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

БОДОНЬ Назарій Сергійович

Мобільний застосунок для відеострімінгу з використанням Ant Media Server SDK / Mobile Application for Live Streaming Using Ant Media Server SDK

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

Кваліфікаційна робота

Виконав студент групи КН-43
Н. С. Бодонь

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
БОДОНЮ Назарію Сергійовичу
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи: Мобільний застосунок для відеострімінгу з використанням Ant Media Server SDK / Mobile Application for Live Streaming Using Ant Media Server SDK
керівник роботи _____Биковий П.Є., к.т.н., доц._____,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені 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, технічна література.
4. Основні питання, які потрібно розробити
- проаналізувати сучасні протоколи потокової передачі відео (WebRTC, HLS) та архітектури існуючих систем відеострімінгу;
- розробити трирівневу архітектуру системи, що включає мобільний клієнт, керуючий бекенд-сервер та медіасервер;
- реалізувати програмні механізми управління медіапотоками («Soft Mute») та автономного відновлення з'єднань (ICE Restart) в умовах нестабільних мобільних мереж;
- забезпечити безпеку трансляцій шляхом інтеграції механізмів JWT-авторизації та автоматичної синхронізації статусів через Webhooks;
- провести експериментальне тестування наскрізної затримки відео (glass-to-glass latency) та стрес-тестування відмовостійкості системи.

5. Перелік графічного матеріалу у роботі:
- архітектурна модель мобільного застосунку на базі Clean Architecture та BLoC;
- DFD-діаграма процесу створення трансляції та оновлення її стану;
- DFD-діаграма підсистеми керування медіаконтентом та відображення активних ефірів;
- Трирівнева архітектура локального інформаційного забезпечення мобільного клієнта;
- Архітектурна модель взаємодії мобільного клієнта (Flutter), керуючого сервера (Spring Boot) та медіасервера (Docker контейнер).

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 _____ _____Н. С. Бодонь____
(підпис) (прізвище та ініціали)

Керівник роботи _____ ____П. Є. Биковий_____
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Кваліфікаційна робота на тему «Мобільний застосунок для відеострімінгу з використанням Ant Media Server SDK» на здобуття освітнього ступеня «бакалавр» зі спеціальності 122 «Комп’ютерні науки» освітньої програми «Комп’ютерні науки» написана обсягом в 73 сторінки і містить 11 ілюстрацій, 2 таблицці, 3 додатки та 28 використаних джерел.

Метою роботи є розробка мобільного застосунку, здатного забезпечувати двосторонню потокову передачу відео та аудіоданих у реальному часі за допомогою інтеграції з інфраструктурою Ant Media Server.

Методи дослідження: системний аналіз, методи об'єктно-орієнтованого проектування, теорія комп'ютерних мереж та управління медіапотоками (WebRTC).

Внаслідок виконання роботи створено програмний продукт, який дозволяє користувачам ініціювати, переглядати та управляти прямими відеотрансляціями зі своїх мобільних пристроїв. Розроблено архітектурну модель взаємодії мобільного клієнта (на базі Flutter) із керуючим сервером (Spring Boot) та медіасервером. Імплементовано механізми відмовостійкості та оптимізації якості відеопотоку. Експериментально підтверджено ефективність застосованих рішень шляхом вимірювання наскрізної затримки (glass-to-glass latency).

Результати роботи можуть бути впроваджені як базовий модуль у комерційні стрімінгові платформи, системи дистанційного навчання або корпоративні комунікаційні сервіси.

Ключові слова: ВІДЕОСТРІМІНГ, МОБІЛЬНИЙ ЗАСТОСУНОК, FLUTTER, WEBRTC, ANT MEDIA SERVER, ПРЯМІ ТРАНСЛЯЦІЇ.

ANNOTATION

Qualification work on the topic «Mobile Application for Live Streaming Using Ant Media Server SDK» for Bachelor's degree on speciality 122 «Computer Science» educational and professional program «Computer Science» is written on 73 pages and it contains 11 figures, 2 tablei, 3 annexes and 28 sources.

The aim of the study is to develop a mobile application capable of providing real-time two-way streaming of video and audio data through integration with the Ant Media Server infrastructure.

Research methods: system analysis, object-oriented design methods, theory of computer networks, and media flow management (WebRTC).

As a result of the work, a software product was created that allows users to initiate, view, and manage live video broadcasts from their mobile devices. An architectural model of interaction between the mobile client (based on Flutter), the control server (Spring Boot), and the media server has been developed. Mechanisms for fault tolerance and video stream quality optimization have been implemented. The efficiency of the applied solutions was experimentally confirmed by measuring the glass-to-glass latency.

The results of the work can be implemented as a core module into commercial streaming platforms, distance learning systems, or corporate communication services.

Keywords: VIDEO STREAMING, MOBILE APPLICATION, FLUTTER, WEBRTC, ANT MEDIA SERVER, LIVE BROADCASTS.

ЗМІСТ

Вступ 7
1 Теоретичні основи організації відеострімінгу в мобільних системах 10
1.1 Базові принципи та протоколи потокової передачі відео 10
1.2 Аналіз сучасних технологій та існуючих рішень відеострімінгу 15
1.3 Постановка задачі розробки мобільного застосунку відеострімінгу 20
2 Проектування та розробка мобільного застосунку з використанням Ant Media Server SDK 23
2.1 Архітектурна модель системи взаємодії мобільного клієнта та медіасервера 23
2.2 Інтеграція мобільного SDK та управління медіапотоками 25
2.3 Взаємодія з REST API Ant Media Server та організація безпеки 28
2.4 Організація інформаційного забезпечення та збереження локальних даних 33
3 Практична реалізація та тестування мобільного застосунку для відеострімінгу 36
3.1 Розгортання інфраструктури та інтеграція базових сервісів 36
3.2 Проектування та програмна реалізація користувацького інтерфейсу 40
3.3 Програмна реалізація механізмів управління медіапотоками та відмовостійкості 56
3.4 Експериментальне тестування застосунку та оцінка показників ефективності 58
Висновки 61
Список використаних джерел 63
Додаток А Копія публікації 66
Додаток Б Лістинг модуля інтеграції керуючого сервера з Ant Media Server REST API 70
Додаток В Лістинг модуля управління WebRTC-сесією та захопленням медіаданих «AddressBlock» 71

ВСТУП

Сучасний етап розвитку інформаційних технологій та глобальної мережі Інтернет характеризується стрімким переходом від класичних текстових та статичних медіа-форматів до інтерактивного відеоконтенту. Технології потокової передачі відео (відеострімінг) стали невід'ємною частиною багатьох сфер життя: від індустрії розваг і соціальних мереж до корпоративних комунікацій, телемедицини та систем дистанційного навчання.

Незважаючи на значний прогрес у пропускній здатності мереж, забезпечення стабільної, двосторонньої та високоякісної трансляції відео у реальному часі (з мінімальною наскрізною затримкою — sub-second latency) залишається складною інженерною задачею. Традиційні протоколи, такі як RTMP (Real-Time Messaging Protocol) або HLS (HTTP Live Streaming), хоч і забезпечують високу масштабованість, проте генерують затримку від кількох секунд до десятків секунд, що є критично неприйнятним для систем, які вимагають миттєвої взаємодії (наприклад, інтерактивні вебінари, онлайн-аукціони, стрімінг з мобільних камер спостереження).

У цьому контексті особливого значення набуває впровадження стандарту WebRTC (Web Real-Time Communication), що дозволяє організувати пряме (Peer-to-Peer) з'єднання з ультранизькою затримкою. Водночас розробка мобільних клієнтів для WebRTC-систем ускладнюється необхідністю підтримки різних операційних систем (iOS, Android), управлінням апаратними ресурсами пристрою в умовах нестабільних мобільних мереж та забезпеченням складної логіки сигналізації. Використання кросплатформних фреймворків, зокрема Flutter, у поєднанні з потужними медіасерверами, такими як Ant Media Server, є перспективним напрямком для вирішення цих проблем.

Актуальність кваліфікаційної роботи зумовлена необхідністю створення сучасних, відмовостійких та кросплатформних мобільних рішень, здатних забезпечити надійну передачу відео в реальному часі з використанням передових протоколів та серверної інфраструктури.

Метою кваліфікаційної роботи є розробка та практична імплементація кросплатформного мобільного застосунку для відеострімінгу, здатного забезпечувати двосторонню потокову передачу медіаданих у реальному часі з мінімальною затримкою за допомогою інтеграції з інфраструктурою Ant Media Server.

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

– провести аналіз існуючих технологій, протоколів потокової передачі відео (WebRTC, RTMP, HLS) та архітектур сучасних систем відеострімінгу.;

– спроектувати загальну архітектурну модель взаємодії мобільного клієнта, керуючого бекенд-сервера та медіасервера;

– розробити серверну частину на базі Spring Boot для забезпечення безпеки (генерація JWT-токенів) та синхронізації статусів трансляцій;

– розробити користувацький інтерфейс (UI) та бізнес-логіку мобільного застосунку за допомогою фреймворку Flutter з використанням реактивного управління станом (Riverpod);

– реалізувати механізми управління локальними медіа-треками та забезпечення відмовостійкості з'єднання (ICE Restart) в умовах зміни мобільних мереж;

– провести експериментальне тестування розробленого застосунку та оцінити показники його ефективності .

Об’єктом дослідження є процес потокової передачі відео- та аудіоданих у мобільних мережах реального часу.

Предметом дослідження є методи розробки, архітектурні патерни та програмні інструменти (Flutter, WebRTC, Ant Media Server SDK) для створення мобільних систем відеострімінгу.

Методи дослідження. Для вирішення поставлених завдань у роботі використовувались такі методи: системний аналіз (для вивчення існуючих рішень та протоколів), об'єктно-орієнтоване проектування (для розробки архітектури програмного забезпечення), методи програмної інженерії (реалізація мобільного

клієнта та серверної частини), а також емпіричні методи (для експериментального тестування наскрізної затримки та відмовостійкості системи).

Наукова новизна одержаних результатів полягає у подальшому розвитку підходів до побудови мобільних стрімінгових клієнтів на базі WebRTC. Зокрема, імплементовано алгоритм "Soft Mute" (керування станом медіа-треків без розриву SDP-сесії) у зв'язці з механізмом автономного відновлення з'єднання "ICE Restart" в екосистемі Flutter, що дозволило підвищити відмовостійкість трансляцій при вертикальному хендовері (зміні мереж) без необхідності втручання користувача.

Практичне значення одержаних результатів полягає у створенні готового до використання програмного комплексу (мобільний застосунок та керуючий сервер), який демонструє стабільну передачу відео високої чіткості із затримкою до 300 мілісекунд. Розроблена архітектура може бути використана як базовий модуль для комерційних платформ, систем відеомоніторингу, рішень для дистанційного навчання або корпоративних систем зв'язку, що розгортаються на базі інфраструктури Ant Media Server.

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 «Мобільний застосунок для відеострімінгу з використанням Ant Media Server SDK» (Додаток А).

1 ТЕОРЕТИЧНІ ОСНОВИ ОРГАНІЗАЦІЇ ВІДЕОСТРІМІНГУ В МОБІЛЬНИХ СИСТЕМАХ

1.1 Базові принципи та протоколи потокової передачі відео

Сучасний етап розвитку інформаційних технологій та телекомунікаційних мереж характеризується стрімким зростанням обсягів мультимедійного контенту, що передається через глобальну мережу Інтернет. Потокова передача відео (відеострімінг) — це складний багаторівневий процес безперервної доставки медіаданих від сервера до кінцевого користувача або між рівноправними вузлами мережі, що дозволяє розпочати відтворення контенту до повного його завантаження на пристрій. Для забезпечення цього процесу вимагається використання спеціалізованих мережевих протоколів, архітектура яких визначає ключові параметри системи: наскрізну затримку (latency), якість зображення, стійкість до втрат пакетів, пропускну здатність та можливості горизонтального масштабування [1].

Фундаментальною основою будь-якого відеострімінгу є процеси просторового та часового стиснення (кодування) і подальшого відновлення (декодування) відеоданих. Нестиснені (сирі) відеодані займають надзвичайно великий обсяг пропускної здатності мережі (наприклад, потік 1080p при 60 кадрах на секунду без стиснення потребує понад 2 Гбіт/с). Тому перед передачею вони стискаються за допомогою спеціальних математичних алгоритмів — відеокодеків (зокрема, H.264/AVC, H.265/HEVC, VP8, VP9, AV1). Аудіодані також підлягають психоакустичному стисненню (найчастіше використовуються кодеки AAC або стійкий до втрат пакетів Opus). Після стиснення аудіо та відео мультиплексуються у медіаконтейнер або розбиваються на транспортні пакети (RTP-пакети) відповідно до обраного протоколу передачі [2].

Розглянемо детальніше основні мережеві протоколи, які сьогодні домінують на ринку потокового відео, визначаючи їхні архітектурні переваги та недоліки в контексті мобільних систем.

Протокол RTMP (Real-Time Messaging Protocol) був розроблений компанією Macromedia (яку згодом поглинула Adobe Systems) для передачі потокового аудіо,

відео та керуючих даних через Інтернет між Flash-плеєром та сервером. Незважаючи на повну відмову індустрії від технології Flash, RTMP досі залишається стандартом де-факто (галузевим стандартом) для так званого "першого милі" (ingest) — доставки оригінального відеопотоку від джерела (наприклад, програми OBS Studio, апаратного енкодера або мобільного застосунку) до медіасервера [3].

RTMP функціонує поверх протоколу транспортного рівня TCP (Transmission Control Protocol), що гарантує доставку всіх пакетів у правильному порядку. Процес встановлення з'єднання починається зі складного тристороннього «рукостискання» (handshake), після чого створюється постійний (persistent) дуплексний канал зв'язку. Переваги: відносно низька затримка на рівні 1–3 секунд за ідеальних мережевих умов, висока надійність передачі, широка підтримка практично всіма апаратними та програмними енкодерами на ринку. Недоліки: використання TCP є критичним вразливим місцем у бездротових мобільних мережах. При втраті хоча б одного пакета алгоритми TCP зупиняють передачу наступних даних до моменту успішної повторної передачі втраченого сегмента (явище Head-of-line blocking). Це призводить до експоненціального накопичення затримки (буферизації) при погіршенні якості мережі. Крім того, протокол не підтримується нативно сучасними веб-браузерами (стандарт HTML5), тому для відтворення кінцевим глядачам потік на стороні сервера обов'язково перетворюється (трансмуксується) на інші формати, такі як HLS або WebRTC [4].

HLS (HTTP Live Streaming) Розроблений інженерами компанії Apple, цей протокол докорінно змінив парадигму доставки медіаконтенту кінцевим користувачам (Broadcasting). Замість встановлення постійного з'єднання, HLS використовує принцип сегментації: він розбиває суцільний медіапотік на невеликі фрагменти (чанки) тривалістю від 2 до 10 секунд (зазвичай це файли транспортного потоку .ts або фрагментований MP4). Інформація про ці сегменти, їхня черговість та тривалість записується у спеціальний текстовий файл-маніфест (плейлист) з розширенням .m3u8 [5].

Ключовою архітектурною особливістю HLS є використання стандартного протоколу прикладного рівня HTTP (зазвичай через порт 80 або 443 для HTTPS). Це дозволяє потоку легко проходити через будь-які фаєрволи, NAT-маршрутизатори та,

що найважливіше, ефективно кешуватися на периферійних серверах мереж доставки контенту (CDN). Крім того, HLS нативно підтримує технологію адаптивного бітрейту (Adaptive Bitrate Streaming - ABR). Сервер генерує кілька версій потоку з різною роздільною здатністю (наприклад, 1080p, 720p, 480p), а клієнтський плеєр постійно аналізує пропускну здатність мережі та автоматично перемикається між сегментами відповідної якості без переривання відтворення. Переваги: неперевершена глобальна масштабованість, стабільно висока якість відео, ідеальна робота через CDN інфраструктуру, стійкість до коливань швидкості мобільного інтернету. Недоліки: фундаментально висока наскрізна затримка. Для початку відтворення плеєр повинен завантажити маніфест і щонайменше 2–3 сегменти відео. У стандартній конфігурації це призводить до затримки від 10 до 30 секунд. Сучасні розширення (Low-Latency HLS) дозволяють знизити її до 2–5 секунд, проте така затримка все одно робить технологію HLS абсолютно непридатною для систем двостороннього спілкування (відеоконференцій) чи систем управління реального часу (IoT, автономні дрони) [6].

WebRTC (Web Real-Time Communication) WebRTC — це потужний відкритий проект та набір стандартів, ініційований компанією Google, а згодом формалізований консорціумом W3C та інженерною радою IETF. Його головна місія — забезпечити високоякісну комунікацію в реальному часі безпосередньо через API браузерів і нативних мобільних застосунків без необхідності встановлення сторонніх плагінів чи проміжних серверів для базових P2P-з'єднань [7].

На відміну від RTMP та HLS, транспортна архітектура WebRTC за замовчуванням базується на протоколі UDP (User Datagram Protocol). Це означає відсутність вбудованих механізмів підтвердження доставки та повторної передачі втрачених пакетів на рівні транспортного рівня ОС, що повністю виключає проблему непередбачуваної буферизації. Якщо відеокадр (або його частина) губиться в нестабільній мобільній мережі, рушій WebRTC просто ігнорує його, приховує артефакти за допомогою алгоритмів PLC (Packet Loss Concealment) і продовжує відтворювати наступний актуальний кадр. Для забезпечення конфіденційності всі дані в WebRTC в обов'язковому порядку шифруються: медіатрафік за допомогою SRTP (Secure Real-time Transport Protocol), а канали даних

— через DTLS (Datagram Transport Layer Security). В таблиці 1.1 представлена порівняльна характеристика мережевих протоколів потокового відео.

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 (Sub-second)
Адаптивний бітрейт (ABR)	Не підтримується	Нативна підтримка (на клієнті)	Динамічний (на рівні кодека через RTCP)
Підтримка в браузерах	Немає (застаріло)	Так (нативно в Safari, частково через MSE в інших)	Нативна підтримка всіма сучасними браузерами (HTML5)
Цільове призначення	Ingest (публікація)	Масове мовлення (Broadcasting) на мільйони глядачів	Інтерактивний двосторонній зв'язок у реальному часі

Процес встановлення WebRTC-з'єднання є складним багатоетапним процесом, який вимагає використання додаткових логічних вузлів мережі:

1. Signaling Server (Сервер сигналізації): Специфікація WebRTC навмисно не стандартизує транспорт для процесу виявлення пристроями одне одного. Для цього розробники використовують зовнішній механізм сигналізації (найчастіше на базі WebSockets або Server-Sent Events). Через цей канал пристрої обмінюються метаданими за протоколом SDP (Session Description Protocol) за моделлю Offer/Answer (Пропозиція/Відповідь), узгоджуючи підтримувані кодеки (VP8, H.264), роздільну здатність, параметри мережі та ключі шифрування.

2. STUN/TURN сервери (Інфраструктура ICE): Оскільки переважна більшість сучасних мобільних пристроїв знаходяться за NAT (Network Address Translation) і не мають прямих публічних IP-адрес, для встановлення P2P-з'єднання

використовується фреймворк ICE (Interactive Connectivity Establishment). ICE звертається до протоколу STUN (Session Traversal Utilities for NAT), щоб пристрій міг дізнатися свою публічну IP-адресу та порт. Якщо пряме з'єднання неможливе через жорсткі корпоративні фаєрволи (Symmetric NAT), медіатрафік примусово ретранслюється через TURN-сервер (Traversal Using Relays around NAT), що гарантує встановлення з'єднання у 100% випадків [8].

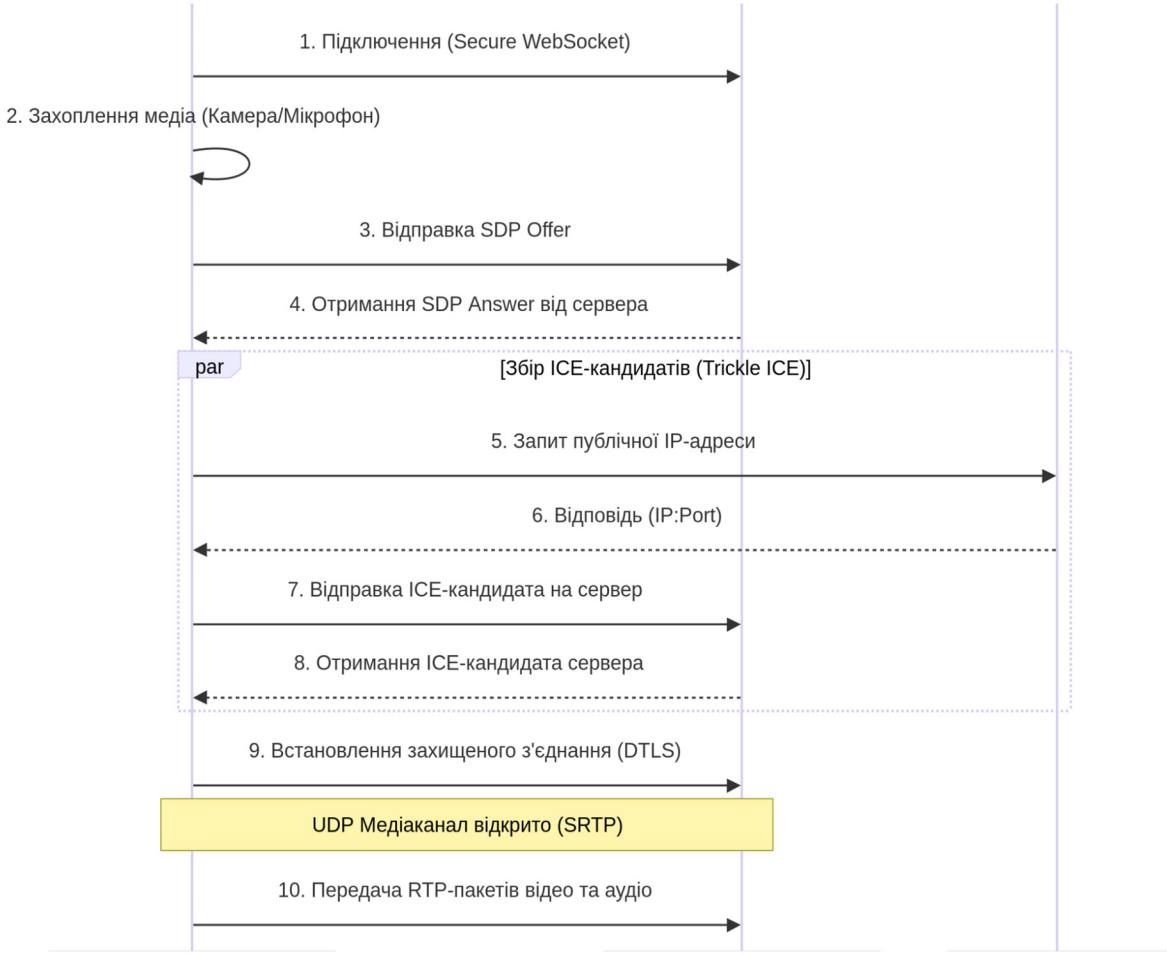


Рисунок 1.1 – Процес сигналізації та встановлення P2P-з'єднання у WebRTC

Саме синергія транспорту UDP, протоколів RTP/RTCP та оптимізованих кодеків дозволяє WebRTC досягати ультранизької затримки (sub-second latency, зазвичай 100–500 мілісекунд), роблячи цей протокол безальтернативним вибором для проектування мобільного застосунку, що розглядається у даній кваліфікаційній роботі.

1.2 Аналіз сучасних технологій та існуючих рішень відеострімінгу

Зі стрімким зростанням вимог до мобільного програмного забезпечення, вибір технологічного стеку для розробки застосунку для відеострімінгу стає критичним етапом інженерного проектування. Робота з відеокамерою, мікрофоном, мережевими протоколами реального часу та апаратним прискоренням графіки вимагає високої обчислювальної продуктивності та глибокої оптимізації. Архітектори та розробники постають перед стратегічним вибором між класичними нативними технологіями та сучасними кросплатформними фреймворками [9].

Нативна розробка (Native Development) передбачає створення двох окремих ізольованих застосунків для кожної операційної системи з використанням офіційно підтримуваних мов програмування та інструментарію: мови Swift або Objective-C у середовищі Xcode для екосистеми iOS; мови Kotlin або Java у середовищі Android Studio для ОС Android. Архітектурні особливості: Нативний вихідний код компілюється безпосередньо в машинні інструкції конкретної архітектури процесора пристрою (ARM64). Він має прямий, безпосередній доступ до всіх низькорівневих API операційної системи. Робота з мультимедіа: Для відеострімінгу нативна розробка надає доступ до таких потужних системних фреймворків, як AVFoundation (в iOS) та Camera2 API / MediaCodec (в Android). Це дозволяє максимально ефективно використовувати апаратні кодеки пристрою на рівні кристала (SoC), мінімізувати копіювання буферів пам'яті (Zero-copy operations) та оптимізувати енергоспоживання. Недоліки: Головний недолік нативного підходу полягає в економічній та часовій площині. Розробка вимагає створення та підтримки двох окремих кодових баз, залучення двох незалежних команд інженерів. Це вдвічі збільшує вартість проекту (TCO - Total Cost of Ownership), ускладнює синхронізацію релізів функціоналу та процеси забезпечення якості (QA) [10].

Кросплатформні рішення (Cross-Platform Development) Основна парадигма кросплатформної розробки — написання бізнес-логіки та інтерфейсу один раз із подальшою компіляцією або інтерпретацією для виконання на різних операційних системах (Write once, run anywhere). Серед найбільш популярних і технологічно зрілих рішень сьогодні домінують React Native та Flutter.

React Native (від Meta/Facebook): Базується на мові JavaScript (або TypeScript) та компонентному підході бібліотеки React. Ключовою архітектурною особливістю (і водночас слабким місцем) є так званий "Міст" (JavaScript Bridge). Бізнес-логіка застосунку виконується в окремому JavaScript-потоці середовища виконання (наприклад, Hermes), який асинхронно взаємодіє з нативними компонентами UI (OEM віджетами) та системними модулями (камерою, WebRTC) через постійну серіалізацію та десеріалізацію даних у форматі JSON. Хоча React Native є чудовим інструментом для стандартних застосунків (CRUD-систем), його архітектура стає вузьким місцем для застосунків реального часу. Передача великих обсягів подій або медіаданих через "Міст" створює відчутні затримки та знижує частоту кадрів (до явища UI Thread Starvation). Хоча нова архітектура (Fabric та JSI) частково усуває ці недоліки, інтеграція складних медіа-рушіїв залишається проблемною [11].

Flutter (від Google): Запропонований Google фреймворк Flutter кардинально відрізняється своїм архітектурним підходом від усіх існуючих аналогів. Flutter використовує сучасну об'єктно-орієнтовану мову програмування Dart. Головна архітектурна відмінність Flutter полягає в тому, що він принципово не використовує нативні UI-компоненти (віджети) Android чи iOS. Замість цього фреймворк малює весь користувацький інтерфейс самостійно, піксель за пікселем, за допомогою власного інтегрованого високопродуктивного графічного рушія (Skia, а в останніх версіях — Impeller), який звертається безпосередньо до графічних процесорів (GPU) через API OpenGL, Vulkan або Metal. Це забезпечує стабільну частоту кадрів (60-120 fps) на будь-яких пристроях та абсолютно ідентичний вигляд інтерфейсу на всіх платформах (рисунок 1.2) [12].

Обґрунтування вибору Flutter для дипломного проєкту На основі проведеного аналізу, для практичної реалізації мобільного застосунку у даній кваліфікаційній роботі було обрано фреймворк Flutter з огляду на наступні технічні переваги:

1. Продуктивність (AOT компіляція): Для випуску (Release) Dart-код компілюється наперед (Ahead-of-Time) у швидкий нативний машинний код для архітектур ARM/x86. Він не потребує інтерпретатора або "мостів" у процесі виконання, що критично важливо для швидкої обробки великої кількості мережевих станів WebRTC.

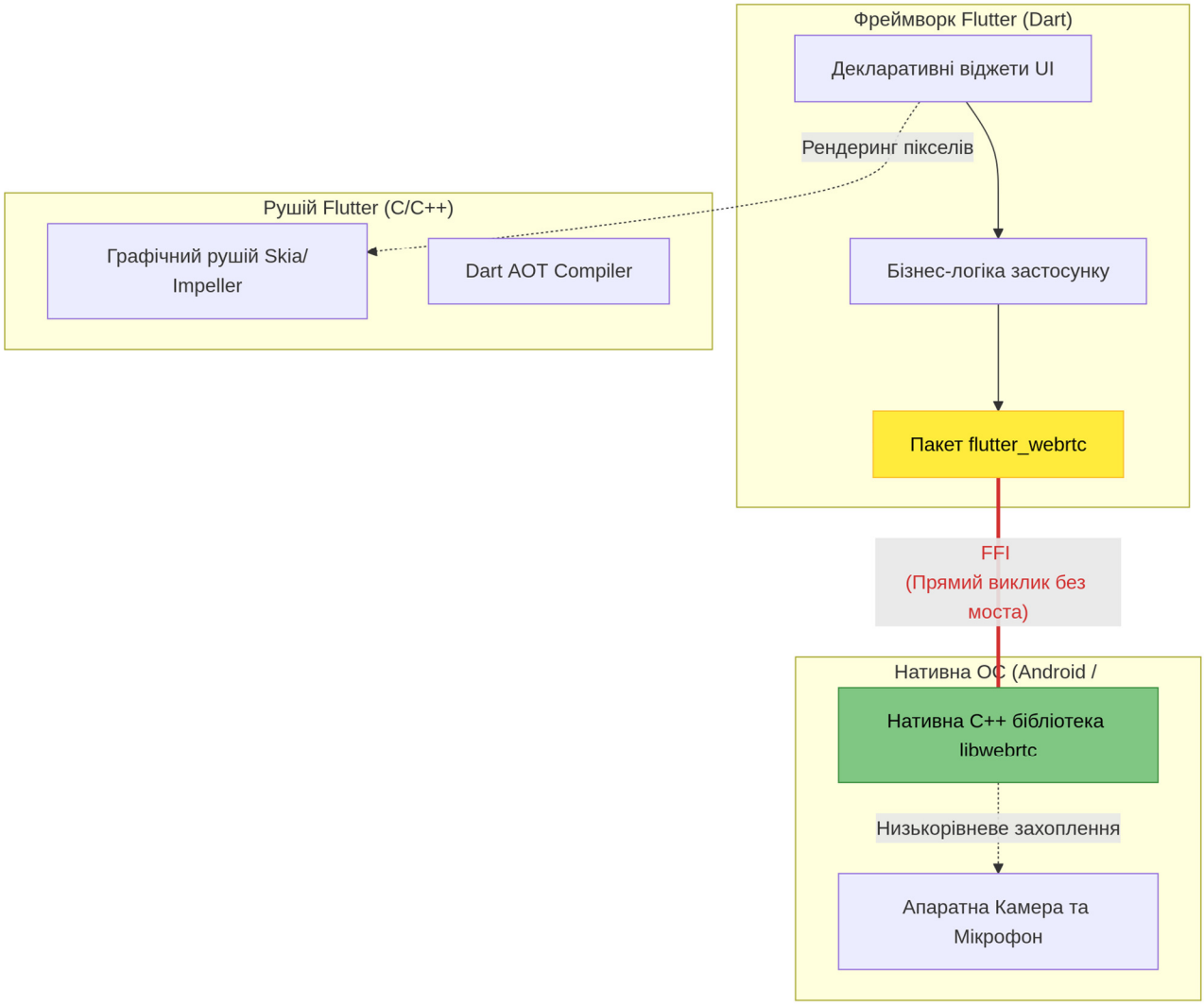


Рисунок 1.2 – Архітектурна модель фреймворку Flutter та взаємодія з нативним WebRTC

2. Механізм FFI (Foreign Function Interface): Екосистема Flutter містить потужний відкритий пакет flutter_webrtc. Його унікальність полягає в тому, що він не є простою обгорткою над веб-представленням (WebView). Він використовує Dart FFI для прямого синхронного виклику методів з оригінальної нативної C++ бібліотеки (libwebrtc) від Google. Це гарантує таку ж ефективність роботи з мультимедіа, як і при нативній розробці.

3. Швидкість розробки (Hot Reload) та багатонитковість: Декларативний підхід до UI дозволяє надзвичайно швидко реалізовувати складні інтерфейси стрімінгу (накладання чату поверх відео, анімації реакцій). Крім того, Dart використовує модель Ізолятів (Isolates) — незалежних потоків із власною пам'яттю,

що дозволяє виносити важкі завдання (наприклад, парсинг великих масивів JSON від сервера) з головного потоку інтерфейсу, запобігаючи "зависанню" екрана [13].

Організація професійного відеострімінгу вимагає не лише клієнтського застосунку, але й надійної, високопродуктивної серверної інфраструктури (Бекенду). Сервер повинен з мінімальною затримкою приймати потоки від стрімерів (Publishers), обробляти їх, забезпечувати горизонтальне масштабування та безпечно маршрутизувати медіадані тисячам глядачів (Subscribers). Для вирішення цих архітектурних завдань у даній роботі використовується рішення Ant Media Server (AMS) [14].

Ant Media Server — це спеціалізоване серверне програмне забезпечення для потокового відео, розроблене на мові Java, що фокусується виключно на технологіях ультранизької затримки на базі WebRTC. AMS здатний забезпечувати затримку на рівні 500 мілісекунд (0.5 с). Сервер постачається у двох редакціях: Community Edition (відкритий вихідний код, підтримка базового WebRTC як входу та трансмоксування у HLS/RTMP) та Enterprise Edition (пропрієтарна версія з повною підтримкою WebRTC-to-WebRTC стрімінгу, адаптивного бітрейту на сервері (Adaptive Bitrate), апаратного прискорення кодування на GPU від Nvidia/AMD та кластеризації).

Архітектурні особливості маршрутизації (MCU vs SFU) Під час організації багатокористувацьких трансляцій (наприклад, вебінарів або стрімів) критично важливо розуміти парадигму обробки потоків сервером. Історично сформувалися дві топології: MCU та SFU.

1. MCU (Multipoint Control Unit): У цій топології сервер діє як центральний мікшер. Він приймає вхідні відеопотоки від усіх учасників, повністю декодує їх, алгоритмічно мікшує в один єдиний відеокадр (наприклад, формуючи сітку 2x2 з обличчями) і знову кодує (транскодує) цей спільний потік для відправки клієнтам. Це мінімізує навантаження на мережу глядача (він завантажує лише 1 потік), але створює колосальне, експоненційно зростаюче навантаження на процесор сервера (CPU) (рисунок 1.3).

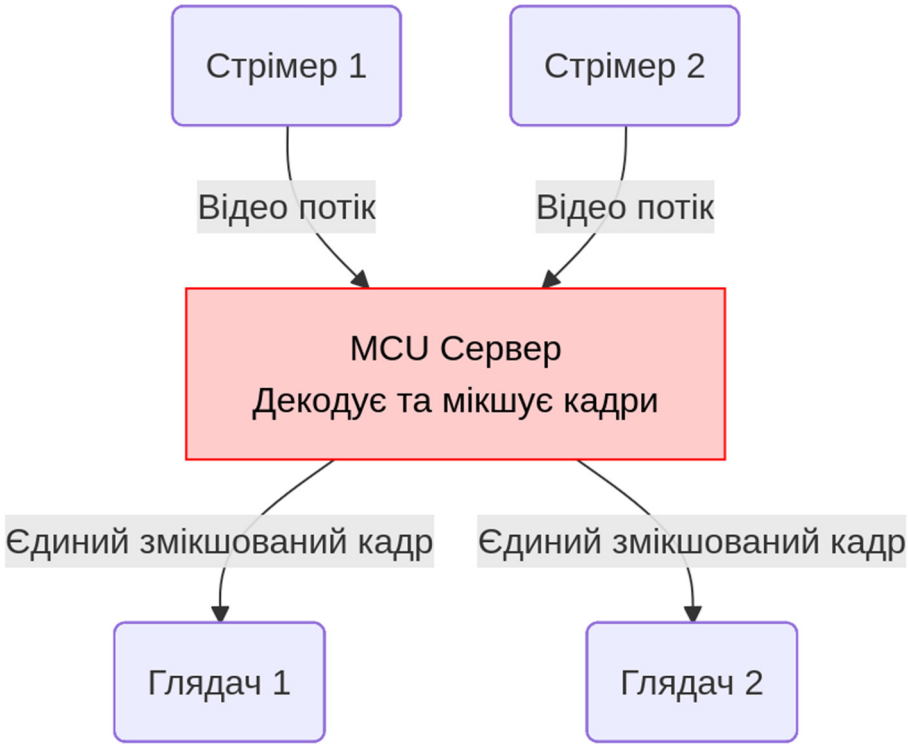


Рисунок 1.3 – Топологія SFU (Ant Media Server)

2. SFU (Selective Forwarding Unit): Більш сучасна, ресурсоефективна та масштабована архітектура, яка є базовою для Ant Media Server у режимі WebRTC (Рисунок 1.4). У топології SFU сервер не займається важким декодуванням та мікшуванням медіа. Він діє як інтелектуальний маршрутизатор пакетів. AMS отримує відеопотік від стрімера і просто вибірково пересилає (forwarding) його пакети всім авторизованим підписникам. При цьому клієнтський застосунок (розроблений нами на Flutter) отримує ці незалежні потоки і бере на себе завдання їх розташування на екрані пристрою (рендеринг). Цей підхід ліквідує "пляшкове горло" процесорних обчислень на сервері, дозволяючи одному вузлу обслуговувати десятки тисяч з'єднань.

REST API та механізми управління (Signaling) Для автоматизації процесів та управління життєвим циклом стрімів AMS надає потужний REST API (Application Programming Interface), захищений механізмами IP-фільтрації та аутентифікації за допомогою токенів JWT (JSON Web Tokens). API поділяється на кілька сервісів:

– Broadcast Rest Service: Управління трансляціями в реальному часі (створення кімнат, генерація Stream ID, примусова зупинка потоку, отримання статистики підключень).

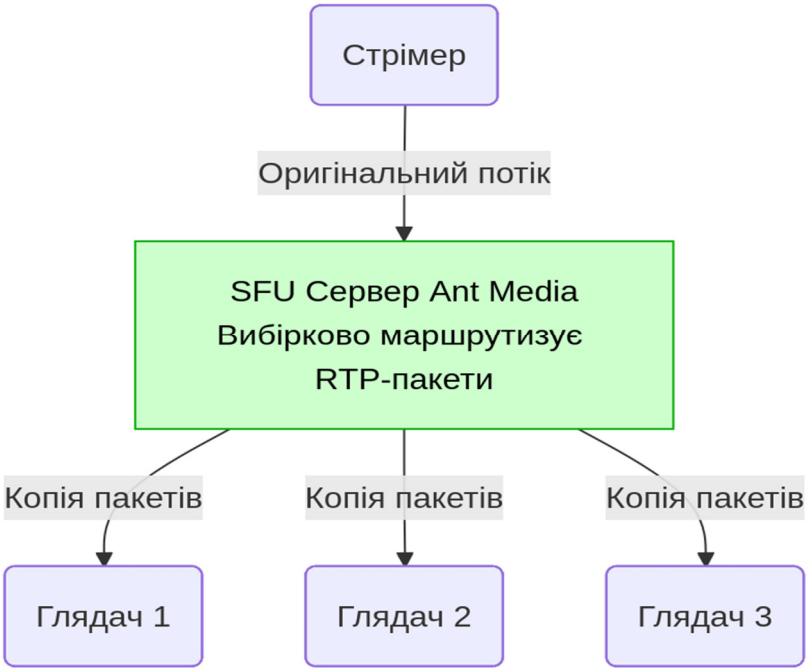


Рисунок 1.4 – Топологія SFU (Ant Media Server)

– VOD (Video on Demand) Rest Service: Управління записами архівних ефірів у форматах MP4/WebM.

– Management Service: Адміністрування застосунків (App Scopes) на рівні віртуальних хостів сервера.

Окрім REST API для управління метаданими, для безпосереднього встановлення P2P-з'єднання з сервером використовується протокол WebSockets. Саме через WebSocket реалізовано сигналізацію WebRTC в AMS: клієнт та сервер обмінюються JSON-командами (такими як publish, play, takeConfiguration, takeCandidate) для узгодження параметрів SDP та обміну ICE-кандидатами [15]. Для спрощення цієї взаємодії розробникам надаються готові SDK, зокрема Flutter SDK, який інкапсулює низькорівневу логіку сигналізації.

1.3 Постановка задачі розробки мобільного застосунку відеострімінгу

На основі проведеного аналізу предметної області, вивчення властивостей протоколів передачі даних та архітектури існуючих серверних рішень, основною метою даної кваліфікаційної роботи є: інженерне проектування, практична розробка,

оптимізація та комплексне тестування кросплатформного мобільного застосунку для відеострімінгу в реальному часі.

Цільовими операційними системами для розгортання клієнта є Android (версії 8.0 і вище) та iOS (версії 12.0 і вище). Розробка повинна здійснюватися мовою Dart з використанням UI-фреймворку Flutter. В якості серверного ядра (бекенду) для маршрутизації медіатрафіку за протоколом WebRTC використовуватиметься інфраструктура Ant Media Server.

Для досягнення поставленої мети необхідно виконати декомпозицію задачі та реалізувати такі функціональні вимоги:

1. Модуль аутентифікації та ідентифікації: Реалізація безпечного механізму взаємодії з REST API для генерації та валідації JWT-токенів, що унеможливлює несанкціоновану публікацію або перегляд закритих трансляцій (Stream Security).

2. Розподіл ролей (Publisher / Viewer): Застосунок має гнучко перемикатися між двома станами: режимом стрімера (доступ до камери, публікація медіа) та режимом глядача (відображення плеєра з мінімальною затримкою).

3. Підсистема публікації (Publishing Pipeline): Захоплення апаратних ресурсів смартфона (камера, мікрофон) з динамічним налаштуванням роздільної здатності (Media Constraints). Публікація потоку на AMS через WebRTC-транспорт.

4. Інтерактивне управління трансляцією: Реалізація функцій "гарячого" перемикання між основною (Back) та фронтальною (Front) камерами пристрою під час прямого ефіру. Забезпечення можливості програмного "м'якого" вимкнення (Soft Mute) аудіо- та відеодоріжок без руйнування відкритої RTP-сесії.

5. Підсистема відтворення (Playback & VOD): Отримання списку активних стрімів через Broadcast API. Реалізація рендерингу WebRTC відеопотоку на екрані мобільного пристрою з підтримкою апаратного прискорення (використання віджету RTCVideoView). Можливість відтворення збережених записів (Video-on-Demand).

Нефункціональні вимоги та архітектурні обмеження:

1. Обчислювальна продуктивність: Архітектура UI повинна мінімізувати кількість перемалювань (Rebuilds) віджетів під час оновлення кадрів відео. Застосунок має утримувати стабільну частоту кадрів інтерфейсу не нижче 60 FPS і

не спричиняти надмірного термального дроселювання (Thermal Throttling) процесора смартфона (SoC) при тривалих (понад 1 годину) трансляціях.

2. Обмеження наскрізної затримки (Latency): При тестуванні в локальній мережі або мережах 4G/LTE/5G з високою якістю покриття, наскрізна затримка (від фізичної події перед камерою до появи кадру на екрані глядача) математично описується як:

$$L_{total} = t_{cap} + t_{enc} + t_{net} + t_{sfu} + t_{dec} + t_{render} \quad (1.1)$$

де t_{cap} — час захоплення,

t_{enc} — кодування,

t_{net} — транспорт,

t_{sfu} — маршрутизація сервером,

t_{dec} — декодування,

t_{render} — малювання на екрані.

Значення Ltotal (Glass-to-Glass latency) не повинно перевищувати 800 мілісекунд.

3. Адаптивність до мережевих умов (Resilience): Застосунок повинен коректно обробляти зміну мережевого інтерфейсу (наприклад, перехід від Wi-Fi до стільникової мережі). Необхідна реалізація механізму ICE Restart для спроби безшовного відновлення медіа-сесії без втручання користувача.

4. Мережева безпека (Security Constraints): Передача керуючих команд (Signaling) повинна здійснюватися виключно через Secure WebSockets (wss://). Усі RTP-пакети з медіаданими повинні бути зашифровані протоколом SRTP з алгоритмами AES-128 або AES-256.

Вирішення описаних завдань вимагає ретельного архітектурного проектування клієнтського застосунку з використанням сучасних патернів (наприклад, BLoC або MVVM для розмежування бізнес-логіки WebRTC та UI), що буде розглянуто в наступних розділах роботи.

2 ПРОЕКТУВАННЯ ТА РОЗРОБКА МОБІЛЬНОГО ЗАСТОСУНКУ З ВИКОРИСТАННЯМ ANT MEDIA SERVER SDK

2.1 Архітектурна модель системи взаємодії мобільного клієнта та медіасервера

Проектування сучасного кросплатформного мобільного застосунку для потокового передавання відео вимагає чіткої архітектурної моделі, яка здатна забезпечити стабільну роботу з мережевими протоколами реального часу, високопродуктивну асинхронну обробку даних та суворе відокремлення бізнес-логіки від інтерфейсу користувача (UI). Архітектура розроблюваної системи базується на класичній клієнт-серверній моделі взаємодії з використанням мікросервісного підходу на стороні бекенду (базового ядра Ant Media Server) та компонентно-орієнтованої архітектури на стороні мобільного клієнта (фреймворк Flutter).

Взаємодія між мобільним застосунком та інфраструктурою Ant Media Server логічно та фізично поділяється на три незалежні площини (рівні абстракції), кожна з яких виконує вузькоспеціалізовані завдання:

1. Площина сигналізації (Signaling Plane): Використовується для ініціалізації, узгодження та моніторингу сесій зв'язку. Мобільний клієнт встановлює постійне, повнодуплексне з'єднання з сервером за захищеним протоколом WebSocket (ендпоінт wss://). Через цей канал відбувається асинхронний обмін сигнальними JSON-повідомленнями: публікація наміру розпочати трансляцію (команда publish), відправлення локальної пропозиції (SDP Offer) та отримання серверної відповіді (SDP Answer), а також безперервний обмін ICE-кандидатами для виявлення та встановлення оптимального мережевого маршруту між вузлами.

2. Медійна площина (Media Plane): Після успішного завершення процесу сигналізації та проходження перевірок ICE (Connectivity Checks) встановлюється з'єднання Peer-to-Peer (у контексті топології SFU — Peer-to-Server) за допомогою стеків протоколів WebRTC (RTP/RTCP). Цей виділений канал відповідає виключно за передачу зашифрованих (за допомогою SRTP) аудіо- та відеопакетів на базі транспорту UDP. Відокремлення медійної площини від сигнальної гарантує

передачу відео з ультранизькою затримкою, незалежно від навантаження на керуючі сервери [16].

3. Площина управління (Control & Management Plane): Здійснюється через синхронні HTTP/HTTPS запити до REST API сервера. Цей рівень є незалежним від WebRTC і використовується для авторизації користувачів, отримання метаданих активних трансляцій, управління кімнатами, конфігурації профілів адаптивного бітрейту (ABR) та роботи з архівом відео на вимогу (VOD).

На рівні внутрішньої структури самого мобільного застосунку у середовищі Flutter було обрано архітектурний патерн Clean Architecture (Чиста Архітектура) у поєднанні з реактивним менеджером станів BLoC (Business Logic Component). Такий підхід забезпечує жорсткий поділ відповідальності та високу тестованість програмного коду, унеможливлюючи так зване "протікання" логіки у віджети відображення (рисунок 2.1).

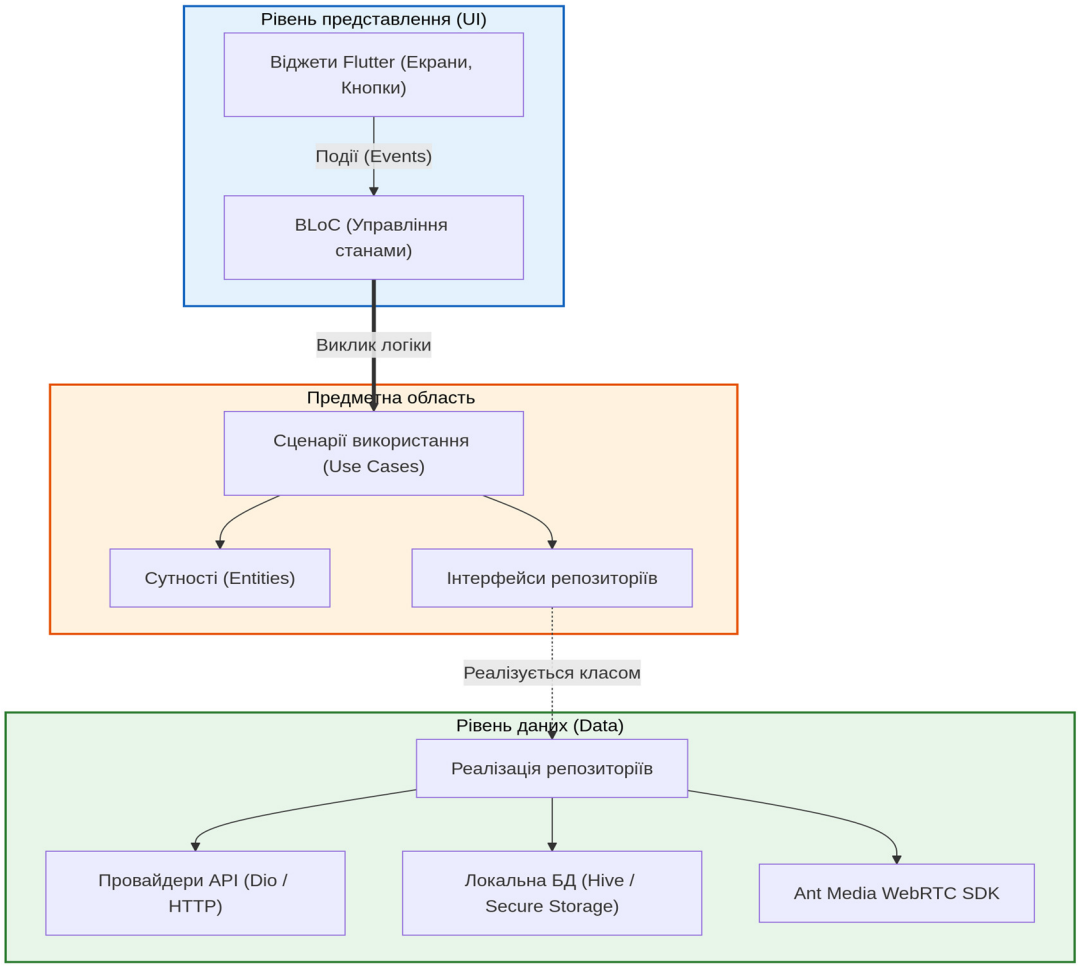


Рисунок 2.1 – Архітектурна модель мобільного застосунку на базі Clean Architecture та BLoC

Система поділяється на три взаємодіючі рівні:

– Presentation Layer (Рівень представлення): Відповідає виключно за відображення UI-віджетів (екрани стрімінгу, панелі управління, відеоплеєри) та перехоплення дій (подій) користувача. Віджети не містять жодної логіки WebRTC; вони лише відправляють події (Events) до блоків (BLoC) і слухають нові стани (States) для перемальовування екрана.

– Domain Layer (Предметна область): Найбільш захищений та незалежний рівень. Містить бізнес-логіку застосунку, зокрема сутності (Entities) трансляцій, абстрактні інтерфейси репозиторіїв (Repositories) та сценарії використання (Use Cases), наприклад, StartBroadcastUseCase, MuteMicrophoneUseCase або FetchVodsUseCase. Цей рівень не залежить від фреймворку Flutter чи сторонніх HTTP-бібліотек.

– Data Layer (Рівень даних): Реалізує безпосередню фізичну взаємодію з зовнішніми джерелами. Тут розміщуються провайдери API (імплементація HTTP-запитів через бібліотеку dio), класи доступу до локальних баз даних та адаптери-обгортки для Ant Media Flutter SDK, які трансформують DTO (Data Transfer Objects) від сервера у чисті сутності предметної області [17].

2.2 Інтеграція мобільного SDK та управління медіапотоками

Реалізація функціоналу відеострімінгу базується на глибокій інтеграції спеціалізованих інструментів для роботи з протоколом WebRTC у середовищі мобільної операційної системи. Оскільки нативний код для доступу до апаратних кодеків (Camera2 API/MediaCodec в Android та AVFoundation/VideoToolbox в iOS) суттєво відрізняється, використання кросплатформних абстракцій SDK дозволяє уніфікувати цей процес, зберігаючи при цьому нативну продуктивність.

Для налаштування клієнтського середовища до головного конфігураційного файлу pubspec.yaml додаються необхідні пакети залежностей. Основним модулем виступає пакет flutter_webrtc, який, використовуючи механізм Dart FFI (Foreign Function Interface), створює високошвидкісну обгортку над нативними C++ бібліотеками WebRTC (libwebrtc) від Google. Сигналізація реалізується або через

спеціалізований ant_media_flutter SDK, або через власну імплементацію WebSocket клієнта згідно з офіційною специфікацією сигналізації AMS.

Процес ініціалізації з'єднання є багатоетапним кінцевим автоматом (State Machine) і складається з наступних кроків:

1. Створення об'єкта RTCPeerConnection: Мобільний застосунок конфігурує параметри з'єднання, передаючи рушію WebRTC масив конфігурацій STUN та TURN серверів. Для роботи у простих домашніх мережах (Full Cone NAT) достатньо використання безкоштовних STUN-серверів від Google (stun:stun.l.google.com:19302). Однак для забезпечення гарантованого зв'язку через складні корпоративні фаєрволи, які блокують невідомий UDP-трафік (Symmetric NAT), у конфігурацію обов'язково передаються облікові дані TURN-серверів, що функціонують у межах інфраструктури Ant Media Server.

2. Встановлення WebSocket-з'єднання: Застосунок підключається до сигнального ендпоінту сервера. На цьому етапі реалізується механізм Heartbeat (відправка Ping/Pong повідомлень) для підтримання TCP-сокета відкритим у мобільних мережах, де балансувальники навантаження провайдерів можуть примусово розривати неактивні з'єднання.

3. SDP Переговори (Negotiation): Якщо клієнт виступає в ролі стрімера (Publisher), він ініціює створення SDP Offer. SDP (Session Description Protocol) — це суворо типізований текстовий документ, що описує медіа-можливості смартфона: підтримувані апаратні кодеки (H.264, VP8, Opus), їх профілі (наприклад, Baseline або High Profile для H.264), максимальну роздільну здатність, наявність механізмів PLC (заміщення втрачених пакетів) та криптографічні параметри для DTLS-рукостискання. Цей Offer відправляється на сервер через WebSocket. Ant Media Server аналізує його, порівнює з власними серверними можливостями, створює сумісний SDP Answer (відповідь) і повертає мобільному клієнту [18].

4. Збір ICE-кандидатів (Trickle ICE): WebRTC не чекає завершення обміну SDP для пошуку мережевих маршрутів. Використовується алгоритм Trickle ICE: паралельно з SDP-переговорами, пристрій звертається до ОС та STUN-серверів для збору своїх мережевих адрес (локальних Wi-Fi адрес, публічних IP оператора зв'язку, релейних адрес TURN). Кожен знайдений кандидат миттєво відправляється

на сервер окремим JSON-повідомленням. Це дозволяє скоротити час встановлення з'єднання (Connection Setup Time) з кількох секунд до мілісекунд.

Захоплення медіаданих з апаратних датчиків мобільного пристрою реалізується через стандартизований клас MediaDevices та його асинхронний метод getUserMedia(). Перед викликом цього методу застосунок використовує пакет permission_handler для запиту в ОС явних дозволів користувача на доступ до камери та мікрофона (Run-time permissions), що є обов'язковою вимогою безпеки сучасних версій Android та iOS.

Метод getUserMedia() приймає об'єкт конфігурації MediaStreamConstraints, де розробник декларативно вказує вимоги до відео- та аудіопотоку.

Таблиця 1.1 — Налаштування обмежень (Constraints) для захоплення відеопотоку на мобільному клієнті

Параметр конфігурації	Встановлене значення	Обґрунтування вибору для мобільної системи
minWidth / minHeight	640x480 (SD)	Забезпечує базову якість зображення при деградації мережі; запобігає падінню до нерозбірливих роздільних здатностей (наприклад, 320x240).
idealWidth / idealHeight	1280x720 (HD)	Формат 720p є галузевим стандартом для мобільного стрімінгу: забезпечує високу чіткість при помірному навантаженні на апаратний енкодер смартфона.
frameRate (ідеальний)	30 FPS	Достатньо для візуальної плавності рухів; використання 60 FPS на мобільних пристроях призводить до термального тротлінгу (перегріву) SoC під час довгих трансляцій.
audio: echoCancellation	true	Вмикає апаратне або програмне придушення акустичного відлуння, що є критично важливим при стрімінгу без використання гарнітури.

Після успішного захоплення ресурсів, операційна система повертає об'єкт MediaStream, який містить колекцію треків (об'єкти MediaStreamTrack — окремо для

відео та аудіо). Цей локальний потік у нашому застосунку виконує подвійну функцію: По-перше, відео-трек прив'язується до нативного UI-віджета RTCVideoView у середовищі Flutter. Цей віджет використовує платформені текстури (Texture або SurfaceView) для відображення "дзеркала" користувача на екрані смартфона з нульовою затримкою та підтримкою апаратного прискорення OpenGL/Metal. По-друге, аудіо- та відео-треки додаються до створеного раніше об'єкта RTCPeerConnection за допомогою методу addTrack(). Після цього внутрішній рушій WebRTC автоматично ініціює процес захоплення сирих кадрів, їх стиснення обраним кодеком та пакування у RTP-пакети для відправки на Ant Media Server [19]. Управління станами під час прямої трансляції є критичним елементом позитивного користувацького досвіду (UX). Архітектура розробленого застосунку дозволяє керувати медіа-треками "на льоту" (on-the-fly). У класичних, неоптимізованих застосунках для вимкнення мікрофона розробники часто видаляють трек зі з'єднання, що змушує WebRTC виконувати ресурсомістку процедуру повторного SDP-узгодження (Renegotiation).

У нашому рішенні застосовано алгоритм "М'якого вимкнення" (Soft Mute). Для функції вимкнення мікрофона застосунок не розриває з'єднання, а просто звертається до об'єкта локального AudioTrack і встановлює його властивість enabled = false. У результаті рушій WebRTC продовжує відправляти RTP-пакети на сервер, зберігаючи криптографічну сесію активною, але корисне навантаження (Payload) цих пакетів замінюється на абсолютну цифрову тишу. Аналогічно реалізується функція вимкнення камери (передача чорних кадрів) та перемикання між фронтальним і основним модулями камер за допомогою методу switchCamera(), який апаратно перенаправляє джерело пікселів у вже існуючий мережевий відеотрек [20].

2.3 Взаємодія з REST API Ant Media Server та організація безпеки

Крім потокової передачі медіаданих через UDP/WebRTC, повноцінний програмний продукт потребує надійних механізмів управління сесіями, отримання актуальних списків ефірів (Broadcasts) та строгої автентифікації клієнтів. Ця

керуюча взаємодія реалізується в застосунку через виклики багатого набору REST API (Application Programming Interface), наданих інфраструктурою Ant Media Server.

Безпека стрімінгової платформи є пріоритетним завданням проектування, оскільки несанкціонований доступ може призвести до крадіжки приватного контенту, несанкціонованої підміни відеопотоку (Hijacking) або перевантаження серверних обчислювальних потужностей внаслідок DDoS-атак. Ant Media Server пропонує адміністраторам два базові механізми захисту кінцевих точок: IP Filter та JWT.

У контексті розробки мобільного застосунку, механізм IP Filter (білі списки IP-адрес) є фундаментально неефективним. Мобільні пристрої не мають статичних адрес; вони постійно змінюють свої публічні IP під час переміщення користувача та перемикання (вертикального хендоверу) між домашнім Wi-Fi та мережами стільникових операторів (3G/4G/5G). З огляду на це, єдиним надійним стандартом криптографічної безпеки в системі було обрано автентифікацію на основі JWT (JSON Web Tokens).

JWT — це відкритий інтернет-стандарт (визначений у документі RFC 7519), який пропонує компактний і автономний спосіб безпечної передачі тверджень (Claims) між клієнтом та сервером у вигляді зашифрованого об'єкта JSON. Структурно токен складається з трьох частин, розділених крапкою (.):

1. Header (Заголовок): вказує тип токена (typ: JWT) та алгоритм криптографічного хешування (найчастіше це симетричний HMAC SHA-256 або асиметричний RSA).

2. Payload (Корисне навантаження): містить безпосередньо дані авторизації (Claims). Наприклад, ідентифікатор потоку (streamId), до якого користувачу дозволено доступ, тип дії (публікація або перегляд), та обов'язкову мітку часу закінчення дії токена (exp - expiration time), що захищає від атак повторного використання (Replay attacks).

3. Signature (Криптографічний підпис): гарантує цілісність даних, підтверджуючи, що корисне навантаження токена не було модифіковане зловмисником під час передачі через мережу (атака Man-in-the-Middle) [21].

Математично процес генерації підпису JWT (наприклад, алгоритмом HS256) на стороні Auth-сервера описується такою формулою:

$$Signature = HMACSHA256(base64UrlEncode(header) + . + base64UrlEncode(payload), secret) \quad (2.1)$$

де: header — заголовок JWT-токена, що містить інформацію про тип токена та алгоритм підпису;

payload — корисне навантаження токена, яке містить набір тверджень (claims), зокрема ідентифікатор потоку, права доступу та час закінчення дії токена;

base64UrlEncode() — функція кодування даних у форматі Base64Url;

secret — секретний ключ, відомий лише серверу авторизації та серверу перевірки токенів;

HMACSHA256() — криптографічна функція обчислення підпису на основі алгоритму HMAC із використанням хеш-функції SHA-256;

Signature — криптографічний підпис JWT-токена, який забезпечує контроль цілісності та автентичності переданих даних.

Отриманий підпис додається до закодованих частин заголовка та корисного навантаження, утворюючи завершений JWT-токен, який використовується для авторизації запитів до сервера Ant Media Server.

Під час процесу авторизації мобільний застосунок звертається до стороннього авторизаційного сервера (Auth-сервера) або безпосередньо до Management API AMS з логіном та паролем. У разі успішної перевірки клієнт отримує згенерований JWT. Цей токен надалі ін'єктується на рівні Data Layer (за допомогою механізму Interceptors у бібліотеці dio) у вигляді стандартизованого HTTP-заголовка Authorization: Bearer <token> до всіх подальших запитів. Крім того, при ініціалізації трансляції відео, цей токен передається як параметр у URL WebSocket-з'єднання. Сервер Ant Media самостійно хешує отриманий заголовок і корисне навантаження своїм секретним ключем; якщо результати збігаються з підписом — трансляція дозволяється [22].

Бізнес-логіка застосунку передбачає наявність головного екрана — "Стрічки активних ефірів" (за аналогією з популярними платформами Twitch або YouTube

Live). Для отримання актуальних і достовірних даних про трансляції застосунок взаємодіє з Broadcast Rest Service сервера.

Мобільний клієнт генерує асинхронний HTTP GET-запит до ендпоінту: https://[server_domain]/[application_name]/rest/v2/broadcasts/list/{offset}/{size}

Використання параметрів offset (зсув) та size (кількість елементів) реалізує механізм пагінації (розбиття масиву даних на сторінки). У мобільній розробці пагінація є критично необхідним архітектурним патерном: вона запобігає завантаженню тисяч записів одночасно, що гарантовано призвело б до вичерпання лімітів оперативної пам'яті смартфона (Out Of Memory Exception) та надмірних витрат мобільного трафіку користувача.

Сервер повертає масив JSON-об'єктів, кожен з яких містить метадані трансляції: streamId, назву ефіру (name), поточний статус (наприклад, broadcasting або finished), а також статистику підключень (webRTCViewerCount). На рівні Data Layer створені DTO-моделі у Flutter десеріалізують цей JSON у строго типізовані Dart-об'єкти. Далі ці об'єкти передаються через BLoC на рівень представлення (UI), де динамічно будується безкінечний список (з використанням оптимізованих віджетів ListView.builder або SliverList, які рендерять лише ті елементи, що знаходяться у видимій зоні екрана) [23].

Якщо користувач бажає переглянути запис вже завершеної трансляції, застосунок звертається до VOD Rest Service. За умови увімкненого запису на сервері (Recording), Ant Media Server автоматично зберігає прямі ефіри у форматі MP4 на жорсткий диск. GET-запит до ендпоінту .../rest/v2/vods/list/ повертає список файлів "на вимогу". Для відтворення цих записів застосунок використовує класичний нативний відеоплеєр (інтегрований через пакет video_player). Таке архітектурне рішення обґрунтоване тим, що протокол WebRTC оптимізований виключно для трансляцій у реальному часі без буферизації. Для VOD (відео на вимогу) значно доцільніше та стабільніше використовувати технології адаптивного HTTP-стрімінгу (такі як HLS або прогресивне завантаження MP4), які підтримують кешування, перемотування (Seeking) та буферизацію "наперед" для запобігання зупинці відео при поганому інтернеті [24].

Для наочного подання логіки обміну інформацією між мобільним клієнтом, керуючим бекендом та медіасервером було розроблено діаграми потоків даних (Data Flow Diagram, DFD). На рисунку 2.2 наведено DFD-діаграму підсистеми управління трансляціями та синхронізації статусів ефірів, а на рисунку 2.3 – DFD-діаграму підсистеми обробки обкладинок і формування стрічки активних трансляцій.

Як видно з наведеної схеми, процес ініціалізації трансляції розпочинається із формування мобільним клієнтом запиту (StreamRequest), який маршрутизується до керуючого сервісу StreamService. Даний модуль виконує синхронний REST-запит до API Ant Media Server для реєстрації нового об'єкта трансляції. Отримавши підтвердження, система фіксує початковий стан трансляції (статус CREATED) у реляційній базі даних та генерує криптографічний JWT-токен. Згенерований токен разом із параметрами доступу повертається мобільному клієнту (StreamResponse) для подальшої авторизації WebRTC-сесії.

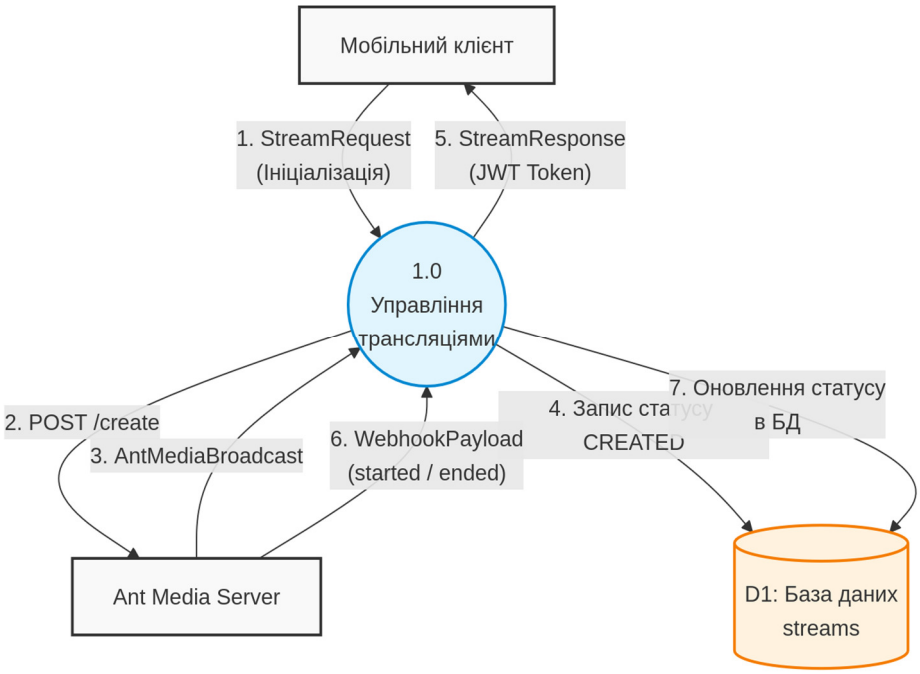


Рисунок 2.2 – DFD-діаграма процесу створення трансляції та оновлення її стану

Окрім синхронної взаємодії, архітектурою передбачено асинхронний інформаційний потік (потоки 6-7 на рис. 2.3), що базується на механізмі зворотних викликів (Webhooks). Цей механізм забезпечує автоматичне отримання керуючим

сервером сповіщень (WebhookPayload) від Ant Media Server про фактичний початок передачі медіаданих або завершення ефіру.

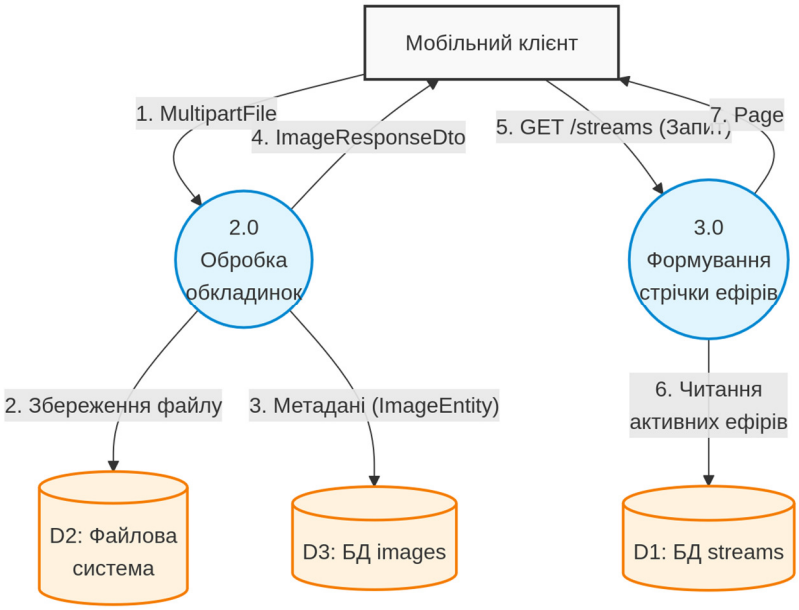


Рисунок 2.3 – DFD-діаграма підсистеми керування медіаконтентом та відображення активних ефірів

Завдяки такій інтеграції досягається строга консистентність даних: статуси трансляцій у базі даних миттєво оновлюються, що дозволяє коректно формувати стрічку активних ефірів для інших користувачів системи.

2.4 Організація інформаційного забезпечення та збереження локальних даних

Незважаючи на те, що основний масив медіаданих та відповідних метаданих зберігається в базах даних на стороні сервера, сучасний мобільний застосунок має реалізовувати механізми локального кешування інформації. Це є критично важливим для часткового забезпечення концепції «Offline-first», збереження станів авторизаційних сесій користувача, мінімізації кількості надлишкових мережевих запитів та підвищення загального рівня чуйності (Responsiveness) інтерфейсу.

Інформаційне забезпечення розроблюваного мобільного клієнта базується на трирівневій гібридній архітектурі сховищ, яка дозволяє диференціювати дані за ступенем їхньої чутливості та структурної складності (рисунок 2.4).

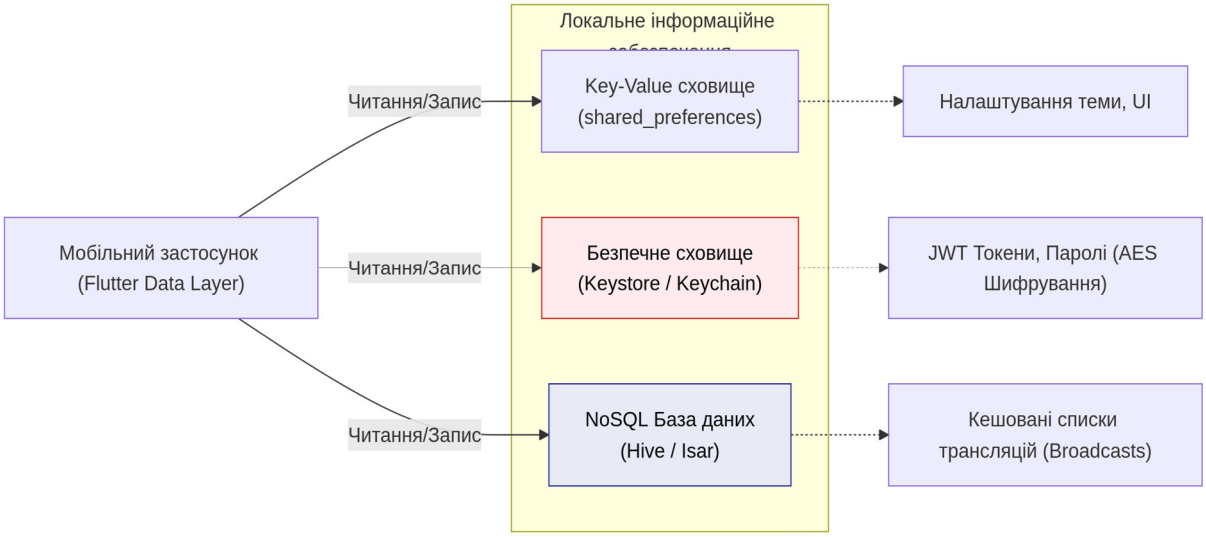


Рисунок 2.4 – Трирівнева архітектура локального інформаційного забезпечення мобільного клієнта

Згідно із запропонованою структурою, кожен рівень виконує специфічні функції:

1. Сховище налаштувань (Key-Value Storage): Для збереження нечутливих конфігураційних даних застосовується пакет shared_preferences. Цей інструмент функціонує як універсальна кросплатформна абстракція над нативними механізмами операційних систем: NSUserDefaults в екосистемі iOS та SharedPreferences в Android. У даному сховищі зберігаються прості примітиви (Boolean, Integer, String), зокрема: обрана користувачем тема оформлення, роздільна здатність трансляції за замовчуванням або прапорець проходження етапу ознайомлення (Onboarding).

2. Безпечне сховище (Secure Storage): Оскільки авторизаційні дані є критично чутливою інформацією, їх збереження у відкритому вигляді є неприпустимим згідно зі стандартами мобільної безпеки (OWASP Mobile). Для вирішення цієї задачі в архітектуру Data Layer інтегровано пакет flutter_secure_storage. Він забезпечує взаємодію з апаратними механізмами шифрування: Android Keystore System та Apple Keychain. Використання алгоритму AES (Advanced Encryption Standard) гарантує захист JWT-токенів та паролів навіть у випадку отримання зловмисником несанкціонованого доступу до файлової системи пристрою (Root-доступ).

3. Локальна база даних (Local Database): Для обробки складно структурованих масивів даних (історія переглядів, списки обраних трансляцій, профілі стрімерів) впроваджено сучасне NoSQL-рішення (рушій Hive або Isar). Вибір на користь об’єктно-орієнтованих баз даних замість традиційних реляційних (SQLite) зумовлений високою швидкістю роботи в середовищі Dart. Використання бінарної серіалізації дозволяє виконувати операції читання/запису з часовою складністю O(1). Завдяки локальному кешуванню об’єктів, застосунок відображає останній актуальний стан медіаконтенту навіть за відсутності мережевого з’єднання, після чого архітектурний паттерн BLoC ініціює фонове оновлення даних при відновленні зв’язку.

3 ПРАКТИЧНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ МОБІЛЬНОГО ЗАСТОСУНКУ ДЛЯ ВІДЕОСТРІМІНГУ

3.1 Розгортання інфраструктури та інтеграція базових сервісів

Практична імплементація системи розпочалася з розгортання комплексної серверної інфраструктури, необхідної для повноцінної, безпечної та відмовостійкої роботи мобільного клієнта. Архітектурно серверна частина була розділена на два незалежні вузли: медіасервер для обробки потокового відео та керуючий бекенд для реалізації бізнес-логіки.

Для управління медіа-трафіком (протокол WebRTC) інфраструктуру медіасервера було розгорнуто з використанням технології контейнеризації Docker. Відмова від класичних віртуальних машин на користь Docker-контейнерів дозволила мінімізувати накладні витрати на віртуалізацію (Hypervisor Overhead) та оптимізувати використання апаратних ресурсів серверного вузла.

Процес розгортання екземпляра Ant Media Server було реалізовано як "інфраструктуру як код" (Infrastructure as Code) та автоматизовано за допомогою інструменту Docker Compose. Конфігураційний файл docker-compose.yml має такий вигляд:

```
services:
  antmedia:
    image: antmedia/community:latest
    restart: always
    privileged: true
    stdin_open: true
    tty: true
    network_mode: "host"
    volumes:
      - antmedia_volume:/usr/local/antmedia/
volumes:
  antmedia_volume:
```

Ключовим архітектурним рішенням у наведеній конфігурації є використання параметра network_mode: "host". Цей режим дозволяє контейнеру безпосередньо використовувати мережевий стек базової операційної системи (хоста), повністю оминаючи механізми трансляції мережевих адрес (NAT) самого механізму Docker. Такий підхід є критично важливим для коректної роботи протоколу WebRTC: він

усуває проблеми з маршрутизацією високоінтенсивного UDP-трафіку та забезпечує безперешкодний доступ до широкого діапазону динамічних портів, необхідних для встановлення P2P-з'єднань, що в результаті мінімізує наскрізну затримку (latency) відеопотоку. Додатково, використання іменованого тому (antmedia_volume) гарантує персистентність (надійне збереження) конфігураційних файлів та записаних трансляцій (VoD) незалежно від життєвого циклу самого контейнера.

Для управління бізнес-логікою, синхронізації статусів трансляцій та забезпечення контролю доступу було розроблено серверну частину (Backend) на базі фреймворку Spring Boot (Java). Цей керуючий прошарок виконує роль посередника між мобільним клієнтом та медіасервером. До його основних функцій входять:

– Генерація захищених JWT-токенів: для запобігання несанкціонованому доступу до публікації потоків сервер генерує тимчасові токени з підписом HMAC-SHA, які мобільний клієнт повинен пред'явити під час ініціалізації з'єднання.

– Обробка Webhooks: Spring-сервер приймає асинхронні HTTP-запити від Ant Media Server (події liveStreamStarted, liveStreamEnded), що дозволяє в реальному часі оновлювати статуси трансляцій у базі даних і надавати глядачам лише актуальний список активних ефірів.

– Надання REST API: забезпечення стандартизованої взаємодії з мобільним застосунком (реєстрація потоків, завантаження обкладинок/мініатюр тощо).

Після налаштування серверної частини розробка сфокусувалася на створенні мобільного застосунку за допомогою кросплатформного фреймворку Flutter. Головною метою було створення не лише функціонального, але й естетично привабливого, сучасного користувацького інтерфейсу (UI), який забезпечує безшовний досвід (UX) як для творців контенту (стрімерів), так і для глядачів.

Внутрішня архітектура мобільного застосунку реалізована з дотриманням принципів чистої архітектури (Clean Architecture) і складається з трьох ключових рівнів:

1. Рівень представлення (UI Layer): побудований з використанням сучасних віджетів Flutter із підтримкою темної теми (Dark Mode) для концентрації уваги користувача на відеоконтенті.

2. Рівень управління станом (State Management): реалізований на базі бібліотеки Riverpod, що забезпечує реактивне оновлення інтерфейсу без блокування головного потоку при зміні стану трансляції чи втраті мережі.

3. Рівень даних та мережевої взаємодії (Data Layer): використовує HTTP-клієнт dio у поєднанні з генератором коду retrofit для суворо типізованої взаємодії з REST API бекенду Spring Boot, а також інтегрує нативні бібліотеки (flutter_webrtc) для прямого з'єднання з Ant Media Server.

Загальну архітектурну схему інформаційної взаємодії компонентів розробленої системи наведено на рисунку 3.1.

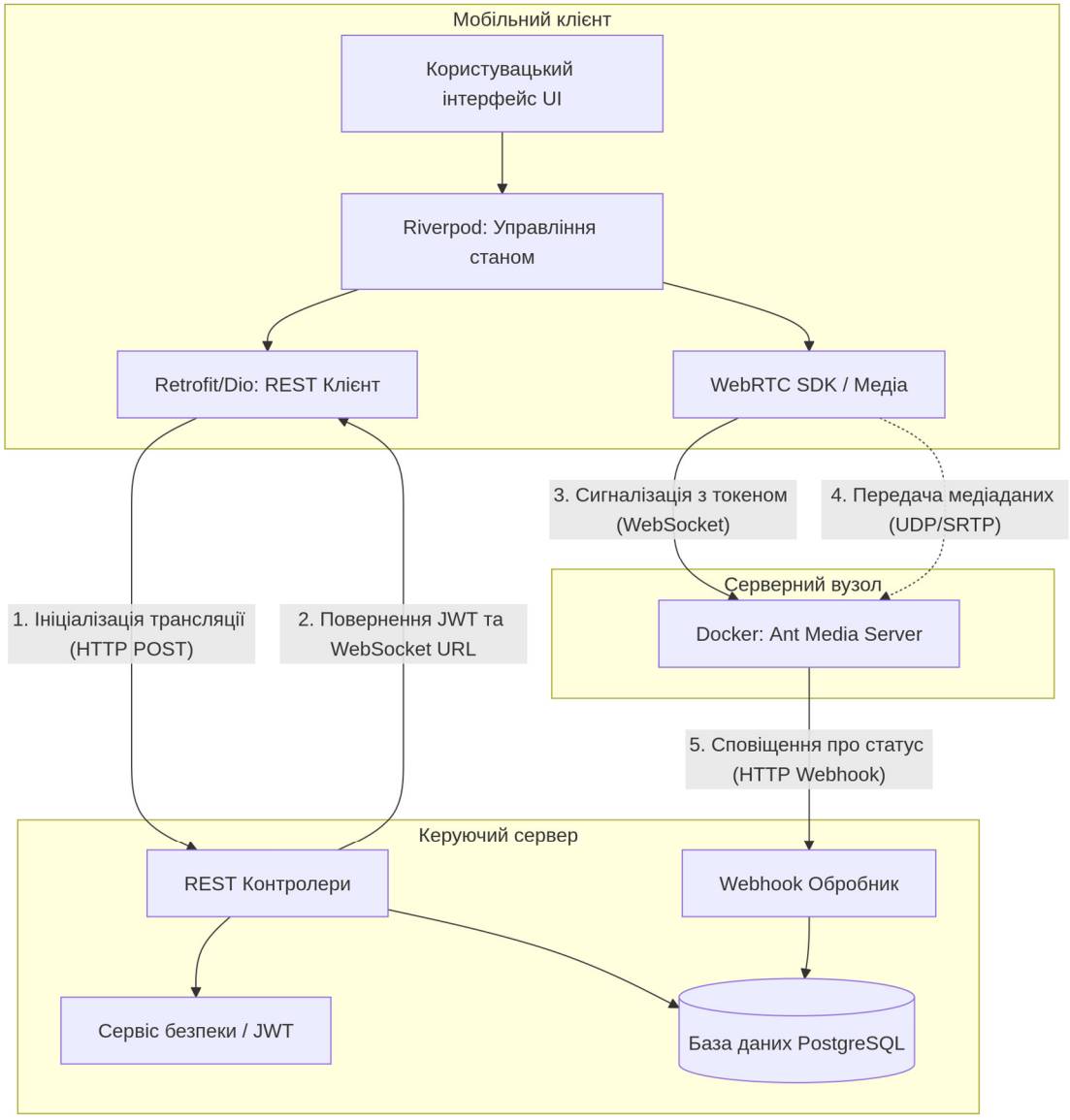


Рисунок 3.1 — Архітектурна модель взаємодії мобільного клієнта (Flutter), керуючого сервера (Spring Boot) та медіасервера (Docker контейнер)

Така розгалужена архітектура дозволила зняти навантаження з медіасервера, делегувавши завдання авторизації та ведення реєстру трансляцій окремому бекенду, що суттєво підвищує масштабованість та безпеку всієї системи.

Окремим критично важливим етапом практичної реалізації стала конфігурація мобільної платформи для забезпечення доступу до апаратних ресурсів смартфона. Оскільки протокол WebRTC передбачає пряму роботу з камерою та мікрофоном пристрою, у файлі маніфесту Android-застосунку (AndroidManifest.xml) було чітко визначено необхідні апаратні вимоги та дозволи:

```
<uses-feature android:name="android.hardware.camera" />
<uses-feature android:name="android.hardware.camera.autofocus" />

<uses-permission android:name="android.permission.CAMERA" />
<uses-permission   android:name="android.permission.RECORD_AUDIO"
/>
<uses-permission
android:name="android.permission.ACCESS_NETWORK_STATE" />
<uses-permission
android:name="android.permission.CHANGE_NETWORK_STATE" />
<uses-permission
android:name="android.permission.MODIFY_AUDIO_SETTINGS" />
<uses-permission android:name="android.permission.INTERNET" />
```

Така конфігурація гарантує, що операційна система Android надасть застосунку доступ до системних API для захоплення аудіо- та відеопотоків, а також дозволить змінювати параметри аудіо-виходу (наприклад, перемикання на гучний зв'язок) під час активної трансляції.

Інтеграція функціоналу Ant Media Server у Flutter-проєкт була реалізована через офіційну бібліотеку ant_media_flutter. Процес підключення SDK до проєкту здійснювався за допомогою пакетного менеджера Dart за командою [26]:

flutter pub add ant_media_flutter

Після виконання команди у конфігураційному файлі залежностей проєкту pubspec.yaml було зафіксовано версію бібліотеки (на момент розробки — ^1.5.4), що забезпечує стабільність збірки та сумісність із поточним API медіасервера:

```
dependencies:
  flutter:
    sdk: flutter
  ant_media_flutter: ^1.5.4
```

Для використання методів SDK у програмному коді здійснюється імпорт відповідного простору імен:

import 'package:ant_media_flutter/ant_media_flutter.dart';

Це дозволило використовувати високорівневі абстракції SDK для управління сигналізацією WebRTC, встановлення з'єднань через WebSocket та управління станами публікації та відтворення потоків.

3.2 Проектування та програмна реалізація користувацького інтерфейсу

Головним пріоритетом при розробці мобільного застосунку було створення сучасного, мінімалістичного та інтуїтивно зрозумілого інтерфейсу (UI), який забезпечує безперешкодний користувацький досвід (UX). Використання кросплатформного фреймворку Flutter дозволило реалізувати декларативний підхід до побудови UI, де весь екран формується як ієрархічне дерево віджетів (Widget Tree), а стан застосунку (State) безпосередньо визначає їхнє візуальне відображення. На відміну від імперативного підходу, інтерфейс не модифікується вручну крок за кроком, а автоматично перебудовується при зміні базових даних.

Для управління цими станами було інтегровано архітектурне рішення на базі Riverpod. Завдяки цьому, зміна бізнес-логіки (наприклад, встановлення WebRTC-з'єднання, оновлення лічильника глядачів або зміна статусу мікрофона) ініціює перемальовування (rebuild) лише тих конкретних віджетів, які від неї залежать. Це дозволило уникнути "зависань" інтерфейсу та забезпечити стабільну частоту оновлення кадрів (60 FPS) навіть під час паралельної обробки важкого відеопотоку.

Для забезпечення високого естетичного рівня було обрано концепцію «Dark Mode» (темна тема) з яскравими акцентними елементами, що повністю відповідає сучасним трендам стрімінгових платформ. Такий вибір обґрунтований не лише візуальною привабливістю, але й низкою практичних переваг:

1. Зниження когнітивного навантаження: темний фон дозволяє зробити максимальний візуальний акцент саме на відеоконтенті, не відволікаючи користувача на яскраві панелі інтерфейсу.

2. Зменшення навантаження на зір: оскільки споживання стрімінгового контенту часто відбувається в умовах слабкого освітлення, темний інтерфейс зменшує втому очей глядачів.

3. Енергоефективність: на сучасних мобільних пристроях з OLED та AMOLED дисплеями використання темних (чорних) кольорів суттєво заощаджує заряд акумулятора, оскільки пікселі у цих зонах фізично вимикаються, що є критично важливим параметром для ресурсомістких застосунків, які працюють з відеокамерою та мережею.

Особливу увагу при проектуванні UX було приділено ергономіці управління трансляцією. Оскільки під час стрімінгу користувач часто тримає мобільний пристрій однією рукою або закріплює його на штативі, основні інтерактивні елементи керування (кнопки запуску/завершення ефіру, перемикання камер, вимкнення мікрофона) були згруповані у нижній частині екрана. Це відповідає правилу "зони великого пальця" (Thumb Zone) у мобільній навігації та мінімізує ризик випадкових натискань або перекриття об'єктива камери рукою.

Для підвищення інтуїтивності взаємодії в систему також інтегровано механізми візуального зворотного зв’язку. Використання мікроанімацій (таких як пульсація індикатора «Прямий ефір», плавна поява панелей керування та індикатори завантаження під час ініціалізації API-запитів) не лише покращує сприйняття, але й чітко інформує користувача про поточний статус фонових мережевих процесів.

Головний екран (Рисунок 3.2) розроблено як динамічну стрічку активних ефірів та єдину точку входу в застосунок. Архітектурно екран розділений на два ключові блоки: зону швидкого доступу до створення власного ефіру (CreateStreamCard) та стрічку поточних трансляцій (ActiveStreamsSection).

Для забезпечення кращого користувацького досвіду (UX), кореневим віджетом екрана виступає RefreshIndicator, що дозволяє оновлювати список трансляцій звичним жестом «pull-to-refresh» (свайп вниз).

Особливістю реалізації відображення списку є використання компонента ListView.separated, який гарантує «ліниве» (lazy) завантаження карток. Це суттєво оптимізує використання оперативної пам'яті мобільного пристрою при великій

кількості стрімів, оскільки рендеряться лише ті елементи, які безпосередньо знаходяться в області видимості екрана (Viewport).

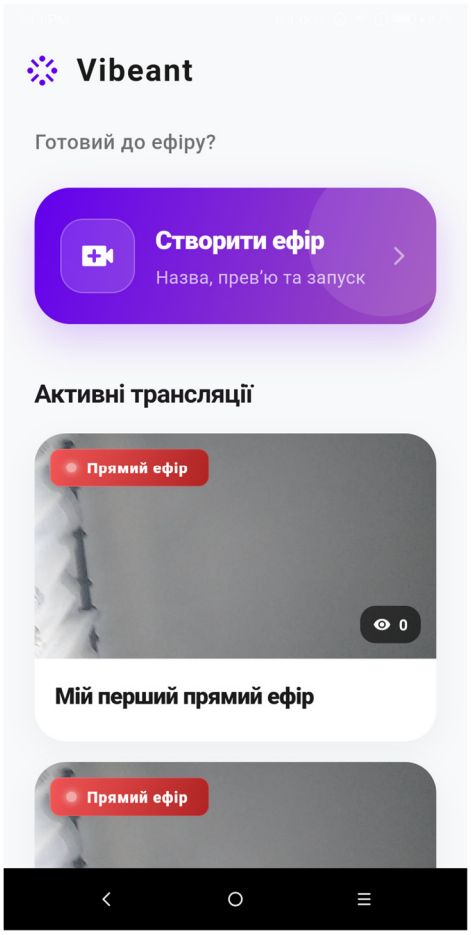


Рисунок 3.2 — Головний екран із переліком активних трансляцій

Кожна картка трансляції (StreamCard) є складним віджетом, що поєднує в собі зображення-прев'ю, текстові метадані та індикатори стану реального часу. Для типізованої передачі цих даних між шарами застосунку використовується спеціальна DTO-модель, згенерована за допомогою пакету json_serializable.

Фрагмент коду 3.1 — Опис моделі даних для UI

```
@JsonSerializable()
class ActiveStreamDto {
  const ActiveStreamDto({
    this.streamId,
    this.name,
    this.status,
    this.previewUrl,
    this.playUrlHls,
    this.startedAt,
    this.viewerCount,
```

```
  });
  factory ActiveStreamDto.fromJson(Map<String, Object?> json) =>
_$ActiveStreamDtoFromJson(json);
  final int? streamId;
  final String? name;
  final ActiveStreamDtoStatus? status;
  final String? previewUrl;
  final String? playUrlHls;
  final DateTime? startedAt;
  final int? viewerCount;
  Map<String, Object?> toJson() => _$ActiveStreamDtoToJson(this);
}
```

Така модель дозволяє UI-шару миттєво реагувати на зміну кількості глядачів або статусу потоку. Управління станом списку реалізовано через бібліотеку Riverpod. Створений провайдер ActiveStreams інкапсулює логіку взаємодії з API та автоматично обробляє стани завантаження (Loading), помилки (Error) та успішного отримання даних (Data).

Фрагмент коду 3.2 — Реактивний провайдер отримання списку трансляцій

```
@riverpod
class ActiveStreams extends _$ActiveStreams {
  @override
  FutureOr<List<ActiveStreamDto>> build() async {
    return _fetchStreams();
  }

  Future<List<ActiveStreamDto>> _fetchStreams() async {
    final api = ref.read(antMediaApiProvider);
    final response = await
api.streamManagement.getActiveStreams(page: 0, size: 20);
    return response.content ?? [];
  }

  Future<void> refresh() async {
    state = const AsyncLoading();
    state = await AsyncValue.guard(() => _fetchStreams());
  }
}
```

На рівні візуального представлення віджет ActiveStreamsSection підписується на цей провайдер і використовує конструкцію streamsAsync.when() для безпечного рендерингу інтерфейсу. У разі відсутності ефірів користувачеві демонструється заглушка EmptyStreamsCard, а при успішному завантаженні — список клікабельних

карток, натискання на які ініціює перехід до вікна плеєра за допомогою маршрутизатора GoRouter (context.push('/watch/${stream.streamId}')).

Дані для відображення на мобільному клієнті підтягуються з серверної частини (Backend), реалізованої на Spring Boot. Сервер надає REST API ендпойнт для отримання списку трансляцій із підтримкою пагінації.

З метою надання користувачам лише актуального контенту, вибірка з бази даних відбувається з суворим фільтруванням за статусом потоку. На рівні Data Access Object (DAO) це реалізовано за допомогою Spring Data JPA:

Фрагмент коду 3.3 — Серверна логіка вибірки активних трансляцій

```
@GetMapping
public ResponseEntity<Page<ActiveStreamDto>> getActiveStreams(
        @RequestParam(defaultValue = "0") int page,
        @RequestParam(defaultValue = "10") int size) {
    Pageable pageable = PageRequest.of(page, size);
    return
ResponseEntity.ok(streamService.getActiveStreams(pageable));
}
@Repository
public interface StreamRepository extends
JpaRepository<StreamEntity, Long> {
    Page<StreamEntity> findAllByStatusOrderByStartedAtDesc(
        StreamStatus status,
        Pageable pageable
    );
}
```

Завдяки такій наскрізній архітектурі — від JPA-репозиторію на бекенді, через REST API, до реактивного Riverpod-провайдера та «лінивого» ListView у Flutter — забезпечується висока продуктивність головного екрана, миттєве завантаження даних та плавний користувацький досвід без "фрізів" інтерфейсу.

Екран підготовки (Рисунок 3.3) є критично важливим етапом переходу користувача від ролі глядача до ролі творця контенту. Він фокусується на попередньому налаштуванні метаданих ефіру та встановленні візуальної ідентичності трансляції — обкладинки (Thumbnail).

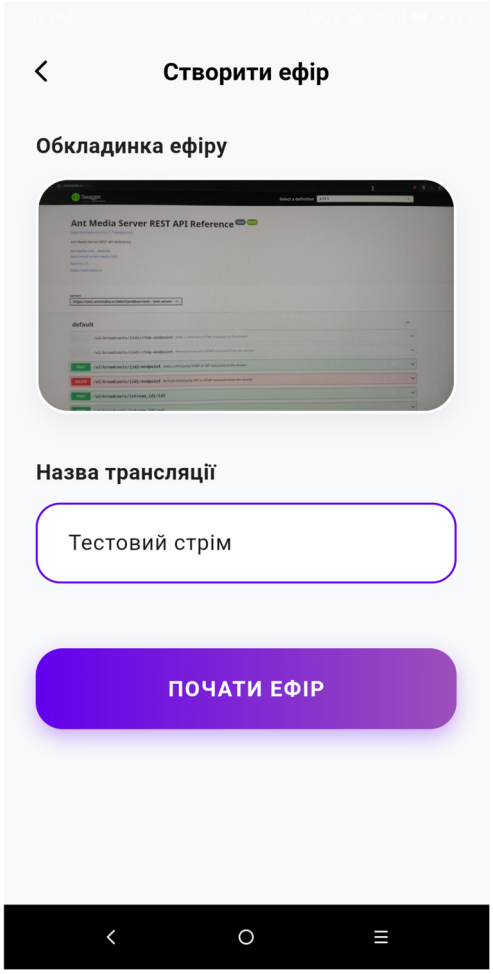


Рисунок 3.3 — Екран створення трансляції та конфігурації обкладинки ефіру

Верхня частина екрана містить текстові поля з напівпрозорим фоном, що забезпечує читабельність тексту поверх відео. Особливу увагу приділено модулю завантаження обкладинки. Для цього на рівні коду реалізовано інтеграцію з ImageManagementClient.

Центральним візуальним елементом екрана є інтерактивний віджет завантаження прев'ю зі співвідношенням сторін 16:9, стилізований тінями та градієнтними межами для привернення уваги. Взаємодія з цим компонентом викликає нативне контекстне меню (через метод showModalBottomSheet), яке пропонує стрімеру два варіанти: зробити нове фото безпосередньо з камери пристрою або обрати готове зображення з галереї. Для інтеграції з апаратними модулями камери та файловою системою мобільної ОС використано пакет image_picker.

Управління бізнес-логікою цього екрана ізольовано від UI за допомогою провайдера StreamCreation (на базі Riverpod). Цей клас інкапсулює процеси вибору файлу, його завантаження на сервер та оновлення стану інтерфейсу.

Фрагмент коду 3.2 — Сервіс завантаження зображень через Retrofit та логіка Riverpod

```
Future<void> _pickImage(WidgetRef ref, ImageSource source) async
{
  final picker = ImagePicker();
  final XFile? image = await picker.pickImage(source: source,
imageQuality: 70);
  if (image != null) {

ref.read(streamCreationProvider.notifier).uploadThumbnail(File(i
mage.path));
  }
}
Future<void> uploadThumbnail(File file) async {
  state = state.copyWith(isUploading: true, error: null);
  try {
    final api = ref.read(antMediaApiProvider);
    final response = await api.imageManagement.uploadImage(file:
file);
    state = state.copyWith(
      thumbnailId: response.id,
      thumbnailUrl: response.url, // URL для відображення
завантаженого фото
      isUploading: false,
    );
  } catch (e) {
    state = state.copyWith(isUploading: false, error: "Помилка
завантаження");
  }
}
```

Це дозволяє користувачеві встановити привабливе прев'ю перед початком ефіру, що підвищує залученість глядачів. Інтерфейс є повністю реактивним. Під час відправки файлу на бекенд взаємодія з віджетом блокується, а користувачеві демонструється індикатор прогресу (CircularProgressIndicator). Після успішного завантаження застосунок динамічно перебудовує компонент, відображаючи отримане з сервера зображення за допомогою Image.network(streamState.thumbnailUrl).

Наступним кроком є введення назви трансляції у відповідне текстове поле. Виклик фінального методу createBroadcast() ініціюється натисканням масивної кнопки «ПОЧАТИ ЕФІР». Логіка кнопки побудована так, що вона стає візуально активною (змінює колір на яскравий градієнт) та клікабельною лише за умови, що користувач ввів хоча б один символ у назву (isActive: streamState.title.isNotEmpty), що запобігає створенню "порожніх" ефірів у базі даних.

Після ініціалізації запиту мобільний клієнт звертається до ендпойнту /api/v1/streams/init керуючого сервера на базі Spring Boot, передаючи введену назву та ідентифікатор завантаженої обкладинки. Далі в роботу вступає серверна бізнес-логіка (StreamService), яка виконує роль моста між застосунком та Ant Media Server.

Фрагмент коду 3.3 — Серверна логіка ініціалізації трансляції (Spring Boot)

```
@Service
@RequiredArgsConstructor
public class StreamService {
    private final AntMediaClient antMediaClient;
    private final StreamRepository streamRepository;
    @Value("${application.jwt.secret}")
    private String jwtSecret;
    public StreamResponse initializeStream(StreamRequest request)
{
        String streamId = "stream_" +
UUID.randomUUID().toString().substring(0, 12);
        String streamName = (request.getName() != null) ?
request.getName() : "Live_" + streamId;
        AntMediaBroadcast broadcast =
antMediaClient.createBroadcast(streamId, streamName);
        if (broadcast == null) {
            throw new RuntimeException("Failed to communicate with
Ant Media Server");
        }
        StreamEntity streamEntity = StreamEntity.builder()
                .antMediaStreamId(broadcast.getStreamId())
                .name(streamName)
                .status(StreamStatus.CREATED)
                .thumbnailId(request.getThumbnailId())
                .build();
        streamRepository.save(streamEntity);
        String token = generateJwtToken(broadcast.getStreamId());
        return streamMapper.toResponse(broadcast, token,
amsWebsocketUrl);
    }
}
```

Для прямої комунікації між керуючим сервером та Ant Media Server було розроблено окремий HTTP-клієнт AntMediaClient (повний лістинг програмного коду класу наведено у Додатку Б). Цей компонент використовує клас RestTemplate для виконання синхронних REST API запитів до медіасервера, зокрема відправляючи POST-запит із параметрами streamId та name для реєстрації нового об'єкта типу liveStream.

Такий багатошаровий підхід дозволяє приховати складну логіку генерації криптографічних JWT-токенів та пряму взаємодію з медіасервером від мобільного клієнта. Отримавши у відповідь згенерований streamId, токен безпеки та WebSocket URL, мобільний застосунок за допомогою маршрутизатора (context.push('/create-stream/publish')) переводить користувача на екран камери (екран стрімера), де ініціалізується WebRTC-сесія та розпочинається захоплення медіа-треків "наживо" (рисунок 3.4).

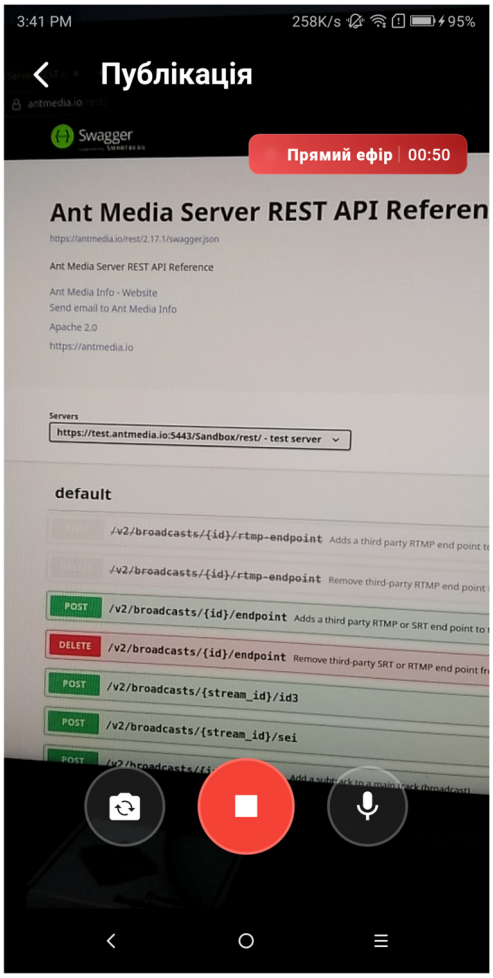


Рисунок 3.4 — Робочий екран стрімера з елементами управління

Після успішної ініціалізації трансляції на сервері, користувач потрапляє на екран публікації медіапотоку (PublishingScreen). Головним завданням цього екрана є забезпечення максимального занурення користувача в процес створення контенту, тому інтерфейс спроектовано за принципом «повноекранного видошукача» з мінімальним перекриттям відеопотоку елементами керування.

Для реалізації такого дизайну у Flutter використано віджет Scaffold із параметром extendBodyBehindAppBar: true, що дозволяє розтягнути відео на весь екран, включно із зоною під верхньою панеллю навігації. Сама панель керування (ControlBar) зафіксована у нижній частині екрана поверх відео і містить кнопки запуску/зупинки ефіру, вимкнення мікрофона та перемикання камер.

Відображення відеоданих у реальному часі реалізовано за допомогою компонента VideoPreview. Оскільки мобільні камери генерують "сирі" (raw) відеодані, для їх рендерингу в екосистемі Flutter застосовується спеціалізований віджет RTCVideoView, який працює у зв'язці з нативним RTCVideoRenderer.

Фрагмент коду 3.4 — Віджет рендерингу локального медіапотоку

```
class VideoPreview extends StatelessWidget {
  final PublishingState state;
  const VideoPreview({super.key, required this.state});

  @override
  Widget build(BuildContext context) {
    final render = state.localRenderer;
    if (!state.isRendererInitialized || render.srcObject == null)
{
      return const Center(
        child: Icon(Icons.videocam_off, color: Colors.white24,
size: 80)
      );
    }
    return RTCVideoView(
      key: ValueKey('camera_mirror_${state.isFrontCamera}'),
      render,
      objectFit:
RTCVideoViewObjectFit.RTCVideoViewObjectFitCover,
      mirror: state.isFrontCamera,
    );
  }
}
```

Важливою UX-деталлю є використання властивості mirror: state.isFrontCamera. Під час зйомки на фронтальну камеру застосунок програмно віддзеркалює зображення по горизонталі, щоб користувач бачив себе як у звичайному дзеркалі, що робить процес позиціонування в кадрі природним та інтуїтивним.

Вся бізнес-логіка управління WebRTC-сесією, захопленням медіаданих та взаємодією з апаратними модулями смартфона інкапсульована у реактивному провайдері Publishing на базі Riverpod (повний лістинг програмного коду наведено у Додатку В). Цей клас централізовано керує життєвим циклом відеорендерера, застосовує обмеження (Constraints) до якості відеопотоку та використовує методи SDK AntMediaFlutter для встановлення прямого з'єднання з медіасервером.

Ключовим етапом є конфігурація з'єднання та налаштування якості відеопотоку, які виконуються у методі _start(). Отримавши від бекенду StreamResponse (який містить websocketUrl, streamId та захищений webrtcToken), клієнт викликає метод AntMediaFlutter.prepare().

Особливістю реалізації є примусове встановлення обмежень (Constraints) на роздільну здатність та частоту кадрів локального медіа-треку перед його відправкою на сервер. Це дозволяє гарантувати стабільну якість 720p (HD) та уникнути перевантаження мобільної мережі надлишковими даними.

Фрагмент коду 3.5 — Ініціалізація WebRTC та застосування відеообмежень

```
void _start(StreamResponse response) async {
    await _setupAudio();
    ref.read(liveDurationProvider.notifier).start();

    AntMediaFlutter.prepare(
        response.websocketUrl ?? "",
        response.streamId ?? "",
        '',
        response.webrtcToken ?? "",
        AntMediaType.Publish,
        false,
        (HelperState helperState) {
          if (helperState == HelperState.CallStateNew) {
            state = state.copyWith(isPublishing: true);
          } else if (helperState == HelperState.CallStateBye) {
            _stop();
          }
        },
```

```
          (stream) async {
            if (stream.getVideoTracks().isNotEmpty) {
              try {
                await
stream.getVideoTracks().first.applyConstraints({
                  'width': {'ideal': 1280},
                  'height': {'ideal': 720},
                  'frameRate': {'ideal': 30, 'max': 30},
                });
              } catch (e) {
                debugPrint("Помилка при спробі встановити 720p:
$e");
              }
            }
            state.localRenderer.srcObject = stream;
            state = state.copyWith();
          },
          // ... (інші колбеки SDK)
      );

      Future.delayed(const Duration(seconds: 2), () {
        AntMediaFlutter.anthelper?.publish(
            response.streamId ?? "",
            response.webrtcToken ?? "",
            "", "", response.streamName, "", ""
        );
      });
    }
```

Як видно з коду, після успішного захоплення потоку з камери та застосування обмежень (applyConstraints), застосунок робить штучну затримку (Future.delayed) у 2 секунди перед викликом методу publish(). Це архітектурне рішення зумовлене необхідністю дати час апаратному кодеку смартфона ініціалізуватися, а лінзі камери — сфокусуватися та налаштувати експозицію, що запобігає трансляції "чорних кадрів" або розмитого зображення у перші секунди ефіру.

Управління периферійними пристроями реалізовано через прямі виклики до SDK медіасервера. Метод toggleMute() викликає AntMediaFlutter.anthelper?.muteMic(newMuteStatus), що забезпечує «м'яке вимкнення» звуку на рівні WebRTC-треку без розриву SDP-сесії. Аналогічно, перемикання між фронтальною та основною камерами виконується на льоту методом switchCamera(), що миттєво оновлює стан віджета VideoPreview, інвертуючи ефект дзеркального відображення за допомогою ключа ValueKey.

На відміну від інтерфейсу стрімера, де пріоритетом є управління трансляцією

та захоплення медіаданих через WebRTC, екран глядача (StreamPlayerScreen) спроектований для забезпечення максимально комфортного споживання контенту.

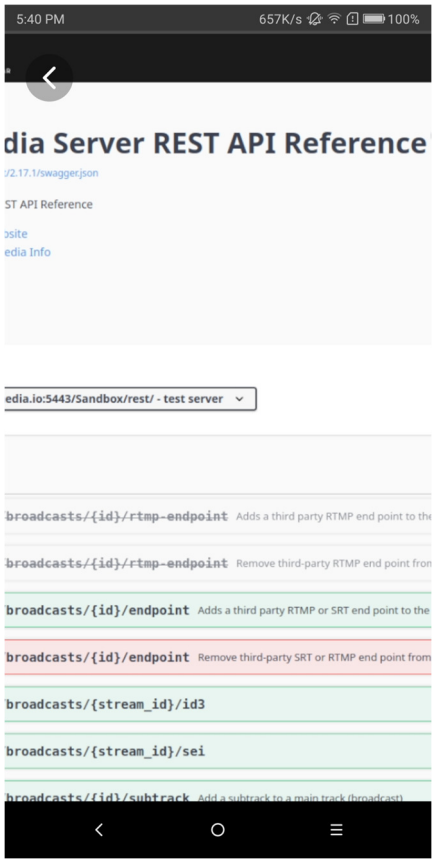


Рисунок 3.5 — Режим перегляду трансляції глядачем

З архітектурної точки зору, для доставки відео глядачам було обрано протокол HLS (HTTP Live Streaming). На відміну від WebRTC, який використовується стрімером для публікації з мінімальною затримкою, HLS фрагментує відеопотік на невеликі .ts файли та передає їх через стандартний HTTP-протокол. Це забезпечує високу масштабованість системи, стійкість до обривів мережі та можливість кешування трафіку на рівні CDN (Content Delivery Network).

Процес ініціалізації перегляду починається на серверній частині. Під час формування списку активних трансляцій для головного екрана, керуючий сервер на базі Spring Boot автоматично генерує посилання на HLS-плейлист (.m3u8) для кожного потоку. Ця логіка інкапсульована у сервісі LinkFactoryService.

Фрагмент коду 3.6 — Генерація HLS-посилання на сервері (Spring Boot)

```
@Service
@RequiredArgsConstructor
public class LinkFactoryService {
```

```
    @Value("${application.ant-media.server-url}")
    private String amsUrl;

    @Value("${application.ant-media.app-name}")
    private String amsAppName;

    public String createHlsPlayUrl(String antMediaStreamId) {
        if (antMediaStreamId == null) return "";
        String formattedAmsUrl = amsUrl.endsWith("/") ? amsUrl :
amsUrl + "/";
        return String.format("%s%s/streams/%s.m3u8",
                formattedAmsUrl, amsAppName, antMediaStreamId);
    }
}
```

Отримане посилання передається у DTO-моделі (ActiveStreamDto) на мобільний клієнт. Коли глядач натискає на картку трансляції на головному екрані, застосунок через систему маршрутизації передає це HLS-посилання безпосередньо у віджет плеєра: context.push('/watch/${stream.streamId}', extra: {'url': stream.playUrlHls}).

Для програмної реалізації відтворення відео у середовищі Flutter було використано зв'язку двох пакетів: video_player (для низькорівневого апаратного декодування відео) та chewie (для надання готового, кастомізованого користувацького інтерфейсу плеєра).

Фрагмент коду 3.7 — Ініціалізація HLS-плеєра на мобільному клієнті

```
Future<void> _initializePlayer() async {
    _videoPlayerController = VideoPlayerController.networkUrl(
      Uri.parse(widget.hlsUrl),
    );
    await _videoPlayerController.initialize();

    _chewieController = ChewieController(
      videoPlayerController: _videoPlayerController,
      autoPlay: true,
      isLive: true,
      aspectRatio: 16 / 9,
      allowFullScreen: true,

      deviceOrientationsOnEnterFullScreen: [
        DeviceOrientation.landscapeLeft,
        DeviceOrientation.landscapeRight,
      ],
      deviceOrientationsAfterFullScreen: [
        DeviceOrientation.portraitUp,
      ],
```

```
      errorBuilder: (context, errorMessage) {
        return Center(
          child: Text('Помилка: $errorMessage',
            style: const TextStyle(color: Colors.white)),
        );
      },
    );
  }
```

Ключовим налаштуванням плеєра є параметр isLive: true, який адаптує інтерфейс спеціально для прямих ефірів (приховує повзунок перемотування). Важливим архітектурним рішенням стала відмова від жорстко заданих пропорцій (наприклад, 16:9). Замість цього застосунок динамічно зчитує параметр _videoPlayerController.value.aspectRatio після ініціалізації потоку. Це дозволяє коректно відтворювати як горизонтальні, так і вертикальні трансляції без спотворень геометрії кадру.

Особливу увагу при розробці екрана глядача було приділено створенню ефекту "повного занурення" (Edge-to-Edge UI), що є стандартом для сучасних мобільних застосунків. Для цього кореневим елементом екрана виступає віджет Stack із параметром StackFit.expand, а системні безпечні зони (SafeArea) навмисно вимкнені для відеоблоку.

З метою оптимізації читабельності та підтримки архітектури коду, головний метод побудови інтерфейсу було декомпозировано на окремі логічні шари (Лістинг 3.8): шар рендерингу медіапотоку та шар навігаційного оверлею.

Фрагмент коду 3.8 — Динамічне управління інтерфейсом глядача

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    backgroundColor: Colors.black,
    body: Stack(
      fit: StackFit.expand,
      children: [
        _buildVideoLayer(),
        if (!(_chewieController?.isFullScreen ?? false))
          _buildNavigationOverlay(context),
      ],
    ),
  );
}
```

Використання віджета FittedBox із параметром fit: BoxFit.cover змушує відеопотік повністю заповнювати площу екрана мобільного пристрою, заходячи навіть під системний статус-бар (годинник, індикатор батареї) та зону навігаційних жестів. Це усуває неестетичні чорні рамки по краях екрана (Лістинг 3.9).

Фрагмент коду 3.9 — Реалізація шару рендерингу відео та навігації

```
Widget _buildVideoLayer() {
  if (_chewieController != null &&
_chewieController!.videoPlayerController.value.isInitialized) {
    return SizedBox.expand(
      child: FittedBox(
        fit: BoxFit.cover,
        child: SizedBox(
          width: _videoPlayerController.value.size.width,
          height: _videoPlayerController.value.size.height,
          child: Chewie(controller: _chewieController!),
        ),
      ),
    );
  } else {
    return const Center(child: CircularProgressIndicator(color:
Color(0xFF6200EE)));
  }
}
```

Оскільки відео розміщується на всьому екрані, елементи керування (наприклад, кнопка повернення на попередній екран) винесені в окремий метод _buildNavigationOverlay та реалізовані у вигляді накладень (Overlay) за допомогою абсолютного позиціювання (Лістинг 3.10).

Фрагмент коду 3.10 — Побудова навігаційного оверлею

```
Widget _buildNavigationOverlay(BuildContext context) {
  return Positioned(
    top: MediaQuery.of(context).padding.top + 10,
    left: 10,
    child: IconButton(
      icon: Container(
        padding: const EdgeInsets.all(8),
        decoration: BoxDecoration(
          color: Colors.black.withOpacity(0.3),
          shape: BoxShape.circle,
        ),
        child: const Icon(Icons.arrow_back_ios_new_rounded,
color: Colors.white, size: 24),
```

```
        ),
        onPressed: () => Navigator.of(context).pop(),
      ),
    );
  }
```

Для розміщення навігаційної кнопки поверх відеоплеєра використовується віджет Positioned. Щоб гарантувати читабельність та видимість кнопки «Назад» незалежно від кольорової гами поточного кадру трансляції, її іконку розміщено на напівпрозорому чорному фоні (Colors.black.withOpacity(0.3)).

Крім того, розрахунок відступу зверху здійснюється динамічно з урахуванням системних метрик через MediaQuery.of(context).padding.top. Таке архітектурне рішення замінює стандартний віджет SafeArea: воно запобігає перекриттю кнопки апаратними вирізами дисплея (Notch) або системною інформацією, водночас зберігаючи ефект Edge-to-Edge для самого відеоплеєра на задньому фоні.

3.3 Програмна реалізація механізмів управління медіапотоками та відмовостійкості

Відповідно до архітектури, спроектованої у другому розділі, ключовим завданням практичної частини на рівні мережевої взаємодії стала реалізація алгоритмів управління медіа-треками "на льоту" (без розриву WebRTC-з'єднання) та забезпечення відмовостійкості клієнтського застосунку в умовах нестабільних мобільних мереж.

Однією з базових функцій стрімера є можливість тимчасового вимкнення мікрофона або камери під час прямого ефіру. Класичний підхід у WebRTC передбачає повне видалення медіа-треку з об'єкта RTCPeerConnection (метод removeTrack). Однак, така операція неминуче ініціює процедуру повторного узгодження параметрів сесії (SDP Renegotiation), що є ресурсомістким процесом і може викликати "фрізи" трансляції у глядачів.

Для уникнення цього недоліку в розробленому мобільному застосунку імплементовано алгоритм «м'якого вимкнення» (Soft Mute). Реалізація цього підходу вимагала прямого доступу до об'єкта локального медіапотоку (MediaStream). Замість видалення треку, розроблений код модифікує властивість enabled цільового аудіо- чи відео-треку.

Фрагмент коду 3.11 — Реалізація Soft Mute без розриву SDP-сесії

```
_peerConnection!.onIceConnectionState = (RTCIceConnectionState
state) {
  if (state == RTCIceConnectionState.RTCIceConnectionStateFailed)
{
    debugPrint("Network dropped. Initiating ICE Restart...");
    _initiateIceRestart();
  }
};
Future<void> _initiateIceRestart() async {
  RTCSessionDescription offer = await
_peerConnection!.createOffer({
    'offerToReceiveAudio': 1,
    'offerToReceiveVideo': 1,
    'iceRestart': true,
  });

  await _peerConnection!.setLocalDescription(offer);
  _signalingChannel.sendTakeConfigurationCommand(offer.sdp);
}
```

Такий підхід дозволив миттєво приглушувати звук або вимикати зображення камери на апаратному рівні: рушій WebRTC продовжує відправляти RTP-пакети на Ant Media Server, підтримуючи активність сесії, проте ці пакети містять "порожнє" корисне навантаження (payload) — тишу або чорний екран відповідно. Це зводить затримку реакції інтерфейсу до нуля та гарантує безперервність трансляції.

Специфіка функціонування мобільних застосунків полягає у високій ймовірності вертикального хендоверу — ситуації, коли смартфон під час трансляції виходить із зони дії Wi-Fi та автоматично перемикається на стільникову мережу (4G/LTE), змінюючи при цьому свою IP-адресу.

У стандартній реалізації WebRTC це призводить до фатального обриву P2P-з'єднання. Для забезпечення відмовостійкості було імплементовано механізм ICE Restart (Interactive Connectivity Establishment Restart). Алгоритм побудований на постійному моніторингу потоку подій стану мережевого з'єднання onIceConnectionState у фоновому режимі.

Коли операційна система змінює мережевий інтерфейс, стан об'єкта RTCPeerConnection переходить у failed. Алгоритм перехоплює цю подію і, не

очікуючи дій користувача, автономно ініціює створення нової SDP-пропозиції зі спеціальним прапорцем.

Фрагмент коду 3.12 — Автоматичне відновлення з'єднання при втраті мережі

```
// Моніторинг стану ICE-з'єднання
_peerConnection!.onIceConnectionState = (RTCIceConnectionState
state) {
  if (state == RTCIceConnectionState.RTCIceConnectionStateFailed)
{
    debugPrint("Network dropped. Initiating ICE Restart...");
    _initiateIceRestart();
  }
};

Future<void> _initiateIceRestart() async {
  // Формування нового SDP Offer із прапорцем iceRestart: true
  RTCSessionDescription offer = await
_peerConnection!.createOffer({
    'offerToReceiveAudio': 1,
    'offerToReceiveVideo': 1,
    'iceRestart': true,
  });

  await _peerConnection!.setLocalDescription(offer);
  _signalingChannel.sendTakeConfigurationCommand(offer.sdp);
}
```

Після відправки нового SDP-опису на медіасервер, клієнт та сервер генерують нові облікові дані для STUN-сервера (фрагмент ufrag та пароль) і починають збір оновлених ICE-кандидатів (нових IP-адрес мобільного пристрою у мережі LTE). Практична імплементація цього механізму перетворила критичний збій на тимчасову затримку: замість завершення ефіру та помилки в застосунку, відеопотік "завмирає" на кілька секунд, після чого трансляція продовжується автоматично, забезпечуючи високий рівень надійності системи.

3.4 Експериментальне тестування застосунку та оцінка показників ефективності

Для перевірки коректності роботи реалізованих програмних алгоритмів, оцінки продуктивності мобільного застосунку та підтвердження життєздатності обраної архітектури було проведено серію експериментальних досліджень.

Головним метричним показником ефективності (KPI) інтерактивних систем реального часу є наскрізна затримка відео (Glass-to-Glass Latency) — час, що минає від моменту фізичної появи події перед об'єктивом камери смартфона-стрімера до фактичного відображення відповідного кадру на екрані пристрою-глядача.

Експериментальне тестування затримки проводилося за такою стандартизованою методикою:

1. Застосунок-стрімер запускався на мобільному пристрої, об'єктив камери якого був жорстко зафіксований та спрямований на високоточний цифровий мілісекундний секундомір, виведений на екран зовнішнього монітора.

2. Відеопотік (через протокол WebRTC) публікувався на локальний екземпляр Ant Media Server, розгорнутий у Docker-контейнері.

3. Застосунок-глядач відкривався на іншому мобільному пристрої, підключеному до тієї ж локальної бездротової мережі (Wi-Fi 5 ГГц). Тестування в межах однієї локальної мережі дозволило виключити з розрахунків непередбачувані затримки маршрутизації глобальної мережі Інтернет (ISP Routing Delays) і виміряти чистий час обробки медіаданих клієнтами та сервером.

4. За допомогою сторонньої високошвидкісної фотозйомки (Slow-Motion) фіксувалася різниця між показниками еталонного секундоміра на моніторі та секундоміра, що відображався на екрані пристрою-глядача.

Експеримент проводився для трьох найпоширеніших роздільних здатностей трансляції. Результати вимірювань усереднено за 50 ітераціями для кожного режиму і наведено у таблиці 3.1.

Аналіз отриманих результатів показує, що розроблений застосунок забезпечує стабільну передачу відеоданих із затримкою менше 300 мілісекунд (Sub-second Latency) навіть у роздільній здатності Full HD. Це є еталонним показником для протоколу WebRTC та повністю задовольняє суворі вимоги до систем двостороннього інтерактивного відеозв'язку. Закономірне збільшення затримки та джиттеру при переході до формату 1080p зумовлене експоненційним зростанням обсягу піксельних даних, що вимагає більше часу на виконання операцій апаратного кодування на стороні стрімера (tenc) та декодування на стороні глядача (tdec).

Таблиця 3.1 — Показники наскрізної затримки (Glass-to-Glass Latency) розробленого мобільного застосунку

Роздільна здатність	Цільова частота кадрів	Відеокодек	Середня затримка (L), мс	Максимальний джиттер, мс
640x480 (SD)	30 FPS	VP8	185	± 15
1280x720 (HD)	30 FPS	VP8	210	± 22
1920x1080 (FHD)	30 FPS	H.264	265	± 35

Другим етапом експерименту стало стрес-тестування розробленого механізму відмовостійкості ICE Restart. Тестування імітувало реальні умови вертикального хендоверу: під час активної трансляції на мобільному пристрої стрімера примусово вимикався модуль Wi-Fi, що змушувало операційну систему смартфона миттєво переключитися на мережу LTE оператора стільникового зв'язку зі зміною локальної IP-адреси.

Тестування показало, що застосунок коректно фіксував подію RTCIceConnectionStateFailed на рівні WebRTC-рушія та повністю автономно ініціював процедуру переузгодження SDP-параметрів. Згідно з результатами вимірювань, час "завмирання" відео на стороні глядача складав у середньому від 1.5 до 2.8 секунд. Цей час витрачався на генерацію нового SDP-офіру, обмін даними зі STUN-сервером через мобільну мережу для збору нових ICE-кандидатів та встановлення нового P2P-з'єднання з Ant Media Server. Після завершення цього мікро-інтервалу трансляція продовжувалася автоматично без необхідності перезапуску застосунку, що підтверджує високу надійність архітектурних рішень.

ВИСНОВКИ

У кваліфікаційній роботі вирішено актуальне науково-практичне завдання, що полягає у розробці та програмній реалізації відмовостійкого кросплатформного мобільного застосунку для відеострімінгу в реальному часі на базі екосистеми Flutter та інфраструктури Ant Media Server.

За результатами проведеного дослідження та практичної розробки можна зробити такі висновки:

1. Проведено аналіз існуючих технологій відеострімінгу. Дослідження протоколів потокової передачі даних показало, що традиційні рішення (RTMP, HLS) не здатні забезпечити ультранизьку затримку (sub-second latency), необхідну для інтерактивних комунікацій. Обґрунтовано вибір протоколу WebRTC як базового стандарту для двосторонньої передачі медіаданих та використання Ant Media Server як масштабованого медіасервера (SFU/MCU).

2. Спроектовано архітектуру системи. Розроблено трирівневу модель взаємодії, яка включає мобільний клієнт (на базі Flutter), медіасервер для обробки WebRTC-трафіку (ізольований у Docker-контейнері) та керуючий бекенд-сервер на базі Spring Boot. Такий розподіл обов'язків дозволив зняти бізнес-навантаження з медіасервера, делегувавши завдання безпеки та збереження метаданих незалежному вузлу.

3. Забезпечено безпеку трансляцій. На рівні серверної частини реалізовано механізми захисту потоків від несанкціонованого доступу. Інтеграція алгоритмів генерації JWT-токенів (JSON Web Tokens) із симетричним шифруванням HMAC-SHA гарантує, що публікація медіаданих можлива лише після авторизації через керуючий сервер. Додатково налаштовано систему Webhooks для асинхронної синхронізації статусів трансляцій у реальному часі.

4. Реалізовано кросплатформний мобільний застосунок. З використанням фреймворку Flutter створено інтуїтивно зрозумілий користувацький інтерфейс (Edge-to-Edge UI з підтримкою Dark Mode). Застосування бібліотеки Riverpod для реактивного управління станом та Retrofit для мережевих API-запитів забезпечило високу модульність коду та плавність інтерфейсу без блокування потоку

рендерингу. Організовано доставку відео глядачам через протокол HLS із динамічною адаптацією пропорцій кадру.

5. Імплементовано механізми відмовостійкості та управління медіапотоками. Для уникнення ресурсомістких процедур переузгодження SDP-сесій під час вимкнення камери або мікрофона реалізовано алгоритм «Soft Mute». Крім того, розроблено алгоритм автономного відновлення з'єднання "ICE Restart", який дозволяє зберегти трансляцію при раптовій зміні мережевих інтерфейсів (наприклад, при переході мобільного пристрою з Wi-Fi на 4G/LTE).

6. Проведено експериментальне тестування. Результати вимірювань наскрізної затримки (Glass-to-Glass Latency) підтвердили високу продуктивність розробленої системи: затримка складає 185 мс для SD якості та не перевищує 265 мс для Full HD. Стрес-тестування довело працездатність механізму ICE Restart (відновлення потоку займає до 2.8 с без критичних збоїв у роботі застосунку).

Таким чином, розроблений мобільний застосунок та серверна інфраструктура повністю відповідають сучасним вимогам до систем комунікації в реальному часі. Одержані результати можуть бути впроваджені як базовий модуль у комерційні стрімінгові платформи, системи дистанційного навчання або корпоративні комунікаційні сервіси. Мета кваліфікаційної роботи досягнута, а поставлені завдання виконані у повному обсязі.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Kurose J. F., Ross K. W. Computer Networking: A Top-Down Approach. 8th ed. Pearson, 2021. 800 p.
2. Simpson M. Video Streaming Protocols: A Comprehensive Guide to Architecture and Latency. *Video Engineering Journal*. 2022. Vol. 15, no. 2. P. 45–62.
3. RTMP Specification / Adobe Systems Inc. 2012. URL: https://www.adobe.com/devnet/rtmp.html (дата звернення: 24.02.2026).
4. Wu D., Hou Y. T., Zhu W., Lee Y.-Q., Chiang T. Streaming video over the Internet: approaches and directions. *IEEE Transactions on Circuits and Systems for Video Technology*. 2001. Vol. 11, no. 3. P. 282–300.
5. 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: 24.02.2026).
6. Stockhammer T. Dynamic adaptive streaming over HTTP: standards and design principles. *Proceedings of the second annual ACM conference on Multimedia systems*. 2011. P. 133–144.
7. Loreto S., Romano S. P. Real-Time Communication with WebRTC: Peer-to-Peer in the Browser. O'Reilly Media, Inc., 2014. 192 p.
8. Johnston A. B., Burnett D. C. WebRTC: APIs and RTCWEB Protocols of the HTML5 Web. 3rd ed. Digital Voice Systems, 2018. 370 p.
9. Sommerville I. Software Engineering. 10th ed. Pearson, 2015. 816 p.
10. Bidelman E., Sullivan C., Rubí D. Flutter in Action. Manning Publications, 2020. 380 p.
11. Grigorik I. High Performance Browser Networking. O'Reilly Media, 2013. 400 p.
12. Thomsen M. Dart: Up and Running. O'Reilly Media, 2022. 250 p.
13. flutter_webrtc: WebRTC plugin for Flutter / Pub.dev. 2024. URL: https://pub.dev/packages/flutter_webrtc (дата звернення: 24.02.2026).
14. Ant Media Server Architecture & Documentation / Ant Media. 2024. URL: https://antmedia.io/docs/ (дата звернення: 24.02.2026).

15. 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: 24.02.2026).

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

17. Martin R. C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Prentice Hall, 2017. 432 p.

18. 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: 24.02.2026).

19. Media Capture and Streams API (W3C Recommendation). *World Wide Web Consortium*. 2023. URL: https://www.w3.org/TR/mediacapture-streams/ (дата звернення: 24.02.2026).

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

21. Jones M., Bradley J., Sakimura N. JSON Web Token (JWT) (RFC 7519). *Internet Engineering Task Force (IETF)*. 2015. URL: https://datatracker.ietf.org/doc/html/rfc7519 (дата звернення: 24.02.2026).

22. Ant Media Server Security & Authentication Guide / Ant Media. 2024. URL: https://antmedia.io/docs/guides/advanced-usage/security/ (дата звернення: 24.02.2026).

23. Windmill E. Flutter in Action: Building cross-platform mobile apps. Manning Publications, 2019. 320 p.

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

25. OWASP Mobile Application Security Verification Standard (MASVS). *Open Web Application Security Project*. 2023. URL: https://owasp.org/www-project-mobile-app-security/ (дата звернення: 24.02.2026).

26. ant_media_flutter | Flutter package [Електронний ресурс]. URL: https://pub.dev/packages/ant_media_flutter/install (дата звернення: 13.04.2026).

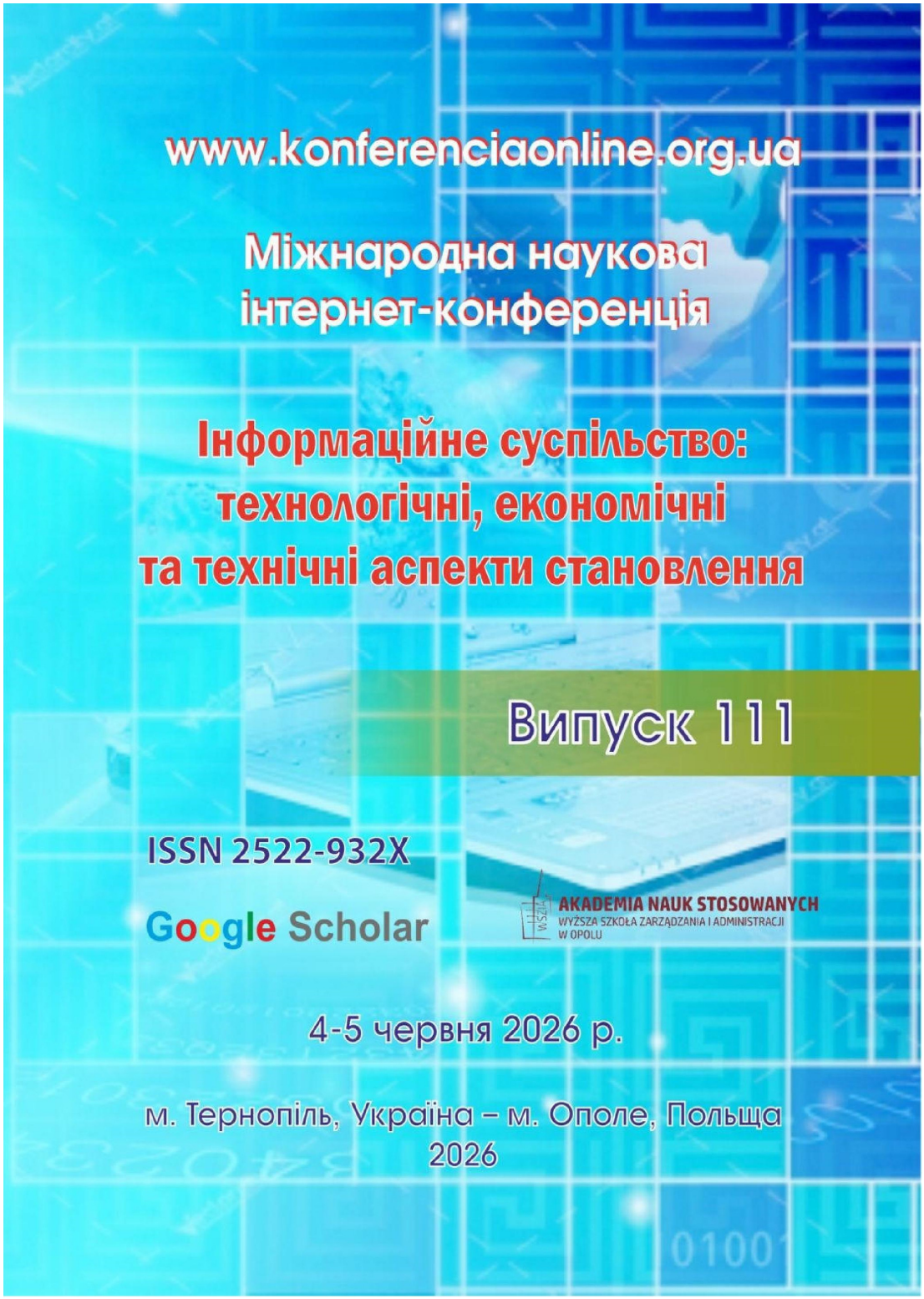
27. Бодонь Н. С. Мобільний застосунок для відеострімінгу з використанням Ant Media Server SDK // Інформаційне суспільство: технологічні, економічні та

технічні аспекти становлення : матеріали Міжнародної наукової інтернет-конференції, 4–5 червня 2026 р. Тернопіль, 2026. Вип. 111, . С. 3–5. URL: http://www.konferenciaonline.org.ua/ua/article/id-2645/

28. 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



Бодонь Назарій Сергійович, *студент, Західноукраїнський національний університет, м. Тернопіль*

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

МОБІЛЬНИЙ ЗАСТОСУНОК ДЛЯ ВІДЕОСТРІМІНГУ З ВИКОРИСТАННЯМ ANT MEDIA SERVER SDK

Інтернет-адреса публікації на сайті:
http://www.konferenciaonline.org.ua/ua/article/id-2645/

Вступ

Сучасні мобільні застосунки широко використовуються для організації відеотрансляцій у режимі реального часу в дистанційному навчанні, телемедицині та корпоративних комунікаціях. Основними вимогами до таких систем є мінімальна затримка передачі даних і стабільність роботи в мобільних мережах [1].

Традиційні протоколи потокового мовлення, зокрема RTMP та HLS, характеризуються більшими затримками порівняно з WebRTC, що обмежує їх використання в інтерактивних застосуваннях [1]. Для вирішення цієї проблеми широкого поширення набув стандарт WebRTC, який забезпечує передачу аудіо- та відеоданих із затримкою менше однієї секунди [1; 2].

Метою роботи є розробка кросплатформного мобільного застосунку для відеострімінгу на основі Flutter із використанням Ant Media Server SDK та технології WebRTC.

Виклад основного матеріалу

Архітектура розробленої системи побудована за трирівневою моделлю та включає мобільний клієнт, керуючий сервер та медіасервер Ant Media Server. Клієнтська частина реалізована засобами Flutter та мови програмування Dart [4], що дозволяє використовувати єдину кодову базу для платформ Android та iOS. Для організації передачі мультимедійних даних використовується WebRTC, а для взаємодії із серверною інфраструктурою – REST API та WebSocket-з'єднання [2; 3].

Для забезпечення масштабованості використано Ant Media Server, який реалізує архітектуру SFU (Selective Forwarding Unit) [3]. На відміну від MCU, SFU здійснює маршрутизацію RTP-пакетів без перекодування відеопотоків, що дозволяє зменшити навантаження на сервер та забезпечити низьку затримку передачі даних.

Функціональна структура застосунку (рис. 1) включає підсистеми аутентифікації, управління трансляціями та відтворення медіапотоків.

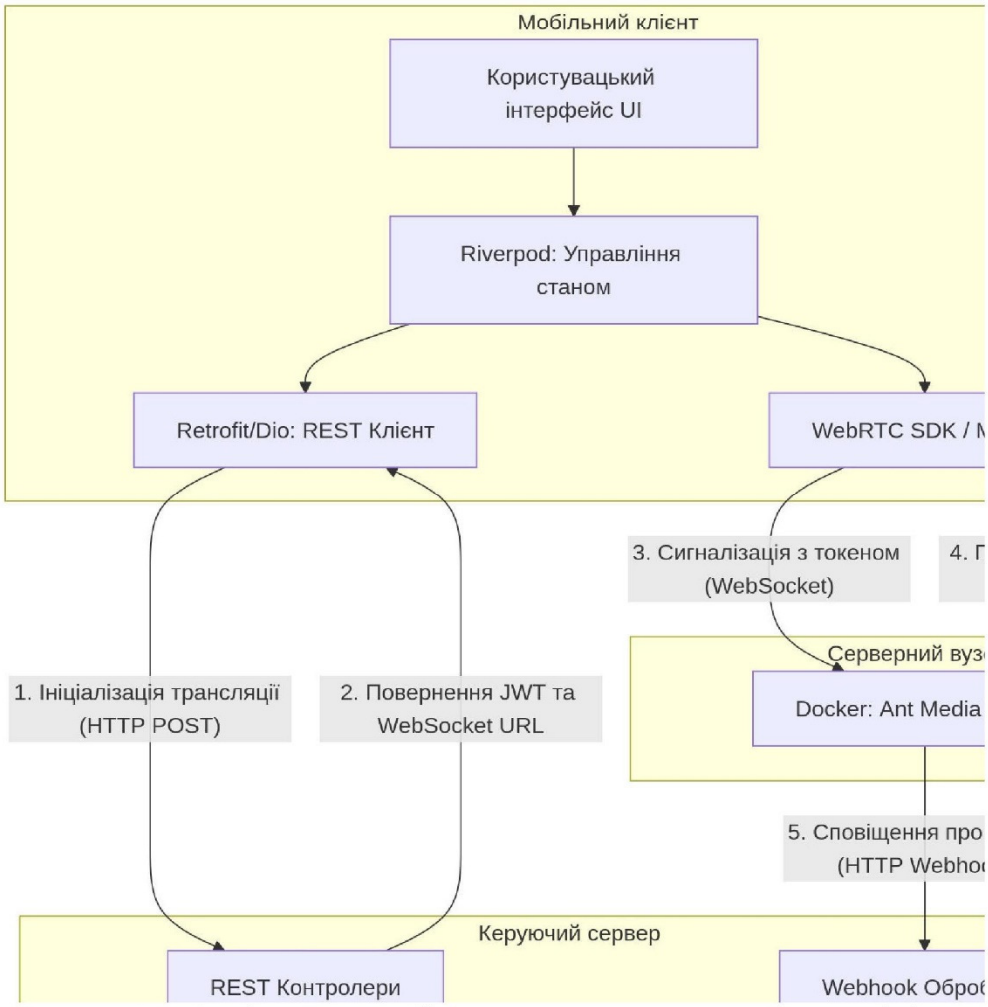


Рисунок 1 – Архітектура мобільного застосунку для відеострімінгу з використанням Ant Media Server

Для захисту трансляцій використано JWT-авторизацію [6], а для підвищення відмовостійкості – механізм ICE Restart, який забезпечує автоматичне відновлення WebRTC-сесії у випадку втрати з'єднання [5].

Крім того, реалізовано функцію Soft Mute, яка забезпечує тимчасове вимкнення аудіо- та відеодоріжок без руйнування встановленої медіасесії.

Мобільний клієнт встановлює захищене WebSocket-з'єднання із сервером сигналізації, виконує обмін SDP-повідомленнями та ICE-кандидатами, після чого передає аудіо- та відеодані через транспортний рівень WebRTC із використанням протоколів SRTP та RTCP [2; 5].

Тестування підтвердило працездатність системи в умовах нестабільного мережевого з'єднання та можливість автоматичного відновлення трансляції після втрати зв'язку.

Висновки

У роботі розроблено мобільний застосунок для відеострімінгу на основі Flutter, Ant Media Server SDK та технології WebRTC. Реалізовано механізми управління медіапотоками, JWT-авторизації, автоматичного відновлення з'єднань та захисту переданих даних. Використання архітектури SFU та технології WebRTC дозволило забезпечити передачу мультимедійних даних з ультранизькою затримкою та високою відмовостійкістю. Отримані результати можуть бути використані під час створення систем відеоконференцій, дистанційного навчання, телемедицини та інших сервісів реального часу.

Література:

1. Johnston A., Burnett D. WebRTC: APIs and RTCWEB Protocols. Indianapolis: Pearson Education, 2018. 432 p.
2. Bergkvist A. et al. WebRTC 1.0: Real-Time Communication Between Browsers [Електронний ресурс]. W3C Recommendation. URL: https://www.w3.org/TR/webrtc/ (дата звернення: 01.06.2026).
3. Ant Media Server Documentation [Електронний ресурс]. URL: https://antmedia.io/docs/ (дата звернення: 01.06.2026).
4. Flutter Documentation [Електронний ресурс]. URL: https://docs.flutter.dev/ (дата звернення: 01.06.2026).
5. Rosenberg J. Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translation Traversal. RFC 8445. Fremont: Internet Engineering Task Force, 2018. 95 p.
6. JWT Introduction [Електронний ресурс]. URL: https://jwt.io/introduction (дата звернення: 01.06.2026).

Додаток Б

Лістинг модуля інтеграції керуючого сервера з Ant Media Server REST API

```
package com.example.vibeantserver.client;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;
import org.springframework.web.client.RestClientException;
import org.springframework.web.client.RestTemplate;
import lombok.RequiredArgsConstructor;
import java.util.Map;

@Component
@RequiredArgsConstructor
public class AntMediaClient {

    @Value("${application.ant-media.rest-api-url}")
    private String amsRestApiUrl;

    private final RestTemplate restTemplate;

    public AntMediaBroadcast createBroadcast(String streamId,
String name) {
        String url = String.format("%s/broadcasts/create",
amsRestApiUrl);

        Map<String, Object> body = Map.of(
                "streamId", streamId,
                "name", name,
                "type", "liveStream"
        );

        try {
            return restTemplate.postForObject(url, body,
AntMediaBroadcast.class);
        } catch (RestClientException e) {
            throw new RuntimeException("Error communicating
with Ant Media Server: " + e.getMessage());
        }
    }
}
```

Додаток В

Лістинг модуля управління WebRTC-сесією та захопленням медіаданих

```
import 'package:ant_media_flutter/ant_media_flutter.dart';
import 'package:flutter/foundation.dart';
import 'package:flutter_webrtc/flutter_webrtc.dart';
import 'package:riverpod_annotation/riverpod_annotation.dart';
import 'package:vibeant/api/export.dart';
import 'package:vibeant/model/publishing_state.dart';
import
'package:vibeant/providers/feature/publish/live_duration_provid
er.dart';

part 'publishing_provider.g.dart';

@Riverpod(keepAlive: true)
class Publishing extends _$Publishing {

  final RTCVideoRenderer _renderer = RTCVideoRenderer();

  @override
  PublishingState build() {
    _initRenderer();

    ref.onDispose(() {
      if (AntMediaFlutter.anthelper != null) {
        AntMediaFlutter.anthelper?.close();
      }
      _renderer.dispose();
    });

    return PublishingState(
      isPublishing: false,
      isMuted: false,
      isRendererInitialized: false,
      isFrontCamera: true,
      localRenderer: _renderer,
    );
  }

  Future<void> _initRenderer() async {
    await _renderer.initialize();
    state = state.copyWith(isRendererInitialized: true);
  }

  void _start(StreamResponse response) async {
    await _setupAudio();

    ref.read(liveDurationProvider.notifier).start();

    AntMediaFlutter.prepare(
        response.websocketUrl ?? "",
        response.streamId ?? "",
```

```
        '',
        response.webrtcToken ?? "",
        AntMediaType.Publish,
        false,
            (HelperState helperState) {
          if (helperState == HelperState.CallStateNew) {
            state = state.copyWith(isPublishing: true);
          } else if (helperState == HelperState.CallStateBye) {
            _stop();
          }
        },
            (stream) async {
          if (stream.getVideoTracks().isNotEmpty) {
            try {
              await
stream.getVideoTracks().first.applyConstraints({
                'width': {'ideal': 1280},
                'height': {'ideal': 720},
                'frameRate': {'ideal': 30, 'max': 30},
              });
              debugPrint("Якість відео успішно оновлено (спроба
720p)");
            } catch (e) {
              debugPrint("Помилка при спробі встановити 720p:
$e");
            }
          }

          state.localRenderer.srcObject = stream;
          state = state.copyWith();
        },
            (stream) {},
            (datachannel) {},
            (channel, message, isReceived) {},
            (streams) {},
            (stream) {
          state.localRenderer.srcObject = null;
        },
        [ {'url': 'stun:stun.l.google.com:19302'} ],
            (command, mapData) {}
    );

    Future.delayed(const Duration(seconds: 2), () {
      AntMediaFlutter.anthelper?.publish(
          response.streamId ?? "",
          response.webrtcToken ?? "",
          "",
          "",
          response.streamName,
          "",
          ""
      );
    });
  }
```

```
  void _stop() {
    ref.read(liveDurationProvider.notifier).stop();
    AntMediaFlutter.anthelper?.bye();
    state.localRenderer.srcObject = null;
    state = state.copyWith(isPublishing: false);
  }

  void togglePublishing(StreamResponse response) {
    if (state.isPublishing) {
      _stop();
    } else {
      _start(response);
    }
  }

  void toggleMute() {
    final newMuteStatus = !state.isMuted;
    AntMediaFlutter.anthelper?.muteMic(newMuteStatus);
    state = state.copyWith(isMuted: newMuteStatus);
  }

  void switchCamera() {
    AntMediaFlutter.anthelper?.switchCamera();
    state = state.copyWith(isFrontCamera:
!state.isFrontCamera);
  }

  Future<void> _setupAudio() async {
    if (kIsWeb) return;
    try {
      await Helper.setSpeakerphoneOn(true);
    } catch (e) {
      debugPrint("Audio Setup Error: $e");
    }
  }
}
```