# WebRTC Multi - Conferencing Framework

1st Johannes Franz Müller
*University of Applied Sciences Cologne*
Cologne, Germany
johannes_franz.mueller@smail.th-koeln.de

***Abstract*—This paper presents a comprehensive framework for multi-conferencing using Web Real-Time Communication (WebRTC) technology. The proposed framework leverages the capabilities of Mediasoup, a Selective Forwarding Unit (SFU), to efficiently manage and forward media streams among participants, ensuring scalability and flexibility in real-time communication. Key features include support for audio, video, screen sharing, and text communication, alongside robust security measures like meeting codes and HTTPS hosting. Additinally NAT traversal techniques, are discussed to provide connectivity via different Networks.**

***Keywords*—WebRTC, Mediasoup, Real-time communication, Signaling-server, Media-server, Coturn**

## I. Introduction

In today's digital age, real-time communication has become increasingly crucial for both personal and professional interactions. The demand for seamless, high-quality audio and video communication has skyrocketed, driven by the rise of remote work, online education, and virtual events. One approach to implement real time communication is by the use of WebRTC, which provides a robust framework for real-time communication directly within web browsers.

This paper focuses on the implementation of a real-time conferencing framework using the WebRTC technology.

### A. Objective

The framework's objective is to facilitate the sharing of audio, video, screen, and text among clients. It will support different user groups by offering the option to create rooms for these groups. Additionally, the framework will facilitate multi-conferencing, allowing multiple users to participate in conferences simultaneously.

To ensure a secure conferencing environment, various security measures will be implemented. Users will be required to provide a meeting code to access rooms, preventing unauthorized entry. The framework will be hosted over HTTPS, necessitating the use of certificates on the server side. In order to ensure connectivity between clients in different networks, NAT traversal techniques will also be implemented.

## II. Research

### A. WebRTC

WebRTC is an open standard technology that enables real-time communication capabilities through web browsers and supports the sharing of video, voice and data. It enables developers to create robust voice and video communication solutions. WebRTC is available in all modern browsers and uses standard JavaScript APIs for web implementation. The WebRTC project is open source and supported by a number of major technology companies including Apple, Google, Microsoft and Mozilla. [1]

### B. Connection Approach

WebRTC utilizes three main connection approaches: Mesh Network, MCU (Multipoint Control Unit), and SFU (Selective Forwarding Unit). Each approach has distinct advantages and disadvantages in terms of scalability, complexity, and resource usage. [2]

*1) Mesh Network:* A Mesh Network in WebRTC involves each participant has a peer-to-peer connecting to every other participant. The pro and cons of the usage of an mesh network can be seen below.

| Pros | Cons |
|---|---|
| Simplicity: Easy to set up and manage | Scalability: Not suitable for large conferences as each participant must send and receive streams from every other participant |

TABLE I
Pros and Cons Mesh Network

*2) MCU:* A MCU is a server that receives streams from all participants, processes them, and sends a single composite stream back to each participant. The pro and cons of the usage of MCU can be seen below.

| Pros | Cons |
|---|---|
| Compatibility: High compatibility with various legacy systems. | Resource Intensive: High CPU usage due to transcoding and mixing of streams. |
| Centralized Control: Simplifies management by centralizing the processing and distribution of media streams. | Unflexible: Client receives one media stream, which makes seperate actions like volumne changes difficult |

TABLE II
Pros and Cons MCU

*3) SFU:* A SFU is a server that receives media streams from all participants and selectively forwards them to others without processing. Both, with SFU and MCU there is no peer-to-peer connection from client to client, but the peer-to-peer connection is between each client and the media server (MCU or SFU). The pro and cons of the usage of SFU can be seen below.

| Pros | Cons |
|---|---|
| Scalability: Efficient handling of a large number of participants by only forwarding selected streams. Additionally, media streams can be processed on the client side, so each client can process streams differently. | Complexity: More complex to set up compared to P2P networks and MCU, because routing of different media streams has to be configured. |
| Bandwidth Management: Can reduce bandwidth usage by selecting which streams to forward. | |

TABLE III
PROS AND CONS SFU

Considering these three approaches with their pros and cons, the approach of an SFU is the most fitting one to realize conferencing with multiple clients, because it handles multiple clients flexible and also contains the possibility for client-side processing of single media streams. This means, for example, clients can handle the volume of each participant differently.

### C. Mediasoup

Mediasoup is a popular implementation of an SFU and is built on top of WebRTC. It efficiently forwards media streams to multiple clients without the need for complex server-side mixing, allowing each client to independently control the handling of received streams. This architecture provides scalability and flexibility in media management, making it suitable for large-scale conferencing solutions. [3]

Mediasoup operates using the concept of `Transports`, which are responsible for carrying the media streams between clients and the SFU. In other words, a Transport is a p2p connection from the client to the server. There are two types of transport which are used, a producer and consumer transport. The producer transport carries the mediastream which the client creates to the SFU and the consumer transport is for sending media stream from the SFU to the client. Im other word every client has 2 p2p connections to the client one for receiving media stream, one for sending media streams. Additionally to the concept of transports, mediasoup uses the concept of **Producer** and **Consumer** objects.

**Producer** refers to objects that contain a producer ID in addition to the media stream. This enables the SFU to keep track of which media stream originates from which client. Instead of sending the media stream directly to the SFU, the producer object is sent through the producer transport. Additionally, each client generates a producer for each media stream that they want to send through the producer transportation.

For instance, should a client desire to transmit both audio and video, they would create a separate producer for each media stream. This ensures that each stream is individually tracked and managed by the SFU.

**Consumer** objects are generated by the SFU for each received Producer. The SFU creates a Consumer for each client that wishes to receive a specific media stream. Each Consumer object stores a unique ID along with the media stream, associating the Consumer with a specific client. Consequently, when a client transmits audio to the SFU, the SFU generates a consumer for each additional client in the meeting and transmits the audio through the consumer transport to the clients.

Furthermore, when a client joins a meeting and other clients are already present and have produced media, the SFU transmits all available media streams to the client. This way the SFU efficiently manage and distribute media streams to multiple clients.

### D. NAT Traversal

Network Address Translation (NAT) traversal is a crucial aspect of enabling peer-to-peer communication in WebRTC applications. NATs are commonly used in networks to allow multiple devices to share a single public IP address, which can complicate direct communication between clients. WebRTC employs two main techniques to overcome NAT-related issues: STUN (Session Traversal Utilities for NAT) and TURN (Traversal Using Relays around NAT) servers. Overcoming NAT is especially important when the client and the server want to establish (Mediasoup consumer and producer transport) p2p connections.

*1) STUN Server:* A STUN server helps clients discover their public IP address and the type of NAT they are behind. It facilitates the establishment of peer-to-peer connections by allowing the clients to determine how they can be reached over the internet.

**Functionality:**

- A client sends a Binding Request to a STUN server. This request typically includes information such as the client's private IP address and port
- The STUN server receives the Binding Request and responds with a Binding Response. This response includes the client's public IP address and port as seen by the STUN server
- Using the public IP address and port obtained from the STUN server, the client can communicate this information to other peers. This enables the peers to establish a direct connection
- While STUN is primarily used for discovering the external network address, it cannot guarantee connectivity through restrictive NATs and firewalls. For such cases, additional techniques like TURN are required

*2) TURN Server:* A TURN server provides a fallback mechanism when direct peer-to-peer communication is not possible due to restrictive NATs or firewalls. It relays media traffic between clients and a media server, ensuring that communication can occur even in challenging network conditions.

**Functionality:**

- The client sends an Allocate Request to the TURN server. This request asks the server to allocate a relay address and port for the client
- The TURN server responds with an Allocate Response, providing the client with a relay address and port. This address and port will be used to relay media traffic
- The client communicates its relay address to the media server. The media server will send its media streams to the TURN server, which then relays the media to the client
- Similarly, the client sends its media streams to the TURN server, which relays them to the media server
- The TURN server maintains an open connection between the client and the media server, ensuring reliable media delivery
- Throughout this process, the TURN server handles the necessary NAT traversal, making it possible for the media streams to pass through restrictive firewalls and NATs

### E. Coturn

Coturn is an open-source implementation of TURN and STUN servers, playing a critical role in facilitating real-time communications in applications by managing NAT traversal and providing relay services when direct peer-to-peer connections are not possible.

In the context of the proposed WebRTC multi-conferencing framework, Coturn is used to enhance connectivity and ensure seamless communication. Initially, the framework uses Coturn as a STUN server to enable clients to attempt direct peer-to-peer connections. If direct connections are not possible, the framework utilizes Coturn as a TURN server to relay media traffic, ensuring reliable communication. By integrating Coturn, the WebRTC multi-conferencing framework should allow clients in different networks to communicate. [4]

## III. Architecture

### A. General Architecture

The general architecture of the framework, the components involved in establishing a WebRTC connection between two clients, through a Google Cloud infrastructure, can be seen in the following figure 1.

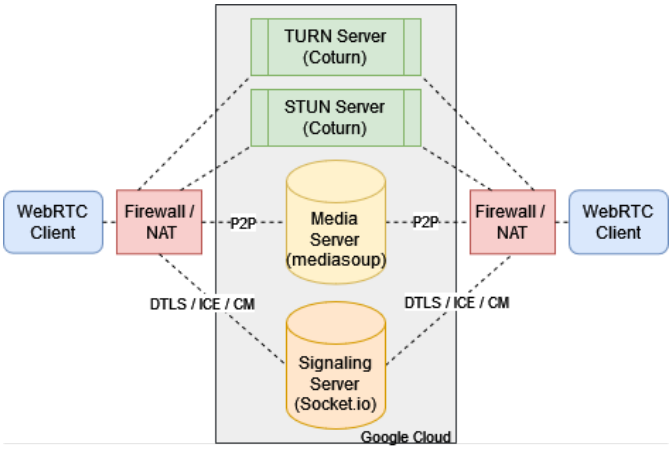


Fig. 1. General Architecture

The main components include WebRTC clients with Firewall, a TURN server, a STUN server, a Media server, and a Signaling server.

- **WebRTC Clients**: These are the end-user clients that joins a meeting room, through the internet
- **Firewall / NAT**: The Clients connections are typically behind a firewall or NAT, which requires traversal techniques to establish a direct peer-to-peer (P2P) connection
- **STUN Server (Coturn)**: The STUN server for helping the clients discover their public IP addresses
- **TURN Server (Coturn)**: The TURN server to relays the media between clients, in case STUN is not possible
- **Media Server (Mediasoup)**: The Media server to handle the media streams between the clients
- **Signaling Server (Socket.io)**: The Signaling server facilitates the exchange of messages needed to initiate, manage, and terminate the WebRTC connection
- **Communication Flow**:
  - `Control Messages (CM)`: Clients exchange control messages through the Signaling server to coordinate the WebRTC connection
  - `DTLS`: Datagram Transport Layer Security (DTLS) is used to encrypt the data exchanged between clients, ensuring secure communication and protecting the integrity and privacy of the transmitted media and control messages
  - `ICE`: The Interactive Connectivity Establishment (ICE) framework enables web browsers to establish connections with other peers. It employs STUN and/or TURN servers to achieve these objectives [5]

On a abstract level, the process of how the user joins a meeting can be described as following:

- A client browses to the website, which is hosted from the Signaling Server via Google Cloud
- The client creates a Meeting Room and receives a meeting link and a meeting code to access the created meeting room.
- A connection gets established from the client through the signaling server to the media server
- The media server and the client try to establish seperate p2p connection without need of signaling server
- When connection is established the User can share video, audio, text and screen

### B. Multi-conferencing functionality

Furthermore the framework provides the functionality for multiple clients from different networks to connect to the meeting room, which is demonstrates on the following figure 3.

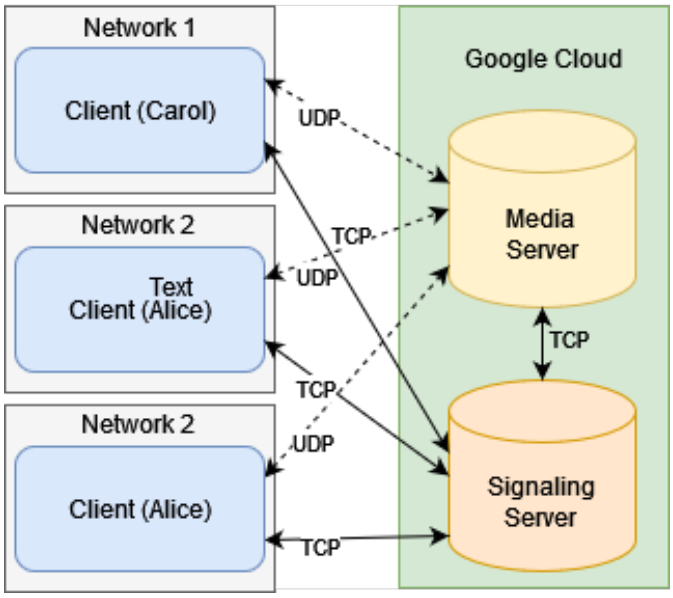


Fig. 2. Multi-conferencing Architecture

It is crucial to emphasise that the messages transmitted between the client and the Signaling server, and between the Signaling server and the Media server, are routed via TCP. This protocol ensures the reliable delivery of control messages and signaling data. In contrast, the client-to-Media server messages are transmitted using UDP. The use of UDP for media data transfer is advantageous as it allows for lower latency and more effective handling of packet loss, which is essential for real-time audio and video communication. [6]

## IV. Session Establishment

### A. DTLS Parameters

For secure communication between the Media server and the client, DTLS parameters are utilized. DTLS ensures that the data exchanged is encrypted and protected against tampering. Both the Media server and the client have their own sets of DTLS parameters, which include fingerprints for various hashing algorithms. These fingerprints are used to verify the integrity of the data being exchanged.

Below are the DTLS parameters for both the Media server and the client:

```
DTLS parameters: {
  fingerprints: [
    {
      algorithm: 'sha-1',
      value: 'E8:35:0D:E0:FB:D3:75:24:A4:E6:94:21:
      F6:6B:FB:AF:9A:BF:0E:06'
    },
    {
      algorithm: 'sha-512',
      value:'2C:0C:C0:25:67:03:C9:88:EC:5C:34:59:
      E7:85:16:33:43:A1:2A:3E:71:8E:5F:B0:37:BD:B0:
      BE:CD:72:B2:71:8B:B7:D9:CB:7B:95:6C:8B:7B:7E:
      D1:BD:B0:9D:12:09:5C:6D:56:49:2F:01:F2:B8:39:
      EC:75:DD:DC:FA:31:C4'
    },
    {
      algorithm: 'sha-224',
      value: '85:9F:4B:4C:D3:38:9F:DF:A3:0A:82:DB:
      95:15:00:3A:FB:2A:03:AD:84:18:08:DA:00:7E:EF'
    },
    {
      algorithm: 'sha-256',
      value: '2B:27:CD:F7:39:B6:67:26:49:F3:8C:2E:
      FB:F5:2D:45:94:81:19:C0:C4:34:AA:0A:54:BB:
      70:1E:02:A8:15:CF'
    },
    {
      algorithm: 'sha-384',
      value: 'C3:CC:46:00:64:71:21:68:13:AD:76:8F:
      35:A4:21:A2:FF:FA:90:20:5D:7C:B0:B0:41:1D:
      E7:B3:06:52:5E:75:52:A0:67:B8:ED:FE:5D:37:
      0E:78:D1:9E:B0:FB:71:86'
    }
  ],
  role: 'auto'
}
```

Listing 1. DTLS Parameter Mediaserver

```
DtlsParameters:
Object { role: "client", fingerprints: (1) [...] }
fingerprints: Array [ {...} ]
0: Object { algorithm: "sha-256", value: "4F:31:DE:
    DA:DB:4A:35:0A:FB:C1:45:18:AD:BD:E3:1B:54:E7:E2
    :91:07:7D:BC:28:80:38:2D:41:AF:45:A0:95" }
length: 1
role: "client"
```

Listing 2. DTLS Parameter Client

It is notable that the client only supports the `sha-256` algorithm. This is reflected in the DTLS parameters, as the client's fingerprint list includes only the `sha-256`algorithm. This indicates that the Media server must also support `sha-256` to establish a secure connection with the client. The use of multiple algorithms on the Media server ensures compatibility with various clients that may support different hashing algorithms, thereby enhancing the flexibility and robustness of the security framework.

### B. ICE

The ICE parameters and candidates establish a connection between peers are listed below.

```
ICE parameters: {
  usernameFragment: '1
    l8q1kan9bxshq7uu6xb6p34egmt6z43',
  password: '3zy0bq81ft7ano9swsztgftiwisn4hix',
  iceLite: true
}
ICE candidates: [
  {
    foundation: 'udpcandidate',
    priority: 1076302079,
    ip: '104.196.228.36',
    address: '104.196.228.36',
    protocol: 'udp',
    port: 20131,
    type: 'host',
    tcpType: undefined
  },
  {
    foundation: 'tcpcandidate',
    priority: 1076276479,
    ip: '104.196.228.36',
    address: '104.196.228.36',
    protocol: 'tcp',
    port: 20074,
```

```
    type: 'host',
    tcpType: 'passive'
  }
]
```

Listing 3. ICE Parameters and Candidates

*1) ICE Parameters:*

- **usernameFragment and password**: A unique identifier fragment used in conjunction with the password to authenticate the ICE session
- **iceLite**: The implementation does not perform the full connectivity checks, because the server are expected to always have a reachable IP address

*2) ICE Candidates:*

- **foundation**: Specifies between UDP and TCP candidate
- **priority**: The priority of the candidate. UDP candidate is preferred, since the priority is higher
- **ip and address**: Contains the IP address of the candidate
- **protocol**: The protocol used by the candidate (UDP/TCP)
- **port**: The port number used by the candidate
- **type**: The type of candidate. `Host` indicating it is a host candidate directly associated with the hosts network interface
- **tcpType**: For TCP candidates, specifies the type of TCP connection. In this case, the TCP candidate is `passive`, meaning it will wait for incoming TCP connections rather than initiating them

This setup allows ICE to determine the optimal path for data transmission, choosing between UDP and TCP candidates based on network conditions and connectivity options.

### C. Mediasoup Transport, Producer and Consumer

The following listing demonstrates the creation and handling of sessions as well as the handling of them the media server with 2 clients connected.

```
Map(1) {
  'fd243026-e925-4274-9d69-49e678ad9faf' => {
    transports: Map(4) {
      '2cRLCZ8-HXt1HPRvAAAo' => [WebRtcTransport],
      '2cRLCZ8-HXt1HPRvAAAo-consumer' => [WebRtcTransport],
      'xx7hvC0Otobn5V2mAAAs' => [WebRtcTransport],
      'xx7hvC0Otobn5V2mAAAs-consumer' => [WebRtcTransport]
    },
    producers: Map(4) {
      'c1763c38-0006-4e9e-9fe4-a58e3da67ff3' => [Producer],
      'bd100e66-5175-46d9-8e08-cf3f96a87f0e' => [Producer],
      '4283d66a-74ef-4ca9-911c-76a0b1a59fc6' => [Producer],
      '988099f8-4eaf-49e1-98a5-c5e330fe62f7' => [Producer]
    },
    consumers: Map(4) {
      'd5e31ca0-8b4f-4574-8c47-8deb45b86b89' => [Consumer],
      '8f0bddcd-5ff8-4b50-ac1e-980257ac83a4' => [Consumer],
      '1788f2c1-f11d-4b63-a7c6-cfd6bfd38084' => [Consumer],
      '6a62a60c-c529-4742-9bd8-aac69c7e1413' => [Consumer]
    },
    isScreenSharingActive: false
  }
}
```

Listing 4. Mediasoup Transport, Producer and Consumer

As illustrated in the listing, the media server maintains a record of all the transports (line 3 -7), consumers and producers of each room. Since the capture of the objects is made with 2 clients in the room, there is a total number of 4 transports stored, 1 producer and 1 consumer transport for each of them. Additional every client created 2 producer, which it has send to the media server, 1 for audio 1 for video, which makes a total of 4 producers (line 9-14). And the media server has created for each producer a consumer, so client A receives both type of media of client B and the other way around (line 15-18).

### D. Coturn

The Coturn configuration provided below outlines the setup for a TURN/STUN server. This server facilitates NAT traversal by helping clients discover their public IP address and relaying media when direct peer-to-peer communication is not possible.

```
# Listening IP of your TURN server
listening-ip=10.138.0.2

# External IP (public IP of the VM)
external-ip=104.196.228.36

# Listening ports for TURN server
listening-port=3478

# Additional TURN relay ports
min-port=20000
max-port=20200

# Authentication mechanism
lt-cred-mech

# Realm for authentication
realm=googlecloud

# Credentials for TURN server
user=....:....

# Relay IP (same as listening IP)
relay-ip=10.138.0.2
```

Listing 5. Coturn Configuration

The configuration comprises the internal IP address at which the TURN server is configured to listen for incoming requests (line 2), as well as the public IP address of the server, which clients are required to use to connect from outside the local network(line 2-5). It specifies the ports for TURN requests(line 8), as well as a range of ports for relaying media traffic (line 11-12). Additionally, the relay IP address is specified (line 24), as well as the user account, which all clients need to use to authenticate against the coturn server (line 15-21).

### E. Google Cloud Firewall

In order to facilitate external access to the virtual machine (VM) services, which are hosted via Google Cloud (signaling

server, coturn, media server), it is necessary to open certain ports in the firewall, to allow incoming traffic. The firewall rules which are needed are illustrated in the figure below.

| Name | Typ | Ziele | Filter | Protokolle/Ports | Aktion | Priorität |
|---|---|---|---|---|---|---|
| coturn-server | Eingehender Traffic | Auf alle | IP- | tcp:3478 udp:3478 | Zulassen | 1000 |
| media-server-traffic | Eingehender Traffic | Auf alle | IP- | udp:20000-20200 | Zulassen | 1000 |
| signaling-server | Eingehender Traffic | Auf alle | IP- | tcp:3000 | Zulassen | 1000 |

Fig. 3. Multi-conferencing Architecture

For coturn, as specified in listing 5, port 3478 has to opened. For the media server, ports are dynamically assigned. The range of UDP ports, which the media server uses are 20000-20200, so this range gets opened in the firewall settings. The last firewall rule is for the signaling server, where the client initially access the framework. Since the signaling server runs on port 3000, this is opened accordingly.

## V. Testing

### A. Scenario 1: Testing features on localhost

In the first scenario, all the features implemented in the framework are tested. The figure below shows the homepage of the framework, where the client can initially create or join a meeting.

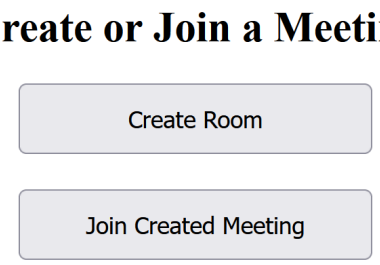


Fig. 4. Framework Homepage

*1) HTTPS:* The following listing shows the first feature, which was the ability to access the home page via HTTPS. This can be seen in the listing below.

```
https://localhost:3000/
```

Listing 6. URL Homepage

As can be seen, the signaling server is available via HTTPS.

*2) Rooms and Meeting Code:* Once the client created a meeting room, he receives a meeting code and gets redirected to the meeting room URL which can be seen below.

```
https://localhost:3000/meeting.html?room=08d8c6ad-ae80-4f33-9dc7-82d0b30f3e7a
```

Listing 7. URL Homepage

In the URL, the unique room uuid is shown. The client has now to enter the meeting code as well as his name so he can join the meeting, which is shown in the figure below.

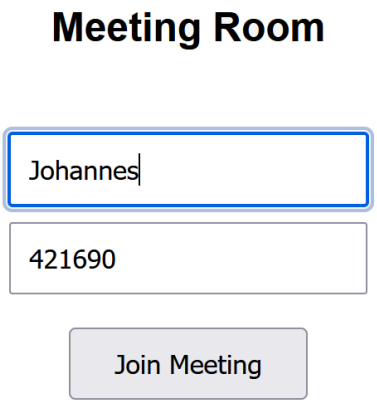


Fig. 5. Framework Homepage

*3) Media Sharing:* After the client entered the correct meeting code he joins the meeting room. The following console log of the meeting room on the client side shows the events happening.

```
1. Successfully joined the meeting:

2. User Johannes joined the room: 08d8c6ad-ae80-4f33-9dc7-82d0b30f3e7a

3. Local stream obtained:
    MediaStream { id: "{e10c4e20-cb28-4618-a7ac-e17773c1c74e}", active: true, onaddtrack: null, onremovetrack: null }

4. Creating Producer Transport
Producing video track:
    MediaStreamTrack { kind: "video", id: "{f312e3e0-bc12-4e8e-8db2-4aa82039593e}", label: "Logitech Webcam C925e", enabled: true, muted: false, onmute: null, onunmute: null, readyState: "live", onended: null }

5. Producing audio track:
    MediaStreamTrack { kind: "audio", id: "{5e1fde7d-eef2-4d3d-8b99-c56cf3a3d07e}", label: "Mikrofon (Logitech Webcam C925e)", enabled: true, muted: false, onmute: null, onunmute: null, readyState: "live", onended: null }

6.  Producing screen track:
    MediaStreamTrack { kind: "video", id: "{562e4703-205a-4299-878b-0d45cd689c94}", label: "", enabled: true, muted: false, onmute: null, onunmute: null, readyState: "live", onended: stopScreenSharing()
}

7. Client1: Hey
```

Listing 8. Conole Log Meeting Room

Once the client has joined the meeting, the framework requests permission to capture audio and video from the client. Should the client grant this permission, a local media stream will be obtained, which is a collection of media tracks (both audio and video).

Subsequently, the framework generates a Producer Transport for the purpose of transmitting the media streams

to the Media Server. This entails the generation of both video and audio tracks on the client side in the form of a `MediaStreamTrack`. A `MediaStreamTrack` represents a single media track, and thus, one `MediaStreamTrack` is created for each media source, which includes both audio and video.

Furthermore, a client is able to share their screen, which produces the sixth entry in the listing. Once again, a `MediaStreamTrack` is created, this time containing the captured screen content. The final feature tested was text chat. When a different client enters a command in the text chat, it is displayed in the group chat and logged on each client's side, as seen in the seventh entry.

### *B. Scenario 2: Testing NAT with Google Cloud*

Scenario 2 is about testing the accessability of the framework over google cloud as well as the function of the configured Coturn server. To accomplish that first the figure below shows the IP address of the google cloud instance, which is `104.196.228.36`. The IP will be refered to as media server, signaling server or Coturn server below.


Fig. 6. Framework Homepage

To determine the functionality of the framework (media server, signaling server and Coturn server), two clients from different networks connect to the framework and the traffic gets captured by wireshark.

*1) TLS Handshake:* The Wireshark capture shown in the figure below represents the TLS handshake between the client and the signaling server.

| Destination | Protocol | Lengt | Info |
|---|---|---|---|
| 104.196.228.36 | TLSv1.3 | 571 | Client Hello |
| 192.168.177.32 | TLSv1.3 | 1474 | Server Hello, Change Cipher Spec, Application Data |
| 192.168.177.32 | TLSv1.3 | 926 | Application Data, Application Data, Application Data |
| 104.196.228.36 | TLSv1.3 | 134 | Change Cipher Spec, Application Data |
| 192.168.177.32 | TLSv1.3 | 628 | Application Data, Application Data |
| 104.196.228.36 | TLSv1.3 | 78 | Application Data |

Fig. 7. DTLS Handshake

- The sequence starts with a `Client Hello` message from the client to the server, initiating the handshake.
- The `Server Hello` message from the server to the client follows, indicating the server's acceptance and readiness to proceed.
- The `Change Cipher Spec` message is transmitted first from the signaling server to the client and subsequently from the client to the signaling server, as can be observed from the destination address, indicating that both the client and server are engaging in encrypted communication.
- The `Application Data` messages are exchanged between the client and server, representing the encrypted data communication.

*2) DTLS Handshake between Client and Media server:* The Wireshark capture shown in the figure below represents the DTLS handshake between the client and the media server.

| Protocol | Lengt | Info |
|---|---|---|
| DTLSv1.2 | 306 | Client Hello |
| DTLSv1.2 | 306 | Client Hello |
| DTLSv1.2 | 743 | Server Hello, Certificate, Server Key Exchange, Certificate Request, Server Hello Done |
| DTLSv1.2 | 627 | Certificate, Client Key Exchange, Certificate Verify, Change Cipher Spec, Encrypted Handshake Message |
| DTLSv1.2 | 627 | Certificate, Client Key Exchange, Certificate Verify, Change Cipher Spec, Encrypted Handshake Message |
| DTLSv1.2 | 133 | Server Hello |
| DTLSv1.2 | 418 | Certificate |
| DTLSv1.2 | 178 | Server Key Exchange |
| DTLSv1.2 | 115 | Certificate Request |
| DTLSv1.2 | 67 | Server Hello Done |
| DTLSv1.2 | 627 | Certificate, Client Key Exchange, Certificate Verify, Change Cipher Spec, Encrypted Handshake Message |
| DTLSv1.2 | 109 | Change Cipher Spec, Encrypted Handshake Message |
| DTLSv1.2 | 60 | Change Cipher Spec |
| DTLSv1.2 | 95 | Encrypted Handshake Message |
| DTLSv1.2 | 60 | Change Cipher Spec |
| DTLSv1.2 | 95 | Encrypted Handshake Message |

Fig. 8. DTLS Handshake

- The sequence starts with a `Client Hello` message from the Client to the media server, initiating the handshake
- The `Server Hello` message from the media server to client follows, along with the server's certificate
- The client and server exchange key information through `Certificate`, `Client Key Exchange`, and `Server Key Exchange` messages
- The `Change Cipher Spec` messages indicate that both parties are switching to encrypted communication
- The final message, `Encrypted Handshake Message`, confirms that the handshake is complete and a secure connection has been established

In the figure, two `Client Hello` messages are displayed, as the client generates both a producer and consumer transport following their joining of the meeting. Consequently, each of the transports necessitates DTLS encryption, resulting in the presentation of two preliminary `Client Hello` messages, followed by the corresponding DTLS handshake message exchanges.

*3) Coturn NAT Traversal:* The STUN Binding Request message is sent from the client to the Coturn STUN server and can be seen in figure 9. This message initiates the process to determine the client's public IP address and port.

Key attributes in this message include the message type (‘0x0001‘ for Binding Request), message transaction ID as well as the encrypted username.

The STUN Binding Success Response message is sent by the Coturn STUN server at to the client and can be seen in figure 10. This response provides the public IP address and port assigned to the client, which are marked with a black line.

The Key attributes in this message include the message type (‘0x0101‘ for Binding Success Response ) and the with the request matching message transaction ID.

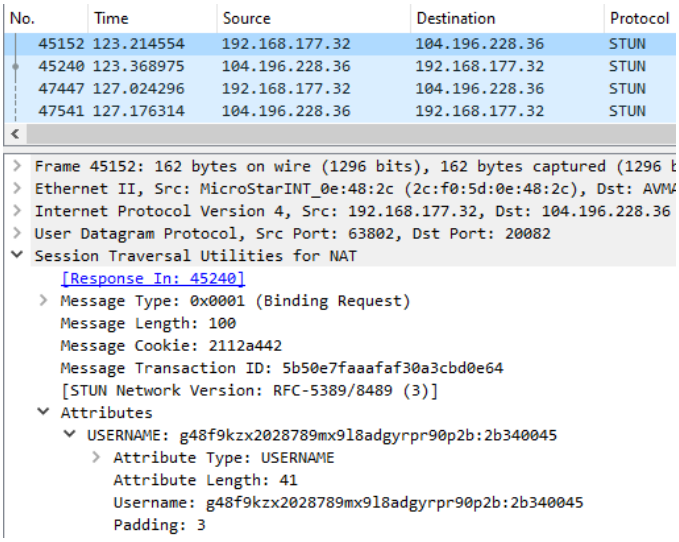


Fig. 9. STUN Binding Request

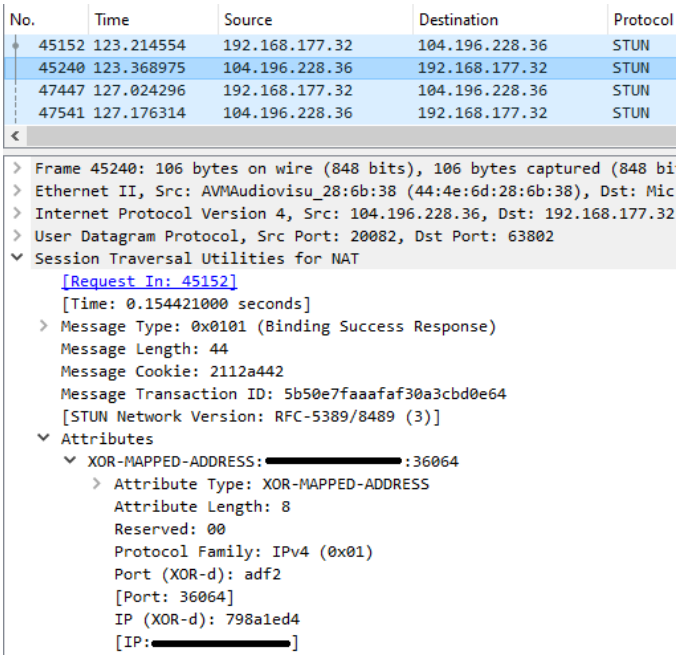


Fig. 10. STUN Binding Response

*4) Media Traffic:* The Wireshark shown on figure 13 capture shows the media traffic between the Media server and the client.

In this capture, the Media server is sending media packets to the client, and the client is also sending media packets back to the Media server. This bidirectional traffic indicates that there are at least two users in the meeting room, exchanging media data.

Additionally, the data lengths of the packets vary, which suggests the transmission of different types of media, namely audio and video. Typically, video packets are larger due to the higher data rate required for video content, while audio packets are smaller.

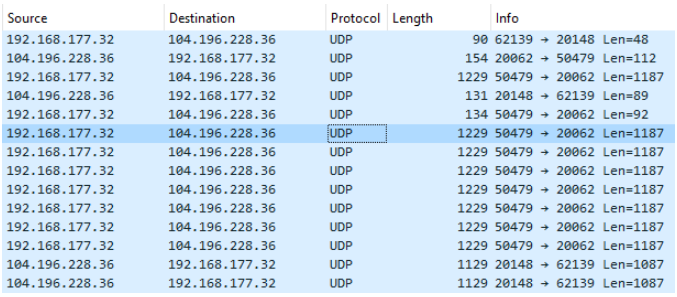


Fig. 11. Media UDP Traffic

The difference in lengths of the packets can also be seen in the RTP packages, which can be seen in the figures below.

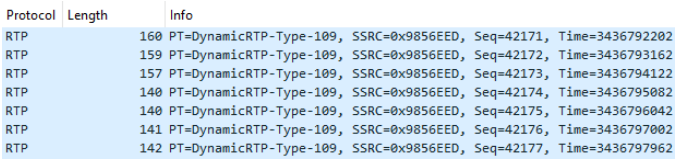


Fig. 12. Media UDP Traffic

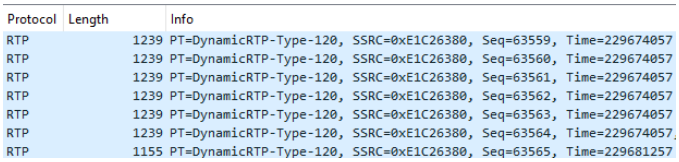


Fig. 13. Media UDP Traffic

RTP messages provide more detailed information about UDP traffic. However, in order to capture RTP traffic as RTP and not just as UDP in Wireshark, it is necessary to enable RTP because the RTP protocol is encapsulated within UDP media traffic, which is shown in Figure 1.

In the figures 11-12, it can be observed that they have a dynamic RTP type. The dynamic RTP type 109 corresponds to UDP traffic with smaller package sizes (audio), while type 120 corresponds to larger package sizes (video). Dynamic RTP types are identifiers used for RTP payloads that do not have a static, predefined type. These types are dynamically assigned during the session setup, typically via signalling protocols, to support various media codecs that are not covered by the standard static payload types.

Furthermore, Mediasoup initially utilises SRTP (Secure Real-time Transport Protocol). By specifying the pre-master secret from the DTLS session in Wireshark, it is possible to decrypt the SRTP traffic. This decryption allows RTP to be displayed as the protocol in the displayed figures.

## VI. Unsolved Challenges

One significant unsolved challenge is related to Chrome's handling of the audio context. When a user joins a website

using Chrome, the browser blocks the audio context by default, requiring user interaction to resume it. This behavior is intended as a security measure to prevent unwanted audio from playing automatically. However, even when the user interacts with the website and the audio context is resumed, there are instances where the audio is not captured properly. This results in the audio streams not being captured properly.

## VII. Conclusion

In conclusion, this paper presents a comprehensive framework for multi-conferencing using WebRTC technology, leveraging Mediasoup as SFU to ensure efficient media stream management. This paper has explored the benefits and challenges of various connection approaches, with the SFU ultimately selected for its scalability and client-side media processing capabilities. The framework's key features include support for audio, video, screen sharing, and text communication, along with robust security measures such as meeting codes and HTTPS hosting.

## VIII. Future Work

Despite the progress that has been made, there is still the unresolved challenge of handling media streams in different browsers. This is a task that will require further examination in future work.

Further future work contains the following areas:

### *A. Video Container Management*

One of the primary areas for improvement is video container management. The objective is to develop a more flexible video container that can dynamically adjust its layout. In particular, when screen sharing is active, the video containers will be resized to maintain uniformity among all video streams, ensuring that all videos are displayed equally and efficiently.

### *B. Audio Stream Handling on Client Side*

Another crucial area for future research is the enhancement of audio stream handling on the client side. The current implementations require enhancements in features such as volume control and muting functionalities.

### *C. Testing on Coturn TURN functionality*

As the connection with the tested clients functioned with STUN, the TURN functionality of Coturn was not tested, and therefore must be examined in future work.

## References


[1] "WebRTC," WebRTC. https://webrtc.org/?hl=de

[2] "SFU, MCU, or P2P: What's the Difference Between These WebRTC Architectures?," getstream.io. https://getstream.io/blog/what-is-a-selective-forwarding-unit-in-webrtc/

[3] "mediasoup," mediasoup.org. https://mediasoup.org/

[4] "coturn/coturn," GitHub, Jul. 03, 2024. https://github.com/coturn/coturn (accessed Jul. 03, 2024).

[5] "Introduction to WebRTC protocols - Web APIs — MDN," developer.mozilla.org. https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Protocols

[6] "TCP vs. UDP: Optimising Video Streaming Performance," Gumlet, Dec. 08, 2023. https://www.gumlet.com/learn/tcp-vs-udp/