

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет комп'ютерних наук і технологій

(повне найменування факультету)

Кафедра програмних засобів

(повне найменування кафедри)

Пояснювальна записка

до дипломного проєкту (роботи)

бакалавр

(ступінь вищої освіти)

на тему ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ ОБМІНУ
МУЛЬТИМЕДІЙНИМИ ДАНИМ З ВИКОРИСТАННЯМ
ТЕХНОЛОГІЇ WEBRTC
SOFTWARE IMPLEMENTATION OF A MULTIMEDIA DATA
EXCHANGE SYSTEM USING WEBRTC TECHNOLOGY

Виконав(ла): студент(ка) 4 курсу, групи КНТ-112сп

Спеціальності 121 Інженерія програмного

(код і найменування спеціальності)

забезпечення

Освітня програма (спеціалізація)

Інженерія програмного забезпечення

ЦИМБАЛИСТИЙ М.В.

(ПРИЗВИЩЕ та ініціали)

Керівник СТЕПАНЕНКО О.О.

(ПРИЗВИЩЕ та ініціали)

Рецензент ПОЛЯКОВ М.О.

(ПРИЗВИЩЕ та ініціали)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»
(повне найменування закладу вищої освіти)

Факультет КНТ
Кафедра програмних засобів
Ступінь вищої освіти бакалавр
Спеціальність 121 Інженерія програмного забезпечення
(код і найменування)

Освітня програма (спеціалізація) Інженерія програмного забезпечення
(назва освітньої програми (спеціалізації))

ЗАТВЕРДЖУЮ

Завідувач кафедри ПЗ, д.т.н, проф.
Сергій СУББОТІН
“ ” 2025 року

З А В Д А Н Н Я
НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА(КИ)

ЦИМБАЛИСТОГО Максима Володимировича
(ПРІЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Програмна реалізація системи обміну
мультимедійними даним з використанням технології WebRTC.
Software Implementation of a Multimedia Data Exchange System Using WebRTC
Technology.

керівник проєкту (роботи) к.т.н., доцент СТЕПАНЕНКО Олександр Олексійович,
(науковий ступінь, вчене звання, ПРІЗВИЩЕ, ім'я, по батькові)

затверджені наказом закладу вищої освіти від “ 28 ” квітня 2025 року № 209

2. Строк подання студентом проєкту (роботи) 02 червня 2025 року

3. Вихідні дані до проєкту (роботи) рекомендована література, технічне
завдання

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно
розробити) 1. Характеристика об'єкта проєктування. 2. Вибір і обґрунтування
структури системи. 3. Основні рішення щодо реалізації компонентів системи.
4. Керівництво програміста. 5. Керівництво користувача.

5. Перелік графічного матеріалу (з з точним зазначенням обов'язкових креслень,
кількість слайдів, плакатів)
Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	ПРИЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-4 Основна частина	СТЕПАНЕНКО О.О., доцент		
Нормоконтроль	ДЕЙНЕГА Л.Ю., ст. викладач		

7. Дата видачі завдання « 14 » квітня 2025 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Постановка завдання роботи.	1 тиждень	Завдання, ТЗ
2	Аналіз предметної області.	1 тиждень	Розділ 1
3	Вибір мови програмування та інших технологій розробки.	2 тиждень	Розділ 2
4	Розробка архітектури програми.	2 тиждень	Розділ 3
5	Розробка програми.	3-4 тижні	Розділ 3, 4
6	Тестування та експериментальне дослідження програмного забезпечення.	5 тиждень	Розділ 4
7	Оформлення пояснювальної записки та документів до неї.	6 тиждень	Додатки
8	Нормоконтроль та рецензування.	7 тиждень	
9	Захист роботи.	8 тиждень	

Студент(ка)

(підпис) Максим ЦИМБАЛИСТИЙ
(Ім'я ПРИЗВИЩЕ)

Керівник проєкту (роботи)

(підпис) Олександр СТЕПАНЕНКО
(Ім'я ПРИЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи бакалавра: 158 с., 10 табл., 29 рис., 3 дод., 17 джерел.

ВІДЕОЗВ'ЯЗОК, МУЛЬТИМЕДІА, WEBRTC, PEER-TO-PEER, ТЕКСТОВИЙ ЧАТ, ДЕМОНСТРАЦІЯ ЕКРАНУ, REACT.JS, TYPESCRIPT, NODE.JS, WEBSOCKET, БРАУЗЕР, СИГНАЛІЗАЦІЯ.

Об'єкт дослідження – процес розроблення програмного забезпечення для обміну мультимедійними даними в реальному часі.

Предмет дослідження – методи та програмні засоби для реалізації peer-to-peer відеозв'язку з використанням технології WebRTC.

Мета роботи – підвищення ефективності обміну мультимедійними даними шляхом розробки веб-застосунку на основі технології WebRTC, що забезпечує передачу відео, аудіо та текстових повідомлень між користувачами в режимі реального часу.

Система обміну мультимедійними даними автоматизує процес встановлення peer-to-peer з'єднань між користувачами, дозволяє здійснювати відеодзвінки, обмінюватися текстовими повідомленнями, демонструвати екран та відтворювати відеофайли для інших учасників конференції.

Вхідними даними для застосунку є відеопотік з камери користувача, аудіопотік з мікрофону, текстові повідомлення та відеофайли для демонстрації.

Вихідними даними для застосунку є відображення відеопотоків всіх учасників конференції, відтворення аудіо, історія текстових повідомлень та статус підключення користувачів.

Система коректно працює у всіх сучасних браузерах, таких як Google Chrome, Microsoft Edge та Opera, які підтримують технологію WebRTC.

ABSTRACT

Explanatory note to the diploma qualifying work of the bachelor: 158 pages, 10 tables, 29 figures, 3 appendixes, 17 sources.

VIDEO COMMUNICATION, MULTIMEDIA, WEBRTC, PEER-TO-PEER, TEXT CHAT, SCREEN SHARING, REACT.JS, TYPESCRIPT, NODE.JS, WEBSOCKET, BROWSER, SIGNALING.

The object of study is the process of developing software for real-time multimedia data exchange.

The subject of study is methods and software tools for implementing peer-to-peer video communication using WebRTC technology.

The purpose of the work is to increase the efficiency of multimedia data exchange by developing a web application based on WebRTC technology, which provides the transmission of video, audio and text messages between users in real time.

The multimedia data exchange system automates the process of establishing peer-to-peer connections between users, allows you to make video calls, exchange text messages, show the screen and play video files for other conference participants.

The input data for the application is the video stream from the user's camera, the audio stream from the microphone, text messages and video files for demonstration.

The output data for the application is the display of video streams of all conference participants, audio playback, text message history and user connection status.

The system works correctly in all modern browsers, such as Google Chrome, Microsoft Edge and Opera, which support WebRTC technology.

ЗМІСТ

	С.
Перелік скорочень та умовних позначок	9
1 Характеристика об'єкта проектування	12
1.1 Поняття систем обміну мультимедійними даними в реальному часі ...	12
1.2 Основні функціональні можливості WebRTC	13
1.3 Аналіз аналогічних рішень	15
1.3.1 Zoom	15
1.3.2 Google Meet.....	17
1.3.3 Discord	19
1.3.4 Підсумки аналізу аналогічних рішень	21
1.4 Висновки за розділом	23
2 Вибір і обґрунтування структури системи	25
2.1 Аналіз сучасних фреймворків для розробки веб-застосунків.....	25
2.1.1 JavaScript/TypeScript фреймворки	26
2.1.1.1 Next.js	26
2.1.1.2 Remix	28
2.1.2 Python фреймворки	30
2.1.2.1 Django	30
2.1.2.2 FastAPI.....	32
2.1.3 Ruby фреймворки	34
2.1.3.1 Ruby on Rails.....	34
2.1.4 Порівняльний аналіз фреймворків для WebRTC застосунків	35
2.1.5 Підсумки та обґрунтування вибору Next.js.....	36
2.2 Аналіз використаних інструментів розробки	38
2.2.2 Інструменти клієнтської частини	38
2.2.2.1 Бібліотека React як основа клієнтської частини	38
2.2.2.3 Shadcn/ui система для побудови бібліотеки компонентів	39
2.2.2.4 TailwindCSS: стилізація компонентів	41
2.2.3 Серверні технології.....	42

2.2.3.1 Node.js: серверне середовище.....	42
2.2.3.2 WebSocket: двостороння комунікація.....	44
2.2.4 Інструменти для роботи з WebRTC	45
2.2.4.1 MediaDevices API	46
2.2.4.2 RTCPeerConnection	47
2.2.4.3 MediaStream API.....	49
2.3 Архітектура системи.....	51
2.3.1 Загальна архітектура.....	51
2.3.2 Клієнт-серверна взаємодія	53
2.3.3 Peer-to-peer комунікація	56
2.4 Інформаційні потоки між компонентами	57
2.4.1 Потоки сигналізації.....	57
2.4.2 Медіа потоки.....	59
2.4.3 Потоки даних	61
2.5 Функціональна схема роботи відеоконференції.....	61
2.5.1 Процес встановлення з'єднання.....	63
2.5 Висновки за розділом	65
3 Основні рішення щодо реалізації компонентів системи.....	66
3.1 Структура проекту	66
3.1.1 Файлова організація Next.js додатку.....	66
3.1.2 Основні компоненти системи	68
3.1.3 Взаємодія між компонентами	69
3.2 Основні алгоритми роботи застосунку	72
3.2.1 Алгоритм створення нової кімнати та приєднання учасників	72
3.2.2 Алгоритм встановлення WebRTC з'єднання між учасниками	74
3.2.3 Алгоритм обміну медіа-потоками між користувачами.....	75
3.2.4 Алгоритм роботи текстового чату через DataChannel	76
3.2.5 Алгоритм демонстрації екрану користувача.....	77
3.2.6 Алгоритм відтворення відеофайлів у конференції.....	78
3.2.7 Алгоритм керування аудіо та відео потоками	79

3.2.8 Алгоритм обробки від'єднання учасників	80
3.3 Реалізація компонентів користувацького інтерфейсу	81
3.3.1 Компонент входу в кімнату та створення ідентифікатора	81
3.3.2 Компонент відеоконференції та розміщення учасників	82
3.3.3 Компоненти керування медіа-можливостями	83
3.4 Реалізація користувацьких хуків	84
3.4.1 Хук useWebRTC та його функціональність	84
3.4.2 Хук useStateWithCallback для управління станом	88
3.5 Реалізація WebSocket взаємодії	89
3.5.1 Види WebSocket повідомлень	89
3.5.2 Управління учасниками кімнати	89
3.6 Висновки за розділом	90
4 Керівництво програміста	92
4.1 Призначення та умови використання системи	92
4.2 Характеристики розробленого веб-застосунку	93
4.3 Структура WebRTC з'єднань	95
4.4 Структурна схема компонентів системи	96
4.5 Висновки за розділом	98
5 Керівництво користувача	100
5.1 Призначення та можливості системи	100
5.2 Основні сценарії роботи з системою	101
5.3 Вимоги до системи та браузера	105
5.4 Висновки за розділом	107
Висновки	108
Перелік джерел посилання	110
Додаток А Текст програми	112
Додаток Б Текст програми	116
Додаток В Слайди презентації	152

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

API	– Application Programming Interface;
CRM	– Customer Relationship Management;
DOM	– Document Object Model;
DTLS	– Datagram Transport Layer Security;
ICE	– Interactive Connectivity Establishment;
ID	– Ідентифікатор;
IP	– Internet Protocol;
JSX	– JavaScript XML;
NAT	– Network Address Translation;
P2P	– Peer-to-peer;
QoS	– Quality of Service;
RAM	– Random Access Memory;
RTP	– Real-time Transport Protocol;
RTCP	– Real-time Transport Control Protocol;
SCTP	– Stream Control Transmission Protocol;
SDP	– Session Description Protocol;
SRTP	– Secure Real-time Transport Protocol;
STUN	– Session Traversal Utilities for NAT;
UI	– User Interface;
WebRTC	– Web Real-Time Communication ;

ВСТУП

У сучасному світі відеозв'язок став невід'ємною частиною нашого повсякденного життя. Пандемія COVID-19 значно прискорила перехід до віддаленої роботи та онлайн-комунікацій, зробивши відеоконференції основним інструментом для проведення ділових зустрічей, навчання та особистого спілкування. Проте традиційні системи відеозв'язку часто вимагають встановлення додаткового програмного забезпечення, мають обмежену функціональність або залежать від централізованих серверів, що може призводити до затримок та проблем з якістю зв'язку.

Технологія WebRTC (Web Real-Time Communication) відкрила нові можливості для створення систем відеозв'язку безпосередньо в веб-браузері, без необхідності встановлення додаткового програмного забезпечення. Ця технологія дозволяє встановлювати прямі peer-to-peer з'єднання між користувачами, що забезпечує високу якість передачі аудіо та відео даних з мінімальними затримками.

Метою цієї роботи є розробка сучасної системи обміну мультимедійними даними на основі технології WebRTC, яка дозволить користувачам здійснювати відеодзвінки, обмінюватися текстовими повідомленнями та демонструвати екран безпосередньо через веб-браузер. Особлива увага приділяється створенню зручного та інтуїтивно зрозумілого інтерфейсу, який дозволить користувачам легко долучатися до відеоконференцій та використовувати всі можливості системи.

У процесі розробки використовуються сучасні веб-технології, такі як React.js та TypeScript для створення динамічного користувацького інтерфейсу, Node.js для серверної частини та WebSocket для організації сигнального обміну між користувачами.

Результатом роботи є повнофункціональна система відеозв'язку, яка може бути використана для проведення онлайн-конференцій, дистанційного навчання, віддаленої роботи та особистого спілкування. Система працює у

сучасних веб-браузерах та не потребує встановлення додаткового програмного забезпечення, що робить її доступною для широкого кола користувачів.

1 ХАРАКТЕРИСТИКА ОБ'ЄКТА ПРОЄКТУВАННЯ

1.1 Поняття систем обміну мультимедійними даними в реальному часі

Системи обміну мультимедійними даними в реальному часі є фундаментальною складовою сучасних комунікаційних технологій. Такі системи забезпечують передачу аудіо, відео та інших типів даних між користувачами з мінімальною затримкою, що створює ефект безпосередньої взаємодії.

В основі роботи таких систем лежить концепція реального часу, що означає передачу даних з настільки малою затримкою, що користувачі сприймають комунікацію як миттєву. Для голосового зв'язку, наприклад, прийнятна затримка не повинна перевищувати 150-200 мілісекунд, інакше розмова стає незручною через помітні паузи між репліками співрозмовників.

Ключовою технологією для реалізації таких систем є протокол RTP (Real-time Transport Protocol), який забезпечує передачу мультимедійних даних через мережу Інтернет. RTP працює разом з протоколом RTCP (RTP Control Protocol), що відповідає за моніторинг якості передачі та обмін службовою інформацією між учасниками сеансу зв'язку.

Архітектура типової системи обміну мультимедійними даними в реальному часі включає наступні основні компоненти:

- клієнтські пристрої з програмним забезпеченням для захоплення та відтворення медіаданих;
- мережна інфраструктура для передачі даних;
- сервери для координації з'єднань між користувачами;
- підсистеми обробки медіапотоків (кодування, декодування, синхронізація).

Особливістю таких систем є необхідність забезпечення стабільної якості обслуговування (QoS) навіть в умовах нестабільної роботи мережі. Для цього використовуються спеціальні механізми:

- буферизація даних для згладжування мережних затримок;
- адаптивне кодування для оптимізації якості під доступну пропускну здатність;
- виявлення та компенсація втрачених пакетів даних;
- синхронізація різних медіапотоків (наприклад, аудіо та відео).

На практиці системи обміну мультимедійними даними в реальному часі знаходять широке застосування в таких сферах:

- відеоконференцзв'язок для бізнес-комунікацій;
- дистанційне навчання та телемедицина;
- онлайн-трансляції подій;
- системи IP-телефонії;
- інтерактивні розважальні сервіси.

Важливо розуміти, що поняття "реального часу" в контексті мультимедійних систем є відносним і залежить від конкретного застосування. Для відеоконференцій критичною є мінімальна затримка, тоді як для потокового відео допустимі більші затримки, але важливіша стабільність потоку даних.

Сучасні системи обміну мультимедійними даними активно розвиваються в напрямку підвищення якості передачі (підтримка форматів високої роздільної здатності, покращення компресії), розширення функціональності (інтеграція з іншими сервісами) та оптимізації використання мережних ресурсів.

1.2 Основні функціональні можливості WebRTC

WebRTC (Web Real-Time Communication) — це відкритий стандарт, що дозволяє веб-додаткам і сайтам здійснювати прямий peer-to-peer зв'язок у реальному часі без потреби встановлення додаткових плагінів або програм. Ця технологія значно спрощує розробку комунікаційних рішень, інтегруючи

можливості голосових, відеодзвінків та обміну даними безпосередньо у веб-браузер.

WebRTC є фундаментом для створення високоякісних відео- та аудіодзвінків безпосередньо у браузері. Він забезпечує захоплення аудіо та відео з пристроїв користувача (камера, мікрофон), кодування та передачу цих даних між учасниками. Завдяки оптимізованим алгоритмам обробки, WebRTC мінімізує затримки та забезпечує плавну передачу потоків, що є критично важливим для інтерактивних комунікацій. Це дозволяє реалізувати різноманітні сценарії, такі як відеоконференції, вебінари, онлайн-навчання та телемедицина.

Крім аудіо- та відеопотоків, WebRTC надає можливість передачі довільних даних між браузерами. Ці канали даних можуть використовуватися для обміну текстовими повідомленнями, файлами, ігровими даними, інформацією про стан додатку або будь-якими іншими даними, які не є медіа. Data Channels працюють за протоколами SCTP (Stream Control Transmission Protocol) поверх DTLS (Datagram Transport Layer Security), що забезпечує надійну, упорядковану або ненадійну, неупорядковану доставку даних відповідно до потреб програми. Ця функціональність розширює можливості WebRTC далеко за межі простих відеодзвінків, дозволяючи створювати повноцінні інтерактивні веб-додатки.

Однією з ключових особливостей WebRTC є його здатність встановлювати пряме з'єднання між двома браузерами (peer-to-peer) без участі центрального сервера для передачі медіа або даних. Це значно зменшує навантаження на сервери та покращує продуктивність, оскільки дані передаються безпосередньо між учасниками. Хоча для встановлення з'єднання (сигналінгу) та обходу NAT/фаєрволів (через STUN/TURN сервери) центральний сервер все ж таки потрібен, після встановлення зв'язку медіа-потоки та дані йдуть безпосередньо. Це підвищує конфіденційність та безпеку, оскільки дані не проходять через треті сторони [1].

WebRTC розроблено з урахуванням мінливих мережних умов. Він автоматично адаптується до пропускної здатності мережі, динамічно регулюючи якість відео та аудіо, щоб забезпечити найкращий можливий досвід комунікації. Це включає в себе використання різних кодеків, зміну бітрейту та застосування алгоритмів компенсації втрати пакетів. Така адаптивність забезпечує стабільну роботу додатків WebRTC навіть у складних мережних середовищах, таких як мобільні мережі або мережі з високою затримкою.

Безпека є невід'ємною частиною WebRTC. Усі комунікації, що проходять через WebRTC, зашифровані за замовчуванням за допомогою DTLS (Datagram Transport Layer Security) для каналів даних та SRTP (Secure Real-time Transport Protocol) для медіа-потоків. Це гарантує, що дані, які передаються між користувачами, залишаються приватними та захищеними від несанкціонованого доступу. Крім того, перед тим, як веб-додаток зможе отримати доступ до камери або мікрофона користувача, браузер завжди вимагає явного дозволу, що підвищує конфіденційність користувача.

1.3 Аналіз аналогічних рішень

1.3.1 Zoom

Zoom – це хмарна платформа для відеоконференцій, що пропонує широкий спектр функцій для онлайн-зустрічей, вебінарів та спілкування. Вона здобула величезну популярність завдяки своїй простоті використання та надійності, особливо під час пандемії. Zoom дозволяє проводити зустрічі з великою кількістю учасників, ділитися екраном, записувати зустрічі, використовувати віртуальні фони та багато іншого [2].

Інтерфейс застосунку Zoom зображено на рисунку 1.1.

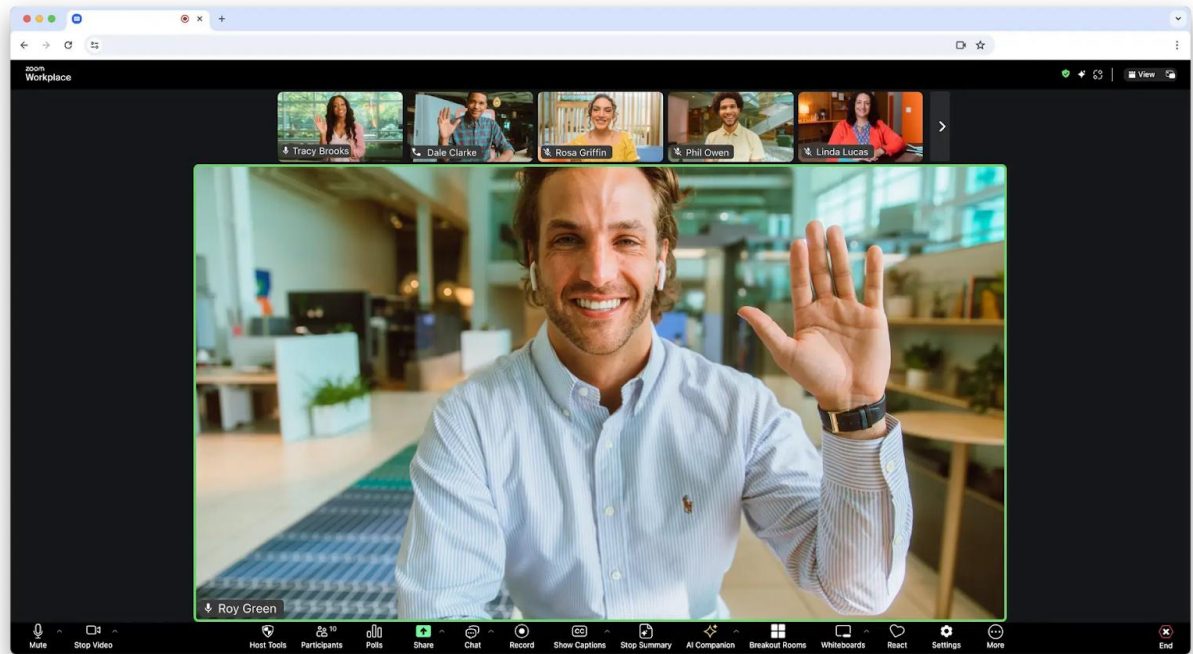


Рисунок 1.1 — Інтерфейс Zoom під час конференції [3]

Особливості:

- Zoom відомий своєю здатністю підтримувати стабільну та якісну передачу відео та аудіо навіть при великій кількості учасників;
- підтримка до 100 учасників на безкоштовному плані та до 500+ на платних;
- можливості спільного доступ до екрану, інтерактивну дошку, анотації, кімнати для обговорень (breakout rooms);
- можливість записувати зустрічі в хмару або локально з автоматичною транскрипцією для платних планів;
- широка підтримка інтеграцій з календарями (Google Calendar, Outlook), CRM-системами та іншими бізнес-інструментами;
- інтуїтивно зрозумілий інтерфейс і просте підключення до зустрічей, навіть для нових користувачів.

Переваги:

- стабільне з'єднання та високу якість зв'язку, що є критично важливим для професійних зустрічей;

- велика кількість функцій, які задовольняють потреби різних користувачів, від індивідуальних дзвінків до масштабних вебінарів;
- безкоштовний план, що дозволяє проводити зустрічі до 40 хвилин з 100 учасниками, що є достатнім для багатьох завдань;
- широке поширення, багато користувачів вже знайомі з Zoom, що спрощує взаємодію.

Недоліки:

- 40-хвилинне обмеження для групових зустрічей на безкоштовному плані може бути незручним;
- для повноцінної роботи часто вимагає встановлення десктопного або мобільного клієнта, хоча є можливість приєднатися через веб-браузер;
- проблеми з безпекою та конфіденційністю хоча компанія активно працює над їх усуненням;
- високоякісне відео та аудіо можуть споживати значну пропускну здатність інтернету [3].

1.3.2 Google Meet

Google Meet – це сервіс для відеозв'язку, розроблений Google, який є частиною пакету Google Workspace. Він призначений для проведення онлайн-зустрічей, уроків та вебінарів, пропонуючи інтеграцію з іншими сервісами Google, такими як Google Календар та Gmail. Google Meet забезпечує зручне та безпечне спілкування в браузері без необхідності встановлення додаткового програмного забезпечення [4].

Інтерфейс застосунку Google Meet зображено на рисунку 1.2.

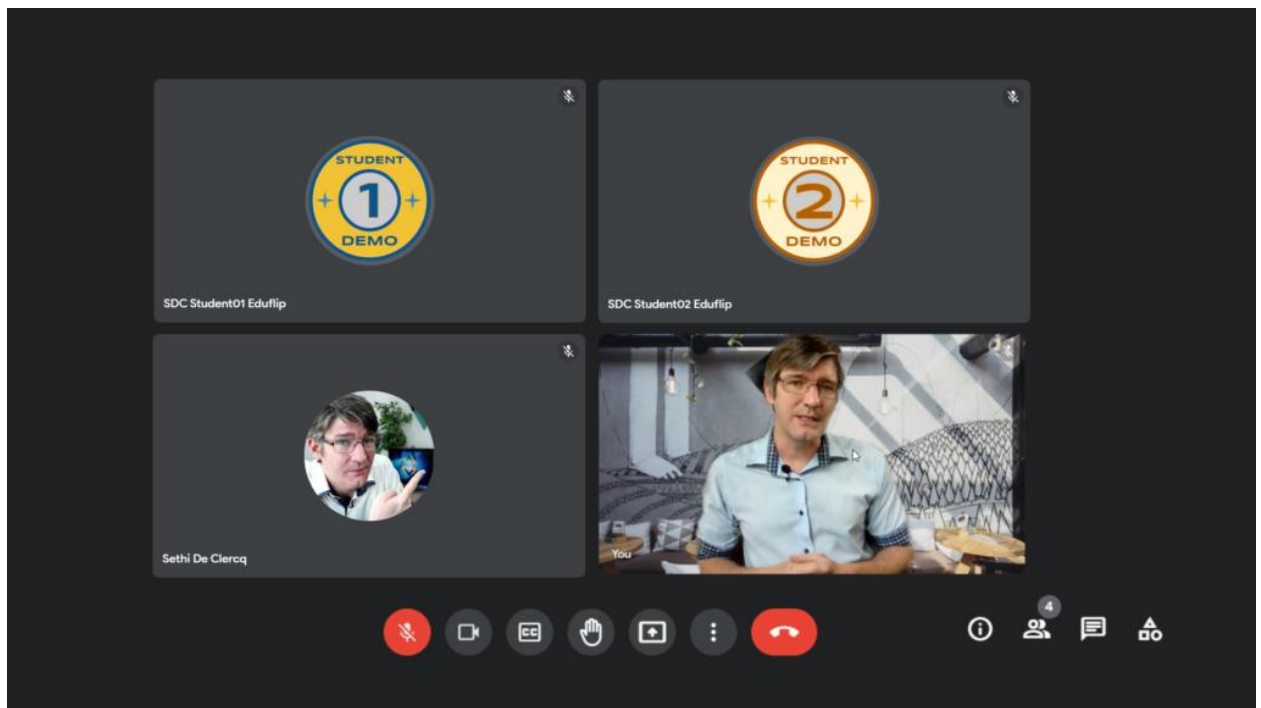


Рисунок 1.2 — Інтерфейс Google Meet під час конференції [4]

Особливості:

- інтеграція з Google Workspace;
- можливість приєднуватися до зустрічей без встановлення клієнта, безпосередньо через веб-браузер;
- підтримка високої якості відео (до 1080p на платних планах) та аудіо, з функцією шумозаглушення;
- автоматичне генерування субтитрів у реальному часі, що підвищує доступність;
- можливості спільного доступу до екрану, інтеграція з Google Jamboard для цифрової дошки, опитування та питання-відповіді;
- механізми шифрування дзвінків, керування доступом до зустрічей та надійні заходи безпеки.

Переваги:

- легкість у використанні, особливо для тих, хто вже користується продуктами Google;
- можливість приєднатися до зустрічі прямо з браузера, що знижує бар'єр входу для нових користувачів;

- інтеграція з Google, що зручно для компаній та освітніх закладів, які вже використовують Google Workspace;
- безкоштовний план, що дозволяє проводити зустрічі до 60 хвилин з 100 учасниками для особистих облікових записів Google.

Недоліки:

- обмежений функціонал безкоштовного плану;
- деякі користувачі відзначають меншу гнучкість у налаштуванні розташування екрану та управлінні учасниками порівняно з іншими платформами;
- проблеми з відображенням презентацій або порожній екран при спільному використанні певних додатків згідно повідомлень деяких користувачів [4].

1.3.3 Discord

Discord – це безкоштовна платформа для голосового, відео- та текстового спілкування, орієнтована на створення спільнот, які називаються серверами. Спочатку розроблений для геймерів, Discord еволюціонував у багатофункціональний інструмент для різноманітних спільнот, включаючи освітні, робочі та соціальні групи. Він пропонує гнучку структуру каналів, можливість створювати голосові та відеодзвінки, обмінюватися файлами та інтегрувати ботів для автоматизації [5].

Інтерфейс застосунку Discord зображено на рисунку 1.3.

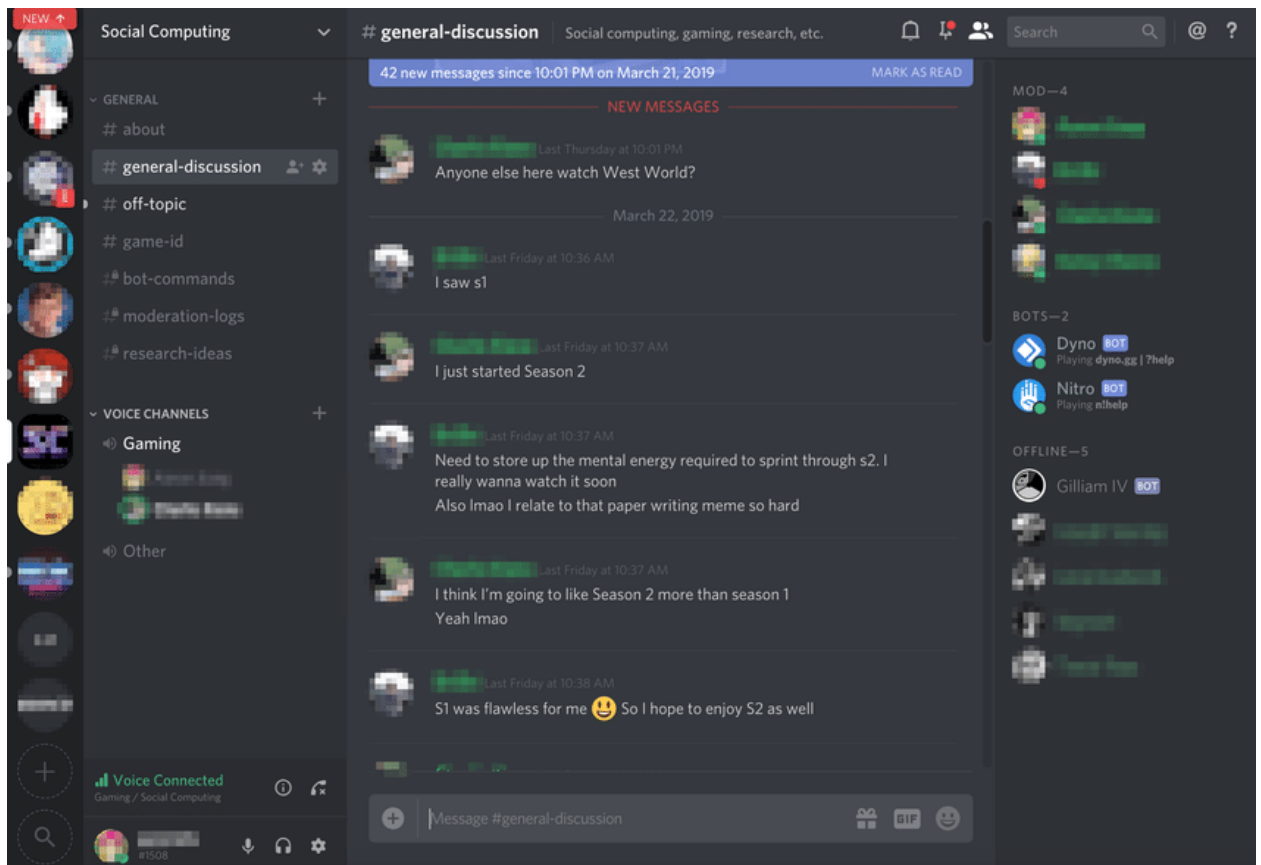


Рисунок 1.3 — Інтерфейс сторінки користувацького серверу Discord [6]

Особливості:

- можливість створювати власні сервери з різними текстовими та голосовими каналами для організації розмов за темами;
- якісний голосовий зв'язок (використовує кодек Opus) та підтримка відеодзвінків, включаючи групові та індивідуальні;
- можливість транслювати екран або конкретний додаток іншим учасникам, що особливо популярно серед геймерів та для демонстрацій;
- широкі можливості для додавання користувацьких ботів, які можуть автоматизувати завдання, керувати модерацією, відтворювати музику та багато іншого;
- доступність на настільних комп'ютерах (Windows, macOS, Linux), мобільних пристроях (iOS, Android) та у веб-браузері;

- платна підписка Discord Nitro, що надає додаткові переваги, такі як збільшений ліміт завантаження файлів, HD-стрімінг та користувацькі емодзі.

Переваги:

- ідеально підходить для створення та управління великими та активними онлайн-спільнотами завдяки гнучкій структурі каналів та модуерації;
- забезпечує чітке та низьколатентне голосове спілкування;
- багато ключових функцій доступні безкоштовно, що робить його привабливим для широкого кола користувачів;
- дозволяє створювати користувацькі ролі, дозволи, ботів та багато іншого для адаптації під конкретні потреби спільноти.

Недоліки:

- для нових користувачів Discord може здатися складним через велику кількість функцій та опцій налаштування;
- у великих публічних серверах можуть виникати проблеми з модуерацією небажаного контенту або поведінки;
- деякі користувачі висловлюють занепокоєння щодо збору даних та використання інформації Discord;
- безкоштовний план має обмеження на розмір файлів, що завантажуються, що може бути незручним для обміну великими медіафайлами [7].

1.3.4 Підсумки аналізу аналогічних рішень

Проаналізовані платформи – Zoom, Google Meet та Discord – демонструють різні підходи до реалізації систем обміну мультимедійними даними, кожна з яких має свої сильні та слабкі сторони. Zoom вирізняється своєю надійністю та широким набором професійних функцій для відеоконференцій, що робить його ідеальним для бізнес-середовища та освіти.

Google Meet, інтегрований у екосистему Google Workspace, є чудовим рішенням для команд, які вже використовують продукти Google, пропонуючи простоту використання та високий рівень безпеки через веб-інтерфейс. Discord, з іншого боку, фокусується на побудові спільнот, надаючи розширені можливості для голосового спілкування, ігрових стрімів та гнучку систему каналів, що робить його популярним серед геймерів та онлайн-спільнот. Вибір конкретного рішення залежить від цільової аудиторії та специфічних вимог до функціоналу, масштабованості та інтеграції. Результати порівняння наведені у таблиці 1.1.

Таблиця 1.1 – Порівняльна таблиця розглянутих рішень

Характеристика	Zoom	Google Meet	Discord
Основне призначення	Професійні відеоконференції, вебінари	Онлайн-зустрічі, інтеграція з Google Workspace	Спілкування в спільнотах, голосовий чат, геймінг
Цільова аудиторія	Бізнес, освіта, великі групи	Корпоративні користувачі Google, освіта	Геймери, онлайн-спільноти, друзі
Встановлення клієнта	Часто вимагається (для повного функціоналу)	Веб-орієнтований, не завжди потрібен клієнт	Рекомендовано (для повного функціоналу)
Якість аудіо/відео	Висока, стабільна	Висока, з шумозаглушенням	Висока (особливо голосовий), низька затримка
Кількість учасників	До 100 (безкоштовно), 500+ (платно)	До 100 (безкоштовно), 500+ (платно)	До 10 (відео), до 999 (голосові)
Інтеграції	Календарі, CRM, багато сторонніх додатків	Google Workspace (Gmail, Calendar, Drive)	Боти, ігрові платформи
Можливості спільної роботи	Спільний доступ до екрану, дошка, анотації, кімнати для обговорень	Спільний доступ до екрану, Jamboard, опитування	Спільний доступ до екрану (Go Live), трансляції

Продовження таблиці 1.1

Характеристика	Zoom	Google Meet	Discord
Безкоштовний план	Обмеження 40 хв для групових дзвінків	Обмеження 60 хв для групових дзвінків	Більшість функцій доступні безкоштовно
Особливості безпеки	Шифрування, керування доступом	Шифрування, захист від несанкціонованого доступу	Гнучкі налаштування дозволів на серверах
Основні переваги	Надійність, функціональність, простота	Інтеграція з Google, веб-орієнтованість, безпека	Орієнтованість на спільноти, гнучкість, боти
Основні недоліки	Обмеження безкоштовного плану, питання безпеки (раніше)	Обмежений функціонал безкоштовного плану	Крива навчання, питання модерації (для великих спільнот)

1.4 Висновки за розділом

У цьому розділі було проведено детальний аналіз сучасних систем обміну мультимедійними даними, таких як Zoom, Google Meet та Discord. Кожна з цих платформ пропонує унікальний набір функцій, переваг та недоліків, що робить їх придатними для різних сценаріїв використання.

Zoom виявився лідером у сфері професійних відеоконференцій, забезпечуючи високу якість зв'язку, широкий спектр інструментів для спільної роботи та масштабованість для великих груп. Його переваги включають надійність та зручність, хоча обмеження безкоштовного плану та попередні проблеми з безпекою є певними недоліками.

Google Meet, як частина екосистеми Google Workspace, демонструє відмінну інтеграцію з іншими сервісами Google, простоту використання через веб-інтерфейс та надійні заходи безпеки. Він ідеально підходить для компаній та освітніх установ, які вже використовують продукти Google, але його безкоштовний функціонал є дещо обмеженим порівняно з платними версіями.

Discord вирізняється своєю орієнтованістю на створення спільнот, пропонуючи гнучку систему серверів та каналів, високоякісний голосовий зв'язок та широкі можливості інтеграції ботів. Він є популярним вибором серед геймерів та онлайн-спільнот, проте може мати вищу криву навчання для нових користувачів та потенційні виклики з модерацією у великих спільнотах.

Загалом, аналіз показав, що ринок систем обміну мультимедійними даними є насиченим і динамічним, з різноманіттям рішень, які задовольняють специфічні потреби користувачів. Розуміння сильних та слабких сторін цих платформ є ключовим для подальшої розробки власної системи. Це дозволить врахувати найкращі практики, уникнути типових помилок та зосередитися на створенні унікальних переваг з використанням технології WebRTC для ефективного обміну мультимедійними даними.

2 ВИБІР І ОБҐРУНТУВАННЯ СТРУКТУРИ СИСТЕМИ

2.1 Аналіз сучасних фреймворків для розробки веб-застосунків

Вибір правильного фреймворку — одне з найважливіших рішень на початкових етапах розробки будь-якого веб-застосунку. Це особливо стосується систем обміну мультимедійними даними з використанням технології WebRTC. Величезне розмаїття доступних фреймворків, мов програмування та архітектурних підходів вимагає ретельного аналізу для забезпечення високої продуктивності, масштабованості та зручності розробки. Для цієї системи критично важливим є забезпечення ефективного зв'язку в реальному часі. Це вимагає як потужного та інтерактивного клієнтського інтерфейсу, здатного ефективно працювати з медіапотоками, так і надійного сигнального сервера, що забезпечує встановлення та управління peer-to-peer з'єднаннями.

Основними фреймворками, які розглядаються для цієї задачі, є ті, що представляють різні екосистеми програмування: JavaScript/TypeScript, Python та Ruby. Кожен з них має унікальні характеристики та переваги, які роблять його придатним для певних аспектів веб-розробки. Ретельна оцінка цих фреймворків дозволяє виявити їхні сильні сторони у контексті розробки фронтенду, бекенду та, зокрема, підтримки протоколів реального часу, таких як WebSockets, що є невід'ємною частиною архітектури WebRTC. Важливим є аналіз здатності кожного фреймворку ефективно керувати даними, забезпечувати безпеку та масштабованість, а також легкість інтеграції з WebRTC API. Результатом цього процесу є обґрунтування вибору фреймворку, який оптимально підходить для архітектури та функціональних вимог системи обміну мультимедійними даними.

2.1.1 JavaScript/TypeScript фреймворки

2.1.1.1 Next.js

Next.js — це універсальний React-фреймворк для розробки веб-застосунків, який пропонує рішення для серверного рендерингу (SSR), статичної генерації сайтів (SSG) та інкрементальної статичної регенерації (ISR). Він побудований на базі React і розроблений для максимальної продуктивності та зручності розробки.

Ключовими особливостям Next.js є серверний рендеринг, статична генерація сайтів, інкрементальна статична регенерація. Розглянемо ці поняття детальніше [8].

Серверний рендеринг (SSR) дозволяє відправляти повністю відрендерену HTML-сторінку на клієнт, що покращує початкове завантаження та SEO. Це особливо важливо для застосунків, де вміст має бути доступним пошуковим системам одразу. Прикладом може бути застосунок для електронної комерції, де кожна сторінка продукту має бути проіндексована, або ж новинний портал, де швидкість завантаження контенту критична для користувацького досвіду. Для WebRTC-застосунків, SSR може не бути критично важливим для самих медіапотоків, але він може значно покращити завантаження інтерфейсу користувача та початкової сторінки, де користувач може обирати кімнату або налаштовувати параметри.

Статична генерація сайтів (SSG) дає змогу генерувати HTML-файли під час збирання проєкту, що забезпечує надзвичайно швидке завантаження та високу продуктивність, оскільки статичні файли можуть бути розгорнуті на CDN. Це ідеально підходить для блогів, документації або будь-яких сайтів зі статичним вмістом. Хоча WebRTC-застосунки є динамічними за своєю природою, SSG може використовуватися для сторінок "про нас", "контакти" або посібників користувача, які не змінюються часто.

Інкрементальна статична регенерація (ISR) комбінує переваги SSG та SSR, дозволяючи оновлювати статичні сторінки у фоновому режимі після першого запиту, не вимагаючи повного перезбирання проєкту. Це корисно для контенту, який оновлюється періодично, але не потребує оновлення в реальному часі.

Також особливістю Next.js є автоматичне розбиття коду на дрібніші частини, які завантажуються лише тоді, коли вони потрібні. Це зменшує початковий розмір JavaScript-бандлу та прискорює завантаження сторінок.

Next.js має систему маршрутизації на основі файлової системи (File-system based routing). Фреймворк створює маршрути автоматично, базуючись на структурі файлів та папок в директорії `pages`. Це спрощує організацію проєкту та робить маршрутизацію інтуїтивно зрозумілою.

Для WebRTC-застосунків Next.js може бути чудовим вибором для інтерфейсу користувача (UI) та сигналізації. Хоча сам WebRTC працює за принципом peer-to-peer, йому потрібен сигнальний сервер для встановлення з'єднання між учасниками. Next.js може слугувати як фронтендом для клієнтського інтерфейсу (відео, аудіо, чат), так і бекендом для обробки сигналізації (через API Routes або WebSocket-сервер, інтегрований з Next.js).

Крім того, Next.js, інтегрований з бібліотеками UI, такими як React, дозволяє розробити привабливий та функціональний інтерфейс, який швидко завантажувється та забезпечує плавну взаємодію з користувачем. Його можливості з оптимізації та розбиття коду гарантують, що застосунок буде швидким та ефективним, навіть якщо в ньому буде багато інтерактивних елементів. Завдяки TypeScript, розробка стає більш безпечною та передбачуваною, що є важливим для складних застосунків, таких як системи обміну мультимедійними даними. Next.js забезпечує міцну основу для побудови масштабованих та високопродуктивних WebRTC-застосунків.

2.1.1.2 Remix

Remix — це сучасний веб-фреймворк, побудований на базі React, який фокусується на продуктивності, надійності та покращенні досвіду розробки через вбудовані механізми для роботи з даними та оптимізацію для веб-стандартів. На відміну від деяких інших фреймворків, Remix активно використовує нативні можливості вебу, такі як форми HTML та HTTP-кешування, для забезпечення високої швидкості та стійкості.

Remix активно використовує базові можливості вебу, такі як HTML-форми, HTTP-кешування та стандартні заголовки. Це призводить до більш передбачуваної та надійної поведінки застосунків, а також спрощує їх кешування та розширення. Замість того, щоб покладатися на клієнтський JavaScript для обробки всіх форм, Remix використовує нативну поведінку браузера, що дозволяє застосунку працювати навіть без JavaScript або при його повільному завантаженні [9].

Вбудовані механізми для роботи з даними (Loaders and Actions):

Remix пропонує потужний підхід до завантаження та мутації даних.

Loaders: Це функції, які виконуються на сервері (або під час статичної генерації) перед рендерингом компонента. Вони відповідають за отримання даних, необхідних для поточної сторінки. Наприклад, для WebRTC-застосунку, Loader може отримати інформацію про доступні відеокімнати або поточні налаштування користувача.

Actions: Це функції, які також виконуються на сервері і призначені для обробки HTTP-запитів POST (наприклад, надсилання форми). Вони дозволяють обробляти дані, отримані від користувача, та виконувати операції запису (наприклад, створення нової кімнати, оновлення профілю користувача).

Нестінгова маршрутизація (Nested Routing): Remix дозволяє створювати вкладені маршрути, де окремі частини інтерфейсу можуть завантажувати власні дані та рендеритися незалежно. Це забезпечує більш

гранульоване управління завантаженням та відображенням контенту, що особливо корисно для складних інтерфейсів. Наприклад, в WebRTC-застосунку, ви можете мати маршрут для загальної сторінки конференції, а всередині нього — вкладені маршрути для чату, списку учасників та налаштувань.

Remix має вбудовані механізми для елегантної обробки помилок як на сервері, так і на клієнті. Це допомагає запобігти "білому екрану смерті" та забезпечити кращий користувацький досвід, відображаючи змістовні повідомлення про помилки.

Remix оптимізує застосунки для швидкого відгуку та інтерактивності. Він автоматично обробляє стани завантаження та повторні спроби надсилання даних, що призводить до більш плавного та надійного користувацького досвіду.

Для WebRTC-застосунків Remix може бути надзвичайно ефективним для управління початковим завантаженням інтерфейсу, налаштуваннями користувачів та серверною сигналізацією. Хоча сам WebRTC працює peer-to-peer, йому потрібен сервер для обміну сигналізацією (SDP, ICE-кандидати) між учасниками.

Remix може використовувати свої Loaders для:

- отримання початкових даних для кімнати WebRTC;
- завантаження попередніх налаштувань користувача.

Actions в Remix можуть бути використані для:

- обробки форм для створення або приєднання до WebRTC-кімнат;
- зберігання налаштувань користувача: Наприклад, збереження вибору пристроїв (камера, мікрофон);
- передачі сигнальних даних (SDP, ICE-кандидати), хоча для постійного обміну сигналізацією зазвичай використовуються WebSockets (як більш ефективний механізм для двостороннього зв'язку в реальному часі).

Remix також може бути хостингом для WebSocket-сервера, який буде обробляти постійний потік сигнальних даних між учасниками WebRTC. Його

архітектура дозволяє легко інтегрувати та управляти цими серверними процесами. Нестінгова маршрутизація може бути використана для створення складного інтерфейсу WebRTC, де різні частини UI (наприклад, чат, спільний екран, список учасників) можуть бути окремими компонентами, що завантажують дані незалежно. Загалом, Remix пропонує надійний та продуктивний підхід до розробки WebRTC-застосунків, поєднуючи серверні можливості з оптимізованим клієнтським досвідом.

2.1.2 Python фреймворки

2.1.2.1 Django

Django — це високопродуктивний веб-фреймворк на Python, який працює за принципом "батарейки в комплекті" (batteries-included). Це означає, що він постачається з великою кількістю вбудованих функцій та інструментів, що дозволяють швидко розробляти складні веб-застосунки без необхідності інтегрувати безліч сторонніх бібліотек. Django зосереджений на принципах DRY (Don't Repeat Yourself) та Конвенції над конфігурацією (Convention over Configuration), що сприяє швидкій розробці та підтримці коду.

Django має потужний ORM, який дозволяє взаємодіяти з базами даних за допомогою Python-коду, а не писати SQL-запити вручну. Це спрощує роботу з даними, робить код більш читабельним та переносимим між різними СУБД (PostgreSQL, MySQL, SQLite, Oracle). Для WebRTC-застосунків ORM може бути використаний для зберігання інформації про користувачів, кімнати конференцій, налаштування та історію зв'язку [10].

Django автоматично генерує зручну та функціональну адміністративну панель для управління моделями даних. Це дозволяє адміністраторам легко переглядати, створювати, редагувати та видаляти дані без написання додаткового коду для інтерфейсу управління. У контексті WebRTC, адміністратори можуть керувати користувачами, створювати чи видаляти кімнати, переглядати журнали подій.

Гнучка система маршрутизації дозволяє легко визначити URL-адреси для вашого застосунку та прив'язати їх до відповідних функцій або класів-обробників (views). Це забезпечує чітке розділення логіки та презентації.

Django має власну потужну та безпечну систему шаблонів, яка дозволяє легко відокремлювати логіку від HTML-коду. Хоча для WebRTC-застосунків часто використовується JavaScript-фронтенд, шаблони Django можуть бути використані для рендерингу початкових сторінок, таких як сторінка входу, реєстрації або загальних відомостей.

Django надає механізм для створення, валідації та обробки веб-форм, що значно спрощує взаємодію з користувачем та збір даних.

Фреймворк має вбудовані механізми захисту від поширених веб-атак, таких як CSRF (Cross-Site Request Forgery), XSS (Cross-Site Scripting) та SQL-ін'єкції. Це робить його надійним вибором для розробки застосунків з високими вимогами до безпеки.

Django розроблений для масштабування, що дозволяє застосункам обробляти зростаючу кількість користувачів та трафіку. Він може працювати з різними типами баз даних та бути розгорнутим у розподілених архітектурах.

Хоча Django не є фреймворком для розробки клієнтської частини в реальному часі, він є чудовим вибором для серверної частини WebRTC-застосунку. Він може виконувати роль:

- сигнального сервера: для обробки початкового обміну сигнальними даними (SDP та ICE-кандидати) між учасниками WebRTC;
- обробки WebSocket-з'єднання, chat-протоколи та інші типи зв'язку, що не базуються на HTTP за допомогою розширення Django Channels;
- управління користувачами та кімнатами;
- зберігання історії;
- backend для REST API;
- адміністративного інтерфейсу для моніторингу активних WebRTC-сесій, управління користувачами, налаштування системних

параметрів без необхідності розробки окремого адміністративного інтерфейсу.

Таким чином, Django забезпечує надійну, безпечну та масштабовану основу для серверної частини WebRTC-системи, дозволяючи розробникам зосередитися на бізнес-логіці та уникнути рутинних завдань.

2.1.2.2 FastAPI

FastAPI — це сучасний, швидкий (високопродуктивний) веб-фреймворк для створення API на Python, заснований на стандартних Python-типах та асинхронній програмуванні (`async/await`). Він розроблений з акцентом на швидкість розробки, продуктивність та автоматичну генерацію документації API, використовуючи OpenAPI (раніше Swagger) та JSON Schema.

FastAPI є одним з найшвидших Python-фреймворків, оскільки він побудований на Starlette (для веб-частини) та Pydantic (для валідації даних). Він використовує асинхронну програмування, що дозволяє ефективно обробляти багато одночасних запитів, не блокуючи виконання інших операцій. Це робить його ідеальним для високонавантажених API, де швидкість відгуку є критичною [11].

Повна підтримка синтаксису `async/await` дозволяє розробникам писати асинхронний код, який ефективно працює з I/O-операціями, такими як запити до бази даних, мережні запити або робота з файловою системою. Це особливо важливо для WebRTC, де потрібна ефективна обробка великої кількості одночасних з'єднань та сигнальних повідомлень.

Завдяки інтеграції з Pydantic, FastAPI автоматично валідує вхідні дані (запити) та серіалізує вихідні дані (відповіді) на основі Python-типів. Це значно зменшує кількість boilerplate-коду, спрощує валідацію та обробку помилок, а також забезпечує надійність даних.

FastAPI автоматично генерує інтерактивну документацію API у форматі OpenAPI (Swagger UI та ReDoc). Це дозволяє розробникам та споживачам API легко розуміти та тестувати API без додаткових зусиль для написання документації вручну. Для WebRTC-застосунків це може бути використано для документування ендпоінтів сигналізації.

Вбудована система впровадження залежностей спрощує організацію коду, дозволяючи легко керувати залежностями між компонентами та повторно використовувати логіку. Це покращує тестування та модульність застосунку.

FastAPI має вбудовану підтримку для сучасних стандартів безпеки, таких як OAuth2 та JWT (JSON Web Tokens), що спрощує реалізацію автентифікації та авторизації.

Завдяки базовому фреймворку Starlette, FastAPI має відмінну підтримку WebSockets. Це є ключовою перевагою для розробки WebRTC-сигналізації, оскільки WebSockets є ідеальним протоколом для двостороннього зв'язку в реальному часі.

FastAPI є винятковим вибором для серверної частини WebRTC-системи, особливо для сигнального сервера. Його асинхронна природа та вбудована підтримка WebSockets роблять його ідеальним для обробки великої кількості одночасних з'єднань та обміну сигнальними даними.

FastAPI може ефективно обробляти WebSocket-з'єднання для обміну SDP-описами та ICE-кандидатами між учасниками відеоконференції. Це критично важливий етап для встановлення peer-to-peer з'єднання в WebRTC. Завдяки `async/await`, FastAPI може керувати тисячами одночасних WebSocket-з'єднань, не блокуючи роботу сервера.

Завдяки автоматичній валідації даних та генерації документації, розробка API для сигналізації WebRTC відбувається дуже швидко та ефективно. Це дозволяє команді швидко імплементувати необхідні функції та надавати чітку документацію для фронтенд-розробників.

Отже, FastAPI є потужним, швидким та сучасним вибором для серверної частини WebRTC-системи, що забезпечує необхідну продуктивність, надійність та зручність розробки для обробки комунікації в реальному часі.

2.1.3 Ruby фреймворки

2.1.3.1 Ruby on Rails

Ruby on Rails (часто скорочується як Rails) — це фреймворк для розробки веб-застосунків, написаний на мові програмування Ruby. Слідуючи принципу "конвенції над конфігурацією" (Convention over Configuration) Rails також активно використовує принцип DRY, що сприяє створенню чистого та підтримуваного коду [12].

Rails строго дотримується патерну MVC, що забезпечує чітке розділення відповідальності між компонентами застосунку:

- моделі (Models) взаємодіють з базою даних і містять бізнес-логіку;
- представлення (Views) відповідають за відображення даних користувачеві (HTML, JSON тощо);
- контролери (Controllers) обробляють запити користувачів, взаємодіють з моделями та передають дані представленням.

Це робить код більш організованим, легким для розуміння та масштабування.

Rails має потужний ORM Active Record, який спрощує взаємодію з реляційними базами даних. Він дозволяє маніпулювати даними, використовуючи об'єкти Ruby, замість написання SQL-запитів. Це значно підвищує продуктивність розробки та зменшує кількість помилок.

Фреймворк має вбудований механізм для управління та оптимізації статичних ресурсів (CSS, JavaScript, зображення). Він дозволяє конкатенувати та мініфікувати файли, що покращує продуктивність завантаження сторінок.

Rails надає вбудовані засоби захисту від поширених веб-атак, таких як Cross-Site Request Forgery (CSRF), Cross-Site Scripting (XSS) та SQL-ін'єкції. Це допомагає розробникам створювати більш безпечні застосунки.

Ruby on Rails, незважаючи на те, що він не є типовим фреймворком для розробки клієнтської частини WebRTC-застосунків (де домінує JavaScript), може служити потужним бекендом для сигналізації та управління даними.

Action Cable — це основний компонент, який робить Rails привабливим для WebRTC. Він дозволяє створити WebSocket-сервер безпосередньо в Rails-застосунку. Це ідеально підходить для обміну SDP-описами і передачі ICE-кандидатів. Action Cable забезпечує двосторонній зв'язок у реальному часі, що є фундаментальним для сигналізації WebRTC.

Таким чином, Ruby on Rails, завдяки своїй повноті, швидкості розробки та інтегрованому Action Cable, може слугувати міцною та ефективною серверною платформою для складних WebRTC-застосунків, що вимагають надійного управління даними та зв'язку в реальному часі.

2.1.4 Порівняльний аналіз фреймворків для WebRTC застосунків

Для вибору оптимального фреймворку для розробки системи обміну мультимедійними даними з використанням технології WebRTC, необхідно провести порівняльний аналіз, зосереджуючись на їхніх ключових характеристиках та відповідності потребам WebRTC-застосунків. WebRTC є переважно клієнтською технологією для peer-to-peer зв'язку, але вона потребує сигнального сервера для встановлення з'єднання та управління сесіями. Також важливим є фронтенд-фреймворк для розробки користувацького інтерфейсу. Порівняння розглянутих фреймворків викладено у таблиці 2.1.

Таблиця 2.1 – Порівняння фреймворків для розробки WebRTC застосунків

Характеристика/Фреймворк	Next.js (JS/TS)	Remix (JS/TS)	Django (Python)	FastAPI (Python)	Ruby on Rails (Ruby)
Призначення	Фулстек / Фронтенд з SSR/SSG	Фулстек / Фронтенд з SSR/SSG	Бекенд	Бекенд	Бекенд
Основна мова	JavaScript/TypeScript	JavaScript/TypeScript	Python	Python	Ruby
Основний фокус	Продуктивність, SEO, DX	Продуктивність, веб-стандарти, DX	Швидка розробка, "батарейки"	Продуктивність, API, async	Швидка розробка, конвенції
Підтримка WebSockets	Потребує інтеграції	Потребує інтеграції Loaders/ Actions для початкових запитів	Django Channels	Вбудована	Action Cable (вбудований)
Переваги для WebRTC	Швидкий фронтенд, API Routes для сигналізації.	Оптимізований фронтенд, ефективна робота з даними.	Надійний бекенд (користувачі, кімнати), Django Channels для сигналізації.	Висока продуктивність для WebSocket, Async/await, автодокументація.	Вбудований Action Cable для WebSockets, швидка розробка бекенду.
Недоліки для WebRTC	WebSockets не вбудовані, API Routes можуть бути обмежені для складного бекенду.	WebSockets не вбудовані, молодша екосистема.	Channels потребує конфігурації, менш нативна асинхронність.	Потрібен окремий фронтенд, "тонкий" для базових функцій.	Ruby повільніший, менш нативна асинхронність.
Сценарії використання	Складні WebRTC з багатим UI, фулстек з акцентом на фронтенд.	WebRTC з акцентом на швидкість завантаження та інтерактивність.	Великі WebRTC-системи з комплексним управлінням даними та користувачами.	Високонавантажнені сигнальні сервери, мікросервіси для WebRTC.	WebRTC, що потребує швидкого прототипування та вбудованих WebSockets.

2.1.5 Підсумки та обґрунтування вибору Next.js

На основі проведеного порівняльного аналізу, для програмної реалізації системи обміну мультимедійними даними з використанням

технології WebRTC обґрунтованим вибором є Next.js. Розглянемо основні переваги цього вибору.

WebRTC є технологією, яка сильно залежить від клієнтської сторони для керування медіапотокami, встановлення peer-to-peer з'єднань та відображення відео/аудіо. Next.js, будучи фреймворком на базі React, автоматично дає нам доступ до потужної екосистеми React. React є де-факто стандартом для розробки складних, інтерактивних користувацьких інтерфейсів. Це дозволяє створити динамічний, швидкий та зручний інтерфейс для відеоконференцій, чату та інших функцій обміну даними. Бібліотеки React, такі як Shadcn/UI та TailwindCSS (які будуть розглянуті далі), чудово інтегруються з Next.js та дозволяють створювати привабливий та адаптивний UI.

Хоча Next.js традиційно є фронтенд-фреймворком, його функція API Routes дозволяє створювати легкі бекенд-ендпоінти безпосередньо в проєкті. Це ідеально підходить для реалізації сигнального сервера WebRTC. Замість розгортання окремого бекенду на Python або Ruby, Next.js може обробляти початкові HTTP-запити для обміну SDP та ICE-кандидатами, а також для управління сесіями та користувачами. Для постійного двостороннього зв'язку в реальному часі, необхідного для WebRTC, API Routes Next.js можуть бути інтегровані з WebSocket-сервером (наприклад, на Node.js). Це дозволяє тримати весь застосунок в одній кодовій базі та на одній технологічній стеці (JavaScript/TypeScript), що спрощує розробку, розгортання та підтримку [8].

SSG не є критичним для динамічних WebRTC-сесій, але SSR може покращити початкове завантаження сторінки користувача та прискорити відображення інтерфейсу, що важливо для хорошого користувацького досвіду. Ці оптимізації сприяють створенню швидкого та чуйного застосунку.

Next.js має величезну та активну спільноту, велику кількість ресурсів, документації та готових компонентів. Це значно прискорює розробку, оскільки можна знайти рішення для більшості типових завдань та отримати

підтримку. Використання TypeScript додає безпеку та передбачуваність у процесі розробки, що особливо важливо для складних систем.

Розробка на Next.js означає, що як фронтенд, так і значна частина бекенд-логіки (API Routes, WebSocket-сервер на Node.js) можуть бути написані на JavaScript/TypeScript. Це зменшує когнітивне навантаження на команду, спрощує обмін знаннями та дозволяє розробникам бути фулстек без перемикання між різними мовами та екосистемами.

Таким чином, Next.js забезпечує комплексне рішення, поєднуючи передові можливості для створення інтерфейсу користувача з необхідною гнучкістю для реалізації серверної частини сигналізації WebRTC, що робить його оптимальним вибором для даної системи.

2.2 Аналіз використаних інструментів розробки

2.2.2 Інструменти клієнтської частини

2.2.2.1 Бібліотека React як основа клієнтської частини

Бібліотека React є однією з найпопулярніших та найвпливовіших JavaScript-бібліотек для побудови користувацьких інтерфейсів (UI). Розроблена компанією Facebook, вона здобула широке визнання завдяки своєму декларативному підходу до розробки, що значно спрощує створення складних та інтерактивних інтерфейсів. Основна ідея React полягає у розбитті інтерфейсу на невеликі, ізольовані та багаторазово використовувані частини, відомі як компоненти. Кожен компонент відповідає за рендеринг певної частини UI і керує своїм власним станом. Використання бібліотеки React є стандартом роботи з обраним фреймворком Next.js. Основними властивостями цього інструменту є декларативний підхід, компонентна архітектура, Віртуальний DOM, JSX

Декларативний підхід полягає у тому, щоб замість того, щоб вказувати, як саме має змінюватися UI (імперативний підхід), React дозволяє розробникам просто описати, як UI має виглядати в будь-який момент часу,

виходячи з поточного стану даних. React самостійно ефективно оновлює DOM, коли дані змінюються. Це робить код більш передбачуваним, легким для розуміння та відлагодження.

Компонентна архітектура реалізована шляхом розбиття інтерфейсу на незалежні компоненти, кожен з яких має власну логіку та вигляд. Це сприяє модульності, повторному використанню коду та спрощує підтримку великих застосунків. Наприклад, у WebRTC-застосунку можна створити окремі компоненти для відеопотоку учасника, кнопки керування мікрофоном, чату або списку учасників.

React використовує віртуальний DOM – легку копію реального DOM. Коли дані компонента змінюються, React спочатку оновлює віртуальний DOM, потім порівнює його з попереднім станом віртуального DOM. Тільки мінімальні зміни, необхідні для синхронізації реального DOM, застосовуються. Це значно підвищує продуктивність, оскільки прямі маніпуляції з реальним DOM є "дорогими" операціями.

JSX – це розширення синтаксису JavaScript, яке дозволяє писати HTML-подібний код безпосередньо всередині JavaScript-файлів. JSX робить код більш читабельним та інтуїтивно зрозумілим, об'єднуючи логіку компонента та його візуальне представлення.

Застосунок, розроблений за допомогою React має оноспрямований потік даних. Це проявляється у тому, що дані зазвичай передаються в одному напрямку – від батьківських компонентів до дочірніх через властивості (props). Це робить потік даних передбачуваним і спрощує відстеження змін та виявлення помилок.

2.2.2.3 Shadcn/ui система для побудови бібліотеки компонентів

Shadcn/ui – це не традиційна бібліотека компонентів у звичному розумінні (як, наприклад, Material-UI або Ant Design), а радше набір повторно використовуваних компонентів React, які можна копіювати та налаштовувати

безпосередньо у своєму проєкті. Це система, яка надає джерельний код готових компонентів (кнопки, форми, діалоги, вкладки тощо), які розробники можуть включати у свій проєкт і повністю контролювати їхню стилізацію та поведінку. Основна ідея Shadcn/ui полягає у забезпеченні максимальної гнучкості та кастомізації, дозволяючи створювати унікальний дизайн, не будучи прив'язаним до жорстких стилів або тем сторонньої бібліотеки [13].

Копіювання коду замість встановлення пакету: Замість того, щоб встановлювати Shadcn/ui як залежність з менеджера пакетів, потрібно копіювати сирцевий код необхідних компонентів безпосередньо у проєкт. Це означає, що розробник має повний контроль над кодом, можете змінювати його як завгодно, виправляти помилки або додавати нові функції, не чекаючи оновлень від розробників бібліотеки.

Оскільки компоненти є частиною коду проєкту розробника, їх можна стилізувати та змінювати без обмежень. Це дозволяє досягти унікального дизайну та ідентичності бренду, що часто є складним при використанні готових бібліотек з фіксованими темами.

Shadcn/ui надає лише компоненти. Він не включає в себе великий рантайм-бандл або складні залежності. Це робить застосунок легшим і швидшим.

Ватро зазначити, що компоненти Shadcn/ui розроблені з урахуванням стандартів доступності (WCAG), що є критично важливим для створення інклюзивних веб-застосунків. Це забезпечує, що застосунок буде зручним для всіх користувачів, включаючи тих, хто використовує допоміжні технології.

Код компонентів Shadcn/ui є чистим, добре структурованим та використовує сучасні практики React. Це покращує досвід розробника, дозволяючи швидко розуміти та адаптувати компоненти.

2.2.2.4 TailwindCSS: стилізація компонентів

Tailwind CSS — це утилітарний (utility-first) CSS-фреймворк, який відрізняється від традиційних CSS-фреймворків (таких як Bootstrap або Bulma) тим, що надає набір низькорівневих, функціональних CSS-класів, які можна комбінувати безпосередньо у HTML-розмітці для стилізації елементів. Замість попередньо визначених компонентів з фіксованими стилями, Tailwind CSS зосереджується на наданні "будівельних блоків" (utility classes) для контролю над розміром, кольором, типографікою, відступами, Flexbox, Grid та багатьма іншими CSS-властивостями [14].

Замість написання кастомного CSS для кожного елемента, розробник застосовує безпосередньо класи Tailwind до HTML-тегів. Наприклад, `text-blue-500` для синього тексту, `p-4` для внутрішнього відступу `1rem` з усіх боків, `flex items-center justify-center` для центрування елементів у Flexbox-контейнері. Це прискорює розробку, оскільки не потрібно перемикатися між HTML та CSS-файлами.

Завдяки готовим утилітарним класам, розробники можуть дуже швидко створювати та змінювати дизайн інтерфейсу. Це ідеально для прототипування та швидких ітерацій.

Tailwind CSS не нав'язує жодного візуального стилю. Це дозволяє створювати абсолютно унікальні дизайни, що повністю відповідають брендбуку проєкту. Кожен клас є невеликим і виконує одну функцію, що дозволяє комбінувати їх для досягнення будь-якого бажаного вигляду. Завдяки файлу конфігурації, можна налаштувати всі значення (кольори, шрифти, відступи, breakpoints) відповідно до дизайн-системи проєкту.

Завдяки інструменту PostCSS, Tailwind CSS автоматично видаляє невикористані CSS-класи під час збирання проєкту. Це гарантує, що фінальний CSS-файл буде максимально малим, що позитивно впливає на швидкість завантаження сторінок.

У поєднанні з React та Shadcn/ui, Tailwind CSS формує потужну та гнучку тріаду для розробки сучасного, швидкого та візуально привабливого клієнтського інтерфейсу WebRTC-застосунку, де кожна деталь дизайну може бути точно налаштована.

2.2.3 Серверні технології

2.2.3.1 Node.js: серверне середовище

Для розробки серверної частини застосунку було обрано середовище Node.js.

Node.js — це кросплатформове середовище виконання JavaScript, яке дозволяє запускати JavaScript-код на сервері, поза межами веб-браузера. Побудований на двигуні V8 JavaScript від Google Chrome, Node.js є асинхронним, подієво-орієнтованим і надзвичайно ефективним для побудови масштабованих мережних застосунків. Він ідеально підходить для розробки бекенд-сервісів, API та застосунків реального часу, які вимагають швидкої обробки великої кількості одночасних з'єднань [15].

Node.js працює на одній нитці (single-threaded) з неблокуючим вводом/виводом. Це означає, що замість очікування завершення I/O-операцій (наприклад, запитів до бази даних або мережних запитів), Node.js реєструє колбек-функції та продовжує обробку інших запитів. Коли I/O-операція завершується, колбек виконується. Цей підхід дозволяє Node.js ефективно обробляти велику кількість одночасних з'єднань без створення додаткових ниток, що значно підвищує продуктивність і масштабованість.

Завдяки неблокуючому вводу/виводу та потужному двигуну V8, Node.js є надзвичайно швидким. Він ідеально підходить для застосунків, які активно працюють з мережею, таких як API, чати та застосунки реального часу.

Node.js має найбільший у світі репозиторій пакетів – npm (Node Package Manager). Тисячі бібліотек та інструментів доступні для будь-яких завдань, від

роботи з базами даних до обробки зображень та реалізації протоколів реального часу. Це значно прискорює розробку та дозволяє використовувати готові, перевірені рішення.

Завдяки своїй асинхронній природі та можливості легко інтегруватись з WebSocket-серверами, Node.js є одним з найкращих рішень для створення застосунків, які потребують комунікації в реальному часі.

Node.js є фундаментальним вибором для серверної частини WebRTC-системи, зокрема для реалізації сигнального сервера:

Обробка сигналізації WebRTC: WebRTC-з'єднання є peer-to-peer, але для їх встановлення потрібен сигнальний сервер. Node.js, завдяки своїй здатності ефективно керувати багатьма одночасними з'єднаннями WebSocket, ідеально підходить для обміну SDP-описами та ICE- між учасниками. Цей обмін даними відбувається до того, як пряме peer-to-peer з'єднання буде встановлено [15].

WebSocket-сервер: Node.js дозволяє легко створити WebSocket-сервер за допомогою бібліотек, таких як ws або Socket.IO. Це забезпечує двосторонній зв'язок у реальному часі, необхідний для безперервного обміну сигнальними повідомленнями між браузерами учасників конференції.

Node.js може виступати як проксі-сервер або допоміжний компонент для інтеграції TURN/STUN, якщо це необхідно для подолання обмежень NAT та брандмауерів. Але зазвичай Проксі-сервер та TURN/STUN є окремими спеціалізованими сервісами

Загалом, Node.js є потужним, гнучким та високоефективним серверним середовищем, яке надає необхідні можливості для побудови масштабованої та надійної сигнальної інфраструктури для системи обміну мультимедійними даними на базі WebRTC.

2.2.3.2 WebSocket: двостороння комунікація

WebSocket — це протокол двостороннього (full-duplex) зв'язку через одне TCP-з'єднання, розроблений спеціально для веб-застосунків, що вимагають обміну даними в реальному часі. На відміну від традиційного HTTP, який є односпрямованим (клієнт відправляє запит, сервер надсилає відповідь, і з'єднання закривається), WebSocket встановлює постійне, відкрите з'єднання між клієнтом (браузером) та сервером. Це дозволяє обом сторонам надсилати дані в будь-який момент без необхідності повторного встановлення з'єднання або частих запитів [16].

Після встановлення WebSocket-з'єднання, як клієнт, так і сервер можуть надсилати дані один одному в будь-який час, без ініціації запиту. Це значно зменшує затримку та покращує швидкість обміну даними.

WebSocket передає дані у вигляді "повідомлень", а не окремих HTTP-запитів/відповідей. Це оптимізовано для частих та невеликих обсягів даних, що є характерним для застосунків реального часу.

Після початкового рукоштовування (handshake), WebSocket-кадри даних мають значно менші накладні витрати (менші заголовки), ніж HTTP-запити. Це призводить до більш ефективного використання мережних ресурсів та менших затримок.

З'єднання залишається відкритим, усуваючи необхідність багаторазового встановлення TCP-з'єднання та обміну HTTP-заголовками, що економить ресурси сервера та клієнта.

WebSocket є критично важливим протоколом для реалізації сигнального сервера в WebRTC. Хоча сам WebRTC є технологією peer-to-peer для прямого обміну медіаданими, йому потрібен сигнальний канал для встановлення цього з'єднання. WebSocket ідеально підходить для цієї ролі:

Обмін сигнальними повідомленнями: Перед тим, як два peer-to-peer з'єднання WebRTC зможуть встановитися, вони повинні обмінятися важливою інформацією, яка називається "сигналізацією". Це включає:

SDP (Session Description Protocol) Offers та Answers: Це описи медіапотоків (аудіо, відео), кодеків, їхніх форматів та іншої інформації, необхідної для узгодження параметрів з'єднання.

ICE-кандидати (Interactive Connectivity Establishment): Це інформація про можливі мережні адреси (IP-адреси, порти) та протоколи, які піри можуть використовувати для з'єднання. Обмін ICE-кандидатами допомагає WebRTC знайти найкращий шлях між двома пристроями, навіть якщо вони знаходяться за NAT або брандмауерами [17].

Керування сесіями: WebSocket-з'єднання може використовуватися для передачі повідомлень про вхід/вихід учасників, зміну статусу (наприклад, увімкнення/вимкнення мікрофона), створення/закриття кімнат, а також для чату в реальному часі.

Синхронізація стану: Будь-які зміни в стані конференції (наприклад, хто говорить, хто ділиться екраном) можуть бути миттєво синхронізовані між усіма учасниками через WebSocket-з'єднання.

Стійкість до розривів з'єднання: WebSocket-сервер може реалізувати логіку перепідключення та відновлення сесії у випадку тимчасових втрат зв'язку, що покращує надійність системи.

У контексті Node.js, WebSocket-сервер легко реалізується за допомогою бібліотек, таких як Socket.IO. Цей тандем (Node.js + WebSocket) є стандартом де-факто для створення ефективних сигнальних серверів для WebRTC-застосунків.

2.2.4 Інструменти для роботи з WebRTC

WebRTC (Web Real-Time Communication) — це відкритий стандарт, який дозволяє веб-браузерам та мобільним застосункам здійснювати комунікацію в реальному часі безпосередньо (peer-to-peer). Це включає передачу відео, аудіо та довільних даних. WebRTC складається з набору JavaScript API, які дозволяють розробникам отримати доступ до камери та

мікрофона користувача, встановити пряме з'єднання між браузерами та обмінюватися медіапотоками та даними [17].

2.2.4.1 MediaDevices API

MediaDevices API є частиною WebRTC і надає веб-застосункам можливість отримувати доступ до медіапристроїв користувача, таких як веб-камери, мікрофони та динаміки. Це перший крок до будь-якої WebRTC-комунікації, оскільки він дозволяє застосунку захопити локальний медіапотік, який потім може бути відправлений іншим учасникам [17].

Розглянемо ключові можливості MediaDevices API:

- `navigator.mediaDevices.getUserMedia(constraints):` Це найважливіша функція API. Вона запитує у користувача дозвіл на доступ до його медіапристроїв (камери та/або мікрофона). Метод приймає об'єкт `constraints`, який визначає бажані параметри медіапотoku (наприклад, `video: true`, `audio: true`, бажана роздільна здатність відео, частота кадрів). Успішне виконання цього методу повертає `Promise`, який вирішується об'єктом `MediaStream`. Приклад використання наведено у лістингу 2.1;

- `navigator.mediaDevices.enumerateDevices():` Цей метод дозволяє отримати список усіх доступних медіапристроїв (камер, мікрофонів, динаміків) на пристрої користувача. Це корисно для того, щоб надати користувачеві вибір, який пристрій використовувати. Кожен пристрій має `kind` (тип, наприклад, `videoinput`, `audioinput`, `audiooutput`), `label` (зрозуміле ім'я) та `deviceId` (унікальний ідентифікатор);

- `navigator.mediaDevices.getDisplayMedia(constraints):` Цей метод (розширення API) дозволяє захопити вміст екрана користувача (або вікна застосунку) та створити `MediaStream` з нього. Це використовується для функцій демонстрації екрану у відеоконференціях;

– події (ondevicechange): MediaDevices API також надає події, які спрацьовують, коли список доступних пристроїв змінюється (наприклад, користувач підключив нову USB-камеру).

Лістинг 2.1 – Приклад використання функції navigator.mediaDevices.getUserMedia(constraints)

```

navigator.mediaDevices.getUserMedia({ video: true, audio: true
}).then(function(stream) {
// Потік успішно отримано
// stream є об'єктом MediaStream, який містить відео- та аудіодоріжки
// можна відобразити його на <video> елементі
document.getElementById('localVideo').srcObject = stream;
})
.catch(function(err) {
// Обробка помилок (наприклад, користувач відмовив у доступі або
пристроїв немає)
console.error("Помилка доступу до медіапристроїв: " + err);
});

```

MediaDevices API є фундаментальним для будь-якої WebRTC-системи обміну мультимедійними даними, оскільки дозволяє застосунку отримати доступ до камери та мікрофона користувача, що є відправною точкою для будь-якої відео- чи аудіокомунікації. Локальний MediaStream є тим, що відправляється іншим учасникам конференції.

Отриманий MediaStream є вхідним даним для RTCPeerConnection, який потім обробляє його та відправляє віддаленим пірам.

2.2.4.2 RTCPeerConnection

RTCPeerConnection — це центральний інтерфейс у WebRTC API, який відповідає за встановлення та управління прямим peer-to-peer з'єднанням між двома браузерами або пристроями. Він обробляє складні аспекти комунікації в реальному часі, такі як виявлення мережних адрес (ICE), встановлення з'єднання (STUN/TURN), узгодження кодеків та медіаформатів (SDP), а також шифрування та передачу аудіо/відео даних.

Ключові можливості та процес роботи `RTCPeerConnection` наступні:

а) створення `RTCPeerConnection`: Кожен учасник WebRTC-з'єднання створює свій власний об'єкт `RTCPeerConnection`;

б) додавання локальних медіапотоків: Локальні аудіо- та відеодоріжки (отримані за допомогою `MediaDevices.getUserMedia`) додаються до `RTCPeerConnection`;

в) створення `SDP Offer`: Один з пірів (ініціатор з'єднання) створює `SDP Offer`. Це опис медіапотоків, кодеків та інших можливостей, які він пропонує;

г) отримання та обробка `SDP Offer`: Віддалений пір отримує `Offer` через сигнальний сервер і встановлює його як свій віддалений опис;

д) створення та обробка `SDP Answer`: Віддалений пір створює `SDP Answer` у відповідь на `Offer`, який містить узгоджені параметри з'єднання, і відправляє його назад ініціатору. Ініціатор встановлює `Answer` як свій віддалений опис;

е) обмін ICE-кандидатами: Паралельно з обміном `SDP`, обидва піри збирають ICE-кандидати. Це різні варіанти мережних адрес, за якими вони можуть бути доступні;

1) `peerConnection.onicecandidate`: Ця подія спрацьовує кожного разу, коли генерується новий ICE-кандидат. Кожен кандидат має бути відправлений іншому піру через сигнальний сервер;

2) `peerConnection.addIceCandidate(candidate)`: Отримані від віддаленого піра кандидати додаються до `RTCPeerConnection`, щоб допомогти йому знайти найкращий шлях для з'єднання;

ж) події медіапотоків (`ontrack`): Коли віддалений пір починає надсилати медіапотоки, спрацьовує подія `ontrack`. Ця подія дозволяє отримати віддалений `MediaStream` і відобразити його в UI.

`RTCPeerConnection` є ядром WebRTC-комунікації. Він відповідає за всі складні етапи встановлення прямого, захищеного, peer-to-peer з'єднання для передачі медіаданих.

RTCPeerConnection автоматично обробляє шифрування (DTLS та SRTP) для захисту медіапотоків, а також керує якістю з'єднання та адаптується до змін у мережних умовах.

API дозволяє додавати та видаляти медіадоріжки (відео, аудіо, дані екрану) у вже встановленій сесії, що дозволяє реалізовувати функції, такі як увімкнення/вимкнення камери або демонстрації екрану в реальному часі.

Крім цього, цей інтерфейс також надає можливість створення Data Channels – двосторонніх каналів для передачі довільних даних (не медіа) у реальному часі. Це може бути використано для чату, спільного використання документів, ігрових даних тощо.

2.2.4.3 MediaStream API

MediaStream API є ключовою частиною WebRTC, яка представляє собою потік медіаданих, таких як аудіо та відео. Ці потоки можуть надходити з різних джерел: з локальних медіапристроїв користувача (камера, мікрофон, екран), з віддаленого peer-to-peer з'єднання (отримані від іншого учасника WebRTC), або з локальних файлів/генераторів. MediaStream є своєрідним "контейнером" для аудіо- та відеодоріжок.

Об'єкт MediaStream – це основний об'єкт, який представляє потік медіаданих. Він містить одну або кілька MediaStreamTrack об'єктів.

Об'єкт MediaStreamTrack – це окрема доріжка в медіапотоці. Вона може бути відео- (video) або аудіо- (audio) доріжкою.

Кожна доріжка має унікальний id, kind (тип: "audio" або "video"), label (опис, наприклад, "Front Camera"), а також стан (readyState: "live" або "ended").

Важливі методи MediaStreamTrack наступні:

- track.stop(): Зупиняє доріжку, що означає припинення захоплення медіаданих цим пристроєм. Це важливо для звільнення ресурсів при вимкненні камери або мікрофона;

– `track.enabled`: Логічне значення, яке дозволяє увімкнути (`true`) або вимкнути (`false`) доріжку. Наприклад, встановлення `audioTrack.enabled = false` вимкне звук з мікрофона без повного припинення захоплення.

Отриманий `MediaStream` можна легко відобразити в HTML-елементах `<video>` або `<audio>` за допомогою властивості `srcObject`, це дозволяє браузеру автоматично відтворювати відео та аудіо, що надходять з потоку.

`MediaStream API` є фундаментальною концепцією в `WebRTC`, яка об'єднує джерела медіаданих та їхнє використання. Він надає стандартизований спосіб роботи з аудіо- та відеоданими, незалежно від їхнього джерела (локальні пристрої, віддалені піри, файли).

`MediaStream` є мостом між захопленням медіаданих (через `getUserMedia`) та їхнім відображенням у користувацькому інтерфейсі.

Передача через `RTCPeerConnection`: Саме об'єкти `MediaStream` та їхні `MediaStreamTrack` передаються до `RTCPeerConnection` за допомогою методу `addTrack()` для відправки віддаленим пірам. Аналогічно, `RTCPeerConnection` повертає `MediaStream` віддаленого піра через подію `ontrack`.

`MediaStream API` дозволяє керувати окремими доріжками — наприклад, програмно вимкнути мікрофон або камеру, не розриваючи `WebRTC`-з'єднання. Це реалізується шляхом маніпулювання властивістю `enabled` відповідної `MediaStreamTrack`. Також дозволяє комбінувати різні джерела (наприклад, потік з камери та потік з екрана), що є основою для складних сценаріїв, таких як демонстрація екрана з одночасним відображенням відео з камери.

Таким чином, `MediaStream API` є невід'ємною частиною `WebRTC`, забезпечуючи механізм для захоплення, керування та передачі мультимедійних даних у системі реального часу.

2.3 Архітектура системи

2.3.1 Загальна архітектура

Розроблена система відеоконференцій побудована на основі гібридної архітектури, що поєднує клієнт-серверну модель для сигналізації та управління, та peer-to-peer модель для обміну медіа-даними між учасниками. Така архітектура забезпечує оптимальний баланс між надійністю, масштабованістю та ефективністю передачі даних. Діаграма загальної архітектури системи зображена на рисунку 2.1.

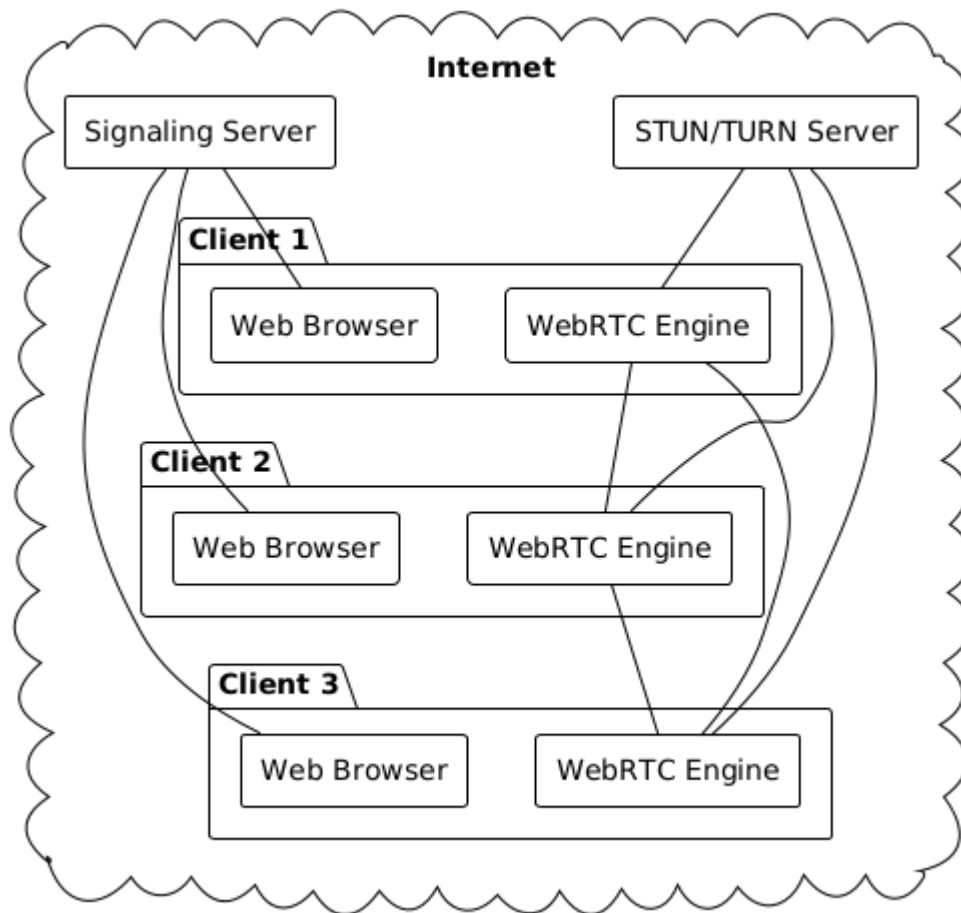


Рисунок 2.1 – Діаграма загальної архітектури системи відеоконференцій

Діаграма демонструє основні компоненти системи та їх взаємозв'язки в межах мережі Інтернет. Розглянемо кожен елемент:

- а) центральні серверні компоненти:

1) Signaling Server (Сервер сигналізації) - центральний компонент, що відповідає за:

- координацію з'єднань між клієнтами;
- управління сесіями;
- обмін службовими повідомленнями;
- маршрутизацію сигнальних повідомлень;

2) STUN/TURN Server - спеціалізований сервер, який забезпечує:

- визначення публічної IP-адреси клієнтів (STUN);
- обхід NAT та файрволів;
- релейну передачу даних у випадках, коли пряме з'єднання неможливе (TURN);

б) Клієнтські компоненти (показані три клієнти, але система може підтримувати більше):

1) Web Browser (Веб-браузер):

- забезпечує користувацький інтерфейс;
- керує взаємодією з користувачем;
- відображає відео та відтворює аудіо;

2) WebRTC Engine (Движок WebRTC):

- обробляє медіа-потіки;
- забезпечує шифрування даних;
- керує мережними з'єднаннями;
- відповідає за кодування/декодування медіа.

Зв'язки між компонентами діаграми наступні:

- суцільні лінії між Signaling Server та веб-браузерами показують канали сигналізації;
- зв'язки між STUN/TURN Server та WebRTC Engine відображають процеси визначення мережної топології;
- лінії між WebRTC Engine різних клієнтів показують прямі P2P з'єднання для обміну медіа-даними.

Особливості архітектури:

- гібридний підхід: централізована сигналізація та децентралізований обмін медіа;
- масштабованість: можливість підключення багатьох клієнтів;
- відмовостійкість: наявність TURN сервера як резервного шляху передачі даних;
- безпека: всі з'єднання захищені шифруванням.

Така архітектура забезпечує наступні властивості системи:

- мінімальну затримку при передачі медіа-даних завдяки P2P зв'язкам;
- надійність завдяки централізованому управлінню;
- можливість роботи в різних мережних умовах;
- безпеку передачі даних;
- гнучкість у масштабуванні системи.

Діаграма наочно демонструє, як різні компоненти системи взаємодіють між собою для забезпечення функціонування відеоконференції, підкреслюючи роль кожного елемента в загальній архітектурі.

2.3.2 Клієнт-серверна взаємодія

Клієнт-серверна взаємодія в системі відеоконференцій забезпечує централізоване управління та координацію між учасниками. Основними компонентами взаємодії є сервер сигналізації та клієнтські застосунки. Діаграма, зображена на рисунку 2.2, демонструє побудовану клієнт-серверну взаємодію системи.

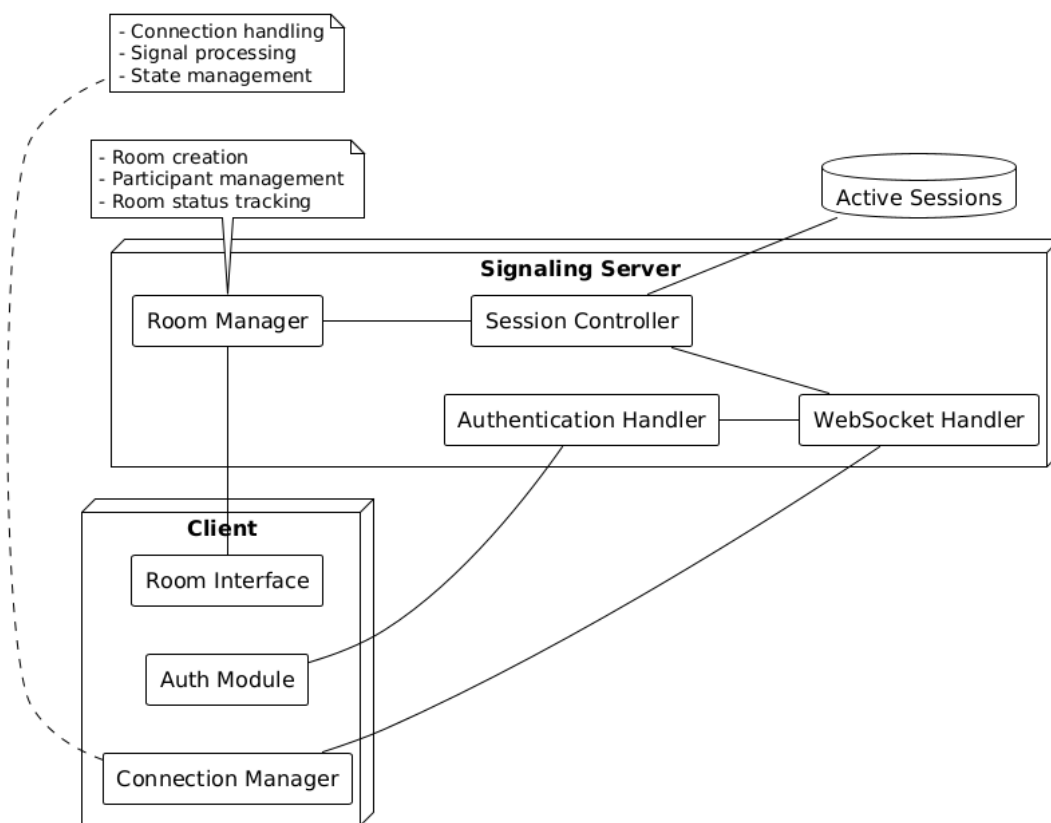


Рисунок 2.2 – Діаграма побудованої клієнт-серверної взаємодії системи

Діаграма відображає основні компоненти системи та їх взаємозв'язки в контексті клієнт-серверної архітектури відеоконференцій. Розглянемо кожен вузол та його компоненти:

а) Вузол "Signaling Server" (Сервер сигналізації) містить чотири ключових компоненти:

1) Room Manager (Менеджер кімнат):

- відповідає за створення та видалення кімнат конференцій;
- керує життєвим циклом кімнат;
- відстежує статус та параметри кожної кімнати;
- контролює список учасників кімнат;

2) Session Controller (Контролер сесій):

- керує активними сесіями користувачів;
- координує обмін сигнальними повідомленнями;
- забезпечує синхронізацію стану між учасниками;

3) Authentication Handler (Обробник автентифікації):

- перевіряє права доступу до кімнат;
- валідує паролі кімнат;
- керує процесом автентифікації користувачів;
- забезпечує безпеку підключень;

4) WebSocket Handler (Обробник WebSocket):

- забезпечує двосторонній зв'язок з клієнтами;
- обробляє вхідні та вихідні повідомлення;
- керує WebSocket з'єднаннями;
- забезпечує надійність передачі даних;

б) вузол "Client" (Клієнт) включає три основні компоненти:

1) Connection Manager (Менеджер з'єднань):

- керує WebSocket з'єднанням з сервером;
- обробляє події підключення та відключення;
- забезпечує відновлення з'єднання при розривах;
- керує станом мережного підключення;

2) Auth Module (Модуль автентифікації):

- відповідає за взаємодію з Authentication Handler;
- зберігає та передає облікові дані;
- обробляє помилки автентифікації;
- керує сесією користувача;

3) Room Interface (Інтерфейс кімнати):

- забезпечує взаємодію з Room Manager;
- надає інтерфейс для управління кімнатою;
- відображає стан та учасників кімнати;
- обробляє події кімнати.

Взаємозв'язки між компонентами наступна:

а) вертикальні зв'язки в Signaling Server:

- 1) Room Manager → Session Controller: передача даних про кімнати;

- 2) Session Controller → WebSocket Handler: маршрутизація повідомлень;
 - 3) Authentication Handler → WebSocket Handler: валідація з'єднань;
- б) горизонтальні зв'язки між клієнтом та сервером:
- 1) WebSocket Handler ↔ Connection Manager: двосторонній обмін даними;
 - 2) Auth Module ↔ Authentication Handler: процеси автентифікації;
 - 3) Room Interface ↔ Room Manager: управління кімнатами.

2.3.3 Peer-to-peer комунікація

В системі реалізована повнозв'язна (full mesh) топологія peer-to-peer комунікації на базі WebRTC технології. Кожен учасник конференції встановлює пряме з'єднання з кожним іншим учасником, створюючи мережу прямих каналів зв'язку. Схема такого зв'язку зображена на рисунку 2.3.

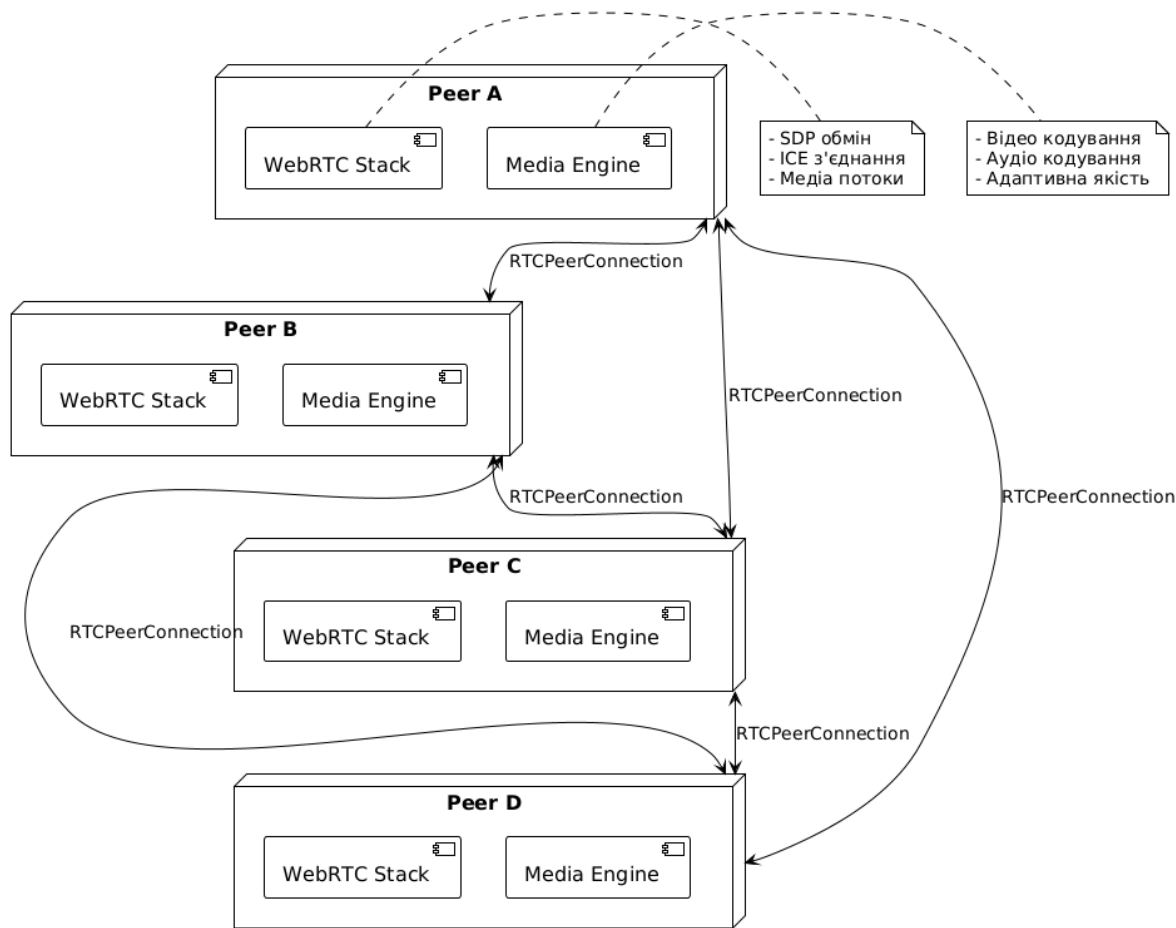


Рисунок 2.3 – Схема зв'язку full mesh для чотирьох учасників

2.4 Інформаційні потоки між компонентами

В системі відеоконференцій існує три основних типи інформаційних потоків: сигнальні, медіа та потоки даних. Кожен з них має свої особливості та призначення.

2.4.1 Потоки сигналізації

Потоки сигналізації є критичним компонентом системи відеоконференцій, що забезпечують координацію та встановлення з'єднань між учасниками. Через сигнальний сервер проходять всі службові повідомлення, необхідні для ініціалізації та підтримки WebRTC з'єднань. Процес включає три основні етапи: управління кімнатами конференцій, обмін SDP (Session Description

Protocol) даними та передачу ICE (Interactive Connectivity Establishment) кандидатів. Діаграма послідовності на рисунку 2.4 демонструє взаємодію між клієнтами через сигнальний сервер та порядок обміну повідомленнями.

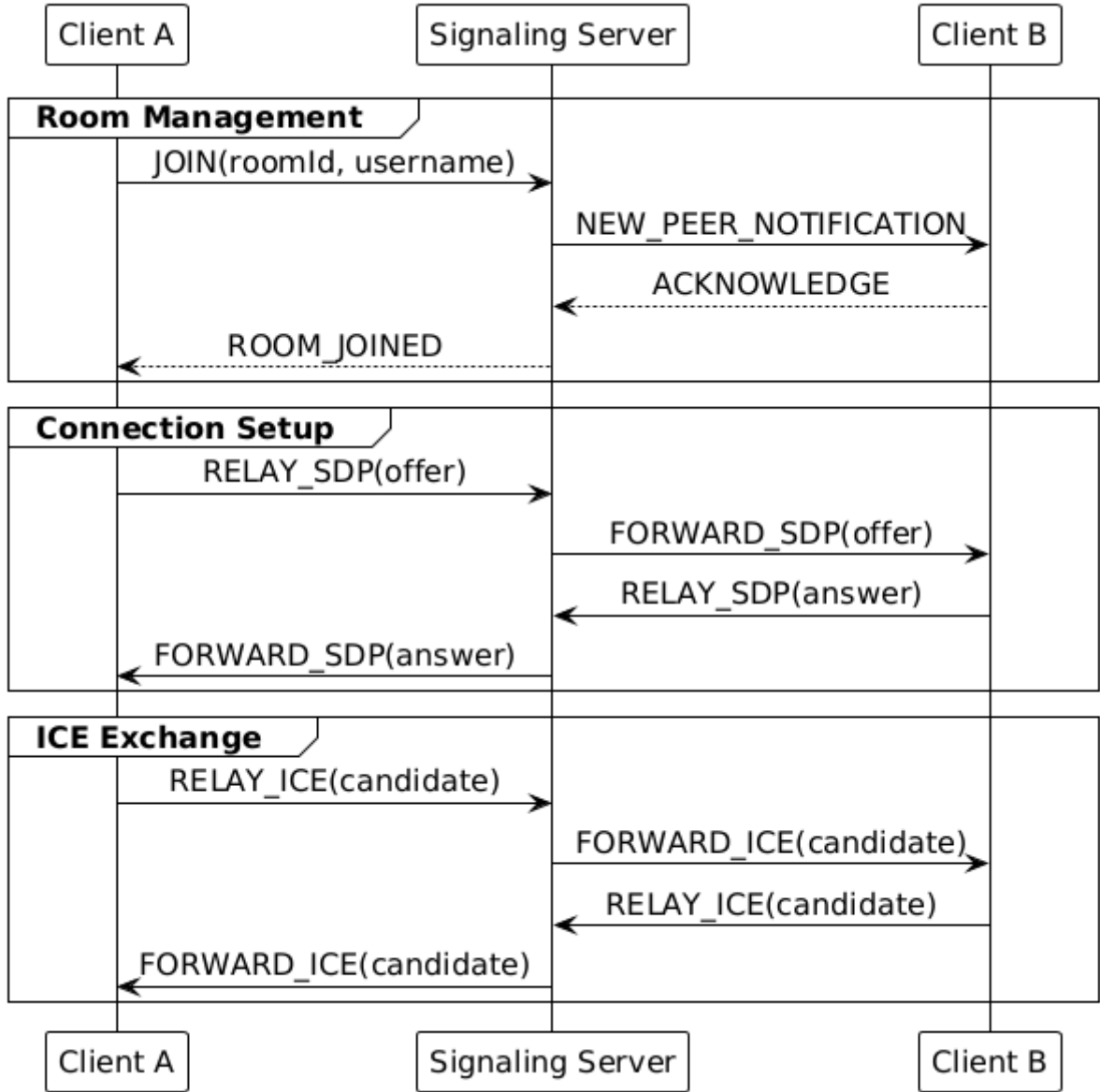


Рисунок 2.4 – Діаграма послідовності потоку сигналізації

Діаграма показує три основні групи сигнальних повідомлень:

- процес приєднання нового учасника;
- обмін SDP офертами та відповідями;
- передача ICE-кандидатів для встановлення P2P з'єднання.

Всі повідомлення проходять через сигнальний сервер. Кожне повідомлення має підтвердження отримання.

2.4.2 Медіа потоки

Медіа потоки представляють собою основний тип даних, що передаються в системі відеоконференцій, включаючи аудіо, відео та контент демонстрації екрану. Передача медіа даних відбувається безпосередньо між учасниками (peer-to-peer) після встановлення WebRTC з'єднання. Процес включає захоплення медіа з пристроїв користувача, обробку та кодування даних, передачу через мережу та відтворення на стороні отримувача. Компонентна діаграма на рисунку 2.5 відображає шлях медіа даних та взаємодію компонентів системи в процесі передачі медіа контенту.

Компоненти діаграми наступні:

а) Peer A (Вузол відправника):

- 1) Media Capture: Компонент захоплення медіа;
- 2) WebRTC Engine: Движок WebRTC;

б) Network (Мережний вузол):

- 1) STUN/TURN Server;
- 2) STUN: визначення публічної IP-адреси;
- 3) TURN: релейна передача при неможливості P2P;
- 4) ICE: узгодження шляху з'єднання;

в) Peer B (Вузол отримувача):

- 1) Media Playback: Відтворення медіа;
- 2) WebRTC Engine: Движок WebRTC.

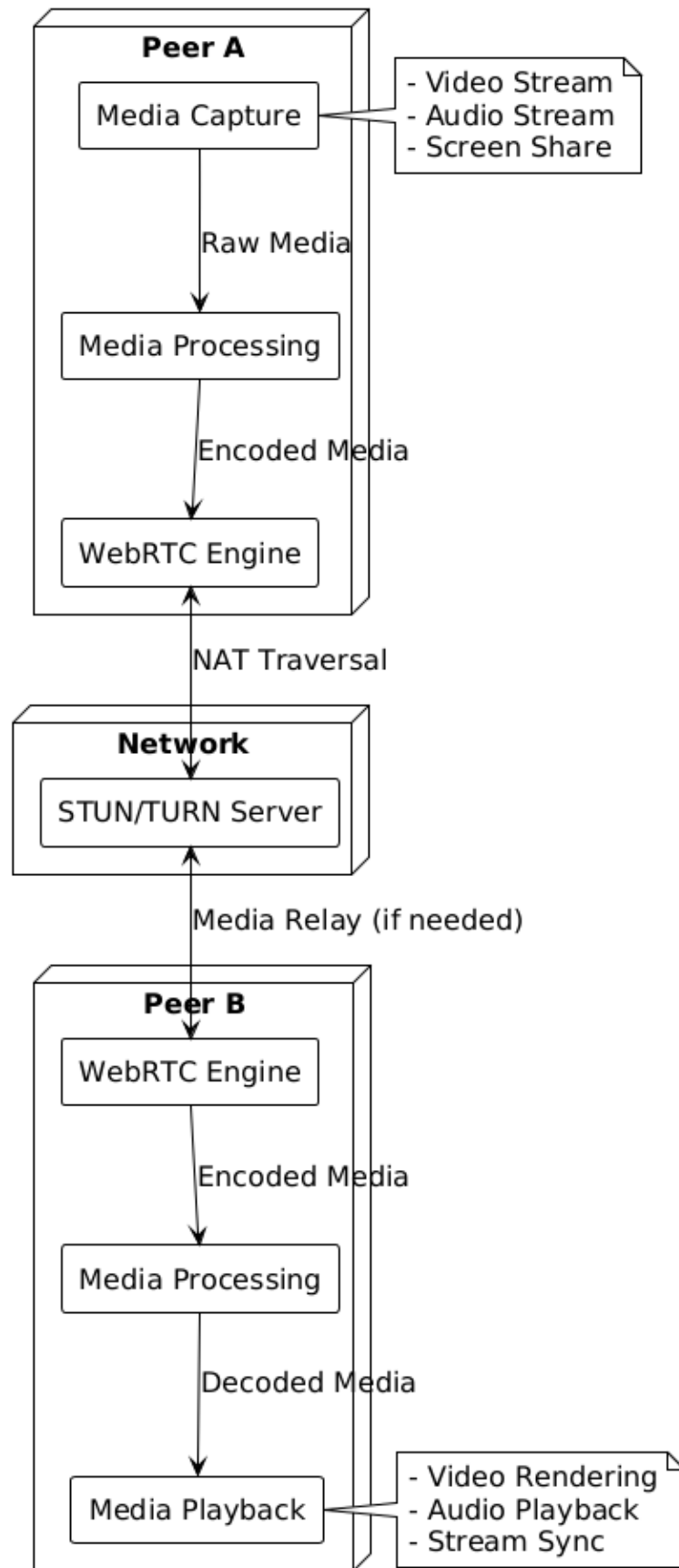


Рисунок 2.5 – Діаграма шляху медіа потоків застосунку

2.4.3 Потоки даних

Потоки даних у системі відеоконференцій забезпечують передачу додаткової інформації, що не відноситься до медіа контенту, але є важливою для повноцінного функціонування системи. Це включає текстові повідомлення чату, статуси учасників, керуючі команди та метадані. Передача даних здійснюється через спеціалізовані WebRTC Data Channels, що забезпечують надійну та швидку доставку інформації між учасниками конференції. Компонентна діаграма на рисунку 2.6 показує структуру та взаємозв'язки різних каналів передачі даних у системі.

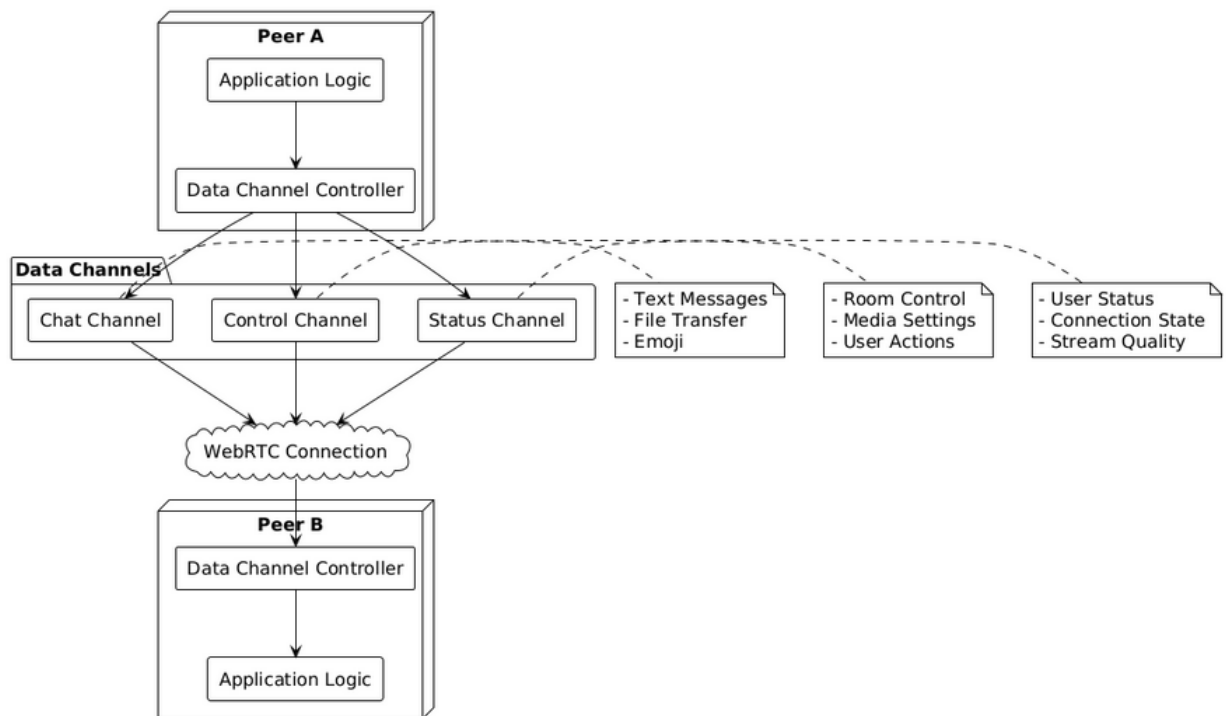


Рисунок 2.6 – Діаграма взаємозв'язки каналів передачі даних у системі

2.5 Функціональна схема роботи відеоконференції

Функціональна схема, зображена на рисунку 2.7, відеоконференції відображає основні компоненти системи та їх взаємодію в процесі роботи. Система побудована на основі клієнт-серверної архітектури для сигналізації

та peer-to-peer з'єднань для обміну медіа-даними. Схема демонструє основні функціональні блоки, їх призначення та потоки даних між ними, що забезпечують повноцінну роботу відеоконференції.

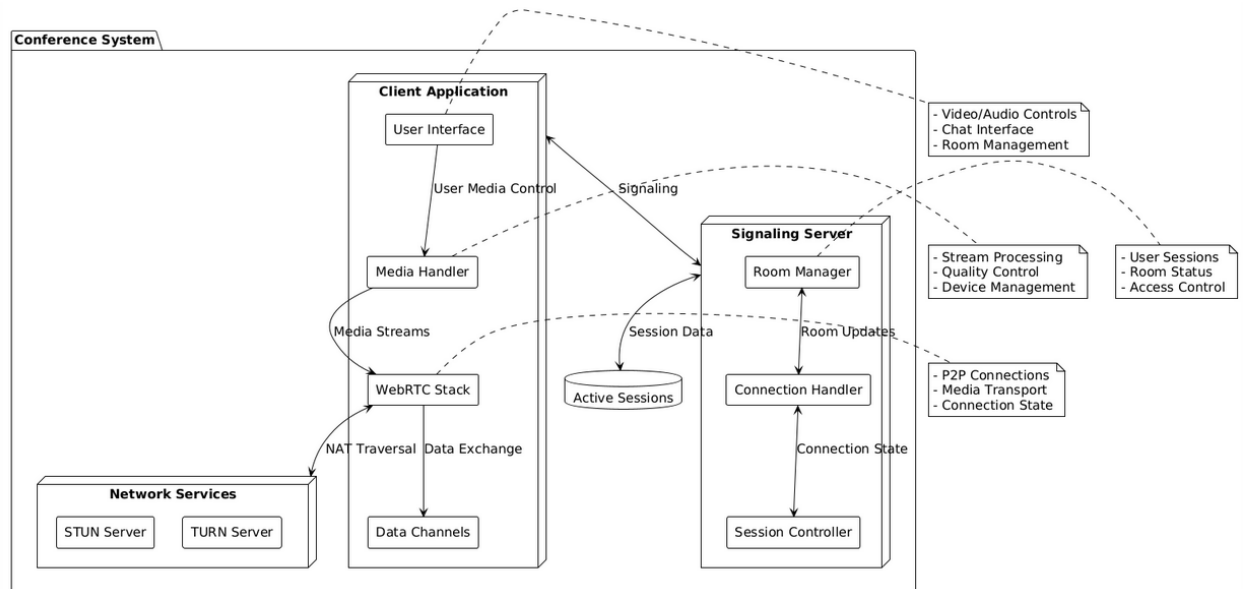


Рисунок 2.7 – Функціональна схема роботи відеоконференції

Елементи схеми наступні:

а) Client Application:

- 1) User Interface: елементи керування та відображення;
- 2) Media Handler: обробка медіа-потоків;
- 3) WebRTC Stack: управління P2P з'єднаннями;
- 4) Data Channels: обмін службовими даними;

б) Signaling Server:

- 1) Room Manager: управління кімнатами конференцій;
- 2) Connection Handler: обробка підключень;
- 3) Session Controller: контроль сесій;

в) Network Services:

- 1) STUN Server: визначення мережних адрес;
- 2) TURN Server: релейна передача даних;

г) Active Sessions:

- 1) зберігання даних активних сесій;
- 2) інформація про кімнати;
- 3) стан користувачів.

2.5.1 Процес встановлення з'єднання

Процес встановлення з'єднання у системі відеоконференцій є критичним етапом, який забезпечує надійну комунікацію між учасниками. Схема, зображена на рисунку 2.8, демонструє послідовність кроків від моменту входу користувача до системи до встановлення повноцінного P2P з'єднання з іншими учасниками конференції, включаючи обмін медіа-потокami та налаштування якості зв'язку.

Короткий опис компонентів схеми, їх складових та функціоналу наступний:

а) User Interface:

- 1) інтерфейс користувача;
- 2) елементи керування конференцією;
- 3) відображення статусу підключення;

б) Connection Manager:

- 1) управління процесом підключення;
- 2) координація між компонентами;
- 3) обробка подій з'єднання;

в) Signaling Server:

- 1) обмін службовими повідомленнями;
- 2) маршрутизація сигналів;
- 3) управління сесіями;

г) WebRTC Engine:

- 1) створення P2P з'єднань;
- 2) обмін SDP та ICE;

- 3) шифрування даних;
- д) Media Handler:
 - 1) доступ до пристроїв;
 - 2) налаштування медіа-потоків;
 - 3) контроль якості;
- е) Remote Peer:
 - 1) віддалений учасник;
 - 2) обмін медіа-даними;
 - 3) підтвердження з'єднання.

Схема відображає чотири основні етапи встановлення з'єднання: автентифікацію та приєднання до кімнати, налаштування медіа-пристроїв, створення WebRTC з'єднання та встановлення обміну медіа-потоків через ICE протокол.

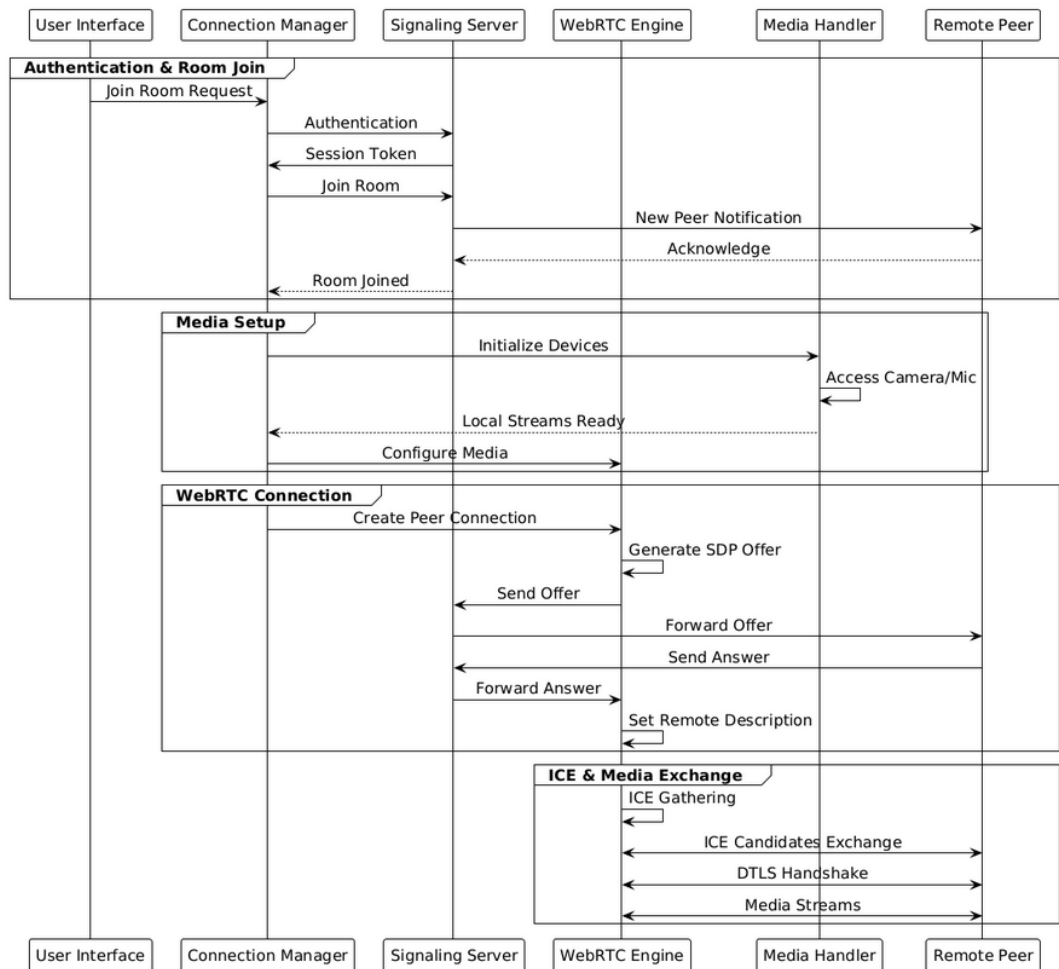


Рисунок 2.8 – Схема встановлення з'єднання у системі відеоконференцій

2.5 Висновки за розділом

У даному розділі було детально розглянуто архітектуру та принципи функціонування системи відеоконференцій. Система побудована на основі гібридної архітектури, що поєднує клієнт-серверну модель для сигналізації та координації учасників, та peer-to-peer модель для обміну медіа-даними. Таке рішення забезпечує оптимальний баланс між надійністю роботи та ефективністю передачі медіа-контенту. Використання WebRTC технології з повнозв'язною (full mesh) топологією дозволяє досягти мінімальних затримок при передачі аудіо та відео даних між учасниками конференції, що є критичним фактором для забезпечення якісного користувацького досвіду.

Аналіз інформаційних потоків показав наявність трьох основних типів даних у системі: сигнальні потоки для координації та встановлення з'єднань, медіа-потоки для передачі аудіо та відео контенту, та службові потоки даних для обміну текстовими повідомленнями та статусами учасників. Кожен тип потоку має свої особливості обробки та передачі, що враховано в архітектурі системи. Детально розглянутий процес встановлення з'єднання демонструє комплексний підхід до забезпечення надійної комунікації, включаючи автентифікацію користувачів, налаштування медіа-пристроїв, створення захищених P2P каналів та адаптивне управління якістю зв'язку. Запропонована архітектура дозволяє системі ефективно масштабуватися для роботи з різною кількістю учасників, зберігаючи при цьому високу якість обслуговування та стабільність роботи.

3 ОСНОВНІ РІШЕННЯ ЩОДО РЕАЛІЗАЦІЇ КОМПОНЕНТІВ СИСТЕМИ

3.1 Структура проекту

Структура проекту представляє собою клієнтський додаток, розроблений з використанням фреймворку Next.js, що забезпечує ефективну організацію та взаємодію компонентів відеоконференції. Проект побудований за принципом модульності, де кожен компонент відповідає за окрему функціональність системи, а взаємодія між ними здійснюється через чітко визначені інтерфейси та події. Архітектура проекту включає клієнтську частину для обробки користувацького інтерфейсу та WebRTC з'єднань, а також інтеграцію з WebSocket сервером для сигналізації та координації між учасниками відеоконференції. Така структура забезпечує масштабованість, легкість підтримки та можливість розширення функціональності системи.

3.1.1 Файлова організація Next.js додатку

Файлова організація проекту відповідає стандартній структурі Next.js додатку з додатковими директоріями для специфічних компонентів відеоконференції. Основні директорії включають:

а) /src - головна директорія з кодом додатку:

1) /app - містить основні компоненти та маршрутизацію Next.js:

- /room/[roomId] - динамічний маршрут для кімнат відеоконференції;
- page.tsx - головна сторінка додатку;
- layout.tsx - загальний шаблон додатку;
- globals.css - глобальні стилі;

2) /components - перевикористовувані компоненти:

- /ui - базові UI компоненти (button, card, dialog, input, label);

- Chat.tsx - компонент текстового чату;

3) /hooks - користувацькі React хуки:

- useStateWithCallback.ts - хук для стану з callback функцією;

- useWebRTC.ts - головний хук для WebRTC функціональності;

4) /lib - утиліти та допоміжні функції:

- socket.ts - налаштування WebSocket з'єднання;

- utils.ts - загальні утиліти;

б) /socket - конфігурація WebSocket взаємодії:

- actions.ts - типи WebSocket повідомлень;

- index.ts - основна логіка WebSocket сервера.

Конфігураційні файли проекту розміщені в кореневій директорії та включають налаштування TypeScript (tsconfig.json), ESLint (eslint.config.mjs), Next.js (next.config.ts) та залежності проекту (package.json). Така організація файлів забезпечує чітку структуру та легке масштабування проекту. Дерево каталогів проекту зображено на рисунку 3.1.

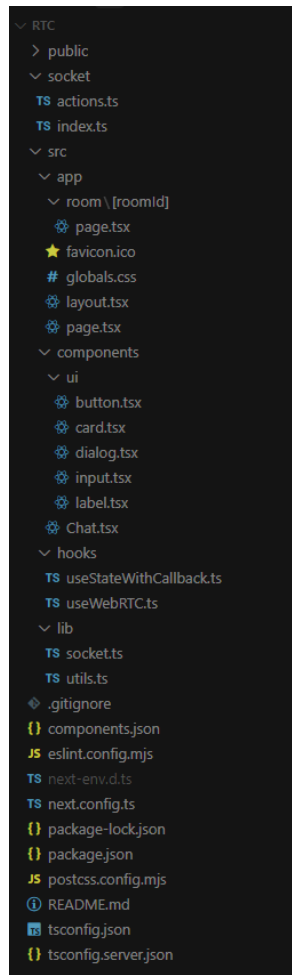


Рисунок 3.1 – Дерево каталогів проєкту

3.1.2 Основні компоненти системи

Основні компоненти розробленої системи наступні:

а) клієнтський додаток Next.js:

- 1) відповідає за користувацький інтерфейс та взаємодію з користувачем;
- 2) керує локальними медіа-потокami (камера, мікрофон, демонстрація екрану);
- 3) реалізує логіку WebRTC з'єднань через хук useWebRTC;
- 4) забезпечує створення та управління peer-to-peer з'єднаннями;

б) сигнальний WebSocket сервер:

- 1) забезпечує обмін SDP та ICE кандидатами між учасниками;

- 2) керує життєвим циклом кімнат відеоконференцій;
 - 3) координує приєднання та від'єднання учасників;
 - 4) передає службові повідомлення між клієнтами (ADD_PEER, REMOVE_PEER, тощо);
- в) WebRTC підсистема:
- 1) встановлює peer-to-peer з'єднання між учасниками;
 - 2) керує медіа-потокami (відео, аудіо);
 - 3) забезпечує роботу DataChannel для текстового чату;
 - 4) відповідає за якість та стабільність з'єднання;
- г) система управління медіа:
- 1) керує доступом до пристроїв користувача (камера, мікрофон);
 - 2) забезпечує захоплення екрану для демонстрації;
 - 3) управляє відтворенням відеофайлів;
 - 4) контролює стан медіа-потоків (увімкнення/вимкнення).

Всі ці компоненти тісно взаємодіють між собою для забезпечення повноцінної роботи відеоконференції. Клієнтський додаток взаємодіє з сигнальним сервером для координації, а WebRTC підсистема використовує отриману інформацію для встановлення прямих з'єднань між учасниками. Система управління медіа інтегрується з WebRTC для забезпечення різних варіантів передачі медіа-контенту.

3.1.3 Взаємодія між компонентами

Взаємодія між компонентами системи базується на комбінації WebSocket та WebRTC протоколів. Клієнтський додаток ініціює з'єднання з сигнальним сервером через WebSocket, який використовується для координації учасників та обміну службовими повідомленнями. Після встановлення з'єднання з сервером, клієнт використовує хук useWebRTC для

створення peer-to-peer з'єднань з іншими учасниками кімнати. WebRTC підсистема керує медіа-потоками та забезпечує пряму передачу даних між клієнтами, в той час як сигнальний сервер залишається відповідальним лише за координацію та управління життєвим циклом кімнати. Повна UML-діаграма взаємодії між усіма компонентами системи показана на рисунку 3.2.

Діаграма відображає комплексну архітектуру системи відеоконференцій, де центральним елементом клієнтської частини виступає хук `useWebRTC`. Цей хук управляє всіма аспектами WebRTC з'єднань, включаючи створення та управління peer-to-peer з'єднаннями, обробку медіа-потоків та організацію текстового чату. У ньому зберігаються важливі стани, такі як список клієнтів, повідомлення чату, стани камери та мікрофону, а також референції на медіа-елементи та з'єднання. Хук реалізує ключові методи для управління медіа (`toggleCamera`, `toggleMicrophone`, `switchToVideoFile`, `toggleScreenShare`) та використовує внутрішні `useEffect` хуки для обробки різних аспектів WebRTC взаємодії. Клієнтський додаток починається з `HomePage`, де користувач може створити або приєднатися до кімнати, після чого відбувається перехід до `RoomPage`.

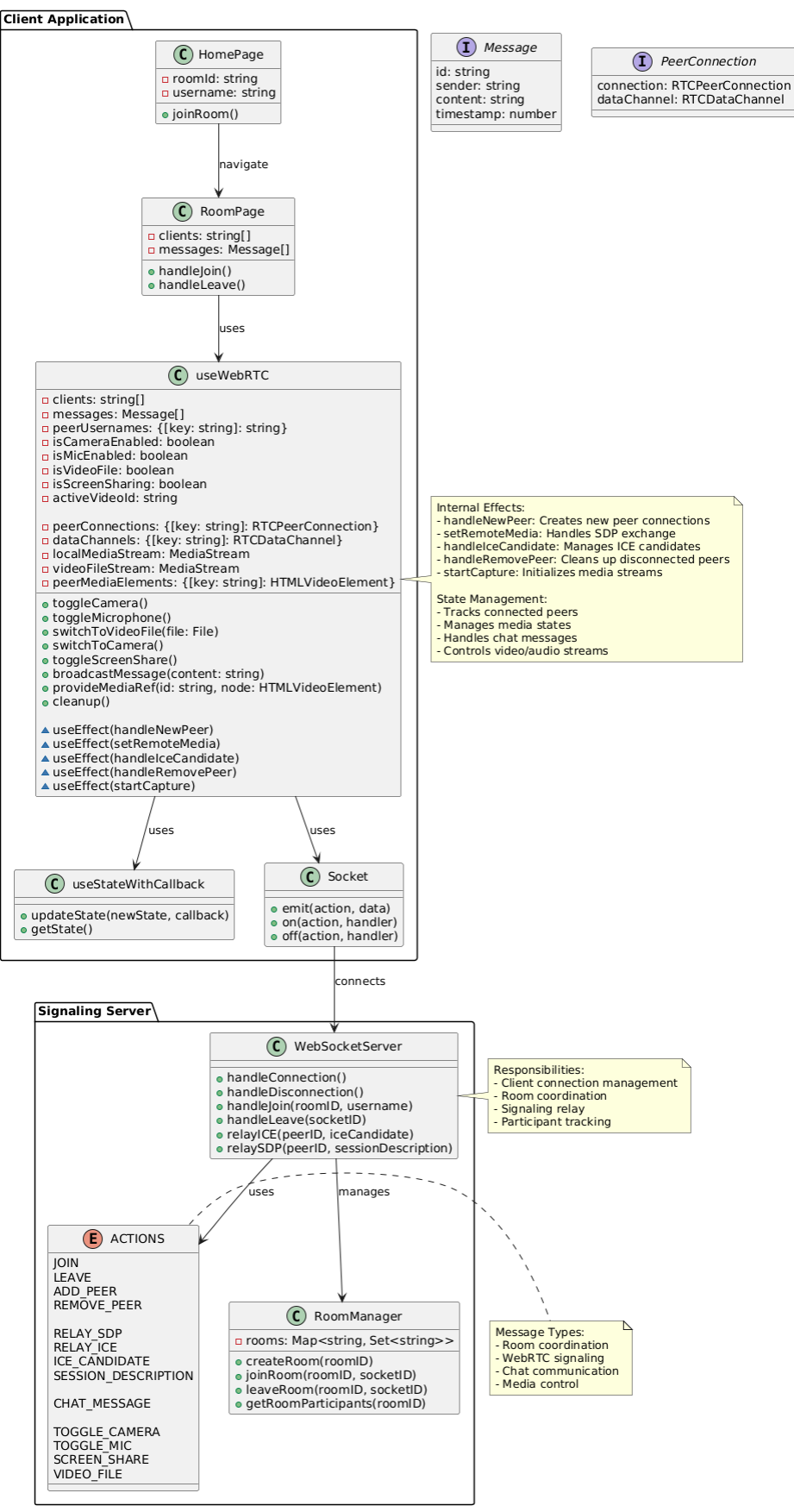


Рисунок 3.2 – UML-діаграма компонентів системи

RoomPage використовує useWebRTC для організації відеоконференції та взаємодіє з іншими учасниками через WebSocket з'єднання. Для управління станом з можливістю зворотного виклику використовується допоміжний хук useStateWithCallback. Серверна частина представлена WebSocketServer, який відповідає за обробку підключень клієнтів та координацію обміну сигнальними повідомленнями між ними. RoomManager керує життєвим циклом кімнат та учасників, зберігаючи інформацію про активні сесії. Взаємодія між клієнтом та сервером відбувається через визначений набір ACTIONS, які включають команди для управління кімнатами (JOIN, LEAVE), сигналізації WebRTC (RELAY_SDP, RELAY_ICE), обміну повідомленнями чату та контролю медіа-потоків. Система використовує декілька ключових інтерфейсів, таких як Message для структурування повідомлень чату та PeerConnection для визначення структури WebRTC з'єднань. Socket забезпечує комунікаційний канал між клієнтом та сервером, дозволяючи обмінюватися повідомленнями через визначені події. Вся ця архітектура працює разом для забезпечення безперебійної роботи відеоконференції, де WebSocket використовується для сигналізації та координації, а WebRTC забезпечує пряму передачу медіа-даних між учасниками.

3.2 Основні алгоритми роботи застосунку

3.2.1 Алгоритм створення нової кімнати та приєднання учасників

Процес створення нової кімнати та приєднання учасників реалізує базову логіку входу користувачів до відеоконференції. Алгоритм координує взаємодію між клієнтською частиною та сигнальним сервером, забезпечуючи коректне створення кімнати або приєднання до існуючої, з подальшим встановленням WebRTC з'єднань між усіма учасниками. Схема алгоритму зображена на рисунку 3.3.



Рисунок 3.3 – Схема алгоритму створення нової кімнати та приєднання учасників

3.2.2 Алгоритм встановлення WebRTC з'єднання між учасниками

Встановлення WebRTC з'єднання між учасниками відеоконференції є ключовим процесом, що забезпечує пряму передачу медіа-даних між користувачами. Алгоритм включає створення `RTCPeerConnection`, обмін SDP-пропозиціями та ICE-кандидатами через сигнальний сервер, що дозволяє встановити оптимальний шлях для передачі даних між пірамі. Схема алгоритму зображена на рисунку 3.4.

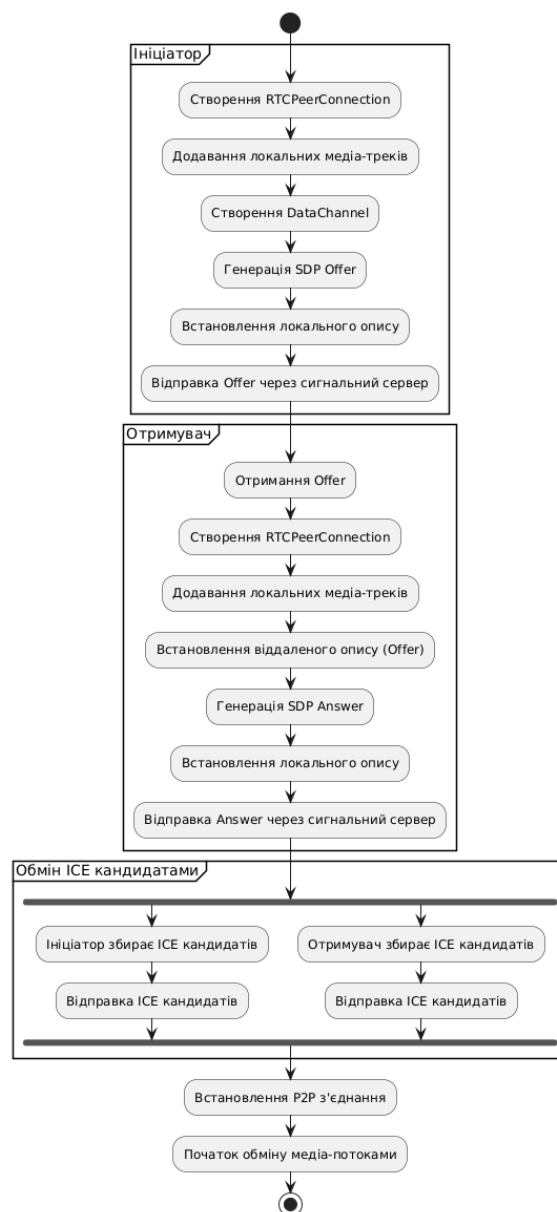


Рисунок 3.5 – Схема алгоритму встановлення WebRTC з'єднання між учасниками

3.2.3 Алгоритм обміну медіа-потокaми між користувачами

Алгоритм базується на використанні MediaStream API та RTCPeerConnection для захоплення, передачі та відтворення медіа-контенту, забезпечуючи двосторонній обмін потоками між усіма підключеними користувачами. Схема алгоритму зображена на рисунку 3.5.

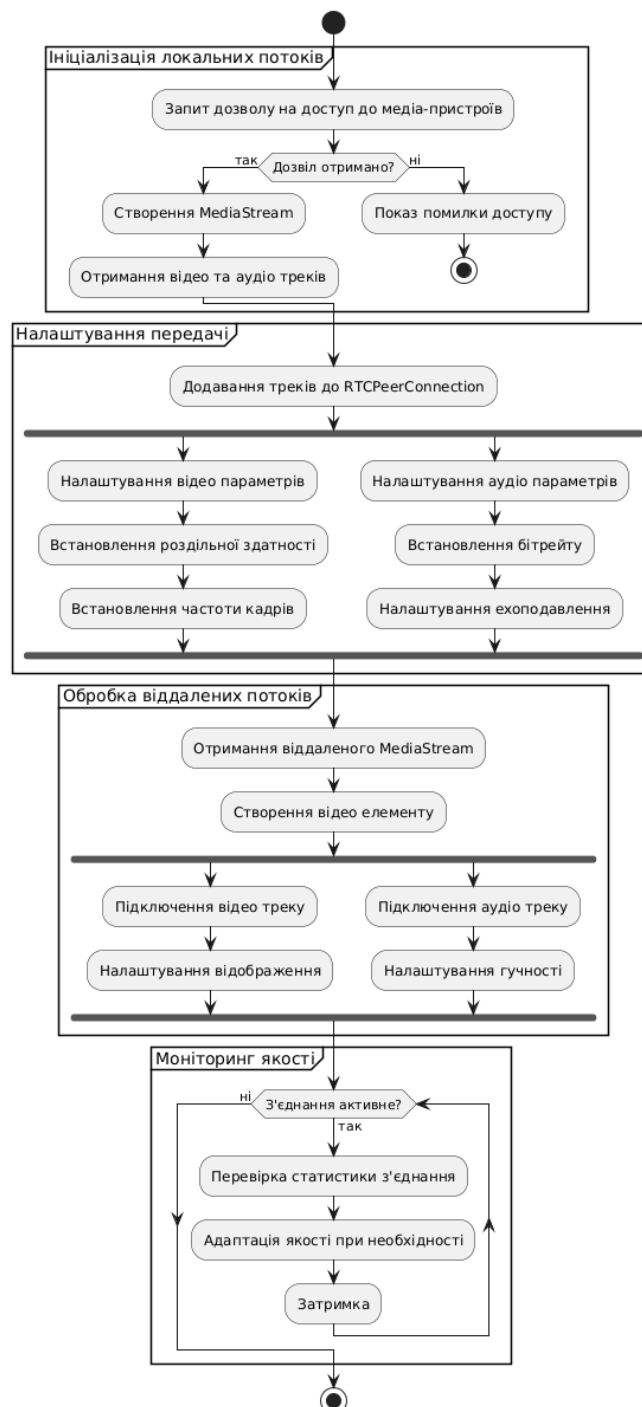


Рисунок 3.5 – Схема алгоритму обміну медіа-потокaми між користувачами

3.2.4 Алгоритм роботи текстового чату через DataChannel

Текстовий чат реалізується через RTCDataChannel, що забезпечує пряму передачу текстових повідомлень між учасниками відеоконференції без навантаження на сигнальний сервер. Алгоритм включає створення каналу даних, обробку повідомлень та їх синхронізацію між усіма учасниками кімнати. Схема алгоритму зображена на рисунку 3.6.

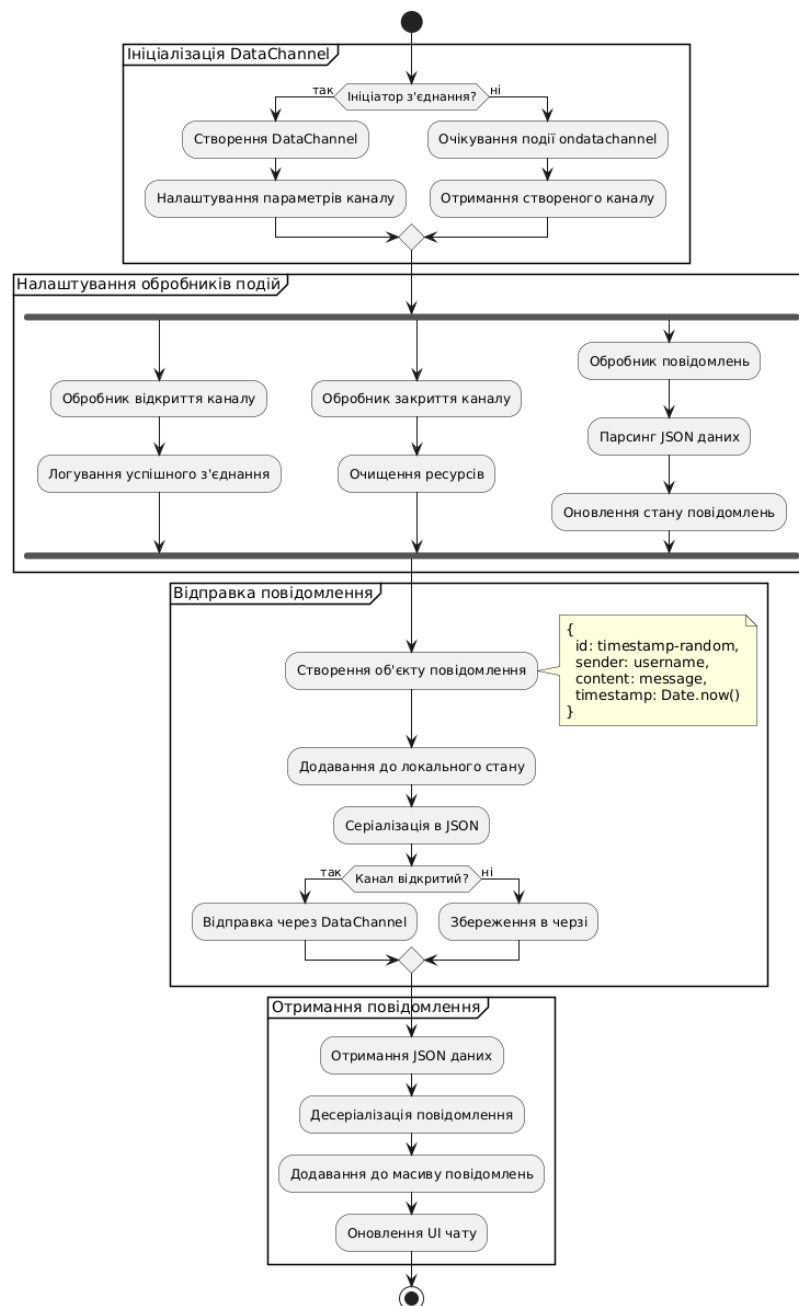


Рисунок 3.6 – Схема алгоритму роботи текстового чату через DataChannel

3.2.5 Алгоритм демонстрації екрану користувача

Демонстрація екрану реалізується через API `getDisplayMedia` та механізм заміни медіа-треків у існуючих `RTCPeerConnection`. Алгоритм забезпечує можливість користувачам ділитися вмістом свого екрану з іншими учасниками конференції, з автоматичним поверненням до веб-камери при завершенні демонстрації. Схема алгоритму зображена на рисунку 3.7.

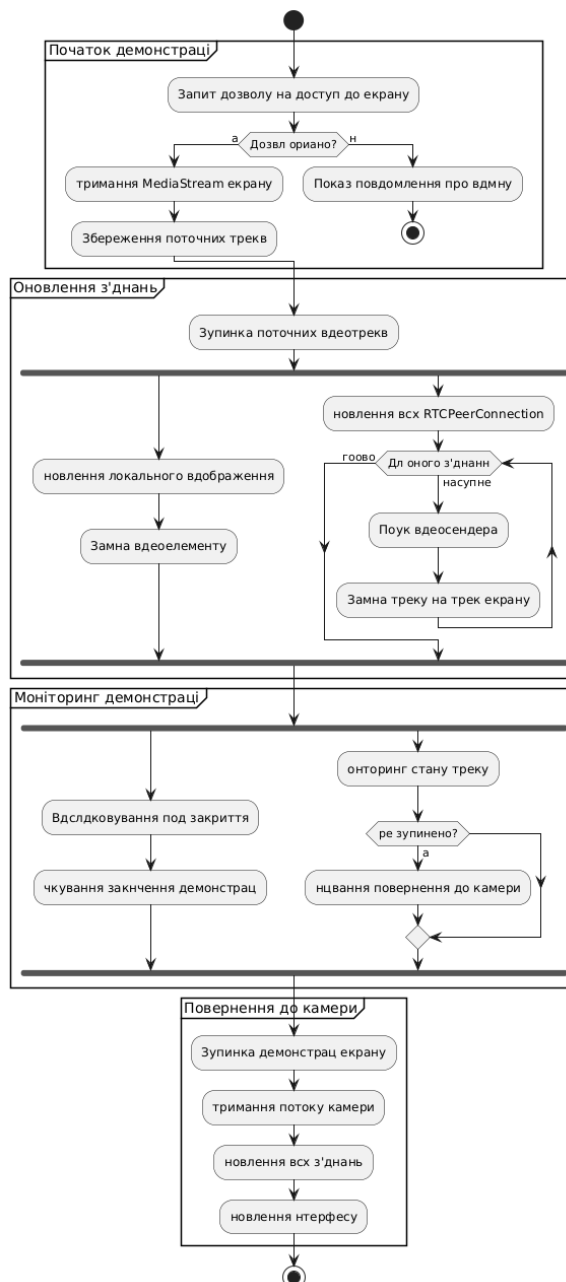


Рисунок 3.7 – Схема алгоритму демонстрації екрану користувача

3.2.6 Алгоритм відтворення відеофайлів у конференції

Відтворення відеофайлів у конференції реалізується через створення відео-елементу, захоплення його потоку за допомогою `captureStream` та передачі через існуючі WebRTC з'єднання. Цей механізм дозволяє учасникам ділитися відеофайлами в реальному часі, зберігаючи синхронізацію відтворення. Схема алгоритму зображена на рисунку 3.8.

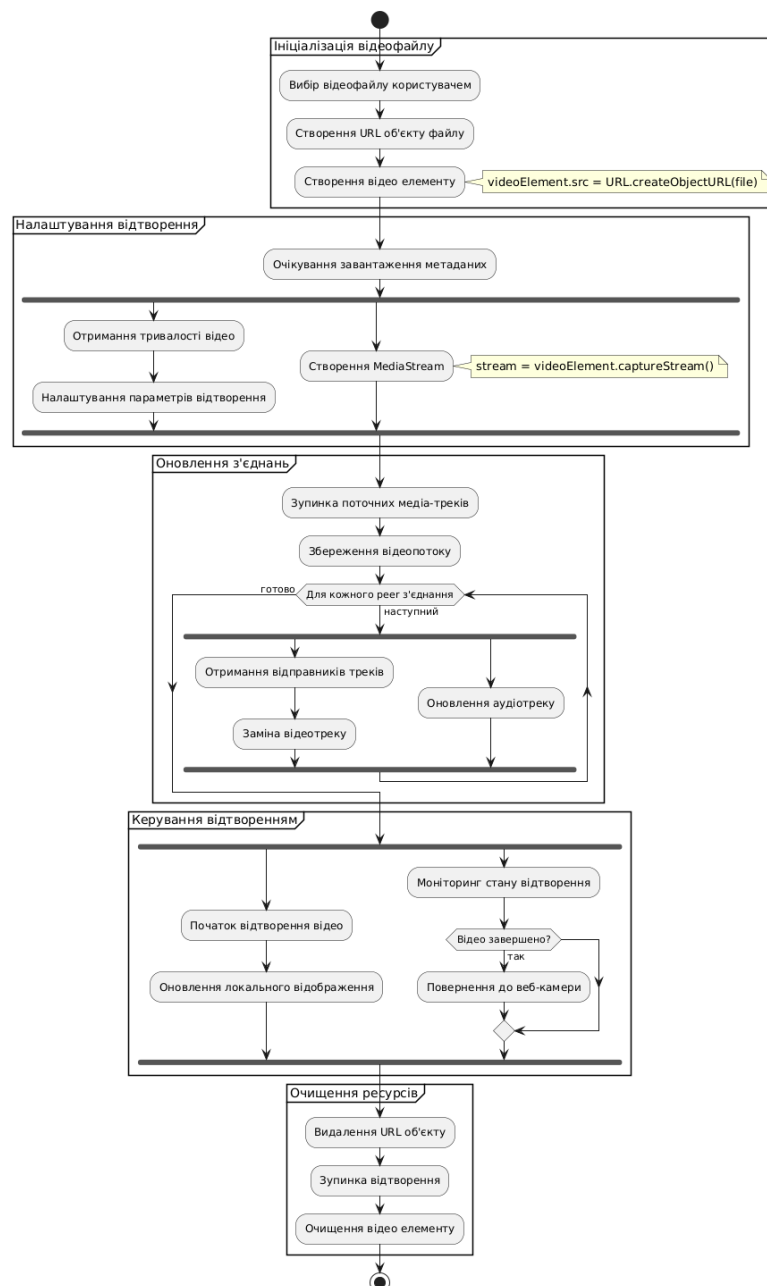


Рисунок 3.8 – Схема алгоритму відтворення відеофайлів у конференції

3.2.7 Алгоритм керування аудіо та відео потоками

Керування аудіо та відео потоками забезпечує можливість вмикання/вимикання камери та мікрофону, а також контроль за якістю передачі медіа-даних. Алгоритм реалізує базові функції управління медіа-пристроями та їх станами для всіх учасників конференції. Схема алгоритму зображена на рисунку 3.9.

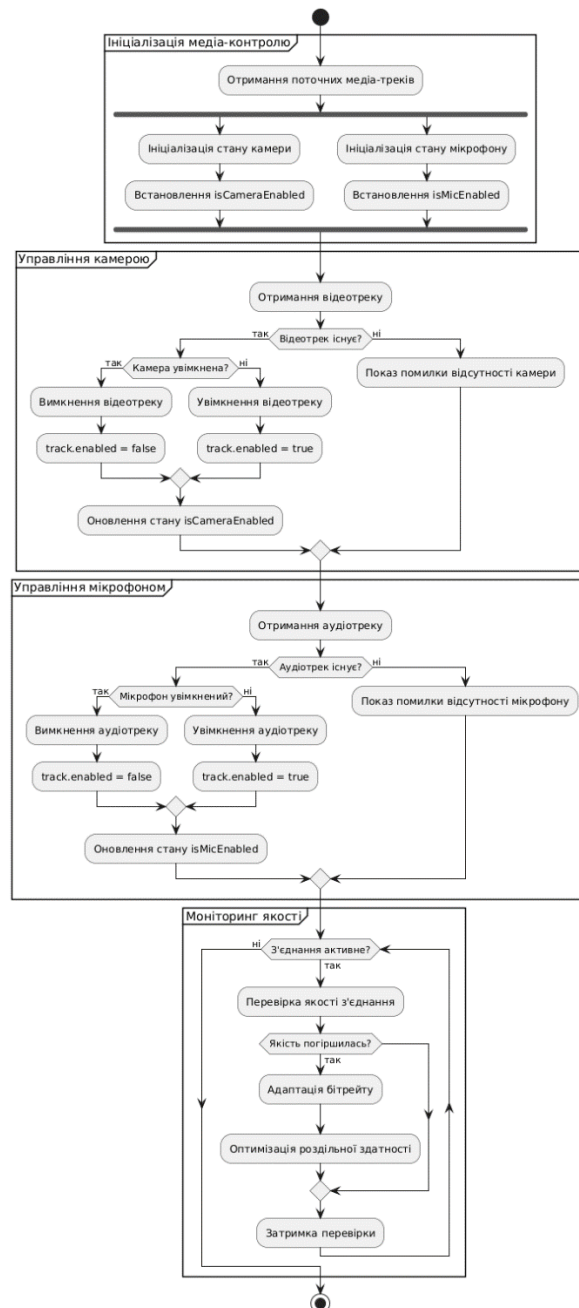


Рисунок 3.9 – Схема алгоритму керування аудіо та відео потоками

3.2.8 Алгоритм обробки від'єднання учасників

Алгоритм обробки від'єднання учасників забезпечує коректне завершення з'єднань та очищення ресурсів при виході користувача з конференції. Процес включає закриття WebRTC з'єднань, видалення медіа-елементів та оновлення стану кімнати для всіх активних учасників. Схема алгоритму зображена на рисунку 3.10.

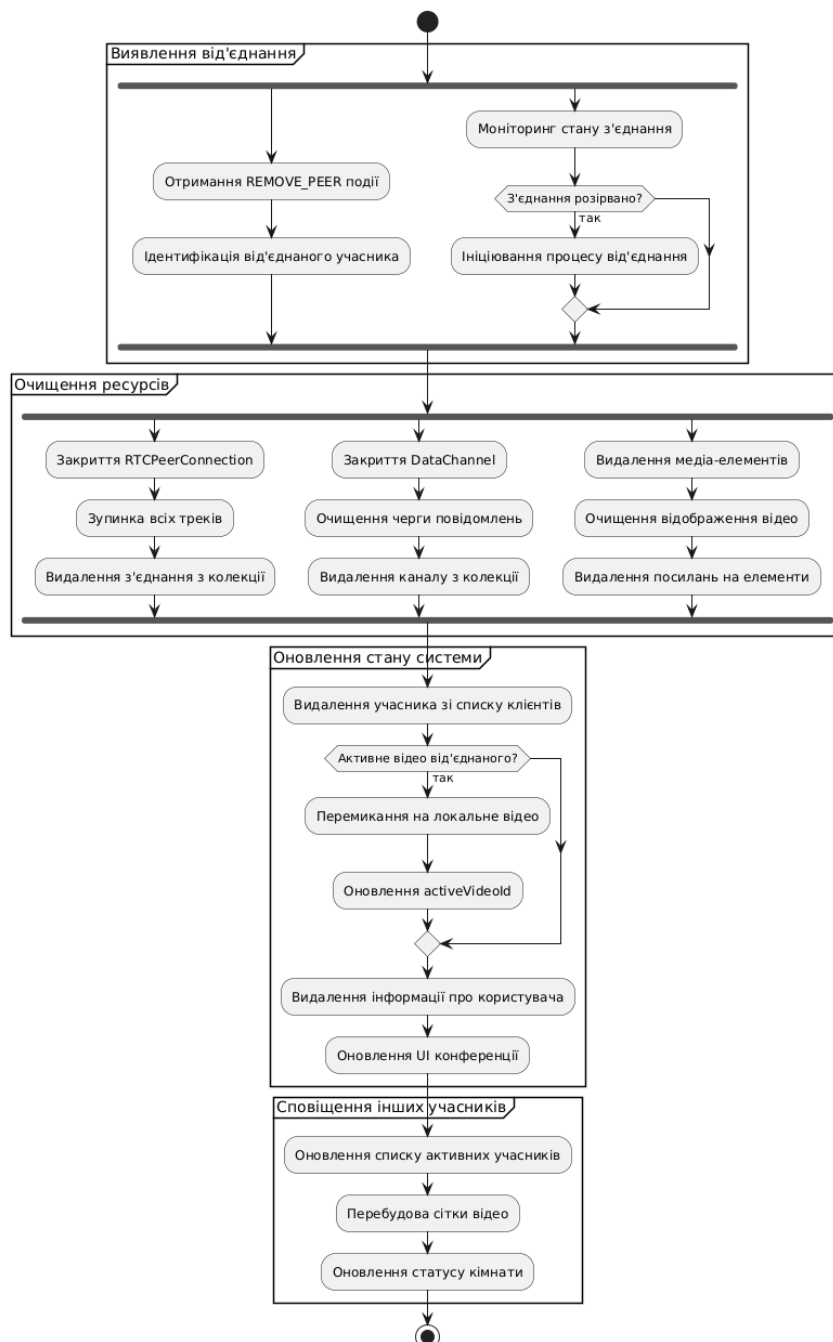


Рисунок 3.10 – Схема алгоритму обробки від'єднання учасників

3.3 Реалізація компонентів користувацького інтерфейсу

Користувацький інтерфейс додатку відеоконференцій реалізований як набір взаємопов'язаних React-компонентів, побудованих з використанням Next.js та бібліотеки компонентів shadcn/ui. Архітектура інтерфейсу розроблена з урахуванням принципів компонентного підходу, що забезпечує модульність, перевикористання коду та легкість підтримки. Основні компоненти системи включають сторінку входу для створення або приєднання до кімнати, компонент відеоконференції для відображення учасників та їх медіа-потоків, а також панель керування медіа-можливостями. Всі компоненти підтримують адаптивний дизайн, забезпечують доступність згідно з WAI-ARIA стандартами та надають користувачам інтуїтивно зрозумілий інтерфейс для взаємодії з функціями відеоконференції.

3.3.1 Компонент входу в кімнату та створення ідентифікатора

Компонент HomePage реалізує початковий екран додатку, де користувач може створити нову кімнату або приєднатися до існуючої. Використовує базові UI компоненти з бібліотеки shadcn/ui для забезпечення консистентного дизайну. При відсутності введеного ідентифікатора автоматично генерує новий за допомогою crypto.randomUUID(). Основні змінні стану компонента описані у таблиці 3.1.

Таблиця 3.1 – Опис змінних стану компонента HomePage

Змінна	Тип	Обов'язковий	Опис
router	NextRouter	Так	Next.js роутер для навігації
defaultRoomId	string	Ні	Початковий ID кімнати

Продовження таблиці 3.1

Змінна	Тип	Обов'язковий	Опис
defaultUsername	string	Так	Початкове ім'я користувача
onJoinRoom	(roomId: string, username: string) => void	Ні	Callback при вході в кімнату

3.3.2 Компонент відеоконференції та розміщення учасників

Компонент RoomPage є основним компонентом відеоконференції, який відповідає за відображення відеопотоків учасників, управління активним відео та розміщення елементів інтерфейсу. Використовує CSS Grid для адаптивного розташування відео учасників та забезпечує плавну анімацію при зміні кількості учасників. Основні змінні стану компонента описані у таблиці 3.2.

Таблиця 3.2 – Опис змінних стану компонента RoomPage

Змінна	Тип	Обов'язковий	Опис
roomId	string	Так	ID кімнати
username	string	Так	Ім'я поточного користувача
clients	string[]	Так	Масив ID підключених клієнтів
activeVideoId	string	Так	ID активного відео
peerUsernames	Record<string, string>	Так	Імена учасників

Продовження таблиці 3.2

Змінна	Тип	Обов'язковий	Опис
onVideoSelect	(clientId: string) => void	Так	Обробник вибору відео
provideMediaRef	(id: string, node: HTMLVideoElement) => void	Так	Референс для відео елементів

3.3.3 Компоненти керування медіа-можливостями

Компонент MediaControls надає інтерфейс для управління всіма медіа-можливостями користувача. Включає кнопки для керування камерою, мікрофоном, демонстрацією екрану та відтворенням відеофайлів. Реалізує візуальний фідбек для кожної дії та підтримує клавіатурні скорочення. Основні змінні стану компонента описані у таблиці 3.3.

Таблиця 3.3 – Опис змінних стану компонента RoomPage

Змінна	Тип	Обов'язковий	Опис
isCameraEnabled	boolean	Так	Стан камери
isMicEnabled	boolean	Так	Стан мікрофону
isScreenSharing	boolean	Так	Стан демонстрації екрану
isVideoFile	boolean	Так	Стан відтворення відео
onToggleCamera	() => void	Так	Перемикач камери

Продовження таблиці 3.3

Змінна	Тип	Обов'язковий	Опис
onToggleMicrophone	() => void	Так	Перемикач мікрофону
onToggleScreenShare	() => void	Так	Перемикач демонстрації
onVideoFileSelect	(file: File) => void	Так	Обробник вибору відео
disabled	boolean	Ні	Стан деактивації кнопок

3.4 Реалізація користувацьких хуків

3.4.1 Хук useWebRTC та його функціональність

Хук useWebRTC є складним користувацьким хуком, який забезпечує повну функціональність WebRTC для відеоконференцій. Він управляє peer-to-peer з'єднаннями, обміном медіа-потоків (відео та аудіо), підтримує чат через data channels, дозволяє перемикатися між камерою та демонстрацією екрану, а також надає можливість трансляції відеофайлів. Хук інкапсулює всю складну логіку WebRTC, надаючи простий інтерфейс для компонентів React.

В основі хука лежать чотири ключових React хуки:

- useState: Керує реактивним станом компонента, зберігаючи та оновлюючи такі дані як список клієнтів, активне відео, повідомлення чату та стани пристроїв;
- useRef: Зберігає мутабельні значення, які не викликають ререндер при зміні, використовується для збереження WebRTC з'єднань, медіа-потоків та DOM-елементів;
- useEffect: Керує побічними ефектами, такими як встановлення WebRTC з'єднань, підписка на події сокетів та управління медіа-потоків;

– useCallback: Мемоізує функції для оптимізації продуктивності, запобігаючи зайвим ререндерам при передачі функцій як пропсів.

Опис використаних стандартних хуків наведено у таблицях 3.4-3.7.

Таблиця 3.4 – Опис використаних хуків useState у реалізації useWebRTC

Змінна/Функція	Тип/Початкове значення	Призначення	Додаткова інформація
activeVideoId	string (LOCAL_VIDEO)	ID активно відтворюваного відео (локальне/peer)	Контролює головний екран
messages	Message[] ([])	Масив повідомлень чату	Зберігає історію чату
peerUsernames	{[id]: string} ({LOCAL_VIDEO: username})	Словник імен користувачів (peerID → username)	Включає локальне ім'я
isCameraEnabled	boolean (true)	Стан камери (увімкнена/вимкнена)	Синхронізується з треком
isMicEnabled	boolean (true)	Стан мікрофона (увімкнений/вимкнений)	Синхронізується з треком
isVideoFile	boolean (false)	Чи використовується відеофайл замість камери	Перемикає режими відео
selectedVideoFile	File null (null)	Обраний файл для відтворення	Зберігає файл відео
isScreenSharing	boolean (false)	Стан демонстрації екрану (активний/неактивний)	Контролює режим скріншару

Таблиця 3.5 – Опис використаних хуків useRef у реалізації useWebRTC

Змінна/Функція	Тип/Початкове значення	Призначення	Життєвий цикл
peerConnections	{[id]: RTCPeerConnection} ({})	Словник з'єднань WebRTC (peerID → об'єкт)	Зберігає між рендерами
dataChannels	{ [id]: RTCDataChannel } ({})	Канали даних для чату (peerID → канал)	Закриваються при cleanup

Продовження таблиці 3.5

Змінна/Функція	Тип/Початкове значення	Призначення	Життєвий цикл
localMediaStream	MediaStream null (null)	Локальний медіапотік (камера/екран)	Зупиняється при cleanup
videoFileStream	MediaStream null (null)	Потік відеофайлу	Зупиняється при перемиканні
peerMediaElements	{[id]: HTMLVideoElement} ({LOCAL_VIDEO: null})	Словник DOM-елементів відео (peerID → елемент)	Оновлюється через колбеки
activeVideoRef	HTMLVideoElement null (null)	Посилання на активний відеоелемент (головний екран)	Використовує provideActiveMediaRef
videoElementRef	HTMLVideoElement null (null)	Допоміжний елемент для відтворення відеофайлів	Створюється тимчасово

Таблиця 3.6 – Опис використаних хуків useEffect у реалізації useWebRTC

Залежності	Призначення	Ключові операції
[retryKey]	Повне перепідключення	Виклик cleanup() при зміні ключа перепідключення
[username]	Синхронізація локального імені	Оновлення peerUsernames для LOCAL_VIDEO
[activeVideoId, clients]	Оновлення активного відео	Встановлення srcObject для головного відео
[addNewClient, isScreenSharing]	Обробка нових peer-ів	Створення RTCPeerConnection, налаштування дата-каналів, ICE-кандидатів
[]	Обробка SDP (Session Description)	Відповідь на офер, створення answer
[]	Обробка ICE-кандидатів	Додавання кандидатів до з'єднань
[updateClients, activeVideoId]	Видалення peer-ів	Закриття з'єднань, видалення зі списку клієнтів, оновлення активного відео

Продовження таблиці 3.6

Залежності	Призначення	Ключові операції
[cleanup]	Глобальний cleanup (наприклад, закриття вкладки)	Додавання обробника beforeunload, виклик cleanup при демонтажі
[roomID, username, password, addNewClient, cleanup, retryKey]	Ініціалізація медіа	Запуск камери, приєднання до кімнати через сокет
[clients.length]	Корекція гучності локального відео	Встановлення volume=1 для локального відеоелемента

Таблиця 3.7 – Опис використаних хуків useCallback у реалізації useWebRTC

Функція	Залежності	Призначення
cleanup	[username, updateClients]	Повне очищення ресурсів: зупинка треків, закриття з'єднань, скидання станів
addNewClient	[updateClients]	Безпечне додавання нового клієнта до списку
broadcastMessage	[username]	Розсилка повідомлень через всі відкриті канали даних
provideMediaRef	[]	Прив'язка DOM-елементів відео до словника peerMediaElements
provideActiveMediaRef	[]	Прив'язка головного відеоелемента до activeVideoRef
toggleCamera	[]	Перемикання стану відеотреку локального потоку
toggleMicrophone	[]	Перемикання стану аудіотреку локального потоку
switchToVideoFile	[activeVideoId]	Перехід на відеофайл: створення потоку, заміна треків у з'єднаннях

Продовження таблиці 3.7

Функція	Залежності	Призначення
switchToCamera	[activeVideoId]	Повернення до камери: зупинка файлового потоку, отримання нового потоку
toggleScreenShare	[isScreenSharing, activeVideoId, switchToCamera]	Перемикання демонстрації екрану: захоплення екрану/повернення до камери

3.4.2 Хук useStateWithCallback для управління станом

Хук useStateWithCallback є розширенням стандартного useState, який додає можливість виконання callback-функції після оновлення стану. Це особливо корисно в ситуаціях, коли потрібно виконати певні дії відразу після того, як стан був оновлений, наприклад, для синхронізації з DOM або виконання side-effects, що залежать від нового значення стану. Опис використаних стандартних хуків наведено у таблиці 3.8.

Таблиця 3.8 – Опис використаних стандартних хуків у реалізації useStateWithCallback

Хук	Приклад з коду	Призначення	Використання
useState	<code>const [state, setState] = useState<T>(initialState)</code>	Основне зберігання стану	Зберігає поточне значення стану типу T
useRef	<code>const cbRef = useRef<Callback<T>></code>	<code>null>(null)`</code>	Зберігання callback-функції
useEffect	<code>useEffect(() => { if (cbRef.current) { cbRef.current(state); } }, [state])</code>	Виконання callback після оновлення	Викликає збережений callback після зміни стану
useCallback	<code>const updateState = useCallback((newState: SetStateAction<T>, cb?: Callback<T>)</code>	Мемоізація функції оновлення	Оптимізована функція оновлення стану з callback

3.5 Реалізація WebSocket взаємодії

У проекті реалізована WebSocket взаємодія для забезпечення комунікації між учасниками відеоконференції. WebSocket використовується як транспортний механізм для обміну сигнальними повідомленнями між пірами (peer-to-peer) у контексті WebRTC з'єднань.

3.5.1 Види WebSocket повідомлень

У системі реалізовано кілька типів WebSocket повідомлень для забезпечення різних аспектів функціонування відеоконференції:

а) повідомлення приєднання (JOIN) - відправляється при вході користувача в кімнату, містить ідентифікатор кімнати, ім'я користувача та опціональний пароль;

б) сигнальні повідомлення для WebRTC:

1) RELAY_SDP - для обміну SDP (Session Description Protocol) описами;

2) RELAY_ICE - для обміну ICE (Interactive Connectivity Establishment) кандидатами;

в) повідомлення управління учасниками:

1) ADD_PEER - сповіщення про нового учасника;

2) REMOVE_PEER - сповіщення про вихід учасника;

г) повідомлення чату - для обміну текстовими повідомленнями між учасниками через Data Channel.

3.5.2 Управління учасниками кімнати

Для керуванням з'єднань учасників однієї кімнати було розроблено комплексний компонент управління учасниками, який складається з:

- менеджера учасників, що відповідає за підтримку актуального списку користувачів;
- підсистеми управління медіа-потоками кожного учасника;
- компонента синхронізації стану відео та аудіо потоків;
- модуля керування ресурсами системи при зміні складу учасників.

3.6 Висновки за розділом

Проект реалізовано як клієнтський додаток на Next.js, що підкреслює його орієнтацію на сучасні веб-технології та масштабованість. Модульний принцип побудови, де кожен компонент відповідає за окрему функціональність, забезпечує легкість підтримки та розширення. Завдяки використанню WebRTC для прямого обміну медіа-даними та WebSocket для сигналізації, система ефективно розділяє потоки даних, оптимізуючи навантаження на сервер та забезпечуючи високу продуктивність. Файлова організація відповідає стандартам Next.js, що робить структуру проекту зрозумілою та зручною для розробки.

Система складається з чотирьох ключових компонентів: клієнтського додатку Next.js, сигнального WebSocket сервера, WebRTC підсистеми та системи управління медіа. Клієнтський додаток відповідає за інтерфейс користувача та керування медіа-потоками, тоді як сигнальний сервер координує учасників та обмінюється службовими повідомленнями. WebRTC забезпечує прямі P2P з'єднання для передачі відео, аудіо та даних чату, а система управління медіа керує пристроями користувача. Взаємодія між цими компонентами базується на гібридному підході: WebSocket використовується для сигналізації, а WebRTC — для безпосередньої передачі медіа-даних.

Описані алгоритми охоплюють всі ключові аспекти функціонування системи: від створення кімнат та приєднання учасників до встановлення WebRTC з'єднань, обміну медіа-потоками, роботи текстового чату через DataChannel, а також демонстрації екрану та відтворення відеофайлів.

Інтерфейс розроблений за допомогою React-компонентів, Next.js та бібліотеки shadcn/ui, що гарантує сучасний дизайн, адаптивність та доступність. Ключовий користувацький хук useWebRTC інкапсулює всю складну логіку WebRTC, надаючи компонентам React простий та зручний інтерфейс для управління з'єднаннями та медіа-потокami.

4 КЕРІВНИЦТВО ПРОГРАМІСТА

4.1 Призначення та умови використання системи

Веб-сайт призначений для проведення відео-конференцій в реальному часі (RTC - Real-Time Communication).

Для запуску в режимі розробки необхідно:

- а) відкрити проект в Visual Studio Code або іншому редакторі коду;
- б) запустити два сервери:
 - 1) основний веб-сервер: `npm run dev`;
 - 2) WebSocket сервер: `npm run start-socket`.

Веб-сайт розроблений як односторінковий додаток (SPA) з використанням сучасного фреймворку Next.js 15.3.2 та React 19.

Основні технічні характеристики застосунку наступні:

- фронтенд: React.js, Next.js, TailwindCSS;
- бекенд: Node.js з WebSocket (Socket.IO) для реалізації відео-зв'язку;
- стилізація: TailwindCSS з додатковими компонентами від Radix UI;
- мова програмування: TypeScript.

Функціональні можливості:

- створення відео-конференцій;
- підключення до існуючих конференцій;
- передача відео та аудіо в реальному часі;
- адаптивний інтерфейс для різних розмірів екрану.

Особливості реалізації:

- використання WebRTC через бібліотеку freeice для P2P відео-зв'язку;
- Socket.IO для сигналізації та координації з'єднань;
- сучасний UI з використанням компонентів Radix UI (shadcn/ui);
- повна підтримка TypeScript для безпечної розробки;

- оптимізація продуктивності за допомогою Next.js та Turbopack.

Умови експлуатації:

- наявність стабільного інтернет-з'єднання;
- доступ до камери та мікрофону;
- сучасний веб-браузер з підтримкою WebRTC;
- достатня пропускна здатність мережі для передачі відео та аудіо.

4.2 Характеристики розробленого веб-застосунку

Сайт може виконувати наступні функції:

- створення відеоконференцій;
- приєднання до існуючих конференцій;
- обмін текстовими повідомленнями в чаті;
- передача аудіо та відео в реальному часі;
- керування власними медіа-потокami (вмикання/вимикання камери та мікрофону).

Для розробки даного веб-застосунку використано компонентно-орієнтовану модель на базі React та TypeScript, а також WebRTC для передачі медіа-даних у реальному часі.

Компонентно-орієнтована модель – це підхід до розробки інтерфейсу, де кожен елемент представлений як незалежний компонент з власним станом та логікою. Компоненти можуть взаємодіяти між собою через props та події, утворюючи складну ієрархічну структуру застосунку.

В процесі проектування даного програмного продукту було виділено наступні основні компоненти:

Компонент "Chat" зберігає та керує повідомленнями в чаті. Основні властивості:

- messages: масив повідомлень;
- onSendMessage: функція для надсилання нових повідомлень;
- currentUsername: ім'я поточного користувача.

Кожне повідомлення (інтерфейс Message) має наступну структуру:

- id: унікальний ідентифікатор;
- content: текст повідомлення;
- sender: відправник;
- timestamp: час надсилання.

Для реалізації відеозв'язку використовується хук useWebRTC, який керує:

- встановленням P2P з'єднань між учасниками;
- обміном медіа-потоків;
- сигналізацією через WebSocket;
- керуванням станом з'єднання.

Взаємодія між компонентами відбувається через:

– props для передачі даних від батьківських до дочірніх компонентів;

- callback-функції для передачі подій знизу вгору;
- WebSocket для комунікації з сервером.

Для стилізації використовується Tailwind CSS, що дозволяє:

- створювати адаптивний дизайн;
- застосовувати готові утиліти для стилів;
- легко модифікувати зовнішній вигляд компонентів.

Всі компоненти типізовані за допомогою TypeScript, що забезпечує:

- статичну типізацію даних;
- автодоповнення в IDE;
- виявлення помилок на етапі компіляції;
- кращу підтримку коду.

4.3 Структура WebRTC з'єднань

WebRTC з'єднання в системі реалізовані за допомогою хука `useWebRTC`, який забезпечує повнодуплексний зв'язок між учасниками конференції. Структура з'єднань включає наступні компоненти:

а) `peer connection management`:

- 1) створення `RTCPeerConnection` для кожного учасника;
- 2) використання STUN серверів (`stun.l.google.com:19302`) для NAT traversal;
- 3) керування ICE кандидатами для встановлення з'єднання;
- 4) моніторинг стану з'єднання (`connectionState`, `iceConnectionState`, `signalingState`);

б) медіа-потіки:

- 1) `localMediaStream`: локальний потік з камери та мікрофону;
- 2) `videoFileStream`: потік для відтворення відео-файлів;
- 3) можливість керування станом пристроїв (камера, мікрофон);
- 4) підтримка screen sharing;

в) канали даних:

- 1) створення `DataChannel` для текстового чату;
- 2) обмін повідомленнями між учасниками;

г) сигналізація:

- 1) використання `WebSocket` для обміну SDP та ICE кандидатами;
- 2) підтримка кімнат (`roomID`);
- 3) автоматичне перепідключення при втраті з'єднання.

4.4 Структурна схема компонентів системи

Система складається з серверної та клієнтської частини. Дерево каталогів проєкту зображено на рисунку 4.1.

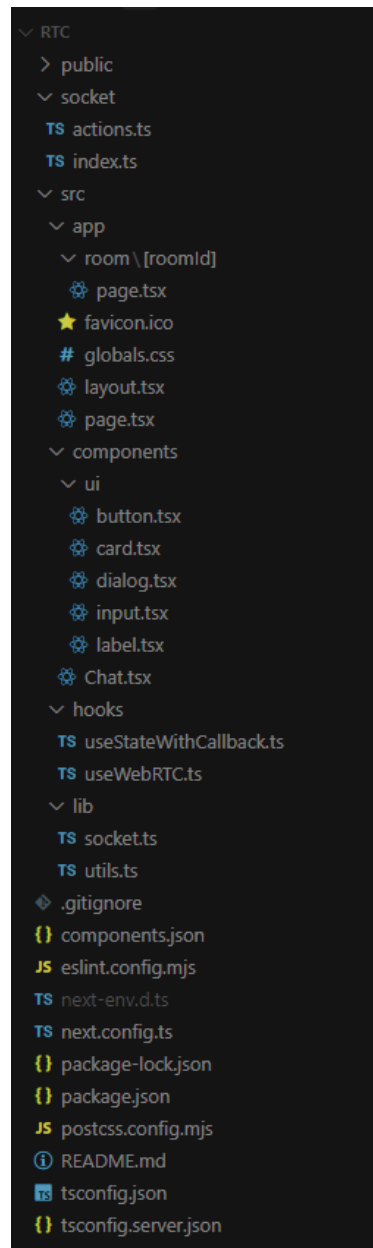


Рисунок 4.1 – Дерево каталогів проєкту

Структурна схема застосунку зображена на рисунку 4.2. Представлена структурна схема відображає архітектуру веб-застосунку для відеоконференцій, реалізованого з використанням технології WebRTC. Схема

розділена на дві основні частини: клієнтську (Client Side) та серверну (Server Side). На клієнтській стороні центральним компонентом виступає Next.js App, який координує роботу всіх інших модулів. WebRTC Manager відповідає за встановлення та підтримку peer-to-peer з'єднань між користувачами, керування медіа-потоків та обмін даними. Socket Client забезпечує зв'язок із сервером через WebSocket протокол для сигналізації та координації учасників конференції. UI Components відповідає за відображення інтерфейсу користувача та взаємодію з користувачем. На серверній стороні WebSocket Server обробляє підключення клієнтів, Room Manager керує віртуальними кімнатами для конференцій, а Signaling Service забезпечує обмін службовими повідомленнями між учасниками для встановлення WebRTC з'єднань. STUN/TURN Servers представляють зовнішні сервіси, необхідні для подолання NAT та встановлення прямих з'єднань між користувачами.

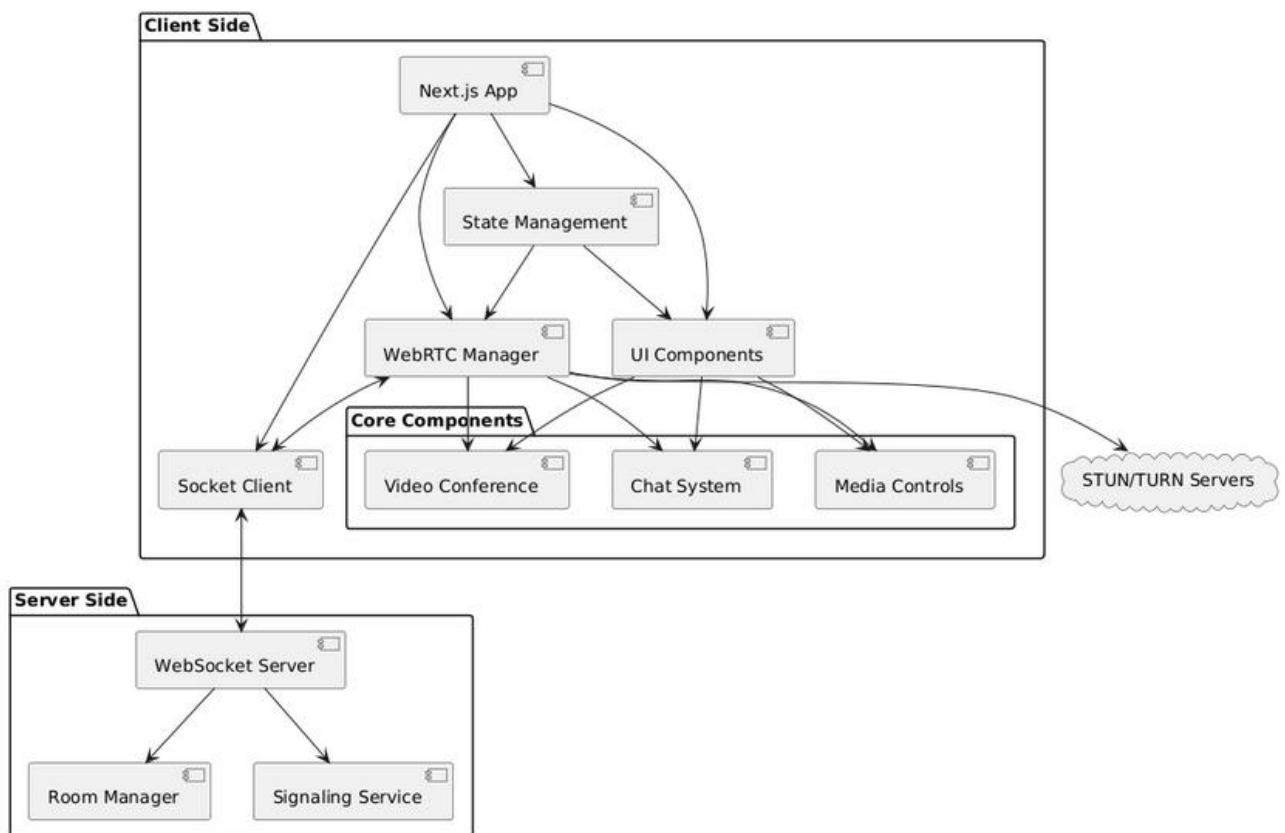


Рисунок 4.2 – Структурна схема застосунку

4.5 Вимоги до системи та браузера

Системні вимоги проєкту наступні:

а) для розробки:

- встановене середовище виконання Node.js;
- пакетний менеджер npm або yarn;
- Visual Studio Code або інший редактор коду;
- Git для версіонування.

б) для тестування і відладки:

- сучасний веб-браузер (Chrome, Firefox, Safari, Edge);
- веб-камера та мікрофон для участі у відеоконференціях;
- стабільне інтернет-з'єднання (рекомендована швидкість від 1 Мбіт/с);
- пристрій з підтримкою WebRTC.

4.5 Висновки за розділом

У даному розділі було детально описано керівництво програміста для веб-застосунку відеоконференцій. Розглянуто основні аспекти розробленої системи, її призначення та умови використання. Система реалізована як сучасний односторінковий додаток з використанням передових технологій веб-розробки, включаючи Next.js, React, TypeScript та WebRTC.

Описано ключові характеристики веб-застосунку, включаючи його функціональні можливості та особливості реалізації. Детально розглянуто компонентно-орієнтовану архітектуру системи, що забезпечує модульність, масштабованість та легкість підтримки коду. Особливу увагу приділено структурі WebRTC з'єднань, яка забезпечує надійний двосторонній зв'язок між учасниками конференції.

Представлено структурну схему компонентів системи, яка чітко демонструє взаємозв'язки між клієнтською та серверною частинами

застосунку. Схема відображає всі ключові компоненти системи та їх взаємодію, що дозволяє зрозуміти архітектурні рішення, використані при розробці.

Визначено системні вимоги та вимоги до браузера, що є необхідними для успішної розробки, тестування та експлуатації системи. Це включає як вимоги до середовища розробки, так і вимоги до кінцевих користувачів системи.

Загалом, розроблений веб-застосунок представляє собою сучасне технологічне рішення для проведення відеоконференцій, що відповідає всім сучасним стандартам веб-розробки та забезпечує високу якість користувацького досвіду. Використання TypeScript, компонентної архітектури та сучасних інструментів розробки гарантує надійність, безпеку та масштабованість системи.

5 КЕРІВНИЦТВО КОРИСТУВАЧА

5.1 Призначення та можливості системи

Веб-застосунок призначений для проведення відеоконференцій в режимі реального часу через браузер. Система дозволяє користувачам створювати віртуальні кімнати для спілкування, приєднуватися до існуючих конференцій та обмінюватися аудіо, відео та текстовими повідомленнями.

Основні можливості системи:

а) управління відеоконференціями:

- 1) створення нової відеоконференції з унікальним ідентифікатором кімнати;
- 2) приєднання до існуючої конференції за ідентифікатором;
- 3) можливість покинути конференцію в будь-який момент;

б) управління медіа-потокami:

- 1) вмикання/вимикання власної веб-камери;
- 2) вмикання/вимикання власного мікрофону;
- 3) демонстрація екрану (screen sharing);
- 4) перегляд відео інших учасників;

в) текстовий чат:

- 1) обмін текстовими повідомленнями з усіма учасниками конференції;
- 2) відображення часу надсилання повідомлень;
- 3) ідентифікація відправника кожного повідомлення;

г) інтерфейс користувача:

- 1) адаптивний дизайн для різних розмірів екрану;
- 2) інтуїтивно зрозуміле розташування елементів керування;
- 3) можливість перемикання між різними режимами відображення відео;

д) безпека та приватність:

- 1) кожна конференція має унікальний ідентифікатор;

2) необхідність надання дозволу на використання камери та мікрофону;

е) додаткові функції:

- 1) автоматичне відновлення з'єднання при втраті зв'язку;
- 2) підтримка різних браузерів (Chrome, Firefox, Safari, Edge).

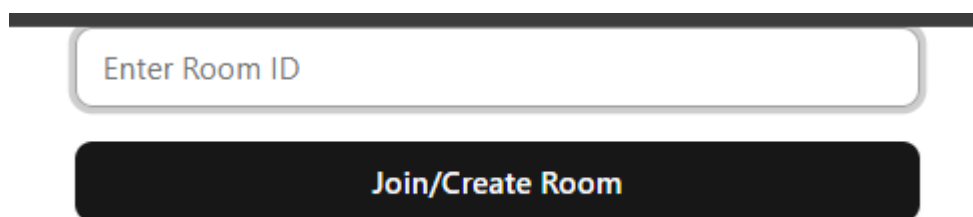
Система розроблена з урахуванням сучасних вимог до веб-застосунків та забезпечує:

- високу якість відео та аудіо зв'язку;
- мінімальну затримку при передачі даних;
- стабільність роботи;
- простоту використання;
- доступність з будь-якого пристрою з підтримкою WebRTC.

Для початку роботи з системою користувачу не потрібно встановлювати додаткове програмне забезпечення - достатньо мати сучасний веб-браузер та пристрої введення (камеру та мікрофон). Система автоматично адаптується під доступну швидкість інтернет-з'єднання та можливості пристрою користувача.

5.2 Основні сценарії роботи з системою

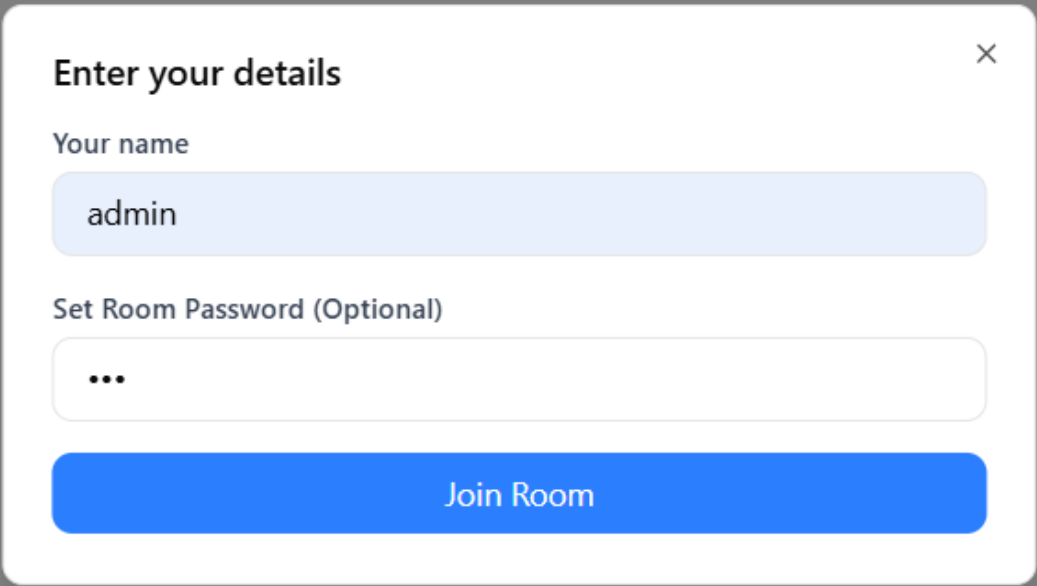
Після запуску проекту, перейдіть за адресою <http://localhost:3000/>. На рисунку 5.1 зображено сторінку входу до відеоконференції.



The image shows a web interface for joining a video conference. At the top, there is a horizontal line. Below it is a light gray rounded rectangular input field containing the text 'Enter Room ID'. Underneath the input field is a dark gray button with the text 'Join/Create Room' in white.

Рисунок 5.1 – Сторінка входу до конференцію

На сторінці входу користувач повинен ввести ID кімнати для приєднання до відеоконференції. Після введення даних та натискання кнопки "Join/Create Room", користувач потрапляє до відеокімнати, та має підтвердити вхід ввівши своє ім'я та пароль кімнати як показано на рисунку 5.2.



The image shows a modal dialog box titled "Enter your details" with a close button (X) in the top right corner. Inside the dialog, there are two input fields. The first is labeled "Your name" and contains the text "admin". The second is labeled "Set Room Password (Optional)" and contains three dots "...". Below these fields is a prominent blue button labeled "Join Room".

Рисунок 5.2 – Форма підтвердження входу у конференцію

Після підтвердження користувач потрапляє у кімнату відеоконференції (рис. 5.3)

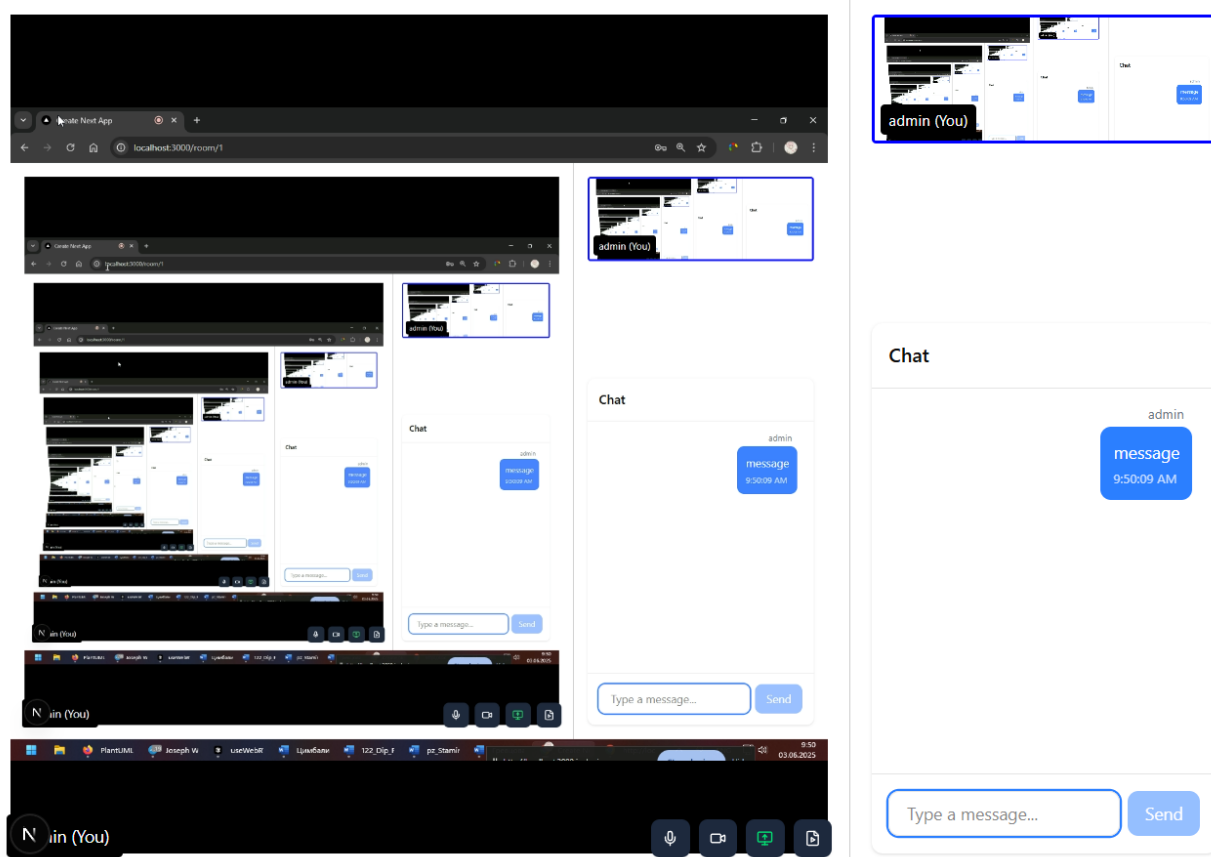


Рисунок 5.3 – Сторінка кімнати відеоконференції

У відеокімнаті доступні наступні функції:

- а) керування камерою та мікрофоном:
 - 1) увімкнення/вимкнення камери;
 - 2) увімкнення/вимкнення мікрофону;
- б) демонстрація екрану: користувач може поділитися своїм екраном з іншими учасниками конференції, натиснувши відповідну кнопку, як показано на рисунку 5.4;
- в) вдтворення відеофайлу: система підтримує можливість трансляції відеофайлу замість камери. Для цього потрібно:
 - 1) натиснути відповідну кнопку;
 - 2) обрати відеофайл з комп'ютера, це показано на рисунку 5.5;
 - 3) відео автоматично почне транслюватися всім учасникам;

г) текстовий чат: під час відеоконференції користувачі можуть обмінюватися текстовими повідомленнями. Вікно чату розташоване праворуч від відео, як показано на рисунку 5.6;

д) перегляд учасників: у верхній частині екрану відображається список усіх підключених учасників конференції з їхніми іменами;

е) активний спікер: при розмові учасника його відео автоматично стає головним у центрі екрану;

ж) вихід з конференції: для завершення відеоконференції достатньо закрити вкладку браузера.

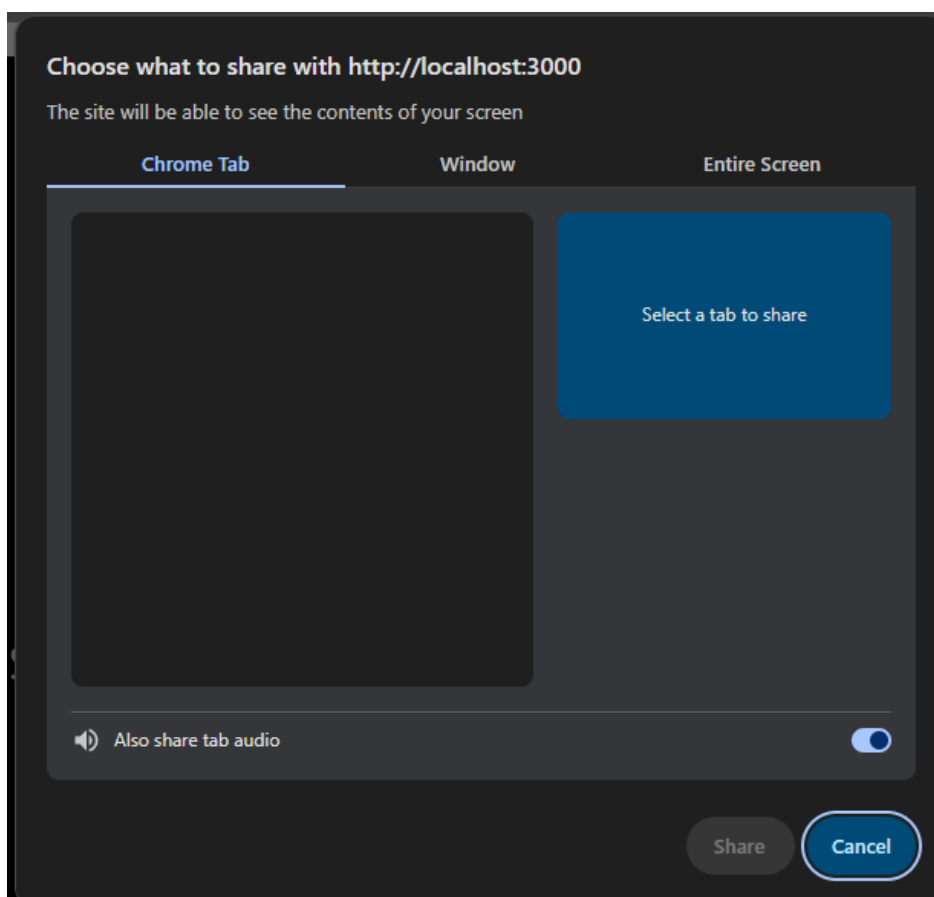


Рисунок 5.4 – Меню вибору демонстрації екрану



Рисунок 5.5 – Кнопки демонстрації екрану та вибору відеофайлу

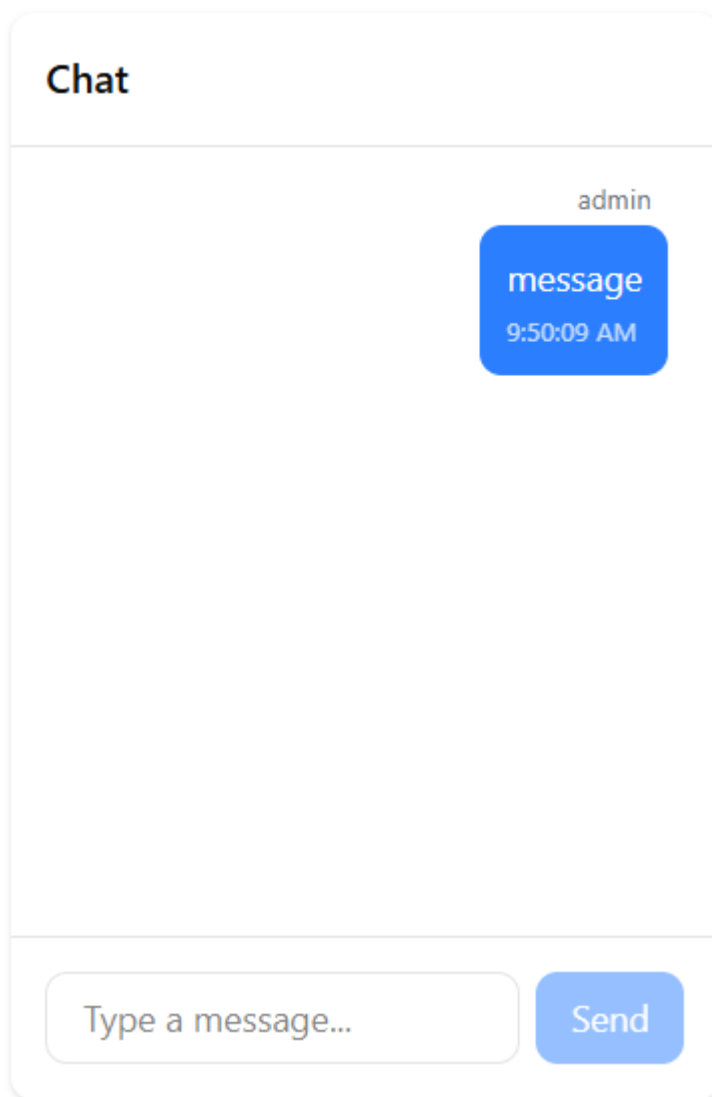


Рисунок 5.6 – Вікно чату

5.3 Вимоги до системи та браузера

Застосунок має певні вимоги для функціонування:

Системні вимоги:

- процесор: двоядерний процесор з частотою 2.0 GHz або кращий;
- оперативна пам'ять: мінімум 4 GB RAM;
- веб-камера з підтримкою роздільної здатності 720p;
- мікрофон (вбудований або зовнішній);
- динаміки або навушники;
- швидкість інтернет-з'єднання: мінімум 1 Mbps;

- стабільне підключення до мережі Інтернет.

Підтримувані операційні системи:

- Windows 10 або новіша версія;
- macOS 10.13 або новіша версія;
- Linux (Ubuntu 18.04+, Debian 10+).

Підтримувані веб-браузери:

- Google Chrome (версія 60 або новіша);
- Mozilla Firefox (версія 55 або новіша);
- Safari (версія 11 або новіша);
- Microsoft Edge (версія 79 або новіша).

Необхідні дозволи браузера:

- доступ до веб-камери;
- доступ до мікрофону;
- доступ до екрану (для демонстрації екрану).

Рекомендації для оптимальної роботи:

- використовувати навушники для уникнення відлуння;
- забезпечити тихе навколишнє середовище;
- закрити неактивні вкладки браузера;
- вимкнути невикористовувані програми;
- регулярно очищати кеш браузера;
- використовувати останню версію браузера.

Обмеження системи:

- максимальна роздільна здатність відео: 1280x720 (HD);
- кількість учасників залежить від потужності пристрою;
- необхідна підтримка WebRTC у браузері;
- необхідна підтримка getUserMedia API.

Дотримання цих вимог забезпечить стабільну роботу системи відеоконференцій.

5.4 Висновки за розділом

В розділі було детально описано процес роботи з системою відеоконференцій, включаючи вхід до конференції, керування аудіо та відео пристроями, демонстрацію екрану та обмін текстовими повідомленнями.

Розроблений інтерфейс системи є інтуїтивно зрозумілим та відповідає сучасним стандартам веб-додатків. Користувач може легко:

- приєднатися до відеоконференції, вказавши своє ім'я та ID кімнати;
- керувати своєю камерою та мікрофоном;
- ділитися екраном з іншими учасниками;
- спілкуватися в текстовому чаті;
- транслювати відеофайли.

Система працює в усіх сучасних браузерях та не потребує встановлення додаткового програмного забезпечення. Всі основні функції доступні через зрозумілі кнопки керування, а процес використання системи не повинен викликати складнощів у користувачів, які мають базові навички роботи з веб-браузером.

ВИСНОВКИ

У процесі виконання дипломної кваліфікаційної роботи було розроблено систему відеоконференцій на основі технології WebRTC.

Після тестування та налагодження функціонування веб-додатку можна зробити наступні висновки:

- функціонал системи відеоконференцій повністю відповідає вимогам технічного завдання;
- система має зручний та інтуїтивно зрозумілий інтерфейс користувача;
- реалізовано всі необхідні функції для проведення відеоконференцій, включаючи трансляцію аудіо та відео, демонстрацію екрану, текстовий чат та обмін файлами;
- система забезпечує високу якість відео (HD 720p) та аудіо зв'язку;
- додаток коректно працює в усіх сучасних браузерах та не потребує встановлення додаткового програмного забезпечення;
- реалізовано автоматичне відновлення з'єднання при втраті зв'язку.

Для розробки проєкту було обрано:

- React.js для створення користувацького інтерфейсу;
- TypeScript для забезпечення типобезпеки та кращої підтримки коду;
- WebRTC API для реалізації peer-to-peer комунікації;
- Socket.IO для організації сигнального сервера;
- STUN/TURN сервери для забезпечення з'єднання через NAT та брандмауери.

Система використовує сучасні веб-технології та архітектурні рішення:

- компонентний підхід до розробки інтерфейсу;
- реактивне програмування з використанням React Hooks;
- асинхронна обробка подій;
- оптимізація передачі медіа-потоків;

- адаптивний дизайн для різних пристроїв.

В результаті розроблена система відеоконференцій дозволяє:

- проводити групові відеоконференції з високою якістю зв'язку;
- здійснювати демонстрацію екрану;
- обмінюватися текстовими повідомленнями;
- транслювати відеофайли;
- керувати аудіо та відео пристроями.

Система успішно пройшла тестування та готова до впровадження в реальних умовах експлуатації. Вона може бути використана для проведення онлайн-зустрічей, дистанційного навчання, веб-конференцій та інших сценаріїв, що потребують якісного відеозв'язку через інтернет.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Saint-Andre, P. WebRTC. Real-time Communication in the Browser [Text] / P. Saint-Andre. – Sebastopol : O'Reilly Media, Inc., 2018. – 132 p.
2. Zoom vs. Google Meet vs. Discord: Which Video Chat Service Is Best? [Electronic resource]. – Access mode: <https://www.howtogeek.com/712398/zoom-vs-google-meet-vs-discord/>
3. Zoom's newest AI innovations help make analytics more accessible and customer experience more impactful [Electronic resource]. – Access mode: <https://news.zoom.com/zooms-newest-ai-innovations-help-make-analytics-more-accessible-and-customer-experience-more-impactful/>.
4. Google Meet Interface - Flipped Classroom Tutorials [Electronic resource]. – Access mode: <https://sethideclercq.com/2021/05/google-meets-new-interface>.
5. What Is Discord and How Do You Use It? / Lifewire. [Electronic resource]. – Access mode: <https://www.lifewire.com/what-is-discord-4171221>
6. 10 mejores aplicaciones de videollamadas - Dr.Fone [Recurso electrónico]. – Modo de acceso: <https://drfone.wondershare.es/topsoftware/10-best-video-calling-apps.html>.
7. The Best Video Conferencing Software for 2024 / PCMag [Electronic resource]. – Access mode: <https://www.pcmag.com/picks/best-video-conferencing-software>
8. Next.js Documentation [Electronic resource]. – Access mode: <https://nextjs.org/learn/pages-router>.
9. Remix Project Documentation [Electronic resource]. – Access mode: <https://remix-project.org/>.
10. Django Documentation [Electronic resource]. – Access mode: <https://docs.djangoproject.com/en/stable/>.
11. FastAPI Documentation [Electronic resource]. – Access mode: <https://fastapi.tiangolo.com/>.

12. Ruby on Rails Guides [Electronic resource]. – Access mode: <https://guides.rubyonrails.org/>.
13. shadcn/ui Documentation [Electronic resource]. – Access mode: <https://ui.shadcn.com/docs>.
14. Tailwind CSS Documentation [Electronic resource]. – Access mode: <https://tailwindcss.com/docs>.
15. Node.js Documentation [Electronic resource]. – Access mode: <https://nodejs.org/en/learn/getting-started/introduction-to-nodejs>.
16. WebSocket Protocol [Electronic resource]. – Access mode: <https://websocket.org/guides/websocket-protocol/>.
17. LiveSwitch. Ultimate Guide to WebRTC [Electronic resource]. – Access mode: <https://www.liveswitch.io/resources/ultimate-guide-to-webrtc>.

ДОДАТОК А
Текст програми

A.1 Підстава для розробки

Підставою для розробки послужило завдання на дипломний проєкт на тему «Програмна реалізація системи обміну мультимедійними даними з використанням технології WebRTC», затверджене наказом Національного університету «Запорізька політехніка» від 28 квітня 2025 року № 209.

A.2 Призначення розробки

Метою роботи є створення веб-додатку для організації відеоконференцій у реальному часі з використанням peer-to-peer технології WebRTC. Система забезпечує:

- проведення групових відеодзвінків із передачею аудіо/відео;
- обмін текстовими повідомленнями через чат;
- демонстрацію екрану та відтворення відеофайлів;
- автоматичне встановлення прямих з'єднань між користувачами.

A.3 Основні вимоги до розроблювальної програми

A.3.1 Вимоги до надійності

Програма повинна коректно обробляти помилки в таких сценаріях:

- втрата мережевого з'єднання (автоматичне перепідключення);
- відмова користувача у доступі до медіапристроїв (камера/мікрофон);
- неможливість встановлення P2P-з'єднання (перехід на TURN-сервер);
- некоректні вхідні дані (ID кімнати, формати файлів).

А.3.2 Умови експлуатації

Під час роботи програми не слід перезавантажувати комп'ютер або вимикати його живлення, оскільки це може призвести до помилок у роботі. Завершення роботи з програмою має відбуватись шляхом послідовного вимкнення клієнтської та серверної частин.

Користування програмним продуктом повинно здійснюватися відповідно до експлуатаційної документації, яка розроблена згідно зі стандартами та містить усю необхідну інформацію для ознайомлення з програмою та її правильного використання.

А.3.3 Вимоги до складу й параметрів технічних засобів

Необхідні вимоги до складу й параметрів технічних засобів, які повинні бути застосовані для роботи застосунку, наступні:

- а) для сервера (Node.js):
 - 1) процесор: 2+ ядер (x64);
 - 2) ОЗУ: 2+ ГБ;
 - 3) мережа: 100+ Мбіт/с, публічна IPv4/IPv6 адреса;
 - 4) ОС: Linux/Windows;
- б) для клієнта:
 - 1) браузер: Chrome 90+, Edge 90+, Opera 75+;
 - 2) вебкамера/мікрофон;
 - 3) інтернет: 5+ Мбіт/с на учасника.

А.3.4 Вимоги до маркування й пакування

Дистрибутив програми розповсюджується у вигляді Docker-образу або архіву з кодом.

На пакуванні вказується назва: «Система обміну мультимедійними даними v1.0».

A.3.5 Вимоги до транспортування й зберігання

Вимоги до транспортування й зберігання відповідають вимогам апаратного носія, на якому записана програма.

A.3.6 Вимоги до програмної документації

Система супроводжується документацією:

- керівництво програміста (інструкція з розгортання сервера);
- керівництво користувача (інструкція з використання веб-додатку).

ДОДАТОК Б
Текст програми

B.1 socket\index.ts

```

import { createServer } from 'http'
import { Server } from 'socket.io'
import path from 'path'
import { readFile } from 'fs/promises'
import { ACTIONS } from "../actions";

const PORT = process.env.PORT || 3001

const server = createServer(async (req, res) => {
  if (req.method === 'GET') {
    try {
      const filePath = path.join(process.cwd(), 'build', 'index.html')
      const fileContent = await readFile(filePath)
      res.writeHead(200, { 'Content-Type': 'text/html' })
      res.end(fileContent)
    } catch (e) {
      res.writeHead(404)
      res.end('Not found')
    }
  } else {
    res.writeHead(405)
    res.end('Method Not Allowed')
  }
})

const io = new Server(server, {

function getClientRooms() {
  const { rooms } = io.sockets.adapter
  // Remove socket IDs from the rooms list
  return Array.from(rooms.keys()).filter(roomID =>
!io.sockets.adapter.sids.has(roomID))
}

function shareRoomsInfo() {
  io.emit(ACTIONS.SHARE_ROOMS, { rooms: getClientRooms() })
}

io.on('connection', (socket) => {
  console.log(`[Server] New connection: ${socket.id}`)
  shareRoomsInfo()
})

```

```

socket.on(ACTIONS.JOIN, async ({ room: roomID, username, password }) => {
  const { rooms: joinedRooms } = socket

  if (Array.from(joinedRooms).includes(roomID)) {
    console.warn(`[Server] User ${username} (${socket.id}) already joined
room ${roomID}`)
    return
  }

  // Get all clients in the room
  const clients = Array.from(io.sockets.adapter.rooms.get(roomID) || [])
  console.log(`[Server] Existing clients in room ${roomID}:`, clients)

  // If there are clients in the room, verify password
  if (clients.length > 0) {
    // Get first client in room to check password
    const firstClient = io.sockets.sockets.get(clients[0])
    if (firstClient?.data.roomPassword && firstClient.data.roomPassword !==
password) {
      // Emit error event if password is incorrect
      console.log(`[Server] User ${username} (${socket.id}) provided incorrect
password for room ${roomID}`)
      socket.emit(ACTIONS.JOIN_ERROR, { message: "Incorrect password"
})
      return
    }
  }

  // Store password if provided (for first user)
  if (password) {
    socket.data.roomPassword = password
  }

  // If there are existing clients, copy password from them
  if (clients.length > 0) {
    const firstClient = io.sockets.sockets.get(clients[0])
    if (firstClient?.data.roomPassword) {
      socket.data.roomPassword = firstClient.data.roomPassword
    }
  }

  console.log(`[Server] User ${username} (${socket.id}) is joining room
${roomID}`)

```

```

// Join the room before notifying peers
socket.join(roomID)
socket.data.username = username // Store username in socket data

// After joining, notify all peers
clients.forEach(clientID => {
  const clientSocket = io.sockets.sockets.get(clientID);
  console.log(`[Server] Notifying existing user
${clientSocket?.data.username} (${clientID}) about new peer ${socket.id}`)

  // Notify existing client about new peer
  io.to(clientID).emit(ACTIONS.ADD_PEER, {
    peerID: socket.id,
    createOffer: false,
    username: username
  })
  console.log(`[Server] Sent ADD_PEER to ${clientID} with createOffer:
false`)

  // Notify new peer about existing client
  socket.emit(ACTIONS.ADD_PEER, {
    peerID: clientID,
    createOffer: true,
    username: clientSocket?.data.username
  })
  console.log(`[Server] Sent ADD_PEER to ${socket.id} with createOffer:
true`)
})

  console.log(`[Server] User ${username} (${socket.id}) joined room
${roomID}`)
  shareRoomsInfo()
})

socket.on(ACTIONS.RELAY_SDP, ({ peerID, sessionDescription }) => {
  console.log(`[Server] Relaying ${sessionDescription.type} from ${socket.id}
to ${peerID}`)
  io.to(peerID).emit(ACTIONS.SESSION_DESCRIPTION, {
    peerID: socket.id,
    sessionDescription,
  })
})

socket.on(ACTIONS.RELAY_ICE, ({ peerID, iceCandidate }) => {
  console.log(`[Server] Relaying ICE candidate from ${socket.id} to ${peerID}`)

```

```

    io.to(peerID).emit(ACTIONS.ICE_CANDIDATE, {
      peerID: socket.id,
      iceCandidate,
    })
  })

socket.on(ACTIONS.LEAVE, () => {
  const { rooms } = socket

  Array.from(rooms).forEach(roomID => {
    const clients = Array.from(io.sockets.adapter.rooms.get(roomID) || [])
    console.log(`[Server] User ${socket.data.username} (${socket.id}) leaving
room ${roomID}`)

    clients.forEach(clientID => {
      if (clientID !== socket.id) {
        console.log(`[Server] Notifying ${clientID} about peer ${socket.id}
leaving`)
        io.to(clientID).emit(ACTIONS.REMOVE_PEER, {
          peerID: socket.id,
        })

        socket.emit(ACTIONS.REMOVE_PEER, {
          peerID: clientID,
        })
      }
    })

    socket.leave(roomID)
    console.log(`[Server] User ${socket.data.username} (${socket.id}) left room
${roomID}`)
  })

  shareRoomsInfo()
})

socket.on('disconnecting', () => {
  console.log(`[Server] User ${socket.data.username} (${socket.id})
disconnecting`)

  // Get all rooms the socket is in before disconnection
  const { rooms } = socket

  // Notify all clients in each room about the disconnection
  Array.from(rooms).forEach(roomID => {

```

```

const clients = Array.from(io.sockets.adapter.rooms.get(roomID) || [])

clients.forEach(clientID => {
  if (clientID !== socket.id) {
    console.log(`[Server] Notifying ${clientID} about peer ${socket.id}
disconnecting`)
    io.to(clientID).emit(ACTIONS.REMOVE_PEER, {
      peerID: socket.id,
    })
  }
})
})
})

// Add disconnect handler
socket.on('disconnect', () => {
  console.log(`[Server] User ${socket.data.username} (${socket.id})
disconnected`)
  shareRoomsInfo()
})
})

server.listen(PORT, () => {
  console.log(`Server started on port ${PORT}`)
})

```

B.2 socket\actions.ts

```

export enum ACTIONS {
  JOIN = 'join',
  LEAVE = 'leave',
  SHARE_ROOMS = 'share-rooms',
  ADD_PEER = 'add-peer',
  REMOVE_PEER = 'remove-peer',
  RELAY_SDP = 'relay-sdp',
  RELAY_ICE = 'relay-ice',
  ICE_CANDIDATE = 'ice-candidate',
  SESSION_DESCRIPTION = 'session-description',
  JOIN_ERROR = 'join-error'
}

```

B.3 src\app\room\[roomId]\page.tsx

```

'use client';

```

```

import { useParams } from 'next/navigation';
import useWebRTC, { LOCAL_VIDEO } from '@/hooks/useWebRTC';
import { useEffect, useState, useRef } from 'react';
import Chat from '@/components/Chat';
import {
  Dialog,
  DialogContent,
  DialogHeader,
  DialogTitle,
} from "@/components/ui/dialog";
import { Button } from "@/components/ui/button";
import { Video, VideoOff, Mic, MicOff, FileVideo, MonitorUp } from "lucide-react";
import socket from '@/lib/socket';
import { ACTIONS } from '../../socket/actions';

export default function Room() {
  const { roomId } = useParams();
  const [username, setUsername] = useState<string>("");
  const [password, setPassword] = useState<string>("");
  const [isNameDialogOpen, setIsNameDialogOpen] = useState(true);
  const [joinError, setJoinError] = useState<string>("");
  const [hasExistingUsers, setHasExistingUsers] = useState(false);
  const [isJoining, setIsJoining] = useState(false);
  const [retryKey, setRetryKey] = useState(0);
  const {
    clients,
    provideMediaRef,
    activeVideoId,
    setActiveVideoId,
    provideActiveMediaRef,
    messages,
    broadcastMessage,
    peerUsernames,
    toggleCamera,
    isCameraEnabled,
    toggleMicrophone,
    isMicEnabled,
    isVideoFile,
    switchToVideoFile,
    switchToCamera,
    selectedVideoFile,
    toggleScreenShare,
    isScreenSharing
  }

```

```
    } = useWebRTC(roomId as string, isNameDialogOpen ? null : username,
isNameDialogOpen ? undefined : password || undefined, retryKey);
```

```
// Check if room has users when component mounts
```

```
useEffect(() => {
  socket.emit(ACTIONS.SHARE_ROOMS);

  const handleShareRooms = ({ rooms }: { rooms: string[] }) => {
    setHasExistingUsers(rooms.includes(roomId as string));
  };

  socket.on(ACTIONS.SHARE_ROOMS, handleShareRooms);

// Handle join errors
  const handleJoinError = ({ message }: { message: string }) => {
    setJoinError(message);
    setIsJoining(false);
    setIsNameDialogOpen(true); // Keep dialog open on error
  };

  socket.on(ACTIONS.JOIN_ERROR, handleJoinError);

  return () => {
    socket.off(ACTIONS.SHARE_ROOMS, handleShareRooms);
    socket.off(ACTIONS.JOIN_ERROR, handleJoinError);
  };
}, [roomId]);
```

```
useEffect(() => {
  if (!isNameDialogOpen) {
    console.log(`[Room] Current participants in room ${roomId}:`, clients);
  }
}, [clients, isNameDialogOpen]);
```

```
if (isNameDialogOpen) {
  return (
    <Dialog open={isNameDialogOpen}
onOpenChange={setIsNameDialogOpen}>
    <DialogContent>
      <DialogHeader>
        <DialogTitle>Enter your details</DialogTitle>
      </DialogHeader>
      <form
onSubmit={async (e) => {
  e.preventDefault();
```

```

    if (username.trim() && (!hasExistingUsers || password.trim())) {
      setJoinError("");
      setIsJoining(true);
      // Increment retry key to reset WebRTC state
      setRetryKey(prev => prev + 1);
      // Wait a bit to see if we get a join error
      await new Promise(resolve => setTimeout(resolve, 500));
      if (!joinError) {
        setIsNameDialogOpen(false);
      }
      setIsJoining(false);
    }
  }}
  className="space-y-4"
>
<div>
  <label className="block text-sm font-medium text-gray-700 mb-
1">
    Your name
  </label>
  <input
    type="text"
    value={username}
    onChange={(e) => setUsername(e.target.value)}
    placeholder="Your name"
    className="w-full px-4 py-2 border rounded-lg focus:outline-
none focus:ring-2 focus:ring-blue-500"
    autoFocus
  />
</div>

  {/* Show password field if room has no users or if it has users (for
entering password) */}
  <div>
    <label className="block text-sm font-medium text-gray-700 mb-
1">
      {hasExistingUsers ? 'Room Password' : 'Set Room Password
(Optional)'}
    </label>
    <input
      type="password"
      value={password}
      onChange={(e) => {
        setPassword(e.target.value);
        setJoinError(""); // Clear error when password changes

```

```

    }}
    placeholder={hasExistingUsers ? 'Enter room password' : 'Set a
password (optional)'}
    className="w-full px-4 py-2 border rounded-lg focus:outline-
none focus:ring-2 focus:ring-blue-500"
    required={hasExistingUsers}
  />
</div>

  {joinError && (
    <div className="text-red-500 text-sm">
      {joinError}
    </div>
  )}

  <button
    type="submit"
    disabled={!username.trim() || (hasExistingUsers &&
!password.trim()) || isJoining}
    className="w-full px-4 py-2 bg-blue-500 text-white rounded-lg
hover:bg-blue-600 disabled:opacity-50 disabled:cursor-not-allowed"
  >
    {isJoining ? 'Joining...' : 'Join Room'}
  </button>
</form>
</DialogContent>
</Dialog>
);
}

return (
  <div style={{ display: 'flex', height: '100vh' }}>
    <div style={{ display: 'flex', height: '100vh' }}>
      <div style={{ flex: 3, display: 'flex', flexDirection: 'column' }}>
        <div style={{ flex: 1, padding: 20 }} className="relative">
          <video
            style={{ width: '100%', height: '100%', backgroundColor: 'black' }}
            autoPlay
            playsInline
            muted={true}
            ref={provideActiveMediaRef}
          />
          <div className="absolute bottom-4 left-4 bg-black bg-opacity-50 px-
3 py-2 rounded text-white">

```

```

    {peerUsernames[activeVideoId] || activeVideoId}
    {activeVideoId === LOCAL_VIDEO && ' (You)'}
  </div>
  {activeVideoId === LOCAL_VIDEO && (
    <div className="absolute bottom-4 right-4 flex gap-2">
      <Button
        onClick={toggleMicrophone}
        className="bg-slate-800 hover:bg-slate-700"
        size="icon"
      >
        {isMicEnabled ? <Mic className="h-5 w-5 text-white" /> :
<MicOff className="h-5 w-5 text-white" />}
      </Button>
      <Button
        onClick={toggleCamera}
        className="bg-slate-800 hover:bg-slate-700"
        size="icon"
      >
        {isCameraEnabled ? <Video className="h-5 w-5 text-white"
/> : <VideoOff className="h-5 w-5 text-white" />}
      </Button>
      <Button
        onClick={toggleScreenShare}
        className="bg-slate-800 hover:bg-slate-700"
        size="icon"
        title={isScreenSharing ? "Stop sharing screen" : "Share screen"}
      >
        <MonitorUp className={`h-5 w-5 ${isScreenSharing ? "text-
green-400" : "text-white"}`} />
      </Button>
      <div className="relative">
        <input
          type="file"
          accept="video/*"
          onChange={(e) => {
            const file = e.target.files?.[0];
            if (file) {
              switchToVideoFile(file);
            }
          }}
          className="hidden"
          id="video-file-input"
        />
        <Button
          onClick={() => {

```

```

        if (isVideoFile) {
            switchToCamera();
        } else {
            document.getElementById('video-file-input')?.click();
        }
    }}
    className="bg-slate-800 hover:bg-slate-700"
    size="icon"
    title={isVideoFile ? "Switch to camera" : "Switch to video
file"}
    >
    <FileVideo className="h-5 w-5 text-white" />
  </Button>
</div>
</div>
)}
</div>
</div>

{/* Sidebar with video previews and chat */}
<div style={{
  flex: 1,
  display: 'flex',
  flexDirection: 'column',
  borderLeft: '1px solid #ccc',
  height: '100vh',
}}>
  {/* Video previews */}
  <div style={{
    flex: 1,
    display: 'flex',
    flexDirection: 'column',
    gap: 10,
    overflowY: 'auto',
    padding: 20,
  }}>
    {clients.map(clientID => (
      <div
        key={clientID}
        className="relative"
        onClick={() => setActiveVideoId(clientID)}
      >
        <video
          style={{
            width: '100%',

```

```

        height: '120px',
        cursor: 'pointer',
        border: clientID === activeVideoId ? '3px solid blue' : '1px
solid gray',
        borderRadius: 4,
        objectFit: 'cover',
    }}
    muted={selectedVideoFile? false :clientID ===
LOCAL_VIDEO}
    autoPlay
    playsInline
    ref={node => provideMediaRef(clientID, node)}
  />
  <div className="absolute bottom-2 left-2 bg-black bg-opacity-50
px-2 py-1 rounded text-white text-sm">
    {peerUsernames[clientID] || clientID}
    {clientID === LOCAL_VIDEO && ' (You)'}
  </div>
</div>
  )))
</div>

  { /* Chat area */}
  <div style={{ flex: 2, padding: '20px' }}>
    <Chat
      messages={messages}
      onSendMessage={broadcastMessage}
      currentUsername={username}
    />
  </div>
</div>
</div>
);
}

```

Б.4 src\app\layout.tsx

```

import type { Metadata } from "next";
import { Geist, Geist_Mono } from "next/font/google";
import "./globals.css";

const geistSans = Geist({
  variable: "--font-geist-sans",
  subsets: ["latin"],

```

```

});

const geistMono = Geist_Mono({
  variable: "--font-geist-mono",
  subsets: ["latin"],
});

export const metadata: Metadata = {
  title: "Create Next App",
  description: "Generated by create next app",
};

export default function RootLayout({
  children,
}: Readonly<{
  children: React.ReactNode;
}>) {
  return (
    <html lang="en">
      <body
        className={` ${geistSans.variable} ${geistMono.variable} antialiased` }
      >
        {children}
      </body>
    </html>
  );
}

```

Б.5 src\app\page.tsx

```

'use client'

import { useRouter } from 'next/navigation'
import { Button } from '@components/ui/button'
import { Input } from '@components/ui/input'
import { useState } from 'react'

export default function HomePage() {
  const router = useRouter()
  const [roomId, setRoomId] = useState("")

  return (
    <div className="max-w-sm mx-auto flex flex-col gap-4">

```

```

        <Input value={roomId} onChange={e => setRoomId(e.target.value)}
placeholder="Enter Room ID" />
        <Button onClick={() => router.push(`/room/${roomId} ||
crypto.randomUUID()`) }>Join/Create Room</Button>
    </div>
)
}

```

B.6 src\components\Chat.tsx

```

import { useState, useRef, useEffect } from 'react';
import { Message } from '@/hooks/useWebRTC';

interface ChatProps {
  messages: Message[];
  onSendMessage: (content: string) => void;
  currentUsername: string;
}

export default function Chat({ messages, onSendMessage, currentUsername }:
ChatProps) {
  const [newMessage, setNewMessage] = useState("");
  const messagesEndRef = useRef<HTMLDivElement>(null);

  const scrollToBottom = () => {
    messagesEndRef.current?.scrollIntoView({ behavior: 'smooth' });
  };

  useEffect(() => {
    scrollToBottom();
  }, [messages]);

  const handleSubmit = (e: React.FormEvent) => {
    e.preventDefault();
    if (newMessage.trim()) {
      onSendMessage(newMessage.trim());
      setNewMessage("");
    }
  };

  return (
    <div className="flex flex-col h-full bg-white rounded-lg shadow">
      <div className="p-4 border-b">
        <h2 className="text-lg font-semibold">Chat</h2>

```

```
</div>
```

```
{/* Messages container */}
```

```
<div className="flex-1 overflow-y-auto p-4">
```

```
  <div className="space-y-4 max-w-[95%] mx-auto">
```

```
    {messages.map((message) => {
```

```
      const isOwnMessage = message.sender === currentUsername;
```

```
      return (
```

```
        <div
```

```
          key={message.id}
```

```
          className={`flex flex-col w-full ${
```

```
            isOwnMessage ? 'items-end' : 'items-start'
```

```
          }`}
```

```
        >
```

```
        <span
```

```
          className={`text-xs text-gray-500 mb-1 px-2 ${
```

```
            isOwnMessage ? 'text-right' : 'text-left'
```

```
          }`}
```

```
        >
```

```
          {message.sender}
```

```
        </span>
```

```
        <div
```

```
          className={`max-w-[80%] rounded-lg p-3 ${
```

```
            isOwnMessage
```

```
              ? 'bg-blue-500 text-white ml-auto'
```

```
              : 'bg-gray-100 text-gray-900 mr-auto'
```

```
          }`}
```

```
        >
```

```
        <p className="break-words">{message.content}</p>
```

```
        <span className={`text-xs mt-1 block ${
```

```
          isOwnMessage ? 'opacity-75' : 'text-gray-500'
```

```
        }`}>
```

```
          {new Date(message.timestamp).toLocaleTimeString()}
```

```
        </span>
```

```
      </div>
```

```
    </div>
```

```
  );
```

```
  )})
```

```
  <div ref={messagesEndRef} />
```

```
</div>
```

```
</div>
```

```
{/* Message input */}
```

```
<form onSubmit={handleSubmit} className="p-4 border-t">
```

```
  <div className="flex gap-2">
```

```

        <input
          type="text"
          value={newMessage}
          onChange={(e) => setNewMessage(e.target.value)}
          placeholder="Type a message..."
          className="flex-1 px-4 py-2 border rounded-lg focus:outline-none
focus:ring-2 focus:ring-blue-500"
        />
        <button
          type="submit"
          disabled={!newMessage.trim()}
          className="px-4 py-2 bg-blue-500 text-white rounded-lg hover:bg-
blue-600 disabled:opacity-50 disabled:cursor-not-allowed"
        >
          Send
        </button>
      </div>
    </form>
  </div>
);
}

```

B.7 src\hooks\useStateWithCallback.ts

```

import { useState, useCallback, useRef, useEffect, SetStateAction } from 'react';

type Callback<T> = (state: T) => void;

function useStateWithCallback<T>(initialState: T | (() => T)): [T, (newState:
SetStateAction<T>, cb?: Callback<T>) => void] {
  const [state, setState] = useState<T>(initialState);
  const cbRef = useRef<Callback<T> | null>(null);

  const updateState = useCallback((newState: SetStateAction<T>, cb?:
Callback<T>) => {
    if (cb) {
      cbRef.current = cb;
    }

    setState(prev => typeof newState === 'function' ? (newState as (prevState: T)
=> T)(prev) : newState);
  }, []);

  useEffect(() => {

```

```

    if (cbRef.current) {
      cbRef.current(state);
      cbRef.current = null;
    }
  }, [state]);

  return [state, updateState];
}

export default useStateWithCallback;

```

B.8 src\hooks\useWebRTC.ts

```

import { useEffect, useRef, useCallback, useState } from 'react';
import useStateWithCallback from './useStateWithCallback';
import socket from '@lib/socket';
import { ACTIONS } from '../socket/actions';

export const LOCAL_VIDEO = 'LOCAL_VIDEO';

type PeerConnections = { [peerID: string]: RTCPeerConnection };
type PeerMediaElements = { [peerID: string]: HTMLVideoElement | null };
type DataChannels = { [peerID: string]: RTCDataChannel };
type PeerUsernames = { [peerID: string]: string };

export interface Message {
  id: string;
  sender: string;
  content: string;
  timestamp: number;
}

interface JoinConfig {
  peerID: string;
  createOffer: boolean;
  username?: string;
}

interface SessionDescription {
  peerID: string;
  sessionDescription: RTCSessionDescriptionInit;
}
interface IceCandidateData {
  peerID: string;
  iceCandidate: RTCIceCandidateInit;
}
interface RemovePeerData {
  peerID: string;
}

```

```

export default function useWebRTC(roomID: string, username: string | null,
password?: string, retryKey: number = 0) {
  const [clients, updateClients] = useStateWithCallback<string[]>([]);
  const [activeVideoId, setActiveVideoId] = useState<string>(LOCAL_VIDEO);
  const [messages, setMessages] = useState<Message[]>([]);
  const [peerUsernames, setPeerUsernames] = useState<PeerUsernames>({
[LOCAL_VIDEO]: username || " " });
  const [isCameraEnabled, setIsCameraEnabled] = useState(true);
  const [isMicEnabled, setIsMicEnabled] = useState(true);
  const [isVideoFile, setIsVideoFile] = useState(false);
  const [selectedVideoFile, setSelectedVideoFile] = useState<File | null>(null);
  const [isScreenSharing, setIsScreenSharing] = useState(false);
  const videoElementRef = useRef<HTMLVideoElement | null>(null);

  const peerConnections = useRef<PeerConnections>({});
  const dataChannels = useRef<DataChannels>({});
  const localMediaStream = useRef<MediaStream | null>(null);
  const videoFileStream = useRef<MediaStream | null>(null);
  const peerMediaElements = useRef<PeerMediaElements>({ [LOCAL_VIDEO]:
null });
  const activeVideoRef = useRef<HTMLVideoElement | null>(null);

  // Add cleanup function
  const cleanup = useCallback(() => {
    console.log('[WebRTC] Running cleanup...');

    // Stop all tracks in local stream
    if (localMediaStream.current) {
      localMediaStream.current.getTracks().forEach(track => {
        track.stop();
        track.enabled = false;
      });
      localMediaStream.current = null;
    }

    // Stop video file stream if exists
    if (videoFileStream.current) {
      videoFileStream.current.getTracks().forEach(track => {
        track.stop();
        track.enabled = false;
      });
      videoFileStream.current = null;
    }

    // Close all peer connections

```

```

Object.entries(peerConnections.current).forEach(([peerID, connection]) => {
  connection.close();
  delete peerConnections.current[peerID];
});

// Close all data channels
Object.values(dataChannels.current).forEach(channel => {
  channel.close();
});
dataChannels.current = {};

// Clear all media elements
peerMediaElements.current = { [LOCAL_VIDEO]: null };

// Clear clients
updateClients(() => []);

// Reset states
setActiveVideoId(LOCAL_VIDEO);
setMessages([]);
setPeerUsernames({ [LOCAL_VIDEO]: username || " " });
setIsCameraEnabled(true);
setIsMicEnabled(true);
setIsVideoFile(false);
setSelectedVideoFile(null);
setIsScreenSharing(false);
}, [username, updateClients]);

// Run cleanup when retryKey changes
useEffect(() => {
  cleanup();
}, [retryKey, cleanup]);

// Update local username when it changes
useEffect(() => {
  if (username) {
    setPeerUsernames(prev => ({ ...prev, [LOCAL_VIDEO]: username }));
  }
}, [username]);

// Додаємо клієнта, якщо його нема
const addNewClient = useCallback((newClient: string, cb?: () => void) => {
  updateClients(list => {
    if (!list.includes(newClient)) {
      return [...list, newClient];
    }
  });
});

```

```

    }
    return list;
  }, cb);
}, [updateClients]);

```

// Встановлення srcObject для активного відео (застосовується тільки тут)

```

useEffect(() => {
  if (!activeVideoId) return;

  const videoEl = activeVideoRef.current;
  const mediaEl = peerMediaElements.current[activeVideoId];

```

// Якщо посилання на відео-елемент в компоненті і в колекції є, ставимо srcObject

```

  if (videoEl && mediaEl) {
    // srcObject можна ставити лише для одного відео, тому беремо стрім з mediaEl
    videoEl.srcObject = mediaEl.srcObject || null;
    videoEl.play().catch(() => {});
  }
}, [activeVideoId, clients]);

```

// Add this new function for broadcasting messages to all peers

```

const broadcastMessage = useCallback((content: string) => {
  if (!username) return; // Don't send if no username

```

```

  const message: Message = {
    id: `${Date.now()}-${Math.random()}`,
    sender: username,
    content,
    timestamp: Date.now()
  };

```

// Add message to local state

```

setMessages(prev => [...prev, message]);

```

// Broadcast to all peers

```

Object.values(dataChannels.current).forEach(channel => {
  if (channel.readyState === 'open') {
    channel.send(JSON.stringify(message));
  }
});
}, [username]);

```

```

useEffect(() => {
  async function handleNewPeer({ peerID, createOffer, username:
peerUsername }: JoinConfig & { username?: string }) {
    if (peerID in peerConnections.current) {
      console.warn(`Already connected to peer ${peerID}`);
      return;
    }

    console.log(`[WebRTC] New peer connection with ${peerUsername ||
peerID}${createOffer ? ' (initiator)' : ''}`);
    console.log(`[WebRTC] Current screen sharing state:`, isScreenSharing);
    console.log(`[WebRTC] Local media stream exists:`,
!!localMediaStream.current);
    if (localMediaStream.current) {
      console.log(`[WebRTC] Local stream tracks:`,
localMediaStream.current.getTracks().map(t => ({ kind: t.kind, enabled: t.enabled,
id: t.id })));
    }

    // Store the peer's username
    if (peerUsername) {
      setPeerUsernames(prev => ({ ...prev, [peerID]: peerUsername }));
    }

    const iceServers = [{ urls: 'stun:stun.l.google.com:19302' }];

    const peerConnection = new RTCPeerConnection({ iceServers });
    peerConnections.current[peerID] = peerConnection;

    // Add connection state logging
    peerConnection.onconnectionstatechange = () => {
      console.log(`[WebRTC] Connection state with peer ${peerID}:`,
peerConnection.connectionState);
    };

    peerConnection.oniceconnectionstatechange = () => {
      console.log(`[WebRTC] ICE connection state with peer ${peerID}:`,
peerConnection.iceConnectionState);
    };

    peerConnection.onsignalingstatechange = () => {
      console.log(`[WebRTC] Signaling state with peer ${peerID}:`,
peerConnection.signalingState);
    };
  }

```

```

// Setup data channel if we're the initiator
if (createOffer) {
  const dataChannel = peerConnection.createDataChannel('chat');
  dataChannels.current[peerID] = dataChannel;

  dataChannel.onmessage = ({ data }) => {
    const message: Message = JSON.parse(data);
    setMessages(prev => [...prev, message]);
  };

  dataChannel.onopen = () => {
    console.log(`[WebRTC] Data channel opened with ${peerID}`);
  };

  dataChannel.onclose = () => {
    console.log(`[WebRTC] Data channel closed with ${peerID}`);
  };
} else {
  // If we're not the initiator, listen for the data channel
  peerConnection.ondatachannel = (event) => {
    const dataChannel = event.channel;
    dataChannels.current[peerID] = dataChannel;

    dataChannel.onmessage = ({ data }) => {
      const message: Message = JSON.parse(data);
      setMessages(prev => [...prev, message]);
    };

    dataChannel.onopen = () => {
      console.log(`[WebRTC] Data channel opened with ${peerID}`);
    };

    dataChannel.onclose = () => {
      console.log(`[WebRTC] Data channel closed with ${peerID}`);
    };
  };
}

peerConnection.onicecandidate = event => {
  if (event.candidate) {
    console.log(`[WebRTC] Sending ICE candidate to peer ${peerID}:`,
event.candidate.type);
    socket.emit(ACTIONS.RELAY_ICE, {
      peerID,
      iceCandidate: event.candidate,

```

```

    });
  }
};

let tracksCount = 0;
peerConnection.ontrack = ({ streams: [remoteStream] }) => {
  console.log(`[WebRTC] Received track from peer ${peerID}:`, {
    streamId: remoteStream.id,
    tracks: remoteStream.getTracks().map(t => ({ kind: t.kind, enabled:
t.enabled, id: t.id })))
  });

  tracksCount++;
  console.log(`[WebRTC] Track count for peer ${peerID}:`, tracksCount);

  // Get expected track count based on stream type
  const expectedTracks = remoteStream.getTracks().length;
  console.log(`[WebRTC] Expected tracks for peer ${peerID}:`,
expectedTracks);

  if (tracksCount === expectedTracks) {
    tracksCount = 0;
    console.log(`[WebRTC] All tracks (${expectedTracks}) received from
peer ${peerID}, adding client`);

    addNewClient(peerID, () => {
      const el = peerMediaElements.current[peerID];
      if (el) {
        el.srcObject = remoteStream;
        console.log(`[WebRTC] Successfully set remote stream for peer
${peerID}`);
      } else {
        console.log(`[WebRTC] Video element for peer ${peerID} not
found, setting up retry interval`);
        let settled = false;
        const interval = setInterval(() => {
          if (peerMediaElements.current[peerID]) {
            peerMediaElements.current[peerID]!.srcObject =
remoteStream;

            settled = true;
            console.log(`[WebRTC] Successfully set remote stream for
peer ${peerID} after retry`);
          }
          if (settled) clearInterval(interval);
        }, 1000);
      }
    });
  }
};

```

```

        }
    });
}
};

// If we're screen sharing, use the screen sharing stream instead of camera
stream
if (isScreenSharing && localMediaStream.current) {
    console.log(`[WebRTC] Adding screen sharing stream to peer ${peerID}`);
    {
        streamId: localMediaStream.current.id,
        tracks: localMediaStream.current.getTracks().map(t => ({ kind: t.kind,
enabled: t.enabled, id: t.id })))
    });
    localMediaStream.current.getTracks().forEach(track => {
        const sender = peerConnection.addTrack(track,
localMediaStream.current!);
        console.log(`[WebRTC] Added ${track.kind} track to peer ${peerID}`);
        { trackId: track.id, enabled: track.enabled });
    });
} else if (localMediaStream.current) {
    console.log(`[WebRTC] Adding camera stream to peer ${peerID}`);
    {
        streamId: localMediaStream.current.id,
        tracks: localMediaStream.current.getTracks().map(t => ({ kind: t.kind,
enabled: t.enabled, id: t.id })))
    });
    localMediaStream.current.getTracks().forEach(track => {
        const sender = peerConnection.addTrack(track,
localMediaStream.current!);
        console.log(`[WebRTC] Added ${track.kind} track to peer ${peerID}`);
        { trackId: track.id, enabled: track.enabled });
    });
} else {
    console.warn(`[WebRTC] No local stream available to send to peer
${peerID}`);
}

if (createOffer) {
    console.log(`[WebRTC] Creating offer for peer ${peerID}`);
    const offer = await peerConnection.createOffer();
    await peerConnection.setLocalDescription(offer);
    console.log(`[WebRTC] Set local description for peer ${peerID}`);
    offer.type);

    socket.emit(ACTIONS.RELAY_SDP, {

```

```

        peerID,
        sessionDescription: offer,
    });
    console.log(`[WebRTC] Sent offer to peer ${peerID}`);
  }
}

socket.on(ACTIONS.ADD_PEER, handleNewPeer);

return () => {
  socket.off(ACTIONS.ADD_PEER, handleNewPeer);
};
}, [addNewClient, isScreenSharing]);

useEffect(() => {
  async function setRemoteMedia({ peerID, sessionDescription:
remoteDescription }: SessionDescription) {
    console.log(`[WebRTC] Received ${remoteDescription.type} from peer
${peerID}`);
    const peerConnection = peerConnections.current[peerID];
    if (!peerConnection) {
      console.warn(`[WebRTC] No peer connection found for ${peerID}`);
      return;
    }

    await peerConnection.setRemoteDescription(new
RTCSessionDescription(remoteDescription));
    console.log(`[WebRTC] Set remote description for peer ${peerID}:`,
remoteDescription.type);

    if (remoteDescription.type === 'offer') {
      console.log(`[WebRTC] Creating answer for peer ${peerID}`);
      const answer = await peerConnection.createAnswer();
      await peerConnection.setLocalDescription(answer);
      console.log(`[WebRTC] Set local description (answer) for peer
${peerID}`);

      socket.emit(ACTIONS.RELAY_SDP, {
        peerID,
        sessionDescription: answer,
      });
      console.log(`[WebRTC] Sent answer to peer ${peerID}`);
    }
  }
}

```

```

socket.on(ACTIONS.SESSION_DESCRIPTION, setRemoteMedia);

return () => {
  socket.off(ACTIONS.SESSION_DESCRIPTION, setRemoteMedia);
};
}, []);

useEffect(() => {
  function handleIceCandidate({ peerID, iceCandidate }: IceCandidateData) {
    peerConnections.current[peerID]?.addIceCandidate(new
RTCIceCandidate(iceCandidate));
  }

  socket.on(ACTIONS.ICE_CANDIDATE, handleIceCandidate);

  return () => {
    socket.off(ACTIONS.ICE_CANDIDATE, handleIceCandidate);
  };
}, []);

// Modify handleRemovePeer to clean up data channels
useEffect(() => {
  function handleRemovePeer({ peerID }: RemovePeerData) {
    console.log(`[WebRTC] Peer ${peerID} disconnected`);

    if (peerConnections.current[peerID]) {
      console.log(`[WebRTC] Closing connection for peer ${peerID}`);
      peerConnections.current[peerID].close();
    }

    if (dataChannels.current[peerID]) {
      dataChannels.current[peerID].close();
      delete dataChannels.current[peerID];
    }

    delete peerConnections.current[peerID];
    delete peerMediaElements.current[peerID];

    // Remove username when peer disconnects
    setPeerUsernames(prev => {
      const newUsernames = { ...prev };
      delete newUsernames[peerID];
      return newUsernames;
    });
  }
}, []);

```

```

updateClients(list => list.filter(client => client !== peerID));

// Якщо активне відео було видалене, скидаємо його на локальне
if (activeVideoId === peerID) {
    console.log(`[WebRTC] Active video was from disconnected peer
${peerID}, switching to local video`);
    setActiveVideoId(LOCAL_VIDEO);
}
}

socket.on(ACTIONS.REMOVE_PEER, handleRemovePeer);
console.log('[WebRTC] Registered REMOVE_PEER handler');

return () => {
    socket.off(ACTIONS.REMOVE_PEER, handleRemovePeer);
    console.log('[WebRTC] Cleaned up REMOVE_PEER handler');
};
}, [updateClients, activeVideoId]);

// Add cleanup function
useEffect(() => {
    const handleBeforeUnload = () => {
        cleanup();
    };

    window.addEventListener('beforeunload', handleBeforeUnload);

    return () => {
        window.removeEventListener('beforeunload', handleBeforeUnload);
        cleanup();
    };
}, [cleanup]);

// Modify the media capture effect
useEffect(() => {
    if (!username) return;

    let isMounted = true;

    async function startCapture() {
        console.log('[WebRTC] Starting local media capture...');
        try {
            const stream = await navigator.mediaDevices.getUserMedia({
                audio: true,
                video: { width: 1280, height: 720 },
            });

```

```

    });

    if (!isMounted) {
      stream.getTracks().forEach(track => track.stop());
      return;
    }

    localMediaStream.current = stream;
    console.log('[WebRTC] Local media capture successful');

    addNewClient(LOCAL_VIDEO, () => {
      const localVideoElement =
peerMediaElements.current[LOCAL_VIDEO];
      if (localVideoElement) {
        localVideoElement.volume = 1;
        localVideoElement.srcObject = localMediaStream.current;
        console.log('[WebRTC] Local video element setup complete');
      }
    });

    // Only emit join after media is ready
    socket.emit(ACTIONS.JOIN, { room: roomID, username, password });
  } catch (e) {
    console.error('[WebRTC] Error getting userMedia:', e);
    if (isMounted) {
      cleanup();
    }
  }
}

startCapture();

return () => {
  isMounted = false;
  cleanup();
};
}, [roomID, username, password, addNewClient, cleanup, retryKey]);

// Add effect to update volume when clients change
useEffect(() => {
  const localVideoElement = peerMediaElements.current[LOCAL_VIDEO];
  if (localVideoElement) {
    localVideoElement.volume = 1;
  }
}, [clients.length]);

```

```

const provideMediaRef = useCallback((id: string, node: HTMLVideoElement |
null) => {
  peerMediaElements.current[id] = node;
}, []);

// Цей ref буде для великого активного відео
const provideActiveMediaRef = useCallback((node: HTMLVideoElement | null)
=> {
  activeVideoRef.current = node;
}, []);

// Add function to toggle camera
const toggleCamera = useCallback(() => {
  if (localMediaStream.current) {
    const videoTrack = localMediaStream.current.getVideoTracks()[0];
    if (videoTrack) {
      videoTrack.enabled = !videoTrack.enabled;
      setIsCameraEnabled(videoTrack.enabled);
    }
  }
}, []);

// Add function to toggle microphone
const toggleMicrophone = useCallback(() => {
  if (localMediaStream.current) {
    const audioTrack = localMediaStream.current.getAudioTracks()[0];
    if (audioTrack) {
      audioTrack.enabled = !audioTrack.enabled;
      setIsMicEnabled(audioTrack.enabled);
    }
  }
}, []);

const switchToVideoFile = useCallback(async (file: File) => {
  if (!file) return;

  try {
    // Create a video element to play the file
    const videoElement = document.createElement('video');
    videoElementRef.current = videoElement;

    // Create object URL for the file
    const objectUrl = URL.createObjectURL(file);
    videoElement.src = objectUrl;
  }
});

```

```

// Wait for metadata to load
await new Promise((resolve) => {
  videoElement.onloadedmetadata = () => resolve(null);
});

// Create MediaStream from the video element
let stream: MediaStream;
try {
  // Try the newer captureStream method first
  stream = (videoElement as any).captureStream();
} catch (e) {
  // Fallback to mozCaptureStream for Firefox
  stream = (videoElement as any).mozCaptureStream();
}

// Stop current stream if exists
if (localMediaStream.current) {
  localMediaStream.current.getTracks().forEach(track => track.stop());
}

// Update stream references
videoFileStream.current = stream;
localMediaStream.current = stream;

// Update all peer connections with the new stream
Object.values(peerConnections.current).forEach(pc => {
  const senders = pc.getSenders();
  stream.getTracks().forEach((track, i) => {
    if (senders[i]) {
      senders[i].replaceTrack(track);
    } else {
      pc.addTrack(track, stream);
    }
  });
});

// Update local video element
const localVideoElement = peerMediaElements.current[LOCAL_VIDEO];
if (localVideoElement) {
  localVideoElement.srcObject = stream;
}

```

```

    // Update active video element if showing local video
    if (activeVideoRef.current && activeVideoId === LOCAL_VIDEO) {
        activeVideoRef.current.srcObject = stream;
    }

    // Start playing the video
    await videoElement.play();

    setSelectedVideoFile(file);
    setIsVideoFile(true);

} catch (error) {
    console.error('Error switching to video file:', error);
    alert('Failed to switch to video file. Please try again.');
```

}

```

}, [activeVideoId]);

const switchToCamera = useCallback(async () => {
    // Clean up video file resources
    if (videoElementRef.current) {
        const videoElement = videoElementRef.current;
        // Pause and reset the video
        videoElement.pause();
        videoElement.currentTime = 0;
        // Revoke object URL
        if (videoElement.src) {
            URL.revokeObjectURL(videoElement.src);
        }
        videoElement.src = "";
        videoElement.srcObject = null;
        videoElementRef.current = null;
    }

    // Stop video file stream if exists
    if (videoFileStream.current) {
        videoFileStream.current.getTracks().forEach(track => {
            track.stop();
            track.enabled = false;
        });
        videoFileStream.current = null;
    }

    // Start camera stream
    try {

```

```

const stream = await navigator.mediaDevices.getUserMedia({
  audio: true,
  video: { width: 1280, height: 720 },
});

localMediaStream.current = stream;

// Update all peer connections with the new stream
Object.values(peerConnections.current).forEach(pc => {
  const senders = pc.getSenders();
  stream.getTracks().forEach((track, i) => {
    if (senders[i]) {
      senders[i].replaceTrack(track);
    }
  });
});

// Update local video element
const localVideoElement = peerMediaElements.current[LOCAL_VIDEO];
if (localVideoElement) {
  localVideoElement.srcObject = stream;
}

// Update active video element if showing local video
if (activeVideoRef.current && activeVideoId === LOCAL_VIDEO) {
  activeVideoRef.current.srcObject = stream;
}

setSelectedVideoFile(null);
setIsVideoFile(false);
} catch (e) {
  console.error('Error switching to camera:', e);
}
}, [activeVideoId]);

const toggleScreenShare = useCallback(async () => {
  try {
    if (!isScreenSharing) {
      // Start screen sharing
      const screenStream = await navigator.mediaDevices.getDisplayMedia({
        video: true,
        audio: true
      }).catch((error) => {
        // Handle specific error cases
        if (error.name === 'NotAllowedError') {

```

```

        // User denied permission or closed the dialog
        console.log('[WebRTC] Screen sharing was cancelled or denied by
the user');
        return null;
    }
    throw error; // Re-throw other errors
});

// If user cancelled or denied, exit early
if (!screenStream) {
    return;
}

// Store old tracks to stop them later
const oldTracks = localMediaStream.current?.getTracks() || [];

// Update local stream
localMediaStream.current = screenStream;

// Update all peer connections with the new stream
Object.values(peerConnections.current).forEach(pc => {
    const senders = pc.getSenders();

    // Handle video track
    const videoTrack = screenStream.getVideoTracks()[0];
    const videoSender = senders.find(sender => sender.track?.kind ===
'video');
    if (videoSender && videoTrack) {
        videoSender.replaceTrack(videoTrack);
    }

    // Handle audio track if it exists
    const audioTrack = screenStream.getAudioTracks()[0];
    const audioSender = senders.find(sender => sender.track?.kind ===
'audio');
    if (audioSender && audioTrack) {
        audioSender.replaceTrack(audioTrack);
    }
});

// Update local video element
const localVideoElement =
peerMediaElements.current[LOCAL_VIDEO];
if (localVideoElement) {
    localVideoElement.srcObject = screenStream;

```

```

    }

    // Update active video element if showing local video
    if (activeVideoRef.current && activeVideoId === LOCAL_VIDEO) {
        activeVideoRef.current.srcObject = screenStream;
    }

    // Stop old tracks
    oldTracks.forEach(track => track.stop());

    // Listen for when user stops screen sharing through browser UI
    screenStream.getVideoTracks()[0].addEventListener('ended', () => {
        console.log('[WebRTC] Screen sharing stopped by user');
        switchToCamera();
        setIsScreenSharing(false);
    });

    setIsScreenSharing(true);
} else {
    // Switch back to camera
    await switchToCamera();
    setIsScreenSharing(false);
}
} catch (error) {
    console.error('[WebRTC] Error in screen sharing:', error);
    // Reset state and switch back to camera if there's an error
    setIsScreenSharing(false);
    await switchToCamera().catch(e =>
        console.error('[WebRTC] Error switching back to camera:', e)
    );
}
}, [isScreenSharing, activeVideoId, switchToCamera]);

return {
    clients,
    provideMediaRef,
    activeVideoId,
    setActiveVideoId,
    provideActiveMediaRef,
    messages,
    broadcastMessage,
    peerUsernames,
    toggleCamera,
    isCameraEnabled,
    toggleMicrophone,

```

```

    isMicEnabled,
    switchToVideoFile,
    switchToCamera,
    isVideoFile,
    selectedVideoFile,
    toggleScreenShare,
    isScreenSharing,
  };
}

```

B.9 src\lib\socket.ts

```

import {io} from 'socket.io-client';

const options = {
  "force new connection": true,
  reconnectionAttempts: Infinity,
  timeout : 10000,
  transports : ["websocket"]
}

const socket = io('http://localhost:3001', options);

export default socket;

```

B.10 src\lib\utils.ts

```

import { clsx, type ClassValue } from "clsx"
import { twMerge } from "tailwind-merge"

export function cn(...inputs: ClassValue[]) {
  return twMerge(clsx(inputs))
}

```

ДОДАТОК В
Слайди презентації

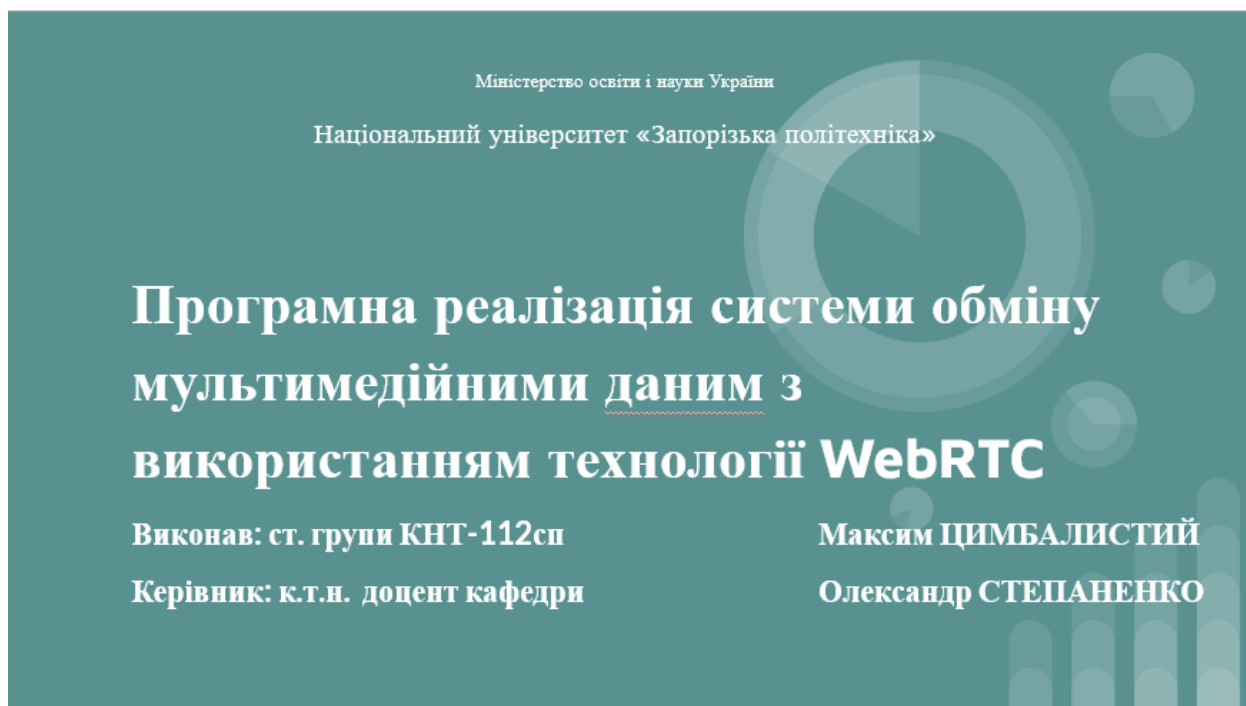


Рисунок В.1 – Слайд №1



Об'єкт дослідження та мета

Об'єкт дослідження – процес розроблення програмного забезпечення для обміну мультимедійними даними в реальному часі.

Предмет дослідження – методи та програмні засоби для реалізації peer-to-peer відеозв'язку з використанням технології WebRTC.

Мета роботи – підвищення ефективності обміну мультимедійними даними шляхом розробки веб-застосунку на основі технології WebRTC, що забезпечує передачу відео, аудіо та текстових повідомлень між користувачами в режимі реального часу.

Рисунок В.2 – Слайд №2



Функціональні вимоги системи

- Проведення групових відеодзвінків із передачею аудіо/відео.
- Обмін текстовими повідомленнями через чат.
- Демонстрацію екрану та відтворення відеофайлів.
- Автоматичне встановлення прямих з'єднань між користувачами.

Рисунок В.3 – Слайд №3



Вибір мови програмування

Мова	Розвиненість екосистеми	Підтримка WebSocket/WebRTC	Зручність роботи з JSON	<u>Кросплатформеність</u>	Простота підтримки	Можливості створення SPA/API
JavaScript / Node.js	Висока	Відмінна	Нативна	Висока	Низька	Так
Python	Висока	Середня	Висока	Висока	Низька	Так
Java	Висока	Середня	Середня	Висока	Середня	Так
Go	Середня	Висока	Середня	Висока	Середня	Так
Rust	Зростаюча	Висока	Середня	Висока	Висока	Так (з обмеженнями)
C#	Висока (для .NET)	Висока	Середня	Висока	Середня	Так

Рисунок В.4 – Слайд №4

Використані технічні засоби

Категорія	Обраний засіб	Аналоги	Переваги обраного засобу
Серверна платформа	Node.js	Python (Django), PHP, Java	Асинхронність, висока продуктивність у реальному часі, єдина мова з фронтом
Фреймворк	Next.js	Create React App, Nuxt, Remix	SSR, App Router, SEO-переваги, гнучка структура проекту
Фронтенд бібліотека	React.js	Vue.js, Angular, Svelte	Велика спільнота, сумісність з екосистемами (Next.js, React Hook Form тощо)
Обмін даними	Socket.IO	WebSocket API, SignalR, MQTT	Простота, автоматичне повторне з'єднання, події в реальному часі
Відеозв'язок	WebRTC	Zoom SDK, Jitsi, Twilio Video	Відкрите API, peer-to-peer, не потребує серверів для стріму
Стилізація UI	TailwindCSS	Bootstrap, MUI, Chakra UI	Утилітарний підхід, адаптивність, швидкість розробки
UI-компоненти	shadcn/ui	Radix UI, Headless UI, MUI	На основі Radix, сучасний дизайн, гнучкість кастомізації

Рисунок В.5 – Слайд №5

Порівняння сучасних фреймворків веб-розробки

Характеристика/Фреймворк	Next.js (JS/TS)	Remix (JS/TS)	Django (Python)	FastAPI (Python)	Ruby on Rails (Ruby)
Призначення	Фулстек / Фронтенд з SSR/SSG	Фулстек / Фронтенд з SSR/SSG	Бекенд	Бекенд	Бекенд
Основна мова	JavaScript/TypeScript	JavaScript/TypeScript	Python	Python	Ruby
Основний фокус	Продуктивність, SEO, DX	Продуктивність, веб-стандарт, DX	Швидка розробка, "батареїки"	Продуктивність, API, async	Швидка розробка, конвенції
Підтримка WebSockets	Потребує інтеграції	Потребує інтеграції Loaders/ Actions для початкових запитів	Django Channels	Вбудована	Action Cable (вбудований)

Рисунок В.5 – Слайд №6

Структурна схема застосунку

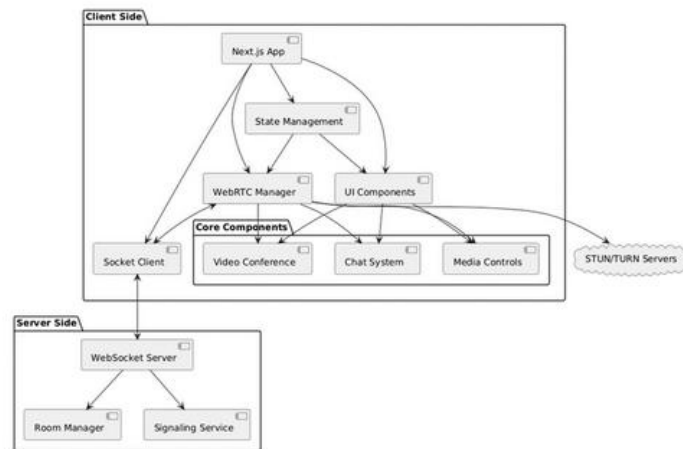


Рисунок В.6 – Слайд №7

Схема зв'язку full mesh для чотирьох учасників

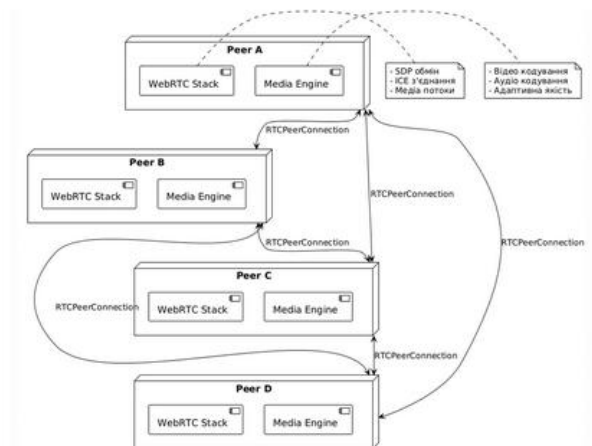


Рисунок В.7 – Слайд №8

Функціональна схема роботи відеоконференції

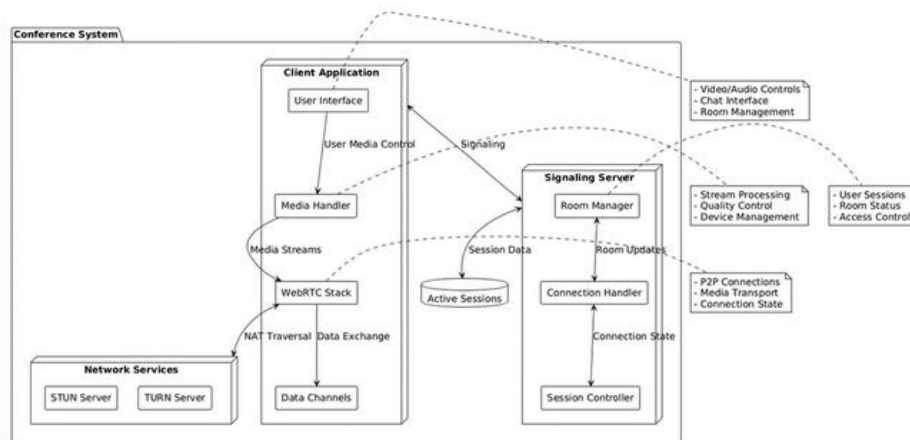


Рисунок В.8 – Слайд №9

Діаграма встановлення зв'язку

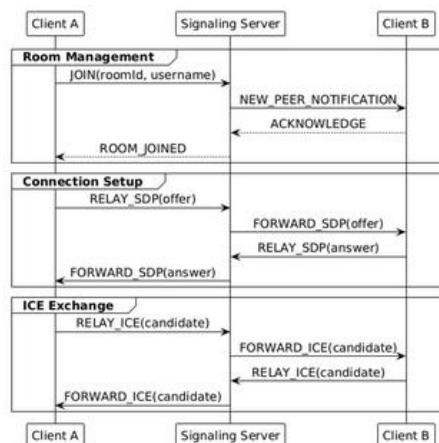


Рисунок В.9 – Слайд №10

Сторінка відеоконференції розробленого застосунку

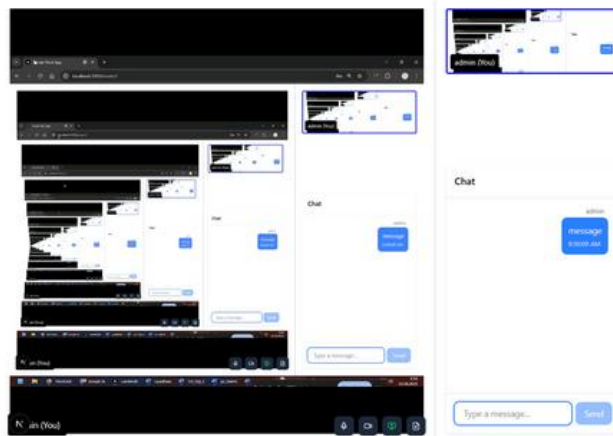


Рисунок В.10 – Слайд №11

Висновки

Проаналізовано існуючі альтернативи розроблюваного сервісу

Проаналізовані сучасні фреймворки веб-розробки

Розроблено веб-додаток для відеоконференцій з використанням WebRTC.

Реалізовано: відео/аудіо зв'язок (HD), демонстрація екрана, чат, обмін файлами, трансляція відео.

Інтерфейс зручний та інтуїтивно зрозумілий.

Працює в усіх сучасних браузерях, без встановлення додаткового ПЗ.

Технології: React.js, TypeScript, WebRTC API, Socket.IO, STUN/TURN.

Використано: компонентний підхід, React Hooks, асинхронність, адаптивний дизайн.

Система протестована та готова до впровадження в освіті, бізнесі, онлайн-зустрічах.

Функціонал системи повністю відповідає вимогам технічного завдання

Рисунок В.11 – Слайд №12