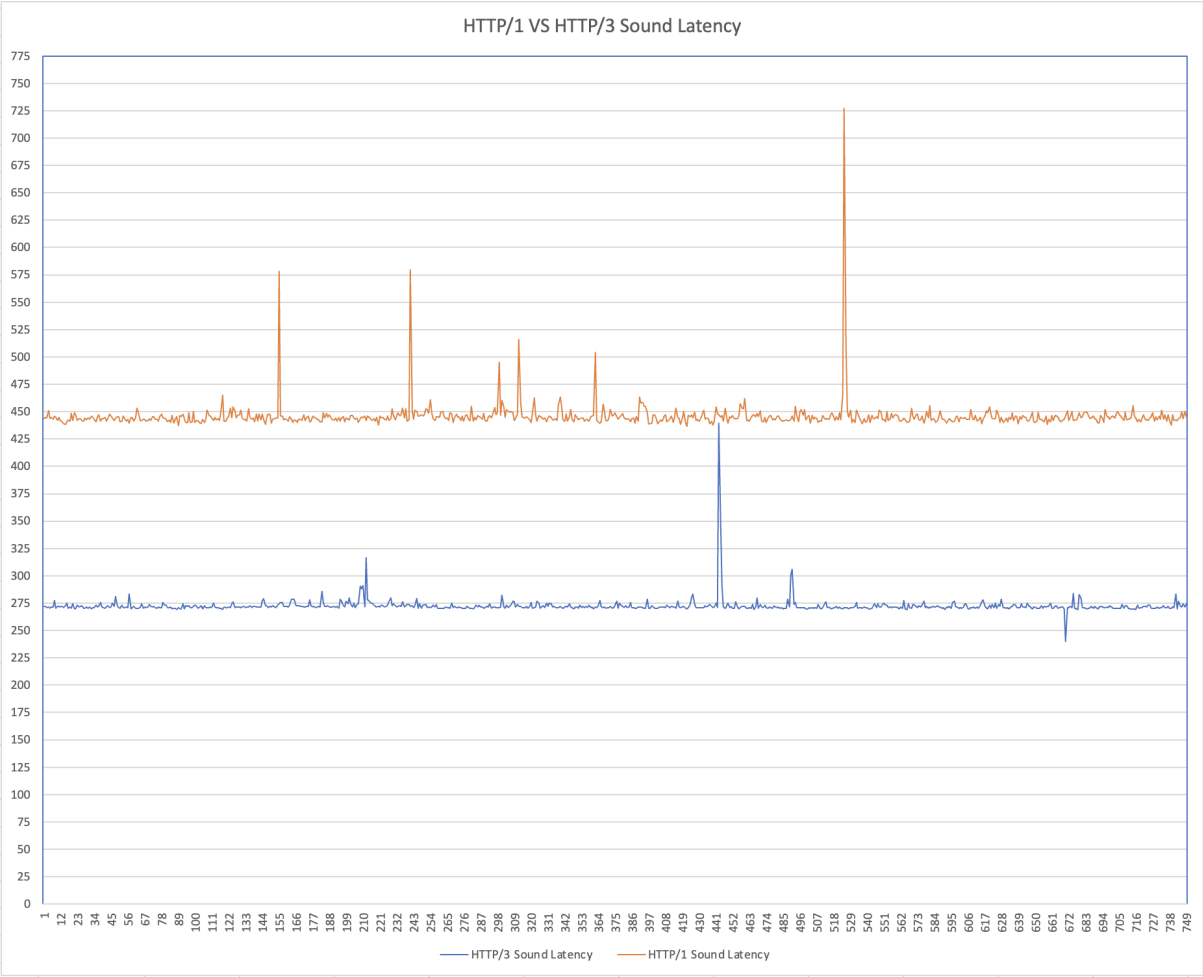

Figure 4.7. Graph plotting the sound latency results of HTTP1 vs. HTTP/3

HTTP/3 sound latency stats		
AVG (milliseconds)	MIN (milliseconds)	MAX (milliseconds)
272.47	240	439.9

Table 4.6. HTTP/3 sound latency stats

HTTP/1 sound latency stats		
AVG (milliseconds)	MIN (milliseconds)	MAX (milliseconds)
445.77	436.5	727.1

Table 4.7. HTTP/1 sound latency stats

In this figure, we have plotted our sound latency results. The plot in blue represents the sound latency results that were generated by the solution that implements the HTTP/3 protocol, and the plot in orange represents the results that were generated by the solution that implements the HTTP/1 protocol. The y-axis represents the latency in milliseconds, which in this context is the duration of time that it took for a client to send certain audio to the server and then receive that same audio from the server. The x-axis represents which packet we're plotting the results for.

It's pretty obvious that the HTTP/3 implementation is much more performant than the HTTP/1 implementation. The HTTP/3 implementation mostly sticks under the 300 milliseconds latency threshold which is considered to be the latency limit for an application to be considered real-time and low-latency. However, HTTP/1's latency is way past the 300 milliseconds limit since its average latency got to 445 milliseconds.

Chapter 5

Conclusion

This master thesis aimed to illustrate how HTTP/3 can significantly enhance the audio conferencing world. While we only worked on audio conferencing software, those enhancements can also be applied to video conferencing software.

By building two audio conferencing software, both having the same architecture but different communication protocols, one using HTTP/1 and one using HTTP/3, we were able to perform tests that showed that HTTP/3 is indeed way more performant than the HTTP/1 protocol, which is a protocol that is already widely used since it is currently the standard communication protocol. More specifically, the round-trip latency tests discussed in section 4.4 proved that HTTP/3 is much better than HTTP/1 when it comes to sending requests and receiving responses. We also showed why we were expecting that to happen in chapter 3 when we had a technical discussion around the architecture of HTTP/3 and HTTP/1, where we mentioned that HTTP/3 is a major improvement to HTTP/1 because it is built on top of QUIC instead of TCP, which means that it will not introduce head-of-line blocking delays which are the main source of latency in a real-time low latency context. We also proved the existence of this phenomenon by doing netstat observations on the client side, which showed that TCP connections are struggling to send packets due to network congestion and packet flow control issues while HTTP/3 was not facing any problems at all.

By doing our sound latency tests in section 4.5, we were able to not only prove that an HTTP/3 solution is a better choice than an HTTP/1 solution but also proved that we can build real-time

low-latency client-server audio conferencing software operating over HTTP/3 because we were able to maintain our sound delays under the low-latency threshold of 300 milliseconds which is considered the normal limit for audio conferencing software. And this observation is by itself a big revelation to the audio conferencing world because a client-server architecture is a way forward since it's much more stable and scalable than its counterpart peer-to-peer.

In conclusion, based on our research and results, it seems like HTTP/3 is the way forward. It is going to be a big step forward for the internet and a big step forward for how digital human interactions are communicated. Audio and video conferencing companies should consider adopting HTTP/3 in their technical infrastructure since it will give them a major technical advantage in a competitive market.

All the code developed during this master's thesis is available here (22).

Annex

Code Snippets

6.1.1 Function used to read audio samples from the circular buffer in the server

```
def wrap_get(queue, start, len_vals) -> Any:
    start_in_queue = start % len(queue)

    if start_in_queue + len_vals <= len(queue):
        return np.copy(queue[start_in_queue:(start_in_queue+len_vals)])
    else:
        second_section_size = (start_in_queue + len_vals) % len(queue)
        first_section_size = len_vals - second_section_size
        assert second_section_size > 0
        assert first_section_size > 0

        return np.concatenate([
            queue[start_in_queue:(start_in_queue+first_section_size)],
            queue[0:second_section_size]
            ])
```

6.1.2 Function used to write audio samples in the circular buffer in the server

```
def wrap_assign(queue, start, vals) -> None:
```

```
    assert len(vals) <= len(queue)
    start_in_queue = start % len(queue)

    if start_in_queue + len(vals) <= len(queue):
        queue[start_in_queue:(start_in_queue+len(vals))] = vals
    else:
        second_section_size = (start_in_queue + len(vals) )% len(queue)
        first_section_size = len(vals) - second_section_size
        assert second_section_size > 0
        assert first_section_size > 0

        queue[start_in_queue:(start_in_queue+first_section_size)] =
vals[:first_section_size]
        queue[0:second_section_size] = vals[first_section_size:]
```

6.1.3 Function that sums the old audio with the new audio and writes them in a circular queue in the server

```
def update_audio(pos, n_samples, in_data, is_monitored):
    old_audio = wrap_get(audio_queue, pos, n_samples)
    new_audio = old_audio + in_data
    wrap_assign(audio_queue, pos, new_audio)
```

6.1.4 Code that was used to create the python HTTP/3 server

```
BIND_ADDRESS = '0.0.0.0'
```

```
BIND_PORT = 4433


H3_DATAGRAM_05 = 0xffd277
ENABLE_CONNECT_PROTOCOL = 0x08


class H3ConnectionWithDatagram(H3Connection):
   def __init__(self, *args, **kwargs) -> None:
       super().__init__(*args, **kwargs)


   # Overrides H3Connection._validate_settings() to enable HTTP Datagram
   def _validate_settings(self, settings: Dict[int, int]) -> None:
       settings[Setting.H3_DATAGRAM] = 1
       return super()._validate_settings(settings)


   # Overrides H3Connection._get_local_settings() to enable HTTP Datagram
and
   # extended CONNECT methods.
   def _get_local_settings(self) -> Dict[int, int]:
       settings = super()._get_local_settings()
       settings[H3_DATAGRAM_05] = 1
       settings[ENABLE_CONNECT_PROTOCOL] = 1
       return settings


class CounterHandler:
```

```
   def __init__(self, session_id, http: H3ConnectionWithDatagram) -> None:
       self._session_id = session_id
       self._http = http
       self._counters = defaultdict(int)

   def HTTP3_post_response(self, data):
       self._http.send_datagram(self._session_id, data)

   def h3_event_received(self, event: H3Event) -> None:
       if isinstance(event, DatagramReceived):
           do_HTTP3_POST(event.data, self.HTTP3_post_response)

       if isinstance(event, WebTransportStreamDataReceived):
           self._counters[event.stream_id] += len(event.data)
           if event.stream_ended:
               if stream_is_unidirectional(event.stream_id):
                   response_id = self._http.create_webtransport_stream(
                       self._session_id, is_unidirectional=True)
               else:
                   response_id = event.stream_id
               payload =
str(self._counters[event.stream_id]).encode('ascii')
               self._http._quic.send_stream_data(
```

```
                    response_id, payload, end_stream=True)
                self.stream_closed(event.stream_id)

    def stream_closed(self, stream_id: int) -> None:
        try:
            del self._counters[stream_id]
        except KeyError:
            pass


# WebTransportProtocol handles the beginning of a WebTransport connection:
it
# responses to an extended CONNECT method request, and routes the transport
# events to a relevant handler (in this example, CounterHandler).
class WebTransportProtocol(QuicConnectionProtocol):

    def __init__(self, *args, **kwargs) -> None:
        super().__init__(*args, **kwargs)
        self._http: Optional[H3ConnectionWithDatagram] = None
        self._handler: Optional[CounterHandler] = None

    def quic_event_received(self, event: QuicEvent) -> None:
        if isinstance(event, ProtocolNegotiated):
            self._http = H3ConnectionWithDatagram(
```

```
                self._quic, enable_webtransport=True)
        elif isinstance(event, StreamReset) and self._handler is not None:
            # Streams in QUIC can be closed in two ways: normal (FIN) and
            # abnormal (resets).  FIN is handled by the handler; the code
            # below handles the resets.
            self._handler.stream_closed(event.stream_id)

        if self._http is not None:
            for h3_event in self._http.handle_event(event):
                self._h3_event_received(h3_event)

    def _h3_event_received(self, event: H3Event) -> None:
        if isinstance(event, HeadersReceived):
            headers = {}
            for header, value in event.headers:
                headers[header] = value
            if (headers.get(b":method") == b"CONNECT" and
                    headers.get(b":protocol") == b"webtransport"):
                self._handshake_webtransport(event.stream_id, headers)
            else:
                self._send_response(event.stream_id, 400, end_stream=True)

        if self._handler:
            self._handler.h3_event_received(event)
```

```
    def _handshake_webtransport(self,
                                stream_id: int,
                                request_headers: Dict[bytes, bytes]) ->
None:
        authority = request_headers.get(b":authority")
        path = request_headers.get(b":path")
        if authority is None or path is None:
            # `:authority` and `:path` must be provided.
            self._send_response(stream_id, 400, end_stream=True)
            return
        if path == b"/counter":
            assert(self._handler is None)
            self._handler = CounterHandler(stream_id, self._http)
            self._send_response(stream_id, 200)
        else:
            self._send_response(stream_id, 404, end_stream=True)

    def _send_response(self,
                       stream_id: int,
                       status_code: int,
                       end_stream=False) -> None:
        headers = [(b":status", str(status_code).encode())]
        if status_code == 200:
```

```
            headers.append((b"sec-webtransport-http3-draft", b"draft02"))
        self._http.send_headers(
            stream_id=stream_id, headers=headers, end_stream=end_stream)


def serveHttp3():
    parser = argparse.ArgumentParser()
    parser.add_argument('certificate')
    parser.add_argument('key')
    args = parser.parse_args()

    configuration = QuicConfiguration(
        alpn_protocols=H3_ALPN,
        is_client=False,
        max_datagram_frame_size=65536,
    )
    configuration.load_cert_chain(args.certificate, args.key)

    loop = asyncio.get_event_loop()
    loop.run_until_complete(
        serve(
            BIND_ADDRESS,
            BIND_PORT,
            configuration=configuration,
            create_protocol=WebTransportProtocol,
```

```
        ))
    try:
        loop.run_forever()
    except KeyboardInterrupt:
        pass
```

6.1.5 Function that will fetch the server_clock and the server_sample_rate

```
export async function query_server_clock(target_url) {
 var request_time_ms = Date.now();
 const fetch_init = { method: "get", cache: "no-store" };
 const fetch_result = await fetch(target_url, fetch_init)
   // Retry immediately on first failure; wait one second after subsequent ones
   .catch(() => {
     console.warn("First fetch failed in query_server_clock, retrying");
     return fetch_with_retry(target_url, fetch_init);
   });

 if (!fetch_result.ok) {
   throw {
     message:
       "Server request gave an error. " +
       "Talk to whoever is running things, or " +
```

```
      "refresh and try again.",
    unpreventable: true,
  };
}

var server_latency_ms = (Date.now() - request_time_ms) / 2.0;
var metadata = JSON.parse(fetch_result.headers.get("X-Audio-Metadata"));
var server_sample_rate = parseInt(metadata["server_sample_rate"], 10);
var server_clock = Math.round(
  metadata["server_clock"] + (server_latency_ms * server_sample_rate) / 1000.0
);
console.info(
  "Server clock is estimated to be:",
  server_clock,
  " (",
  metadata["server_clock"],
  "+",
  (server_latency_ms * server_sample_rate) / 1000.0
);
return { server_clock, server_sample_rate };
}
```

Bibliography

1. Berners-Lee, T. Hypertext Transfer Protocol -- HTTP/1.0 - W3. https://www.w3.org/Protocols/HTTP/1.0/spec.html. [Online]

2. HTTP/2. (2022, August 14). In Wikipedia. https://en.wikipedia.org/wiki/HTTP/2. [Online]

3. HTTP/3. (2022, October 9). In Wikipedia. https://en.wikipedia.org/wiki/HTTP/3. [Online]

4. WebSocket. (2022, August 6). In Wikipedia. https://en.wikipedia.org/wiki/WebSocket. [Online]

5. Head-of-line blocking. (2022, September 13). In Wikipedia. https://en.wikipedia.org/wiki/Head-of-line_blocking. [Online]

6. Transmission Control Protocol. (2022, October 4). In Wikipedia. https://en.wikipedia.org/wiki/Transmission_Control_Protocol. [Online]

7. WebRTC. (2022, August 17). In Wikipedia. https://en.wikipedia.org/wiki/WebRTC. [Online]

8. WebTransport - W3. https://www.w3.org/TR/webtransport/. [Online]

9. Internet Engineering Task Force. (2022, September 7). In Wikipedia. https://en.wikipedia.org/wiki/Internet_Engineering_Task_Force. [Online]

10. User Datagram Protocol. (2022, September 27). In Wikipedia. https://en.wikipedia.org/wiki/User_Datagram_Protocol. [Online]

11. QUIC. (2022, October 5). In Wikipedia. https://en.wikipedia.org/wiki/QUIC. [Online]

12. Introduction to WebRTC protocols - Web APIs | MDN - Mozilla. https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Architecture. [Online]

13. Maximum transmission unit. (2022, September 2). In Wikipedia. https://en.wikipedia.org/wiki/Maximum_transmission_unit. [Online]

14. Transport Layer Security. (2022, September 17). In Wikipedia. https://en.wikipedia.org/wiki/Transport_Layer_Security. [Online]

15. Opus (audio format). (2022, September 30). In Wikipedia. https://en.wikipedia.org/wiki/Opus_(audio_format). [Online]

16. OpenSSL. (2022, October 4). In Wikipedia. https://en.wikipedia.org/wiki/OpenSSL. [Online]

17. RSA (cryptosystem). (2022, September 17). In Wikipedia. https://en.wikipedia.org/wiki/RSA_(cryptosystem). [Online]

18. ping (networking utility). (2022, September 5). In Wikipedia. https://en.wikipedia.org/wiki/Ping_(networking_utility). [Online]

19. Automatic repeat request. (2022, September 6). In Wikipedia. https://en.wikipedia.org/wiki/Automatic_repeat_request. [Online]

20. netstat. (2022, July 18). In Wikipedia. https://en.wikipedia.org/wiki/Netstat. [Online]

21. Iperf. (2022, October 2). In Wikipedia. https://en.wikipedia.org/wiki/Iperf. [Online]

22. [Online] https://github.com/tonyYSaliba/Audio-Conferencing-Over-HTTP3-Benchmark.