MŰEGYETEM 1782

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Telecommunications and Media Informatics

Scalable, Multi-Regional Real-Time Communication Services over QUIC

Students' Scientific Association Report

Author:

Richárd Tamás Váradi

Advisor:

Dr. Gábor Rétvári

2023

Contents

1 Introduction **2**

2 Background **6**

2.1 Real-Time Communication 6

2.2 WebRTC 7

2.2.1 Initiating a session 8

2.2.2 Media Transport 9

2.2.3 APIs 10

2.2.4 Media servers 11

2.2.5 Scaling 11

2.3 QUIC 12

2.3.1 RTP over QUIC 14

2.3.2 Media over QUIC Transport 14

2.3.3 Rush 14

2.3.4 QuicR 15

2.3.5 The QuicR delivery tree 15

3 Architecture design **17**

3.1 Media plane 18

3.2 Control plane 19

3.3 Client 20

3.4 Workflows 20

3.4.1 createRoom 21

3.4.2 joinRoom 21

3.4.3 deleteRoom 22

4 Implementation **23**

4.1 Media server 23

4.2 Application server 24

4.3 Client 25

5 Evaluation **27**

5.1 Deployment model 27

5.2 Session scaling 28

5.3 Geographic Scaling 29

6 Conclusion **31**

Bibliography **32**

Chapter 1

Introduction

The Web Real-Time Communication (WebRTC) technology has revolutionized the way we interact in today's digital world [1]. WebRTC is a critical enabler for applications, such as videoconferencing, live streaming, and online gaming. These *real-time* applications fundamentally differ from traditional "bulk" Web applications, like email, file transfer or browsing, due to the strict requirement on latency and jitter (latency variation) imposed on end-to-end data transfer. Indeed, a videoconferencing application becomes unusable if the time it takes for the network to transfer audio/video frames between participants exceeds about 300 ms and delay variation surpasses 40-60 ms [2]. Certain real-time applications, like cloud-based video-gaming or online auctions, pose even more demanding delay/jitter requirements. WebRTC is a suite of open standards that enables standard Web browsers to run such real-time communication applications, driving popular audio/video conferencing services (MS Teams, Google Meet, Facebook Messenger, Discord [3]), digital assistants (Amazon Alexa [4]), cloud-gaming (Microsoft Xbox Cloud Gaming, [5]), live broadcasting/ streaming (Twitch [6]), and online security systems (Ring.com [3]). The wide variety of use-cases created a large market valued at 4 billion USD in 2022 [7].

Figure 1.1 shows WebRTC usage before and during the COVID-19 pandemic as the percentage of total page loads over the years. The most salient feature is that the popularity of real-time communications increased sharply during worldwide lockdowns, letting millions of people to switch to remote work overnight. Usage fell after the lockdowns, but still stabilized at three times the popularity before the pandemic.

WebRTC is the de-facto way to build real-time Web services today: it is supported by all major browsers, offers great performance and quality, and makes it relatively easy to build new real-time services on top. On the other hand, WebRTC uses lots of legacy protocols (like Voice over IP (VoIP) [8]), which makes it difficult to use it in today's Internet. For instance, WebRTC sessions may break when the network path contains a *middlebox* (network address translator (NAT), firewall, or load-balancer) between the sender and the receiver. With the growing popularity of NAT-traversal protocols (ICE, STUN, TURN, [9]), however, this causes fewer problems these days. Nevertheless, there remain two major WebRTC limitations that currently go unsolved:

1. the complexity of scaling out WebRTC sessions to potentially thousands or hundreds of thousands of users (*session scaling*),

2. the difficulty in deploying WebRTC servers close to users in a geographically distributed setup (*geographic scaling*).

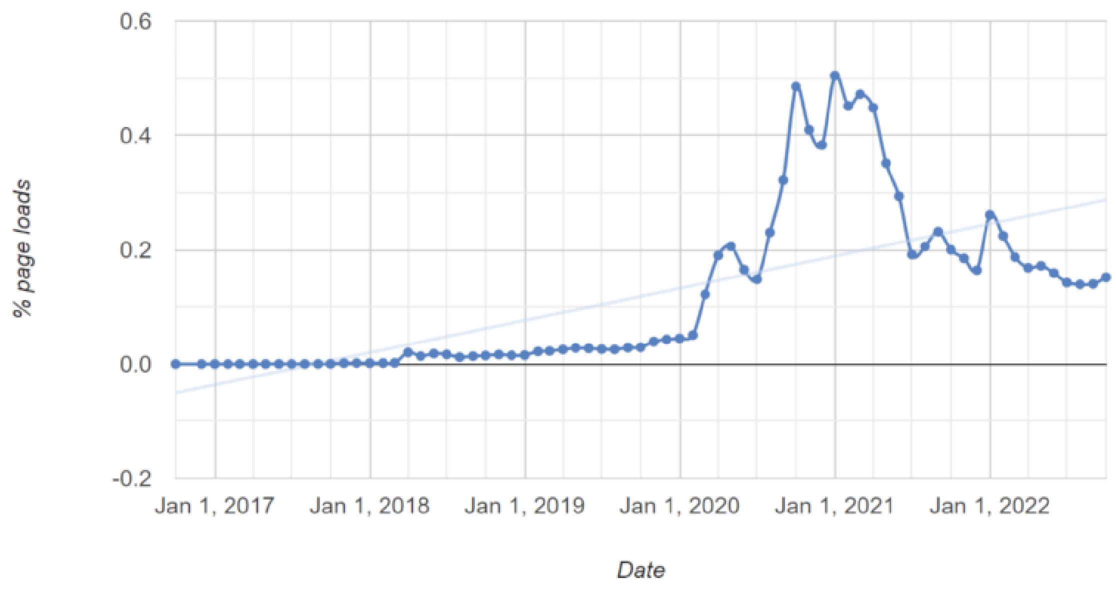

Figure 1.1: The growth in use of WebRTC during the COVID-19 pandemic [1].

To understand the session scaling problem, consider a worldwide live-streaming service where a publisher posts an audio/video stream online and an arbitrary number of viewers can watch the stream *in real-time*. An example of such a service is a live feed from a scientific conference where remote viewers can post questions online, or the live broadcast of a large-scale e-sports event. Suppose that initially there are only a handful of viewers, so a single WebRTC media server is enough to broadcast the stream to all viewers. However, if the number of online viewers grows beyond the capacity of a single server, then the server becomes overloaded. Unfortunately, the default WebRTC communication model does not allow established connections to be moved dynamically across servers, since in WebRTC, every communication session is bound permanently to a concrete endpoint (e.g., a particular WebRTC media server). Even distributing a session over multiple servers is already beyond the capabilities of most open-source WebRTC servers [10]. Our evaluations (see later) show that the popular Jitsi [11] WebRTC media server also suffers from this limitation: in our experiments, a single Jitsi media server with limited resources stops working after 15 users try to join.

The left panel of Figure 1.2 shows this setup, where all users are bound to a single server. An obvious solution would be to scale out the session to two or more servers; e.g., we could organize the servers into a media distribution tree topology. The right side of Figure 1.2 shows that the same user base is now distributed to two servers. This, as we will show later with evaluations, allows a session to be scaled beyond the capacity limit of a single server, solving the session scaling problem.

The other WebRTC limitation we address in this work is geographic scaling. Consider a WebRTC based video-conferencing application. The upper part of Figure 1.3 shows a setup where the first participant who joins a video-conferencing session is located in the USA. Typically, video-conferencing applications decide to host the media server serving a video-conferencing session closest to the first participant, so the media server will also be hosted in the USA. Then, if a large number of participants join the session from Europe, they will be connected through the media server located in the USA, which results large latency due to the physical distance. If the US participant leaves the session, the rest of

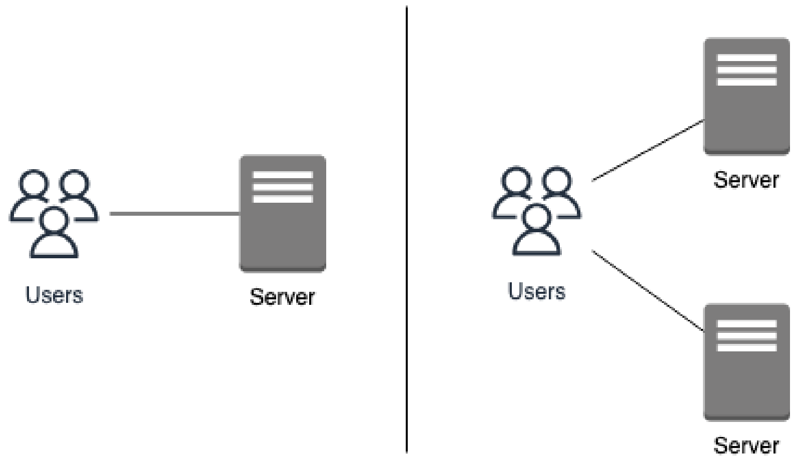

Figure 1.2: Session scaling limitation with WebRTC on the left and scaling with our service on the right.

the participants will continue to use the same US server since, recall, there is no way to move WebRTC connections around.

In this context, geographic scaling means to exchange real-time traffic as close as possible to users. The lower part of the Figure 1.3 shows an ideal setup, the audio/video streams of EU participants would be exchanged over a server located in the EU, yielding minimal latency, and only traffic that must cross continents would experience large delay (but this is unavoidable). Earlier work showed that Jitsi [11] also suffers from the geographic scaling limitation, in that the built-in geographic scaling solution does not provide significant improvement in terms of latency [12].

The major contribution of this work is the *design and implementation of a new real-time communications architecture that supports sharing sessions across multiple servers (session scaling) and across multiple geographic locations (geographic scaling).* We use our design to build a conferencing application and, using real audio-video traces and servers deployed to the US and EU, we show that the new real-time communications architecture reliably delivers smaller delay than state-of-the-art WebRTC conferencing applications [11, 12].

The rest of this work is structured as follows. Chapter 2 gives a brief description of the background needed to understand WebRTC and its protocols, and dives deeper into session scaling and geographical scaling issues. Chapter 3 explains the design of our new real-time communications architecture and describes how it provides both session scaling and geographical scaling. Chapter 4 presents our implementation and the main challenges we solved during the development. Finally, Chapter 5 presents experimental evaluations comparing our new architecture to a commercially available WebRTC service and finally Chapter 6 concludes the work and describes future work.

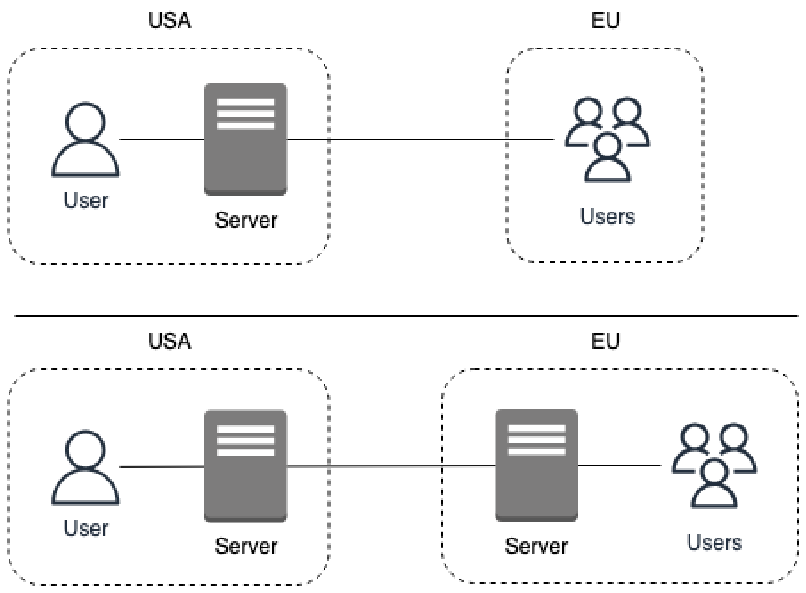

Figure 1.3: Geographic scaling limitation with WebRTC on the top and geographic scaling with our solution.

Chapter 2

Background

2.1 Real-Time Communication

Real-Time Communication (RTC) refers to almost instantaneous, low-latency communication. Examples of RTC include:

- **VoIP**: Enables real-time voice communication over the Internet.
- **Video conferencing**: Enables real-time video communication over the Internet.
- **Online gaming**: Updating game information or remote rendering of the game itself in real-time.
- **Remote rendering**: Rendering complex scenes or images on a remote server and streaming the video back to the user's device.
- **Telehealth**: Experts give medical consultation over a video conference using custom Internet of Things (IoT) devices.

RTC has revolutionized businesses and the individuals' live by allowing remote collaboration, worldwide teamwork, and instant connectivity. However, RTC has its own challenges. It requires a reliable and high-speed Internet connectivity to ensure a stable connection. But even with a good Internet connection, certain complicating factors may deteriorate user experience.

- **Network congestion**: Network congestion occurs when a network component becomes overloaded by the excessive amount of data. Congestion usually results packet loss, leading to audio/video glitches.
- **Latency**: The time it takes for packets to traverse the network. RTC use-cases require latency to be as low as possible.
- **Technical issues**: A poorly written client or server, a hardware issue, or even a misconfigured middlebox can easily interfere with the stringent delay requirements of RTC applications.

In order to minimize latency, RTC requires various protocols to facilitate seamless and efficient data transmission. In the following sections, we discuss the typical protocols used for efficient low-latency data transmission.

2.2 WebRTC

WebRTC is an open-source technology that enables real-time communication within Web browsers. WebRTC was standardized in 2010 [8] and from then, only minor changes made to the standard. WebRTC consists of multiple protocols and Application Programming Interfaces (APIs) that allow developers to build custom voice, video, peer-to-peer file sharing and real-time data exchange applications. In this chapter, we review the most important services and components that comprise WebRTC, including session initiation, media transport, APIs, media servers and scaling.

The core components of WebRTC include three main technologies:

- Clients
- Media and application servers
- Protocols and APIs for session establishment and media transport

A WebRTC client is a software that uses the WebRTC API to enable RTC between browsers and devices. WebRTC clients can be implemented in any programming language. WebRTC media servers are relaying between WebRTC clients to process audio, video, and data streams. They are necessary when the session requires advanced media handling, such as group calls, recording, transcoding, or broadcasting. There are multiple popular open-source media servers available, including Kurento, LiveKit, Jitsi, and Ant Media [7]. WebRTC application servers are responsible for hosting the application logic, serving the user interface of a WebRTC service using web technologies, and implementing the signaling functions needed for clients to establish sessions. Standard WebRTC distributions usually include the application server.

In summary, the WebRTC client is basically your browser. The user interface is accessible through the application server, while the media server will relay the media if needed. This architecture is shown in Figure 2.1. Here, the clients are web browsers, initiating a WebRTC session through the application server, and they communicate through a WebRTC media server.

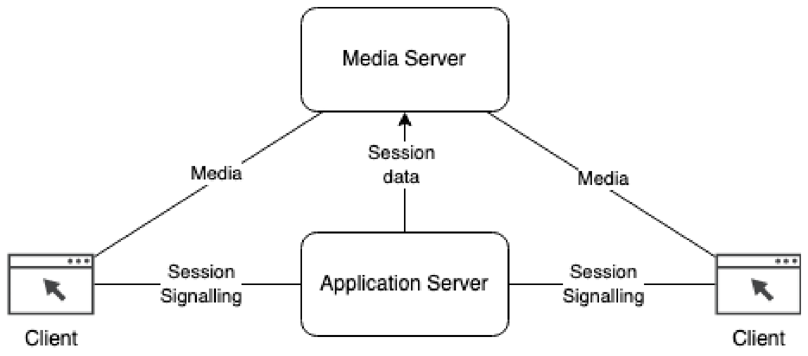

Figure 2.1: A WebRTC architecture where clients initiating a session through and application server and transmit media over a media server.

2.2.1 Initiating a session

Initiating a WebRTC session is a standard process if the clients, media servers, and application servers have direct IP-level connectivity. However, if the network path contains one or mode middleboxes that modify IP addresses and/or transport protocol ports (e.g., NAT), then clients and servers will not be able to identify their own public IP addresses. For such cases, the Interactive Connectivity Establishment (ICE [9]) protocol is a standard NAT-traversal utility included in WebRTC that helps clients behind a NAT to connect to a server. Figure 2.2 shows the updated reference scenario of Figure 2.1 extended with NAT-traversal.

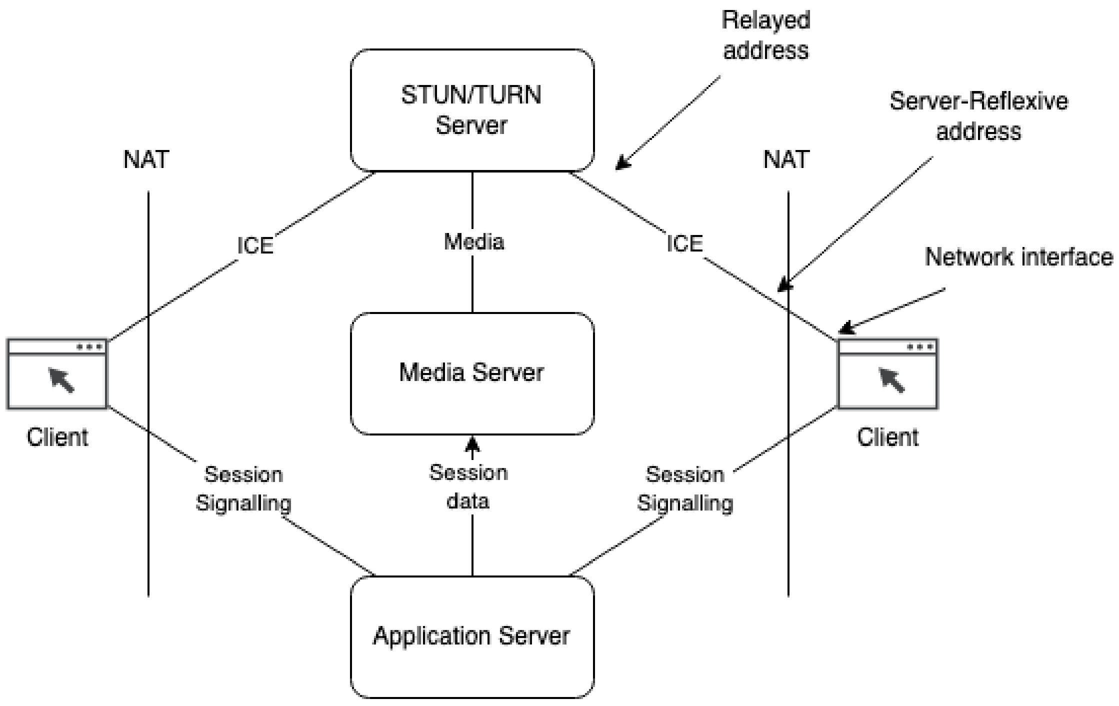

Figure 2.2: WebRTC architecture, where clients are behind NATs. Clients initiating a session through the application server and get their addresses from the Session Traversal Utilities for NAT/Traversal Using Relays around NAT (STUN/TURN) servers using the ICE protocol. After finding out their address they will communicate through the TURN server.

ICE allows the agents to discover enough information about their topology to potentially find one or more paths by which they can establish a data session. This is performed by ICE agents engaging in a candidate exchange process through a signaling server, whose purpose is to find a pair of publicly usable transport addresses that allow the agents to establish a data session between themselves.

- Transport address can be a directly attached network interface (host address).
- Transport address can be on the public side of NAT (server-reflexive address).
- Transport address can be allocated from a TURN server (relayed address). A TURN server can relay media packets between clients if their public IP addresses are not visible.

ICE will try to match each candidate to create candidate pairs. Note that not every candidate will be used. For example, if one agent is behind NAT, it will create a host candidate with its directly attached network interface. This candidate will not be usable because it will be not visible behind the NAT.

It is important to notice that *the transport address of WebRTC clients and servers is used permanently during the entire lifetime of the communication session*, because clients identify media servers through this address and also the other way around. If there's a change in address, the WebRTC session will break. This limitation is critical for session sharing and geographical scaling, since it prevents WebRTC sessions from being moved across servers (to avoid overload) or distribute WebRTC sessions across multiple geographic locations (to minimize latency over a heterogeneous user base). Custom commercial solutions exist to solve this problem, but these are typically proprietary, complicated and cause massive overhead [12].

2.2.2 Media Transport

Media processing and sending involves several steps to capture, encode, decode, and render audio and video in real-time:

1. **Media Capture**: Capture media from input sources such as webcam, microphone, or screen sharing.
2. **Media Encoding**: Sending media requires encoding to a suitable format. With these codecs, the raw media size is reduced and the quality remains acceptable. WebRTC supports multiple codecs, like Opus, VP8, or H.264 [13].
3. **Media Packaging**: Segment the encode media streams into Real-time Transport Protocol (RTP) packets.
4. **Media Transport**: Sends RTP packets through User Datagram Protocol (UDP) or Transmission Control Protocol (TCP). Peer-to-peer communication will use the raw RTP packets. In the case of a TURN server, the RTP packets will travel encapsulated in TURN packets.
5. **Media Reception**: On the receiving end, the media packets are received and reassembled.
6. **Media Decoding**: The media packets are decoded into raw media.
7. **Media Rendering**: The received audio is played through speakers or headphones and video frames are presented on the screen or within a video element on the Web page.

Figure 2.3 shows the previously mentioned process. There are three important components on the figure besides the whole process:

1. WebRTC has access to the microphone, webcam, and screen share thanks to the MediaStream API.
2. With WebCodecs [14], it is possible to edit the captured raw media streams in real-time.
3. Processing power is required during media capture, encoding, decoding, and rendering.

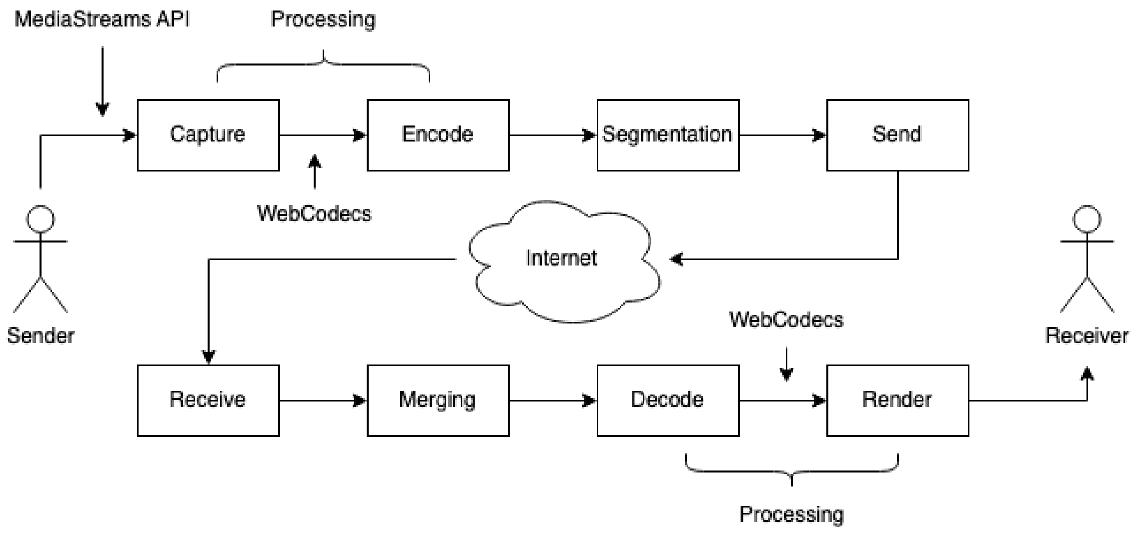

Figure 2.3: Media transport pipeline from capturing until rendering.

2.2.3 APIs

There are two main sets of APIs connecting to WebRTC [15]. The first set of APIs allows media and generic application data to be sent to and received from another browser or device implementing the appropriate set of real-time protocols. The second set of APIs allows local media, including audio and video, to be requested from a platform.

Sending and receiving media and generic application data from another browser or device requires the below WebRTC APIs [16]:

- **RTP Media API**: This API allows web browsers to send and receive media stream over a peer-to-peer connection.
- **RTCPeerConnection API**: Enables real-time audio, video, and data communication between web browsers using a peer-to-peer connection established through ICE candidates and signaling.
- **Peer-to-peer Data API**: This API allows web browsers to establish direct connections and exchange data without relying on central servers.

The MediaStream API [17] is a web standard that provides developers a toolset to capture, manipulate and stream media. It gives access to audio and video streams from various sources, such as webcam, microphone, and screen sharing. The main concept of the MediaStream API is to gather multiple streams into one MediaStream object that represents a continuous stream of data.

The key components and functionality of the MediaStream API are as follows:

- **MediaStreamTrack**: Represents media of a single type that originates from one media source in the User Agent.
- **MediaStream**: Group several MediaStreamTrack objects into one unit that can be recorded or rendered in a media element.

- **MediaDevices**: Entry point to the API used to examine and get access to media devices available to the User Agent.
- **getUserMedia()**: Prompts the user for permission to use their web camera or other video or audio input.

2.2.4 Media servers

Media servers facilitate the processing, routing, and distribution of media streams between peers. Media servers act as intermediaries, helping to overcome network limitations, enhance performance, and enable add-on features in real-time communication applications. A media server also provides the following features:

- **Signaling and Session Establishment**: Exchange session information, establish connections, and negotiate capabilities between peers.
- **NAT Traversal and Firewall Traversal**: Establish connectivity between peers and handle scenarios where direct peer-to-peer connections are not possible due to NATs.
- **TURN**: TURN servers may be included in the media server to relay media traffic between peers where connectivity is not feasible due to restrictive NATs.
- **Media Processing and Transcoding**: Mix multiple audio and video streams, apply filters and effects, and adapt media to different formats or bitrates.
- **Recording and Archiving**: Store media for later viewing.
- **Analytics and Quality Monitoring**: Monitor performance and user experience online.

Media servers provide great functionality and flexibility, but they may introduce extra latency and add additional costs compared to peer-to-peer connections. Media servers require physical servers and staff to manage them.

Media servers are divided into two groups based on the stream forwarding model: Selective Forwarding Units (SFU) and Multi-point Control Units (MCU). An SFU utilizes an "RTP mixer" to selectively forward media streams on a per-endpoint basis. This is done by selectively transmitting media to each endpoint based on the list of active speakers, unmuted participants, etc. [18]. Selectively routing media, SFUs usually cause higher CPU costs. On the other hand, an MCU [18] does not distinguish RTP sessions. Every user in a session will receive every other stream in parallel and the desired stream is selected at the client's side. This approach scales poorly and requires a huge network bandwidth. The difference between an SFU and MFU shown in Figure 2.4: the SFU forwards only the selected streams, while an MCU sends all streams to every connected client. Both kinds of a media server have their own place, but SFU solutions are more widespread because of the enhanced scalability.

2.2.5 Scaling

Scaling WebRTC services is difficult [19]. This is mostly attributed to the below issues:

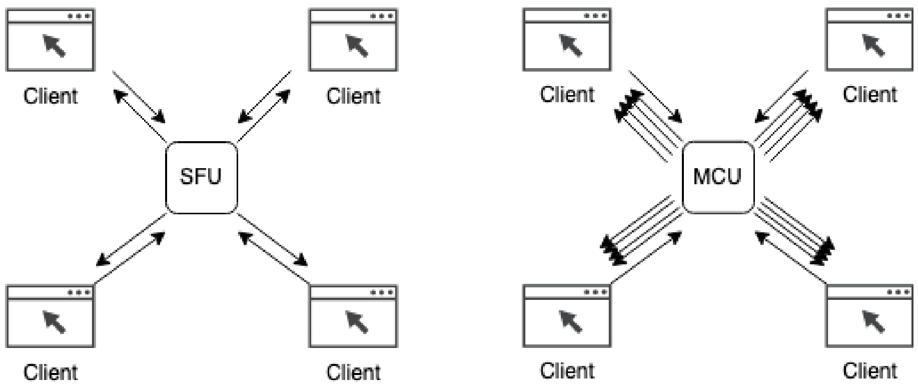

Figure 2.4: The difference between SFU and MCU media servers. The SFU only forwards the selected streams for client, while the MCU forwards every stream for every client.

1. How can media be shared between media servers?
2. How to synchronize data between media servers?
3. How can the application server manage multiple media servers?
4. How to keep low latency and good QoS?
5. How does the cost change with different scaling strategies?

Due to the above complexities, most popular WebRTC frameworks simply avoid scaling WebRTC sessions across multiple servers all together [10], fixing each WebRTC session to a particular server for its entire lifetime. In this way media servers can be as simple as possible. However, the capability of a single server is now a hard threshold on the number of users that can join a session. This is a massive problem for certain applications, like live streaming, which often involve thousands or hundreds of thousands of users.

The main motivation behind this work is to allow media servers to share sessions. While this requires more complex session handling logic and requires media servers to communicate among themselves, the obvious advantages are that it allows to join an arbitrary number of users into a single session (recall *session scaling*) and it facilitates deploying servers as close as possible to users in order to reduce latency (recall *geographic scaling*). For this, however, we will need to redesign the real-time communication stack from scratch. Most importantly, we will use Quick UDP Internet Connections (QUIC) as the transport protocol, which will obviate the need for NAT traversal and public IP addresses. This also means that it is not important to hold records about the clients in media server any more, which will allow us to simplify media servers to only processing and relaying media and scale media servers to any number of clients.

2.3 QUIC

QUIC [20] is the foundation for WebTransport [21] protocol, providing a reliable and efficient transport layer for web communication. QUIC is a secure general-purpose and connection-oriented protocol that creates a stateful interaction between a client and server.

- QUIC builds on top of UDP instead of TCP, what is used in Hypertext Transfer Protocol (HTTP) and WebSocket. Using a quick handshake, QUIC reduces connection establishment time, enabling faster data transmission.
- QUIC is capable of multiplexing multiple independent streams of data into a single QUIC connection. This reduces the overhead for establishing multiple connections.
- QUIC enables seamless migration of connections between different network interfaces or IP addresses.
- QUIC has an adaptive congestion control mechanism that dynamically adjusts the data transmission rate based on network conditions.
- QUIC connections are secured by Transport Layer Security (TLS) by default.

Figure 2.5 compares the QUIC handshake to the well-known TCP/TLS handshake. Crucially, the TCP/TLS handshake consists of twice the number of messages than QUIC, since QUIC combines the negotiation of cryptography and transport parameters into a single exchange.

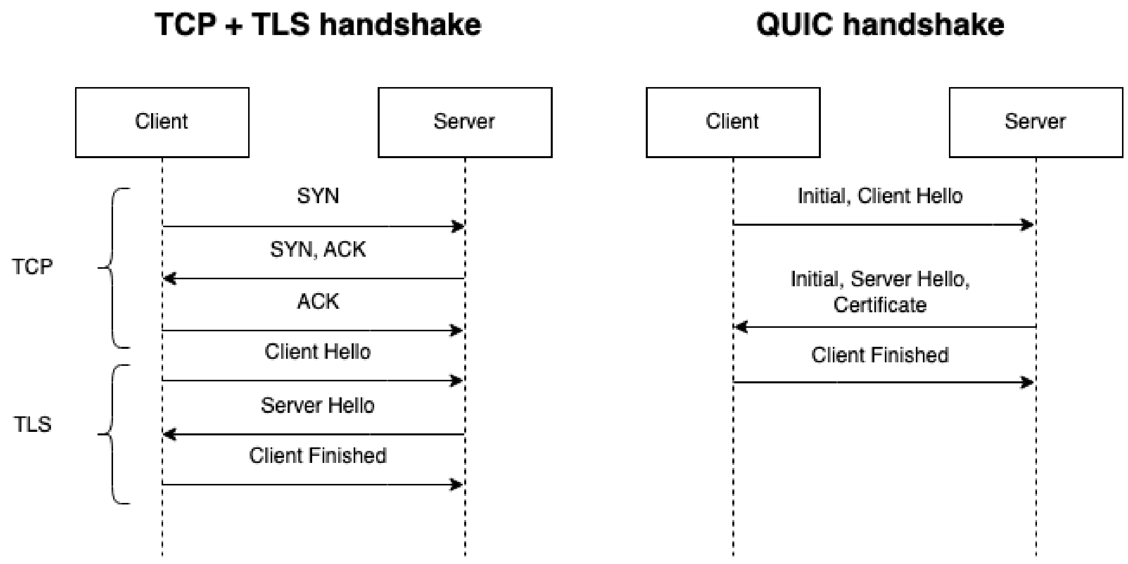

Figure 2.5: Comparison of the TCP/TLS and QUIC handshake.

Figure 2.6 shows an established QUIC connection. Endpoints communicate by exchanging QUIC frames, which carry control information and application data between endpoints. QUIC packets are encapsulated as UDP datagrams to facilitate deployment in existing systems and networks.

Each QUIC connection may contain multiple streams. Streams are ordered collections of data. Bidirectional streams let both endpoints submit data, whereas unidirectional streams let just one endpoint send data. The generation of streams is constrained, and the maximum quantity of data that may be transferred is limited, using a credit-based system. QUIC places transport control fields inside the encryption envelope, so QUIC has minimal exposure to the public network.

Currently, there is no standard way to send media over QUIC, but several approaches and research prototypes have been developed by industry leaders and academic institutions. These approaches aim to enhance media delivery, improve user experience, and leverage

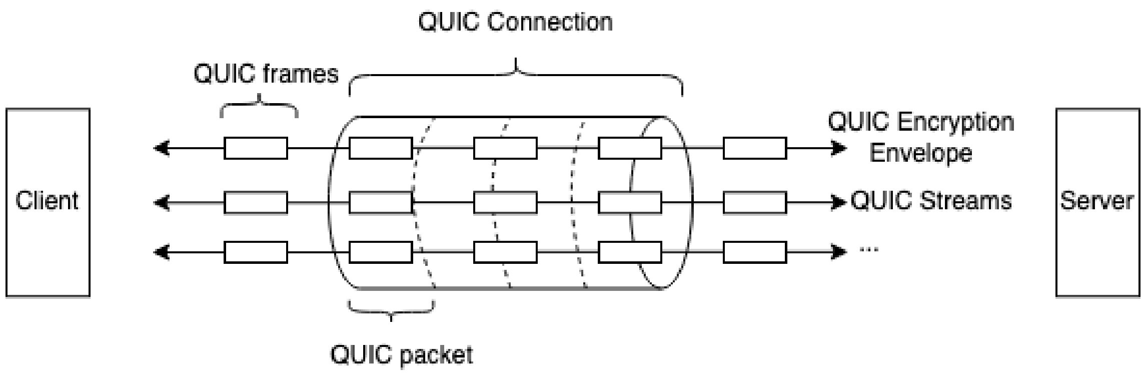

Figure 2.6: A QUIC connection with its components.

the benefits of QUIC's fast and reliable transport services. The four major initiatives are RTP over QUIC by Technical University Munich, Media over QUIC Transport (MOQT) by Twitch, Rush by Meta (formally known as Facebook), and QuicR by Cisco.

2.3.1 RTP over QUIC

RTP over QUIC [22] enables the transport of real-time data using QUIC streams and datagrams. The QUIC implementation must support the datagram extension and provide a way to determine the maximum datagram size for RTP packets.

RTP over QUIC takes a different approach to multiplexing compared to traditional RTP sessions. It uses flow identifiers instead of network addresses and port numbers to multiplex multiple RTP sessions over a single QUIC connection. QUIC does not provide built-in de-multiplexing for flows on datagrams, but applications can implement de-multiplexing mechanisms if required.

2.3.2 Media over QUIC Transport

MOQT [23] (fromally Warp) is a media transport protocol over QUIC. MOQT allows producer of media to publish data and have it consumed via subscription by any number of endpoints, providing high scale and low latency. MOQT utilizes the QUIC network protocol, either directly or via WebTransport, for the distribution of media.

MOQT is created to cooperate with various Media over QUIC (MoQ) streaming formats. These MoQ streaming formats specify the policies for discovery and subscription, as well as the encoding, packaging, and mapping of content to MOQT objects. Most importantly, MOQT creates a separate QUIC stream per each group of frames in the media.

2.3.3 Rush

Rush [24] is a bidirectional application-level protocol for ingesting live video on top of QUIC. Rush is intended to support new audio and video codecs, flexibility in the form of additional message types, and multi-track capability. Additionally, by using QUIC streams, Rush allows apps the opportunity to manage data delivery guarantees.

Rush separates each frame into a different QUIC stream. After sending one frame per stream, the stream is closed and a new one is opened for each new frame.

2.3.4 QuicR

QuicR [25] is a unified media delivery protocol over QUIC. QuicR aims to provide ultra-low latency for applications, such as interactive communication and gaming. It is based on a publish/subscribe model where entities publish and subscribe to media objects. QuicR uses one or more relays where the clients can publish media to and request media from. This relay architecture enables efficient large-scale deployments. Failure of a relay has a minimal impact on clients, as relays can easily redirect each client to a different relay.

QuicR uses named objects to send audio and video. Named objects are application level chunk of data with a unique name, a limited lifetime, and priority. Each named object represents a synchronization point in an audio or video stream. By putting media into named objects, QuicR can use other QUIC-based media mappings like Rush and MOQT. In addition, named objects can be cached in intermediate QuicR media relays, which allows QuicR to work similarly to a Content Distribution Network (CDN) for extreme scalability.

2.3.5 The QuicR delivery tree

One of the most useful features of QuicR is that it allows relays to be organized into a logical distribution tree topology [26], as shown in Figure 2.7 and Figure 2.8. The distribution tree is rooted in a particular relay, called the origin relay, which is responsible for controlling the namespace for a specific application. Publish messages are issued in the direction of the tree's origin relay and down from the root to the leaves towards the subscribers of the object. Note that the origin relay does not constitute a limit on scalability, because all users connected to a leaf relay will exchange media through that relay and hence only the traffic bisected by the root will cross the origin. Furthermore, it is easy to scale the tree by deploying a separate relay topology per application. The tree topology is not mandatory in QuicR, it is chosen as the default distribution topology only for its simplicity.

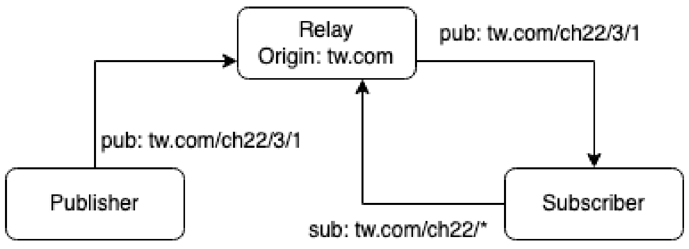

Figure 2.7: Single relay QuicR delivery tree [26].

Figure 2.7 shows a streaming architecture rooted in the Origin relay for the domain of *tw.com.* In this architecture, the Publisher publishes its media on channel 22, while media consumption happens at the Subscriber, which subscribed for every message received on channel 22.

Figure 2.8 shows a two layer deliver tree. In this example, we have 4 participants where Alice is the only publisher, while everybody else is a subscriber. Alice publishes a high and low resolution stream. While Bob subscribes to the low resolution stream only, the others subscribe to every stream. All subscriptions are sent to the origin relay and the on-path relays cache these subscribes making a short-circuited delivery of published data

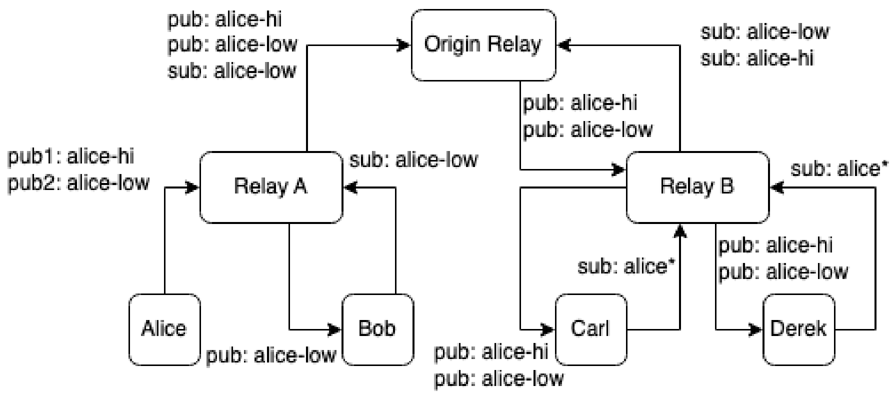

Figure 2.8: Multi relay QuicR delivery tree with one publisher and three subscriber [26].

at the relays. This means that Alice sends the low resolution stream to Bob through only the A relay, while the others get the streams originated from the origin relay.

We found QuicR relay topologies ideal for our work for a number of reasons. First, QuicR naturally supports session scaling, since publishers and subscribers can be connected to any relay, but they can still communicate via the relay topology. QuicR will handle relay-to-relay communication automatically. Second, geographic scaling is also easily supported by deploying a relay to each geographic location of interest and joining these to a common origin. Since it naturally supports both session scaling and geographic scaling, we will build our new scalable RTC architecture on top of the QuicR media transport protocol.

Chapter 3

Architecture design

With the help of the QuicR standard, we build a new RTC service addressing the session scaling and geographic scaling of typical WebRTC-based services. In Section 2.2.5, we found that both session scaling and geographic scaling require the need to be able to share sessions between servers. Since QuicR uses a similar approach to the CDN services, it is capable of cache data while relaying media between clients. In this chapter, we dive into the architecture of the system we built on top of QuicR and the required workflows to transfer real-time media.

Our architecture consists of three main components: Media plane, Control plane, and the client itself. We also discuss the workflows that allow clients to create, join, and delete sessions.

1. **Media Plane**: The media plane will handle the relaying of all audio and video data. This media plane resembles to a CDN.
2. **Control Plane**: The control plane is responsible to establish a connection between the client and the media plane without communicating with the media plane.
3. **Client**: The client will be able to generate audio and video traffic to test the system.

The overall architecture is depicted in Figure 3.1. Clients establish connections through the application server and then exchange media through the relays.

Session establishment requires a set of commands between the client and the application server. Commands exist to a create a session, join an existing session, and delete an existing session. The application server will process every command from the clients and notify the clients if something changes with the session they participate in. We decided to create a design where the application server will never communicate with media servers. This makes it possible for the application server to distribute this information to its clients, improving scalability.

We organize the relays into a tree topology, where only the leaf relay servers are accessible to clients through the cloud load-balancer. This ensures an even traffic distribution over the relays, making session scaling and geographical scaling possible. In the media plane we use a two-level tree, where we have a single origin relay and multiple leaf relays. Session sharing among the leaf relays occurs through the origin relay. Our architecture allows the operator to create as many parallel relay trees as needed; this makes sure that the origin relay never becomes a bottleneck.

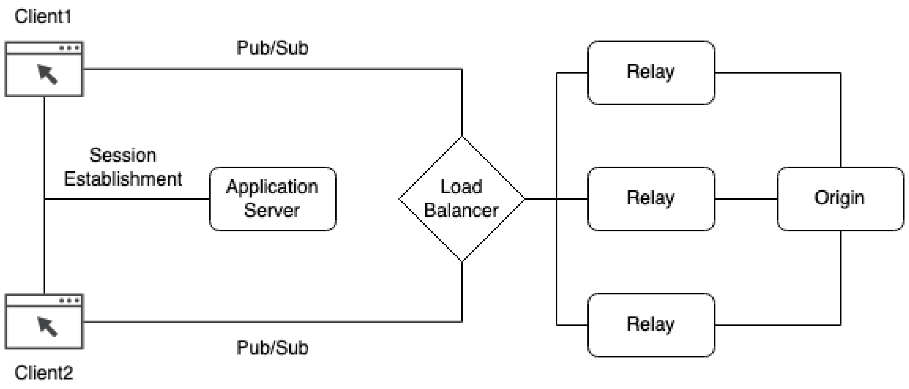

Figure 3.1: The architecture of our service.

3.1 Media plane

A CDN is a group of geographically distributed servers that forward and cache content close to end users. CDNs allow quick transfer of assets needed for loading Internet content, including HTML pages, JavaScript files, images, and videos. CDN are popular among large-scale service providers like Meta, Netflix, and Amazon.

Using a CDN will help improve the performance of a service because the hosted media can be cached on the edge of the network. Therefore, users do not need to communicate with the origin server, which has all the media. Also, it will help to reduce the required bandwidth within our service. The advantages of utilizing a CDN are:

- Improving load times.
- Reducing bandwidth cost.
- Increasing content availability and redundancy.
- Improving security.

Figure 3.2 shows the Origin server, CDN relays, and clients connected to each other. The Origin server has all the content and the CDN relays are connected to this server on a high-speed link, while users are connected to the relays on the regular network. Thanks to caching, the CDN servers can store content for the users. If some content is not available, they will ask this content from the origin server via a high-speed link.

In the case of large data such as video, the relays use the publish/subscribe model. This model will enable the users to subscribe to a specific content and get the data chunks when they want it. Also, if a user wants to upload something, the relay can subscribe to the user's feed, and the user can publish data chunks to the relay.

Since all communication occurs via the cache and potentially via multiple relays, end-to-end latency may increase substantially. One of the most interesting conclusions of our work is the finding that despite the added latency due to caching, the CDN architecture is still perfectly capable of serve RTC use-cases without prohibitive delay penalty (see the evaluations later). At the same time, the CDN model will allow us to scale our system to essentially any number of clients and geographic locations in the cloud.

To demonstrate the use of the CDN model in our architecture, Figure 3.3 shows measurement results from the earlier work [28]. This work used data centers (red squares)

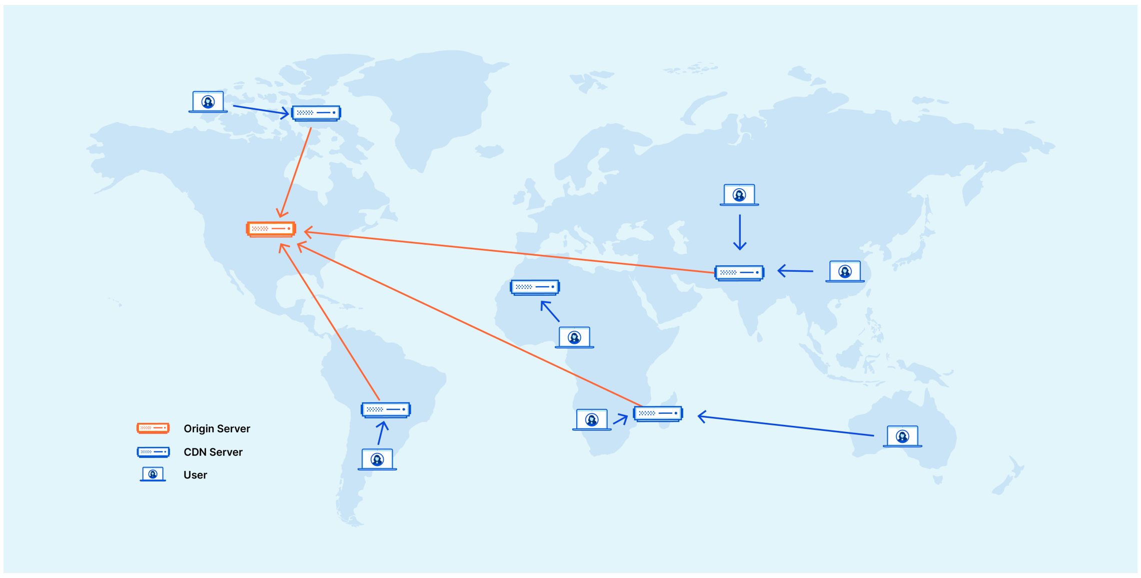

Figure 3.2: A possible distribution of a CDN network [27].

and `Speedtest.net` server (green circles) to evaluate latency increase between clients and servers that are further apart. The red star designates the client's location where the tests originate. The results show that a CDN-like media plane architecture indeed supports both session scaling and geographic scaling.

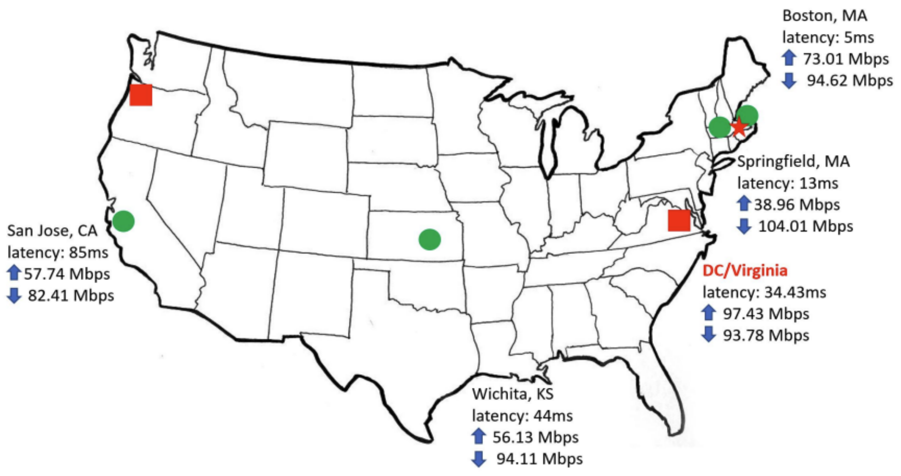

Figure 3.3: Demonstrate the increasing latency that accompanies access to servers that are further from the client, where the client is the red star [28].

3.2 Control plane

The control plane is the bridge between clients and the media plane. In our case, the control plane is the application server which enables clients to initiate sessions.

The application server in WebRTC use-cases communicates with the media servers, which makes it harder to scale the number of media servers. Because of this reason, we decided that the application server will not communicate with the media servers. The application server knows a single access point to the media servers' load-balancer, and this is enough for the clients to publish and subscribe media.

The application servers have to following tasks:

- Register sessions and users.
- Join clients into an existing session.
- Notify clients.
- Provide reachability information for clients to the media servers.

For this work, we are implemented a control plane component from scratch.

3.3 Client

The client side of our architecture plays a pivotal role in the orchestration of sessions and serves as a vital component in traffic generation. Clients are endowed with the responsibility of initiating and managing sessions. To initiate and manage sessions, the client should be able to do following tasks:

- Create a new session.
- Delete a session.
- Join into an already existing session.
- Leave a session.
- Publish media stream.
- Subscribe to a media stream.

Session management is happening through a client written by us, while media publishing and subscribing is the job of the QuicR-based media plane.

3.4 Workflows

The client has four commands, which means three different workflows for initiating and managing sessions. These three commands are the following:

1. `createRoom`: Creating a named session.
2. `joinRoom`: Joining into a named session.
3. `deleteRoom`: Deleting a named session.

3.4.1 createRoom

The Figure 3.4 shows the process of creating a new session. The client first sends a `createRoom` message to the application server. Then the application server will create a record about the created room and the registered user and send back an acknowledgment message to the client that the room is created and the client can start publishing media to the media server.

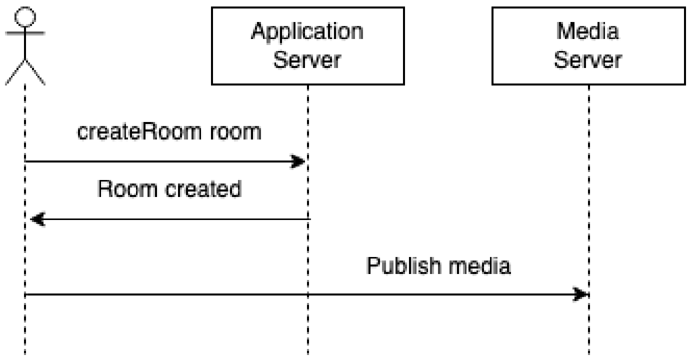

Figure 3.4: The process of creating a room.

3.4.2 joinRoom

Figure 3.5 shows how the `joinRoom` command works. There is an existing room where a user is publishing media, and another user joins into this room. The application server implements the `joinRoom` command similarly to the `createRoom` workflow, with two exceptions. First, the application server will not create a new room record, just update the clients participating in the same room. Second, every other client in the same room gets a `newUser` message with information about the new client's stream information.

The new client starts to publish its media stream and subscribes to every other media stream in the same room. In addition, every other client in the same room will subscribe to the new client's media stream.

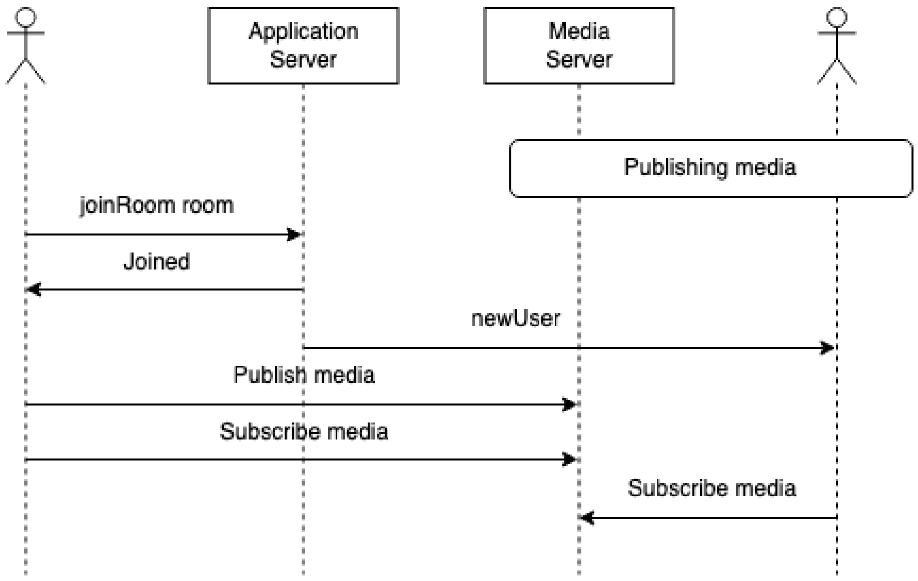

Figure 3.5: The process of joining a room.

3.4.3 deleteRoom

Figure 3.6 shows how a client can delete a room. First, the client will stop publishing media and unsubscribe from every other media stream. Then, the client will send a `deleteRoom` message to the application server. The application server will delete the room from the records and the clients within the room. However, before the application does it, it will notify every other client in the room that the room is deleted, so stop the publishing media and unsubscribe from every other stream. At the end, the client gets an acknowledgment of the room deletion.

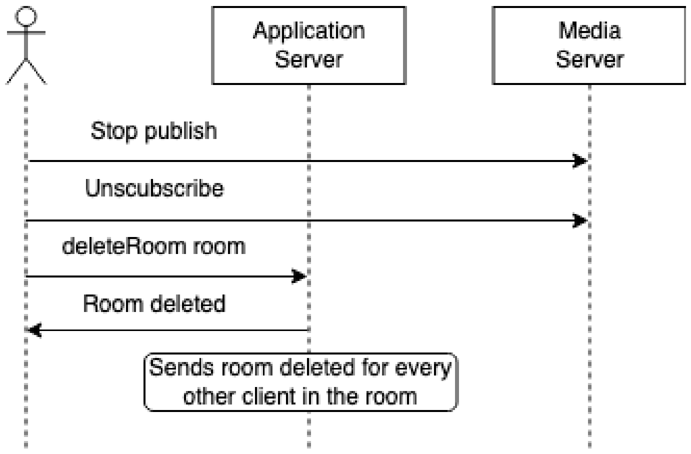

Figure 3.6: The process of deleting a room.

Chapter 4

Implementation

In this chapter, we describe the implementation details of our system, including the media server, application server, and the client. The media server will serve us as the media plane and the application server is the control plane. The key goals of the implementation were to achieve session scaling and geographic scaling. In the subsequent sections of this chapter, we describe the implementation details of each component of the system.

4.1 Media server

There are multiple available implementations for the Media over QUIC standard (see Section 2.3), such as RTP over QUIC, MOQRT, Rush, and QuicR. We decided to use QuicR because it has the most developed code among the listed implementations. However, it can be replaced by any other implementation from the MoQ Internet Engineering Task Force (IETF) working group easily (recall, the application server never communicates with the media relays, which makes relays easy to replace).

We used the existing QuicR implementation called QUICRQ. QUICRQ is a media server that uses the QuicR implementation for media transport. It is written in C. Note that QUICRQ is not a commercial grade software. This means that not all parts of the implementation are fine tuned. QUICRQ provides an executable named `quicrq_app`, which makes it possible to use it in four different modes:

- **Origin**: The origin server can be a relay between users and relays at the same time.
- **Relay**: Relays media between producer and consumer. It supports caching and selective dropping to optimize the media transmission performance.
- **Producer**: Endpoint which generates the media stream.
- **Consumer**: Endpoint which receives the media stream.

Regarding media processing and relaying, we did not have to modify the code at all. However, we added new functionality to implement logging for evaluation purposes. Our logging functionality will report for each packet sent or received the following information:

- **Timestamp**: The Unix timestamp for each packet in the sending and receiving momentum.

- **Length**: The length of the packet.
- **Group ID**: The group id of the sent frame.
- **Object ID**: The object id of each object in a group.

4.2 Application server

The application server is the bridge between the clients and the media servers. When a client wants to use our service, they have to register themselves through the application server. If they are registered, they can create sessions or connect to existing sessions. During this process, the clients provide information about themselves, while application server provides information about the media servers, the connection details, and media descriptors.

The benefits of using an application server is to be able to keep the media plane as simple as possible. If the media server handles these tasks too alongside with processing and relaying the media, it can stress out the media server causing performance and/or reliability degradation. In this setup, we were able to use QUICRQ, which does not store information about the sessions at all. This makes it possible to create an application server that does not communicate with the media server at all. By eliminating the communication between the media server and the application server the session scaling and geographic scaling become easier to implement.

The application server was written in the Go programming language. Go is an open-source programming language that makes it simple to build secure, scalable systems. Go implements a variant of the Communications Service Provider (CSP [29]) model, in which channels are the preferred method for two threads (called goroutines in Go) to share data. A goroutine is a lightweight user-space thread-like parallel process. This approach is the opposite of the frequently used model, where data shared globally across threads. This means low latency with intercommunicating goroutines and makes it possible the share data between goroutines via channels.

Go is a good fit for writing our application server, because it needs to receive messages constantly and in parallel from the clients and process them quickly. In this way, we can start a goroutine, which handles the incoming messages and creates new goroutines to process the received messages and send responses to the clients. The main goroutine handles the incoming messages, which has three types:

1. **`createRoom <room name>`**: A client can create a room with a specified room name.
2. **`joinRoom <room name>`**: A client can connect to a room with a specified room name.
3. **`deleteRoom <room name>`**: A client can delete a room with a specified room name.

Figure 4.1 shows how the `joinRoom` message is handled by the application server. The message receiver goroutine is responsible for receiving messages from the clients, while the handler goroutine is responsible for processing the message. The message handler will spawn a new goroutine after finishing the message processing. The notify goroutine will notify every client in the room via a `newUser` message.

The room and client information is stored in the following structure by the application server.

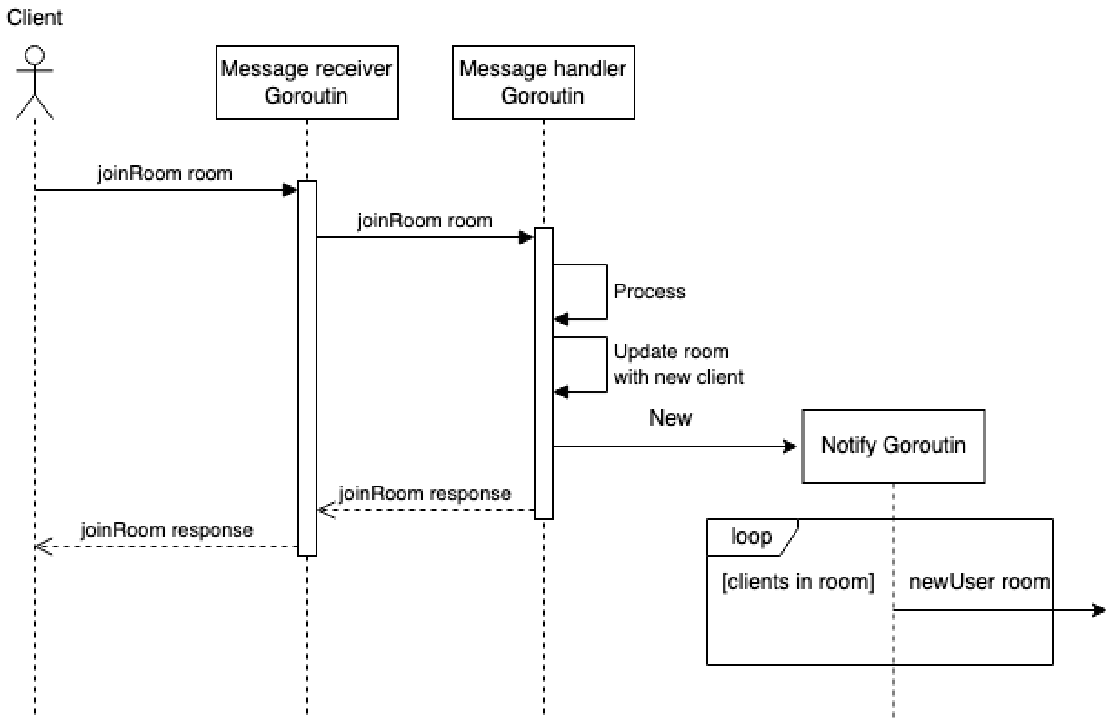

Figure 4.1: The process of processing a `joinRoom` messages within the application server.

- **Room**
  - **Id**: Identifier of the room.
  - **Users**: List of clients in the room.
  - **Server**: Media server's address and port.
- **User**
  - **Id**: Identifier of the client.
  - **Conn**: Connection context for the client.

4.3 Client

Our service cannot be used without a client. The client is a piece of software, which makes it possible for users to connect to a service. The client is able to initiate a session through the defined messages, such as `createRoom`, `joinRoom`, and `deleteRoom`. The client is also able to ingest media into our service through `quicrq_app`. With `quicrq_app`, we were able to ingest a prerecorded video stream into our service and simulate actual traffic.

As the client and the application server share a significant amount of code, the client is also written in Go. The clients communicate with the application server through WebSocket and use the message types described above. The client has a Command-Line Interface (CLI) interface, letting the user to type commands (like `joinRoom`, or `deleteRoom`) that will be sent to the application server for processing.

The main challenge was scaling the number of clients. To make the number of rooms and clients configurable for our evaluations, we wrote a test script as well. The script creates background processes for each client and pipe commands into these processes. In this way, we can test performance and identify the maximum number of users our application server and media server can handle in parallel with restricted resources.

Chapter 5

Evaluation

In this chapter, we conduct a series of extensive evaluations to understand whether the proposed CDN architecture can deliver the real-time latency requirements and demonstrate that our design supports session scaling and geographic scaling beyond what the state-of-the-art WebRTC systems can provide. Our results show that our system has similar performance as the widely used Jitsi WebRTC media server distribution [11, 12], while it can perform session scaling and geographic scaling as well.

5.1 Deployment model

In order to obtain a fair evaluation, we created a deployment model so that our service and Jitsi are compared with equal resources. We consider four deployment scenarios:

1. Baseline deployment, where both the services and the traffic generator are deployed on the same machine (baseline).
2. Local traffic generator with remote services located in Europe. We used this scenario to demonstrate session scaling (local-relay-eu).
3. Local traffic generator with remote services located in the USA (local-relay-us).
4. One traffic generator and server in the USA (New York), and one traffic generator and server in Europe. We used this scenario to demonstrate geographic scaling.

The traffic generator in each case was a laptop running Ubuntu 22.04 with the following hardware: Ryzen 3 4300U (4 cores, 4 threads), 16 GB of RAM, and 1 Gigabit Ethernet. This was enough to generate enough traffic to compare our service with Jitsi.

During the evaluations we measured three standard media transmission quality metrics: the round-trip-time (RTT), defined as the average time between sending a packet and receiving a response, jitter, defined as the mean delay variation, and packet loss, measured as the ratio of the number of packets lost at the destination and the total number of packets sent. High RTT and jitter can cause problems, such as dropped packets or delayed audio and video playback. The acceptable scale of RTT is everything below 300 ms [2], and for both the RTT and the jitter the smaller the better.

For the Jitsi measurements, we used webrtcperf to generate realistic WebRTC traffic. Webrtcperf is an open-source project, which automatically joins a room, plays back media, and measures performance. For the measurements with our system, we used the test scripts described in Chapter 4 to generate traffic and measure the media transmission quality metrics.

5.2 Session scaling

First, we measured the effects of session scaling on Jitsi and our system. We compared three scenarios in the baseline deployment model:

- **Single Jitsi**: Jitsi service with 1 CPU and 1 GB of memory
- **Single Relay**: Our service with a single relay with 1 CPU and 1 GB of memory.
- **Two Relays**: Our service with two relays, both with 1 CPU and 1 GB of memory.

We ran these scenarios with varying number of users, in particular, 2, 10, 15, 20 users. The RTT can be seen at Figure 5.1, while the jitter results are given on Figure 5.2. The packet loss in every scenario was negligible, always remaining under 0.5%, except for the scenario where Jitsi was measured with 20 users where traffic loss was 100% (i.e., all traffic was lost).

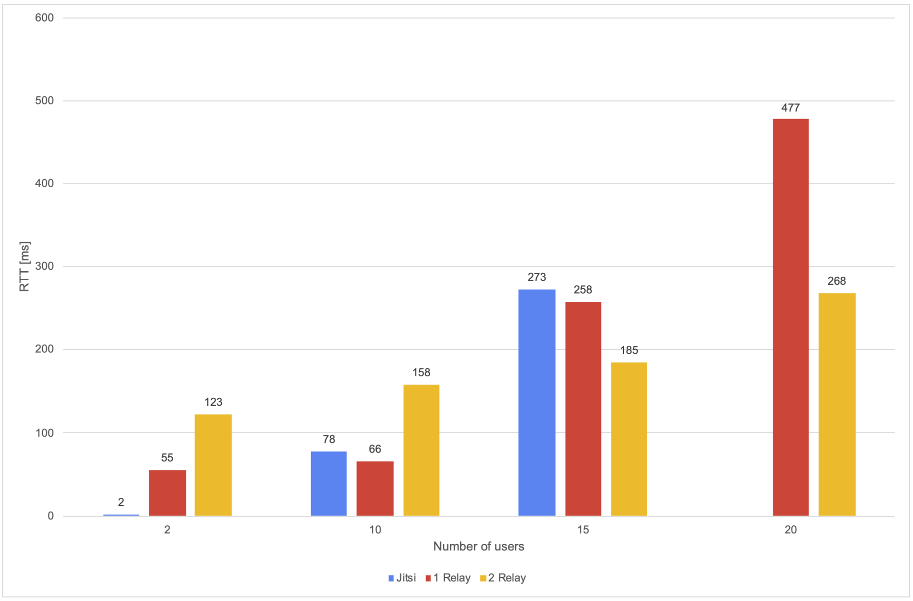

Figure 5.1: Changes of RTT with scaling the number of users.

Our most important observations are as follows. At small load (2 users), Jitsi produces much smaller RTT and jitter than our system does. We believe this is the consequence of the extra delay introduced by the cache(s) in the QUICRQ relays in our system. However, we note that the RTT and jitter caused by our system still guarantee good quality media transmission. However, as the load increases, the advantages of session scaling become more apparent. With 10 and 15 users, Jitsi and our system produce similar RTT and jitter. However, *Jitsi stops working with 20 users* due to exhausting the resources of the single server. In contrast, QUICRQ keeps working, but the RTT is much higher now, which indicates resource exhaustion for the 1-relay case. As the number of relays is scaled to 2, we can see a significant improvement in the RTT, which demonstrates the usefulness of session scaling.

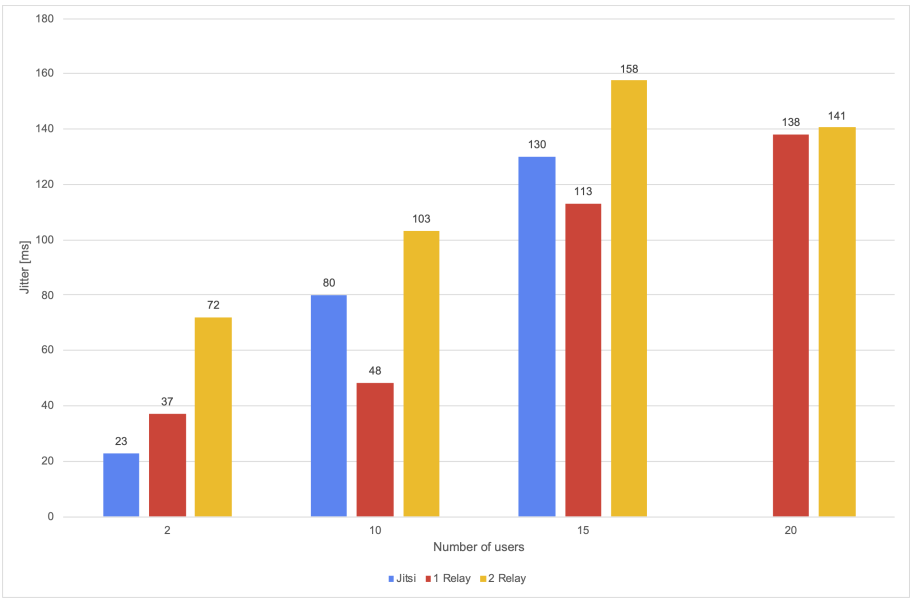

Figure 5.2: Changes of jitter with scaling the number of users.

We note, however, that at small load the RTT, and especially the jitter, are worse with 2 relays than with 1 relay. This is because of the load-balancing mechanism, which randomly distributes users' `joinRoom` messages across the relays. This results that some users are connected through 2 relays even at small load, which shows up in our measurements as worse RTT and jitter. The negative effect of session scaling is low, and at high loads session scaling clearly delivers significant improvement.

5.3 Geographic Scaling

Next, we demonstrate the importance of geographic scaling. With geographic scaling, users are always connected to the closest server. In this way, users that are located close to each other exchange information over the same local relay in a short-circuit way, and only send data through the remote relay if they want to publish media for remote subscribers using the other relay.

Figure 5.3 and Figure 5.4 show the difference between Jitsi and our service in terms of RTT and jitter in different deployment scenarios.

Our observations are as follows. First, the RTT clearly rises as the distance is longer, but for both services, it remains under 300 ms [2]. In addition, jitter is lower with our service than with Jitsi. This may be a positive effect of caching, that can smooth delay variation over large distances.

To understand the effects of geographic scaling, consider again the video-conferencing service discussed previously. Recall, in this scenario the first user joins from the US, so the video-conferencing server is deployed into the US, and then lots of users join from the EU and the US-based user leaves. Since with traditional WebRTC services like Jitsi users are pinned to media servers for the entire lifetime of the session and sessions cannot be moved or shared across media servers, all users are now in the EU yet they exchange media traffic over a remote server located in the US. This means they experience a performance as in the local-relay-us scenario, i.e., 117 ms RTT and 101 ms jitter. In contrast, our system is geographically scaled across a US-based and an EU-based relay. Thus, all EU-

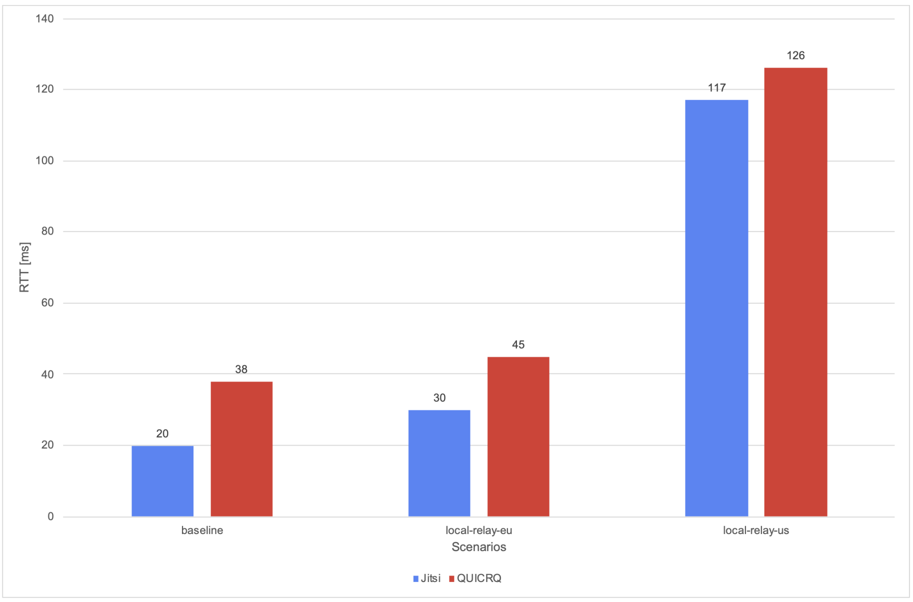

Figure 5.3: RTT comparison of Jitsi and QUICRQ with different scenarios.

based users can exchange traffic over a local, EU-based relay. This corresponds to the results for the local-relay-eu scenario for QUICRQ, i.e., 45 ms RTT (about 2.5 times as small as with Jitsi, or 40% of the RTT of Jitsi) and 37 ms jitter (about 2.7 times as small as with Jitsi, or 36% of the jitter of Jitsi). This clearly demonstrates the positive effects of geographic scaling.

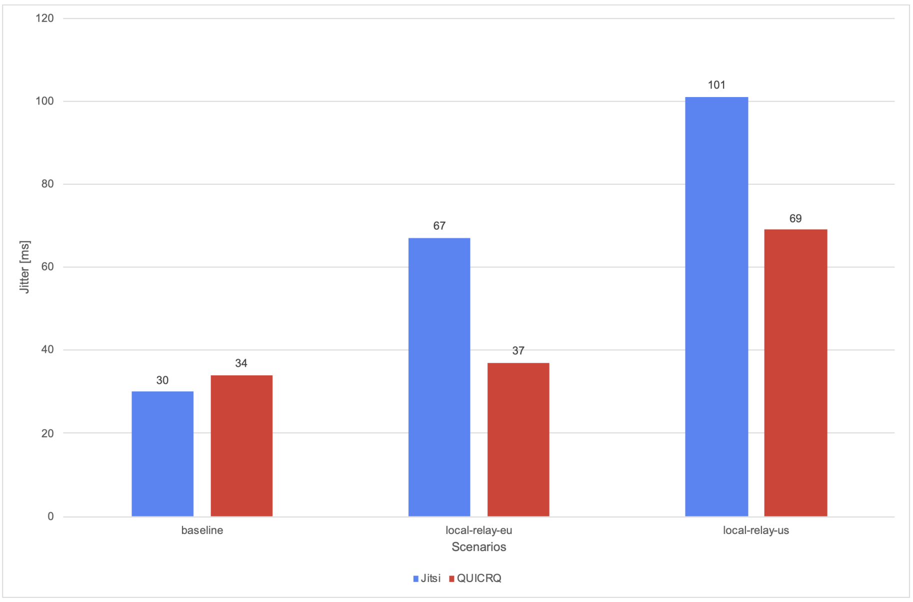

Figure 5.4: Jitter comparison of Jitsi and QUICRQ with different scenarios.

Chapter 6

Conclusion

The WebRTC technology reshaped the way we ecommunicate in the digital world. It serves as the de-facto way for developing RTC applications, including videoconferencing, live streaming, and online gaming. These applications set a strict requirement for latency and jitter, different from traditional Web services. As we have witnessed, the demand for RTC solutions has increased over the years, especially during the COVID-19 pandemic, when millions of individuals had to work remotely.

Despite the success of WebRTC, certain limitations have remained unresolved. The issues of session scaling and geographic scaling have posed significant challenges. The former challenge involves the complexity of scaling out WebRTC sessions to accommodate an increasing number of users, while the latter addresses the difficulty of deploying WebRTC servers as close to users as possible.

In response to these challenges, this work presents a new RTC architecture designed to address both session scaling and geographic scaling. We explored the complexity of WebRTC and its protocols, delving into the core issues that hinder its scalability and efficiency. Our design and implementation introduced a solution that enables the distribution of sessions across multiple servers and multiple geographic locations, enhancing the performance of large-scale RTC applications.

To validate our approach, we built a new application server with a custom client and conducted comprehensive evaluations using a QuicR media plane with prerecorded video streams. We deployed our service into two cloud data centers, one in the USA and another in the EU. Our results show that for large sessions, our RTC architecture delivers a consistent performance improvement over the state-of-the-art WebRTC applications, such as Jitsi, while it reliably provides session scaling and geographic scaling.

The impact of WebRTC on RTC services cannot be overstated, and as the technology continues to evolve, it becomes even more important to address its limitations. This work presents a new approach to build a future of RTC services with improved scalability and user experience. We believe this is an important step toward addressing the challenges of WebRTC's session scaling and geographic scaling limitations. However, there is more work to do in the future, such as:

- Experiment with different QUIC-based media servers and understand the consequences of caching to media performance.
- Write a real video-conferencing server that is accessible from a standard Web browser, using the webcam and microphone of users as media source.

Bibliography

1. Levent-Levi, T. *What is WebRTC and What is it Good For? (2023. 10.)* https://bloggeek.me/what-is-webrtc/.
2. Union, I. T. *One-way transmission time (2023. 10.)* https://www.itu.int/rec/T-REC-G.114-200305-I/en. May 2003.
3. Blum, N., Lachapelle, S. & Alvestrand, H. WebRTC: Real-Time Communication for the Open Web Platform (2023. 10.) *Commun. ACM* **64,** 50–54 (2021).
4. *WebRTC streaming on Echo Show (2023. 10.)* Amazon. available online: https://amazon.developer.forums.answerhub.com/questions/197956/webrtc-streaming-on-echo-show.html. 2022.
5. Iqbal, H., Khalid, A. & Shahzad, M. Dissecting Cloud Gaming Performance with DECAF (2023. 10.) *Proc. ACM Meas. Anal. Comput. Syst.* **5** (2021).
6. Murillo, S. & Gouaillard, A. *WebRTC-HTTP ingestion protocol (WHIP) (2023. 10.)* IETF draft. 2022.
7. Inc., G. M. I. *Web Real-Time Communication (WebRTC) Market (2023. 10.)* https://www.gminsights.com/industry-analysis/web-real-time-communication-market.
8. For the Curious, W. *History (2023. 10.)* https://webrtcforthecurious.com/docs/10-history-of-webrtc/.
9. Keranen, A., Holmberg, C. & Rosenberg, J. *Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal (2023. 10.)* RFC 8445. 2018.
10. *LiveKit documentation: Distributed Setup (2023. 10.)* https://docs.livekit.io/realtime/self-hosting/distributed. 2023.
11. 8x8, I. *Jitsi (2023. 10.)* https://jitsi.org/.
12. Grozev, B., Politis, G., Ivov, E. & Noel, T. Considerations for deploying a geographically distributed video conferencing system (2023. 10.) (Nov. 2018).
13. Foundation, M. *Codecs used by WebRTC (2023. 05.)* https://developer.mozilla.org/en-US/docs/Web/Media/Formats/WebRTC_codecs.
14. Foundation, M. *WebCodecs API (2023. 05.)* https://developer.mozilla.org/en-US/docs/Web/API/WebCodecs_API.
15. Consortium, W. W. W. *Web APIs (2023. 05.)* https://webrtc.github.io/webrtc-org/web-apis/.
16. Consortium, W. W. W. *WebRTC: Real-Time Communication in Browsers (2023. 05.)* https://w3c.github.io/webrtc-pc/.

17. (Cisco), C. J., Corporation), B. A. (, (Mozilla), J.-I. B., (Google), H. B. & (Apple), Y. F. *Media Capture and Streams (2023. 05.)* https://www.w3.org/TR/mediacapture-streams/.

18. Westerlund, M. & Wenger, S. *RTP Topologies (2023. 10.)* https://datatracker.ietf.org/doc/html/rfc7667. Nov. 2015.

19. Silverlock, M., van der Mandele, A. & Allworth, J. *Real-Time Communications at Scale (2023. 10.)* The Cloudflare Blog. available online: https://blog.cloudflare.com/announcing-our-real-time-communications-platform. 2021.

20. Iyengar, J. & Thomson, M. *QUIC: A UDP-Based Multiplexed and Secure Transport (2023. 10.)* https://datatracker.ietf.org/doc/rfc9000/. May 2021.

21. Vasiliev, V. *The WebTransport Protocol Framework (2023. 10.)* https://datatracker.ietf.org/doc/draft-ietf-webtrans-overview/.

22. Ott, J. & Engelbart, M. *RTP over QUIC (2023. 02.)* https://datatracker.ietf.org/doc/draft-ietf-avtcore-rtp-over-quic/.

23. Curley, L., Pugin, K., Nandakumar, S. & Vasiliev, V. *Media over QUIC Transport (2023. 05.)* https://datatracker.ietf.org/doc/draft-lcurley-moq-transport/.

24. Pugin, K., Frindell, A., Ferret, J. C. & Weissman, J. *RUSH - Reliable (unreliable) streaming protocol (2023. 05.)* https://datatracker.ietf.org/doc/draft-kpugin-rush/.

25. Jennings, C. F. & Nandakumar, S. *QuicR - Media Delivery Protocol over QUIC (2023. 03.)* https://datatracker.ietf.org/doc/draft-jennings-moq-proto/.

26. Jennings, C. F. & Nandakumar, S. *QuicR - Media Delivery Protocol over QUIC (2022. 09.)* https://www.ietf.org/archive/id/draft-jennings-moq-quicr-arch-00.html.

27. Cloudflare, I. *What is a content delivery network (CDN)? How do CDNs work? (2023. 10.)* https://www.cloudflare.com/en-gb/learning/cdn/what-is-a-cdn/.

28. McClellan, M. & Bauer, S. WebRTC Based Network Performance Measurements (2023. 10.)

29. Anand, A. Communicating sequential processes(CSP) for Go developer in a nutshell. (2023. 10.) *Medium* (Mar. 2019).