People's Democratic Republic of Algeria
Ministry of Higher
Education and Scientific Research

IBN KHALDOUN
UNIVERSITY OF TIARET

Dissertation

Presented to:

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
DEPARTEMENT OF COMPUTER SCIENCE

in order to obtain the degree of:

MASTER

Specialty:

Software engineering

Presented by:

REFIK Youcef Abdelillah

On the theme:

Implementation of a private messaging service Peer-to-Peer using WebRTC technology

Defended publicly on / / 2025 in Tiaret in front the jury composed of: :

Mr Kharroubi Sahraoui	Tiaret University	Chairman
Mr Dahmani Youcef	Tiaret University	Supervisor
Mr Mokhtari Ahmed	Tiaret University	Examiner

2024-2025

Acknowledgments

I extend my sincere thanks and deep gratitude to the esteemed professor, Mr. **Dahmani Youcef**, who graciously entrusted me with supervising this dissertation. He was a beacon of knowledge and patience and a guiding light throughout the research process.

He spared no effort or guidance, and his valuable comments greatly contributed to shaping this work. May Allah reward him on my behalf with the best of rewards, elevate his status in this life and the hereafter, and place his contributions in the scale of his good deeds.

I also extend my heartfelt thanks to everyone who supported me through guidance or moral encouragement throughout my academic journey.

To all those who offered a kind word or uplifting encouragement, I say:

"May Allah protect you and bless your efforts."

All praise is due to Allah, by whose grace good deeds are completed.

Contents

Chapter 4

System Implementation.

Abstract:

This thesis presents the design and implementation of a real-time, peer-to-peer communication service using Web Real-Time Communication (WebRTC) technology. WebRTC is a powerful open-source framework that enables direct audio, video, and data exchange between web browsers and mobile applications without requiring external plugins. With the growing demand for decentralized and low-latency communication systems, WebRTC offers a modern solution to build secure and scalable applications.

The project aims to explore the architectural components and protocols involved in establishing a peer-to-peer connection, including signaling, Session Description Protocol (SDP), Interactive Connectivity Establishment (ICE), NAT traversal using STUN and TURN servers, and the use of DTLS-SRTP for encrypted media transmission. A custom web application was developed with a frontend built in Vanilla JavaScript and a backend implemented using ASP.NET Core, incorporating a SignalR-based signaling server and a JWT-based authentication mechanism.

The system supports real-time audio/video calls and screen sharing between authenticated users, relying solely on peer-to-peer media exchange. The implementation highlights the challenges of NAT traversal, user discovery, session management, and media stream handling, while offering a lightweight and responsive user experience.

This work contributes to the field by providing a practical and extensible example of how WebRTC can be integrated into web-based communication services. The results demonstrate the effectiveness of WebRTC in building decentralized communication platforms, with minimal server load and strong end-to-end security.

Chapter 1

Project Foundation and Methodology

Chapter 1 Project Foundation and Methodology

Introduction :

This chapter provides a background of video conferencing, why it became so important and how the technology evolved. After this short background, the new emerging standard for making video conferencing solutions is discussed: WebRTC and the relevance of this research is described. This is followed by the goals and approach of this thesis and concluded by an outline of what is discussed in the remainder of this thesis.

Background

Communication has evolved greatly in the last century; from post letters, to communication via the telephone, to Voice Over Internet Protocol (VOIP) and now Video Conferencing (VC). Video Conferencing is a conference between two or more different users using telecommunications to transmit audio and video data. Due to companies expanding internationally there has been a need to bridge communication distances without traveling. Video conferencing has evolved rapidly in the last decades and is getting more accessible to users every day. This is due to the emerging computer and smartphone industry with continuously improving capabilities and also due to better network connectivity of (mobile) devices with a built-in camera. Video conferencing is widely supported and available through many platforms and applications (e.g. Skype, Google Hangouts, Apple Facetime). Video conferencing either works in a Server-to-Client (S/C) manner, where the video and audio streams get channeled via a server or Peer-to-Peer (P2P), where video and audio directly get exchanged between peers without the extra overhead of a centralized server. A P2P browser protocol that has been getting attention lately and which this thesis will be focusing on, is Web Real-Time Communication (WebRTC). This is an open-source network protocol for the browser set up by World Wide Web Consortium (W3C) and the WebRTC group from the Internet Engineering Task Force (IETF) and allows to transmit video, audio and data between browsers in P2P without the need to install additional software or plugins.

Motivation

Before WebRTC was introduced, real-time communications were only available to bigger companies or via extra browser plugins like Adobe Flash. WebRTC makes it really easy for application developers to develop their own video chatting applications because it exposes a very high-level Application Programing Interface (API). This makes it really approachable for developers, without the need to understand how the underlying protocols work.

In this thesis, a real time connection between peers using WEBRTC is created, and its video chatting capabilities is going to be analyzed, and compared with the client/server's protocols to get a better grasp on how WebRTC performs. The main goal of this thesis is establishing a service of communication using this technology and providing the full documentation of the implementation.

Goals of the Project

The main research question for this thesis is:

"How to implement the WebRTC protocol in a web-based application and delivering a complete documentation of the process?"

This main question is answered through these sub questions:

Q1 What is WebRTC and what architecture it uses?
Q2 What are its main components?
Q3 What the difference between peer-to-peer and Client/Server architectures?
Q4 How does WebRTC work under the hood?
Q5 How to implement the WebRTC technology?
Q6 What are the advantages that WebRTC did bring and the problems it solved?

Method

The Method to resolve the issues discussed in section 1-3 is detailed below:

1. First the WebRTC protocol is described, and the peer-to-peer architecture is explained.
2. WebRTC is then described in more details, covering its different components.
3. The underlying protocols of WebRTC are mentioned and how they work with each other is explained.
4. An experimental implementation of WebRTC is done with a step-by-step guide and explanation.
5. The advantages and performance of WebRTC and potential downsides are discussed.

Outline

An in-depth analysis of WebRTC is given in chapter 2 where the requirements are discussed, how WebRTC's underlying protocols work is explained, and how to setup a WebRTC call is shown in details. In Chapter 3, a thorough description of the web application's design is provided, accompanied by code snippets that guide the reader through the implementation steps, thereby concretizing many of the technical concepts introduced earlier.

Chapter 2

Theoretical background

2.1 Introduction to WebRTC

2.1.1 What is WebRTC?

Web Real-Time Communication (WebRTC) is a free, open-source project that enables real-time communication of video, audio, and data between peers directly in web browsers without requiring additional plugins or software. It is designed to facilitate peer-to-peer (P2P) connections, allowing two browsers to communicate directly with each other in peer-to-peer (figure 2-1b), which is unlike most browser communication, which flows through a server (figure 2-1a), allowing for low-latency and high-performance communication over the internet. [1]

WebRTC was first introduced and made open source by Google in May 2011 [2], but not available for the masses till late 2012 in the most recent version of Google Chrome.

Figure 2-1: Difference between Server to Client and Peer to Peer communication

Firstly, a comparison between peer-to-peer and server-to-client architectures should be made in order to understand WebRTC technology very well.

2.2 Peer Connection vs. Client-Server Communication

2.2.1 Peer Connection

Peer connection refers to a direct communication link between two devices over the internet without relying on a central server for data transfer once the connection is established. WebRTC uses a combination of technologies such as ICE (Interactive Connectivity Establishment), STUN (Session Traversal Utilities for NAT), and TURN (Traversal Using Relays around NAT) to overcome network limitations and establish a stable peer-to-peer connection. [3]

2.2.3 Client-Server Communication

Traditional client-server communication involves a centralized server that facilitates data transfer between clients. This architecture is commonly used in applications such as HTTP-based websites and cloud-based services**.**

2.3 When a Peer Connection is The Better Solution?

As demonstrated in (Table 2-1), Peer-to-peer connections are preferable when low latency and direct data exchange are crucial.

Criteria	Peer-to-Peer (WebRTC)	Client-Server
Latency	Ultra-low (direct path). Typically, <100ms for media streams.	Higher due to server hops. Often 200–500ms in VoIP systems.
Scalability	Limited by NAT traversal overhead. Each peer maintains O(n) connections in mesh topologies.	Centralized scaling (O(1) per client). Cloud servers handle load balancing.
NAT/Firewall Traversal	Requires ICE/STUN/TURN. Fails under symmetric NATs without TURN relays.	No NAT issues (clients connect outward to the server).
Privacy	End-to-end encrypted (DTLS-SRTP). No intermediary data storage.	Server decrypts/inspects traffic (e.g., TLS termination).
Server Load	Minimal (only signaling). Media/data bypasses servers.	High (server processes all traffic). Bandwidth costs grow linearly.
Use Case Fit	Video conferencing (e.g., GoogleMeet), IoT (low-latency). Struggles with >10 peers.	Mass broadcasting (e.g., Netflix), centralized apps.
Hybrid Potential	TURN relays introduce client-server dependencies.	Edge computing can reduce latency (e.g., CDNs).

Table 2-1: A comparison between peer-to-peer and client/server architectures.

2.4 WebRTC from a developer perspective

To set up a WebRTC application, a connection must first be established between browsers. Since WebRTC operates as a peer-to-peer communication technology, it does not provide a built-in signaling mechanism. Instead, developers can choose any method to exchange signaling data—such as an HTTP server, WebSocket, a tweet or even with a pigeon like old times. This exchange process, known as signaling, involves sending session-related information from one client to another to initiate the peer-to-peer connection. This information is conveyed using the Session Description Protocol (SDP), typically implemented in JavaScript (see Section 2.5.2), which describes the supported media formats and capabilities of each endpoint.

One of the most crucial pieces of information exchanged during this process is the client's public IP address. In order for one browser to receive a connection from another, each must know the other's public IP. However, this can be challenging because clients are often behind firewalls or Network Address Translators (NATs) (see Section 2.5.1), which obscure their public IP addresses. To address this, a STUN server (see Section 2.5.3) is commonly used to help clients discover their public IP and identify the type of NAT they are using. In some cases, particularly with Symmetric NAT, obtaining this information becomes more difficult. When this occurs, a TURN server (see Section 2.5.4) is used as a fallback solution. Without

delving into the implementation details—since this lies outside the scope of the thesis—it is important to note that if a STUN server fails to retrieve the client's public IP, a TURN server can relay media traffic between peers.
After the SDP messages are exchanged, the peers begin sharing ICE candidates (see Section 2.5.5). These candidates provide the network details required to determine the most efficient route for a direct connection.
In summary, the workflow begins with the client discovering its public IP using a STUN or TURN server. The client then generates an SDP offer and sends it to a peer via an external signaling method. If the offer is accepted, the peer replies with an SDP response. Following this exchange, both peers share ICE candidates to establish the best possible network path for the peer-to-peer connection.

Figures 2-2 explain the architecture of WebRTC in detail.

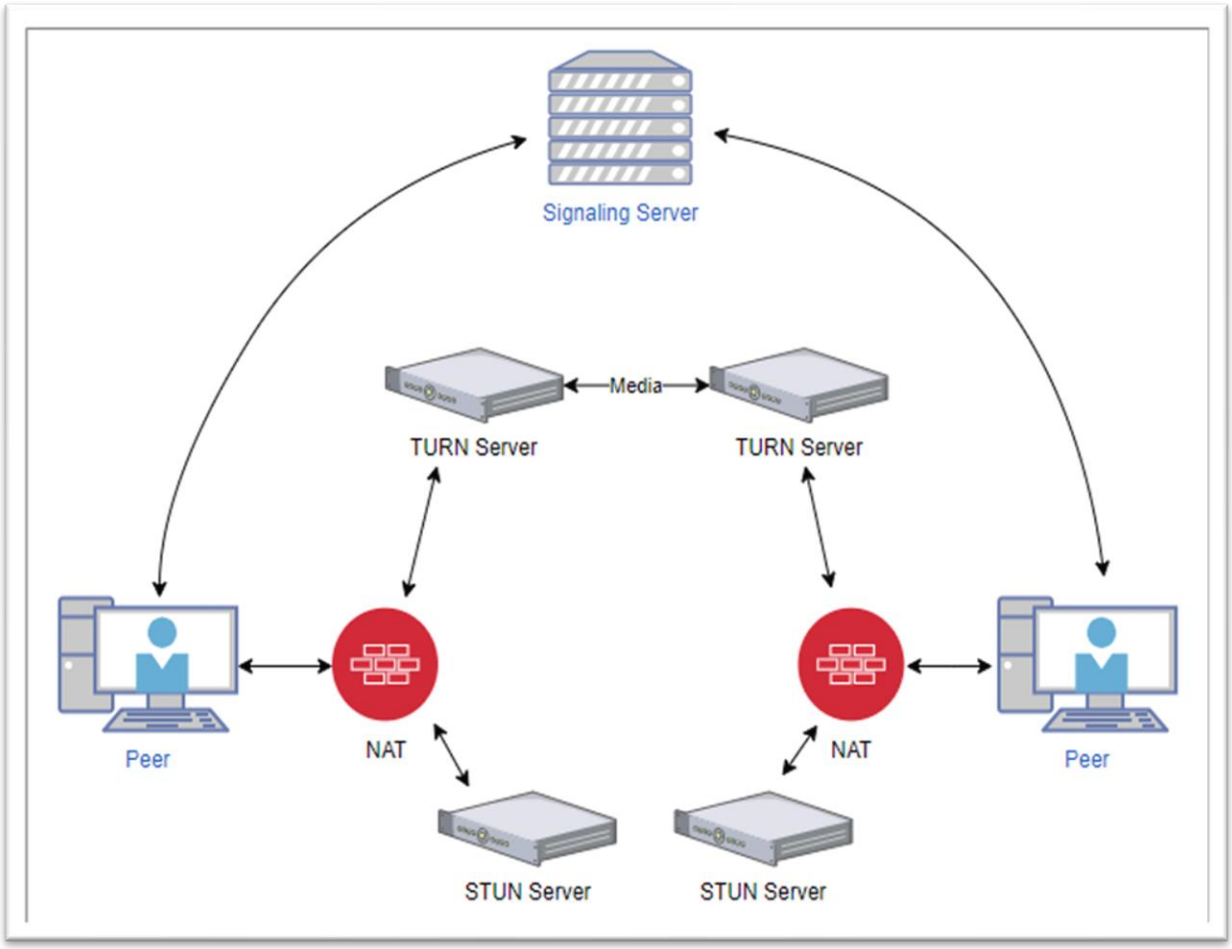

Figure 2-2: A WebRTC architecture with a signaling server, NAT, STUN server, and TURN server.

2.5 WebRTC Related Technologies

The WebRTC standard is built on several different existing technologies and standards to allow users to establish reliable and secure peer-to-peer connections to exchange live media. These are primarily the Session Description Protocol (SDP), Interactive Connectivity Establishment

(ICE), Session Traversal Utilities for NAT (STUN), and Traversal Using Relay NAT (TURN). The WebRTC standard specifies how and when to use the technologies mentioned above and how they relate to the API. WebRTC also requires the use of a signaling service to exchange information between the peers that wish to establish a connection, however, this is intentionally left out of the standard to allow for developers to use whichever method and technologies they prefer.
This allows for additional functionality to be moved to the signaling service such as authentication and keeping track of the available STUN and TURN servers the peers can use. What follows is a description of relevant technologies that is required to understand WebRTC and the underlying problem that merits implementing WebRTC, and finally how they come together to actually solve this problem.

2.5.1 Network Address Translation (NAT)

NAT is a router function that modifies the network address in an IP header and is often used to route traffic from local networks to the public network. A NAT is installed at the gateway to the public backbone network, where all addresses are globally unique. A translation matrix is stored by the NAT, linking internal IP addresses to an external one. Since translation happens on the gateway where that NAT is configured, the end points that are located behind this NAT do not know which address they can be reached with from the public network. This becomes a problem for peer-to-peer applications where both peers need to find out the address of the other peer to be able to start communication. A NAT that is configured as dynamic has a pool of public IP addresses but can reassign the mapping it has between local IP addresses and public IP addresses. If the NAT cannot route traffic for an internal host because it currently has no public IP addresses available, it responds with an ICMP "Destination Unreachable" message. When a public IP address that has been mapped to a local IP address has not been used for a timeout duration, the NAT deletes this mapping, allowing the public IP address to be reassigned to another device on the local network. The standard and most frequently implemented configuration is Network Address/Port Translator (NAPT), or Port Address Translator (PAT). Using this configuration, multiple local IP addresses can use the same public IP address simultaneously by utilizing different ports. This is achieved by mapping the local IP address that wants to initiate communication to a public IP address and port. In summary, there are two ways this mapping can be done.

NAT behaviors were originally defined in RFC 4787 [12], and the description is derived from that document.

2.5.2 Session Description Protocol (SDP)

The Session Description Protocol (SDP) [9] is used by WebRTC to exchange information regarding the session that the peers wish to establish. This session is negotiated between the peers to try to establish the various forms of media, encodings, transport protocols, and more. The anatomy of an SDP message consists of several lines of text, where the type consists of a single case-sensitive character and the value is text, where the structure is dependent on the type. Each SDP message has several required and optional type-value pairs. The required types are the following:

• v: The protocol version used in the SDP message.

• o: The originator and the identifier of the session.

• s: The session's name.

• t: The time the session should be active, specified as a start and stop time.

In WebRTC, the SDP is used in conjunction with the Session Initiation Protocol (SIP) [15] using the Offer-Answer model. [16] The Offer-Answer model entails that the peers who wish to establish a session have one peer (initiator) who generates an offer describing the session they wish to establish, the receiving peer (respondent) receives the offer and generates their answer, and sends it to the initiator

Figure 2.4 illustrates a sample SDP offer.

```
v=0
o=- 4385423089851900022 0 IN IP4 0.0.0.0
s=-
t=0 0
a=ice-options:trickle
a=group:BUNDLE video0 application1
m=video 9 UDP/TLS/RTP/SAVPF 96
c=IN IP4 0.0.0.0
a=setup:actpass
a=ice-ufrag:lsJx+7d6hsCyL8K6m8/KbgcqMqizaZqy
a=ice-pwd:zFUTJmx6hNnr/JRAq2b3w0tmm88XERb3
a=rtcp-mux
a=rtcp-rsize
a=sendrecv
a=rtpmap:96 H264/90000
a=rtcp-fb:96 nack pli
a=framerate:30
a=fmtp:96 packetization-mode=1;profile-level-id=42E01F;sprop-parameter-sets=Z00AKeKQDwBE/LNwEBAaUABt3QAZv8wA8SIq,a048gA==
a=ssrc:3776670536 msid:user3344942761@host-c94b5db webrtctransceiver11
a=ssrc:3776670536 cname:user3344942761@host-c94b5db
a=mid:video0
a=fingerprint:sha-256 AE:1C:59:19:00:7B:C2:1C:85:95:0C:6C:8C:14:E8:67:A4:7D:D0:AE:90:5D:8F:BB:D7:5B:95:49:03:6E:94:8F
m=application 0 UDP/DTLS/SCTP webrtc-datachannel
c=IN IP4 0.0.0.0
a=setup:actpass
a=ice-ufrag:lsJx+7d6hsCyL8K6m8/KbgcqMqizaZqy
a=ice-pwd:zFUTJmx6hNnr/JRAq2b3w0tmm88XERb3
a=bundle-only
a=mid:application1
a=sctp-port:5000
a=fingerprint:sha-256 AE:1C:59:19:00:7B:C2:1C:85:95:0C:6C:8C:14:E8:67:A4:7D:D0:AE:90:5D:8F:BB:D7:5B:95:49:03:6E:94:8F
```

Figure 2.4: An example WebRTC SDP Offer Message

2.5.3 Session Traversal Utilities for NAT (STUN)

STUN enables an endpoint to determine the public-facing IP and port that the NAT is using as a mapping for the endpoint's actual IP and port. Furthermore, it is often used to check connectivity between endpoints, and it is also used as a keep-alive to maintain the bindings that a NAT has created to ensure that a binding stays alive as long as needed without the need for re-establishing the connection The STUN protocol functions via binding request and response

messages, where the request asks the server to return a binding response. When the request passes a NAT, the NAT creates a mapping and forwards the packet with the mapping as the source of the packet to the STUN server. The binding response from the STUN server includes an attribute called "mapped-address", and the server sets the "mapped address" attribute as the source of the incoming packet. When the originator receives the STUN binding response, it learns the address that the NAT has mapped its local address to through reading the "mapped-address" attribute. Keep-alive is practical because dynamic NATs will remove a mapping as soon as they think it is not being used, so it can be reused by another connection. Therefore, a STUN server must make sure that this mapping is not removed until connectivity checks are completed. A simple diagram describing the STUN binding request and response can be seen in Figure 2.5.

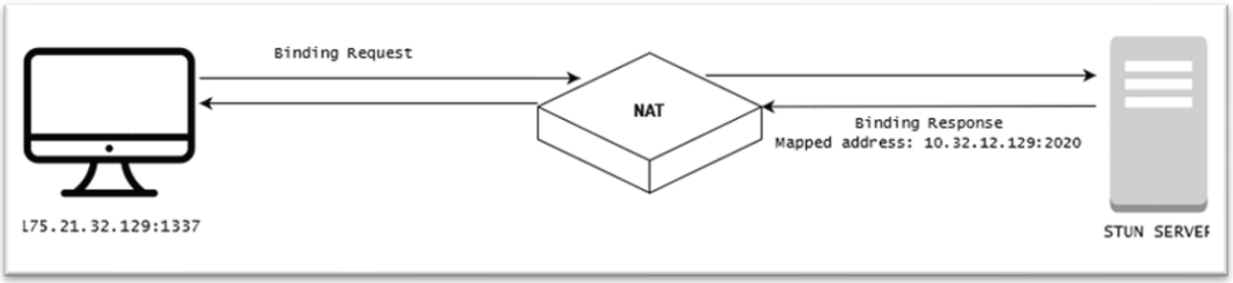

Figure 2.5: An example of the STUN binding request and response**.**

STUN is run using UDP, however it can also be implemented with the Transmission Control Protocol (TCP) and Transport Layer Security (TLS), which allows for additional security to be implemented.

2.5.4 Traversal Using Relay around NAT (TURN)

A TURN (Traversal Using Relays around NAT) server is a network relay mechanism defined in RFC 5766 (and updated by subsequent RFCs) to facilitate communication between devices behind restrictive NATs (e.g., symmetric NATs) or firewalls. It acts as an intermediary when direct peer-to-peer (P2P) connections are impossible, ensuring connectivity for real-time applications like VoIP, WebRTC, and video conferencing. [17]

Some network conditions require a TURN server instead of STUN, like Symmetric NATs, when some NAT types assign different public IPs and ports for each connection, preventing direct peer communication, or firewall restrictions: Corporate or institutional firewalls may block direct peer connections, necessitating a relay server.

Figure 2.6 details how the addresses are allocated using the TURN server.

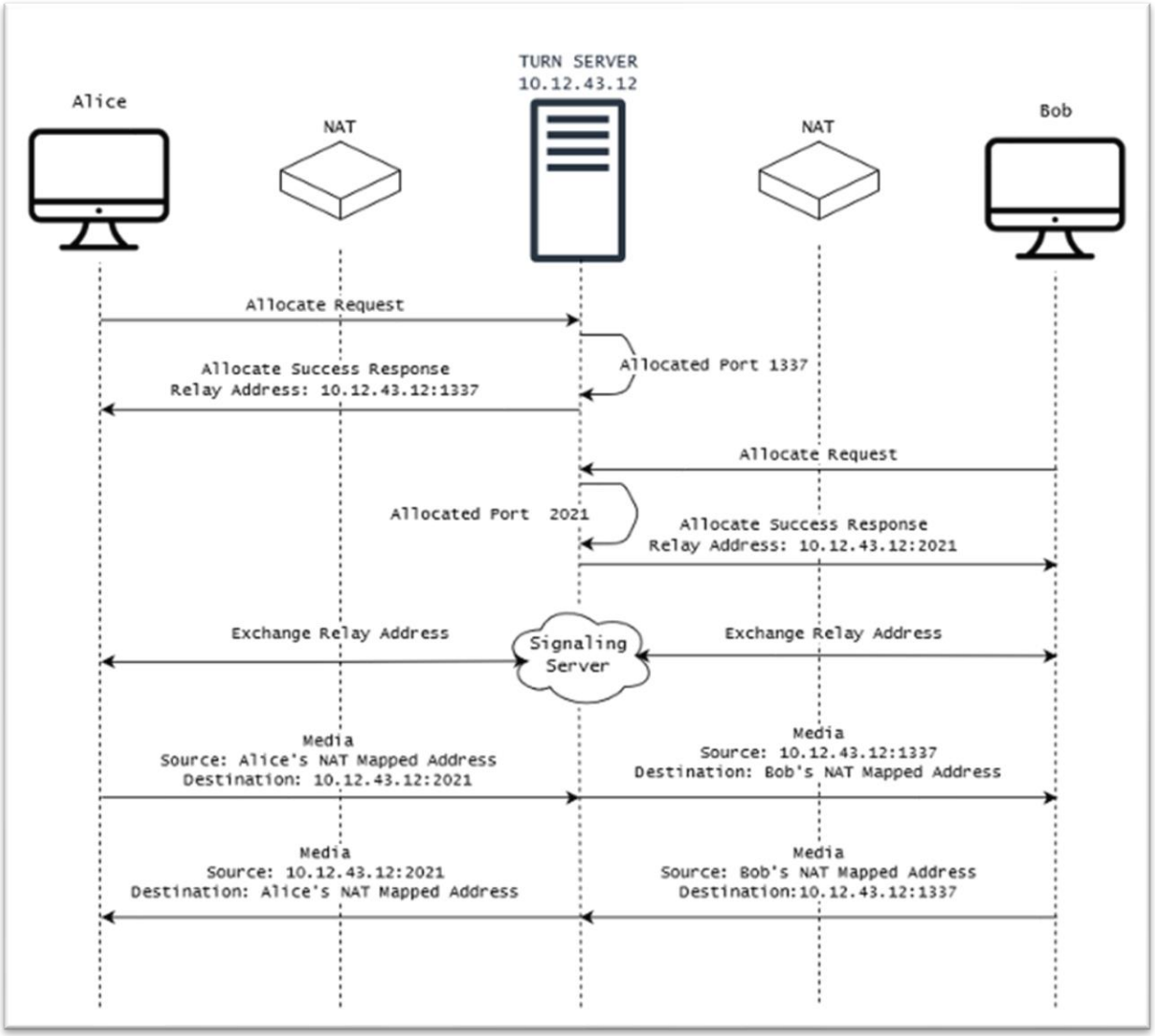

Figure 2.6: Allocating addresses in a TURN server and relaying media through it.

2.5.5 Interactive Connectivity Establishment (ICE)

Interactive Connectivity Establishment (ICE) is a framework used to establish a connection by selecting the best possible network path between peers [18]. It is a technique used in computer networking to find ways for two computers to talk to each other as directly as possible in peer-to-peer networking.

```
candidate:2116 1 udp 659136 10.85.129.66 65396 typ host generation 0
```

Figure 2.7: Example ICE Candidate, with the type host with a priority of 659136.

2.6 WebRTC Main Components

2.6.1 MediaStream API

The MediaStream API in WebRTC handles the acquisition of media content, such as audio and video, from a user's device. For example, audio is captured via the microphone, and video is sourced from the camera. MediaStream manages both the input (media collected from local hardware) and the output, which refers to transmitting the stream to remote peers. A common way to initiate a media stream in the browser is by using the getUserMedia() method. This API prompts the user for permission to access their camera and microphone. [4]
If access is granted, the browser can retrieve audio, video, or screen-sharing streams, represented by a positive response (e.g., a return value of 1). If the user denies access, the method returns a negative outcome, preventing stream capture. [5]

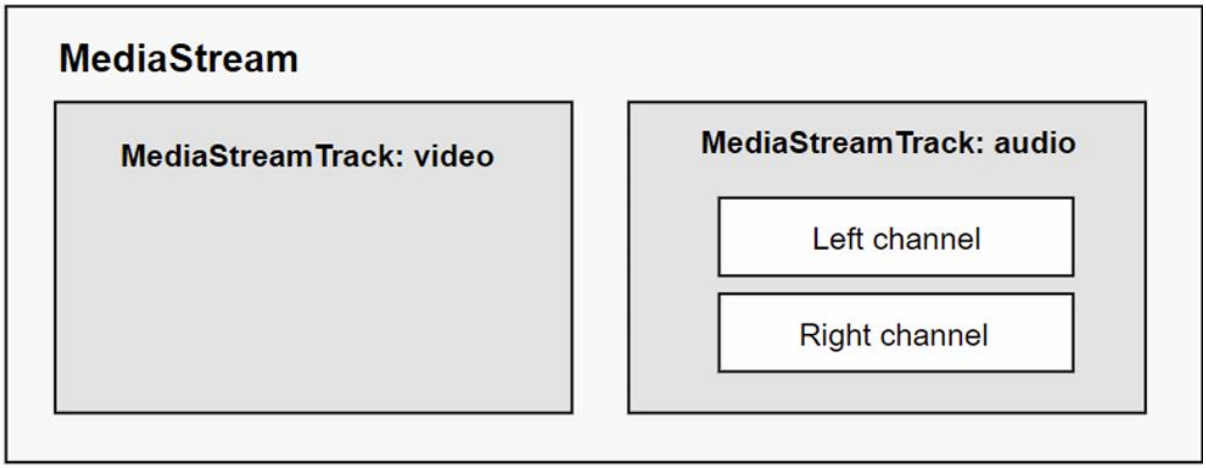

Figure 2-8: Visualization of a MediaStream Object. [33]

2.5.2 RTCPeerConnection API

In WebRTC, the RTCPeerConnection component is used to establish a direct connection between browsers. It also helps set the connection from the signaling phase to a stable state. The WebRTC system is mainly built on three parts: audio, video, and transport. [6]
The audio part handles things like echo cancellation and noise reduction. It includes codecs such as iSAC and iLBC, where iSAC is mainly used for streaming audio. This codec was originally created by Global IP Solutions in 2011 and was added to WebRTC to improve audio quality.
The video part manages the video stream and uses the VP8 codec. VP8 was developed by On2 Technologies in 2008 and is supported by all major browsers. It helps improve video quality and reduces issues like audio and video delay (jitter).
The transport part uses a protocol called SRTP (Secure Real-Time Protocol), which handles the secure delivery of media streams.
WebRTC also offers a getStats() API that developers can use to collect statistics from the client side. This API is useful for monitoring the performance of WebRTC applications. It provides data in four main categories [7]: sender media capture statistics, sender RTP statistics, receiver RTP statistics, and Receiver media statistics

2.6.3 RTCDataChannel API

Data streams in WebRTC are exchanged in both directions from the clients. This exchange is done through the RTCDataChannel mechanism. Using RTCDataChannel, the delivery status of the message can be known. Text messaging and file sharing are done using RTCDataChannel. By this, there would be low latency. [8]

2.7 A Generic WebRTC application flow

With a MediaStream (Section 2.6.1), the client can start sending the stream using the WebRTC API. To create a new real-time connection, we use the primary interface in the WebRTC API, which is RTCPeerConnection (Section 2.6.2). This interface allows the application to start the procedure to connect with a peer, given that some pre-conditions are met (as described in section 2.4).

To begin using the RTCPeerConnection object, it must first be created. The MediaStream Tracks that have been collected with the Media Stream API can be attached with the help of the RTCPeerConnection.addTrack() method. The attached media tracks will be sent to the remote peer when the full exchange of information and agreement is done. To start the exchange with a remote peer, an SDP must be generated and sent. The application's signaling layer should manage to send and receive SDPs. The RTCPeerConnection.createOffer() method can be used to generate this information. The SDP can then be attached to the RTCPeerConnection object with a method called RTCPeerConnection.setLocalDescription(), and this method will also trigger the ICE Negotiation phase, which is the second phase.

In the ICE Negotiation phase, the PeerConnection ICE agent will start to create ICE candidates. Usually, a STUN server is needed to traverse networks behind firewalls. i.e., if a client is behind a NAT. The ICE agent will reach out to STUN servers and receive a Server Reflex ICE candidate back. In some cases where STUN is not enough, as mentioned earlier in section 2.4, a TURN server can be used to relay the stream in order to overcome more complex network topologies. When both the initializing phase and the ICE negotiation phase are done, the SDP and all the ICE Candidates must be delivered to the remote peer. This part of a WebRTC application is referred to as “signaling” and is up to the developer to figure out and implement, as it is not a part of the WebRTC API.

When the remote peer has received the SDP offer from the other peer, the remote peer will go through pretty much the same phase as the other peer did. The remote peer will create its RTCPeerConnection object because the RTCPeerConnection object will have all the information about the session and will represent an association with the other peer. The remote peer will attach the incoming SDP to its RTCPeerConnection object and at the same time create an answer (with RTCPeerConnection.createAnswer() method) which will generate an SDP to send back. [13]

When both peers have received each other’s SDP information and the ICE servers have completed the agreement of ICE candidates, a peer-to-peer connection is fully established. Now, both peers can receive each other’s MediaTracks by adding MediaTracks to the RTCPeerConnection object with the RTCPeerConnection.addTrack method. [14]

A detailed flow of a peer-to-peer connection setup described above can be seen in Figure 2-9.

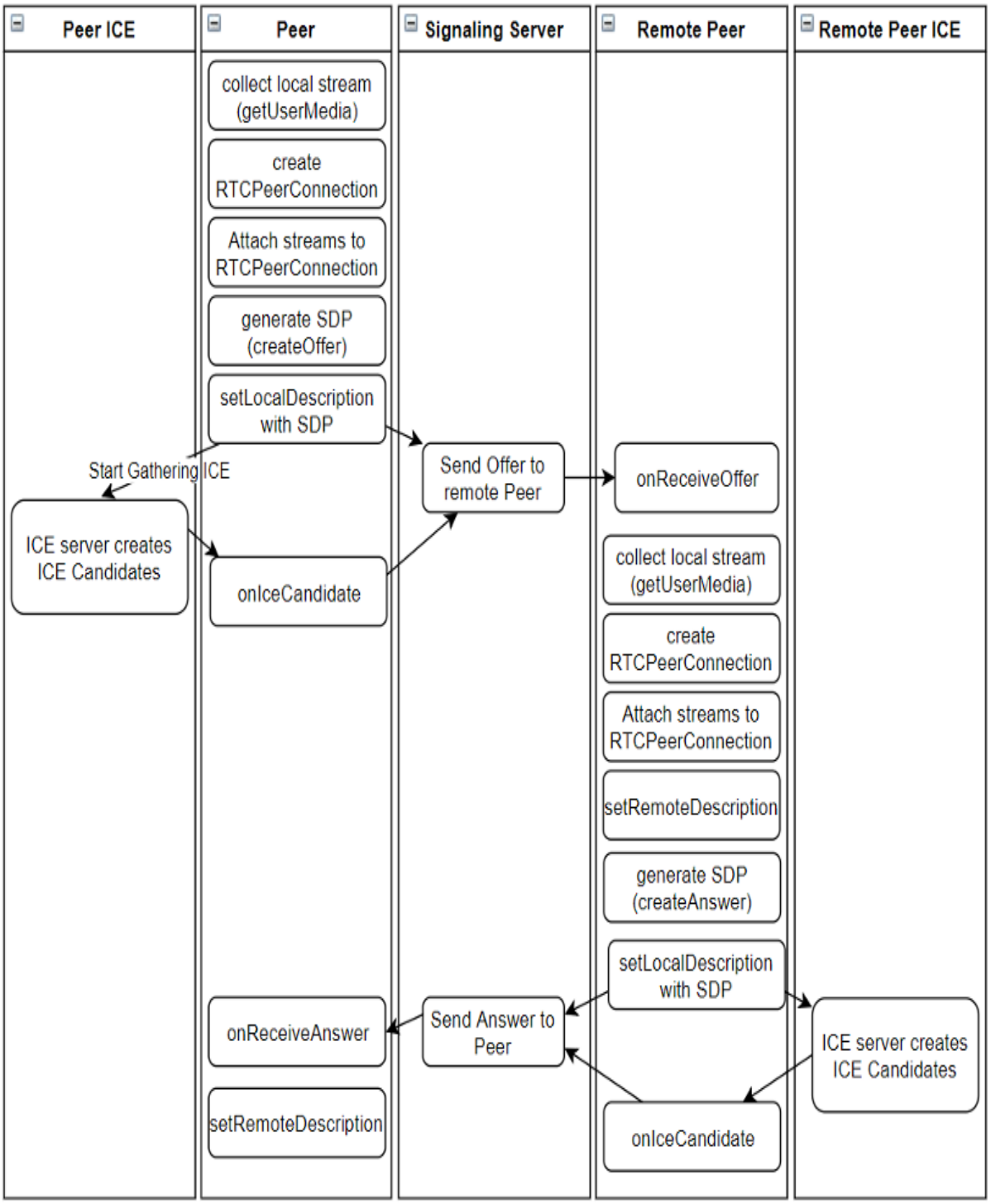

Figure 2-9: Workflow for Setting Up a WebRTC Peer-to-Peer Connection.

Conclusion:

In this chapter, a complete, detailed study on WebRTC and how this technology works is done, with an in-depth analysis and explanation of WebRTC's underlying protocols, and a discussion about the requirements needed was made. In chapter 3, a full study on designing a WebRTC-based web application is delivered to concretize all the theoretical explanations.

Chapter 3

System Analysis and Design

Introduction :

This chapter presents the detailed system analyses and design process of the WebRTC-based video conferencing application. It identifies the key actors, specifies the functional and non-functional requirements and outlines the system architecture, component interactions through UML diagrams.

3.1 Actors Identification

Actors are external entities interacting with your system. In the application, the actors are:

- **User or Client:** it's the one who accesses the web application to initiate or receive video/audio calls.
- **Signaling Server:** A server facilitating the exchange of signaling data (e.g., SDP, ICE candidates) necessary for establishing peer-to-peer connections.
- **WebRTC Peer Connection:** represented as an abstract actor like the browser or the application component that handles the media/data peer connection.
- **STUN/TURN Servers:** External servers that assist with NAT traversal and relay media if needed.

3.2 Functional Requirements

Functional requirements describe what the system should do. The functional requirements to which our application meets are as follows:

- The system shall allow users to register/or log in using a stateless authentication (stateless) thanks to JWT to access the various features.
- The system shall display a list of online users available for connection.
- The system shall allow a user to initiate a WebRTC call by selecting a peer.
- The system shall allow users to accept or reject incoming connection offers.
- The system shall exchange SDP offers/answers and ICE candidates between peers.
- The system shall initiate a direct media stream (audio/video) after connection establishment.
- The system shall disconnect the session upon user request or network interruption.

3.3 Non-Functional Requirements

The non-functional requirements define the quality attributes and constraints that the WebRTC-based web application must meet:

3.3.1 Security

- The application must ensure secure authentication of users by implementing JSON Web Tokens (JWT).
- All signaling communication between clients and the signaling server must occur over secure WebSocket (WSS) connections to protect against man-in-the-middle attacks.
- Media streams must be encrypted using Secure Real-Time Transport Protocol (SRTP) to guarantee the confidentiality and integrity of transmitted audio and video data.

3.3.2 Interface Ergonomics

- The application must provide a simple, intuitive, and user-friendly interface to enhance usability.
- The interface design must be ergonomic, minimizing user effort in navigation and operation.
- Common user actions, such as initiating or answering calls, must require no more than two clicks.

3.3.3 Compatibility

- The application must be fully compatible with all major modern web browsers, including Google Chrome, Mozilla Firefox, and Microsoft Edge.
- It must also support access from various device types, including desktop computers, laptops, tablets, and smartphones, ensuring a consistent user experience across platforms.

3.3.4 Availability and Scalability

- The application must achieve a minimum service availability of 99.99% to ensure continuous access.
- The system must support a minimum of **10 concurrent signaling users** without degradation of service quality.

3.3.5 Performance

- The application must respond to user actions within milliseconds to provide a seamless experience.
- Establishing a WebRTC connection between two peers, from offer creation to media stream activation, must be completed in less than 2 seconds under normal network conditions.
- The application must efficiently manage resource usage, minimizing server CPU and memory consumption, to maintain responsiveness even under load.

3.4 Modelling of functional needs

3.4.1 Introduction to UML in System Design

Given the defined objectives of the project, the system will be modular and must maintain extensibility for future enhancements. As a result, adopting **UML** as standardized modeling language is critical to ensure design clarity and streamline communication.

3.4.2 Definition

The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. UML provides a standard way to write a system's blueprints, including conceptual components such as business

processes and system functions, as well as concrete elements like programming language classes, database schemas, and reusable software components. [18]

3.4.3 UML advantages:

UML modeling offers several key benefits, making it indispensable in software and systems engineering: [19]

1. **Universality**
    - Widely recognized by designers and developers worldwide, ensuring a common language for cross-team and cross-border collaboration.
2. **Industry Adoption**
    - Used by major enterprises (e.g., IBM, Boeing, Siemens) to model systems of varying scales and complexities, proving its scalability and robustness.
3. **Standardized Notation**
    - Provides a unified set of diagrams (e.g., class, sequence, use case) that reduce ambiguity and improve consistency in documentation.
4. **Clarity & Readability**
    - Visual representations simplify complex system structures, making them accessible to both technical and non-technical stakeholders.
5. **Error Reduction**
    - Early visualization of system logic helps identify design flaws before implementation, lowering development costs and risks.
6. **Domain Agnosticism**
    - Applicable beyond IT (e.g., business processes, healthcare workflows, industrial systems), supporting interdisciplinary projects.
7. **Tool Support**
    - Supported by powerful tools (e.g., Enterprise Architect, Lucidchart, StarUML) that enable code generation, reverse engineering, and automated documentation.
8. **Lifecycle Coverage**
    - Supports all phases of development—from requirements analysis (use cases) to deployment (component diagrams)—ensuring continuity.
9. **Flexibility**
    - Compatible with agile methodologies and can be adapted to specific project needs (e.g., lightweight UML for startups).

10. **Educational Value**

- Serves as a foundational teaching tool in computer science curricula, bridging theory and practical design.

3.4.4 System's diagrams

For requirements modeling, we use the following UML diagrams: use case diagrams, sequence diagrams, class diagrams, and component diagrams. For this purpose, we have selected **StarUML** as our modeling software.
StarUML is an open-source software modeling tool that supports UML (Unified Modeling Language) standards. It provides a versatile environment for creating UML diagrams, including use case, class, sequence, component, and deployment diagrams. StarUML facilitates model-driven development with features such as code generation, reverse engineering, and extensibility through plugins. [20]

3.4.5 Use case Diagram:

In the Unified Modeling Language (UML), a use case diagram can summarize the details of your system's users (also known as actors) and their interactions with the system. To build one, you'll use a set of specialized symbols and connectors. An effective use case diagram can help your team discuss and represent: [21]

- Scenarios in which your system or application interacts with people, organizations, or external systems
- Goals that your system or application helps those entities (known as actors) achieve
- The scope of your system.

Below, it is the **Use case Diagram** of the system:

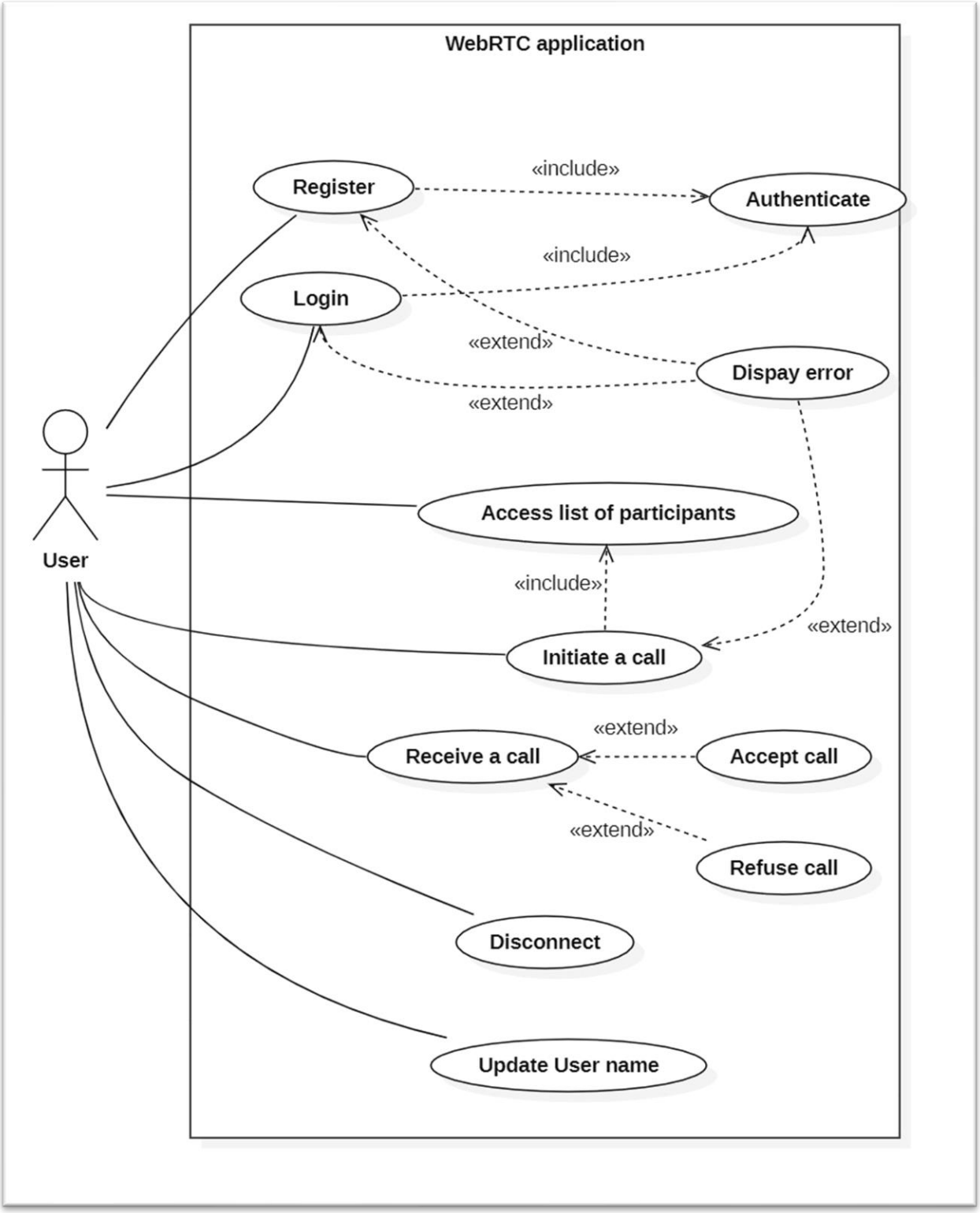

Figure 3-1: Use Case Diagram of the WebRTC application

The following use case diagram illustrates the functional interactions between the user and the WebRTC-based web application. The goal of this diagram is to capture and organize the key functionalities that the system must support from the user's perspective.

Actor:

User: The only actor represented in this use case diagram is the end user, who interacts with the application through the web interface.

Use Cases

1. **Register**
   This use case allows new users to create an account within the application. It includes the Authenticate use case to verify and securely store user credentials. The Register use case may also trigger the Display Error use case if any errors arise during the registration process, like an invalid input or network errors.

2. **Login**
   Existing users authenticate themselves by logging into the system. Similar to registration, it includes the Authenticate use case to validate the provided credentials and extends the Display Error use case in case of incorrect login attempts or service failure.

3. **Authenticate (<<include>>)**

   This is a common functionality included by both the Register and Login use cases. It handles the actual process of checking user credentials (with JWT-based authentication) and either granting access or denying it.

4. **Display Error (<<extend>>)**

   This use case represents the system's behavior when an error occurs during various operations. It is conditionally triggered by other use cases such as Register, Login, and Initiate Call.

5. **Access Participant List**

   This use case allows the user to view a real-time list of currently available participants, which is a prerequisite for initiating a peer-to-peer call.

6. **Initiate Call**

   The user can start a WebRTC call after selecting a participant from the list. This use case includes the 'Access Participant List' functionality and may also extend the 'Display Error' use case if issues such as network failure or participant unavailability occur.

7. **Receive Call**

   This use case models the scenario where a user is the recipient of an incoming WebRTC call. When another user initiates a call, the system sends a prompt to the recipient, asking whether to 'accept' or 'reject the call'.

8. **End Call**

   This use case enables the user to terminate an active call session at any time.

9. **Update Display Name**

   This optional use case lets users update their display name, which is used to identify participants in the application interface.

10. **Disconnect**

    This use case represents the user intentionally or unintentionally leaving the application. It involves removing the user from the participant list.

3.4.6 Sequence Diagram:

A sequence diagram is a type of UML diagram that depicts the order of interactions between participants in a system. It is an interaction diagram that illustrates how operations are carried out and includes objects, links, and messages. [22]

Sequence diagrams are particularly useful in modeling the logic of complex operations, functions, or procedures by showing the sequence of messages exchanged between objects. They play a decisive role in both the design phase, to validate system architecture and behavior, and in the documentation phase, to provide clear insights into system interactions.

In this section, we present some sequence diagrams of our system:

Scenario 1: WebRTC SDP and ICE Candidate Exchange.

The sequence diagram presented illustrates the process of establishing a peer-to-peer connection between two entities using Web Real-Time Communication (WebRTC). This process involves the exchange of Session Description Protocol (SDP) messages and Interactive Connectivity Establishment (ICE) candidates to negotiate and establish a direct media path between peers.

- **SDP Offer/Answer Exchange:**

Initiator Peer: Begins the connection by generating an SDP offer (SDP contains media configuration details).

Signaling Server: relays the SDP offer to the **Receiver Peer**.

Receiver Peer: after receiving the offer, generates an SDP answer that aligns with the offered parameters and sends it back through the signaling server.

- **ICE Candidate Gathering and Exchange**

Peer ICE Components: Both peers initiate the ICE gathering process to discover potential network paths.

Candidate Exchange: As ICE candidates are found, each peer sends them to the other via the signaling server.

This sequence ensures that both peers agree on the media parameters and establish a reliable and efficient communication channel.

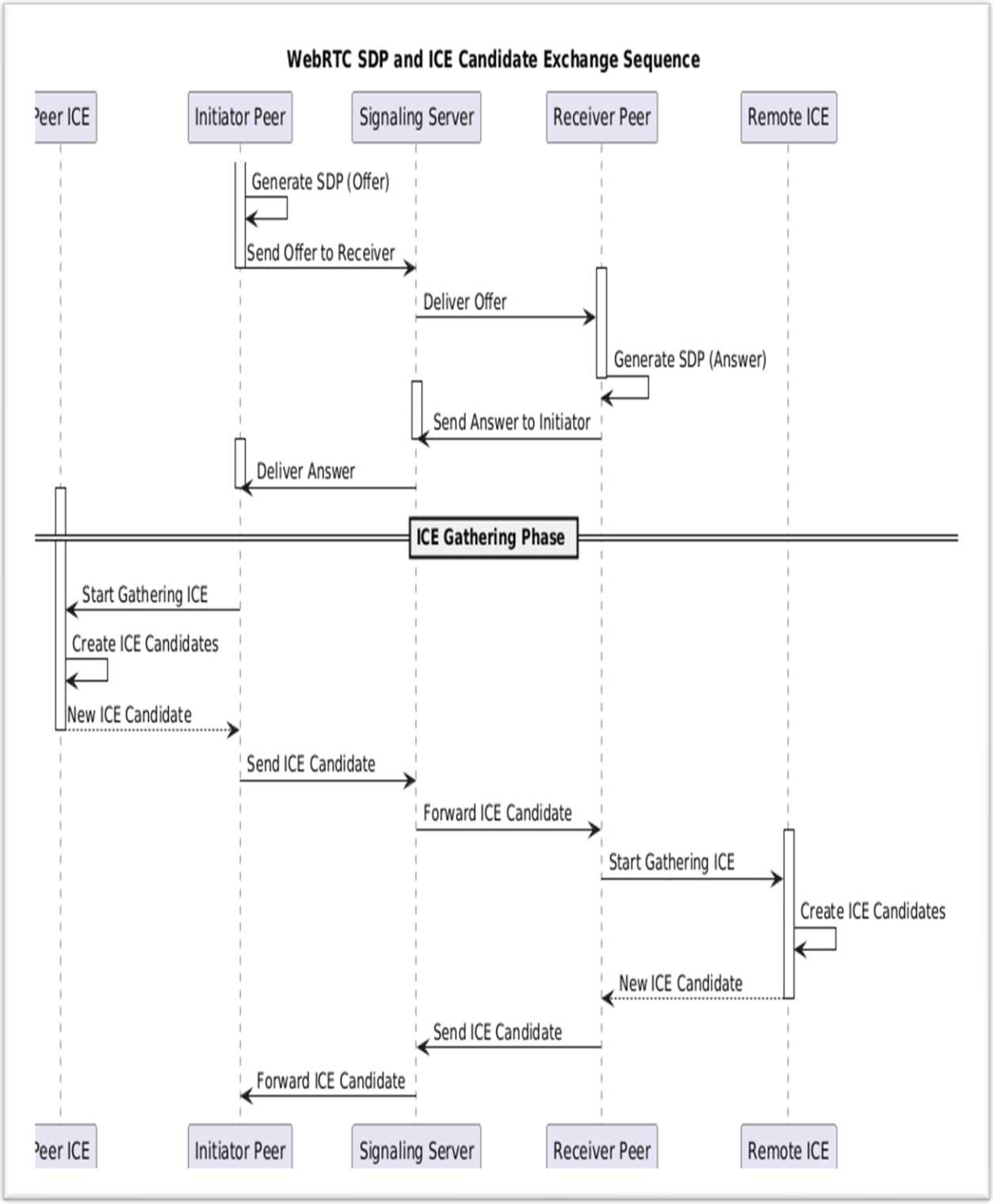

Figure 3-2: Sequence Diagram of WebRTC SDP and ICE Candidate Exchange Process

Scenario 2: Call Initiation and Response.

The sequence diagram illustrates the user interaction in initiating a peer-to-peer WebRTC call and the handling of the response. It captures the behavior between two actors: the Peer (initiator) and the Remote Peer (receiver).
The interaction begins when the initiator peer selects a remote peer from the available participants list and clicks the "Start Call" button which triggers a notification prompt on the receiving end about a call request.
The diagram then models two alternate outcomes:

- **If the remote peer accepts the call**, he clicks the "Accept" button. A response message indicating acceptance is sent back to the initiating peer, and they prepare themselves for connection setup.
- **If the remote peer rejects the call**, he clicks the "Reject" button. A rejection notification is sent to the initiating peer, who then receives a message such as "Call Rejected" in his interface.

The diagram is important as it captures the interactive behavior and the user's decision-making logic in this WebRTC-based application.

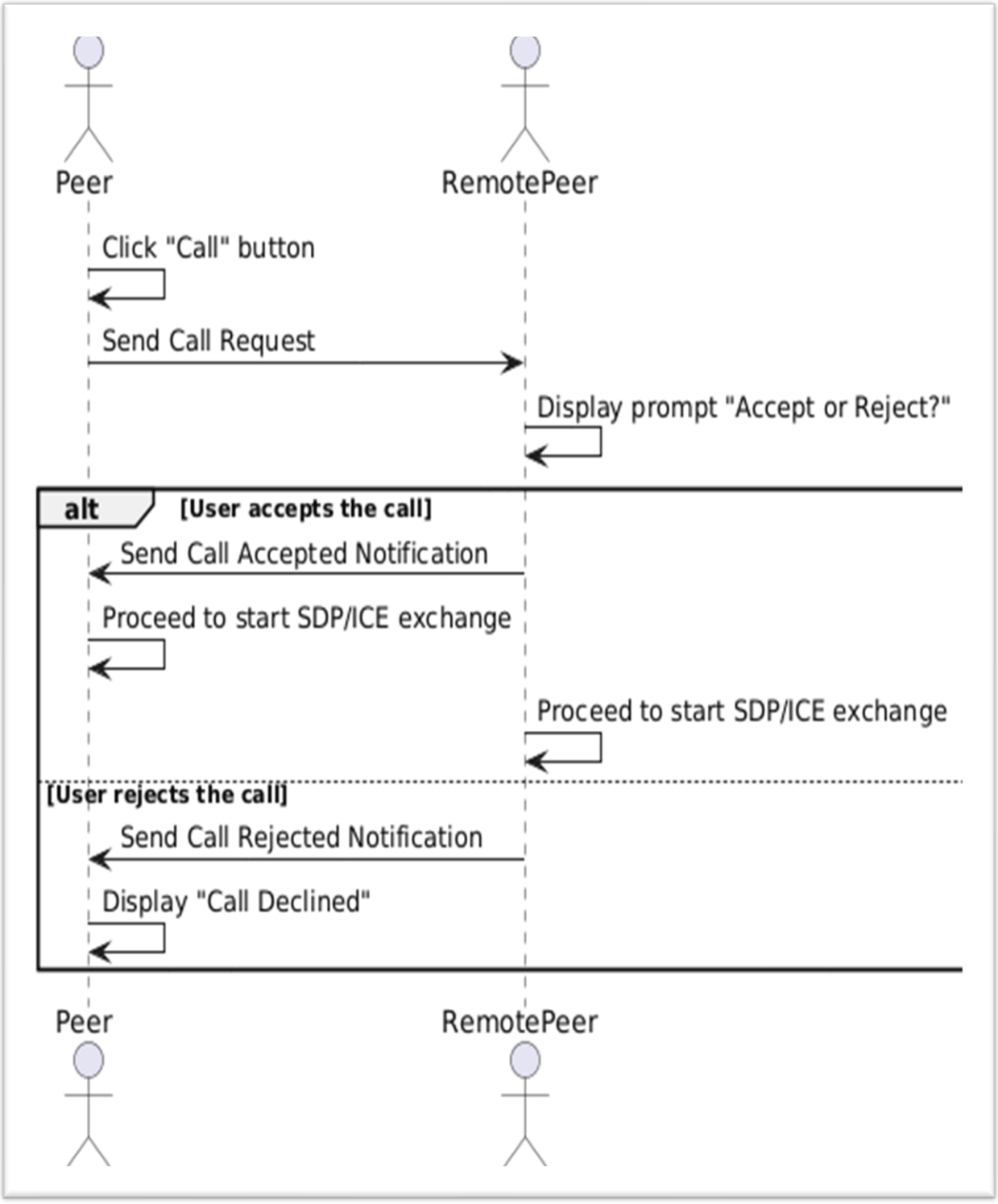

Figure 3-3: Sequence diagram of Call Initiation and Response in WebRTC application.

Scenario 3: User's registration & login.

The sequence diagram illustrates the user registration process in the WebRTC application, which includes JWT-based authentication. The only actor involved is the Peer, representing the end user. The peer interacts with the application interface by providing a name, email, and password, then submitting a registration request.

The UI sends the data to the backend web application. The system verifies whether the chosen name or email is already associated with an existing account by querying the user management component.
The diagram then models two alternate outcomes:

- **If the name or email is already in use**, the system responds with an error, and the UI displays appropriate feedback to the user.
- **If the input is valid**, the system proceeds to securely store the new user's data, and a JWT (JSON Web Token) is then generated by the authentication module to represent the authenticated user session. This token is returned to the client, enabling the peer to access authenticated features of the application.

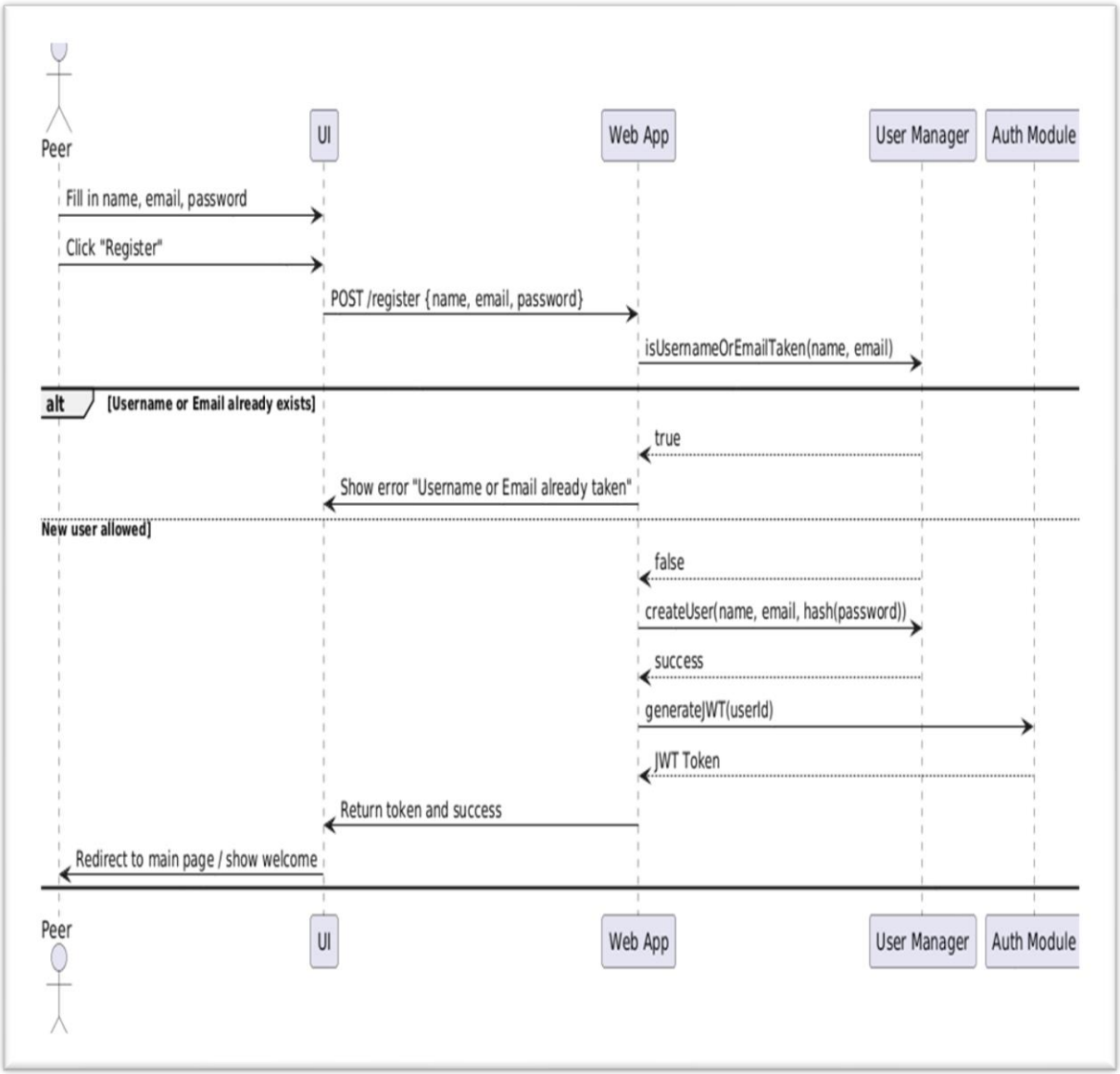

Figure 3-4 : Sequence Diagram of the registration flow in the WebRTC app.

3.4.7 Activity Diagram

Activity Diagram is a type of behavioral diagram in the Unified Modeling Language (UML) that represents the dynamic aspects of a system by modeling the flow of control from one activity to another. Activity diagrams are used to show how a process or workflow works step by step. They are helpful for explaining business logic, user actions, or system operations. These diagrams include elements like actions, decisions, and parallel steps, which help visualize how tasks are carried out, where choices are made, and how different parts of a system run at the same time or come back together. [23]

Key Characteristics:

- **Nodes:** Represent activities (actions), decisions, forks, joins, and object nodes.
- **Edges:** Represent control and object flows between nodes.
- **Swim lanes:** Organize activities into categories (e.g., roles, departments) to show responsibility.
- **Concurrency:** Fork and join nodes depict parallel execution.

Activity diagram for the WebRTC app will model the flow from registration/login to call lifecycle. It includes registration/login validation, JWT issuance, listing participants, initiating/receiving/rejecting/accepting calls, and ending calls. No logout or session refresh logic is required. If a peer is busy, the system does not notify others; calls simply do not go through. No time limits on call prompts.

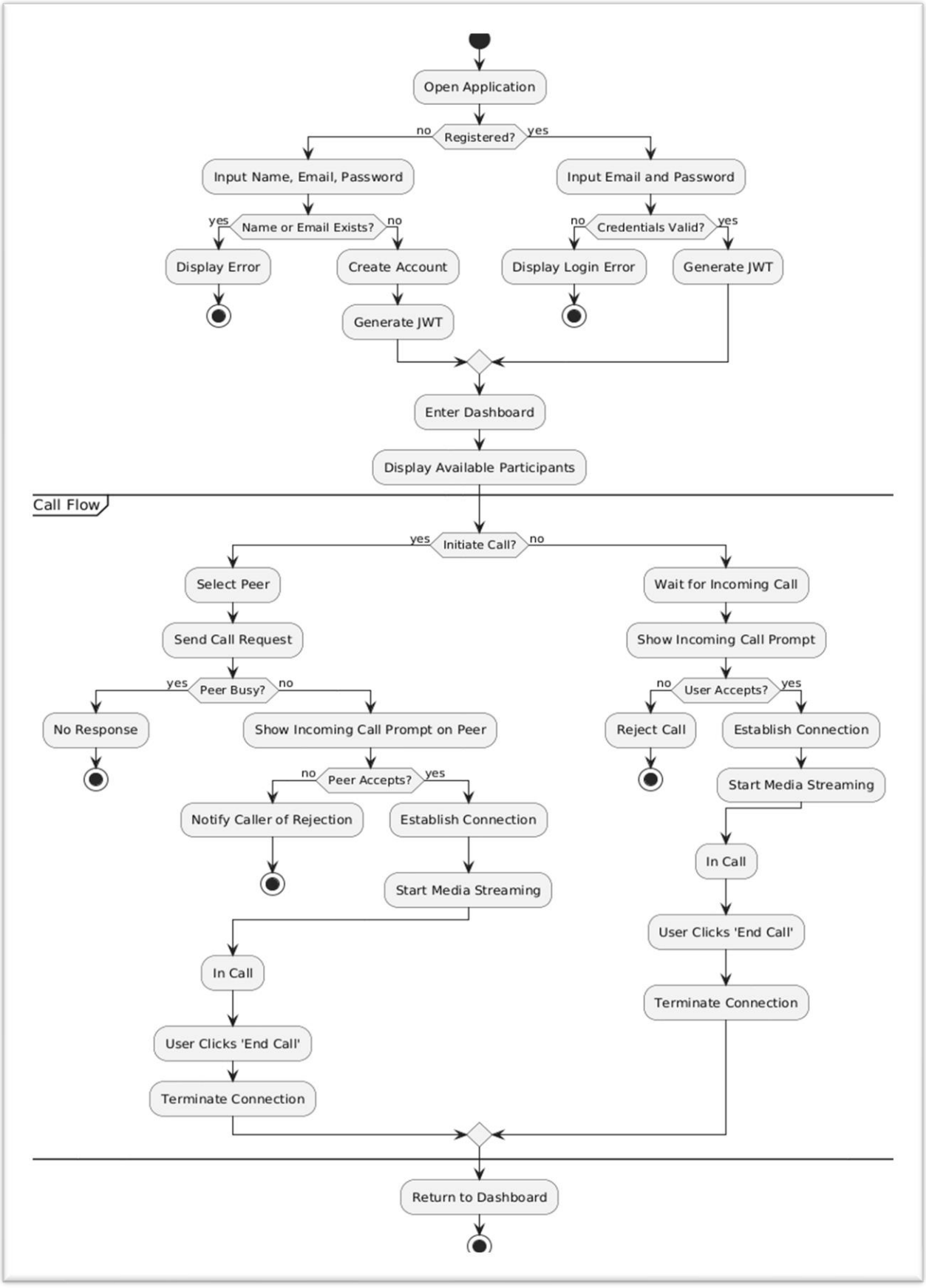

Figure 3-5: Activity Diagram of User Registration, Authentication, and Call Lifecycle in the WebRTC Application.

3.4.8 Components Diagram

A **Component Diagram** is a type of structural diagram in UML that represents how software components are organized and how they depend on each other within a system. It captures high-level, reusable elements such as executables, libraries, modules, and files, showing how they interact through clearly defined interfaces. This diagram offers a static view of a system's implementation, highlighting principles like modularity, encapsulation, and reusability in software design. [24]

Key Elements:

- **Components:** Represented as rectangles with the «component» stereotype, depicting reusable and replaceable modules.
- **Interfaces:** Shown as lollipop (provided interface) or socket (required interface) symbols, defining contracts between components.
- **Dependencies:** Arrows indicating that one component relies on another.
- **Ports:** Explicit interaction points for components.
- **Connectors:** Links between components, often through interfaces.

The component diagram in **Figure 3-6** illustrates the high-level modular architecture of the backend system for a WebRTC-based communication application. It highlights the major components, their responsibilities, and the interactions among them through provided and required interfaces.

Components Overview

- **SignalR Hub Component (WebRTCHub).**
    - Serves as the entry point for all WebRTC signaling messages.
    - Exposes interfaces for managing peer connections (e.g., Offer, Answer, ICE exchange).
    - Requires the Authentication Component for token validation.
- **Authentication Component (AuthService).**
    - Handles login, token issuance (JWT), and token validation.
    - Provides services to both the WebRTCHub and any future HTTP endpoints.
- **User Repository Component (UserRepository).**
    - Abstracts access to the user data in persistent storage.
    - Required by the AuthService to validate credentials or retrieve user information.
- **Data Storage**.
    - A database that stores user credentials.

- Accessed exclusively by the UserRepository.

- **Client (External System).**
  - Represents browser clients or applications interacting with the backend.
  - Communicates with the WebRTCHub over a WebSocket connection using SignalR.

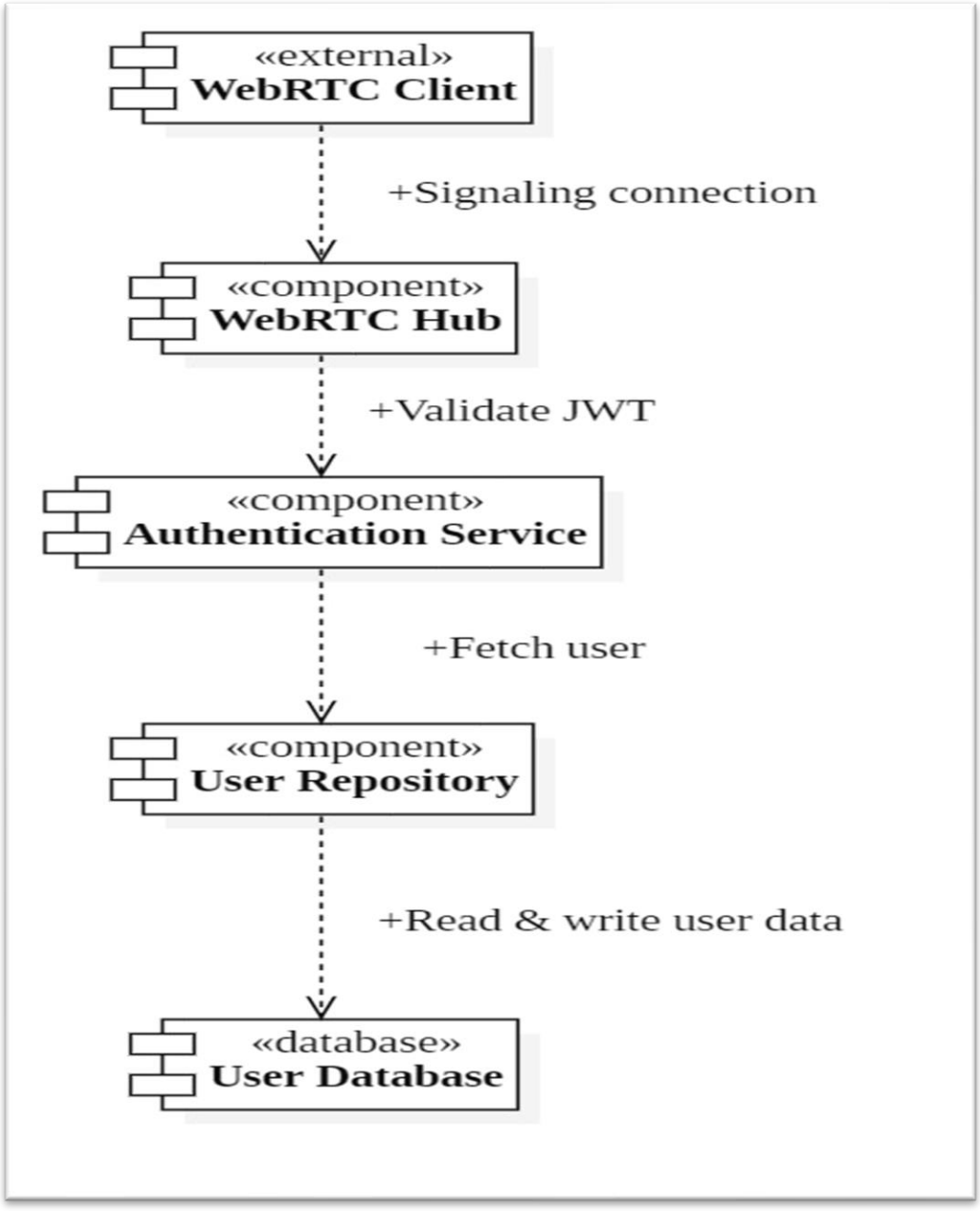

Figure 3-6: Component Diagram of the Backend Architecture for the WebRTC Application.

3.4.9 Class Diagram:

A Class Diagram is a structural diagram in the Unified Modeling Language (UML) that depicts the static structure of a system by showing classes, their attributes, operations (methods), and the relationships among them. It serves as a blueprint for software design, illustrating object-oriented concepts such as inheritance, association, aggregation, composition, and dependency. Class diagrams are fundamental in object-oriented analysis and design (OOAD) for visualizing, specifying, and documenting system architecture. [25]

Key Elements:

- **Classes**: Represented as rectangles divided into compartments for the class name, attributes, and methods.
- **Relationships:** Include associations (with multiplicities), generalizations (inheritance), dependencies, aggregations, and compositions.
- **Interfaces & Abstract Classes:** Used to define contracts and polymorphism.

The class diagram in **Figure X** represents the backend structure of the WebRTC-based application, focusing on authentication, user management, and real-time communication via the signaling server. The diagram strictly focuses on backend classes, as no frontend logic or signaling protocol details (such as SDP or ICE) are illustrated here.

1.WebRTCHub

The WebRTCHub class is responsible for managing real-time communication between clients. It facilitates signaling by relaying offers, answers, and ICE candidates. Internally, it maintains a ConcurrentDictionary<string, string> named _participants to map active connection identifiers to user names, enabling the system to track connected peers during communication sessions.

2. AuthService

The AuthService encapsulates the logic required to authenticate users and issue JWT (JSON Web Token) tokens. It verifies user credentials and generates secure access tokens that clients use to authenticate WebSocket connections.

3. UserRepository

The UserRepository class abstracts data access logic for the user domain. It exposes methods for common operations such as adding and retrieving user records. While a direct association with AuthService is modeled, its relationship to the User class is captured as a **dependency**. This is because the repository's methods utilize User objects as parameters or return type. without retaining direct ownership of them.

4. User

The User class is a data model representing application users. It defines key properties, including Name, Email, and PasswordHash. These attributes are required for user registration, authentication, and authorization processes.

5. Persistence and Application Context

Only user data and authentication tokens are persisted in the backend storage. The active participant list within the WebRTCHub is transient and maintained entirely in memory using a thread-safe dictionary structure.

Relationship Overview

- **WebRTCHub → AuthService**: Modelled as a direct association; WebRTCHub uses AuthService to authenticate incoming connections.
- **AuthService → UserRepository**: Direct association, indicating delegation of persistence-related tasks.
- **UserRepository - - > User**: Dependency relationship; User is used as a method parameter or return type, but not composed within the repository.

Below is the class diagram of the software:

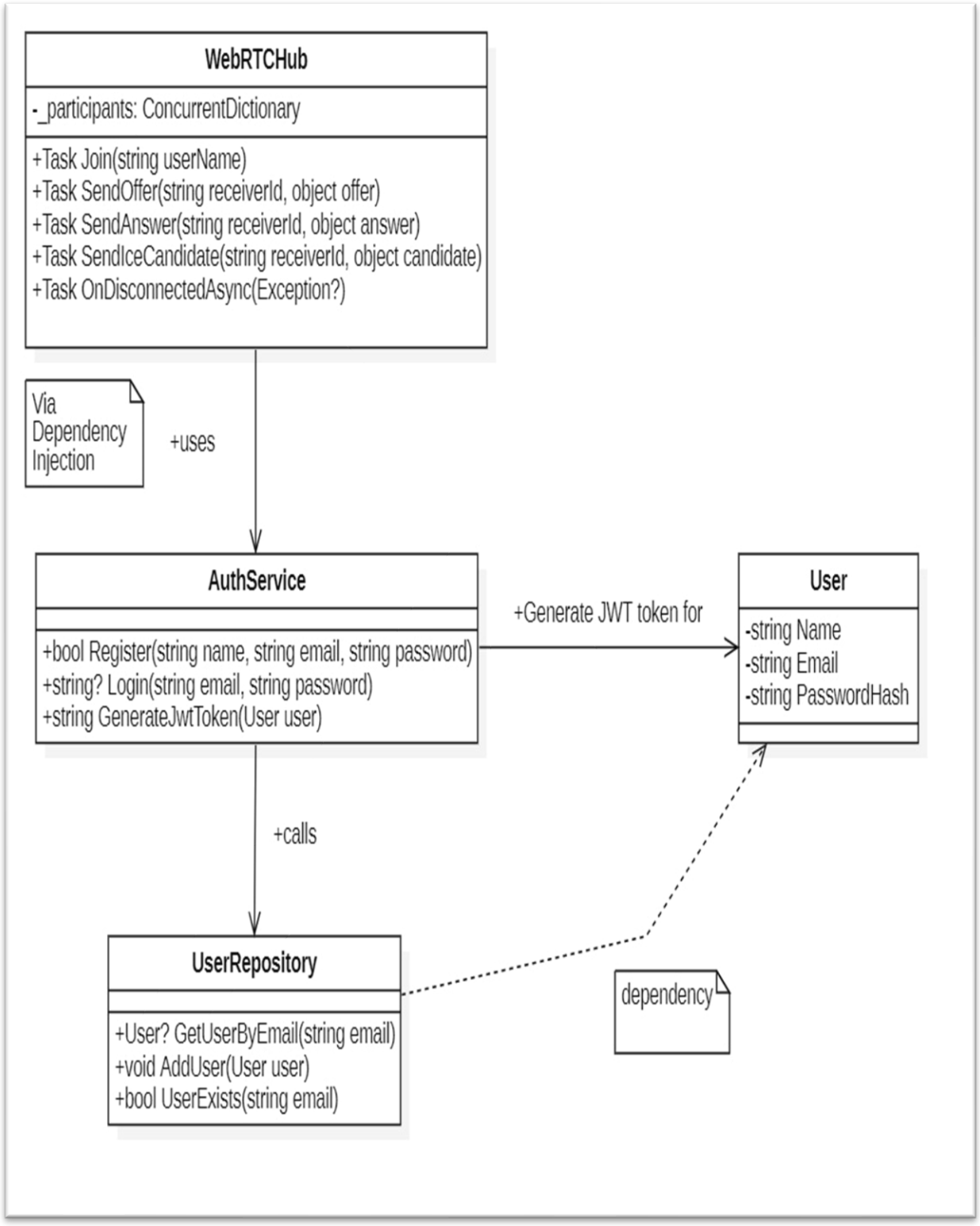

Figure 3-7 : Class Diagram of Backend Components for WebRTC App

Conclusion

This chapter provided a comprehensive analysis of the WebRTC-based application's requirements, and it translated it to a set of UML diagrams, including use case, sequence, activity, component, and class diagrams. Which offers a clear understanding of the functional and structural design of the system where the interactions between the system components and users were modeled and explained. In the next chapter, we transition to the implementation phase, detailing how the designed architecture is realized in practice.

Chapter 4

System Implementation.

Introduction:

This chapter presents the implementation process of the WebRTC-based video conferencing application. It outlines the selected technology stack, describes the development steps taken to realize the system, and discusses the challenges encountered throughout the process. Building upon the analysis and design models presented in the previous chapter. This phase translates theoretical models into functional software components that form the core of the application.
We will see first how a modern web application must be structured. Afterwards, we will detail the technologies used by this WebRTC-based video conferencing application. finally, we will focus on the source code (implementation).

4.1 Technologies and programming languages used.

4.1.1 Web application.

A web application is software that runs in a web browser and uses client-side or server-side logic to provide interactive features over the internet. While it usually follows a client-server model, modern web apps can also use peer-to-peer technologies like WebRTC to let users communicate directly without always going through a server. [26]

4.1.2 RESTful web services

RESTful web services are stateless, client-server-based software systems that conform to the principles of Representational State Transfer (REST). They use standard HTTP methods (GET, POST, PUT, DELETE) to expose resources identified by URIs and allow uniform, scalable, and interoperable communication between distributed components. [27]

4.1.3 Authentication using JWT

JSON Web Token (JWT) is an open standard that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the HMAC algorithm) or a public/private key pair using RSA or ECDSA. [28]

4.1.3.1 JWT Structure

JWT is a character string with three main parts: the head (header), the payload (payload) and signature and separated by points (.) As indicated below. [29]

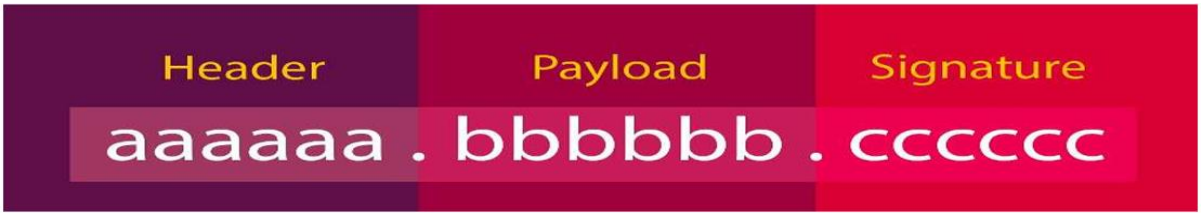

Figure 4.1: Structure of JWT

- **Header:** It contains information on how signature must be calculated, and generally has the following format:
  {
  "algorithm": "HS256",
  "type": "JWT"
  }
  Generally, it is made up of two parts: the type of token which is the JWT and the signature algorithm used as: HMAC SHA256 or RSA.
  Then, this JSON is base64url encoded to form the first part of the JWT.

- **Payload:** It is the data stored in the JWT. We can also have other useful charges
  predefined which are not compulsory but recommended for good practices
  Like: **ISS** (the issuer), **exp** (the time of expiration, **sub** (the subject), **aud** (audience) and others.
  A simple example of Payload can be:
  {
  "sub": "1234567890",
  "name": "Refik Youcef",
  “email”: “Youcef@example.com”,
  "admin": true
  }

  The payload is also base64url encoded to form the second part of the JWT.

- **Signature:** The signature is used to check if the message has not been altered along the way and it is coded using a private key to ensure that the issuer is the one he pretends to be.

  The final encoded token looks similar to this:
  eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36P Ok6yJV_adQssw5c

One of the best-known uses in JWT is authorization that goes hand in hand with authentication and has a scenario like this:

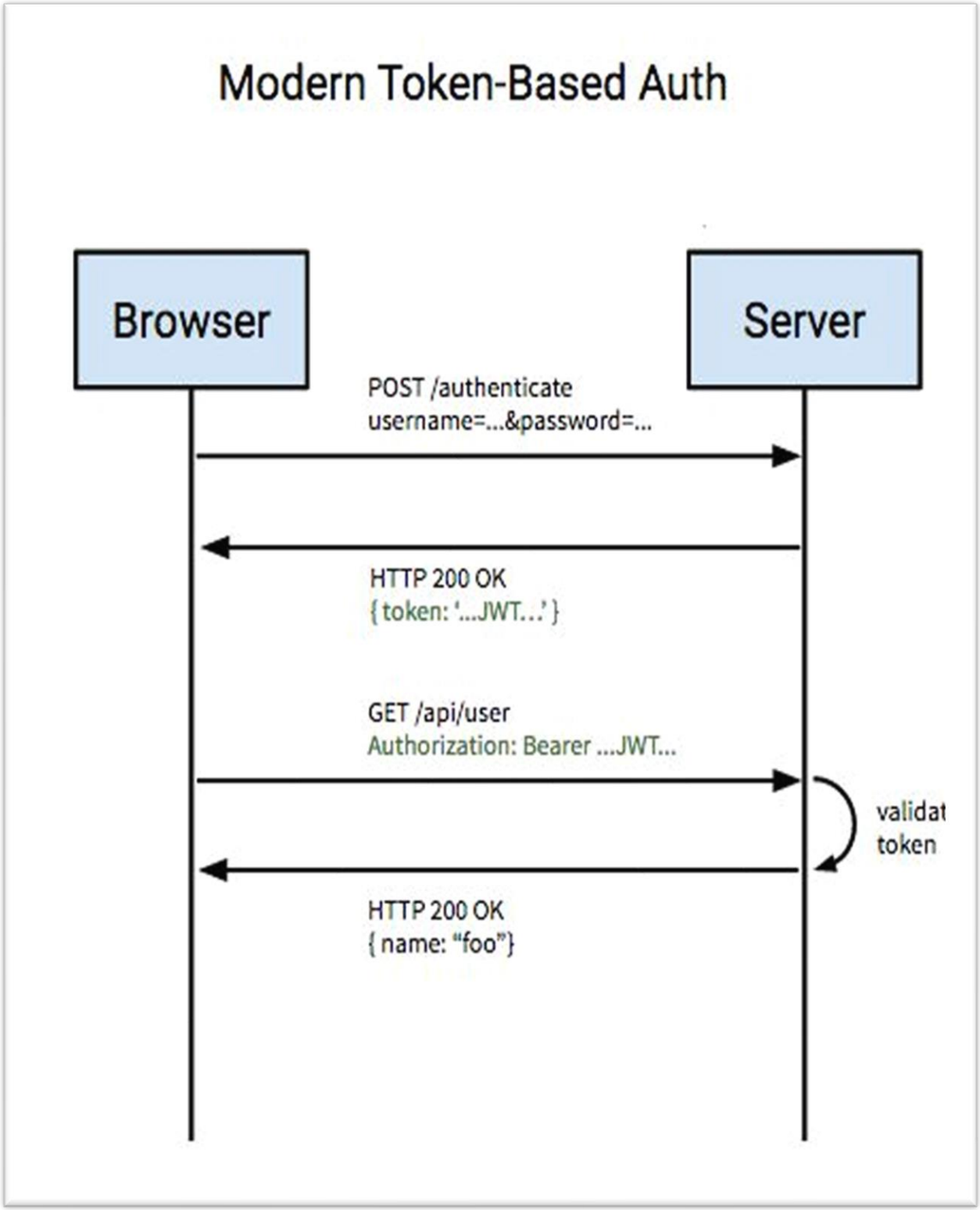

Figure 4.1: Authentication based on JWT.

4.1.4 SignalR

SignalR is a real-time communication library in ASP.NET that facilitates bidirectional communication between a server and multiple clients [30]. It is commonly used for signaling in WebRTC applications, where it helps establish connections by exchanging SDP offers, answers, and ICE candidates. SignalR supports multiple transport mechanisms:

- WebSockets: the preferred transport for real-time communication.
- Server-Sent Events (SSE): a fallback for streaming events from server to client.
- Long Polling: a legacy fallback where the client continuously requests updates from the server.

SignalR simplifies WebRTC signaling by managing real-time message exchange between peers and servers.

4.1.5 ASP .NET CORE

ASP.NET Core is an open-source, cross-platform framework developed by Microsoft for building modern, cloud-based, and internet-connected applications, such as web apps, IoT apps, and mobile backends. It is a redesign of the original ASP.NET framework, offering improved performance, modularity, and support for containerized environments. ASP.NET Core supports multiple platforms, including Windows, macOS, and Linux, and integrates seamlessly with modern development workflows and tools. [31]

4.1.6 SQL Server

Microsoft SQL Server is a relational database management system (RDBMS) designed for storing, retrieving, and managing structured data. It offers a secure, scalable, and reliable platform suitable for building enterprise-level data-driven applications. [34]

Below in figure 4.2 a representation of the entire technology stack used in this application.

4.1.7 Technology Stack:

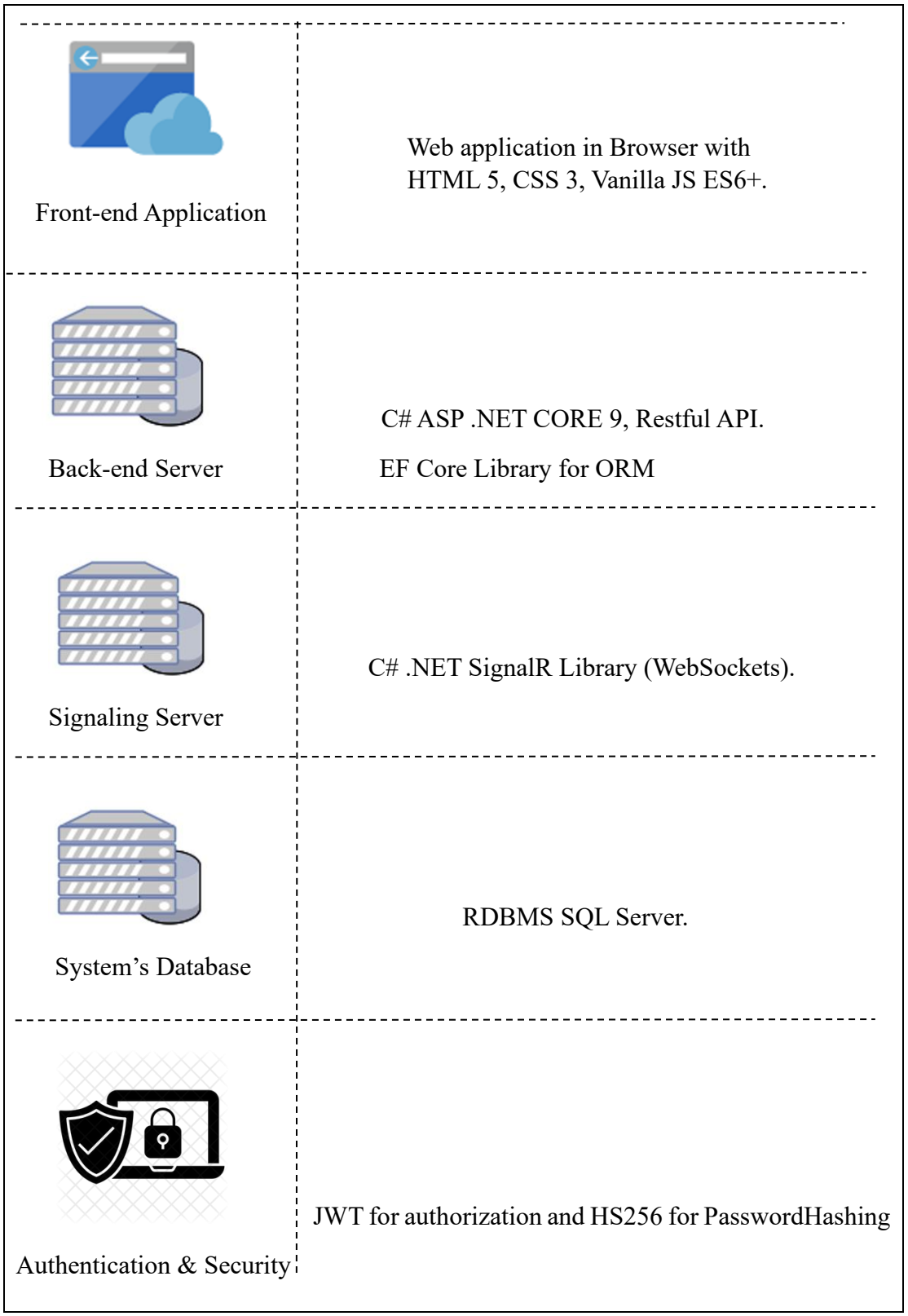

Figure 4.2: An overview of the technology stack.

4.2 Application architecture:

The application is a peer-to-peer WebRTC app that requires a backend server to handle signaling, as well as user registration and login, secured using JWT authentication. The backend is built with ASP.NET Core 8, exposing RESTful APIs (/register, /login) and hosting a SignalR Hub (/webrtcHub) to relay signaling messages (SDP offers/answers and ICE candidates) over WebSockets.

It communicates with a SQL Server database using Entity Framework Core for data persistence.

On the frontend, the app uses HTML, CSS, and JavaScript, with separate templates auth.html (for login/register), dashboard.html (launch interface), and index.html (main call interface). WebRTC functionality is handled by a custom 'WebRTCManager' class in JavaScript which manages media capture, peer connections, and signaling interaction.

Peers connect directly using 'RTCPeerConnection' and exchange media streams. For NAT traversal, the browsers interact with STUN/TURN servers to establish a connection path, acting as ICE servers when direct connections aren't possible.

The architecture separates concerns between signaling, authentication, media transmission, and database persistence, resulting in a clean, maintainable, and scalable design. Below is a figure representing the entire architecture.

The accompanying diagram in next figure visually reinforces this explanation by clearly outlining the interaction flow between the **frontend**, **backend**, **database**, **and peer browsers,** while also showing the supporting role of **STUN/TURN servers** in NAT traversal.

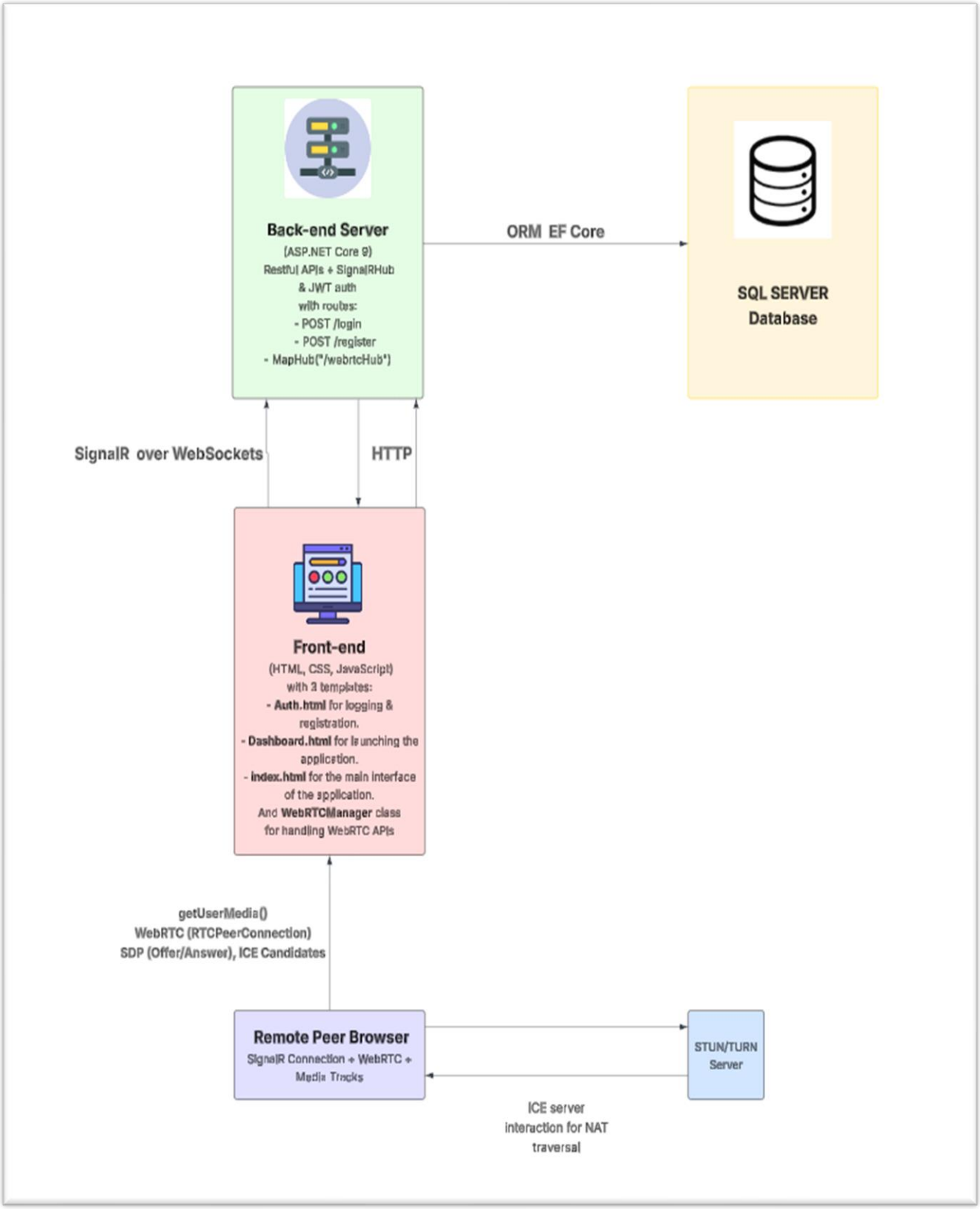

Figure 4.3: The application architecture.

4.3 User Interface Overview.

4.3.1 Login and Registration Interface.

This screen allows users to securely access the application. The login form requires an email and password, while the registration form collects the user's name, email, and password to create a new account. Both interfaces interact with the backend via HTTP requests and use JWT for authentication upon successful login.

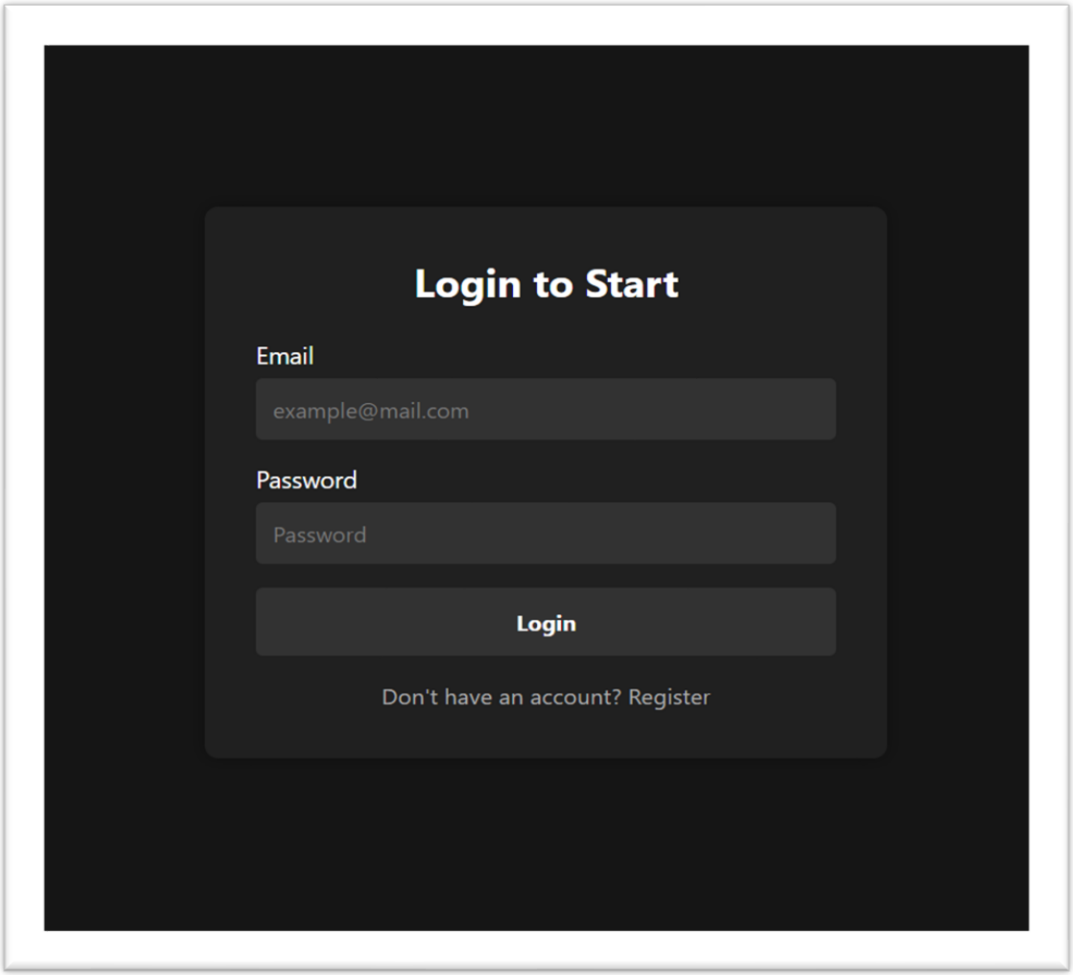

Figure 4.4: Login Interface

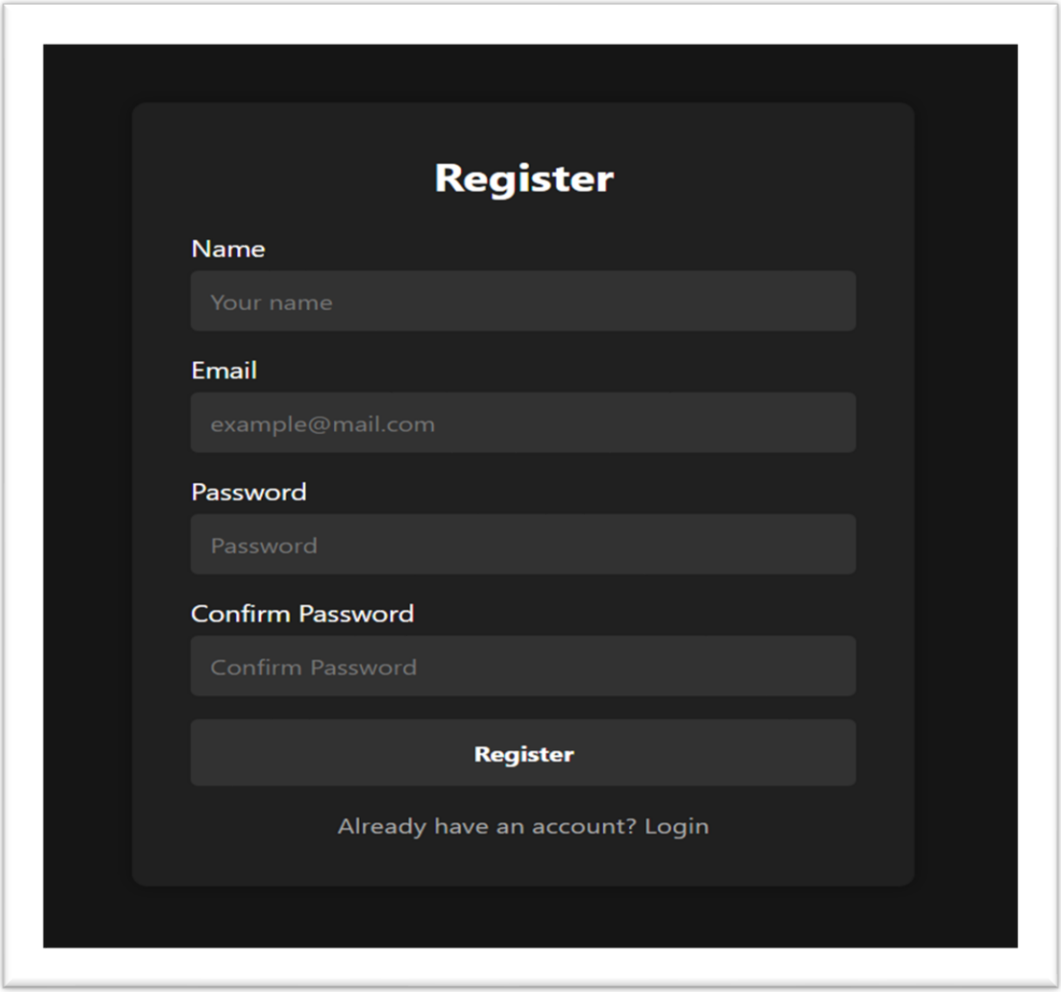

Figure 4.5: Registration Interface

4.3.2 Dashboard Interface

This screen serves as a simple launch point after a successful login. The user is greeted and given two options: proceed to open the main application interface or log out. It provides a clear transition between authentication and active use of the WebRTC features.

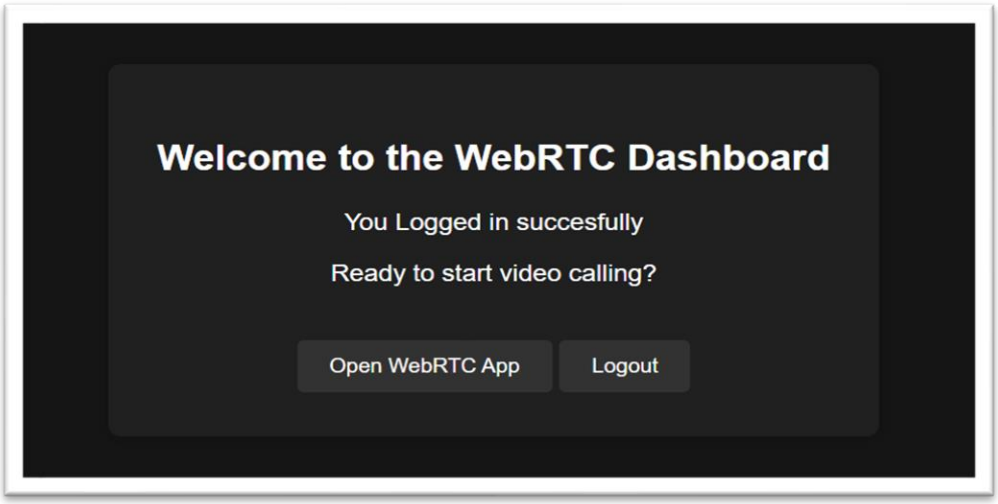

Figure 4.6: The Dashboard Interface

4.3.3 Main Video Conference Interface

This is the core interface where the peer-to-peer video call takes place. It displays all available users and allows receiving and sending call requests in real time. Once in a call, users have essential controls such as toggling the microphone and camera, and leaving the call.

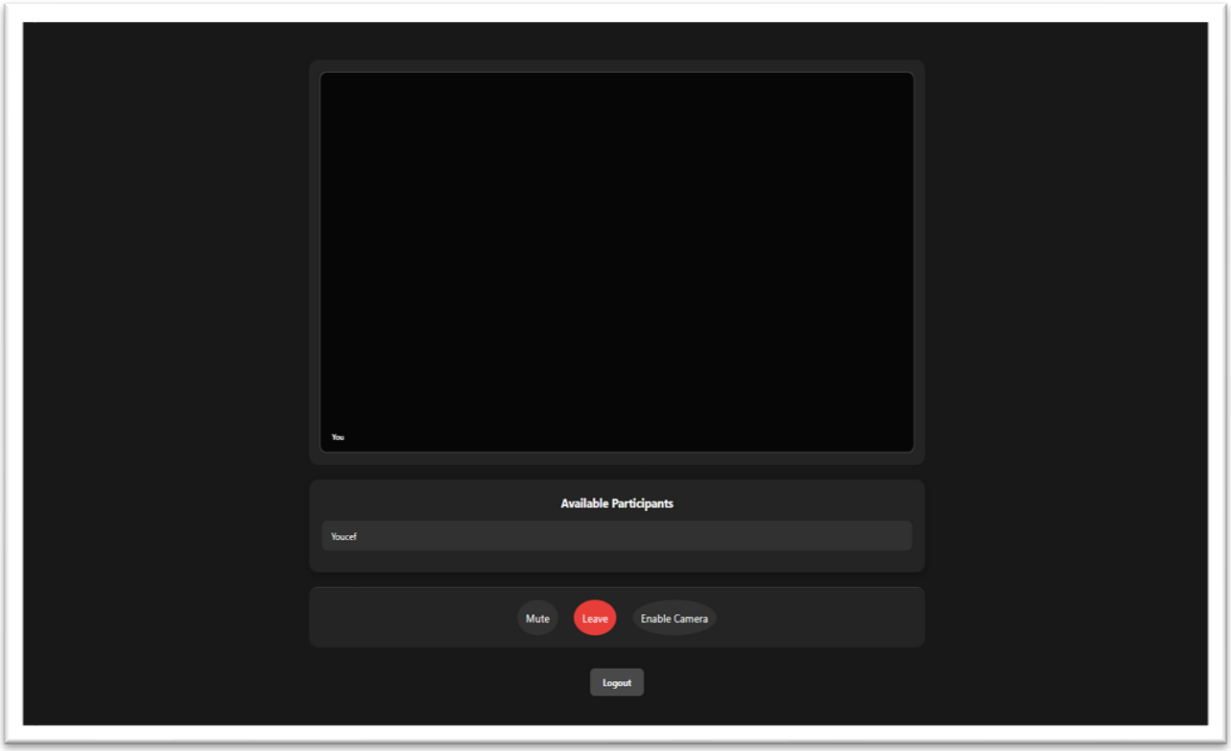

Figure 4.7: Main Video Conference Interface**.**

4.4 Technical Implementation

4.4.1 Signaling phase using SignalR Library:

In this WebRTC application, SignalR acts as the signaling mechanism that facilitates the exchange of signaling data (offers, answers, and ICE candidates) between peers. It also handles user presence management (who is online, who joined or left).
In short version, SignalR simply helps peers find and connect to each other before WebRTC media flow begins from peer-to-peer.
A central component in this system is the 'WebRTCHub' class. This class inherits from Hub and acts as the real-time message broker between clients. Each method in this hub facilitates a specific step in the connection process, from joining the session to sharing signaling messages and handling disconnections.

Hub Methods:

1. **"Join" method**

   This method allows a user to join the system with a unique username. When this method is called:

   - The user is added to an in-memory user list.
   - Their connection ID is mapped to the username.
   - The server then updates all connected clients with the new list of participants.

```
public async Task Join()
{
    var userName = Context.User?.Claims?
    .FirstOrDefault(c => c.Type == JwtRegisteredClaimNames.Name)?.Value ?? "Unknown";

    _participants[Context.ConnectionId] = userName;

    await Clients.Others.SendAsync("UserArrived", Context.ConnectionId, userName);
    await Clients.Caller.SendAsync("ExistingParticipants", _participants);
}
```

Figure 4.8: the 'JOIN' method

2. 'SendOffer' method.

This method is used when one user initiates a call to another. The offer parameter contains the WebRTC session description (SDP offer), and 'targetConnectionId' specifies which client should receive it.

The method sends this offer to the target user so they can decide whether to accept the call.

```
public async Task SendOffer(string
targetConnectionId, string offer)
{
    await
Clients.Client(targetConnectionId).SendAsync("Re
ceiveOffer", Context.ConnectionId, offer);
}
```

Figure 4.9: The 'SendOffer' method

3. 'SendAnswer' method.

After a user receives an offer and decides to accept the call, they respond with an SDP answer. This method sends that answer back to the original caller using their connection ID.

```
public async Task SendAnswer(string targetConnectionId, string answer)
{
    await Clients.Client(targetConnectionId)
                        .SendAsync("ReceiveAnswer", Context.ConnectionId, answer);
}
```

Figure 4.10: The ''SendAnswer' method.

4. 'SendIceCandidate' method

ICE candidates are shared to help find the best path for communication between peers. This method sends an ICE candidate from one peer to another during the connection setup.

```
public async Task SendIceCandidate(string targetConnectionId, string candidate)
{
    await Clients.Client(targetConnectionId)
                        .SendAsync("ReceiveIceCandidate", Context.ConnectionId
                                                        , candidate);
}
```

Figure 4.11: The 'SendIceCandidate' method.

5. 'OnDisconnect' method.

When a client disconnects, this method is triggered automatically. It removes the user from the internal list and informs the other clients so they can update their participant lists accordingly.

```
public override async Task OnDisconnectedAsync(Exception exception)
{
    if (_users.TryRemove(Context.ConnectionId, out var username))
    {
        await Clients.All.SendAsync("UserListUpdated", _users.Values);
    }
    await base.OnDisconnectedAsync(exception);
}
```

Figure 4.12: The 'OnDisconnect' method.

4.4.2 'WebRTCManager' JavaScript Class Overview

The 'WebRTCManager' class acts as the central controller for managing real-time communication and user interactions in the frontend of the WebRTC application. It handles three core responsibilities: media device access, peer-to-peer connection setup, and **SignalR**-based signaling.

- Upon initialization, the class verifies the presence of a valid JWT token.

```
checkTokenAndAutoJoin() {
    const token = localStorage.getItem("token");
    if (!token) {
        window.location.href = "auth.html";
        return;
    }
    const payload = this.parseJwt(token);
    if (!payload || !payload.name) {
        window.location.href = "auth.html";
        return;
    }

    this.mainContainer.style.display = 'flex';

    this.initializeMedia().then(() => {
        this.initializeSignalR(token);
        this.connectToSignalR();
    });
}
```

Figure 4.13: Verification of authorized JWT.

- The class then accesses the user's camera and microphone using ‘getUserMedia’ function by first verifying their availability and the compatibility of the browser used.

```
async initializeMedia() {
    try {
        if (!navigator.mediaDevices ||
!navigator.mediaDevices.getUserMedia) {
            throw new Error("getUserMedia is not
supported in this browser.");
        }

        this.localStream = await
navigator.mediaDevices.getUserMedia({
            video: true,
            audio: true
        });

        this.localVideo.srcObject =
this.localStream;
    } catch (error) {
        console.error("Error accessing media
devices:", error);
    }
}
```

Figure 4.14: Media Device Access (Camera & Microphone).

- After that, the class creates an RTCPeerConnection, adds local tracks, and sets up event handlers for ICE and remote streams.

  Note that we are using a **STUN** server from **Google.**

```
createPeerConnection(connectionId) {
    const peerConnection = new
RTCPeerConnection({
        iceServers: [{ urls:
"stun:stun.l.google.com:19302" }]
    });

    if (!this.localStream) {
        console.error("Local media stream is not
initialized.");
        return null;
    }

    this.localStream.getTracks().forEach(track
=> peerConnection.addTrack(track,
this.localStream));

    peerConnection.ontrack = event => {
        this.addRemoteVideo(connectionId,
event.streams[0]);
    };

    peerConnection.onicecandidate = event => {
        if (event.candidate) {

 this.signalRConnection.invoke("SendIceCandidate
", connectionId, event.candidate);
        }
    };

    return peerConnection;
}
```

Figure 4.15: Peer-to-Peer Connection Setup.

4.5 Tests and Validation

4.5.1 Test Cases

a. Peer Connection Call Test:

Two users opened the app in Microsoft Edge and Chrome. Caller initiated a call; callee accepted. Result: Video and audio were exchanged successfully.

Result: ✅ **Pass** (Media stream established)

Note: The tests run on **Chrome** (Version 136.0.7103.114) & **Microsoft Edge** (Version 136.0.3240.64).

b. JWT Token Verification Test

Tried login & register endpoint to determine if the result is a valid JWT by sending HTTP requests with **POSTMAN** using the REST endpoints of the application and verifying result with **jwt.io** website by decoding it.

Result:

- Sending a **POST** request to the **/login** endpoint.

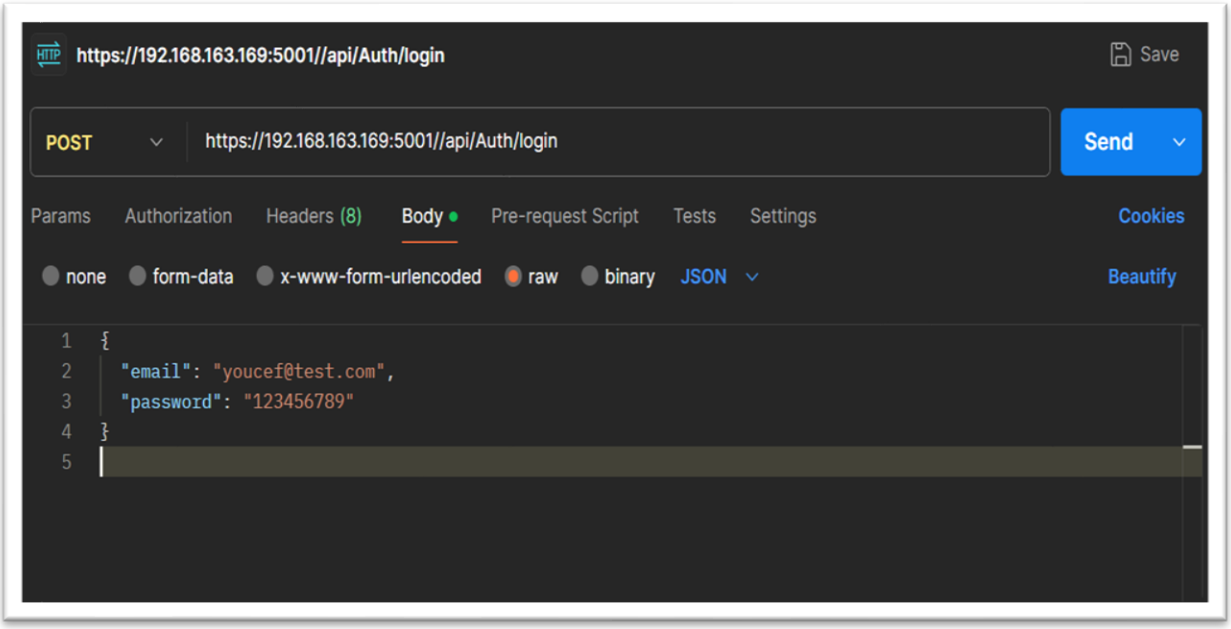

Figure 4.16

- The result was a 200 Code Response (OK) and the result was a coded token.

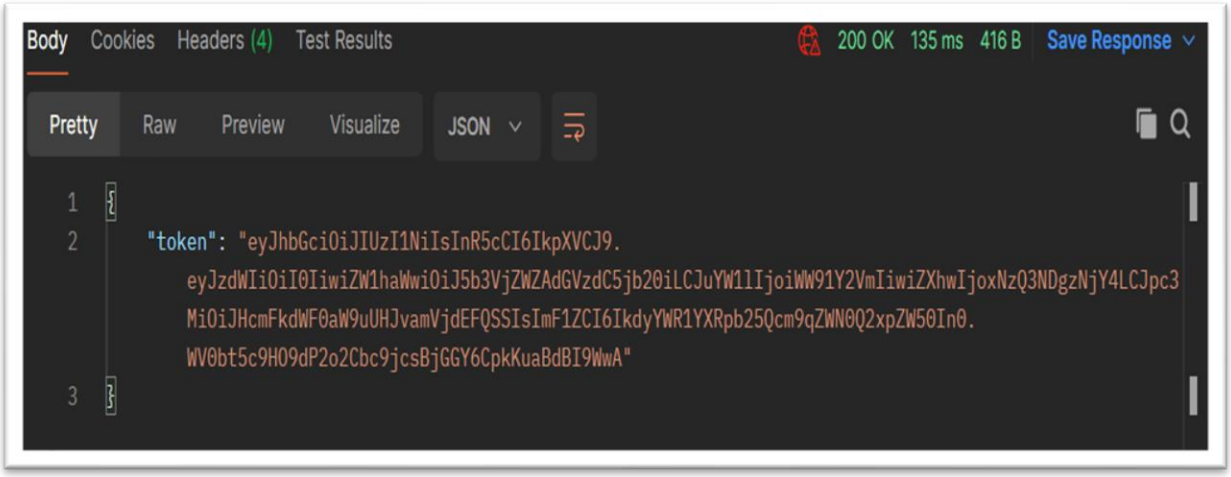

Figure 4.17

- After decoding it using **jwt.io** website this result was obtained.

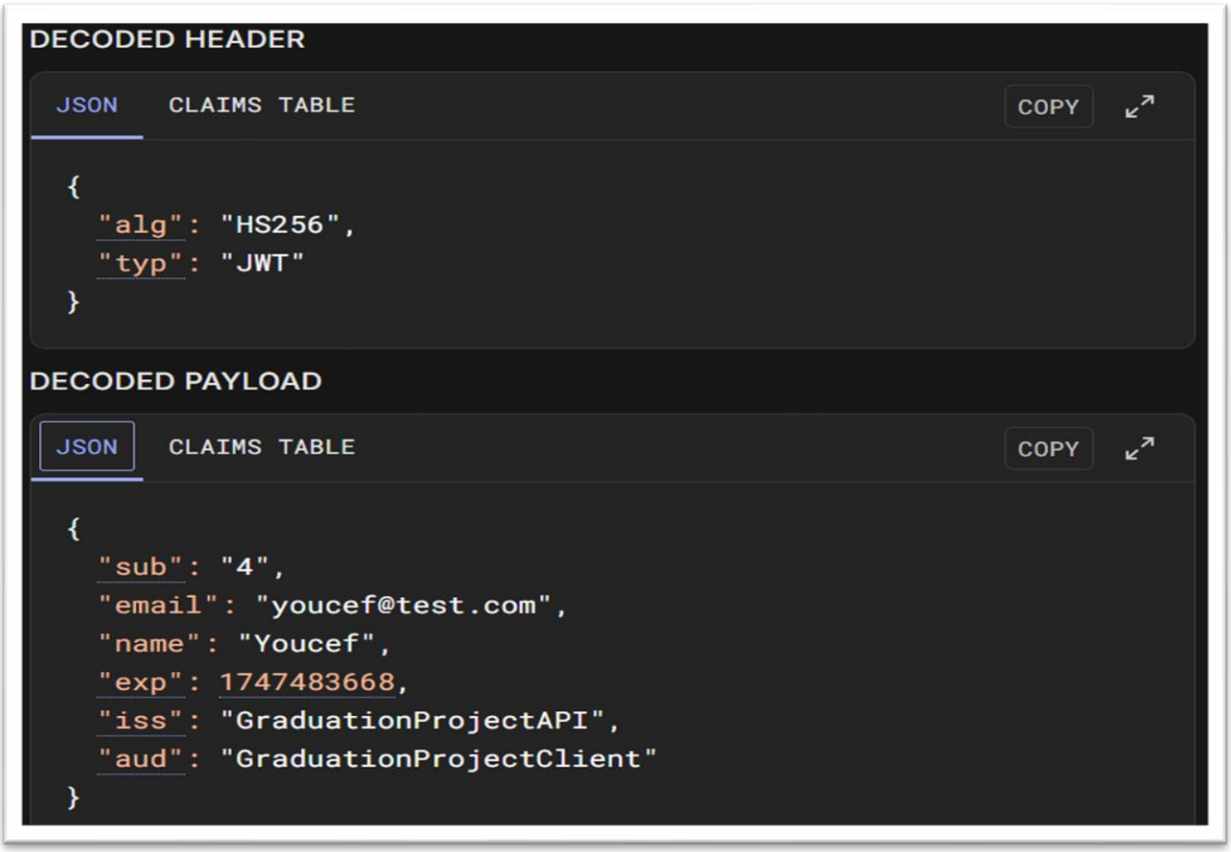

Figure 4.18

✅ **Pass** (JWT is valid & Access is granted).

c. Media Access Denial Test:

Blocked the camera/mic in browser. Result: App showed appropriate error. In this test we blocked the access to the camera/mic by accessing them first from a different bowser which results in their unavailability in the experiential browser.
Result:
Error message appears in console log tab in the browser and it prevents user from initializing calls.

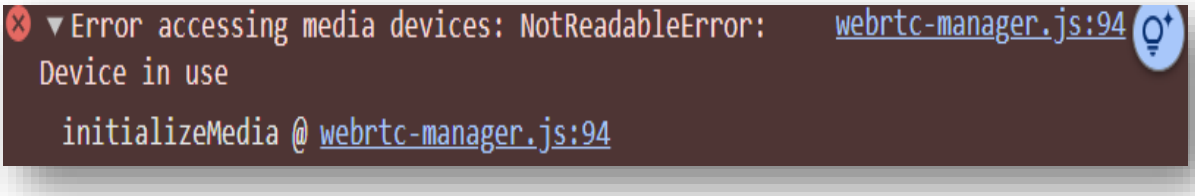

Figure 4.19

✅ **Pass** (Browser denies call initiating & Error is shown).

d. Connection Handling Test:

When a user closed the tab, the other users should be updated and video broadcast will stop.
Result: ✅ **Pass.**

4.6 Future Work

Despite achieving the primary objective of enabling real-time peer-to-peer communication using WebRTC and SignalR, and adding authentication with JWT tokens, the current implementation can benefit from more features.

a. Call History Logs.

The application does not maintain any history of call, and there is no record of call start/end times, participants, or durations. This limits post-session analysis and user tracking functionalities commonly found in other common communication platforms.

b. Dedicated Meeting Rooms.

The app lacks the concept of virtual "rooms" or session IDs for organizing calls. All users are visible in a shared space without any form of meeting segregation, which restricts scalability for concurrent conversations and does not support invite-only sessions.

c. User Privacy

Currently, all connected users are visible to each other. There are no visibility controls, so user can reach and be reached by anyone which undermines the confidentiality experience.

d. Screen Sharing Capability

The system only supports audio and video streaming between peers. Screen sharing is an essential feature in collaborative and educational contexts; which limits how applicable is the system among others in the same environment.

4.7 Conclusion

The goal of this chapter was to clarify the software architecture by explaining the implementation steps of the development process with some UI screenshots to showcase and backend aspects to explain.

General Conclusion

This graduation project is about implementing a WebRTC-based service application for the goal of understanding how its protocols function and how they interact with each other, and learning the system design principles and improving my knowledge about software architecture by practicing what I learned in the past five years in the computer science class.

I was motivated the most by the opportunity to apply my academic learning in a practical manner, like the object oriented paradigm, software engineering and design using UML, security encryption and authentication, and some UI/UX principle that I'm really bad at, while also exploring new technologies that I hadn't used before, like .NET ecosystem libraries (e.g. SignalR for real time communication and Entity Framework for Object Relational Mapping) which made a technical challenge for me so I think this will help me grow as a developer.

In the end, the project was a technical and personal milestone, it helped me reflect my skills and learn new ones, and it grow my interest more in real-time communication-based software.

References

Reference

Webography

[1] Google WebRTC Overview.

Available: https://webrtc.org/ .

[2] H. Alvestrand. (2011, May) Google release of WebRTC source code.

Available : https://lists.w3.org/Archives/Public/public-webrtc/2011May/0022.html.

[3[Peer-To-Peer Networks: Features, Pros, and Cons - Spiceworks.

Available : https://www.spiceworks.com/tech/networking/articles/what-is-peer-to-peer/.

[4] "Getting Started with WebRTC - HTML5 Rocks," HTML5 Rocks - A resource for open web HTML5 developers.

Available : http://www.html5rocks.com/en/tutorials/webrtc/basics/.

[5] "MediaStream API," Mozilla Developer Network.

Available : https://developer.mozilla.org/en-US/docs/Web/API/Media_Streams_API.

[6] "RTCPeerConnection," Mozilla Developer Network.

Available : https://developer.mozilla.org/en-US/docs/Web/API/RTCPeerConnection.

[7] L. L. Singh Varun, "Basics of WebRTC getStats() API — callstats.io."

Available : http://www.callstats.io/2015/07/06/basics-webrtc-getstats-api/

[8] "WebRTC data channels: WebRTC data channels for high performance data exchange - HTML5 Rocks," HTML5 Rocks - A resource for open web HTML5 developers.

Available : http://www.html5rocks.com/en/tutorials/webrtc/datachannels/

[9] "SDP : Session Description Protocol," RFC4566.

Available: https://datatracker.ietf. org/doc/html/rfc4566.

[10] "RTCIceCandidate," MDN Web Docs.

Available : https://developer.mozilla.org/en-US/docs/Web/API/RTCIceCandidate.

[11] "WebRTC connectivity," MDN Web Docs.

Available : https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Connectivity.

[12] "RFC 4787" Network Address Translation (NAT) Behavioural Requirements for Unicast UDP.

Available : https://www.rfc-editor.org/rfc/rfc4787

[13] "RTCPeerConnection.createAnswer()," MDN Web Docs.

Available : https://developer.mozilla.org/en-US/docs/Web/API/RTCPeerConnection/createAnswer

Reference

[14] “Signaling and video calling,” MDN Web Docs.

Available: https://developer.mozilla.org/en US/docs/Web/API/WebRTC_API/Signaling_and_video_calling

[15] “SIP: Session Initiation Protocol”. Rfc3261, June 2002.

Available: https://datatracker.ietf.org/ doc/html/rfc3261

[16] “An Offer/Answer Model with the Session Description Protocol (SDP)”. Rfc3264, June 2002.

Available: https://datatracker.ietf.org/doc/html/rfc3264

[17] “Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)” RFC 5766.

Available : https://datatracker.ietf.org/doc/html/rfc5766

[18] Object Management Group (OMG). (2017). Unified Modeling Language™ (UML®) Version 2.5.1.

Available: https://www.omg.org/spec/UML/2.5.1/

[19] "UML in Practice" by Petre, 2013, IEEE Pape

Available : https://ieeexplore.ieee.org/document/6606618

[20] StarUML. (2023). StarUML Documentation.

Available : https://staruml.io

[21] UML Use Case Diagram Tutorial | Lucidchart

Available : https://www.lucidchart.com/pages/uml-use-case-diagram

[22] UML for Developing Knowledge Management Systems. By Anthony J. Rhem.

Available: https://www.routledge.com/UML-for-Developing-Knowledge-Management-Systems/Rhem/p/book/9780429209024.

[23] Object Management Group (OMG). (2017). Unified Modeling Language (UML), Version 2.5.1. OMG Document Number formal/2017-12-05.

Available : https://www.omg.org/spec/UML/2.5.1/

[28] Introduction to JSON Web Tokens.

Available: https://jwt.io/introduction

[29] JSON Web Token (JWT) RFC 7519.

Available : https://datatracker.ietf.org/doc/html/rfc7519

[30] "ASP.NET Core SignalR," Microsoft Learn.

Available: https://learn.microsoft.com/en-us/aspnet/core/signalr/introduction

Reference

[31] Microsoft. (n.d.). Introduction to ASP.NET Core.

Available : https://learn.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core

[33] Visualization of a MediaStream Object.

Available: https://dev.w3.org/2011/webrtc/editor/getusermedia-20120813.html.

[34] Microsoft. (n.d.). SQL Server documentation. Microsoft Learn.

Available: https://learn.microsoft.com/en-us/sql/sql-server/

Reference

Bibliography

[24] Larman, C. (2005). “Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development” (3rd ed.). Prentice Hall.

[25] Booch, G., Rumbaugh, J., & Jacobson, I. (2005). *The Unified Modeling Language User Guide* (2nd ed.). Addison-Wesley Professional.

[26] Sommerville, I. (2016). Software Engineering (10th ed.). Pearson.

[27] Fielding, R. T. (2000). Architectural Styles and the Design of Network-based Software Architectures (Doctoral dissertation, University of California, Irvine).

[32] Coronel, C., & Morris, S. (2015). *Database Systems: Design, Implementation, & Management* (11th ed.). Cengage Learning.

Reference

List of Figures :

Reference

Reference

List of Acronyms :

- **VOIP** – *Voice Over Internet Protocol*

Technology that allows voice communication over the Internet.

- **VC** – *Video Conferencing*

Real-time video communication between two or more users over a network.

- **API** – *Application Programming Interface*

A set of rules and tools that allows software applications to communicate with each other.

- **RTC** – *Real-Time Communication*

Technology enabling live audio, video, and data transfer between peers.

- **W3C** – *World Wide Web Consortium*

An international community that develops open web standards.

- **IETF** – *Internet Engineering Task Force*

An organization that develops and promotes internet standards, including those used in WebRTC.

- **S/C** – *Server-to-Client*

Communication flow from the server to the client in a network architecture.

- **P2P** – *Peer-to-Peer*

A decentralized network model where each device (peer) can act as both a client and a server.

- **ICE** – *Interactive Connectivity Establishment*

A framework used in WebRTC to find the best path to connect two peers through NATs and firewalls.

- **NAT** – *Network Address Translation*

A method that remaps one IP address space into another, commonly used in routers to share a single IP.

- **STUN** – *Session Traversal Utilities for NAT*

A protocol used to discover the public IP and port assigned by a NAT device.

- **TURN** – *Traversal Using Relays around NAT*

A protocol that relays media through a server when direct P2P is blocked by NAT/firewalls.

- **SDP** – *Session Description Protocol*

A format for describing multimedia communication sessions, used in WebRTC signaling.

- **HTTP** – *Hyper Text Transfer Protocol*

The protocol used by the World Wide Web to transfer and display web pages.

- **SRTP** – *Secure Real-time Transport Protocol*

A protocol for encrypting and securing RTP streams in real-time communication.

Reference

- **DTLS** – *Datagram Transport Layer Security*
A protocol that provides encryption for datagram-based protocols like SRTP.

- **NAPT** – *Network Address/Port Translation*
A type of NAT that maps multiple private IPs using different port numbers to a single public IP.

- **PAT** – *Port Address Translation*
Another name for NAPT, focusing on the port number remapping aspect.

- **RFC** – *Request for Comments*
Documents that describe standards and protocols for the Internet and software systems.

- **iSAC codec** – *Internet Speech Audio Codec*
A wideband audio codec developed by Google, used for high-quality voice.

- **iLBC codec** – *Internet Low Bitrate Codec*
An audio codec designed for robust speech transmission over unreliable networks.

- **VP8 codec** – *Video Compression Format*
A free video codec developed by Google, commonly used in WebRTC for video streaming.

- **UML** – *Unified Modeling Language*
A standardized modeling language used in software engineering to visualize system design.

- **IT** – *Information Technology*
The use of computers and networks to store, retrieve, and transmit information.

- **JWT** – *JSON Web Token*
A compact, URL-safe means of representing claims to be transferred between two parties (commonly used for authentication).

- **ASP (.NET)** – *Active Server Pages (.NET Framework)*
A web framework developed by Microsoft for building web applications and APIs.

- **EF (EF Core)** – *Entity Framework Core*
A lightweight, extensible, and cross-platform version of Microsoft's ORM (Object-Relational Mapper) for .NET.

- **REST APIs** – *Representational State Transfer APIs*
Web services that follow REST principles, enabling interaction with resources using HTTP methods.

- **HS256** – *HMAC using SHA-256*
A cryptographic algorithm used to sign JWTs (HMAC with SHA-256 hash function).

- **RDMS** – *(Relational) Database Management System*
A system for managing relational databases, using tables with rows and columns.

Reference

- **HTML** – *Hyper Text Markup Language*

The standard language used to create and structure content on the web.

- **CSS** – *Cascading Style Sheets*

A stylesheet language used to describe the visual appearance of HTML elements.