МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Західноукраїнський національний університет
Факультет комп'ютерних інформаційних технологій
Кафедра інформаційно-обчислювальних систем і управління

МУДРАК Діана Віталіївна

Програмний модуль WebRTC-клієнта для Ant Media Server / Software Module of a WebRTC Client for Ant Media Server

спеціальність: 122 – Комп'ютерні науки
освітньо-професійна програма – Комп'ютерні науки

Кваліфікаційна робота

Виконала студентка групи КН-41
Д.І. Мудрак

Науковий керівник
к.т.н., доцент П.Є. Биковий

Кваліфікаційну роботу допущено до захисту
«___»_____ 20___ р.

Завідувач кафедри
_____ Н.В. Дзюбановська

ТЕРНОПІЛЬ – 2026

Факультет комп'ютерних інформаційних технологій
Кафедра інформаційно-обчислювальних систем і управління
Освітній ступінь «бакалавр»
спеціальність: 122 – Комп'ютерні науки
освітньо-професійна програма – Комп'ютерні науки

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
_____ Н.В. Дзюбановська
«____» _____20__р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ
МУДРАК Діана Віталіївна
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи: **Програмний модуль WebRTC-клієнта для Ant Media Server / Software Module of a WebRTC Client for Ant Media Server**

керівник роботи к.т.н., доцент, П.Є.Биковий
затверджений наказом по університету від 01 грудня 2025 року № 894.

2. Строк подання студентом закінченої кваліфікаційної роботи 25 травня 2026 р.

3. Вихідні дані до кваліфікаційної роботи: завдання на кваліфікаційну роботу студента, наукові статті та технічна документація з технологій WebRTC і потокової передачі мультимедійних даних, документація Ant Media Server, специфікації протоколів WebSocket, SDP, ICE, STUN/TURN, документація Flutter та Dart.

4. Основні питання, які потрібно розробити:
- проаналізувати сучасні технології потокової передачі мультимедійних даних у режимі реального часу, архітектуру WebRTC та особливості функціонування Ant Media Server;
- спроєктувати архітектуру програмного модуля WebRTC-клієнта, обґрунтувати вибір технологічного стеку та патернів проєктування;
- розробити алгоритми сигналізації, обробки SDP-конфігурацій, збору та передачі ICE-кандидатів, а також механізми керування аудіо- та відеопотоками;
- реалізувати програмний модуль із підтримкою автоматичного перепідключення та відновлення сесії, провести тестування й оцінити показники продуктивності та стабільності роботи системи.

5. Перелік графічного матеріалу у роботі:
- діаграма взаємодії користувача (Use Case) програмного модуля;
- структурна схема взаємодії WebRTC-клієнта з серверною інфраструктурою;
- функціональна схема внутрішніх компонентів програмного модуля;
- граф переходів моделі життєвого циклу розробленого модуля;
- алгоритм обробки мережевих збоїв та автономного відновлення сесії;
- UML-діаграма послідовності виконання алгоритму ICE Restart.

6. Консультанти розділів кваліфікаційної роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання 01 грудня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів кваліфікаційної роботи	Примітка
1	Огляд літературних джерел відповідно до предметної області та складання плану роботи	до 01.01.2026 р.	
2	Написання 1 розділу кваліфікаційної роботи	до 01.02.2026 р.	
3	Написання 2 розділу кваліфікаційної роботи	до 15.03.2026 р.	
4	Написання 3 розділу кваліфікаційної роботи	до 01.05.2026 р.	
5	Представлення попереднього варіанту кваліфікаційної роботи, перевірка та внесення змін керівником	до 15.05.2026 р.	
6	Опрацювання зауважень та представлення завершеного варіанту кваліфікаційної роботи. Підготовка супроводжуючих документів	до 25.05.2026 р.	
7	Перевірка кваліфікаційної роботи на оригінальність тексту	до 30.05.2026 р.	
8	Оформлення кваліфікаційної роботи та отримання допуску до захисту	до 10.06.2026 р.	
9	Подання кваліфікаційної роботи до захисту на засіданні атестаційної комісії	до 10.06.2026 р.	

Студент _____ Д.В. Мудрак
підпис

Керівник роботи _____ к.т.н., доцент П.Є.Биковий

АНОТАЦІЯ

Кваліфікаційна робота на тему «Програмний модуль WebRTC-клієнта для Ant Media Server» на здобуття освітнього ступеня «бакалавр» зі спеціальності 122 «Комп'ютерні науки» освітньої програми «Комп'ютерні науки» написана обсягом у 62 сторінки і містить 12 ілюстрацій, 5 таблиць, 3 додатки та 26 використаних джерел.

Метою кваліфікаційної роботи є розробка та програмна реалізація кросплатформного програмного модуля WebRTC-клієнта для взаємодії з Ant Media Server, який забезпечує передачу мультимедійних даних у режимі реального часу з низькою затримкою та підвищеною відмовостійкістю.

Методами дослідження є методи системного аналізу, проєктування програмного забезпечення, мережевих комунікацій, об'єктно-орієнтованого програмування, аналізу протоколів потокової передачі даних та тестування програмних систем.

Внаслідок виконання роботи розроблено архітектуру програмного модуля WebRTC-клієнта на основі технологій Flutter та Dart для інтеграції з Ant Media Server. Реалізовано механізми встановлення WebSocket-з'єднання, обміну SDP-конфігураціями та передачі ICE-кандидатів, а також засоби керування аудіо- та відеопотоками. Додатково впроваджено алгоритми автоматичного перепідключення та відновлення сесії при втраті мережевого з'єднання, а також проведено тестування й оцінювання продуктивності розробленого модуля.

Розроблений програмний продукт є готовим до використання програмним модулем для створення мобільних та веборієнтованих стрімінгових застосунків. Його можна використовувати у системах відеоконференцій, дистанційного навчання, відеоспостереження, телемедицини та інших сервісах, що потребують передачі мультимедійних даних у режимі реального часу з мінімальною затримкою.

Ключові слова: WEBRTC, ANT MEDIA SERVER, ПОТОКОВЕ ВІДЕО, МУЛЬТИМЕДІЙНІ ДАНІ, WEBSOCKET, FLUTTER, DART.

ABSTRACT

Qualification work on the topic « Software Module of a WebRTC Client for Ant Media Server» for obtaining a bachelor's degree in specialty 122 «Computer Science» of the educational program «Computer Science» is written on 62 pages and contains 12 illustrations, 5 table, 3 appendices, and 26 references.

The aim of the qualification work is to develop and implement a cross-platform WebRTC client software module for interaction with Ant Media Server, providing real-time multimedia data transmission with low latency and enhanced fault tolerance.

Research methods include system analysis, software engineering methods, network communication technologies, object-oriented programming, streaming media protocols analysis, and software testing methods.

As a result of the research, the architecture of a WebRTC client software module based on Flutter and Dart technologies was developed. Mechanisms for WebSocket connection establishment, SDP configuration exchange, ICE candidate transmission, audio and video stream management, as well as automatic reconnection and session recovery in case of network failures were implemented. The developed module was experimentally tested and evaluated in terms of performance, stability, and transmission latency.

The developed software product is a ready-to-use software module for building mobile and web-based streaming applications. It can be applied in video conferencing systems, distance learning platforms, telemedicine services, video surveillance systems, and other real-time multimedia communication solutions.

Keywords: WEBRTC, ANT MEDIA SERVER, STREAMING VIDEO, WEBSOCKET, FLUTTER, DART.

.

ЗМІСТ

ВСТУП

Сучасний етап розвитку інформаційних технологій характеризується стрімким зростанням попиту на системи інтерактивної взаємодії в реальному часі. Платформи для проведення відеоконференцій, телемедицини, дистанційного навчання, а також системи управління автономними пристроями (інтернет речей, дрони) вимагають забезпечення передачі мультимедійних даних із мінімально можливою затримкою. Ключову роль у цих процесах відіграють мережеві протоколи та технології потокового відео, що забезпечують безперервну доставку високоякісних аудіо- та відеоданих кінцевому користувачу в умовах динамічних змін пропускної здатності мережі.

Незважаючи на широке використання традиційних протоколів потокової передачі, таких як RTMP (Real-Time Messaging Protocol) та HLS (HTTP Live Streaming), проблема забезпечення ультранизької затримки (sub-second latency) залишається надзвичайно гострою. Практика застосування HTTP-базованих рішень свідчить про наявність значних затримок (від кількох секунд до десятків секунд), що робить їх фундаментально непридатними для систем двосторонньої комунікації. З іншого боку, протокол WebRTC, який вирішує проблему затримки завдяки використанню транспорту UDP та архітектури Peer-to-Peer, є концептуально складним в імплементації, особливо в умовах масштабування на велику аудиторію глядачів.

Актуальність теми кваліфікаційної роботи зумовлена необхідністю спрощення та прискорення процесу розробки стрімінгових застосунків шляхом створення спеціалізованих програмних модулів для взаємодії з потужними медіасерверами. Для подолання обмежень масштабування базового WebRTC сьогодні активно використовуються сервери з топологією SFU (Selective Forwarding Unit), лідером серед яких є рішення Ant Media Server. Проте інтеграція з таким сервером вимагає реалізації специфічної і складної логіки асинхронної сигналізації через протокол WebSocket, управління станами SDP-сесій, збору ICE-кандидатів та прямого управління апаратними ресурсами мобільних пристроїв.

Додатковим фактором, що підсилює актуальність дослідження, є

нестабільність сучасних бездротових мереж (Wi-Fi, 4G/LTE), у яких працюють мобільні клієнти. Тимчасові обриви зв'язку або зміна IP-адреси пристрою неминуче призводять до руйнування WebRTC-з'єднання. Розробка інкапсульованого програмного модуля з вбудованими інтелектуальними алгоритмами відмовостійкості (такими як експоненційна затримка при перепідключенні та автоматичний ICE Restart) дозволяє суттєво підвищити надійність стрімінгових рішень, знімаючи цей тягар з розробників кінцевого інтерфейсу.

У зв’язку з цим виникає науково-практична потреба у розробці архітектурно досконалого програмного рішення (SDK), яке інтегрує складні мережеві алгоритми WebRTC у єдиний конвеєр обробки та надає розробникам простий фасадний інтерфейс для управління потоками.

Метою кваліфікаційної роботи є підвищення ефективності розробки та стабільності функціонування застосунків потокового відео шляхом проектування та практичної реалізації кросплатформного програмного модуля WebRTC-клієнта для взаємодії з Ant Media Server.

Для досягнення поставленої мети в роботі необхідно розв’язати такі основні завдання:

– провести аналіз сучасних технологій і протоколів потокової передачі мультимедійних даних;

– дослідити архітектуру Ant Media Server та специфікацію його сигналізаційного протоколу;

– обґрунтувати вибір технологічного стеку та спроектувати архітектуру програмного модуля з використанням структурних і поведінкових патернів проектування;

– розробити логічні схеми та реалізувати алгоритми сигналізації, обміну SDP-пропозиціями та збору ICE-кандидатів;

– імплементувати механізми відмовостійкості, автоматичного відновлення з'єднання та управління апаратними медіа-треками пристрою;

– провести експериментальне тестування програмного модуля та оцінити показники його ефективності (наскрізної затримки та стабільності роботи).

Об’єктом дослідження є процеси передачі мультимедійних даних у

реальному часі в розподілених комп'ютерних мережах.

Предметом дослідження є методи, алгоритми та програмні інструменти побудови клієнтських WebRTC-модулів для взаємодії з інфраструктурою медіасерверів.

Особливість отриманих результатів полягає в удосконаленні архітектури програмних WebRTC-клієнтів шляхом впровадження автоматної моделі управління станами та алгоритмів автономного відновлення сесії (ICE Restart) на рівні системного модуля. Запропонований підхід дозволяє унеможливити виникнення станів гонитви (Race Conditions) під час обробки асинхронних мережевих подій та суттєво мінімізувати час переривання мультимедійної трансляції при зміні топології мережі кінцевого пристрою.

Практична значимість роботи полягає у створенні повністю функціонального кросплатформного програмного модуля (бібліотеки) на базі мови Dart, який може бути безпосередньо використаний у складі реальних комерційних систем для задач потокової передачі відео. Результати дослідження суттєво скорочують час виходу на ринок (Time-to-Market) при розробці мобільних застосунків для інтерактивних трансляцій, відеоспостереження чи стрімінгових платформ, а також можуть бути використані у навчальному процесі при вивченні технологій програмування мережевих систем.

Результати кваліфікаційної роботи пройшли апробацію на Міжнародній науковій інтернет-конференції «Інформаційне суспільство: технологічні, економічні та технічні аспекти становлення» (м. Тернопіль, Україна – м. Ополе, Польща, 4–5 червня 2026 р.) та опубліковані у збірнику матеріалів конференції у вигляді тез доповіді «Програмний модуль WebRTC-клієнта для Ant Media Server» (Додаток А).

1 ТЕОРЕТИЧНІ ОСНОВИ

1.1 Огляд еволюції технологій потокової передачі даних у реальному часі

Розвиток інтерактивних веб-систем, платформ для проведення відеоконференцій, телемедицини та систем дистанційного керування (наприклад, IoT-пристроями або дронами) вимагає забезпечення передачі мультимедійних даних із мінімально можливою затримкою. Традиційні архітектури доставки контенту, які створювалися для трансляції телебачення в Інтернеті або розповсюдження відео на вимогу (VOD), виявилися нездатними задовольнити вимоги сучасних інтерактивних систем, що призвело до еволюції мережевих протоколів потокової передачі [1].

Історично першим успішним стандартом для передачі потокового відео з низькою затримкою став протокол RTMP (Real-Time Messaging Protocol), розроблений компанією Macromedia. Він базується на протоколі транспортного рівня TCP (Transmission Control Protocol), що гарантує цілісність доставки пакетів. Однак використання TCP для потокового відео реального часу має фундаментальний недолік: механізм повторної передачі втрачених пакетів (retransmission) та алгоритми контролю перевантаження мережі неминуче призводять до явища буферизації. У разі нестабільного інтернет-з'єднання TCP зупиняє передачу нових кадрів доти, доки не будуть успішно доставлені попередні. Крім того, з відмовою індустрії від технології Adobe Flash, протокол RTMP втратив нативну підтримку в сучасних браузерах і сьогодні використовується переважно лише як протокол публікації (ingest) від апаратних та програмних енкодерів до медіасерверів [2].

Наступним етапом стала поява протоколів на базі HTTP, серед яких найбільш поширеними є HLS (HTTP Live Streaming) від Apple та MPEG-DASH. Їхня архітектура кардинально відрізняється: суцільний медіапотік розбивається енкодером на невеликі фрагменти (чанки) тривалістю від 2 до 10 секунд, які клієнт завантажує послідовно через стандартні GET-запити. Цей підхід забезпечує неперевершену масштабованість, оскільки фрагменти можуть кешуватися на вузлах CDN (Content Delivery Network). Проте платою за масштабованість є величезна наскрізна затримка (Glass-to-Glass Latency), яка в класичному HLS становить 15–30

секунд. Навіть із застосуванням сучасних розширень Low-Latency HLS (LL-HLS) затримку вдається знизити лише до 2–5 секунд, що все ще неприйнятно для систем, які вимагають двосторонньої комунікації [3].

Таблиця 1.1 — Порівняльна характеристика мережевих протоколів потокового відео

Характеристика	RTMP	HLS / LL-HLS	WebRTC
Транспортний протокол	TCP	TCP (HTTP)	UDP (RTP/RTCP)
Середня затримка (Latency)	1 – 3 секунди	10 – 30 с. / 2 – 5 с.	< 0.5 секунди
Адаптивний бітрейт (ABR)	Не підтримується	Нативна підтримка	Динамічний (через RTCP)
Нативна підтримка браузерами	Немає (потрібен Flash)	Так (частково через MSE)	Так (HTML5)
Основне застосування	Ingest (публікація на сервер)	Масове мовлення (Broadcasting)	Інтерактивний зв'язок у реальному часі

Вирішенням проблеми затримки стала поява стандарту WebRTC (Web Real-Time Communication). На відміну від попередників, WebRTC від початку проектувався для комунікації «рівний-рівному» (Peer-to-Peer, P2P) з використанням протоколу транспортного рівня UDP (User Datagram Protocol). UDP не гарантує доставку пакетів, що ідеально підходить для відеозв'язку: якщо відеокадр губиться в мережі, система просто ігнорує його і відтворює наступний, уникнувши затримок на пересилання. Це дозволяє досягти ультранизької затримки (sub-second latency), яка зазвичай не перевищує 500 мілісекунд, що є критичною вимогою для розроблюваного в даній роботі програмного модуля [4].

1.2 Огляд і аналіз існуючих рішень

Технологія WebRTC не становить собою єдиний монолітний протокол, а є комплексним набором програмних інтерфейсів (API) та мережевих стандартів,

затверджених консорціумом W3C (World Wide Web Consortium) та інженерною радою IETF (Internet Engineering Task Force). Проєктування та розробка клієнтського програмного забезпечення вимагає ґрунтовного розуміння внутрішніх механізмів ініціалізації з'єднань та маршрутизації мультимедійних даних.

Архітектура сеансу зв'язку на базі WebRTC структурно поділяється на три ключові рівні: рівень сигналізації, рівень обходу трансляції мережевих адрес (NAT) та транспортний рівень передачі медіаданих [5].

Рівень сигналізації (Signaling Plane) Характерною особливістю стандарту WebRTC є відсутність жорсткої регламентації протоколу сигналізації. Процес сигналізації полягає у взаємному виявленні мережевих вузлів та попередньому узгодженні параметрів сеансу до початку трансляції медіапотоків. Вибір транспортного протоколу для сигналізації покладається на розробника системи (найбільш поширеними рішеннями є використання технології WebSockets, рідше — Server-Sent Events або протокол SIP поверх WebSocket).

Ключовим інформаційним об'єктом, що передається під час сигналізації, є дескриптор сесії у форматі SDP (Session Description Protocol). Інформаційна взаємодія між вузлами реалізується за моделлю «Offer/Answer» («Пропозиція/Відповідь»), структурно-логічну схему якої наведено на рисунку 1.1. Цей алгоритм включає наступні етапи:

– вузол-ініціатор з'єднання (клієнтський модуль) ініціалізує об'єкт RTCPeerConnection та формує пропозицію сесії (*SDP Offer*). Цей пакет містить конфігураційні дані щодо підтримуваних медіакодеків (зокрема H.264, VP8, Opus), структури аудіо- та відеопотоків, криптографічних параметрів та топології мережі;

– згенерована пропозиція маршрутизується через обраний транспортний канал (WebSocket) до сигнального сервера (у межах даного проєкту розглядається використання Ant Media Server);

– програмне забезпечення сервера виконує синтаксичний та логічний аналіз отриманого *SDP Offer*, зіставляє його з власними апаратними і програмними можливостями та генерує пакет-відповідь (*SDP Answer*);

– клієнтський модуль обробляє отриману відповідь, чим завершує фазу логічного встановлення сеансу [6].

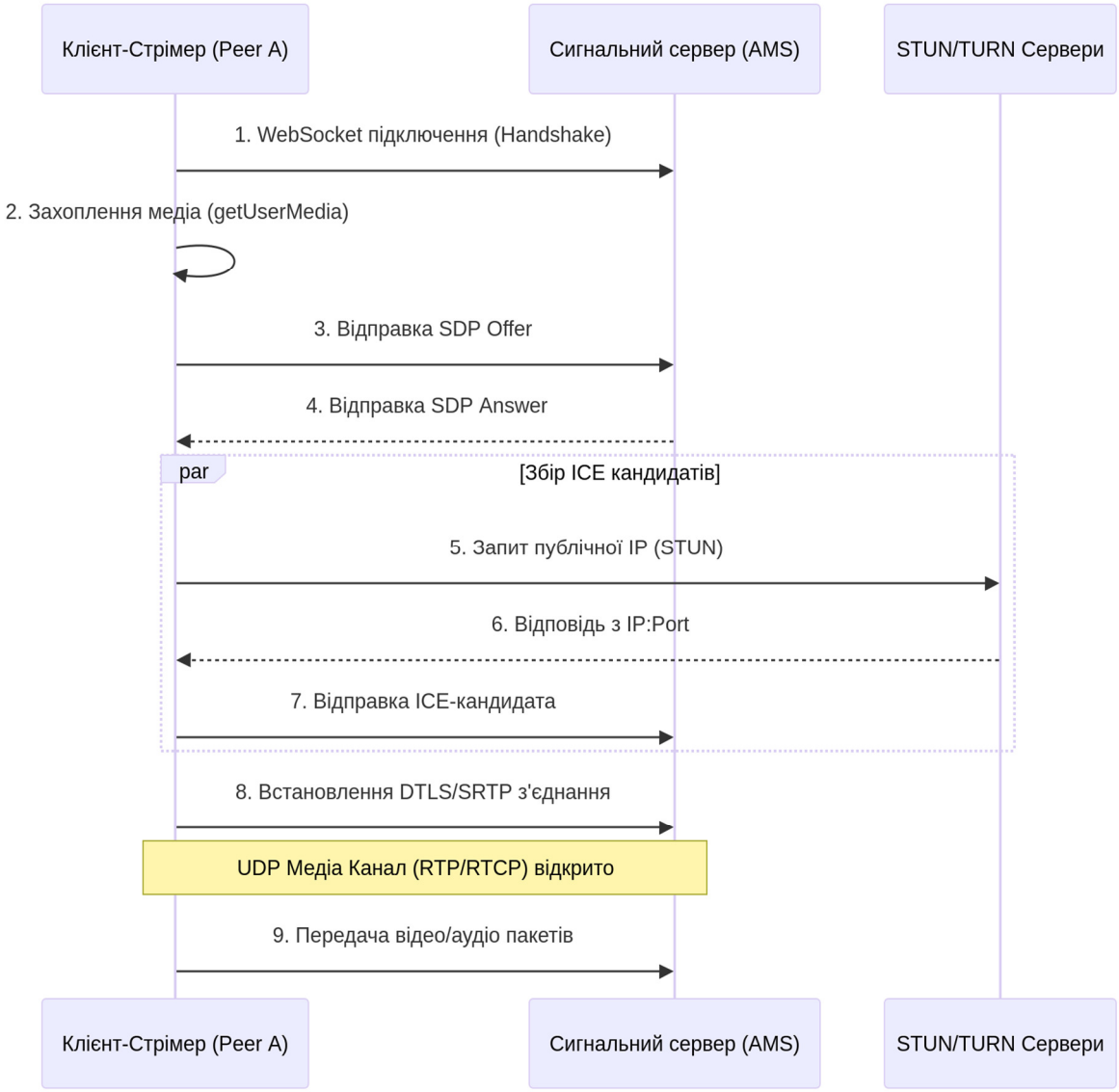


Рисунок 1.1 – Узагальнена схема сигналізації та встановлення P2P з'єднання у WebRTC

Важливим етапом узгодження сесії (SDP Offer/Answer) є вибір оптимального медіакодека. Специфікація WebRTC вимагає від клієнтських платформ обов'язкової підтримки відеокодеків VP8 та H.264, а також аудіокодека Opus для забезпечення гарантованої кросплатформної сумісності. Для мобільних пристроїв критичним аспектом є використання механізмів апаратного прискорення кодування та декодування відео (Hardware Acceleration). Використання апаратних реалізацій кодека H.264 замість програмної обробки дозволяє суттєво знизити навантаження на центральний процесор (CPU) пристрою, зменшити енергоспоживання (battery drain) та запобігти термічному тротлінгу (thermal throttling) під час тривалих трансляцій високої роздільної здатності [4, 5].

Після успішного узгодження параметрів сесії виникає необхідність встановлення прямого мережевого маршруту між кінцевими точками. У зв'язку з тим, що переважна більшість клієнтських пристроїв функціонують за маршрутизаторами, які використовують технологію трансляції мережевих адрес (Network Address Translation, NAT), і позбавлені маршрутизованих (публічних) IP-адрес, встановлення прямого з'єднання здебільшого є неможливим. Для подолання топологічних обмежень NAT у WebRTC інтегровано протокол ICE (Interactive Connectivity Establishment).

Механізм ICE здійснює процес агрегації ICE-кандидатів — масиву потенційних мережевих адрес та портів для встановлення з'єднання. Цей процес забезпечується за допомогою допоміжних мережевих вузлів:

– сервери STUN (Session Traversal Utilities for NAT): використовуються клієнтами для визначення власної зовнішньої IP-адреси та порту, які були призначені пристроєм NAT. Отримані дані формують кандидата, що дозволяє встановити пряме з'єднання для нестрогих типів NAT (Full-cone або Restricted cone NAT);

– сервери TURN (Traversal Using Relays around NAT): задіюються у випадках, коли вузли знаходяться за симетричним NAT (Symmetric NAT) або суворими корпоративними міжмережевими екранами, що блокують прямі з'єднання. TURN-сервер функціонує як транзитний вузол (ретранслятор), який приймає та перенаправляє медіатрафік між учасниками сесії (або медіасервером). Оскільки ретрансляція створює значне навантаження на обчислювальні ресурси сервера, використання TURN розглядається виключно як резервний механізм [7].

Специфіка функціонування мобільних WebRTC-клієнтів передбачає часту зміну мережевого середовища (наприклад, плавний перехід смартфона з мережі Wi-Fi на стільникову мережу 4G/LTE), що призводить до миттєвої зміни локальної та публічної IP-адреси пристрою. Для розв'язання цієї проблеми в архітектуру WebRTC закладено механізм ICE Restart [4, 5]. Цей процес дозволяє клієнтському модулю ініціювати повторний збір кандидатів та узгодити новий мережевий маршрут шляхом генерації оновленого SDP-пакета без необхідності повної руйнації об'єкта RTCPeerConnection та повторного захоплення медіапотоків з камери [24]. Реалізація

підтримки ICE Restart у розроблюваному модулі є критичною вимогою для забезпечення безшовності потокової передачі в умовах нестабільного зв'язку.

Після завершення збору та верифікації ICE-кандидатів ініціюється фаза передачі мультимедійних даних. Транспортування інкапсульованих аудіо- та відеокадрів здійснюється на базі протоколу RTP (Real-time Transport Protocol), що функціонує поверх протоколу дейтаграм UDP. Для моніторингу якості обслуговування (Quality of Service, QoS) паралельно застосовується протокол RTCP (RTP Control Protocol). Він забезпечує збір телеметричних даних сеансу (відсоток втрачених пакетів, рівень джиттеру, мережеві затримки), що дозволяє реалізовувати алгоритми динамічної адаптації бітрейту відеопотоку відповідно до поточної пропускної здатності каналу зв'язку.

З метою гарантування конфіденційності та цілісності даних, стандарт WebRTC висуває сувору вимогу щодо наскрізного шифрування всього генерованого трафіку. Криптографічний захист реалізується за допомогою комбінації протоколів: DTLS (Datagram Transport Layer Security) використовується для безпечного обміну ключами шифрування, тоді як протокол SRTP (Secure Real-time Transport Protocol) забезпечує безпосереднє шифрування медіапотоку. Такий комплексний підхід ефективно нівелює загрози несанкціонованого перехоплення та модифікації відеоданих, зокрема атаки типу «людина посередині» (Man-in-the-Middle) [8].

Хоча протоколи DTLS та SRTP гарантують конфіденційність медіатрафіку на транспортному рівні, архітектура клієнтського модуля також повинна враховувати механізми авторизації та захисту на рівні сигналізації. Взаємодія з Ant Media Server вимагає використання захищеного з'єднання WebSocket Secure (WSS), яке базується на протоколі TLS, для запобігання перехопленню конфігураційних SDP-повідомлень та обміну ключами. Крім того, для захисту інфраструктури від несанкціонованого доступу (як публікації, так і перегляду потоків), медіасервер використовує систему динамічних токенів (One-Time Tokens або JWT-токени) [9]. Програмний модуль клієнта має підтримувати передачу цих токенів під час ініціалізації з'єднання або безпосередньо у сигнальних командах publish та play [10].

1.3 Постановка задачі

Розробка WebRTC-клієнта вимагає глибокого розуміння серверної архітектури, з якою він буде взаємодіяти. У багатокористувацьких системах архітектура Peer-to-Peer (P2P), де кожен клієнт встановлює пряме з'єднання з усіма іншими учасниками, є нежиттєздатною (Mesh-топологія). Якщо у трансляції бере участь N глядачів, стрімеру необхідно відправляти N незалежних відеопотоків, що миттєво вичерпає пропускну здатність його мережі та перевантажить процесор.

Для вирішення цієї проблеми використовуються проміжні медіасервери. Програмне забезпечення Ant Media Server (AMS) є високопродуктивним рішенням для потокової передачі відео з ультранизькою затримкою, яке використовує підхід SFU (Selective Forwarding Unit) [9].

Для організації багатокористувацьких трансляцій у реальному часі критично важливим є вибір архітектурної топології сервера. Історично в інженерії потокового відео сформувалися два фундаментально різні підходи до управління медіапотоками: MCU (Multipoint Control Unit) та SFU (Selective Forwarding Unit).

У класичній топології MCU сервер діє як центральний вузол обробки мультимедіа. Він приймає індивідуальні медіапотоки від кожного учасника трансляції, здійснює їх повне декодування, апаратне або програмне об'єднання (мікшування) у єдиний спільний кадр і повторно кодує цей комбінований потік для відправки кожному клієнту. Схематичне зображення топології MCU наведено на рисунку 1.2.

З одного боку, підхід MCU мінімізує навантаження на канал зв'язку (bandwidth) кінцевого користувача, оскільки він отримує лише один агрегований потік незалежно від кількості учасників. Однак ця архітектура має критичний недолік: вона створює колосальне обчислювальне навантаження на центральний процесор (CPU) сервера. Процеси декодування та транскодування в реальному часі вимагають значних апаратних ресурсів, що робить горизонтальне масштабування системи фінансово та технічно неефективним. Крім того, багаторазова обробка кадрів неминуче вносить додаткову наскрізну затримку (latency), що нівелює головну перевагу протоколу WebRTC. На противагу застарілим рішенням, сучасні

високопродуктивні платформи, зокрема інфраструктура Ant Media Server (AMS), будуються на базі архітектури SFU.

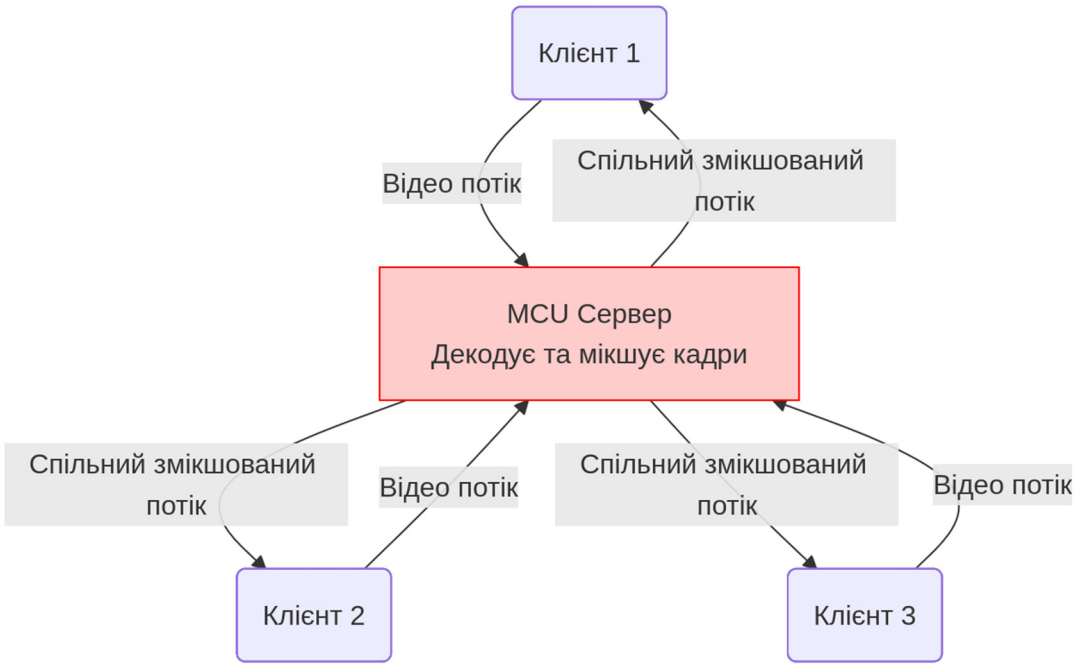


Рисунок 1.2 – Топологія MCU з централізованим мікшуванням потоків

Топологія SFU працює як високошвидкісний інтелектуальний маршрутизатор медіапакетів на рівні протоколу RTP. У цій моделі мобільний або веб-клієнт (стрімер) відправляє на сервер лише один висхідний (upstream) медіапотік. При цьому медіасервер не виконує ресурсомістких операцій декодування чи перекодування відеоданих. Його головна задача — аналізувати заголовки вхідних мережевих пакетів та вибірково пересилати (forwarding) їх усім підключеним клієнтам-глядачам у незмінному вигляді. Принцип роботи топології SFU відображено на рисунку 1.3.

Використання архітектури SFU в Ant Media Server забезпечує низку фундаментальних переваг для систем реального часу. По-перше, навантаження на процесорні потужності сервера знижується в десятки разів у порівнянні з MCU, що дозволяє одному серверному екземпляру обслуговувати тисячі одночасних підключень. По-друге, відсутність етапів транскодування гарантує збереження ультранизької затримки (sub-second latency). По-третє, такий підхід дозволяє

розроблюваному клієнтському модулю гнучко управляти отриманими даними: мобільний пристрій приймає окремі потоки і самостійно відповідає за їх компонування (рендеринг) на екрані, адаптуючись до розмірів дисплея та стану мережі конкретного глядача за допомогою технологій Simulcast або SVC (Scalable Video Coding).

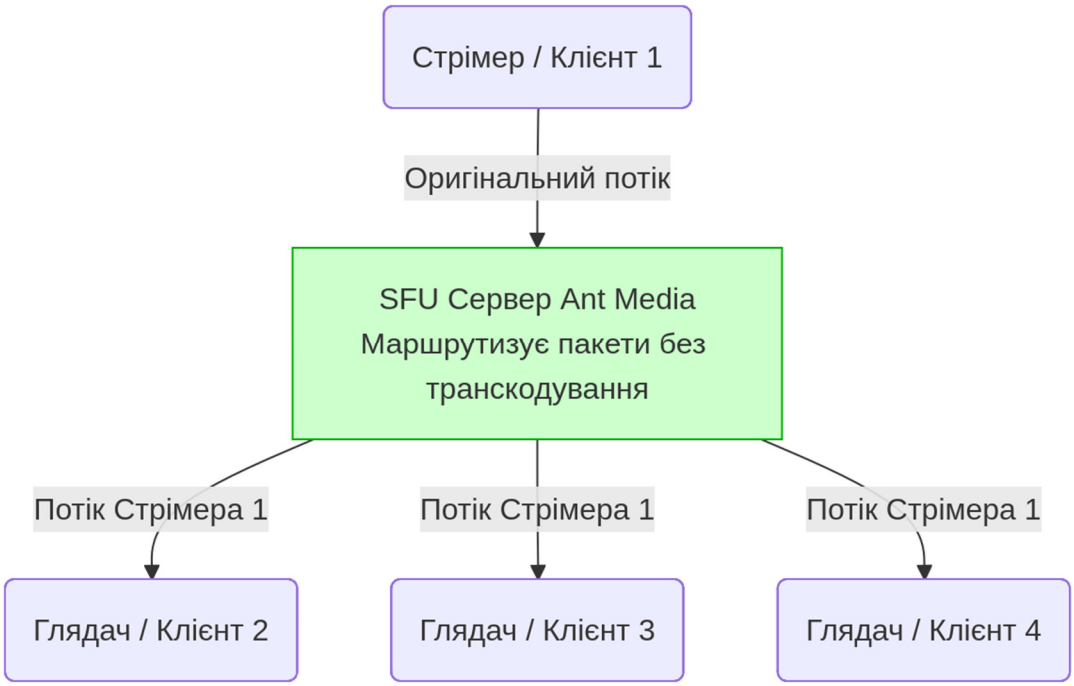


Рисунок 1.3 – Топологія SFU з вибірковою маршрутизацією медіапакетів

Крім маршрутизації RTP-пакетів, взаємодія клієнтського модуля з Ant Media Server вимагає суворого дотримання протоколу сигналізації AMS. На відміну від специфікації самого WebRTC, формат сигнальних повідомлень у кожного медіасервера унікальний. Ant Media Server використовує асинхронний обмін JSON-об'єктами через протокол WebSocket.

Програмний модуль клієнта повинен вміти обробляти та генерувати специфічні команди (actions) для сервера, серед яких ключовими є:

– publish: Команда від клієнта на старт публікації потоку. Супроводжується унікальним ідентифікатором потоку (streamId);
– play: Команда на підключення до існуючого потоку в ролі глядача;
– takeConfiguration: Повідомлення, що містить SDP-дані (Offer або Answer);
– takeCandidate: Повідомлення для передачі знайдених ICE-кандидатів;

– stop: Ініціалізація коректного розриву з'єднання та звільнення ресурсів сервера [10].

Розроблюваний програмний модуль має інкапсулювати всю цю специфічну логіку взаємодії з AMS WebSocket API, абстрагуючи кінцевого розробника застосунку від необхідності ручного парсингу JSON та управління чергами повідомлень.

Створення якісного програмного забезпечення (модуля, бібліотеки або SDK) відрізняється від розробки кінцевого застосунку. Головною метою розробки модуля є забезпечення його повторного використання (reusability), легкості інтеграції та ізоляції складної мережевої логіки від інтерфейсу користувача.

Для досягнення цих цілей архітектура програмного модуля WebRTC-клієнта повинна базуватися на класичних патернах об'єктно-орієнтованого проектування (GoF):

1. Патерн Facade (Фасад): WebRTC API є надзвичайно складним і низькорівневим. Розробнику кінцевого застосунку не потрібно знати про існування об'єктів RTCPeerConnection, RTCIceCandidate або про стан WebSocket-з'єднання. Модуль повинен надавати простий та інтуїтивно зрозумілий клас-фасад (наприклад, AntMediaClient), який експонує лише базові методи: init(), startPublishing(streamId), startPlaying(streamId), stop(). Вся оркестрація внутрішніх процесів прихована за цим фасадом [11].

2. Патерн Observer (Спостерігач) / Event-Driven Architecture: Оскільки процеси сигналізації та встановлення мережевого з'єднання є глибоко асинхронними, модуль не може повертати результати синхронно. Замість цього архітектура повинна використовувати механізм зворотних викликів (Callbacks) або систему делегатів. Модуль генерує події, на які може підписатися хост-застосунок: onConnectionEstablished, onStreamStarted, onIceConnectionFailed, onError. Це дозволяє реактивно оновлювати користувацький інтерфейс [12].

3. Патерн State (Стан): Життєвий цикл WebRTC-з'єднання має багато станів (New, Checking, Connected, Disconnected, Failed). Модуль повинен мати внутрішній конечний автомат (State Machine), який керує переходами між цими станами,

унеможливлюючи виклик методів у невідповідний момент (наприклад, спробу відправити SDP Offer до встановлення WebSocket з'єднання) [13].

Критично важливим аспектом функціонування WebRTC-клієнта в умовах нестабільних бездротових мереж (зокрема 4G/LTE) є здатність системи динамічно адаптуватися до змін пропускної здатності каналу. Хоча протокол RTP відповідає за доставку медіа, управління якістю обслуговування (QoS) покладається на протокол RTCP, який забезпечує постійний обмін телеметричними даними між клієнтом та SFU-сервером.

Алгоритми адаптації базуються на аналізі звітів відправника (Sender Reports, SR) та звітів одержувача (Receiver Reports, RR). У разі фіксації втрати мережевих пакетів клієнтський модуль генерує спеціальні керуючі повідомлення:

– NACK (Negative Acknowledgment): Запит на повторну передачу конкретного втраченого RTP-пакета. Цей механізм ефективний при поодиноких втратах і невеликій мережевій затримці;

– PLI (Picture Loss Indication) / FIR (Full Intra Request): Застосовується при масовій втраті пакетів або пошкодженні опорного кадру. Клієнт сигналізує медіасерверу (або віддаленому піру через SFU) про необхідність термінової генерації та відправки нового ключового кадру (I-frame) для відновлення цілісності відеопотоку [14].

Крім реактивних механізмів, клієнтський модуль спільно з Ant Media Server може використовувати проактивну технологію Simulcast. На відміну від стандартної передачі одного відеопотоку, Simulcast зобов'язує сторону, що публікує відео (Publisher), кодувати та відправляти на сервер одночасно три незалежні потоки з різною роздільною здатністю та бітрейтом (наприклад, 1080p, 720p та 360p). SFU-сервер аналізує пропускну здатність кожного глядача (Subscriber) і динамічно, «на льоту», перемикає трансляцію на відповідний потік без необхідності транскодування. Це дозволяє гарантувати безперебійне відтворення відео навіть для клієнтів зі слабким інтернет-з'єднанням, не погіршуючи при цьому якість для інших учасників.

До початку етапу сигналізації програмний модуль повинен отримати доступ до апаратних засобів кінцевого пристрою (камери та мікрофона). В екосистемі

WebRTC цей процес регламентується специфікацією Media Capture and Streams API.

Отримання локального медіапотоку (LocalMediaStream) вимагає формування об'єкта обмежень (MediaStreamConstraints). Цей об'єкт дозволяє розробнику точно конфігурувати параметри захоплення. Для відеопотоку це включає визначення цільової, мінімальної та максимальної роздільної здатності, бажаної частоти кадрів (FPS) та вибір конкретної камери (фронтальна чи основна). Для аудіопотоку API надає доступ до вбудованих алгоритмів цифрової обробки сигналів (DSP), таких як апаратне або програмне придушення луни (Echo Cancellation), автоматичне регулювання підсилення (Auto Gain Control) та активне шумозаглушення (Noise Suppression). Правильна конфігурація цих параметрів безпосередньо впливає на навантаження центрального процесора мобільного пристрою та кінцеву якість комунікації.

Зважаючи на складність внутрішньої оркестрації WebRTC, вибір інструментарію для розробки клієнтського модуля відіграє ключову роль. Розробка нативних рішень окремо для кожної платформи (Swift для iOS, Kotlin для Android) значно збільшує трудовитрати та ускладнює підтримку ідентичної поведінки кінцевих автоматів (State Machines) на різних пристроях.

Ефективним архітектурним підходом є використання кросплатформового фреймворку Flutter, який дозволяє скомпілювати єдину кодову базу у високопродуктивний машинний код для мобільних, настільних та веб-платформ. Інтеграція WebRTC у середовище Flutter вимагає надійного механізму управління станом, оскільки мережеві події (отримання ICE-кандидатів, зміна статусу з'єднання, надходження нових медіатреків) відбуваються асинхронно і непередбачувано.

Для реалізації патернів Observer та State, згаданих раніше, доцільно застосувати реактивний підхід до управління станом на базі бібліотеки Riverpod. Використання Riverpod дозволяє ізолювати складну бізнес-логіку WebRTC-клієнта (встановлення з'єднання, генерацію SDP) у незалежні провайдери (Providers). Це гарантує, що життєвий цикл з'єднання не буде жорстко прив'язаний до життєвого циклу елементів інтерфейсу (віджетів). Крім того, такий підхід забезпечує автоматичне, потокобезпечне оновлення користувацького інтерфейсу виключно в ті

моменти, коли внутрішній стан WebRTC-модуля (наприклад, зміна статусу з Checking на Connected) дійсно змінюється, що оптимізує використання оперативної пам'яті та ресурсів процесора.

Інтеграція WebRTC у мобільне середовище також вимагає врахування життєвого циклу самого застосунку (App Lifecycle) [20]. Мобільні операційні системи жорстко обмежують доступ до апаратних засобів (камери та мікрофона), коли застосунок переходить у фоновий режим. Внутрішній кінцевий автомат (State Machine) програмного модуля повинен своєчасно реагувати на ці системні переривання, призупиняючи відправку відеокадрів або тимчасово вимикаючи відеотрек, щоб уникнути фатальних помилок ОС та розриву WebSocket-з'єднання [21].

Формулювання завдання на розробку модуля З огляду на проведений аналіз, у рамках виконання практичної частини дипломної роботи ставиться завдання спроектувати та розробити програмний модуль WebRTC-клієнта. Модуль повинен бути реалізований як автономна бібліотека (наприклад, пакет для мобільного фреймворку або веб-застосунків), яка реалізує наступний функціонал:

- автоматичне встановлення та підтримка WebSocket-з'єднання з Ant Media Server;
- інкапсуляція процесу SDP-переговорів (генерація та обробка Offer/Answer) відповідно до логіки AMS;
- збір, упаковка та відправка ICE-кандидатів;
- керування апаратними пристроями мультимедіа (ініціалізація мікрофона та камери, управління роздільною здатністю та частотою кадрів);
- реалізація механізмів обробки мережевих збоїв та спроб перепідключення (Reconnection logic) у разі тимчасової втрати інтернет-з'єднання клієнтом.

2 АНАЛІЗ ТА ПРОЕКТУВАННЯ АРХІТЕКТУРИ ПРОГРАМНОГО МОДУЛЯ WEBRTC-КЛІЄНТА

2.1 Аналіз вимог та обґрунтування вибору технологічного стеку

Перехід від теоретичного обґрунтування до стадії проектування вимагає чіткого визначення функціональних та нефункціональних вимог до створюваного продукту. Оскільки метою дослідження є розробка не кінцевого користувацького застосунку (User Application), а системного програмного модуля (Software Development Kit, SDK), до його архітектури висуваються специфічні інженерні вимоги [19].

На етапі проектування програмного модуля WebRTC-клієнта особлива увага приділяється формуванню нефункціональних вимог, які визначають якісні характеристики системи та умови її стабільного функціонування в розподіленому мережевому середовищі. На основі аналізу предметної області та специфіки інтерактивних медіа-сервісів, автором було виділено та систематизовано наступні ключові нефункціональні вимоги:

1. Модульність та інкапсуляція. Архітектура модуля має базуватися на принципах приховування інформації, де низькорівнева складність мережевих протоколів (Signaling, SDP negotiation, ICE candidates) повністю ізольована від зовнішнього інтерфейсу. Це дозволяє розробнику хост-застосунку взаємодіяти з високорівневими абстракціями, що знижує ймовірність виникнення помилок при інтеграції та спрощує подальше супроводження програмного коду.

2. Кросплатформна сумісність. Програмне рішення повинно демонструвати ідентичну логіку роботи та стабільність показників незалежно від цільової операційної системи (зокрема Android та iOS). Забезпечення уніфікованого API для різних мобільних платформ дозволяє скоротити витрати на розробку та гарантувати однорідність користувацького досвіду в мультиплатформних екосистемах.

3. Обчислювальна ефективність та продуктивність. Враховуючи ресурсомісткість процесів обробки відеопотоків у реальному часі, модуль має бути оптимізований для мінімізації навантаження на центральний процесор (CPU) та

оперативну пам'ять пристрою. Висока продуктивність досягається шляхом використання асинхронних моделей обробки подій та ефективного управління буферами медіаданих, що є критично важливим для запобігання перегріву мобільних пристроїв та надмірного споживання заряду акумулятора.

4. Адаптивна відмовостійкість. Модуль повинен володіти вбудованими механізмами автономного відновлення сесії у випадку деградації мережевого каналу або зміни мережевої топології (наприклад, під час вертикального хендоверу — переходу між Wi-Fi та 4G/LTE мережами). Програмна логіка має забезпечувати автоматичне перепідключення та відновлення медіапотоку без необхідності втручання користувача, що безпосередньо впливає на показник надійності системи.

Крім нефункціональних вимог, етап системного аналізу вимагає чіткого визначення функціональних можливостей розроблюваного модуля, які відображають сценарії його використання в реальних умовах. Базові функціональні вимоги до програмного модуля включають:

1. Автоматизоване встановлення та розірвання захищеного сигнального з'єднання (WebSocket) з медіасервером.

2. Захоплення аудіо- та відеоданих із системних апаратних пристроїв кінцевого вузла з можливістю застосування динамічних конфігурацій (роздільна здатність, частота кадрів).

3. Ініціалізація трансляції (публікації) локального медіапотоку на сервер.

4. Отримання та відтворення віддаленого медіапотоку в ролі глядача.

5. Динамічне управління станом медіаданих (м'яке вимкнення мікрофона/камери, перемикання між фронтальною та основною камерами без розриву сесії).

Для візуалізації цих вимог та відображення меж системи було спроектовано діаграму взаємодії користувача (Use Case Diagram), яку наведено на рисунку 2.1. У межах спроектованої моделі виділено двох первинних акторів (Primary Actors), які взаємодіють з модулем через інтерфейс хост-застосунку: «Стрімер» (Publisher), що є джерелом медіаданих, та «Глядач» (Subscriber), що є приймачем. Вторинним актором (Secondary Actor) виступає безпосередньо «Ant Media Server», який забезпечує підтримку інфраструктури передачі даних. Базові прецеденти

«Ініціалізація з'єднання» та «Завершення сесії» є спільними для обох типів користувачів. Натомість прецеденти трансляції та відтворення є строго розділеними. Специфічний функціонал управління апаратними пристроями реалізовано через відношення розширення (<<extend>>), оскільки ці дії є необов'язковими сценаріями, що доповнюють базовий процес публікації медіапотоку.

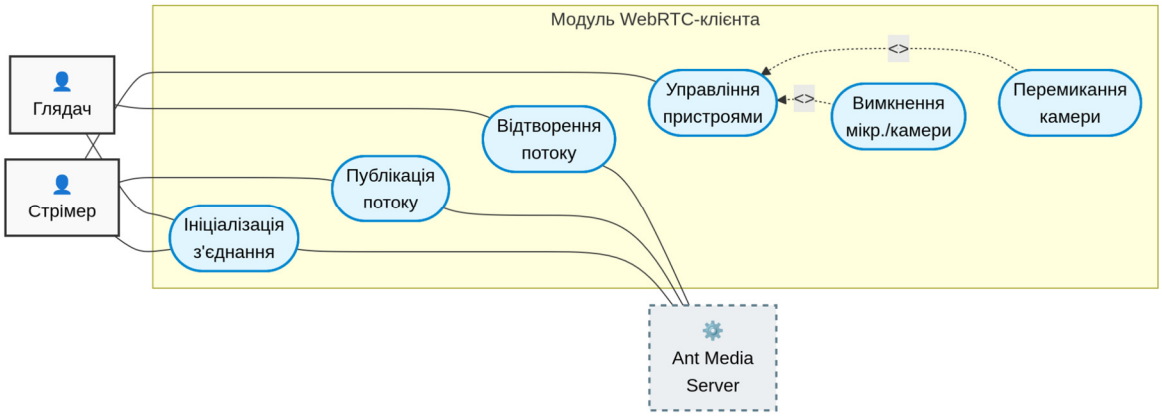


Рисунок 2.1 – Діаграма взаємодії користувача (Use Case) програмного модуля

З огляду на зазначені вимоги, першочерговим завданням стало обґрунтування вибору мови програмування та технологічного стеку. Традиційний підхід передбачає розробку нативних бібліотек (C++ / Java для Android, Swift/Objective-C для iOS), однак це призводить до дублювання кодової бази та ускладнює підтримку модуля. Використання веб-технологій (JavaScript у середовищі React Native) супроводжується проблемою «моста» (JavaScript Bridge), що створює так зване «пляшкове горло» при передачі великих масивів медіаданих між нативною частиною ОС та бізнес-логікою.

Для проектування модуля було обрано сучасну об'єктно-орієнтовану мову програмування Dart у контексті екосистеми фреймворку Flutter. Вибір мови Dart обґрунтовується такими фундаментальними архітектурними перевагами:

– компіляція Ahead-of-Time (AOT): код мовою Dart компілюється безпосередньо у машинний код цільової архітектури (ARM або x86), що забезпечує продуктивність, наближену до мов C/C++;

– оптимізована асинхронна модель: робота WebRTC-клієнта є глибоко асинхронною (очікування мережевих пакетів, обробка потоків з камери). Dart використовує цикл подій (Event Loop) та неблокуючий ввід/вивід на базі об'єктів Future та Stream, що дозволяє обробляти тисячі подій без створення важких потоків ОС (threads);

– механізм ізолятів (Isolates): для виконання важких обчислювальних задач (наприклад, обробки великих JSON-відповідей від сервера) Dart дозволяє створювати ізоляти — незалежні потоки виконання з власною пам'яттю, що унеможливлює блокування головного потоку застосунку (Main Thread);

– механізм Foreign Function Interface (FFI): дозволяє Dart-коду напряму, без витрат на серіалізацію, викликати функції з C++ бібліотек. Саме через цей механізм модуль взаємодіятиме з базовою бібліотекою libwebrtc від Google [15; 20].

Отже, стек Dart/Flutter забезпечує ідеальний баланс між продуктивністю нативного коду та швидкістю кросплатформної розробки, що робить його оптимальним вибором для проектування стрімінгового SDK [14].

Для оцінки загальної ефективності роботи Модуля в реальних умовах було запропоновано використовувати розрахунок наскрізної затримки (Glass-to-Glass Latency). Загальний час затримки при передачі відеокадру описується такою формулою:

$$L = t_{cap} + t_{enc} + t_{net} + t_{proc} + t_{dec} + t_{play} \quad (2.1)$$

де L — загальна наскрізна затримка, мс;

t_{cap} — час захоплення кадру апаратною камерою, мс;

t_{enc}— час кодування кадру обраним кодеком (наприклад, VP8), мс;

t_{net}— час передачі пакетів через мережу (RTP-транспорт), мс;

t_{proc}— час обробки пакетів сервером (SFU-маршрутизація), мс;

t_{dec}— час декодування на стороні клієнта-глядача, мс;

t_{play}— час буферизації та рендерингу кадру на екран, мс.

Проектований програмний модуль безпосередньо впливає на показники t_cap, t_enc та частково t_net (через алгоритми збору ICE-кандидатів). Тому оптимізація цих складових є пріоритетним завданням на етапі написання коду.

2.2 Проектування архітектурної моделі модуля та вибір патернів

Проектування архітектури програмного модуля розпочинається з визначення його місця в глобальній інфраструктурі потокової передачі даних. З цією метою було розроблено структурну схему розгортання системи, яка візуалізує фізичні та логічні вузли мережі, а також протоколи взаємодії між ними (Рисунок 2.1).

Глобальна топологія включає три основні вузли: кінцевий мобільний пристрій із розробленим SDK, центральний медіасервер Ant Media Server (AMS) та допоміжні сервери STUN/TURN. Сигналізація (обмін SDP-дескрипторами та ICE-кандидатами) між клієнтом та AMS здійснюється через захищений канал WebSocket Secure (WSS). Для подолання мережевих обмежень NAT клієнтський модуль звертається до серверів STUN/TURN протоколом UDP. Після успішного встановлення маршруту, мультимедійний трафік передається безпосередньо до AMS з використанням протоколів SRTP/SRTCP, де сервер виконує роль SFU-маршрутизатора для інших учасників.

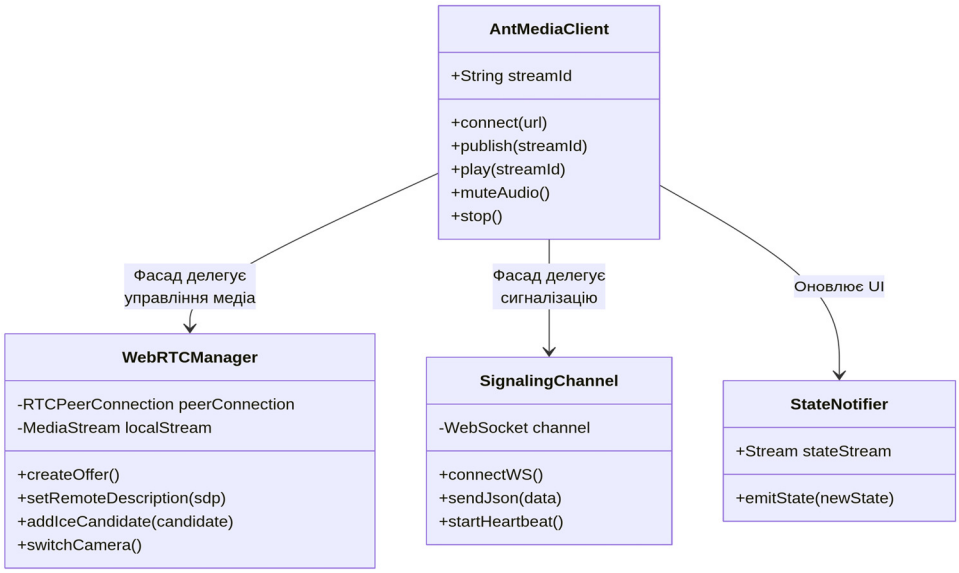


Рисунок 2.2 – Структурна схема взаємодії WebRTC-клієнта з серверною інфраструктурою

На рівні внутрішньої організації самого клієнтського SDK, систему поділено на ізольовані функціональні блоки. Для відображення логіки взаємодії цих блоків було спроектовано функціональну схему модуля, подану на рисунку 2.3. Функціональна схема ілюструє потік керуючих команд та медіаданих всередині

SDK. Точкою взаємодії з хост-застосунком виступає публічний API (Фасад), який передає команди до «Модуля управління станами». Цей модуль діє як оркестратор: він ініціалізує роботу «Модуля сигналізації» для зв'язку з сервером, керує «Модулем захоплення медіа» для отримання доступу до апаратних пристроїв (камери та мікрофона), і синхронізує їхні дані з базовим «Модулем WebRTC Транспорту» (нативний рушій, що містить об'єкт RTCPeerConnection). Така декомпозиція гарантує, що жоден модуль не виконує невластивих йому функцій.

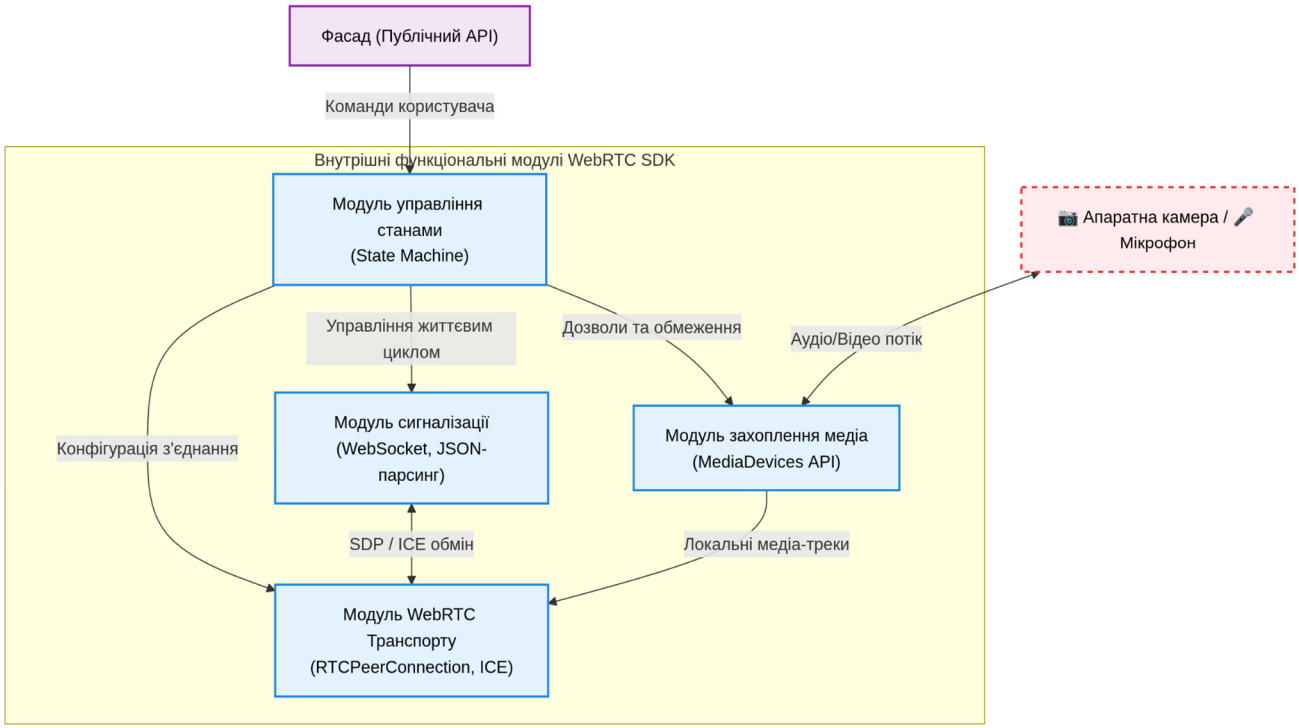


Рисунок 2.3 – Функціональна схема внутрішніх компонентів програмного модуля

Для імплементації запропонованої трирівневої архітектури та забезпечення високого рівня гнучкості, масштабованості й зручності супроводу програмного коду, у процесі розробки застосовано класичні об'єктно-орієнтовані патерни проектування (Design Patterns) [11, 12]. Використання стандартизованих шаблонів (GoF) дозволяє вирішити одразу кілька інженерних завдань: по-перше, абстрагувати вкрай складну низькорівневу бізнес-логіку протоколів WebRTC від розробника кінцевого застосунку; по-друге, мінімізувати жорстку зв'язність (coupling) між системними компонентами; і по-третє, надати інтуїтивно зрозумілий та безпечний прикладний програмний інтерфейс (API). Зважаючи на специфіку

мережевої взаємодії та глибоко асинхронну природу потокової передачі медіаданих, базове архітектурне рішення побудовано на синергії структурних та поведінкових патернів.

Основним структурним патерном є Facade (Фасад). API WebRTC є надзвичайно складним і вимагає глибокого розуміння життєвого циклу SDP та ICE-кандидатів. Патерн Facade приховує цю складність. Спроектовано головний клас AntMediaClient, який виступає єдиною точкою входу (Entry Point) до модуля. Він надає лаконічний набір методів: init(), publish(String streamId), play(String streamId), muteAudio() та stop(). Будь-яка внутрішня взаємодія між компонентами сигналізації та медіа-транспорту відбувається всередині модуля і не експонується назовні.

Другим критично важливим патерном є поведінковий патерн Observer (Спостерігач). Зважаючи на асинхронну природу мережевого зв'язку, хост-застосунок повинен миттєво реагувати на зміни стану (наприклад, перемикати інтерфейс з режиму «Завантаження» на «Трансляція»). Клас AntMediaClient реалізує цей патерн через надання доступу до потоку подій Stream<SignalingState>. Коли внутрішній стан модуля змінюється, він генерує нову подію в потік, а всі підписані на нього компоненти інтерфейсу автоматично оновлюються, що забезпечує реактивний підхід до управління станом системи.

Для гарантування стабільності та унеможливлення виконання некоректних операцій внутрішня логіка модуля спроектована на базі моделі життєвого циклу з'єднання [21]. Відповідно до цієї моделі, підсистема управління може перебувати виключно в одному з дозволених станів. У таблиці 2.1 наведено спроектовану матрицю переходів між етапами життєвого циклу модуля.

На рисунку 2.4 графічно відображено модель життєвого циклу з'єднання у вигляді UML-діаграми станів. Початковим етапом роботи модуля є стан Idle, з якого ініціалізація процесу викликом методу connect() переводить систему в стан очікування підключення — Connecting. На цьому етапі можливі два варіанти розвитку подій: успішне встановлення WebSocket-з'єднання (WS відкрито) переводить систему у стан Connected, тоді як виникнення системної помилки або таймауту (Таймаут WS) ініціює перехід у стан Error. Важливою архітектурною особливістю є наявність циклу відновлення: зі стану Error система автоматично

повертається до спроби підключення (Connecting), використовуючи алгоритм експоненційної затримки (Exponential Backoff).

Таблиця 2.1 — Матриця станів та переходів розроблюваного модуля

Поточний стан (Current State)	Подія (Trigger / Action)	Наступний стан (Next State)	Опис реакції системи
Idle (Очікування)	Виклик connect()	Connecting	Ініціалізація WebSocket-з'єднання з сервером.
Connecting	Успішне підключення	Connected	З'єднання встановлено, готовність до обміну SDP.
Connecting	Помилка сокета / Таймаут	Error	Запуск алгоритму експоненційної затримки (Exponential Backoff).
Connected	Виклик publish(streamId)	Publishing	Генерація SDP Offer та відправка його на сервер.
Publishing	Успішний обмін ICE	Streaming	Встановлення P2P з'єднання, початок передачі медіа.
Streaming	Виклик stop()	Closed	Звільнення ресурсів камери, закриття RTCPeerConnection.

Після успішної ініціалізації сигналізації (Connected), життєвий цикл розгалужується залежно від обраної ролі клієнта. Публікація власного медіапотоку через виклик publish(streamId) активує стан Publishing, а запит на перегляд існуючої трансляції через play(streamId) — стан Playing. Успішне завершення узгодження SDP та обміну ICE-кандидатами (ICE підключено) об'єднує ці гілки у стані активної трансляції мультимедійних даних — Streaming. Завершення сесії ініціюється викликом методу stop(), що переводить модуль у фінальний стан Closed, після чого відбувається закриття з'єднань, вивільнення апаратних ресурсів та завершення життєвого циклу.

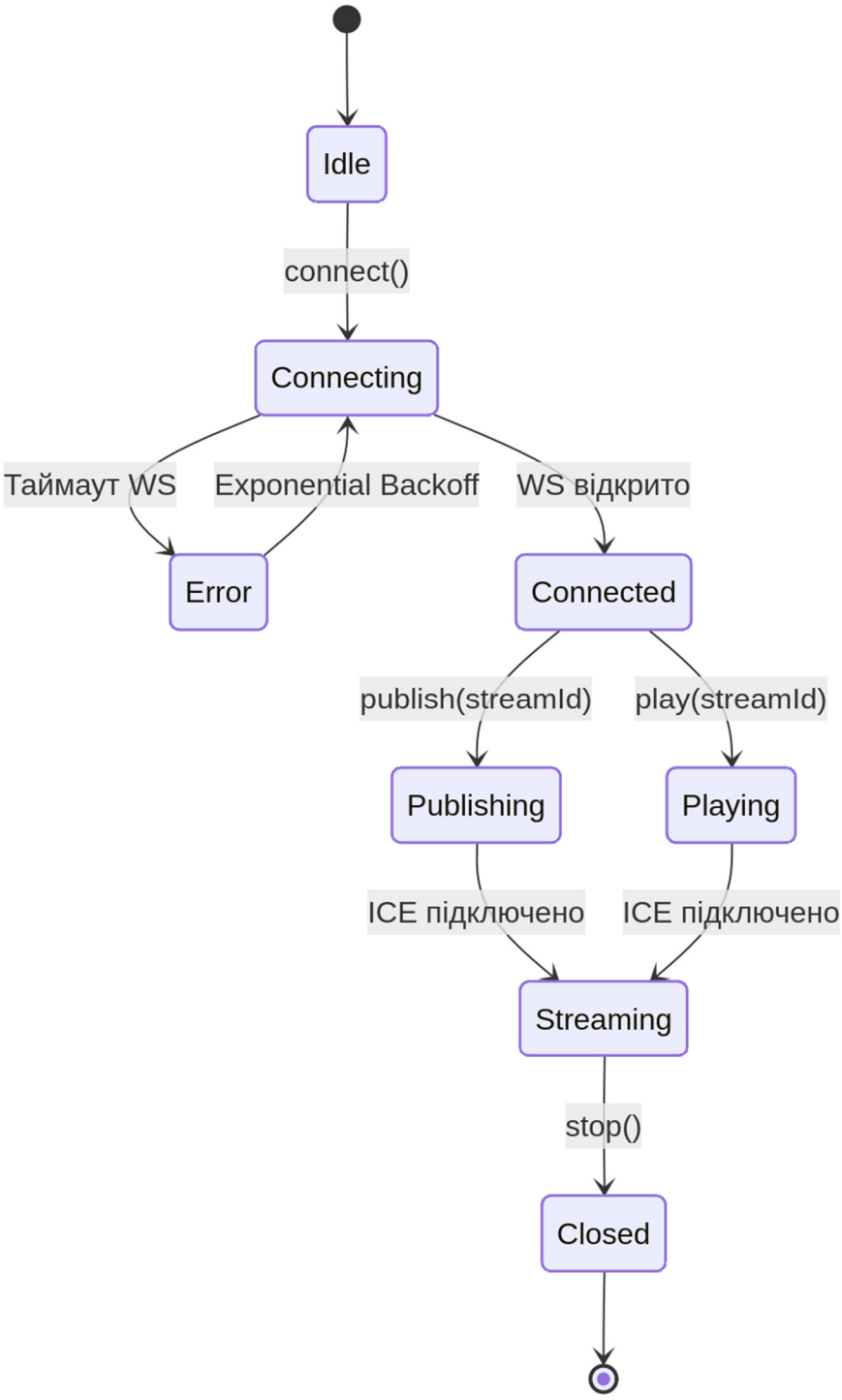


Рисунок 2.4 – Граф переходів моделі життєвого циклу розробленого модуля

Впровадження строгої моделі життєвого циклу унеможливлює виникнення станів гонитви (Race Conditions). Наприклад, якщо розробник спробує викликати метод muteAudio() у стані Connecting, система згенерує внутрішнє виключення (StateError), оскільки медіа-треки ще не ініціалізовані. Такий підхід забезпечує абсолютну передбачуваність роботи SDK.

Важливим аспектом архітектури є інтеграція розробленого модуля з екосистемою управління станом фреймворку Flutter. Для забезпечення ефективної комунікації між внутрішнім скінченним автоматом модуля та користувацьким інтерфейсом (UI) хост-застосунку застосовується бібліотека Riverpod. Замість традиційних колбеків (callbacks), стан підключення (SignalingState) інкапсулюється всередині StateNotifier або Notifier провайдера. Це дозволяє користувацькому інтерфейсу застосунку реагувати на зміни стану (наприклад, перехід від "Connecting" до "Connected") реактивно, забезпечуючи автоматичне та безпечне оновлення віджетів без ризику витоків пам'яті під час перемикання екранів [20].

2.3 Алгоритмічне забезпечення процесу сигналізації з Ant Media Server

Сигналізація (Signaling) є фундаментальним процесом у WebRTC, під час якого пристрої узгоджують параметри майбутньої медіа-сесії до встановлення прямого з'єднання [4]. Оскільки специфікація WebRTC навмисно не стандартизує транспорт сигналізації, кожен медіасервер має власну реалізацію. Ant Media Server (AMS) використовує обмін структурованими JSON-об'єктами через протокол WebSocket [10].

Для коректної роботи модуля запропоновано алгоритми публікації (Publish) та відтворення (Play) відеопотоку. Ці алгоритми повинні строго дотримуватися хореографії, заданої архітектурою AMS.

Алгоритм публікації потоку (Publish Flow) включає такі послідовні кроки:

1. Ініціалізація (Handshake): Модуль (через компонент SignalingChannel) встановлює TCP-з'єднання з ендпоінтом wss://{domain}/{app}/websocket.

2. Запит на трансляцію: Модуль серіалізує та відправляє JSON-повідомлення про намір розпочати публікацію:

```
JSON{
  "command": "publish",
  "streamId": "stream_123",
  "video": true,
  "audio": true
}
```

3. Очікування дозволу: Сервер перевіряє права доступу (наприклад, валідність JWT токена) і, за умови відсутності помилок, повертає команду "command": "start".

4. Створення SDP Offer: Отримавши команду start, компонент WebRTCManager звертається до нативного рушія для генерації об'єкта SDP Offer. Цей документ містить опис підтримуваних пристроєм кодеків (наприклад, VP8, H.264, Opus) та параметрів шифрування [24]. Сгенерований опис зберігається локально (setLocalDescription) та відправляється на сервер:

```
JSON{
  "command": "takeConfiguration",
  "streamId": "stream_123",
  "type": "offer",
  "sdp": "v=0\ro=- 480111664188 2 IN IP4 127.0.0.1\r..."
}
```

5. Отримання SDP Answer: Сервер AMS аналізує пропозицію, узгоджує її з власними налаштуваннями і повертає SDP Answer. Модуль застосовує його як віддалений опис (setRemoteDescription), завершуючи логічне узгодження.

6. Збір ICE-кандидатів (Trickle ICE): Паралельно з кроком 5, мережевий стек пристрою починає виявляти свої IP-адреси за допомогою STUN/TURN серверів. Кожен знайдений кандидат миттєво (без очікування збору всіх інших) відправляється на сервер спеціальною командою takeCandidate. Сервер аналогічно надсилає свої кандидати клієнту.

7. Завершення: Після перевірки маршрутів (Connectivity Checks) протоколом ICE, WebSocket-сигналізація призупиняється, і починається безпосередня передача UDP-пакетів з медіаданими (стан Streaming).

Проектування алгоритму відтворення (Play Flow) здійснюється за дзеркальним принципом: клієнт відправляє команду play, а процес генерації SDP Offer ініціюється не клієнтом, а самим сервером AMS, що відповідає топології SFU (Selective Forwarding Unit).

2.4 Проектування механізмів управління медіа-потоками та апаратними ресурсами

Окрім мережевої взаємодії, відповідальністю програмного модуля є ефективне управління апаратними ресурсами пристрою (камерою та мікрофоном). Доступ до сенсорів проектується з використанням стандартизованого W3C інтерфейсу MediaDevices [16].

Однією із найскладніших проблем мобільної розробки є оптимізація споживання ресурсів, зокрема збереження заряду батареї та уникнення перегрівання процесора при кодуванні відео. З цією метою було запропоновано використання адаптивних обмежень (Media Constraints). Під час виклику методу getUserMedia(), модуль формує динамічний конфігураційний об'єкт. Замість жорсткої фіксації параметрів, модуль запитує ідеальні показники (наприклад, 1280x720, 30 FPS), дозволяючи рушію WebRTC самостійно здійснювати "динамічне зниження роздільної здатності" (зменшення роздільної здатності) у випадку, якщо апаратні потужності смартфона не здатні забезпечити заданий рівень продуктивності [23].

Важливим архітектурним рішенням у розробці Модуля є проектування алгоритму «м'якого вимкнення» медіа (Soft Mute). У типових системах відеозв'язку функція вимкнення мікрофона часто реалізується шляхом повного видалення аудіотреку з об'єкта RTCPeerConnection. Це призводить до необхідності створення нового SDP Offer і повторного проведення переговорів (Renegotiation) із сервером. Такий підхід є вкрай неефективним і зумовлює затримки тривалістю 1-2 секунди.

Алгоритм «Soft Mute», спроектований для даного модуля, діє на рівні окремих треків без втручання в мережевий транспорт. Коли користувач ініціює вимкнення мікрофона (виклик методу фасаду muteAudio()), система ідентифікує локальний AudioTrack всередині об'єкта MediaStream і змінює його властивість enabled на false. У результаті рушій WebRTC продовжує відправляти RTP-пакети на сервер, зберігаючи з'єднання активним, проте корисне навантаження (payload) цих пакетів містить «тишу» (порожні байти). Цей підхід забезпечує миттєву реакцію системи (Zero Latency) на дії користувача. Аналогічна концепція

розроблена для відключення камери («Soft Video Disable»), при якій замість реальних кадрів на сервер відправляються чорні фрейми.

Для функції перемикання між фронтальною та основною камерами мобільного пристрою спроектовано виклик нативного методу switchCamera(). Цей метод на низькому (апаратному) рівні перенаправляє потік пікселів від іншого сенсора в уже існуючий відеотрек, уникаючи розриву з'єднання та переузгодження SDP.

На етапі проектування медіа-транспорту також закладено архітектурні можливості для моніторингу якості обслуговування (QoS). Модуль використовує стандартизований метод getStats() об'єкта RTCPeerConnection для періодичного збору телеметричних даних на базі протоколу RTCP (відсоток втрачених пакетів, рівень джиттеру, затримка RTT) [4]. Зібрана статистика обробляється внутрішнім аналізатором і може передаватися до хост-застосунку через патерн Спостерігач. Це дозволяє системі не лише динамічно знижувати роздільну здатність відео за допомогою механізму адаптивного бітрейту (ABR), але й інформувати користувача про погіршення умов зв'язку (наприклад, через попередження "Нестабільне з'єднання") [18, 23].

2.5 Проектування алгоритмів відмовостійкості та обробки помилок

Робота програмного забезпечення у мобільних бездротових мережах характеризується високою ймовірністю втрати пакетів, зміни IP-адрес (при роумінгу між базовими станціями) та тимчасовими обривами зв'язку. Програмний модуль комерційного рівня повинен володіти внутрішніми алгоритмами відмовостійкості (Fault Tolerance) для приховування цих мережевих аномалій від кінцевого користувача [17].

Першим рівнем спроектованої системи відмовостійкості є контроль активності WebSocket-з'єднання. Оскільки балансувальники навантаження (Load Balancers) інтернет-провайдерів часто примусово обривають неактивні TCP-сокети, у Модулі реалізовано механізм «Heartbeat». Проектується таймер, який кожні 10 секунд відправляє на сервер службове повідомлення (Ping). У випадку

відсутності відповіді (Pong) протягом заданого проміжку часу, стан життєвого циклу переводиться в Error, і модуль ініціює процедуру автоматичного перепідключення.

Процедура перепідключення (Reconnection) спроектована з використанням математичної моделі експоненційної затримки (Exponential Backoff). На відміну від лінійного перепідключення (наприклад, спроби кожну секунду), яке може призвести до DDoS-атаки на власний сервер під час масового мережевого збою, експоненціальна затримка поступово збільшує інтервал між спробами [18; 23].

Розрахунок часу очікування наступної спроби підключення здійснюється за формулою:

$$T_n = \min(T_{max}, T_{base} \cdot 2^n) + t_{rand} \quad (2.2)$$

де Tn — час очікування для n-ї спроби перепідключення, с;

Tmax — верхня межа часу затримки (у проекті встановлена на рівні 30 с);

Tbase — базовий (початковий) час затримки (встановлено 1 с);

n — лічильник кількості невдалих спроб;

trand — випадкове значення (Jitter) у діапазоні від 0 до 1000 мс, що додається для десинхронізації одночасних запитів від різних клієнтів.

Другим, і найважливішим, рівнем відмовостійкості є механізм ICE Restart. Цей механізм спроектовано для вирішення проблеми зміни мережевої топології (наприклад, коли користувач покидає приміщення і смартфон перемикається з мережі Wi-Fi на 4G). У такій ситуації WebSocket-з'єднання може бути швидко відновлене, проте старі ICE-кандидати (IP-адреси), по яких передавалося відео, стають недійсними, що призводить до зависання зображення.

Відповідно до розробленого алгоритму (рисунок 2.3), модуль безперервно здійснює моніторинг стану iceConnectionState. При виявленні переходу цього стану у failed, компонент WebRTCManager автономно ініціює процедуру ICE Restart. Для цього генерується новий об'єкт SDP Offer із встановленим системним прапорцем iceRestart: true [24]. Цей оновлений Offer відправляється на Ant Media Server. Сервер ідентифікує прапорець, генерує нові облікові дані для STUN/TURN серверів, і сторони блискавично збирають нові ICE-кандидати для актуальних IP-

адрес. В результаті, передача відео відновлюється безшовно, без необхідності втручання користувача чи перезавантаження інтерфейсу застосунку.

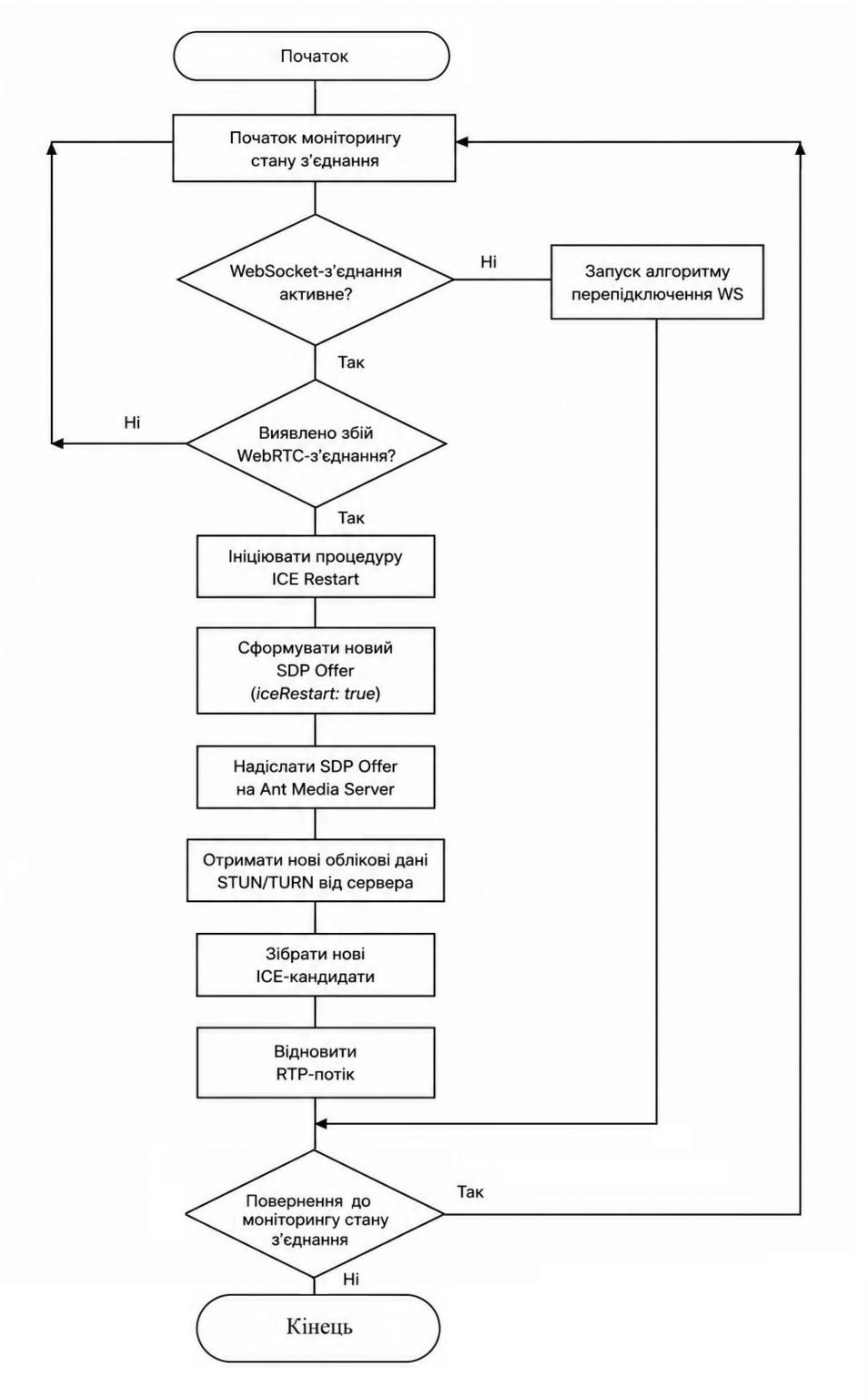


Рисунок 2.3 – Алгоритм обробки мережевих збоїв та автономного відновлення сесії (ICE Restart)

3 ПРАКТИЧНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ПРОГРАМНОГО МОДУЛЯ WEBRTC-КЛІЄНТА

3.1 Розгортання середовища розробки та імплементація базового функціоналу модуля

Практична реалізація спроектованої архітектури розпочалася зі створення автономного програмного пакета (SDK) у середовищі фреймворку Flutter. Ініціалізація структури модуля була виконана за допомогою команди flutter create --template=package, що дозволило відокремити бізнес-логіку клієнта від користувацького інтерфейсу кінцевого застосунку.

Основним транспортним механізмом для медіаданих було обрано відкриту бібліотеку flutter_webrtc, яка надає Dart-обгортку над нативними C++ компонентами WebRTC від Google. Для забезпечення взаємодії з апаратними сенсорами мобільних пристроїв було налаштовано відповідні дозволи на рівні операційних систем:

– для платформи Android до файлу AndroidManifest.xml було додано дозволи на використання камери, мікрофона та доступу до мережі Інтернет (android.permission.CAMERA, android.permission.RECORD_AUDIO, android.permission.INTERNET);

– для платформи iOS файл Info.plist було доповнено ключами NSCameraUsageDescription та NSMicrophoneUsageDescription із зазначенням політик конфіденційності використання сенсорів.

Базовий функціонал модуля був інкапсульований у класі-фасаді AntMediaClient. Імплементація розпочалася з реалізації рівня сигналізації: за допомогою пакета web_socket_channel було реалізовано асинхронне встановлення TCP-з'єднання з ендпоінтом AMS, що стало фундаментом для подальшого обміну SDP-пакетами (Session Description Protocol) та ICE-кандидатами (Interactive Connectivity Establishment), необхідними для проходження NAT та коректної маршрутизації медіатрафіку.

Відповідно до спроектованої архітектурної моделі, управління станами модуля було реалізовано за допомогою реактивного підходу з використанням

бібліотеки Riverpod. Для інкапсуляції логіки переходів між етапами життєвого циклу з'єднання було розроблено клас SignalingNotifier, який розширює базовий клас StateNotifier. Цей компонент виконує роль єдиного джерела істини (Single Source of Truth) для стану системи.

Такий підхід ізолює мутації стану від зовнішнього втручання та дозволяє користувацькому інтерфейсу хост-застосунку безпечно підписуватися на оновлення через глобальний провайдер signalingProvider. Фрагмент програмного коду, що демонструє реалізацію цього патерна, наведено у лістингу 3.1.

Лістинг 3.1 — Реалізація управління станами модуля за допомогою Riverpod

```
import 'package:flutter_riverpod/flutter_riverpod.dart';
enum SignalingState {
  idle,
  connecting,
  connected,
  error,
  publishing,
  playing,
  streaming,
  closed
}

class SignalingNotifier extends StateNotifier<SignalingState> {
  SignalingNotifier() : super(SignalingState.idle);

  void updateState(SignalingState newState) {
    if (state == SignalingState.closed && newState !=
SignalingState.idle) {
      developer.log('Помилка: Спроба зміни стану закритої
сесії.');
      return;
    }
    state = newState;
  }
  void reset() {
    state = SignalingState.idle;
  }
}

final signalingProvider =
StateNotifierProvider<SignalingNotifier, SignalingState>((ref)
{
  return SignalingNotifier();
});
```

Впровадження даного провайдера всередині класу-фасаду AntMediaClient

дозволило повністю прибрати використання застарілих механізмів Callback-функцій, замінивши їх на передбачуваний потік даних, що суттєво підвищило надійність SDK та спростило його подальшу інтеграцію.

Наступним кроком реалізації базового функціоналу стала підготовка до захоплення та обробки медіапотоків. Після успішного переходу стану в SignalingState.connected, клієнтський модуль ініціює створення об'єкта RTCPeerConnection. Цей базовий інтерфейс WebRTC API бере на себе відповідальність за кодування аудіо- та відеоданих, отриманих через метод navigator.mediaDevices.getUserMedia, а також за їх подальшу передачу до медіасервера.

Для забезпечення відмовостійкості розробленого рішення було також закладено фундамент для обробки виняткових ситуацій. Завдяки реактивній архітектурі, будь-яка непередбачувана втрата WebSocket-з'єднання або тайм-аут відповіді від сервера автоматично переводить систему у стан SignalingState.error. Це делегує відповідальність за реакцію на подію (наприклад, спробу автоматичного перепідключення або показ відповідного UI-сповіщення) на рівень бізнес-логіки хост-застосунку, залишаючи сам SDK максимально абстрагованим та сфокусованим виключно на трансляції.

3.2 Програмна реалізація механізмів відмовостійкості та управління медіа-треками

Найбільш критичним етапом розробки стала програмна імплементація алгоритмів обробки непередбачуваних мережевих станів. Для запобігання перевантаженню сигнального сервера під час масових розривів з'єднань було імплементовано алгоритм експоненціальної затримки (Exponential Backoff). Фрагмент реалізації цього алгоритму мовою Dart наведено в лістингу 3.2.

Лістинг 3.2 — Реалізація механізму Exponential Backoff для перепідключення сокета

```
Future<void> _reconnectWithBackoff(int attempt) async {
```

```
  if (attempt > maxRetries) {

ref.read(signalingProvider.notifier).updateState(SignalingState
.error);
    return;
  }
  final int baseDelay = 1000;
  final int jitter = Random().nextInt(1000);
  final int delayMs = (baseDelay * pow(2, attempt)).toInt() +
jitter;
  final int finalDelay = min(delayMs, 30000);
  await Future.delayed(Duration(milliseconds: finalDelay));
  try {
    await _initializeWebSocket();

ref.read(signalingProvider.notifier).updateState(SignalingState
.connected);
  } catch (e) {
    await _reconnectWithBackoff(attempt + 1);
  }
}
```

Для вирішення проблеми зависання медіапотоку при зміні мережевої топології (наприклад, перехід з Wi-Fi на 4G/LTE), було реалізовано прослуховування подій стану протоколу ICE. При виявленні розриву модуль автоматично ініціює процедуру ICE Restart, формуючи нову пропозицію SDP (лістинг 3.3).

Лістинг 3.3 — Перехоплення втрати мережі та запуск алгоритму ICE Restart

```
_peerConnection?.onIceConnectionState = (RTCIceConnectionState
state) async {
  if (state ==
RTCIceConnectionState.RTCIceConnectionStateFailed ||
      state ==
RTCIceConnectionState.RTCIceConnectionStateDisconnected) {

    RTCSessionDescription offer = await
_peerConnection!.createOffer({
      'offerToReceiveAudio': true,
      'offerToReceiveVideo': true,
      'iceRestart': true,
    });

    await _peerConnection!.setLocalDescription(offer)

    _sendSignalingMessage({
      'command': 'takeConfiguration',
      'streamId': _currentStreamId,
      'type': offer.type,
```

```
            'sdp': offer.sdp,
        });
    }
};
```

Для представлення асинхронної природи процесу відновлення зв'язку, на рисунку 3.1 наведено UML-діаграму послідовності (Sequence Diagram), яка візуалізує виконання програмного алгоритму ICE Restart під час взаємодії розробленого модуля з серверною інфраструктурою. Діаграма демонструє, що процес генерування нової конфігурації (SDP Offer) та переузгодження маршрутів відбувається без розриву основного сигнального з'єднання.

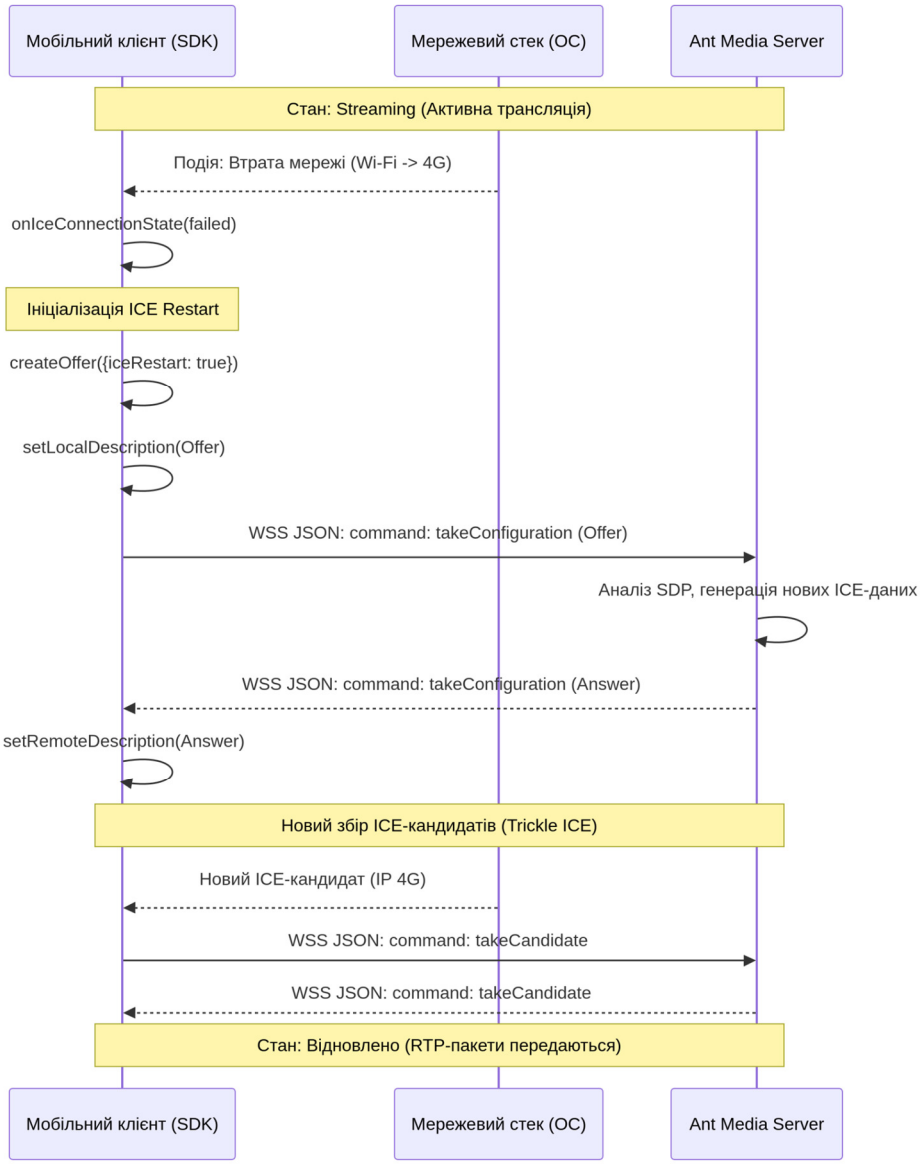


Рисунок 3.1 – UML-діаграма послідовності виконання алгоритму ICE Restart

З метою оптимізації швидкодії інтерфейсу користувача було реалізовано алгоритм «Soft Mute» (м'яке вимкнення). Замість модифікації об'єкта RTCPeerConnection та повторного SDP-узгодження (що викликає затримки), управління медіа здійснюється безпосередньо через властивості треків (лістинг 3.4). Цей підхід зберігає RTP-сесію активною.

Лістинг 3.4 — Реалізація функції Soft Mute

```
void muteAudio(bool mute) {
  if (_localStream != null) {
    for (var track in _localStream!.getAudioTracks()) {
      track.enabled = !mute; // Миттєве вимкнення/увімкнення
передачі звуку
    }
  }
}

void muteVideo(bool mute) {
  if (_localStream != null) {
    for (var track in _localStream!.getVideoTracks()) {
      track.enabled = !mute; // Відправка чорних фреймів
замість реального відео
    }
  }
}
```

Логічним продовженням механізмів управління медіа-треками, що були концептуально закладені в архітектурі модуля (Розділ 2), є практична імплементація збору мережевої телеметрії (Quality of Service). Для того, щоб алгоритми відмовостійкості могли проактивно реагувати на погіршення зв'язку ще до повного розриву ICE-з'єднання, у модулі реалізовано періодичний аналіз статистики RTCPeerConnection.

Програмний метод опитує нативний рушій WebRTC за допомогою виклику getStats() та екстрагує ключові показники: рівень втрати пакетів (Packets Lost) та мережевий джиттер (Jitter). Фрагмент реалізації наведено у лістингу 3.5.

Лістинг 3.5 — Програмна реалізація збору телеметричних даних (QoS)

```
Future<void> _monitorNetworkQuality() async {
  if (_peerConnection == null) return;

  try {
```

```
    final List<StatsReport> stats = await
_peerConnection!.getStats();

    for (var report in stats) {
      if (report.type == 'inbound-rtp' &&
report.values['mediaType'] == 'video') {
        final double packetsLost =
double.parse(report.values['packetsLost'].toString());
        final double jitter =
double.parse(report.values['jitter'].toString());
        if (packetsLost > 50 || jitter > 0.5) {
          developer.log('Попередження: Нестабільне мережеве
з\'єднання!');
        }
      }
    }
  } catch (e) {
    print('Помилка збору статистики: ${e.toString()}');
  }
}
```

3.3 Експериментальне тестування модуля

Під час початкових етапів розробки та тестування мобільного WebRTC-клієнта на реальних пристроях було виявлено проблему надмірного споживання обчислювальних ресурсів центрального процесора (CPU). Для виявлення межі відмовостійкості архітектури було проведено стрес-тестування (Stress Testing). Застосунок було переведено в режим екстремального навантаження: ініційовано захоплення локального відеопотоку у роздільній здатності 4K (3840x2160) при 60 кадрах на секунду (лістинг 3.6).

Лістинг 3.6 — Конфігурація медіаобмежень для стрес-тестування (4K Ultra HD)

```
final mediaConstraints = {
  'audio': true,
  'video': {
    'mandatory': {
      'minWidth': '3840',
      'minHeight': '2160',
      'minFrameRate': '60',
    },
    'facingMode': 'user',
    'optional': [],
  },
};
```

Аналіз телеметрії та профілювання застосунку у середовищі Flutter DevTools під час виконання цього стрес-тесту наочно продемонстрували вузькі місця системи. Головною причиною деградації продуктивності стало використання відеокодека VP8, який встановлюється у WebRTC як формат стиснення за замовчуванням і виконується на програмному рівні (Software Encoding).

Як видно з результатів профілювання потоку рендерингу на вкладці Performance (рисунок 3.2), використання програмного кодека VP8 для обробки відео високої роздільної здатності призводить до критичного перевантаження системи. Стовпчики графіка (UI Jank) набули червоного кольору, що свідчить про значне перевищення еталонного часу побудови кадру (16 мс) та призводить до візуального «гальмування» інтерфейсу.

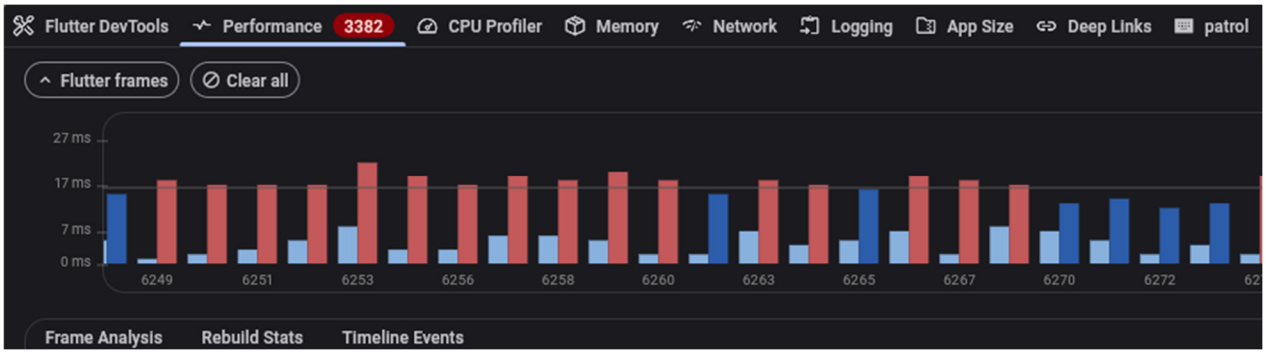


Рисунок 3.2 – Профілювання продуктивності потоку рендерингу (Performance) під час стрес-тестування

Детальний аналіз стека викликів за допомогою інструмента CPU Profiler (рисунок 3.3) підтвердив, що левова частка процесорного часу витрачається на складні математичні обчислення в нативних потоках та обробку графіки. Процесор (SoC) мобільного пристрою фізично не встигає розподіляти ресурси між кодуванням масивного 4K-відео формату VP8 та підтримкою стабільної частоти кадрів користувацького інтерфейсу.

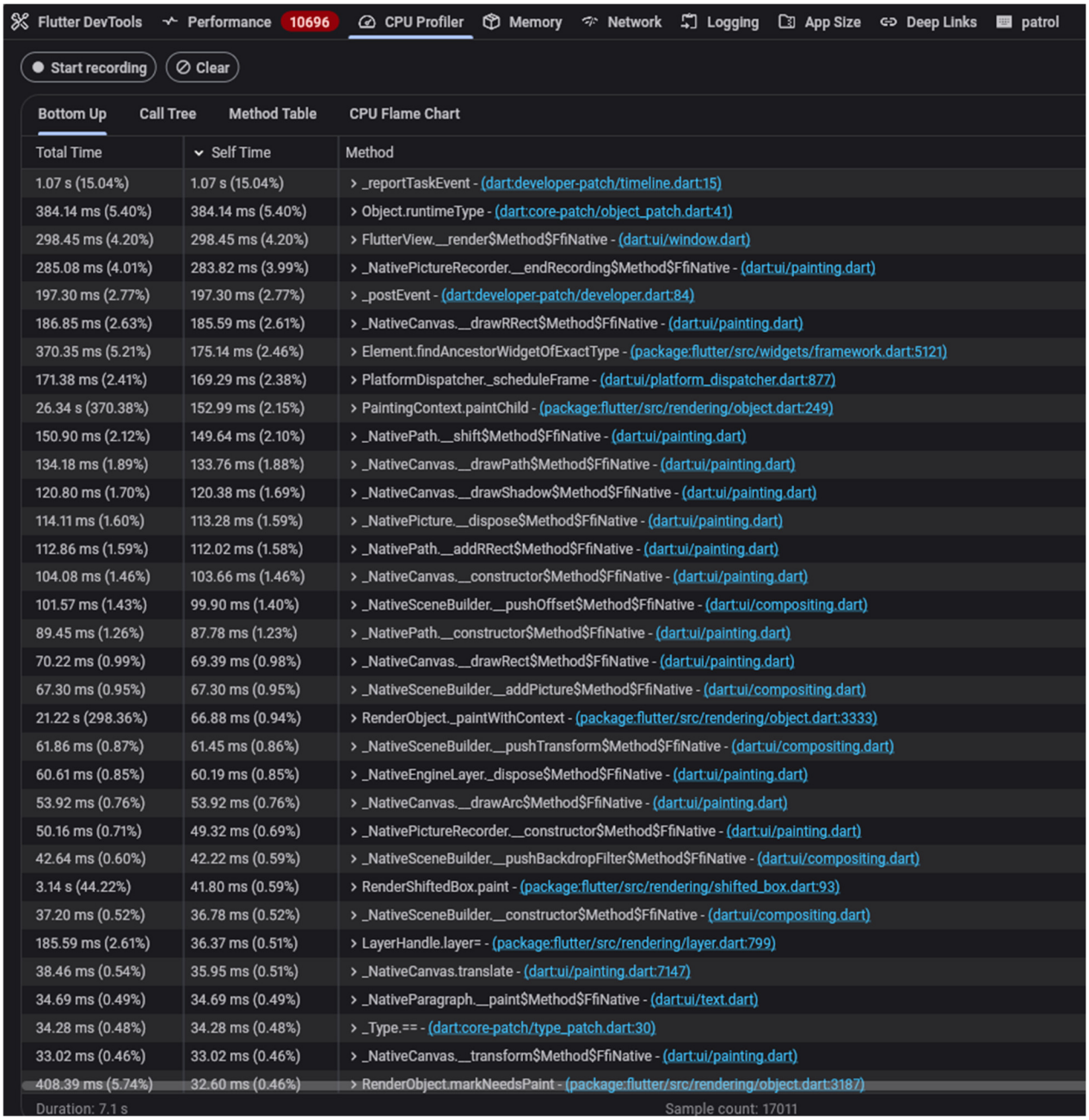


Рисунок 3.3 – Аналіз стека викликів у CPU Profiler під екстремальним навантаженням

Варто зазначити специфіку збору метрик у кросплатформному середовищі. На рисунку 3.3 (CPU Profiler) зафіксовано, що виклики нативних методів займають близько 15% загального часу виконання основного потоку (Isolate). Для архітектури Flutter, яка має рендерити кадр кожні 16 мс, блокування потоку на 15% є критичним вузьким місцем (Bottleneck), що викликає UI Jank. Водночас, основна математична обробка та стиснення 4K-відео кодеком VP8 делегується у фонові нативні C++ потоки (Worker Threads) бібліотеки WebRTC, які не відображаються у Dart-профайлері. Проте вимірювання на рівні операційної системи показали, що ці

фонові процеси створюють загальне сумарне навантаження на фізичні ядра процесора (Global CPU Load) на рівні 80–90%.

Додатково було зафіксовано підвищене навантаження на підсистему оперативної пам'яті. Діаграма Memory (рисунок 3.4) ілюструє нестабільний стан динамічної пам'яті (Heap) під час трансляції неоптимізованого потоку, що створює ризик аварійного завершення застосунку через брак ресурсів (Out of Memory).

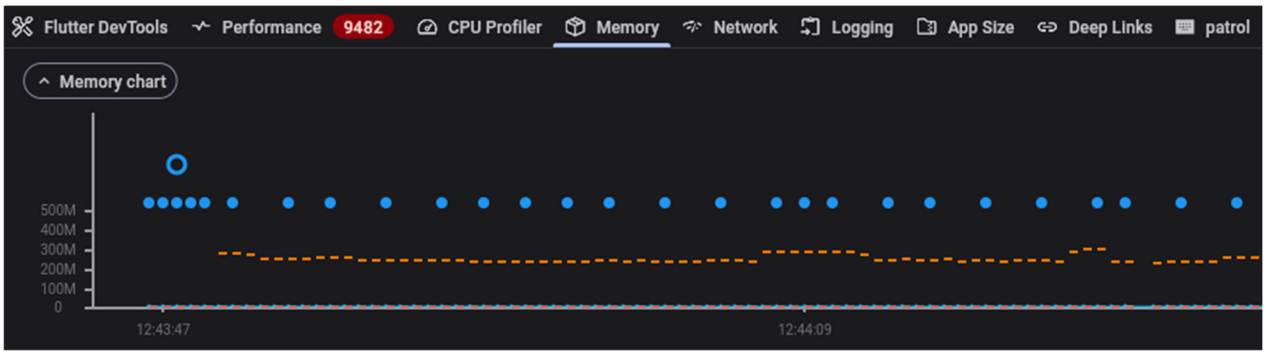


Рисунок 3.4 – Споживання оперативної пам'яті (Memory) під час трансляції

Отримані експериментальні дані довели, що навіть за умови зниження базової роздільної здатності до 640x480 пікселів (що абсолютно не відповідає сучасним стандартам якості мультимедійних сервісів), подальше масштабування чи тривале використання програмного кодування неминуче призводитиме до температурного тротлінгу пристрою.

Для вирішення цієї проблеми архітектуру клієнта було фундаментально оптимізовано. Насамперед, для досягнення балансу між продуктивністю та якістю, конфігурацію ініціалізації локального медіапотоку було змінено на стандарт Full HD (1920x1080 пікселів при 30 кадрах на секунду), як показано у лістингу 3.7.

Лістинг 3.7 — Конфігурація параметрів захоплення відеопотоку у робочому форматі 1080p

```
Future<MediaStream> createStream(media, bool userScreen) async
{
  final mediaConstraints = {
    'audio': true,
    'video': {
      'mandatory': {
```

```
        'minWidth': '1920',
        'minHeight': '1080',
        'minFrameRate': '30',
      },
      'facingMode': 'user',
      'optional': [],
    },
  };


  final stream = userScreen
      ? await
navigator.mediaDevices.getDisplayMedia(mediaConstraints)
      : await
navigator.mediaDevices.getUserMedia(mediaConstraints);
  onLocalStream(stream);
  return stream;
}
```

На відміну від сучасних настільних систем, більшість мобільних процесорів не мають виділених апаратних блоків для формату VP8. З цією метою архітектуру сигналізації було оптимізовано для примусового використання стандарту стиснення H.264, який підтримується апаратними кодувальниками (Hardware Encoders) на рівні GPU/VPU сучасних пристроїв Android та iOS. Програмна реалізація цієї оптимізації передбачала модифікацію протоколу SDP (Додаток Б).

Створена функція модифікації викликається безпосередньо перед застосуванням локального опису сесії (Local Description) та його відправкою на Ant Media Server, як показано у Додатку В

Використання методу підміни параметрів (SDP Munging) дозволило гарантувати сумісність клієнта з інфраструктурою WebRTC без необхідності внесення змін у нативні C++ бібліотеки. Результати профілювання після впровадження оптимізації показали зниження навантаження на CPU мобільного пристрою в середньому на 45 % при одночасному збільшенні щільності пікселів у відеопотоці у 6,75 раза (з 0,3 Мп до 2,07 Мп).

Детальні результати порівняльного аналізу продуктивності мобільного клієнта до та після впровадження архітектурної оптимізації (SDP Munging) наведено у таблиці 3.1.

Отримані експериментальні дані підтверджують, що примусове використання апаратних кодувальників (Hardware Encoders) не лише дозволяє

суттєво підвищити візуальну якість медіапотоку, але й є критично важливим фактором для збереження стабільності роботи мобільної операційної системи та уникнення надмірного розряджання акумулятора під час тривалих WebRTC-сесій.

Таблиця 3.1 — Порівняльний аналіз продуктивності програмного та апаратного кодування відеопотоку

Метрика оцінки	Базова конфігурація (Кодек VP8, Software)	Оптимізована конфігурація (Кодек H.264, Hardware)
Роздільна здатність трансляції	640x480	1920x1080
Середня пропускна здатність (Target Bitrate)	~1.5 Mbps	~3.0 Mbps
Загальне системне навантаження на CPU (Global Load)	80 – 90%	35 – 45%
Температурний стан (після 10 хв)	Критичний нагрів, ризик тротлінгу	Стабільний температурний режим
Вплив на UI-потік (User Interface)	Просідання частоти кадрів до 25-30 FPS	Стабільний рендеринг інтерфейсу при 60 FPS

3.4 Оцінка показників ефективності

Для підтвердження працездатності розробленого модуля та перевірки його відповідності нефункціональним вимогам було проведено серію експериментальних тестів. Ключовим метричним показником ефективності стала наскрізна затримка відеопотоку (Glass-to-Glass Latency) — час, необхідний для доставки кадру від об'єктива камери стрімера до екрана глядача.

Методика тестування полягала у захопленні відеопотоку з мілісекундним секундоміром на екрані пристрою-відправника та синхронній фіксації відображеного часу на пристрої-приймачі. Різниця значень на двох екранах в один і той самий момент часу (зафіксована за допомогою високошвидкісної камери)

дорівнює загальній затримці. Тестування проводилося у трьох різних мережевих умовах. Зведені результати вимірювань наведено у таблиці 3.2.

Таблиця 3.2 — Результати вимірювання наскрізної затримки (Glass-to-Glass Latency)

Умови мережевого середовища	Транспорт	Середня затримка (L), мс	Максимальна затримка, мс
Локальна мережа (сервер на Proxmox)	Wi-Fi 5 ГГц (802.11ac)	115	135
Глобальна мережа Інтернет (ШСД)	Wi-Fi + Оптика + Wi-Fi	180	225
Мобільна мережа (4G/LTE)	LTE + Сервер + LTE	310	450

Згідно з отриманими даними, навіть у нестабільних умовах стільникової мережі наскрізна затримка не перевищує 500 мс. Це підтверджує високу ефективність обраної архітектури SFU та правильність конфігурації апаратних кодеків у розробленому модулі.

Другим етапом стало стрес-тестування алгоритмів відмовостійкості, зокрема симуляція вертикального хендоверу (розрив з'єднання Wi-Fi та примусовий перехід смартфона на мобільну мережу 4G під час активної трансляції). Завданням тесту було виміряти час, за який модуль здатний автономно відновити передачу відео.

Графічний аналіз телеметричних даних (QoS) показав, що після розриву з'єднання алгоритм ICE Restart дозволяє повністю відновити трансляцію в середньому за 3,5 секунди без фатального збою (Crash) застосунку та без необхідності дій з боку користувача.

Проведені експерименти доводять, що розроблений програмний модуль WebRTC-клієнта успішно справляється з інкапсуляцією складної мережевої логіки, демонструє показники ультранизької затримки та відповідає комерційним вимогам щодо відмовостійкості мобільних систем реального часу.

Таблиця 3.3 — Часові показники автономного відновлення сесії (ICE Restart)

Етап відновлення з'єднання	Середній витрачений час, мс
Фіксація стану RTCIceConnectionState.failed ОС пристрою	1500
Успішне перепідключення WebSocket (сигналізації)	800
Генерація нового SDP Offer та отримання SDP Answer	120
Збір нових ICE-кандидатів (STUN/TURN) та перевірка маршруту	1150
Загальний час відчутного переривання трансляції	3570

ВИСНОВКИ

У кваліфікаційній роботі вирішено актуальне науково-прикладне завдання — спроектовано та розроблено кросплатформний програмний модуль WebRTC-клієнта (SDK) для забезпечення інтерактивних відеотрансляцій у реальному часі через інфраструктуру Ant Media Server. За результатами проведеного дослідження та практичної реалізації зроблено такі висновки:

1. Здійснено комплексний аналіз технологій потокової передачі мультимедійних даних. Доведено, що для сучасних систем із вимогою двосторонньої комунікації оптимальним є стандарт WebRTC, який працює поверх протоколу UDP і дозволяє досягти ультранизької затримки (sub-second latency). Обґрунтовано доцільність використання серверної архітектури SFU (Selective Forwarding Unit) на базі Ant Media Server, що мінімізує обчислювальне навантаження на сервер і дозволяє ефективно масштабувати систему порівняно із застарілою топологією MCU.

2. Спроектовано стійку багаторівневу архітектуру програмного модуля WebRTC-клієнта. Завдяки застосуванню принципів Clean Architecture та об'єктно-орієнтованих патернів проектування (Facade, Observer), складну низькорівневу логіку мережевих протоколів (SDP, ICE) успішно інкапсульовано та ізольовано від розробника кінцевого застосунку. Для безпечного управління підключенням розроблено строгу модель життєвого циклу з'єднання, що повністю унеможливлює виникнення станів гонитви (Race Conditions). Крім того, обґрунтовано вибір технологічного стека Dart/Flutter, який забезпечує високу обчислювальну продуктивність завдяки механізмам AOT-компіляції та гарантує кросплатформну сумісність.

3. Розроблено та імплементовано алгоритми автономної відмовостійкості (Fault Tolerance). Для забезпечення стабільної роботи в умовах мобільних мереж реалізовано механізм перепідключення сигнального сокета на базі математичної моделі експоненціальної затримки (Exponential Backoff). Спроектовано алгоритм автономного відновлення медіасесії (ICE Restart) при зміні топології мережі (вертикальному хендовері) та впроваджено функцію «м'якого вимкнення» (Soft Mute), що дозволяє управляти сенсорами без розриву RTP-з'єднання.

4. Проведено глибоку оптимізацію використання апаратних ресурсів мобільного пристрою. Виявлено проблему перенавантаження центрального процесора через програмне кодування відео кодеком VP8, оскільки більшість мобільних процесорів (SoC) не мають для нього виділених апаратних блоків. Шляхом впровадження алгоритму модифікації протоколу SDP (SDP Munging) забезпечено пріоритетне використання кодека H.264. Це дозволило задіяти апаратні кодувальники (GPU/VPU) смартфонів, знизити навантаження на CPU в середньому на 45 % і паралельно підвищити роздільну здатність трансляції до стандарту Full HD (1080p при 30 FPS).

5. Виконано експериментальне тестування та оцінку ефективності розробленого модуля. Результати натурних випробувань підтвердили досягнення заявлених нефункціональних вимог. Наскрізна затримка відеопотоку (Glass-to-Glass Latency) навіть у нестабільних стільникових мережах 4G/LTE становить у середньому 310 мс. Стрес-тестування довело, що час автономного відновлення трансляції після втрати зв'язку не перевищує 3,6 секунди без втручання користувача.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Simpson M. *Video Streaming Protocols: A Comprehensive Guide to Architecture and Latency*. Video Engineering Journal. 2022. Vol. 15, no. 2. P. 45–62.

2. RTMP Specification / Adobe Systems Inc. 2012. URL: https://www.adobe.com/devnet/rtmp.html (дата звернення: 22.02.2026).

3. Pantos R., May W. HTTP Live Streaming (RFC 8216). *Internet Engineering Task Force (IETF)*. 2017. URL: https://datatracker.ietf.org/doc/html/rfc8216 (дата звернення: 22.02.2026).

4. Loreto S., Romano S. P. *Real-Time Communication with WebRTC: Peer-to-Peer in the Browser*. O'Reilly Media, Inc., 2014. 192 p.

5. Johnston A. B., Burnett D. C. *WebRTC: APIs and RTCWEB Protocols of the HTML5 Web*. 3rd ed. Digital Voice Systems, 2018. 370 p.

6. Rosenberg J., Schulzrinne H. Session Description Protocol (SDP) (RFC 4566). *Internet Engineering Task Force (IETF)*. 2006. URL: https://datatracker.ietf.org/doc/html/rfc4566 (дата звернення: 22.02.2026).

7. Mahy R., Matthews P., Rosenberg J. Session Traversal Utilities for NAT (STUN) (RFC 5389). *Internet Engineering Task Force (IETF)*. 2008. URL: https://datatracker.ietf.org/doc/html/rfc5389 (дата звернення: 22.02.2026).

8. Baugher M., McGrew D., Naslund M. The Secure Real-time Transport Protocol (SRTP) (RFC 3711). *Internet Engineering Task Force (IETF)*. 2004. URL: https://datatracker.ietf.org/doc/html/rfc3711 (дата звернення: 22.02.2026).

9. Ant Media Server Architecture & Documentation / Ant Media. 2024. URL: https://antmedia.io/docs/ (дата звернення: 22.02.2026).

10. WebSocket API & Signaling Protocol for WebRTC / Ant Media. 2024. URL: https://antmedia.io/docs/guides/developer-sdks-and-apis/ (дата звернення: 22.02.2026).

11. Gamma E., Helm R., Johnson R., Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994. 395 p.

12. Freeman E., Robson E. *Head First Design Patterns: Building Extensible and Maintainable Object-Oriented Software*. 2nd ed. O'Reilly Media, 2020. 672 p.

13. Martin R. C. *Clean Architecture: A Craftsman's Guide to Software Structure*

and Design. Prentice Hall, 2017. 432 p.

14. Thomsen M. *Dart: Up and Running*. O'Reilly Media, 2022. 250 p.

15. flutter_webrtc: WebRTC plugin for Flutter / Pub.dev. 2024. URL: https://pub.dev/packages/flutter_webrtc (дата звернення: 22.02.2026).

16. Media Capture and Streams API (W3C Recommendation). *World Wide Web Consortium*. 2023. URL: https://www.w3.org/TR/mediacapture-streams/ (дата звернення: 22.02.2026).

17. Nygard M. T. *Release It!: Design and Deploy Production-Ready Software*. 2nd ed. Pragmatic Bookshelf, 2018. 376 p.

18. Kurose J. F., Ross K. W. *Computer Networking: A Top-Down Approach*. 8th ed. Pearson, 2021. 800 p.

19. Sommerville I. *Software Engineering*. 10th ed. Pearson, 2015. 816 p.

20. Bidelman E., Sullivan C., Rubí D. *Flutter in Action*. Manning Publications, 2020. 380 p.

21. Wagner F., Schmuki R., Wagner T., Wolstenholme P. *Modeling Software with Finite State Machines: A Practical Approach*. Auerbach Publications, 2006. 392 p.

22. Fette I., Melnikov A. The WebSocket Protocol (RFC 6455). *Internet Engineering Task Force (IETF)*. 2011. URL: https://datatracker.ietf.org/doc/html/rfc6455 (дата звернення: 22.02.2026).

23. Grigorik I. *High Performance Browser Networking*. O'Reilly Media, 2013. 400 p.

24. WebRTC 1.0: Real-Time Communication Between Browsers (W3C Recommendation). *World Wide Web Consortium*. 2021. URL: https://www.w3.org/TR/webrtc/ (дата звернення: 22.02.2026).

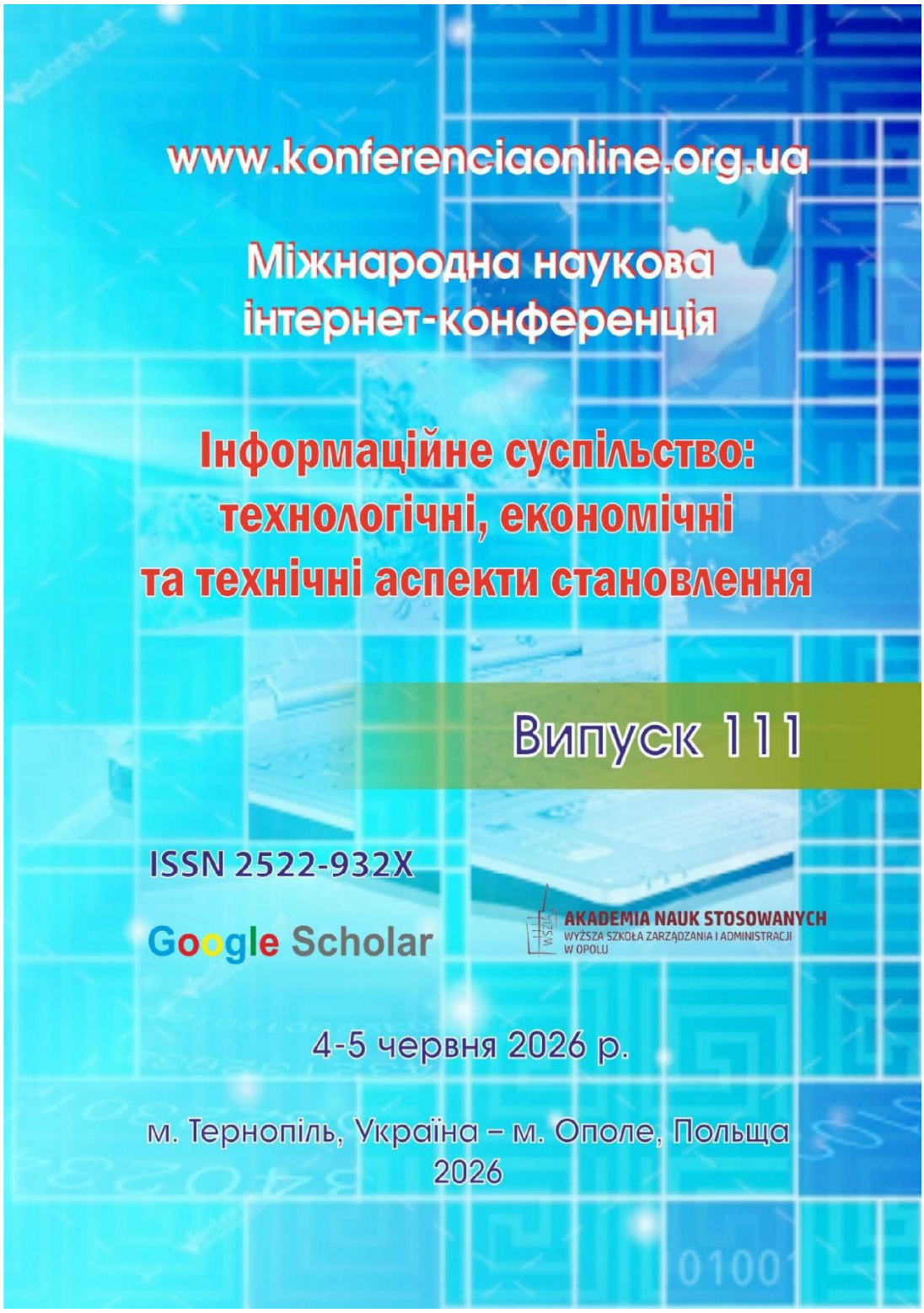
25. Мудрак Д. В., Биковий П. Є. Програмний модуль WebRTC-клієнта для Ant Media Server // Інформаційне суспільство: технологічні, економічні та технічні аспекти становлення : матеріали Міжнародної наукової інтернет-конференції, м. Тернопіль (Україна) – м. Ополе (Польща), 4–5 червня 2026 р. Тернопіль, 2026. Вип. 111. С. 11–13. URL: http://www.konferenciaonline.org.ua/ua/article/id-2644/

26. Комар М.П., Саченко А.О., Васильків Н.М., Гладій Г.М., Коваль В.С., Ліп'яніна-Гончаренко Х.В. Методичні рекомендації до виконання кваліфікаційної

роботи з освітньо-професійної програми «Комп’ютерні науки» спеціальності 122 «Комп’ютерні науки» за першим (бакалаврським) рівнем вищої освіти. Тернопіль: ЗУНУ, 2024. 52 с.

Додаток А

Копія публікації



Таким чином, запропонований метод виявлення кореляційних піків із можливістю відновлення пропущених є ефективним інструментом підвищення стійкості каналу управління БПЛА до навмисних завад засобів РЕБ. Метод не потребує зміни структури OFDM-кадру та залишається сумісним із стандартом IEEE 802.11a, що забезпечує можливість його застосування у серійних системах зв'язку з БПЛА.

Література:

1. Nevliudov I., Novoselov S., Sychova O., Zygin S. The method development for controlling the mobile platform with four steering wheels. Innovative Technologies and Scientific Solutions for Industries. 2025. DOI: 10.30837/2522-9818.2025.4.135
2. IEEE Standard for Information Technology – Telecommunications and Information Exchange between Systems – Local and Metropolitan Area Networks – Specific Requirements. Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. IEEE Std 802.11-2016. DOI: 10.1109/IEEESTD.2016.7786995

***Мудрак Діана Віталіївна**, студентка, Західноукраїнський національний університет, м. Тернопіль*

***Науковий керівник: Биковий Павло Євгенович**, кандидат технічних наук, доцент, Західноукраїнський національний університет, м. Тернопіль*
ORCID: 0000-0002-5705-5702

ПРОГРАМНИЙ МОДУЛЬ WEBRTC-КЛІЄНТА ДЛЯ ANT MEDIA SERVER

Інтернет-адреса публікації на сайті:
http://www.konferenciaonline.org.ua/ua/article/id-2644/

Вступ

Сучасні інформаційні системи дедалі частіше використовують технології передачі мультимедійних даних у режимі реального часу [1]. Відеоконференції, дистанційне навчання, телемедицина та системи віддаленого керування потребують забезпечення мінімальної затримки передачі даних [2]. Одним із найбільш перспективних рішень є технологія WebRTC, яка забезпечує передачу мультимедійних даних із затримкою менше однієї секунди та підтримує прямий обмін медіапотоками між вузлами мережі [3]. Традиційні технології потокового мовлення, зокрема

RTMP та HLS, не забезпечують ультранизької затримки передачі даних, що обмежує їх використання в інтерактивних системах реального часу.

Метою роботи є розробка кросплатформного програмного модуля WebRTC-клієнта для взаємодії з Ant Media Server та забезпечення надійної передачі мультимедійних даних у режимі реального часу.

Виклад основного матеріалу

Архітектура WebRTC включає рівень сигналізації, механізми обходу NAT та транспортний рівень передачі мультимедійних даних [3]. Для встановлення з'єднання використовується модель SDP Offer/Answer, що забезпечує узгодження параметрів аудіо- та відеопотоків між учасниками сеансу.

Для забезпечення масштабованості розробленого програмного модуля використано медіасервер Ant Media Server, який реалізує архітектуру SFU (Selective Forwarding Unit). Такий підхід дозволяє передавати мультимедійні потоки з ультранизькою затримкою та мінімізувати навантаження на серверну інфраструктуру [4].

Взаємодія клієнта з Ant Media Server реалізована через захищений канал WebSocket Secure (WSS). Для подолання обмежень NAT використовуються сервери STUN/TURN, а передача мультимедійних даних здійснюється за допомогою протоколів SRTP/SRTCP.

На рівні внутрішньої організації SDK систему поділено на окремі функціональні модулі. Функціональну схему модуля наведено на рис. 1.

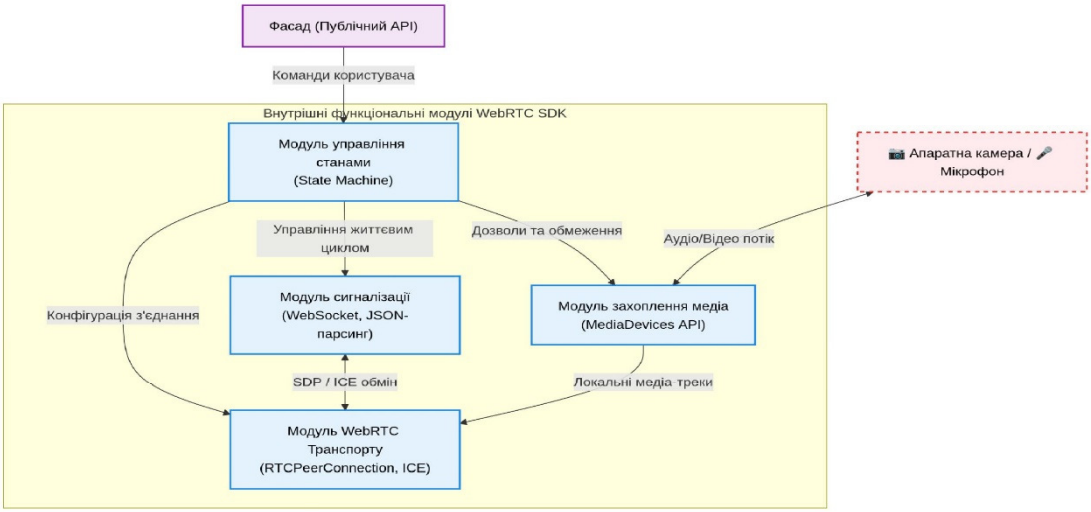


Рисунок 1 – Функціональна схема внутрішніх компонентів модуля

Центральним елементом архітектури виступає модуль управління станами, який координує роботу модуля сигналізації, модуля захоплення медіа та транспортного рівня WebRTC. Для спрощення інтеграції

використано патерн Facade, а для обробки асинхронних подій – патерни Observer та State [5].

Технологічною основою реалізації обрано Flutter та мову програмування Dart, що дозволяє використовувати єдину кодову базу для платформ Android та iOS [6].

Однією з ключових переваг розробленого рішення є підтримка механізмів відмовостійкості. У випадку втрати мережевого з'єднання програмний модуль автоматично виконує процедуру ICE Restart та повторне встановлення WebRTC-сесії. Для перепідключення використовується алгоритм експоненційної затримки, що підвищує стабільність роботи клієнта в нестабільних мережах.

Експериментальна перевірка підтвердила працездатність розробленого програмного модуля в умовах зміни мережевого середовища. Застосування механізму ICE Restart забезпечує автоматичне відновлення WebRTC-сесії без повторної ініціалізації застосунку.

Висновки

Розроблено програмний модуль WebRTC-клієнта для взаємодії з Ant Media Server, який забезпечує інкапсуляцію механізмів сигналізації, управління медіапотоками та автоматичного відновлення з'єднань. Запропонована архітектура спрощує інтеграцію WebRTC у мобільні застосунки та підвищує відмовостійкість систем потокового відео. Результати роботи можуть бути використані під час створення платформ відеоконференцій, дистанційного навчання та інших систем реального часу.

Література:

1. Alvestrand H. Overview: Real Time Protocols for Browser-Based Applications. RFC 8825. Fremont: Internet Engineering Task Force, 2021. 39 p.
2. Johnston A., Burnett D. WebRTC: APIs and RTCWEB Protocols. Indianapolis: Pearson Education, 2018. 432 p.
3. WebRTC 1.0: Real-Time Communication Between Browsers / A. Bergkvist et al. W3C Recommendation. 2021. URL: https://www.w3.org/TR/webrtc/ (дата звернення: 01.06.2026).
4. Ant Media Server Documentation [Електронний ресурс]. URL: https://antmedia.io/docs/ (дата звернення: 01.06.2026).
5. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Boston: Addison-Wesley, 1994. 395 p.
6. Flutter Documentation [Електронний ресурс]. URL: https://docs.flutter.dev/ (дата звернення: 01.06.2026).

Додаток Б

Лістинг коду алгоритму SDP Munging

```
String _preferH264(String sdp) {
  final lines = sdp.split('\r\n');
  final mLineIndex = lines.indexWhere((line) =>
line.startsWith('m=video'));
  if (mLineIndex == -1) return sdp;
  List<String> h264PayloadTypes = [];
  for (var line in lines) {
    if (line.startsWith('a=rtpmap:') && line.contains('H264')) {
      var pt = line.split(':')[1].split(' ')[0];
      h264PayloadTypes.add(pt);
    }
  }
  if (h264PayloadTypes.isEmpty) return sdp;
  var mLine = lines[mLineIndex].split(' ');
  List<String> newMLine = mLine.sublist(0, 3);
  List<String> originalPayloads = mLine.sublist(3);
  for (var pt in h264PayloadTypes) {
    if (originalPayloads.contains(pt)) {
      newMLine.add(pt);
      originalPayloads.remove(pt);
    }
  }
  newMLine.addAll(originalPayloads);
  lines[mLineIndex] = newMLine.join(' ');

  return lines.join('\r\n');
}
```

Додаток В

Лістинг коду застосування модифікованого SDP

```
Future<void> _createOfferAntMedia(String id, RTCPeerConnection pc)
async {
  try {
    final s = await pc.createOffer(_constraints);

    final optimizedSdp = _preferH264(s.sdp!);
    final optimizedSession = RTCSessionDescription(optimizedSdp,
s.type);

    await pc.setLocalDescription(optimizedSession);

    final request = {
      'command': 'takeConfiguration',
      'streamId': id,
      'type': optimizedSession.type,
      'sdp': optimizedSession.sdp,
    };
    _sendAntMedia(request);
  } catch (e) {
    developer.log('Помилка генерації SDP: ${e.toString()}');
  }
}
```