

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи освітнього ступеня «бакалавр»
за спеціальністю 121 «Інженерія програмного забезпечення»
(освітня програма «Інженерія програмного забезпечення»)

на тему:

«Система ефективної потокової передачі медіа: оптимізація та
мобільна інтеграція P2P Media Loader»

Виконав студент групи ПЗ-21-5
ДЕМЧЕНКО Дмитро Олександрович

Керівник роботи:
САВІЦЬКИЙ Роман Святославович

Рецензент:
ТОЛСТОЙ Ігор Анатолійович

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

«ЗАТВЕРДЖУЮ»

Зав. кафедри інженерії
програмного забезпечення

Тетяна Вакалюк

«23» лютого 2025 р.

ЗАВДАННЯ
на кваліфікаційну роботу

Здобувач вищої освіти: **ДЕМЧЕНКО Дмитро Олександрович**

Керівник роботи: **САВІЦЬКИЙ Роман Святославович**

Тема роботи: **«Система ефективної потокової передачі медіа: оптимізація та мобільна інтеграція P2P Media Loader»**,

затверджена Наказом закладу вищої освіти від **«14» лютого 2025 р., №74/с**

Вихідні дані для роботи: об'єктом дослідження є системи ефективної потокової передачі медіа на основі гібридних P2P-мереж. Предметом дослідження є принципи та моделі побудови ефективних P2P-систем доставки медіа, методи їх оптимізації для мобільних платформ та архітектурні патерни інтеграції з нативними додатками.

Консультанти з бакалаврської кваліфікаційної роботи із зазначенням розділів, що їх стосуються:

Розділ	Консультант	Завдання видав	Завдання прийняв
1	Савіцький Р. С.	27.02.2025	15.04.2025
2	Савіцький Р. С.	27.02.2025	01.05.2025
3	Савіцький Р. С.	27.02.2025	27.05.2025

РЕФЕРАТ

Кваліфікаційна робота бакалавра присвячена розробці та дослідженню системи ефективної P2P-доставки медіаконтенту для платформи Android на основі бібліотеки P2P Media Loader. Пояснювальна записка до роботи містить 78 сторінок, 31 ілюстрацію та 17 використаних джерел.

Метою роботи є підвищення ефективності потокової передачі медіа шляхом оптимізації базової P2P-бібліотеки та створення її мобільної інтеграції, що забезпечує повну сумісність з вебклієнтами.

В ході роботи було проведено комплексний аналіз предметної області, включаючи технології P2P-стрімінгу, існуючі комерційні та відкриті рішення, а також альтернативні архітектурні підходи. На основі аналізу було здійснено низку оптимізацій базової бібліотеки P2P Media Loader, спрямованих на покращення управління ресурсами та підвищення стабільності. Ключовим результатом роботи є спроектована та розроблена бібліотека мобільної інтеграції p2pml-kotlin, яка вирішує задачу інтеграції веборієнтованих P2P-технологій у нативне Android-середовище, гарантуючи при цьому повну сумісність з існуючими вебклієнтами.

Ефективність розробленої системи підтверджено експериментально. Результати продемонстрували успішну кросплатформну взаємодію та можливість розвантаження CDN до 96% за наявності активних пірів при помірному та контрольованому навантаженні на ресурси мобільного пристрою. Розроблена бібліотека є готовим відкритим рішенням для Android-розробників.

КЛЮЧОВІ СЛОВА: P2P, СТРИМІНГ, WEBRTC, KOTLIN, ANDROID, WEBVIEW, HLS, CDN.

					ІПЗ.КР.Б – 121 – 25 – ПЗ			
Змн.	Арк.	№ докум.	Підпис	Дата	Система ефективної потокової передачі медіа: оптимізація та мобільна інтеграція P2P Media Loader	Лім.	Арк.	Аркушів
Розроб.		Демченко Д.О.						
Керівник		Савицький Р. С.					3	75
Рецензент		Толстой І. А.				ФІКТ Гр. ІПЗ-21-5		
Зав. каф.		Вакалюк Т. А.						

ABSTRACT

The bachelor's thesis is dedicated to the development and research of an effective P2P media content delivery system for the Android platform, based on the P2P Media Loader library. The explanatory note to the work contains 78 pages, 31 illustrations, and 17 references.

The main goal of the work is to increase the efficiency of media streaming by optimizing the base P2P library and creating its mobile integration (p2pml-kotlin), which ensures full compatibility with web clients.

The work provides a comprehensive analysis of the subject area, including modern P2P streaming technologies, existing commercial and open-source solutions, and alternative architectural approaches. Based on this analysis, a series of optimizations were made to the base P2P Media Loader library, aimed at improving resource management and increasing stability. The key result of the work is the designed and developed mobile integration library, p2pml-kotlin, which solves the task of integrating web-oriented P2P technologies into a native Android environment, while guaranteeing full compatibility with existing web clients.

The effectiveness of the developed system was confirmed experimentally. The results demonstrated successful cross-platform interaction and the ability to offload the CDN by up to 96% in the presence of active peers, with a moderate and controlled load on the mobile device's resources. The developed library is a ready-to-use open-source solution for Android developers.

KEYWORDS: P2P, STREAMING, WEBRTC, KOTLIN, ANDROID, WEBVIEW, HLS, CDN.

					ИПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
						4

ЗМІСТ

РЕФЕРАТ	3
ABSTRACT	4
ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ.....	6
ВСТУП	8
РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ: ПОТОКОВА ПЕРЕДАЧА МЕДІА ТА P2P-ТЕХНОЛОГІЇ.....	11
1.1. Сучасні протоколи та технології потокової передачі медіа.....	11
1.2. P2P-технології для потокової передачі медіа: принципи, засоби та існуючі рішення.....	14
1.3. Архітектура й принципи роботи бібліотеки P2P Media Loader.....	19
1.4. Аналіз вимог та обґрунтування вибору інструментальних засобів.	23
Висновки до розділу 1	27
РОЗДІЛ 2. РОЗРОБКА ТА ОПТИМІЗАЦІЯ СИСТЕМИ P2P-ДОСТАВКИ МЕДІА ДЛЯ МОБІЛЬНИХ ПЛАТФОРМ НА ОСНОВІ P2P MEDIA LOADER.....	29
2.1. Оптимізація P2P Media Loader для роботи у WebView та управління ресурсами	29
2.2. Архітектура мобільної інтеграції.....	33
2.3. Реалізація ключових механізмів	44
Висновки до розділу 2	56
РОЗДІЛ 3. ТЕСТУВАННЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ РОЗРОБЛЕНОЇ СИСТЕМИ	58
3.1. Методика та середовище тестування	58
3.2. Результати тестування та їх аналіз	60
Висновки до розділу 3	69
ВИСНОВКИ.....	71
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	73

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

ABR – Adaptive Programing Interface (адаптивна потокова передача даних)

API – Application Programming Interface (інтерфейс прикладного програмування)

CDN – Content Delivery Network (мережа доставки контенту)

XML – Extensible Markup Language (розширювана мова розмітки)

WebRTC – Web Real-Time Communication (веб-комунікація в реальному часі)

URL – Uniform Resource Locator (уніфікований локатор ресурсів)

UML – Unified Modeling Language (уніфікована мова моделювання)

TURN – Traversal Using Relays around NAT (проходження з використанням ретрансляторів навколо NAT, компонент WebRTC)

STUN – Session Traversal Utilities for NAT (утиліти для проходження сесій через NAT, компонент WebRTC)

SDP – Session Description Protocol (протокол опису сесії, компонент WebRTC)

SCTP – Stream Control Transmission Protocol (протокол передачі з керуванням потоком)

RTSP – Real-Time Streaming Protocol (протокол потокової передачі в реальному часі)

NAT – Network Address Translation (механізм перетворення мережевих адрес)

MSE – Media Source Extensions (розширення API для відтворення медіа в браузерях)

MPEG – Moving Picture Experts Group (група експертів з рухомих зображень, що розробляє стандарти)

JSON – JavaScript Object Notation (текстовий формат обміну даними)

ICE – Interactive Connectivity Establishment (інтерактивне встановлення з'єднань, компонент WebRTC)

HTTPS – HyperText Transfer Protocol Secure (захищений протокол передачі гіпертексту)

HTTP – HyperText Transfer Protocol (протокол передачі гіпертексту)

HLS – HTTP Live Streaming (протокол потокової передачі медіа від Apple)

DRM – Digital Rights Management (управління цифровими правами)

DASH – Dynamic Adaptive Streaming over HTTP (динамічний адаптивний стрімінг через HTTP, стандарт MPEG-DASH)

DHT – Distributed Hash Table (розподілена хеш-таблиця)

KMP – Kotlin Multiplatform (технологія для створення кросплатформних додатків на Kotlin)

RAM – Random-Access Memory (оперативна пам'ять)

CPU – Central Processing Unit (центральний процесор)

					ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
						7

ВСТУП

Цифровий світ неможливо уявити без потокового відео, яке стало домінуючим типом інтернет трафіку. Частка відео контенту в глобальному трафіку продовжує стрімко зростати, особливо через поширення мобільних пристроїв та підвищення якості контенту. Зазвичай для доставки медіаконтенту використовуються Content Delivery Network, які кешують дані ближче до кінцевих користувачів. Однак у CDN є суттєві недоліки: висока вартість обслуговування для постачальників контенту, проблеми масштабування під час пікових навантажень та обмежена ефективність у регіонах з недостатньо розвиненою інфраструктурою.

Альтернативним та перспективним підходом є використання peer-to-peer технологій для передачі медіа. У P2P-системах користувачі, які переглядають однаковий контент, обмінюються його фрагментами напряму між собою, значно знижуючи навантаження на сервери джерела та потенційно покращуючи швидкість завантаження для інших учасників мережі.

Одним із таких рішень є бібліотека P2P Media Loader, яка дозволяє інтегрувати P2P-доставку у вебплеєри. Проте, для забезпечення максимальної ефективності та стабільності, подібні бібліотеки потребують постійної оптимізації алгоритмів обміну даними, управління ресурсами та виправлення помилок, що виникають у реальних умовах експлуатації.

Окремим важливим завданням є адаптація P2P-стрімінгу для мобільних платформ. Мобільні пристрої мають специфічні обмеження, а інтеграція JavaScript орієнтованих P2P-бібліотек у нативні мобільні додатки становить нетривіальну технічну проблему. Розробка ефективного та легкого у використанні рішення для мобільної інтеграції P2P-Media Loader, яке б зберігало сумісність з існуючими вебклієнтами, є актуальною науково-практичною задачею.

Таким чином, **актуальність теми** даної роботи полягає дослідженні методів оптимізації P2P-системи передачі медіа та розробка її ефективної

мобільної інтеграції у мобільні платформи, що спрямовано на підвищення масштабованості, надійності та економічності систем потокового відео.

Метою випускної кваліфікаційної роботи бакалавра є підвищення ефективності системи потокової передачі медіа шляхом оптимізації відкритої бібліотеки P2P Media Loader та розробка і дослідження її мобільної інтеграції для платформи Android, що забезпечує сумісність клієнтів між веб та мобільною платформами.

Практичним завданням кваліфікаційної роботи є:

1. Провести аналіз предметної області, включаючи технології потокової передачі медіа, принципи роботи P2P-мереж, а також оцінити існуючі комерційні та відкриті рішення для P2P-стрімінгу.
2. Адаптувати та вдосконалити алгоритми базової бібліотеки P2P Media Loader з метою ефективного використання системних ресурсів та забезпечення її стабільної роботи в мобільному середовищі.
3. Спроектувати та розробити архітектуру бібліотеки мобільної інтеграції p2pml-kotlin для платформи Android, обґрунтувавши вибір ключових технологій та реалізувавши механізми взаємодії нативних компонентів з P2P-ядром.
4. Розробити методику тестування та провести експериментальну оцінку ефективності системи, зокрема проаналізувавши показники розвантаження CDN та вплив на системні ресурси, і порівняти отримані результати з традиційним методом доставки контенту.

Виконання цих етапів дозволить створити оптимізовану систему P2P-доставки медіа, адаптовану для мобільних пристроїв, та надати рекомендації щодо її практичного застосування.

Об'єктом дослідження є системи ефективної потокової передачі медіа на основі гібридних P2P-мереж.

Предметом дослідження є принципи та моделі побудови ефективних P2P-систем доставки медіа, методи їх оптимізації для мобільних платформ та архітектурні патерни інтеграції з нативними додатками.

					ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
						9

За тематикою теми дипломної роботи були опубліковані тези [1].

					ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
						10

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ: ПОТОКОВА ПЕРЕДАЧА МЕДІА ТА P2P-ТЕХНОЛОГІЇ

1.1. Сучасні протоколи та технології потокової передачі медіа

Потокова передача медіа стала домінуючим способом споживання аудіо та відео в Інтернеті. На відміну від звичайного повного завантаження файлу перед початком відтворення, стрімінг дозволяє користувачеві почати перегляд майже одразу, оскільки контент передається безперервним потоком і відтворюється по мірі надходження даних. Ранні технології стрімінгу часто використовували спеціалізовані протоколи, які могли блокуватися мережевими екранами. Сучасні підходи переважно базуються на стандартному протоколі HTTP(S), що забезпечує кращу сумісність та проходження через корпоративні та домашні мережі.

Ключовою інновацією, що дозволила значно покращити якість сприйняття для користувачів з різною швидкістю інтернет-з'єднання, стала адаптивна потокова передача даних (Adaptive Bitrate Streaming) [3]. Суть ABR полягає в тому, що контент кодується в кількох варіантах з різною якістю, а медіаплеєр на стороні клієнта динамічно оцінює умови мережі, щоб автоматично запитувати сегменти оптимальної якості та уникати буферизації. Для цього використовуються різні алгоритми. Наприклад, підходи, що базуються на пропускній спроможності, аналізують швидкість завантаження попередніх сегментів. Вони швидко реагують на зміни, але можуть бути схильні до коливань якості. На противагу їм, алгоритми, орієнтовані на буфер, прагнуть підтримувати певний його обсяг, перемикаючи якість залежно від заповнення. Це забезпечує стабільніше відтворення, хоча й може уповільнювати реакцію на різкі зміни в мережі. Існують також і гібридні методи, які для прийняття рішення комбінують різні метрики. Незалежно від алгоритму, головна мета ABR – забезпечити безперервне відтворення без видимих пауз, навіть ціною тимчасового зниження якості відео. Двома популярними протоколами, що реалізують ABR через HTTP, є HLS та DASH.

HTTP Live Streaming

Розроблений Apple, протокол HLS спочатку мав найкращу нативну підтримку на пристроях цієї компанії, але згодом став широко використовуватися на різноманітних платформах. Процес роботи полягає в тому, що на сервері відеопотік сегментується на короткі медіафайли, зазвичай у форматі Transport Stream, тривалістю від 2 до 10 секунд. Для керування цими фрагментами генерується текстовий файл-плейлист у форматі M3U8, який містить посилання на медіасегменти та їхні метадані.

Для реалізації адаптивного бітрейту створюється так званий "майстер-плейлист", що містить посилання на окремі плейлисти для кожного доступного рівня якості. Клієнтський плеєр спочатку завантажує цей майстер-плейлист, обирає з нього потік відповідної якості та починає послідовно завантажувати й відтворювати сегменти. Плеєр періодично оновлює плейлист, щоб отримувати посилання на нові сегменти та адаптуватися до умов мережі, перемикаючись між потоками різної якості.

Перевагами HLS є дуже широка підтримка на пристроях Apple та багатьох інших платформах, відносна простота реалізації та надійне проходження через фаєрволи. До недоліків відносять вищу затримку порівняно з іншими методами, хоча це частково вирішується розширенням Low-Latency HLS, яке дозволяє знизити затримку до 2-5 секунд. Також HLS має меншу гнучкість у виборі кодеків порівняно з DASH.

Dynamic Adaptive Streaming over HTTP

MPEG-DASH є міжнародним стандартом, орієнтованим на уніфікацію та гнучкість адаптивного стрімінгу. Його робота схожа на HLS, зокрема контент так само сегментується, але головна відмінність полягає в тому, що DASH є кодек-агностичним, тобто не прив'язаний до конкретних відео- чи аудіокодеків.

Замість плейлиста .m3u8 тут використовується маніфестний файл у форматі XML, що називається Media Presentation Description. Цей файл детально описує доступні варіанти якості, кодеки, URL-адреси сегментів та іншу

					ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
						12

необхідну інформацію. Клієнтський плеєр завантажує .mpd файл, аналізує та, аналогічно до HLS, вибирає відповідні сегменти для завантаження і відтворення, спираючись у веббраузерах на Media Source Extensions (MSE) API.

Перевагами DASH є стандартизація, відкритість та незалежність від кодеків, що дає більшу гнучкість, а також потенційно нижча затримка порівняно з HLS. Серед недоліків — гірша нативна підтримка на пристроях Apple та складніша структура XML-маніфесту порівняно з текстовим плейлистом .m3u8.

Роль CDN у потоковій передачі медіа

Ефективна доставка HLS та DASH потоків у великих масштабах неможлива без мереж доставки контенту [4]. CDN оптимізують доставку, кешуючи маніфести та медіасегменти на серверах, розташованих близько до кінцевих користувачів. Окрім базового кешування та зменшення затримки, сучасні CDN надають розширені функції, критичні для стрімінгу:

По-перше, вміння CDN швидко нарощувати ресурси грає ключову роль у цих системах. Під час пікових навантажень платформа може серйозно перевантажитися, та завдяки розподіленим сховищам контенту мережа ефективно «поглинає» пік трафіку, що дозволяє обслуговувати одночасно навіть мільйони глядачів без гальм. У стрімінгу це означає відсутність раптових буферизацій і зривів у трансляції.

По-друге, сучасні CDN ретельно дбають про безпеку. Вони не тільки відсікають підозрілий трафік під час розподілених атак відмови в обслуговуванні, а й підтримують складні механізми токен-автентифікації, щоб жоден сторонній користувач не міг перехопити чи підмінити URL-адресу потоку. Багато провайдерів навіть інтегрують власні DRM-модулі, що забезпечують додатковий рівень захисту відеоконтенту.

Окремо варто згадати про можливості граничних обчислень (edge computing), які сьогодні підтримують лідери ринку CDN. Замість того щоб відправляти запит назад до центрального дата-центру, система може виконувати певну обробку (наприклад, динамічно змінювати маніфести, перевіряти

авторизацію чи вставляти таргетовану рекламу) безпосередньо на найближчому сервері до користувача. Це зменшує загальний час відповіді й дає змогу оперативніше реагувати на індивідуальні запити.

Крім того, аналітичні можливості CDN ускладнилися настільки, що вони перетворилися на потужний інструмент для маркетологів і технічних фахівців. Збір інформації про географію глядачів, якість перегляду, відсоток успішних підключень та середню тривалість сесії дозволяє вчасно виявляти прогалини системи й покращувати її роботу. Реальні кейси показують, що саме на основі даних стрімінгові сервіси оптимізують алгоритми відтворення й мінімізують затримки на певних ділянках мережі.

Таким чином, CDN є критично важливим компонентом інфраструктури, що забезпечує не лише швидку та надійну доставку HLS/DASH контенту, але й надає додаткові інструменти для безпеки, аналітики та оптимізації стрімінгових сервісів.

1.2. P2P-технології для потокової передачі медіа: принципи, засоби та існуючі рішення

Peer-to-peer мережі, що функціонують на принципах децентралізації та рівноправності вузлів [5], кардинально відрізняються від класичної клієнт-серверної моделі. У них кожен учасник одночасно є і клієнтом, і сервером, що дозволяє ефективно розподіляти навантаження та зменшувати залежність від централізованої інфраструктури. Такий підхід забезпечує важливі переваги, серед яких чудова масштабованість (ефективність зростає з кількістю користувачів), висока стійкість до відмов та значне зниження витрат на сервери. Однак ці переваги мають і зворотний бік: децентралізований характер ускладнює контроль якості сервісу, впровадження надійних механізмів безпеки та загальне керування мережею.

Основні типи P2P-мереж

Коли говорять про класифікацію P2P-мереж за структурою, найчастіше виокремлюють три види мереж.

Перший тип – неструктуровані мережі [5]. У таких системах піри підключаються до випадково обраних сусідів, без жорсткої організації. Щоб знайти потрібний файл або ресурс, програма розсилає запит усім вузлам навколо або ж «мандрує» мережевими вузлами навмання – коли кожний крок обирається випадково. Хоч це й доволі проста ідея, на практиці вона не гарантує, що шуканий пір знайдеться навіть за умови, що він присутній у мережі. До того ж, такий спосіб створює надлишковий трафік і може суттєво навантажувати канали зв'язку.

Другий підхід – структуровані мережі, які працюють за допомогою розподілених хеш-таблиць [5]. Кожний файл чи ресурс отримує унікальний ідентифікатор завдяки хешуванню, і за цим ключем система одразу «знає», на яких вузлах шукати потрібний контент. Внаслідок цього запитами забезпечується впорядкована та оптимізована маршрутизація, що усуває необхідність перенавантажувати мережу або здійснювати випадкові пошуки. Достатньо всього кількох кроків, щоб одразу потрапити до джерела необхідної інформації. Це, у свою чергу, дозволяє масштабувати систему з мінімальними витратами трафіку.

Також, існують гібридні моделі, які поєднують особливості централізованих і децентралізованих рішень [5]. Тут зазвичай є один або кілька головних серверів, які ведуть реєстр наявних файлів і вказують, на яких пірах розташована потрібна інформація. Проте, обмін безпосередньо відбувається між пірами. Завдяки такій «змішаній» архітектурі спрощується початковий пошук: не доводиться опитувати всю мережу, достатньо звернутися до центрального каталогу. З іншого боку, наявність централізованого компонента може стати єдиною точкою відмови та потенційним «вузьким місцем» системи.

Гібридний підхід є найпоширенішим серед сучасних систем P2P-стрімінгу, оскільки початкові сегменти для швидкого старту відтворення контенту

					ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
						15

завантажуються через HTTP, а основна частина даних – через P2P. Якщо P2P-доставка з якихось причин не вдається, система автоматично може перемкнутися на завантаження по HTTP. Такий підхід дає змогу економити на інфраструктурі при чому зберігаючи комфорт користувачів.

Застосування P2P-мереж для передачі медіаконтенту

P2P-мережі ефективно застосовуються для передачі великих медіафайлів, потокового відео та аудіо. Популярність таких мереж для стрімінгу пов'язана з можливістю суттєво знизити навантаження на сервери за рахунок передачі медіаконтенту безпосередньо між користувачами, що переглядають один і той самий контент.

Типова схема роботи P2P-стрімінгу:

- медіафайл сегментується на невеликі сегменти;
- інформація про контент та способи знаходження зазвичай міститься в метафайлі;
- піри завантажують сегменти одночасно з централізованих серверів та один від одного, використовуючи P2P;
- нові користувачі в мережі отримують доступ до сегментів із мінімальними затримками за рахунок високої доступності сегментів у мережі.

WebRTC як ключова технологія для реалізації P2P-стрімінгу

Web Real-Time Communication є відкритою технологією, яка дозволяє браузерам та мобільним додаткам встановлювати прямі з'єднання для обміну мультимедійними потоками в режимі реального часу [6]. WebRTC містить набір стандартів та API, що підтримуються більшістю сучасних браузерів, і слугує основою для численних P2P-додатків. Одним із ключових компонентів для передачі контенту є RTCDataChannel [6], який забезпечує двонаправлений канал для передачі довільних даних між двома пірами. Важливою особливістю RTCDataChannel є можливість гнучкого налаштування рівня надійності й

					ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
						16

порядку доставки повідомлень завдяки використанню протоколу SCTP (Stream Control Transmission Protocol).

Першим етапом налагодження такого P2P-каналу є обмін метаданими, відомий як сигналізація. Браузери чи додатки відправляють одне одному повідомлення SDP-Offer і SDP-Answer, погоджуючи, яким чином шифрувати потік і через які порти й IP-адреси вони можуть спілкуватися. Оскільки WebRTC не диктує, як саме передавати ці SDP-повідомлення, розробник зазвичай пише власний невеликий сервер на вебсокетах, який просто ретранслює такі повідомлення від одного користувача до іншого.

Як тільки SDP-обмін відбувся, настає час для Interactive Connectivity Establishment (ICE) [7] — процесу, що шукає найкоротший і найстабільніший шлях через інтернет, навіть якщо обидва піри перебувають за Network Address Translation чи брандмауером. Щоб «розкрити» зовнішню адресу кожного клієнта, використовується STUN-сервер. Якщо ж мережеві обмеження не дозволяють встановити прямий канал, операція передається TURN-серверу. TURN фактично виступає як ретранслятор: весь трафік іде через нього, але це дозволяє підтримувати зв'язок навіть у найскладніших мережевих сценаріях.

WebRTC є важливою технологією для реалізації ефективних P2P-систем передачі медіаконтенту, забезпечуючи швидку, безпечну та сумісну між платформами передачу даних. Однак, для ефективного масштабування і стабільності роботи систем, необхідно застосовувати додаткові рішення та механізми оптимізації, які враховують особливості WebRTC.

Огляд та аналіз існуючих P2P-рішень

Ринок технологій peer-to-peer для передачі медіаконтенту активно розвивається останні десятиліття. Значна кількість компаній та проектів створюють рішення, що використовують переваги P2P для оптимізації доставки потокового контенту. Аналіз конкурентів у цій сфері дозволяє чітко зрозуміти місце та перспективи P2P систем, а також визначити ключові напрямки її розвитку.

					ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
						17

Аналіз ринку P2P-рішень для потокової передачі медіа виявляє кількох вагомих комерційних гравців, як історичних, так і сучасних.

Одним із ключових історичних гравців є Streamroot, який нині інтегрований у CDN Lumen після його придбання компанією CenturyLink у 2019 році. Технологічно Streamroot був піонером у застосуванні WebRTC для передачі відео між браузером без потреби у додаткових плагінах. Компанія використовувала гібридний підхід, де традиційний HTTP CDN доповнювався P2P-компонентом для зниження навантаження в пікові моменти, а наявність мобільних SDK дозволяла охоплювати і мобільні пристрої. Ключовими перевагами цього рішення були висока економія трафіку CDN, що сягала 70-80%, та простота інтеграції з популярними вебплеєрами. З іншого боку, до недоліків відносили комерційну закритість алгоритмів та високу вартість використання.

Іншим важливим конкурентом був Peer5, якого у 2021 році придбала Microsoft для інтеграції в Microsoft Teams як корпоративну технологію eCDN. Peer5, як і Streamroot, використовував гібридну P2P-систему, але пропонував рішення виключно для вебсередовища, де їхня система автоматично оптимізовувала завантаження сегментів. Рішення вирізнялося надзвичайною ефективністю, забезпечуючи до 90% розвантаження CDN, та повною сумісністю з HLS і MPEG-DASH. Проте після поглинання Microsoft воно стало повністю закритим і недоступним для зовнішніх користувачів поза корпоративним сегментом.

На противагу цим уже поглинутим компаніям, на ринку досі активно працює SwarmCloud, що має широку клієнтську базу. Ця компанія також робить ставку на WebRTC для створення peer-to-peer мережі між глядачами з метою розвантаження CDN. До головних переваг SwarmCloud відносять широку мультиплатформну підтримку (Web, Android, iOS), гнучкі налаштування P2P-мережі та детальну аналітику в реальному часі. Водночас рішення є комерційно закритим, а його налаштування може бути складним для дрібних проектів.

Проведений аналіз провідних комерційних рішень для P2P-доставки медіаконтенту демонструє чітку тенденцію до використання технології WebRTC як основи для побудови гібридних P2P+CDN систем. Такі системи довели свою ефективність у зниженні навантаження на CDN та покращенні масштабованості, особливо під час пікових навантажень. Вони пропонують підтримку стандартних протоколів HLS та DASH, інтеграцію з популярними плеєрами та, у багатьох випадках, SDK для мобільних платформ.

Однак, ключовими обмеженнями цих комерційних рішень є їхня комерційна закритість, неможливість глибокої кастомізації та вартість, що може бути бар'єром для невеликих проєктів, стартапів або освітніх цілей.

Ці обмеження підкреслюють важливість та актуальність відкритих бібліотек, які надають розробникам гнучкі, прозорі та доступні інструменти для впровадження P2P-оптимізації. Однією з таких бібліотек, що фокусується на інтеграції з HLS/DASH стрімінгом у вебсередовищі, є P2P Media Loader. Її архітектура, принципи роботи, є предметом детального розгляду в наступному підрозділі.

1.3. Архітектура й принципи роботи бібліотеки P2P Media Loader

P2P Media Loader – це JavaScript бібліотека з відкритим вихідним кодом, розроблена для інтеграції peer-to-peer технологій у процес потокової передачі медіа безпосередньо у веббраузерах [10]. Основна мета бібліотеки – знизити навантаження на CDN та покращити масштабованість стрімінгових сервісів шляхом прямого обміну медіасегментами між користувачами, які переглядають однаковий контент.

Ключовою особливістю P2P Media Loader є її орієнтація на легку інтеграцію з популярними JavaScript-плеєрами, що підтримують стандарти HLS та DASH, такими як Hls.js та Shaka Player. Бібліотека використовує сучасні вебтехнології, зокрема WebRTC, для встановлення P2P-з'єднань та обміну даними, і функціонує без необхідності встановлення користувачем будь-яких додаткових плагінів чи розширень браузера.

P2P Media Loader реалізує гібридну модель доставки контенту, де P2P-мережа працює паралельно з традиційною HTTP-доставкою з CDN. Це означає, що бібліотека не замінює CDN повністю, а доповнює його виконуючи наступні функції:

- завантаження початкових сегментів потоку через HTTP для швидкого старту відтворення;
- спроби завантаження наступних сегментів через P2P-мережу від інших пірів;
- автоматичне перемикання на завантаження через HTTP, якщо сегмент недоступний у P2P-мережі або якщо P2P-завантаження відбувається надто повільно;
- надання завантажених сегментів іншим пірам у мережі.

Такий підхід гарантує надійність — відеоконтент відтворюватиметься навіть за відсутності P2P, а також ефективність — знижує навантаження на CDN, коли P2P працює.

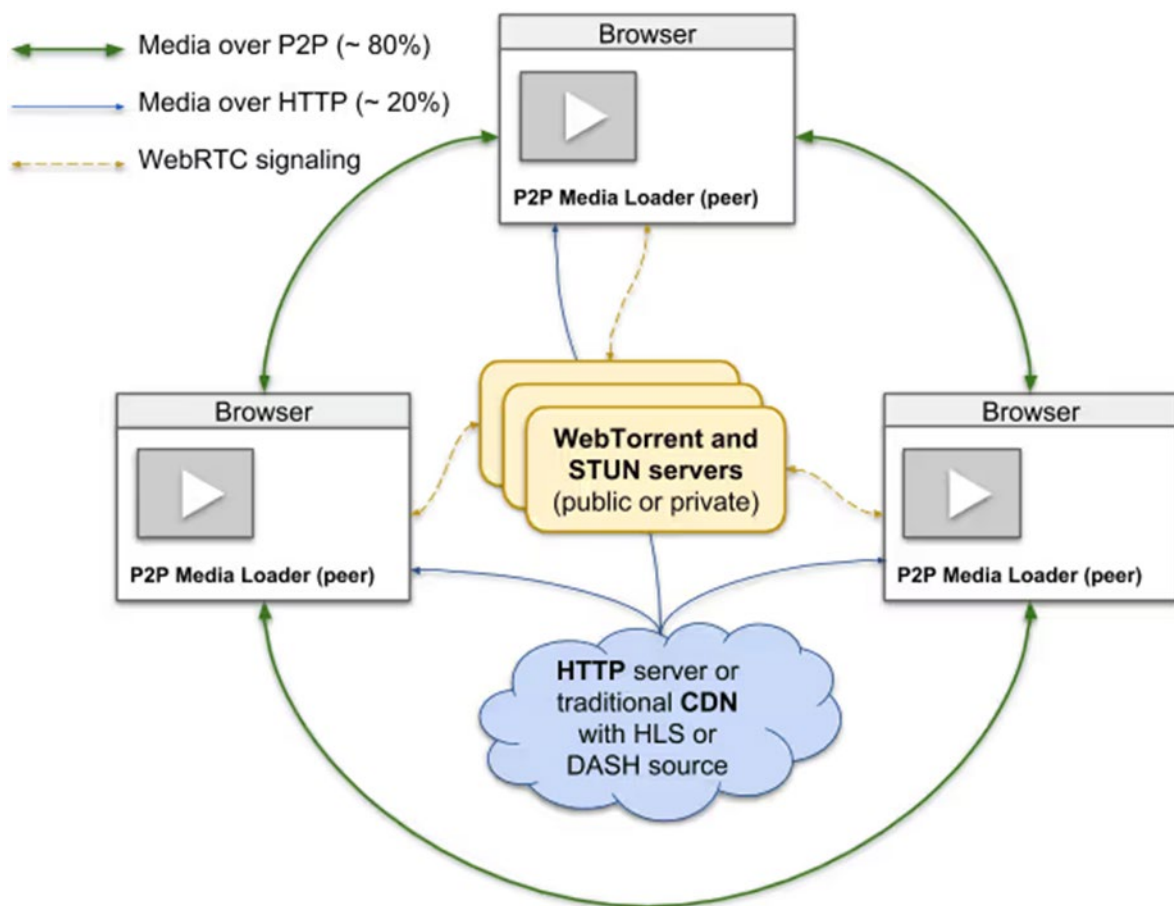


Рис. 1.1 – Принцип роботи P2P Media Loader

Процес роботи бібліотеки P2P Media Loader, загальну схему якого проілюстровано на рисунку 1.1, можна описати як послідовність скоординованих дій. Все починається з ініціалізації, коли розробник створює екземпляр бібліотеки, передаючи їй необхідні конфігураційні параметри, такі як список URL-адрес трекерів для пошуку пірів, налаштування STUN/TURN серверів для проходження NAT, та параметри інтеграції з конкретним медіаплеєром, наприклад, Hls.js або Shaka Player. На цьому етапі інтеграційний шар бібліотеки підключається до плеєра, готуючись перехоплювати його запити.

Коли відтворення ініційовано і медіаплеєр, обробивши маніфест HLS або DASH, намагається завантажити черговий відеосегмент за його стандартним HTTP URL, інтеграційний шар бібліотеки перехоплює цей запит. У цей момент P2P-рушій активує механізм пошуку інших учасників мережі. Для цього він підключається до одного або кількох трекерів зі списку конфігурації через протокол WebSockets, надсилаючи свій унікальний ідентифікатор (peerId) та

ідентифікатор контенту (swarmId). У відповідь трекер надає список інших пірів, що дивляться той самий контент. Цей процес періодично повторюється для підтримки актуального списку учасників рою. Отримавши список потенційних пірів, P2P Media Loader намагається встановити з ними пряме з'єднання за допомогою RTCPeerConnection, використовуючи трекер як сигнальний сервер для обміну метаданими, необхідними для узгодження параметрів WebRTC-з'єднання.

Ключовим етапом є обробка перехопленого запиту на сегмент, яка відбувається за чітким пріоритетним алгоритмом. В першу чергу, бібліотека перевіряє наявність потрібного сегмента у власному локальному кеші. Якщо сегмент знайдено, він негайно повертається плеєру. Якщо ж сегмента в кеші немає, P2P Media Loader надсилає запити на його наявність усім активним P2P-пірам. У випадку успішної відповіді від одного чи кількох пірів, сегмент завантажується напряму від них через захищений канал RTCDataChannel. І лише якщо сегмент недоступний у пірів, або P2P-завантаження не вдається чи триває надто довго, система автоматично перемикається на резервний механізм і завантажує цей сегмент за його оригінальним HTTP URL з CDN чи сервера-джерела. Після будь-якого успішного завантаження, чи то через P2P, чи через HTTP, сегмент зберігається у внутрішньому кеші для подальшого відтворення та для роздачі іншим учасникам мережі.

Важливою частиною роботи бібліотеки є і зворотний процес – надання власних закешованих сегментів іншим пірам на їхні запити, що також відбувається через RTCDataChannel. Для підтримки стабільності та ефективності P2P-рою бібліотека постійно управляє активними з'єднаннями, відключаючи неактивних пірів та періодично оновлюючи список доступних учасників через запити до трекера.

1.4. Аналіз вимог та обґрунтування вибору інструментальних засобів.

Розробка мобільного P2P-рішення вимагала чіткого визначення цільової аудиторії, сценаріїв використання та набору функціональних і нефункціональних вимог. Ці елементи стали основою для вибору архітектурного підходу та інструментальних засобів.

Цільові користувачі та варіанти використання

Основним цільовим користувачем розробленої бібліотеки `p2pml-kotlin` є Android-розробник, який прагне інтегрувати P2P-функціонал для потокової передачі медіа у свій мобільний додаток. Кінцевий користувач мобільного додатку опосередковано отримує переваги від роботи бібліотеки.

З точки зору Android-розробника, ключові можливості бібліотеки полягатимуть у тому, щоб легко інтегрувати складний P2P-функціонал. Розробник матиме можливість легко налаштовувати параметри P2P-сесії, наприклад, вказувати ідентифікатор рою, адреси трекерів, встановлювати ліміти для кешування та керувати іншими специфічними налаштуваннями ядра P2P Media Loader.

Бібліотека має забезпечити взаємодію P2P-механізму з популярними нативними Android-плеєрами. Це досягається шляхом надання плеєру спеціального, модифікованого URL-адреси маніфесту, завдяки чому плеєр працює зі звичним для нього потоком даних, не "знаючи" про P2P-складову.

Також, розробник матиме змогу програмно контролювати P2P-активність: він зможе запускати та зупиняти її в потрібний момент, а також динамічно змінювати деякі робочі параметри безпосередньо під час роботи додатку. Для забезпечення гнучкості та можливості реагувати на зміни, бібліотека буде надсилати сповіщення про ключові події, що відбуваються в P2P-ядрі, дозволяючи розробнику належним чином обробляти їх у своєму додатку.

Функціональні вимоги

До розробленої системи мобільної P2P-інтеграції було висунуто наступні функціональні вимоги:

- повинна забезпечувати гібридну доставку HLS медіасегментів шляхом поєднання P2P-обміну та традиційного HTTP-завантаження;
- повинна дозволяти мобільним Android-клієнтам взаємодіяти з вебклієнтами, що використовують оригінальну JavaScript-бібліотеку P2P Media Loader;
- повинна надавати гнучке Kotlin API для Android-розробників для ініціалізації, запуску, зупинки та конфігурації P2P-функціоналу;
- повинна перехоплювати запити на медіасегменти від нативного медіаплеєра та спрямовувати їх на обробку P2P-рушієм;
- повинна забезпечувати fallback-механізм до HTTP у випадку неможливості отримання сегмента через P2P;
- повинна дозволяти конфігурувати можливість вимкнення вивантаження сегментів у P2P-мережу мобільним клієнтом;
- повинна коректно обробляти життєвий цикл Android-компонентів для запобігання витоків ресурсів.

Нефункціональні вимоги

Окрім функціональних аспектів, було визначено наступні нефункціональні вимоги:

- забезпечити значущий відсоток розвантаження CDN за наявності достатньої кількості пірів;
- мінімізувати вплив роботи P2P-механізму на час початкового завантаження відео;
- зменшити ймовірність та тривалість подій буферизації порівняно з сценаріями обмеженої пропускної здатності CDN;
- обмежити додаткове навантаження на CPU мобільного пристрою;

					ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
						24

- контролювати використання оперативної пам'яті, особливо в середовищі WebView;
- надати можливість контролювати споживання трафіку;
- гарантувати стабільне функціонування на платформі Android версії 9 або вище;
- забезпечити сумісність з нативним плеєром ExoPlayer;
- забезпечити стабільну роботу P2P-механізму та коректне перемикання на HTTP у випадку збоїв P2P-з'єднань.

API бібліотеки `p2pml-kotlin` має бути простим, зрозумілим та добре документованим для легкої інтеграції в існуючі Android-проекти.

Обґрунтування вибору архітектурного рішення та інструментальних засобів

Основною цільовою платформою для першої реалізації мобільної інтеграції було обрано Android, як найпоширенішу мобільну ОС у світі. Мовою розробки було обрано Kotlin, оскільки це сучасна, офіційно підтримувана мова для розробки під Android, що пропонує кращу продуктивність та інтеграцію з системою порівняно з кросплатформними фреймворками для багатьох сценаріїв [15]. Вибір Kotlin також відкривав перспективу для майбутньої міграції на Kotlin Multiplatform, що потенційно дозволило б розширити розроблену бібліотеку для підтримки iOS з мінімальним дублюванням коду бізнес-логіки.

Перед початком розробки `p2pml-kotlin` ми стояли перед вибором між двома кардинально різними підходами до реалізації P2P-функціоналу на мобільній платформі. Перший полягав у повному нативному переписі P2P-логіки та реалізації WebRTC-стеку «з нуля». Хоча такий варіант міг би дати максимальну оптимізацію під конкретну платформу, він вимагав би величезних витрат часу на розробку та тестування складних P2P-протоколів, а крім того, передбачав би необхідність паралельної підтримки двох кодових баз – нативної та JavaScript-бібліотеки P2P Media Loader. З огляду на це, ми вирішили відмовитися від цього

					ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
						25

шляху. Другий підхід полягав у перевикористанні вже готового JavaScript-ядра P2P Media Loader. Він дозволяв значно пришвидшити розробку, уникнути дублювання добре протестованої логіки та, найголовніше, гарантував повну протокольну сумісність мобільних клієнтів із вебклієнтами.

Щойно ми ухвалили рішення використовувати JavaScript-ядро, перед командою постало завдання обрати середовище, в якому воно виконуватиметься на Android, і забезпечити ефективну взаємодію з нативною частиною додатка. Для цього ми розгорнули прихований WebView – стандартний компонент Android, налаштований так, щоб працювати без відображення користувачу та при цьому надавати повноцінне середовище виконання JavaScript із підтримкою WebRTC API. Завдяки цьому не довелося писати складну P2P-логіку «з нуля», а мобільні клієнти використовують той самий код ядра, що й вебверсія.

Нативна частина додатка також потребувала інструментів для HTTP-взаємодії та обробки медіа. Для організації взаємодії між нативним плеєром і P2P-логікою у WebView ми обрали Ktor – асинхронний Kotlin-фреймворк. Усередині Android-додатка він запускає легковагий HTTP-сервер на localhost, який спочатку завантажує в WebView необхідні HTML- та JavaScript-файли, а потім проксує запити плеєра: залежно від наявності пір сегмент може завантажуватися через P2P-мережу або стандартним HTTP-запитом. Для прямих HTTP-запитів із Kotlin-частини (наприклад, під час початкової завантаження маніфестів або в разі повного відключення P2P-механізму) ми використовуємо OkHttp, що є надійним і поширеним клієнтом для Android. А щоб не створювати власний HLS-парсер, ми скористалися готовим рішенням з бібліотеки Media3 – HlsPlaylistParser, що забезпечило коректну обробку плейлистів і полегшило модифікацію URL-адрес сегментів.

Водночас, обраний підхід з використанням WebView, попри значні переваги у швидкості розробки та сумісності, передбачав і певні потенційні виклики. До них можна віднести можливе додаткове навантаження на використання ресурсів самим компонентом WebView, а також специфіку

передачі даних між Kotlin та JavaScript середовищами. Ці аспекти враховувалися при подальшій реалізації p2pml-kotlin.

Таким чином, вибір архітектури, що базується на перевикористанні існуючого JavaScript-ядра P2P Media Loader у середовищі Headless WebView з використанням Ktor як локального проксі-сервера та Kotlin як основної мови розробки, був обґрунтований прагненням до швидкої реалізації, забезпечення повної сумісності з вебecosистемою та створення підґрунтя для майбутнього розвитку.

Висновки до розділу 1

Потокова передача медіаконтенту є основним методом споживання відео та аудіо в Інтернеті, і зростаюче використання мобільних пристроїв та високої якості контенту значно збільшує попит на ефективні методи доставки медіа. Традиційні мережі доставки контенту мають свої обмеження, зокрема високі витрати на інфраструктуру та труднощі зі збільшенням пропускної здатності під час пікових навантажень. У таких умовах альтернативним і перспективним підходом є застосування технологій peer-to-peer, що дозволяє значно знизити навантаження на сервери та підвищити ефективність доставлення медіаконтенту.

Аналіз P2P-технологій, зокрема з використанням WebRTC, та огляд існуючих рішень підтвердили їхню ефективність для потокового відео. Бібліотека P2P Media Loader виділяється як перспективне відкрите рішення для інтеграції P2P у вебплеєри. Однак, для забезпечення стабільності та масштабованості, особливо при адаптації для мобільних платформ, необхідні подальші оптимізації та розробка спеціалізованих рішень. Мобільні пристрої мають специфічні обмеження, що робить розробку ефективної мобільної інтеграції P2P Media Loader важливою науково-практичною задачею.

У рамках проведеного в першому розділі аналізу було також сформульовано вимоги до такої мобільної інтеграції та обґрунтовано вибір ключових технологій та архітектурного підходу для її реалізації на платформі Android. Зокрема, було прийнято рішення про перевикористання існуючого

JavaScript-ядра P2P Media Loader за допомогою Headless WebView, з використанням Kotlin для нативної обгортки. Розглянуті технології адаптивного бітрейту та принципи P2P-мереж закладають теоретичну основу для цих розробок.

Підсумовуючи, P2P-технології мають значний потенціал для оптимізації потокових сервісів, особливо в мобільних додатках. У наступних розділах цієї дипломної роботи буде детально описано процес оптимізації базової бібліотеки та розробку обґрунтованої мобільної інтеграції для платформи Android, спрямованих на покращення ефективності та стабільності доставки медіаконтенту.

РОЗДІЛ 2. РОЗРОБКА ТА ОПТИМІЗАЦІЯ СИСТЕМИ P2P-ДОСТАВКИ МЕДІА ДЛЯ МОБІЛЬНИХ ПЛАТФОРМ НА ОСНОВІ P2P MEDIA LOADER

2.1. Оптимізація P2P Media Loader для роботи у WebView та управління ресурсами

Незважаючи на потенціал P2P-технологій та існування таких бібліотек, як P2P Media Loader, залишається низка викликів, пов'язаних з їхньою ефективністю, стабільністю та використанням ресурсів, особливо в обмежених середовищах виконання, таких як мобільні WebView. Метою робіт, описаних у даному підрозділі, було здійснення низки оптимізацій та виправлень у базовій бібліотеці Novage/p2p-media-loader для покращення її ключових характеристик. Ці вдосконалення були спрямовані на підвищення продуктивності, зменшення споживання ресурсів, покращення стабільності роботи, підвищення надійності передачі даних та розширення функціональних можливостей бібліотеки.

Рефакторинг системи зберігання сегментів та впровадження динамічних лімітів пам'яті

Одним із ключових аспектів ефективності P2P-бібліотеки є механізм кешування завантажених медіасегментів у пам'яті пристрою. Ці сегменти використовуються як для відтворення плеєром, так і для передачі іншим пірам у мережі. Попередня реалізація сховища сегментів у бібліотеці P2P Media Loader мала низку обмежень, які ставали особливо помітними в умовах обмежених ресурсів, таких як середовище WebView на мобільних пристроях.

Зокрема, попередня реалізація використовувала просту структуру даних для зберігання об'єктів StorageItem, що містили дані сегмента (ArrayBuffer), метадані та мітку часу останнього доступу (lastAccessed). Логіка очищення кешу базувалася переважно на двох параметрах: часі зберігання сегмента та максимальній кількості сегментів у кеші. При перевищенні ліміту кількості сегментів видалялися найстаріші за часом останнього доступу, без урахування їхньої актуальності для поточного відтворення. Важливим недоліком було те, що

не існувало явного механізму контролю за загальним обсягом пам'яті, який займає кеш сегментів, що могло призводити до надмірного споживання ресурсів, особливо на пристроях з обмеженою RAM. Крім того, реалізація була монолітною, тісно пов'язаною з ядром бібліотеки, що ускладнювало її розширення або заміну на інші типи сховищ. Спрощена UML-діаграма класу SegmentsMemoryStorage, що ілюструє цю структуру, наведена на рисунку 2.1.

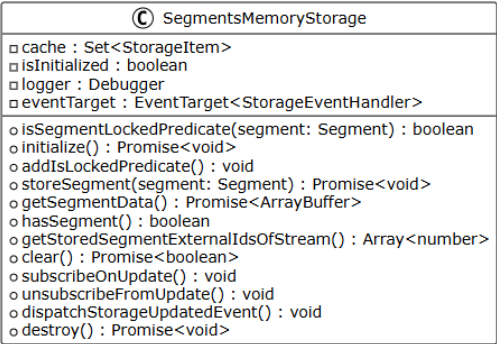


Рис. 2.1 – Спрощена UML-діаграма класу SegmentsMemoryStorage до рефакторингу

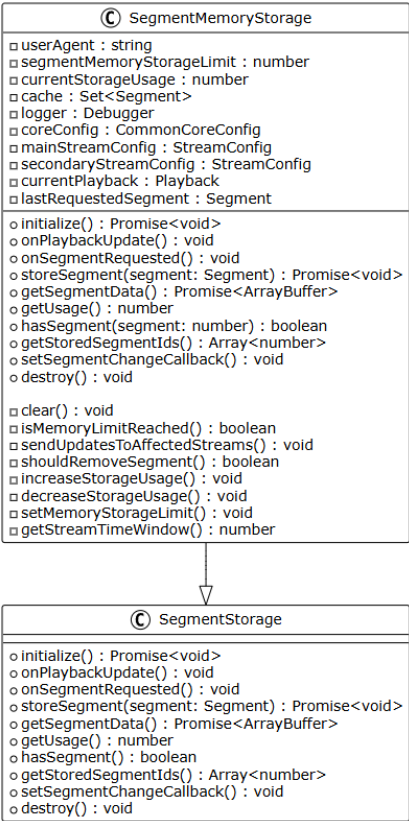


Рис. 2.2 – Спрощена UML-діаграма інтерфейсу SegmentStorage та класу SegmentMemoryStorage після рефакторингу

Необхідність рефакторингу була зумовлена виявленими недоліками попереднього підходу, а саме: ризиком неконтрольованого росту споживання пам'яті, неефективною стратегією витіснення сегментів з кешу (видалення потенційно потрібних для найближчого відтворення сегментів), та архітектурною негнучкістю. Особливо гостро ці проблеми проявлялися у WebView, де ресурси є ще більш обмеженими.

З метою вирішення зазначених проблем було проведено рефакторинг системи зберігання сегментів. Першою ключовою зміною стало створення інтерфейсу SegmentStorage, який визначає чіткий контракт взаємодії між ядром бібліотеки та механізмом зберігання. Цей інтерфейс включає методи для ініціалізації, збереження (storeSegment), отримання (getSegmentData), перевірки наявності (hasSegment), отримання інформації про використання ресурсів (getUsage) та інші необхідні операції. Такий підхід декапсулює логіку зберігання, роблячи архітектуру більш модульною та розширюваною. Це відкриває можливість легкої заміни реалізації сховища – наприклад, додати підтримку IndexedDB або інших механізмів зберігання, не змінюючи основну логіку бібліотеки.

Другою важливою зміною стало повне переписування класу SegmentMemoryStorage для реалізації нового інтерфейсу SegmentStorage та впровадження покращеної логіки управління кешем. Було введено поняття ліміту пам'яті для кешу (segmentMemoryStorageLimit) та механізм відстеження поточного розміру кешу (currentMemoryStorageSize). Цей ліміт може встановлюватися через конфігурацію ядра або, що важливо для мобільних пристроїв, динамічно визначатися на основі типу пристрою, встановлюючи нижчі значення для WebView на iOS та Android для кращої адаптації до обмежень платформи. Кардинально було змінено критерії видалення сегментів з кешу: замість видалення за часом останнього доступу чи фіксованою кількістю, нова логіка пріоритезує збереження сегментів, актуальних для відтворення. Вона аналізує поточну позицію відтворення і в першу чергу видаляє сегменти, чий час закінчення (endTime) вже минув відносно цієї позиції. При цьому сегменти

сортуються за часом початку (startTime) для послідовного видалення найстаріших з тих, що вже не потрібні для відтворення. Для live-потоків додатково враховується часове вікно (highDemandTimeWindow) для збереження невеликого буфера сегментів перед актуальною позицією відтворення. Крім того, новий інтерфейс та його реалізація отримують інформацію про поточну позицію відтворення (onPlaybackUpdated) та останній запитуваний сегмент (onSegmentRequested), що дозволяє приймати більш обґрунтовані та ефективні рішення щодо керування кешем. Оновлена архітектура сховища, що включає інтерфейс SegmentStorage та нову реалізацію SegmentMemoryStorage, показана на рисунку 2.2.

Таким чином, рефакторинг алгоритму зберігання сегментів став важливим кроком в оптимізації бібліотеки P2P Media Loader, спрямованим на підвищення її ефективності та надійності в різноманітних середовищах виконання.

Додавання можливості вимкнення P2P-вивантаження

Стандартна модель роботи P2P-мереж передбачає, що кожен пір є не тільки споживачем, але й постачальником даних, тобто вивантажує (upload) завантажені сегменти іншим учасникам рою. Хоча це є основою ефективності P2P, вивантаження даних може бути небажаним або проблематичним у певних сценаріях, особливо для мобільних користувачів.

Основна мотивація для впровадження можливості вимкнення P2P-вивантаження пов'язана з особливостями мобільних мереж, таких як LTE або 5G. Часто такі мережі мають асиметричну пропускну здатність: достатньо високу швидкість завантаження, але значно нижчу швидкість вивантаження. Тому виникла потреба надати розробникам можливість гнучко керувати поведінкою P2P-клієнта щодо вивантаження даних.

Для вирішення цієї проблеми до конфігурації потоку було додано новий логічний параметр isP2PUploadDisabled. Цей параметр може бути встановлений як під час початкової ініціалізації ядра бібліотеки, так і динамічно змінений під час роботи за допомогою методу applyDynamicConfig. За замовчуванням

параметр `isP2PUUploadDisabled` має значення `false`, що означає, що вивантаження увімкнене.

Коли параметр `isP2PUUploadDisabled` встановлено в `true`, поведінка піра суттєво змінюється. По-перше, пір перестає повідомляти інших учасників рою про сегменти, які він має у своєму кеші. Це реалізовано шляхом припинення надсилання повідомлень як при підключенні нового піра, так і під час періодичного анонсування стану свого кешу. В результаті, інші піри просто не знатимуть, що цей користувач має певні сегменти, доступні для завантаження. По-друге, якщо інший пір все ж таки надсилає запит на сегмент (наприклад, якщо він отримав інформацію про наявність сегмента до того, як вивантаження було вимкнено на даному пристрої), пір з активованою опцією `isP2PUUploadDisabled` негайно відповідає повідомленням `SegmentAbsentCommand`, фактично відмовляючись передавати дані. При цьому важливо зазначити, що такий пір продовжує завантажувати сегменти від інших учасників P2P-мережі, якщо загалом P2P-функціонал для нього не вимкнено.

Додавання цієї опції робить бібліотеку P2P Media Loader більш адаптивною до реальних умов використання, особливо в мобільному сегменті.

2.2. Архітектура мобільної інтеграції

Реалізація мобільної інтеграції `p2pml-kotlin` базується на архітектурних рішеннях, обґрунтованих у попередньому розділі, і спрямована на забезпечення ефективної P2P-доставки медіаконтенту в Android-додатках при збереженні сумісності з вебкосистемою P2P Media Loader. Для наочного представлення принципу роботи розробленої системи та взаємодії її основних функціональних блоків, розглянемо спочатку загальну схему потоків даних (рис. 2.3).

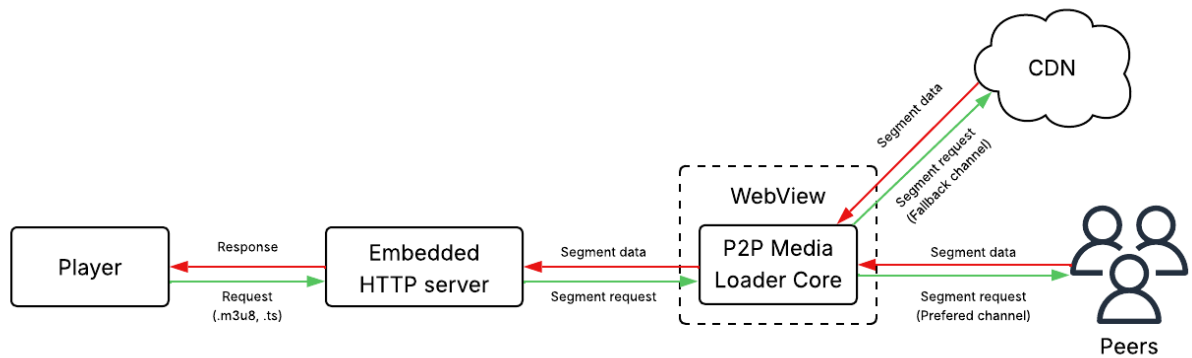


Рис. 2.3 – Загальна схема взаємодії компонентів та потоків даних у системі

Ця схема ілюструє, як запити від медіаплеєра проходять через локальний сервер, обробляються P2P-рушієм у WebView та вибір між P2P та HTTP джерелами для завантаження сегментів. Система складається з кількох ключових взаємопов'язаних елементів: нативного медіаплеєра, вбудованого HTTP-сервера, компонента WebView, що містить ядро P2P Media Loader, CDN та мережі пірів. Локальний HTTP-сервер виступає в ролі проксі, перенаправляючи запити плеєра до P2P-рушія, який приймає рішення про джерело завантаження сегмента – P2P-мережа або CDN.

Для глибшого розуміння внутрішньої будови самої Kotlin-бібліотеки p2pml-kotlin, її основних класів, їхніх обов'язків та взаємозв'язків, далі буде проаналізовано її статичну структуру. Для кращого сприйняття та деталізації її ключові логічні фрагменти будуть представлені послідовно, ілюструючи відповідні функціональні блоки бібліотеки.

Отже, для детального аналізу архітектури p2pml-kotlin розглянемо її основні компоненти, їхню структуру, взаємодію та застосовані патерни проєктування. Розпочнемо з ядра управління бібліотекою та її публічного інтерфейсу, ключові класи якого зображені на рисунку 2.4.

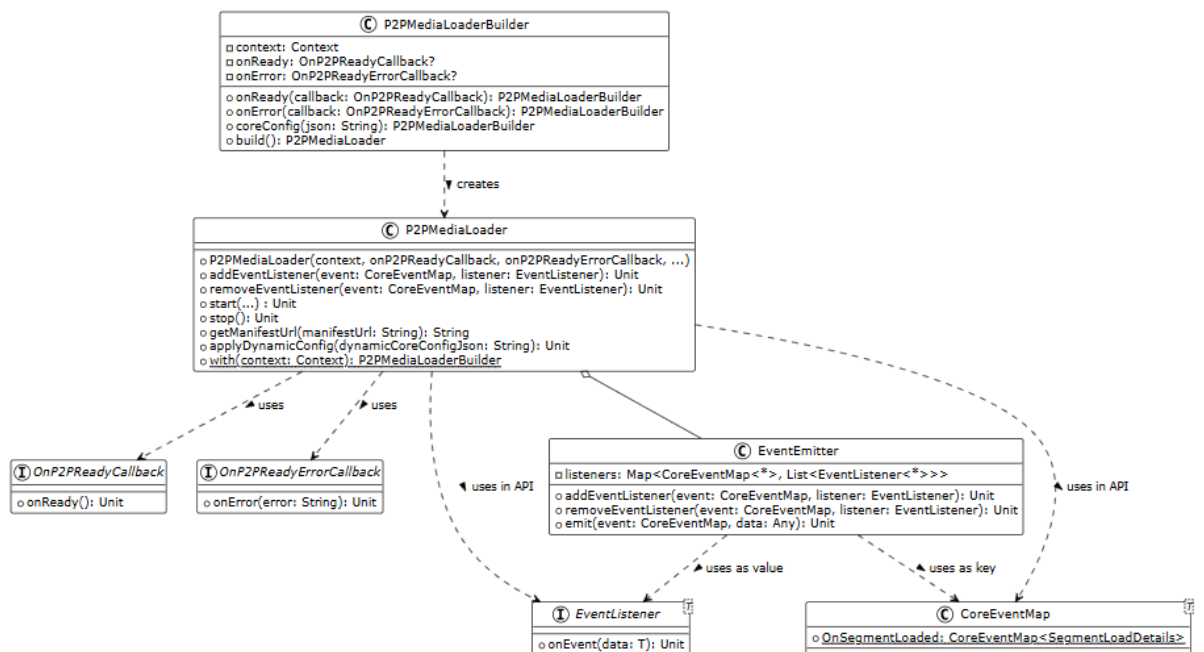


Рис. 2.4 – UML-діаграма класів ядра управління та публічного API бібліотеки

Ядро управління та публічний API

Центральним елементом бібліотеки p2pml-kotlin, що відповідає за координацію всіх внутрішніх процесів та надання публічного інтерфейсу для розробників Android-додатків, є клас P2PMediaLoader. Його створення та конфігурація реалізовані за допомогою патерну Будівельник (Builder Pattern), представленого класом P2PMediaLoaderBuilder [16].

Клас P2PMediaLoaderBuilder надає зручний ланцюжковий API для налаштування екземпляра P2PMediaLoader. За його допомогою розробник може послідовно встановити контекст Android-додатку, обов'язкові функції зворотного виклику onReady та onError. Також конфігуруються параметри ядра P2P Media Loader JS у форматі JSON-рядка, порт для локального HTTP-сервера, режим детального логування та є можливість додати кастомні JavaScript-інтерфейси для WebView. Завершується конфігурація викликом методу build, який повертає готовий до використання екземпляр P2PMediaLoader. Такий підхід значно спрощує створення складного об'єкта з великою кількістю опціональних параметрів та покращує читабельність коду ініціалізації.

Клас P2PMediaLoader виступає в ролі Фасаду (Facade Pattern) [16], інкапсулюючи складність внутрішньої взаємодії численних компонентів бібліотеки. Він агрегує та управляє ключовими модулями, такими як WebViewManager (відповідальний за WebView), ServerModule (HTTP-сервер), HlsParserService (обробка HLS-маніфестів), PlaybackProvider (отримання даних про стан плеєра), EventEmitter (система внутрішніх подій) та P2PStateManager (менеджер стану P2P). Основний публічний API класу P2PMediaLoader включає методи для запуску системи (start, який приймає екземпляр ExoPlayer або функцію для отримання PlaybackInfo, реалізуючи патерн Стратегія (Strategy) [16] через інтерфейс PlaybackProvider для гнучкої інтеграції з різними плеєрами), її зупинки для коректного звільнення ресурсів, а також метод getManifestUrl для модифікації URL HLS-маніфесту для перенаправлення запитів на локальний - сервер. Метод applyDynamicConfig дозволяє динамічно змінювати конфігурацію JavaScript-ядра P2P Media Loader під час роботи. Для обробки подій від P2P-ядра передбачені методи addEventListener та removeEventListener. Окрім цього, клас P2PMediaLoader відповідає за управління загальним станом роботи бібліотеки та життєвим циклом пов'язаних компонентів.

Система обробки подій реалізована за допомогою класів CoreEventManager, EventEmitter та інтерфейсу EventListener. CoreEventManager визначає типи можливих подій, дзеркалюючи структуру подій JavaScript-ядра. Події, що надходять з JavaScript-середовища через JavaScriptInterface, передаються до EventEmitter. EventEmitter, у свою чергу, реалізує патерн Спостерігач (Observer Pattern) [16], дозволяючи різним частинам Kotlin-коду, включаючи клієнтський код Android-додатку, підписуватися на ці події.

Цей набір класів формує основний інтерфейс та логіку управління життєвим циклом бібліотеки. Наступним важливим функціональним блоком є система управління компонентом WebView та забезпечення взаємодії між Kotlin та JavaScript. Ключові класи, що реалізують цей функціонал та їхні зв'язки показані на рисунку 2.5.

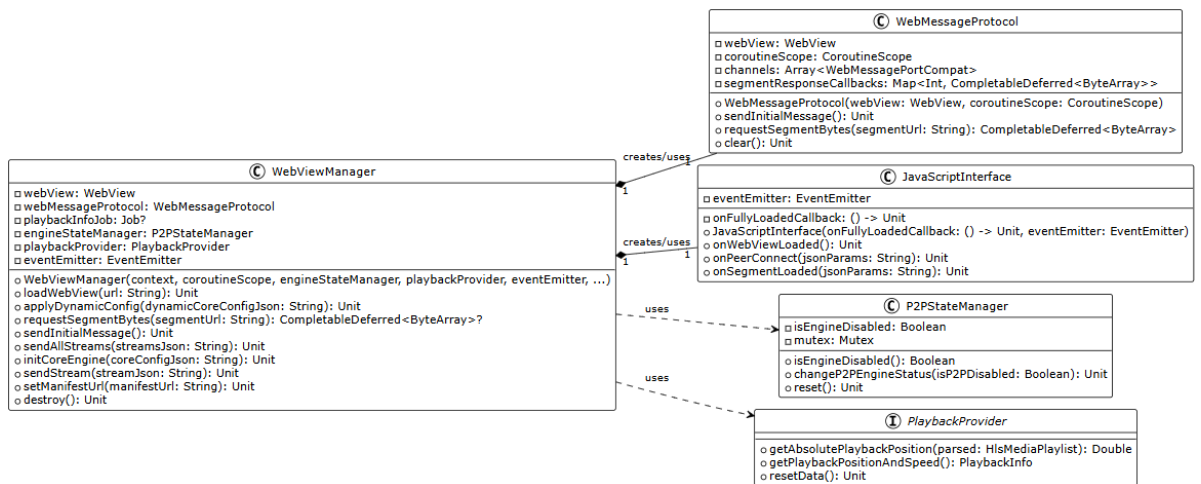


Рис. 2.5 – UML-діаграма класів модуля управління WebView, Kotlin-JS взаємодії та його залежностей

Управління WebView та взаємодія Kotlin-JavaScript

Ключову роль в інтеграції JavaScript-ядра P2P Media Loader з нативним Android-середовищем відіграє клас `WebViewManager`. Цей компонент відповідає за повний життєвий цикл прихованого `WebView`, який слугує середовищем для виконання JavaScript-коду P2P-бібліотеки. До основних завдань `WebViewManager` належить створення та конфігурація `WebView`, зокрема увімкнення підтримки JavaScript, DOM Storage та налаштування каналів спілкування між Kotlin та JavaScript.

Для забезпечення взаємодії між двома середовищами, `WebViewManager` реєструє об'єкт `JavaScriptInterface` під іменем "Android" у `WebView`, що дозволяє JavaScript-коду викликати публічні методи Kotlin, позначені відповідною анотацією. Крім того, `WebViewManager` ініціалізує та використовує `WebMessageProtocol` для організації двонаправленого асинхронного обміну даними з JavaScript, що є особливо важливим для запиту медіасегментів. `WebViewManager` також відповідає за надсилання команд до JavaScript-ядра:

- `initCoreEngine` – одержує параметр `configJson` для ініціалізації ядра;
- `applyDynamicConfig` – приймає параметр `configJson` для динамічної зміни налаштувань;

- `sendAllStreams` – обробляє параметр `streamsJson` для передачі інформації про всі потоки;
- `sendStream` – отримує параметр `streamJson` для передачі одного потоку;
- `setManifestUrl` – параметр `manifestUrl` передається в метод для встановлення URL маніфесту.

Для коректної роботи P2P-алгоритмів `WebViewManager` періодично надсилає інформацію про стан відтворення, отриману від `PlaybackProvider`, до JavaScript-ядра через метод `sendPlaybackInfo`. При зупинці бібліотеки `WebViewManager` забезпечує коректне знищення `WebView` та пов'язаних з ним ресурсів через виклик методу `destroy`.

Клас `JavaScriptInterface` слугує мостом для викликів з JavaScript до Kotlin. Його методи, позначені анотацією `@JavascriptInterface`, викликаються безпосередньо P2P-рушієм, що працює у `WebView`. Отримані дані, часто у форматі JSON-рядків, десеріалізуються та передаються до `EventEmitter` для подальшого сповіщення підписаних Kotlin-слухачів про події в P2P-мережі.

Ці компоненти забезпечують тісну інтеграцію та керовану взаємодію між нативним Kotlin-середовищем та JavaScript-ядром P2P Media Loader.

Далі розглянемо компоненти, що відповідають за реалізацію локального HTTP-сервера та обробку запитів від медіаплеєра. Їхня структура представлена на рисунку 2.6.

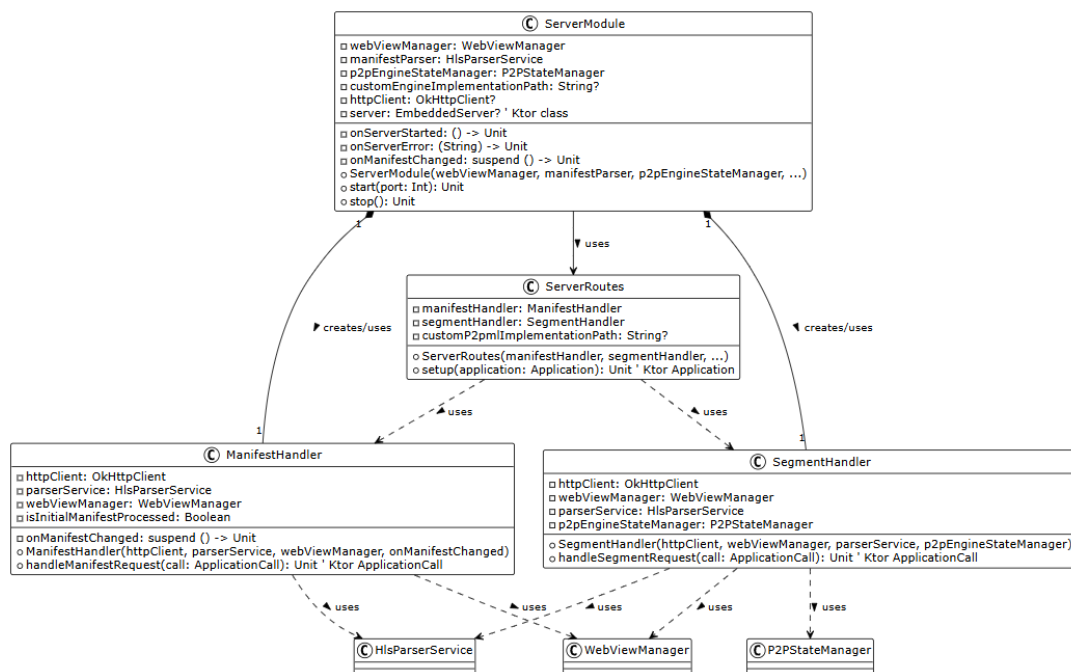


Рис. 2.6 – UML-діаграма класів локального HTTP-сервера та обробників запитів

Локальний HTTP-сервер та обробка запитів

Для перехоплення запитів від нативного медіаплеєра та їх подальшої обробки з використанням P2P-логіки, в бібліотеці `p2pml-kotlin` реалізовано вбудований HTTP-сервер на базі Ktor. Цей серверний модуль інкапсулює логіку налаштування, запуску, маршрутизації та обробки специфічних для P2P-стрімінгу HTTP-запитів.

Клас `ServerModule` відповідає за конфігурацію, запуск та зупинку HTTP-сервера Ktor, використовуючи для цього асинхронний рушій CIO (Coroutine-based I/O). Він також управляє екземпляром `OkHttpClient`, який використовується для резервних HTTP-запитів до CDN або сервера-джерела. Логіку маршрутизації для Ktor-сервера визначає клас `ServerRoutes`. Він налаштовує обробку запитів до кореневого шляху, делегуючи їх відповідним обробникам залежно від наявності параметрів запиту `?manifest=` або `?segment=`. Крім того, `ServerRoutes` забезпечує обслуговування статичних ресурсів, таких як HTML-сторінка та JavaScript-код P2P Media Loader для WebView, зазвичай зі шляху `/static` та налаштовує підтримку CORS.

Обробкою запитів на HLS-маніфести займається ManifestHandler. Коли нативний медіаплеєр надсилає запит на HLS-маніфест (URL якого вже модифіковано на локальний сервер), ManifestHandler спочатку завантажує оригінальний HLS-маніфест за вказаним URL, використовуючи OkHttpClient та копіюючи необхідні HTTP-заголовки з початкового запиту плеєра. Потім отриманий маніфест передається до HlsParserService для аналізу та модифікації URL-адрес сегментів і вкладених плейлистів таким чином, щоб вони також вказували на локальний сервер. Після цього ManifestHandler інформує JavaScript-ядро P2P Media Loader через WebViewManager про структуру потоку, отриману від HlsParserService, і, нарешті, повертає модифікований маніфест нативному плеєру з відповідним Content-Type (application/vnd.apple.mpegurl).

Запити на медіасегменти обробляються класом SegmentHandler. URL сегмента в такому запиті зазвичай є base64-кодованим оригінальним URL, можливо, з інформацією про байтовий діапазон. SegmentHandler спочатку декодує цей URL. Далі він перевіряє через HlsParserService, чи є цей сегмент частиною поточного плейлиста та чи активний P2P-рушій через P2PStateManager. Якщо умови для P2P-завантаження виконуються, SegmentHandler запитує дані сегмента у WebViewManager (який, у свою чергу, через WebMessageProtocol звертається до JavaScript-ядра), і отримані байти повертаються плеєру. У випадку помилки P2P-завантаження, якщо P2P-рушій вимкнено, або сегмент не вважається актуальним для P2P, SegmentHandler виконує пряме завантаження сегмента за оригінальним URL через OkHttpClient (fallback-механізм), після чого дані повертаються плеєру. Обробник також враховує наявність заголовка Range для коректної підтримки часткових запитів на сегменти.

Цей механізм проксування дозволяє зробити роботу P2P-шару прозорою для нативного медіаплеєра.

Наступний блок класів відповідає за аналіз HLS-маніфестів та управління метаданими потоків, як показано на рисунку 2.7.

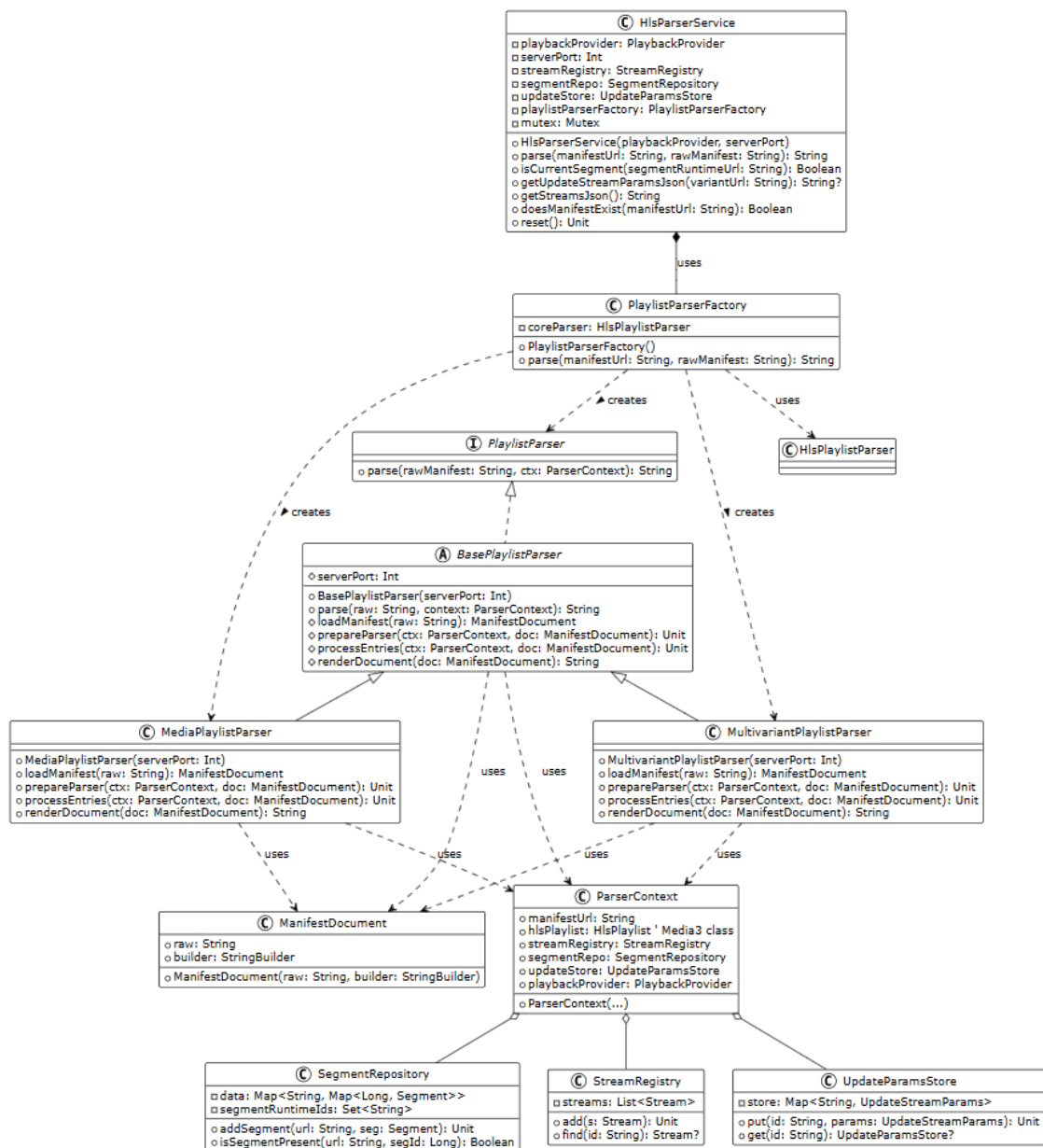


Рис. 2.7 – UML-діаграма класів модуля парсингу HLS-маніфестів та управління метаданими

Парсинг HLS-маніфестів та управління метаданими потоків

Для коректної роботи P2P-механізму та локального проксі-сервера необхідна детальна інформація про структуру HLS-потoku. Забезпечення цієї інформації та управління нею покладено на наступну групу класів.

Основним компонентом цього блоку є HlsParserService. Цей сервіс інкапсулює всю логіку, пов'язану з отриманням, парсингом та модифікацією HLS-маніфестів. Для створення конкретного об'єкта-парсера залежно від типу

плейлиста, HlsParserService використовує PlaylistParserFactory. Крім того, HlsParserService активно взаємодіє зі спеціалізованими сховищами метаданих – StreamRegistry, SegmentRepository та UpdateParamsStore – для накопичення та управління інформацією про потоки та сегменти. Ці дані згодом передаються в JavaScript-ядро P2P Media Loader для коректної ініціалізації та синхронізації P2P-сесії.

Клас PlaylistParserFactory реалізує патерн проєктування Фабрика (Factory Pattern). Його основне завдання – на основі типу HLS-плейлиста, розпізнаного за допомогою стандартного парсера HlsPlaylistParser з бібліотеки Media3, створити та повернути відповідний екземпляр кастомного парсера: MediaPlaylistParser для обробки медіа-плейлистів або MultivariantPlaylistParser для мастер-плейлистів.

Безпосередню логіку парсингу реалізують класи MediaPlaylistParser та MultivariantPlaylistParser, які успадковуються від абстрактного класу BasePlaylistParser. BasePlaylistParser визначає загальний шаблон обробки маніфесту, тоді як його нащадки специфічно обробляють відповідні типи HLS-плейлистів. Головне завдання цих парсерів – рекурсивно пройти по всіх URL-адресах сегментів та вкладених плейлистів у вихідному маніфесті та замінити їх на URL, що вказують на локальний сервер, передаючи при цьому оригінальний URL як параметр запиту. Паралельно вони збирають детальну інформацію про кожен сегмент (час початку/кінця, оригінальний URL, можливі байтові діапазони) для збереження у SegmentRepository.

Для забезпечення парсерів усією необхідною інформацією під час їхньої роботи використовується об'єкт класу ParserContext. Він передається в методи парсингу і містить URL маніфесту, вже розпарсений об'єкт HlsPlaylist, а також посилання на сховища StreamRegistry, SegmentRepository, UpdateParamsStore та на PlaybackProvider для доступу до інформації про стан відтворення.

Класи SegmentRepository, StreamRegistry та UpdateParamsStore виконують роль внутрішніх сховищ метаданих. StreamRegistry відповідає за зберігання інформації про відомі потоки. SegmentRepository зберігає інформацію про окремі медіасегменти для кожного потоку. UpdateParamsStore, у свою чергу, акумулює

Для взаємодії з медіаплеєром та отримання інформації про стан відтворення використовується абстракція, показана на рисунку 2.8.



Для забезпечення гнучкості бібліотеки `p2pml-kotlin` та її інтеграції з різними медіаплеєрами, а не лише з найпопулярнішим `ExoPlayer`, було впроваджено спеціальний рівень абстракції. Основою цієї абстракції є інтерфейс `PlaybackProvider`. Він визначає контракт для отримання ключових даних про поточний стан відтворення, таких як абсолютна позиція відтворення та поточна позиція і швидкість відтворення. Такий підхід є реалізацією патерну проєктування Стратегія (Strategy Pattern) [16], що дозволяє ядру `P2PMediaLoader` отримувати необхідну інформацію про відтворення, не залежачи від конкретної реалізації використовуваного медіаплеєра.

Бібліотека надає дві основні конкретні реалізації цього інтерфейсу. Клас `ExoPlayerPlaybackProvider` призначений для безпосередньої роботи з `ExoPlayer`. Він отримує необхідні дані про позицію та швидкість безпосередньо з екземпляра `ExoPlayer`. Для live-потоків цей провайдер також відповідає за обчислення абсолютного часу відтворення, спираючись на метадані з `HlsMediaPlaylist`. Як альтернатива, клас `ExternalPlaybackProvider` пропонує більш універсальний механізм інтеграції. Він дозволяє розробнику Android-додатку

надати власну функцію, яка буде повертати об'єкт `PlaybackInfo` з актуальними даними. Це робить бібліотеку `r2pml-kotlin` сумісною з будь-яким медіаплеєром, для якого розробник може реалізувати таку функцію-постачальника даних.

Впровадження такої абстракції є важливим кроком для забезпечення універсальності та розширюваності розробленої бібліотеки.

2.3. Реалізація ключових механізмів

Асинхронна комунікація Kotlin-JavaScript

Для забезпечення ефективного та надійного двонаправленого зв'язку між нативним Kotlin-кодом та JavaScript-ядром `P2P Media Loader`, що виконується у `WebView`, було розроблено клас `WebMessageProtocol`. Цей механізм є критично важливим, зокрема, для асинхронного запиту медіасегментів з Kotlin-частини та отримання їхніх даних або повідомлень про помилки від JavaScript-частини.

Ініціалізація каналу зв'язку відбувається при створенні екземпляра `WebMessageProtocol`, якому передається посилання на `WebView` та `CoroutineScope` для управління асинхронними операціями. Одразу після цього створюється захищений двосторонній канал зв'язку за допомогою методу `createWebMessageChannel` з класу `WebViewCompat` [17], який приймає екземпляр `WebView`. Цей метод повертає масив з двох об'єктів `WebMessagePortCompat`. Один з цих портів залишається на стороні Kotlin, і на ньому встановлюється `WebMessageCallbackCompat` для обробки вхідних повідомлень від JavaScript. Інший порт передається до JavaScript під час первинного встановлення зв'язку через метод `sendInitialMessage` класу `WebMessageProtocol` (рис. 2.9).

```

@SuppressLint("RequiresFeature")
private fun initializeWebMessageCallback() {
    channels[0].setWebMessageCallback(
        object : WebMessagePortCompat.WebMessageCallbackCompat() {
            override fun onMessage(
                port: WebMessagePortCompat,
                message: WebMessageCompat?,
            ) {
                try {
                    when (message?.type) {
                        WebMessageCompat.TYPE_ARRAY_BUFFER -> {
                            handleSegmentIdBytes(message.arrayBuffer)
                        }

                        WebMessageCompat.TYPE_STRING -> {
                            handleMessage(message.data!!)
                        }
                    }
                } catch (e: Exception) {
                    Logger.e(TAG, "Error while handling message: ${e.message}", e)
                }
            }
        }
    )
}

@SuppressLint("RequiresFeature")
suspend fun sendInitialMessage() {
    if (wasInitialMessageSent) return
    withContext(Dispatchers.Main) {
        val initialMessage = WebMessageCompat(
            data = "", arrayOf(channels[1]))
        WebViewCompat.postWebMessage(
            webView,
            initialMessage,
            Uri.parse("http://*"),
        )
        wasInitialMessageSent = true
    }
}

```

Рис. 2.9 – Фрагмент коду з класу WebMessageProtocol: ініціалізація WebMessageChannel та передача порту до JavaScript

Процес запиту сегмента з Kotlin до JavaScript починається, коли Kotlin-частині, наприклад, SegmentHandler через WebViewManager, потрібно отримати сегмент. У цьому випадку викликається метод requestSegmentBytes, який примймає URL сегменту для ідентифікації запрошеного сегменту, класу WebMessageProtocol. В середині цього методу генерується унікальний цілочисельний ідентифікатор запиту. Далі створюється об'єкт CompletableDeferred, призначений для асинхронного очікування відповіді. Цей CompletableDeferred разом з requestId зберігається у внутрішній мапі segmentResponseCallbacks, доступ до якої для потокобезпечності синхронізується за допомогою Mutex. Після цього формується об'єкт запиту SegmentRequest, що містить requestId та segmentUrl, який серіалізується в JSON-рядок й надсилається до JavaScript ядра через метод sendSegmentRequest, що приймає серіалізовані дані для відправки (рис. 2.10).

```

suspend fun requestSegmentBytes(segmentUrl: String): CompletableDeferred<ByteArray> {
    val deferred = CompletableDeferred<ByteArray>()
    val requestId = generateRequestId()
    val segmentRequest = SegmentRequest(requestId, segmentUrl)
    val jsonRequest = Json.encodeToString(segmentRequest)

    addSegmentResponseCallback(requestId, deferred)
    sendSegmentRequest(jsonRequest)

    return deferred
}

private suspend fun sendSegmentRequest(segmentUrl: String) {
    withContext(Dispatchers.Main) {
        webView.evaluateJavascript(
            script: "javascript:window.p2p.processSegmentRequest('$segmentUrl');",
            resultCallback: null,
        )
    }
}

```

Рис. 2.10 – Фрагмент коду з класу WebMessageProtocol: реалізація запиту на отримання сегмента з Kotlin до JavaScript

Обробка відповіді від JavaScript-ядра P2P Media Loader на стороні Kotlin відбувається через раніше встановлений WebMessageCallbackCompat на Kotlin-порті. Після обробки запиту JavaScript-ядро надсилає відповідь назад у Kotlin через переданий йому порт WebMessagePortCompat. Цей процес зазвичай відбувається у два етапи. Спочатку, перед надсиланням самих байтів сегмента, JavaScript надсилає рядкове повідомлення, що містить requestId, на який готується відповідь. WebMessageProtocol у методі handleMessage зберігає цей incomingRequestId. Слідом за цим JavaScript надсилає дані сегмента у вигляді ArrayBuffer. На Kotlin-стороні це повідомлення приймається як ByteArray у методі onMessage. Метод handleSegmentIdBytes використовує раніше отриманий incomingRequestId, знаходить відповідний CompletableDeferred з мапи segmentResponseCallbacks, наповнює його отриманими байтами та видаляє відповідний запис з мапи. У випадку, якщо JavaScript-ядро не може надати сегмент, воно надсилає рядкове повідомлення про помилку, яке також містить requestId та тип помилки. Метод handleErrorMessage в WebMessageProtocol аналізує це повідомлення, знаходить відповідний CompletableDeferred та

завершує його винятком, таким як `SegmentNotFoundException`, `SegmentAbortedException` або загальний `Exception` (рис. 2.11).

```
private fun handleSegmentIdBytes(arrayBuffer: ByteArray) {
    val requestId =
        incomingRequestId
        ?: throw IllegalStateException("Received segment bytes without a segment ID")

    coroutineScope.launch {
        val deferred =
            getSegmentResponseCallback(requestId)
            ?: throw IllegalStateException("No deferred found for request ID: $requestId")

        deferred.complete(arrayBuffer)
        removeSegmentResponseCallback(requestId)

        incomingRequestId = null
    }
}

private fun handleMessage(message: String) {
    if (message.contains(other: "Error")) {
        handleErrorMessage(message)
    } else {
        handleSegmentIdMessage(message)
    }
}
}
```

Рис. 2.11 – Фрагмент коду з класу `WebMessageProtocol`: обробка вхідних повідомлень від JavaScript

Варто згадати про обмеження та особливості даного механізму. Передача `ArrayBuffer` з JavaScript до Kotlin через `WebMessagePortCompat` зазвичай призводить до копіювання даних, а не до передачі володіння (transferable object). Це пов'язано з тим, що JavaScript-ядру P2P Media Loader необхідно зберігати ці дані у власному кеші для подальшої роздачі іншим пірам. Таке копіювання даних створює додаткову витрату ресурсів, що є усвідомленим компромісом обраної архітектури (рис. 2.12).

```

async processSegmentRequest(segmentRequestJSON) {
    const segmentRequest = JSON.parse(segmentRequestJSON);
    const { segmentUrl, requestId } = segmentRequest;

    const onSuccess = (response) => {
        try {
            const copyData = new ArrayBuffer(response.data.byteLength);
            new Uint8Array(copyData).set(new Uint8Array(response.data));

            this.port.postMessage(`${requestId}`);
            this.port.postMessage(copyData, [copyData]);
        } catch (e) {
            this.port.postMessage(`Error:${requestId}:failed|${segmentUrl}`);
        }
    };

    const onError = (error) => {
        if (error.type === "aborted") {
            this.port.postMessage(
                `Error:${requestId}:${error.type}|${segmentUrl}`
            );
            return;
        }
        this.port.postMessage(
            `Error:${requestId}:${error.type}|${segmentUrl}`
        );
    };

    try {
        await this.core.loadSegment(segmentUrl, {
            onSuccess,
            onError,
        });
    } catch (e) {
        this.port.postMessage(`Error:${requestId}:not_found|${segmentUrl}`);
    }
}

```

Рис. 2.12 – Фрагмент JavaScript-коду: обробка запиту processSegmentRequest та надсилання відповіді до Kotlin

Незважаючи на це, описаний механізм через WebMessageProtocol забезпечує надійний та ефективний асинхронний зв'язок для ключової операції – запиту та отримання медіасегментів між нативною частиною та P2P-рушієм, що працює у WebView.

Логіка роботи вбудованого Ktor-сервера та проксування запитів

Важливою складовою архітектури p2pml-kotlin є вбудований HTTP-сервер, реалізований за допомогою фреймворку Ktor. Він виконує дві основні ролі: обслуговування статичних файлів для ініціалізації JavaScript-ядра P2P Media Loader у WebView та, що найголовніше, перехоплення запитів на медіаконтент від нативного плеєра для їх подальшої обробки P2P-механізмом.

Ініціалізація та маршрутизація Ktor-сервера відбувається в класах `ServerModule` та `ServerRoutes`. Зокрема, `ServerModule` відповідає за конфігурацію та запуск Ktor-сервера, використовуючи асинхронний рушій `Coroutine-based I/O`. При старті сервера, ініціалізується маршрутизація, визначена в класі `ServerRoutes` (рис. 2.13). `ServerRoutes`, у свою чергу, налаштовує обробку запитів до кореневого шляху сервера. Залежно від наявності параметрів запиту `?manifest=` або `?segment=`, запити делегуються відповідним обробникам – `ManifestHandler` або `SegmentHandler`. Також налаштовується маршрут для роздачі статичних файлів, таких як HTML-сторінка та JavaScript-код P2P Media Loader для `WebView`, зазвичай зі шляху `/static`, як визначено в `Constants`.

```
fun setup(application: Application) {
    application.routing {
        get(path: "/") {
            when {
                call.parameters["manifest"] != null ->
                    manifestHandler.handleManifestRequest(call)
                call.parameters["segment"] != null -> segmentHandler.handleSegmentRequest(call)

                else ->
                    call.respondText(
                        text: "Missing required parameter",
                        status = io.ktor.http.HttpStatusCode.BadRequest,
                    )
            }
        }
    }

    staticResources(remotePath: "/static", CORE_FILE_PATH) {
        default(path: "index.html")
    }

    if (customP2pmlImplementationPath != null) {
        staticResources(CUSTOM_FILE_PATH, customP2pmlImplementationPath) {
            default(path: "index.html")
        }
    }
}
```

Рис. 2.13 – Фрагмент коду з класу `ServerRoutes`: налаштування основних маршрутів Ktor-сервера

Обробка запитів на маніфест покладається на `ManifestHandler` (рис. 2.14). Коли нативний медіаплеєр запитує HLS-маніфест за URL-адресою, попередньо модифікованою методом `getManifestUrl` з класу `P2PMediaLoader`, цей запит надходить до `ManifestHandler` на локальному Ktor-сервері. Спочатку `ManifestHandler` отримує закодований URL оригінального маніфесту з параметра запиту `?manifest=`. Далі, за допомогою `OkHttpClient`, він завантажує вміст

оригінального HLS-маніфесту з мережі, при цьому копіюючи необхідні HTTP-заголовки з початкового запиту плеєра для забезпечення коректної відповіді від CDN або сервера-джерела. Отриманий текстовий вміст маніфесту та його URL передаються до сервісу HlsParserService. Цей сервіс, залучаючи PlaylistParserFactory та відповідні парсери, аналізує структуру маніфесту. У процесі аналізу всі URL-адреси медіасегментів та вкладених плейлистів у маніфесті модифікуються таким чином, щоб вони вказували на локальний HTTP-сервер, наприклад: `http://localhost:PORT/?segment=ENCODED_ORIGINAL_URL` або `http://localhost:PORT/?manifest=ENCODED_SUB_MANIFEST_URL`. Після парсингу HlsParserService готує структуровану інформацію про потоки у форматі JSON та інформацію про початкові сегменти. ManifestHandler передає ці дані через WebViewManager до JavaScript-ядра P2P Media Loader, що працює у WebView, що дозволяє P2P-рушію ініціалізуватися для конкретного медіапотоку. Нарешті, ManifestHandler повертає нативному плеєру вже модифікований текст маніфесту з відповідним Content-Type (`application/vnd.apple.mpegurl`). ManifestHandler також відстежує, чи є запит початковим (`isInitialManifestProcessed`), для коректної ініціалізації P2P-ядра, та реагує на зміни маніфесту через відповідний колбек (`onManifestChanged`).

```

suspend fun handleManifestRequest(call: ApplicationCall) {
    val manifestParam =
        call.request.queryParameters["manifest"]
        ?: return call.respondText(
            text = "Missing 'manifest' parameter",
            status = HttpStatusCode.BadRequest,
        )
    val decodedManifestUrl = manifestParam.decodeURLQueryComponent()

    try {
        val fetchResult = fetchManifest(call, decodedManifestUrl)
        val doesManifestExist = parserService.doesManifestExist(decodedManifestUrl)

        if (!doesManifestExist) {
            reset()
            onManifestChanged()
        }

        val modifiedManifest =
            parserService.parse(
                fetchResult.responseUrl,
                fetchResult.manifestContent,
            )
        val needsInitialSetup = checkAndSetInitialProcessing()

        handleUpdate(decodedManifestUrl, needsInitialSetup)
        call.respondText(modifiedManifest, ContentType.parse(MPEGURL_CONTENT_TYPE))
    } catch (e: Exception) {
        val message = "Failed to process manifest request: ${e.message}"

        Log.e(TAG, message)
        call.respondText(message, status = HttpStatusCode.InternalServerError)
    }
}

```

Рис. 2.14 – Фрагмент коду з класу ManifestHandler: ключові етапи обробки запиту на HLS-маніфест

Обробкою запитів на сегменти займається клас SegmentHandler. Коли плеєр, обробивши модифікований маніфест, запитує конкретний медіасегмент, цей запит також надходить на локальний Ktor-сервер. SegmentHandler спочатку отримує закодований оригінальний URL сегмента, а також можливу інформацію про байтовий діапазон, з параметра ?segment= та декодує URL. Далі виконується перевірка актуальності запитуваного сегмента (чи є він частиною поточного, відомого плейлиста, через метод isCurrentSegment у класі HlsParserService) та стану P2P-рушія. Якщо P2P-механізм активний і сегмент актуальний, SegmentHandler ініціює спробу P2P-завантаження, викликаючи метод requestSegmentBytes з класу webViewManager й передаючи в нього декодовану URL сегменту. Цей виклик, як було описано раніше, через WebMessageProtocol ініціює запит до JavaScript-ядра P2P Media Loader. SegmentHandler асинхронно очікує на результат цього запиту. У разі успішного отримання даних сегмента

через P2P, вони повертаються плеєру, враховуючи HTTP статус 206 Partial Content, якщо запитувався байтовий діапазон. Однак, якщо P2P-завантаження не вдалося або якщо P2P-рушій вимкнено, чи сегмент не вважається актуальним для P2P, SegmentHandler виконує пряме завантаження сегмента через HTTP. Для цього він використовує OkHttpClient, запитуючи сегмент за його оригінальним URL та копіюючи необхідні заголовки з запиту плеєра. Отримані таким чином дані повертаються плеєру. Детальніше логіка обробки запиту на медіасегмент представлена у Додатку А (рис. А.1).

Така архітектура локального сервера та обробників запитів дозволяє ефективно інтегрувати P2P-логіку, що виконується у WebView, у стандартний потік завантаження даних нативним медіаплеєром, роблячи весь P2P-механізм прозорим для нього.

Управління життєвим циклом WebView та P2P-рушія з Kotlin

Ефективна робота бібліотеки p2pml-kotlin вимагає чіткого управління життєвим циклом її ключових компонентів, зокрема прихованого WebView, в якому виконується JavaScript-ядро P2P Media Loader, та самого P2P-рушія. Ця координація здійснюється переважно класами P2PMediaLoader та WebViewManager.

Процес ініціалізації та запуску системи активується, коли розробник викликає один з методів start класу P2PMediaLoader, передаючи або екземпляр EchoPlayer, або функцію, що повертитиме PlaybackInfo. Спочатку перевіряється поточний стан бібліотек, якщо вона вже запущена, операція не виконується. Далі створюється новий Job та CoroutineScope для управління асинхронними операціями в рамках поточної сесії. Послідовно ініціалізуються відповідний PlaybackProvider та HlsParserService, відповідальний за обробку HLS-маніфестів. Фрагмент коду з класу P2PMediaLoader, що ілюструє ініціалізацію компонентів, наведено у Додатку Б (рис. Б.1).

Наступним кроком є створення екземпляра WebViewManager. У його конструкторі відбувається створення та налаштування Android WebView:

					ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
						52

вмикається JavaScript, DOM storage, встановлюється WebViewClientCompat, а сам WebView робиться невидимим для користувача через конфігурацію параметру visible (рис. 2.15). До WebView також додається екземпляр JavaScriptInterface під іменем "Android", що дозволяє JavaScript-коду викликати Kotlin-методи, та ініціалізується WebMessageProtocol для двостороннього зв'язку з JavaScript. Важливо, що у конструктор WebViewManager передається колбек onPageLoadFinished, який буде викликаний з P2PMediaLoader після повного завантаження сторінки у WebView. Після налаштування WebViewManager створюється та запускається екземпляр ServerModule на вказаному порту, при цьому ServerModule отримує посилання на WebViewManager та HlsParserService.

```
@SuppressWarnings("SetJavaScriptEnabled")
private val webView =
    WebView(context).apply {
        settings.javaScriptEnabled = true
        settings.domStorageEnabled = true
        webViewClient = WebViewClientCompat()
        visibility = View.GONE
        addJavaScriptInterface(
            JavaScriptInterface(onPageLoadFinished, eventEmitter),
            name: "Android",
        )
    }
private val webMessageProtocol = WebMessageProtocol(webView, coroutineScope)
```

Рис. 2.15 – Фрагмент коду з класу WebViewManager: ініціалізація
WebView

Завантаження контенту у WebView відбувається після успішного старту Ktor-сервера, коли викликається його колбек onServerStarted. У відповідь P2PMediaLoader ініціює завантаження URL в WebView через клас WebViewManager викликаючи метод loadWebView, в який передається URL. Цей URL вказує на index.html, який містить JavaScript-міст та посилання на ядро P2P Media Loader JS (рис. 2.16). Коли сторінка index.html та її скрипти повністю завантажені у WebView, JavaScript викликає метод onWebViewLoaded, який був зареєстрований у JavaScriptInterface. Цей виклик активує колбек onPageLoadFinished в WebViewManager, який передає управління до P2PMediaLoader. Далі P2PMediaLoader запускає корутину, яка викликає метод initCoreEngine на екземплярі класу WebViewManager.

					ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
						53

Цей метод надсилає команду JavaScript-рушію для його ініціалізації з переданою JSON-конфігурацією. Після успішної ініціалізації JavaScript-ядра, P2PMediaLoader викликає onP2PReadyCallback, сповіщаючи клієнтський код про те, що P2P-система готова до роботи. На цьому етапі внутрішній стан бібліотеки встановлюється в STARTED.

```
private fun onWebViewLoaded() {
    scope!!.launch {
        try {
            Logger.d(TAG, message: "WebView loaded, initializing P2P engine")
            webViewManager!!.initCoreEngine(coreConfigJson)
            Logger.d(TAG, message: "P2P engine initialized, notifying onP2PReadyCallback")
            onP2PReadyCallback.onReady()
        } catch (e: Exception) {
            onP2PReadyErrorCallback.onError( error: e.message ?: "Unknown error")
        }
    }
}

private fun onServerStarted() {
    val urlPath =
        if (customEngineImplementationPath != null) {
            Utils.getUrl(serverPort, CUSTOM_FILE_URL)
        } else {
            Utils.getUrl(serverPort, CORE_FILE_URL)
        }

    try {
        Logger.d(TAG, message: "Loading WebView with URL: $urlPath")
        webViewManager!!.loadWebView(urlPath)
    } catch (e: Exception) {
        onP2PReadyErrorCallback.onError( error: e.message ?: "Unknown error")
    }
}
```

Рис. 2.16 – Фрагмент коду з класу P2PMediaLoader: послідовність ініціалізації JavaScript P2P-рушія

Під час роботи системи, після успішного запуску та при надходженні запитів на сегменти, WebViewManager активує періодичне завдання. Це завдання відповідає за надсилання актуальної інформації про стан відтворення, отриманої з PlaybackProvider, до JavaScript-ядра. Така інформація необхідна P2P-рушію для прийняття обґрунтованих рішень щодо пріоритезації завантаження сегментів. Окрім цього, при виклику методу applyDynamicConfig нова JSON-конфігурація передається через WebViewManager до JavaScript-ядра, що дозволяє динамічно змінювати параметри P2P-логіки. WebViewManager також оновлює

P2PStateManager щодо стану isP2PDisabled і може зупинити playbackInfoJob, якщо P2P-функціонал повністю вимикається (рис. 2.17).

```
private fun startPlaybackInfoUpdate() {
    if (playbackInfoJob != null) return

    playbackInfoJob =
        coroutineScope.launch {
            while (isActive) {
                try {
                    if (engineStateManager.isEngineDisabled()) {
                        playbackInfoJob?.cancel()
                        playbackInfoJob = null
                        break
                    }

                    val currentPlaybackInfo =
                        playbackProvider.getPlaybackPositionAndSpeed()
                    val playbackInfoJson = Json.encodeToString(currentPlaybackInfo)

                    sendPlaybackInfo(playbackInfoJson)

                    delay( timeMillis: 400)
                } catch (e: Exception) {
                    Logger.d(TAG, message: "Playback info update failed: ${e.message}")
                }
            }
        }
}
```

Рис. 2.17 – Фрагмент коду з класу WebMessageProtocol: періодичне надсилання playbackInfoUpdate

Коректне завершення роботи системи та звільнення ресурсів, особливо для WebView, який може споживати значну кількість пам'яті, забезпечується методом stop з класу P2PMediaLoader. Перед виконанням операцій зупинки метод перевіряє, чи система перебуває у стані STARTED. Далі послідовно викликаються методи destroy або stop для ключових компонентів, які були створені при запуску (рис. 2.18).

```

/**
 * Stops P2P streaming and releases all resources.
 * Call [start] to reinitialize after stopping.
 *
 * @throws IllegalStateException if P2PMediaLoader is not started
 */
fun stop() {
    ensureStarted()

    Logger.d(TAG, message: "Stopping P2PMediaLoader...")

    runBlocking {
        webViewManager?.destroy()
        webViewManager = null

        serverModule?.stop()
        serverModule = null

        parserService?.reset()
        parserService = null

        playbackProvider?.resetData()
        playbackProvider = null

        engineStateManager.reset()
        eventEmitter.removeAllListeners()

        appState = AppState.STOPPED

        job?.cancel()
        job = null
        scope = null
    }
    Logger.d(TAG, message: "P2PMediaLoader stopped and resources freed.")
}

```

Рис. 2.18 – Фрагмент коду з класу P2PMediaLoader: звільнення ресурсів при зупинці P2P-системи

Таке детальне управління життєвим циклом забезпечує правильну ініціалізацію всіх залежностей, їхню скоординовану роботу під час активної сесії P2P-стрімінгу та належне звільнення системних ресурсів при її завершенні.

Висновки до розділу 2

У даному розділі було представлено комплекс робіт, спрямованих на вдосконалення існуючої технології P2P-доставки медіа та її адаптацію для використання на мобільних платформах. Робота велася у двох ключових напрямках: оптимізація базової JavaScript-бібліотеки P2P Media Loader та розробка нативної Kotlin-бібліотеки p2pml-kotlin для її інтеграції в Android-додатки.

В рамках оптимізації базової бібліотеки P2P Media Loader було здійснено значущі зміни, що підвищили її ефективність та гнучкість, особливо для роботи в ресурсообмежених середовищах, таких як мобільні WebView.

Другим ключовим напрямком стала розробка мобільної інтеграційної бібліотеки `p2pml-kotlin` для платформи Android. Було детально описано архітектуру цієї бібліотеки, яка базується на використанні `Headless WebView` для запуску JavaScript-ядра `P2P Media Loader`, що гарантує повну сумісність з вебпірами, та вбудованого HTTP-сервера на базі `Ktor` для прозорого проксування запитів від нативного медіаплеєра.

Розроблена бібліотека `p2pml-kotlin` надає Android-розробникам зручне API для легкої інтеграції P2P-функціоналу в їхні додатки, роблячи процес доставки медіа більш ефективним та економічно вигідним.

Таким чином, у рамках другого розділу було представлено як теоретичні обґрунтування, так і практичну реалізацію запропонованих рішень, що спрямовані на підвищення ефективності, гнучкості та доступності технології P2P-доставки медіа, особливо в контексті мобільних додатків. Експериментальна оцінка розроблених оптимізацій та мобільної інтеграції буде представлена у наступному розділі.

РОЗДІЛ 3. ТЕСТУВАННЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ РОЗРОБЛЕНОЇ СИСТЕМИ

3.1. Методика та середовище тестування

Для об'єктивної оцінки розробленої системи було визначено методику проведення тестування, сформовано тестове середовище, обрано ключові метрики ефективності та інструментарій для збору й аналізу даних.

Конфігурація тестового стенду

Тестовий стенд, на якому проводилися тести, складався з серверної та клієнтської частин, що функціонували у визначеному мережевому оточенні. Серверна частина включала джерело медіаконтенту та трекер-сервер. Як джерело контенту використовувався публічно доступний тестовий HLS потік (<https://test-streams.mux.dev/x36xhzz/x36xhzz.m3u8>), що є частиною офіційної демонстрації бібліотеки hls.js. Для експериментів було обрано максимальну доступну якість цього потоку (1920x1080, бітрейт близько 6.2 Mbps, кодеки mp4a.40.2, avc1.640028), при цьому характеристики самого сервера невідомі. Координація P2P-клієнтів здійснювалася за допомогою WebTorrent трекера [wss://tracker.novage.com.ua](https://tracker.novage.com.ua), розгорнутого на сервері з центральним процесором Ampere A1, 24 ГБ оперативної пам'яті та мережевим інтерфейсом 1 Гбіт/с.

Клієнтська частина тестового стенду була представлена двома типами пристроїв, що дозволило тестувати та досліджувати дваземодію пірів. Першим пристроєм був настільний клієнт з процесором Intel Core i5-10400H 2.6 ГГц, 16 ГБ RAM, під управлінням ОС Windows 10 та веббраузером Google Chrome v137.0.7151.68. Другим пристроєм виступав мобільний клієнт – смартфон Samsung Galaxy A50 SM-A505FN з чипсетом Exynos 9610, 6 ГБ RAM, під управлінням ОС Android 11.

Щодо мережевого середовища, то під час основної частини експериментів обидва клієнтські пристрої були підключені до мережі Інтернет через Wi-Fi з'єднання з пропускнуою здатністю до 100 Мбіт/с. Для специфічних сценаріїв мобільний клієнт підключався через мережу LTE.

Тестові сценарії використання

Було визначено та проведено декілька ключових сценаріїв експериментів для всебічної оцінки системи.

Перший сценарій полягав у базовому тестуванні прямого HTTP/CDN відтворення на мобільному клієнті Samsung A50 без залучення бібліотеки p2pml-kotlin, що дозволило отримати контрольні показники продуктивності та ресурсоспоживання.

Другий сценарій був спрямований на оцінку додаткового навантаження, створюваного самою інтеграцією p2pml-kotlin, для якого відтворення також відбувалося через HTTP/CDN без активних P2P-пірів, але вже з активною бібліотекою.

Третій сценарій передбачав базове P2P-тестування взаємодії мобільного клієнта Samsung A50 та одного вебклієнту при одночасному старті відтворення через Wi-Fi, з метою оцінки початкової ефективності P2P-обміну та сумісності.

Четвертий сценарій досліджував ефективність P2P offload для мобільного клієнта при його підключенні до вже існуючого активного рою з 3-5 пірів, які мали попередньо закешований контент.

П'ятий сценарій спрямований на дослідження стабільності та надійності системи за умови плинності пірів. Метою було оцінити, як мобільний клієнт p2pml-kotlin реагує на раптове відключення та повторне підключення одного з пірів, зокрема, наскільки швидко та безшовно відбувається перемикання на HTTP і чи призводить це до переривань відтворення.

Кожен експериментальний запуск тривав 10 хв та повторювався 3 рази для усереднення результатів та підвищення їхньої статистичної значущості.

Інструментарій для збору та аналізу даних й ключові метрики оцінки ефективності

Проведення тестування, збір та подальший аналіз даних здійснювалися з використанням комплексу програмних засобів. Для моніторингу використання ресурсів на Android-пристрої застосовувався Android Studio Profiler. Аналіз

обсягів трафіку P2P-механізму забезпечувався вбудованими механізмами логування в бібліотеці p2pml-kotlin.

Для кількісної оцінки розробленої системи було обрано наступні метрики. Ефективність P2P розраховувалася як відсоткове співвідношення обсягу даних, отриманих через P2P, до загального обсягу відтворених даних. Використання ресурсів мобільного пристрою включало вимірювання середнього та пікового навантаження на CPU у відсотках та середнього і пікового споживання оперативної пам'яті додатком у МБ. Якість відтворення оцінювалася за часом початкового завантаження відео та за кількістю й загальною тривалістю подій переривання на буферизацію.

3.2. Результати тестування та їх аналіз

У даному підрозділі представлено результати тестування, проведених згідно з методикою, описаною в підрозділі 3.1. Аналіз результатів дозволяє кількісно оцінити ефективність розробленої системи p2pml-kotlin та вплив P2P-інтеграції на використання ресурсів пристрою.

Результати базових тестів та оцінка витрат ресурсів інтеграції

Першочерговим завданням було визначення базових показників продуктивності пристрою при відтворенні контенту напряму з CDN та оцінка додаткового навантаження, яке створює бібліотека p2pml-kotlin при роботі в режимі HTTP-fallback без P2P-активності.

На рисунках 3.1 та 3.2 представлено використання ресурсів центрального процесора та оперативної пам'яті для сценаріїв, отримані за допомогою Android Studio Profiler.

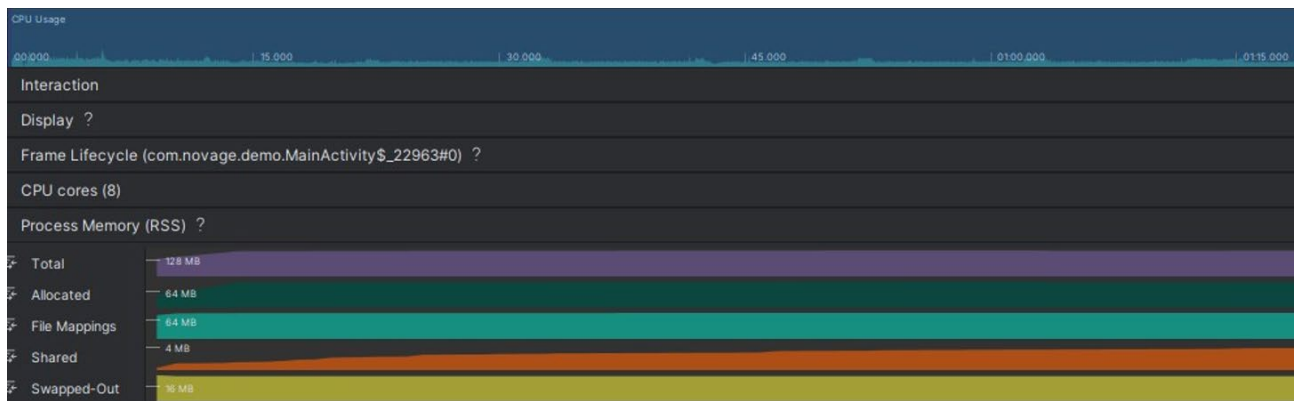


Рис. 3.1 – Споживання ресурсів мобільного клієнту без інтеграції p2pml

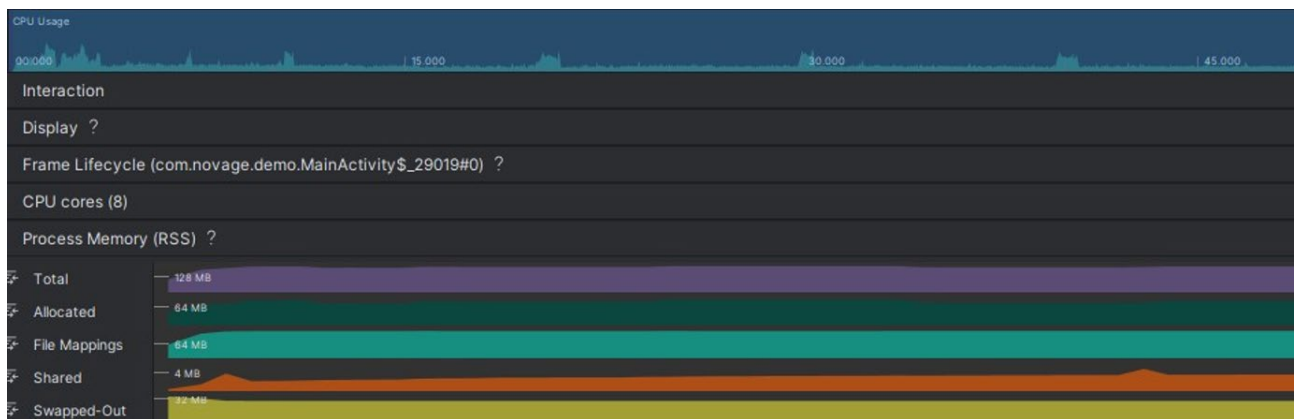


Рис. 3.2 – Споживання ресурсів мобільного клієнту з інтеграцією p2pml

З представлених даних, при відтворенні контенту напряму з CDN середнє навантаження на CPU становило близько 10-15% з піковими значеннями, що не перевищували 25%. При використанні бібліотеки p2pml-kotlin в режимі HTTP-fallback без P2P-активності, середнє навантаження на CPU зросло і становило вже 20-25%, а пікові значення сягали 45%. Це свідчить про додаткове навантаження, що вноситься роботою додаткових компонентів бібліотеки, зокрема фоновією роботою прихованого WebView, вбудованого Ktor-сервера, який проксує запити, та JavaScript-ядра P2P Media Loader.

Аналогічна тенденція спостерігається і для споживання оперативної пам'яті. У першому сценарії середнє споживання RAM тестовим додатком становило близько 140 МБ. У другому сценарії, з активованою бібліотекою p2pml-kotlin, цей показник зріс до 230 МБ. Додаткове споживання пам'яті обсягом приблизно 90 МБ пояснюється ресурсами, необхідними для ініціалізації та роботи компонента WebView, а також для внутрішніх буферів та сховищ метаданих самої бібліотеки. Важливо відзначити, що, незважаючи на зростання,

показники використання ресурсів залишалися в межах допустимих для стабільної роботи тестового пристрою.

В обох тестових сценаріях весь медіатрафік надходив виключно з HTTP-джерела, оскільки P2P-обмін був відсутній. Загальний обсяг завантажених даних для повного відтворення відео становив 478.94 МБ. Ідентичність цього показника в обох сценаріях підтверджує коректність роботи fallback-механізму бібліотеки p2pml-kotlin, яка за відсутності пірів успішно завантажує весь необхідний контент через HTTP. Цей обсяг трафіку є базовим показником, відносно якого в наступних експериментах буде розраховуватися ефективність розвантаження CDN.

Проведені базові тести показали, що інтеграція бібліотеки p2pml-kotlin створює очікуване додаткове навантаження на CPU та RAM мобільного пристрою. Додаткове навантаження є "ціною" за можливість використання P2P-технологій. Наступні експерименти мають на меті визначити, наскільки переваги від P2P-обміну можуть компенсувати це додаткове навантаження на ресурси.

Оцінка ефективності P2P-доставки та кросплатформної сумісності

Наступним кроком було дослідження роботи P2P-механізму в мережі, що складається з різних типів клієнтів. Було проаналізовано базовий сценарій взаємодії одного мобільного та одного вебклієнту, а також сценарій підключення мобільного клієнта до вже активного P2P-рою. На рисунках 3.3 та 3.4 показані інтерфейси тестових додатків зі збором статистики трафіку на Android та у веббраузері.



Video URL (HTTPS only):

<https://test-streams.mux.dev/x>

Player:

Vidstack

Apply

Create new peer

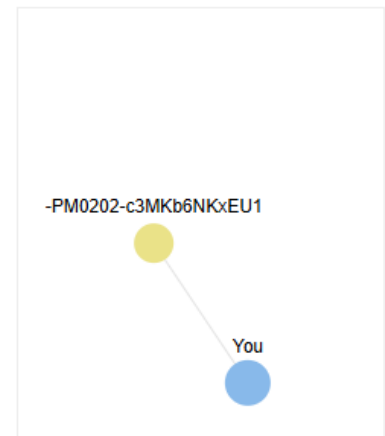
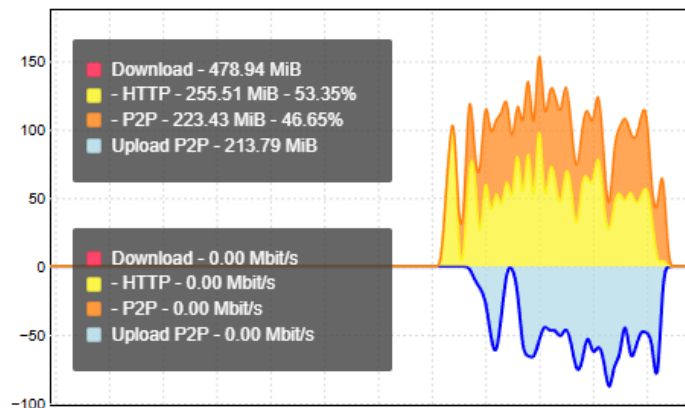


Рис. 3.3 – Інтерфейс вебклієнта зі статистикою трафіку

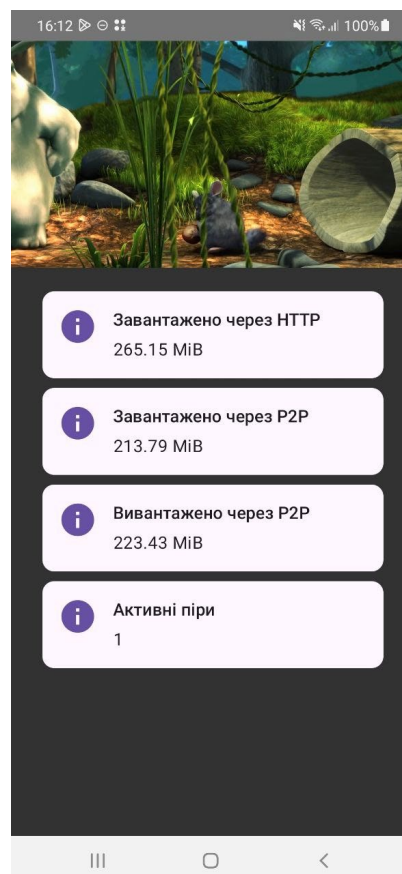


Рис. 3.4 – Інтерфейс мобільного клієнта зі статистикою трафіку

У сценарії, де тестувалася взаємодія мобільного клієнта та одного вебклієнту при одночасному старті відтворення через Wi-Fi, були отримані значущі результати (рис. 3.5). Мобільний клієнт зміг отримати 44.71% медіатрафіку через P2P-канал, завантаживши 213.79 MiB через P2P та 265.15 MiB з CDN. Це дозволило розвантажити CDN майже вдвічі порівняно з базовим сценарієм. Водночас вебклієнт показав схожий результат, отримавши 46.65% трафіку через P2P, а саме 223.43 MiB з P2P та 255.51 MiB з HTTP. Важливо відзначити, що обсяг даних, вивантажених мобільним клієнтом – 223.43 MiB, дорівнює обсягу, завантаженому вебклієнтом через P2P. Це наочно підтверджує успішну кросплатформну взаємодію між реалізаціями p2pml-kotlin та P2P Media Loader JS.

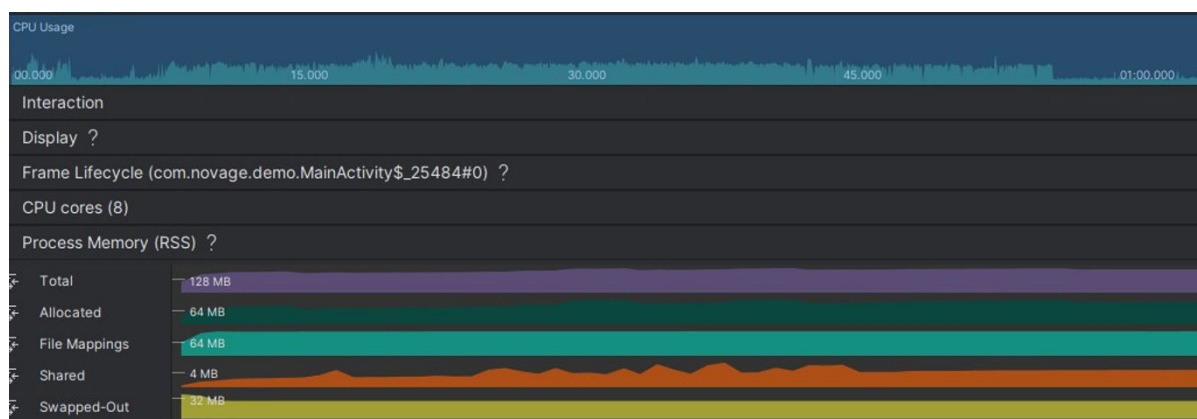


Рис. 3.5 – Використання ресурсів мобільного клієнта при передачі й отриманні даних з одним активним піром

При цьому активний P2P-обмін призвів до зростання використання ресурсів на мобільному пристрої: середнє навантаження на CPU зросло до ~30%, а використання RAM – до 263 МБ, порівняно з показниками в сценарії HTTP-fallback.

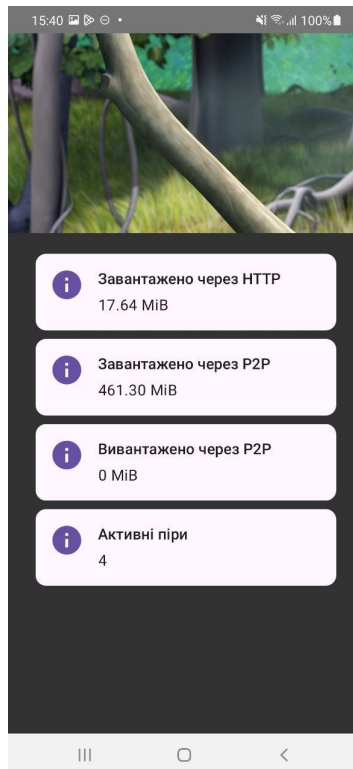


Рис. 3.6 – Інтерфейс мобільного клієнта зі статистикою трафіку, який підключений до мережі з 4 активних пірів

У сценарії, який моделював підключення мобільного клієнта до вже активного P2P-рою з 3-5 пірів (рис. 3.6), що мали попередньо закешований контент, ефективність P2P-обміну досягла пікових значень. Мобільний клієнт зміг отримати 96.32% всього необхідного медіатрафіку від інших пірів. Це означає, що було завантажено 461.30 MiB через P2P, і лише 17.64 MiB з CDN. Такий високий показник розвантаження CDN демонструє значний потенціал та ефективність масштабування розробленої системи в сприятливих умовах.

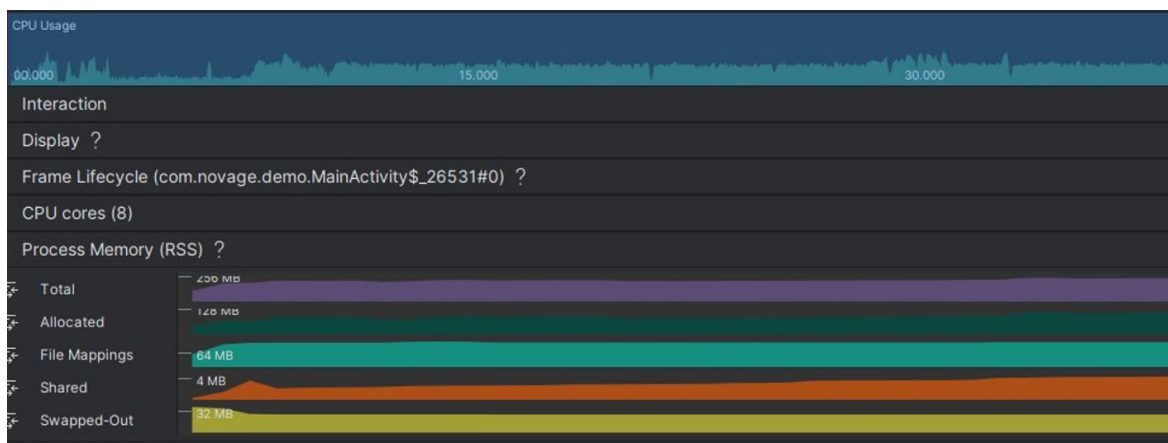


Рис. 3.7 – Використання ресурсів мобільного клієнта при передачі й отриманні даних з активним пірами

Середнє навантаження на CPU в цьому сценарії зросло до 30-34%, що є дещо вищим, ніж у сценарії з одним піром, і пояснюється необхідністю управляти з'єднаннями та обмінюватися даними з кількома пірами одночасно (рис. 3.7). При цьому споживання оперативної пам'яті залишилося на стабільному рівні близько 258 МБ.

Аналіз роботи системи за умов нестабільного P2P-з'єднання

Окрім оцінки ефективності в ідеальних умовах, важливим є дослідження поведінки системи в реалістичних сценаріях, де учасники P2P-мережі можуть несподівано її покидати. Для цього було проведено експеримент з плинністю пірів, де один з пірів періодично відключався та підключався до рою. Основна увага приділялася стабільності відтворення та джерелам трафіку на мобільному клієнті.

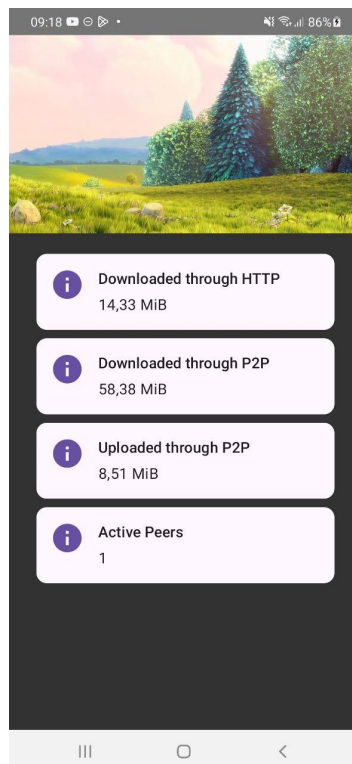


Рис. 3.8 – Робота системи в стабільному режимі з активним P2P-обміном

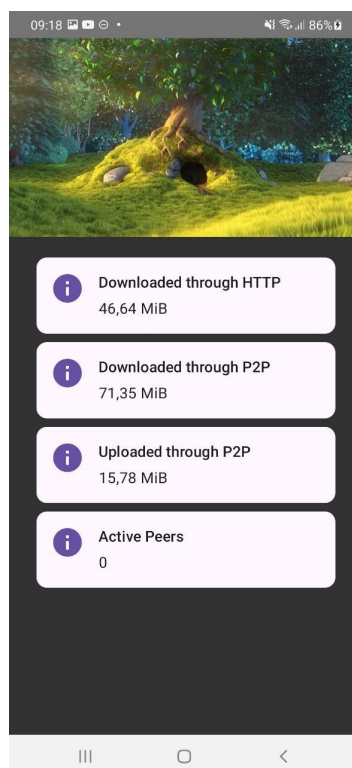


Рис. 3.9 – Робота системи одразу після відключення P2P-джерела

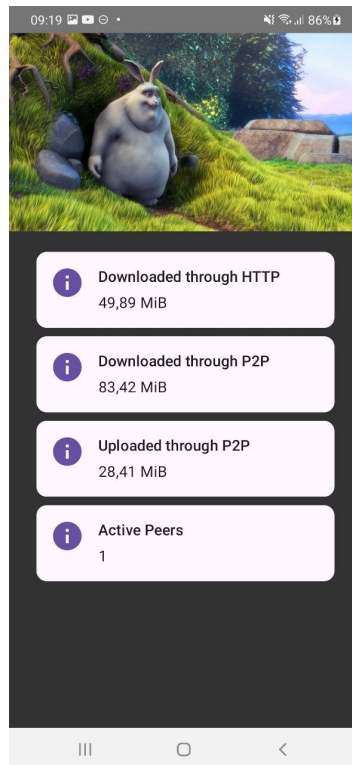


Рис. 3.10 – Робота системи після відновлення P2P-джерела

На рисунку 3.8 показано початковий етап експерименту, коли мобільний клієнт успішно обмінюється даними з іншим вебклієнтом. Статистика трафіку показує, що частина даних надходить через P2P-канали, забезпечуючи високий рівень розвантаження CDN. Система працює у стабільному режимі, відтворення відбувається без переривань.

На наступному етапі вебклієнт був навмисно відключений від мережі. Рисунок 3.9 фіксує стан системи невдовзі після цього моменту. Ключовим спостереженням є те, що бібліотека `p2pml-kotlin` коректно обробила розрив з'єднання. P2P-трафік від відключеного піра припинився, однак система автоматично та безшовно компенсувала цю втрату, миттєво збільшивши завантаження з резервного HTTP-джерела. Найважливішим результатом на цьому етапі є відсутність будь-якої буферизації або "зависання" на мобільному клієнті, що підтверджує високу надійність реалізованої системи.

Рисунок 3.10 демонструє здатність системи до самовідновлення. Після того, як "нестабільний" вебклієнт повернувся в мережу, P2P-рушій через певний час виявив його, успішно відновив з ним з'єднання і знову почав отримувати від нього медіасегменти. Це видно по відновленню P2P-трафіку від цього джерела.

Система знову автоматично знизилася обсяг завантажень з CDN, повернувшись до ефективного P2P-режиму.

Проведений експеримент з плинністю пірів демонструє високу стійкість та надійність розробленої системи. Бібліотека p2pml-kotlin здатна коректно обробляти раптові відключення учасників мережі, забезпечуючи безперебійне відтворення для кінцевого користувача за рахунок ефективного та швидкого перемикавання на HTTP-джерело. Здатність до відновлення P2P-з'єднань після повернення пірів у мережу підтверджує динамічність та адаптивність реалізованого P2P-механізму.

Висновки до розділу 3

У даному розділі було представлено результати експериментального дослідження, проведеного за розробленою методикою для оцінки ефективності розробленої системи p2pml-kotlin на платформі Android. Експерименти показали, що інтеграція P2P-механізму створює помірне та очікуване додаткове навантаження на ресурси мобільного пристрою, яке повністю компенсується перевагами від P2P-обміну в реальних умовах.

Було експериментально підтверджено успішну кросплатформну сумісність між мобільним клієнтом p2pml-kotlin та вебклієнтом, а також високу ефективність розвантаження CDN, яка сягала понад 96% при підключенні до активного P2P-рою з кількох пірів. Аналіз використання ресурсів показав, що приріст навантаження на CPU є помірним і пов'язаний з обробкою P2P-з'єднань, тоді як споживання оперативної пам'яті залишається стабільним завдяки реалізованим у розділі 2 оптимізаціям кешування.

Окрім показників ефективності, ключовим результатом дослідження стало підтвердження високої надійності та стійкості системи. Експеримент з плинністю пірів продемонстрував, що розроблена бібліотека здатна коректно обробляти раптові відключення учасників P2P-мережі, забезпечуючи безперебійне відтворення для кінцевого користувача за рахунок ефективного та

швидкого перемикання на HTTP-джерело. Також було доведено здатність системи до самовідновлення P2P-з'єднань після повернення пірів у мережу.

Таким чином, результати експериментального дослідження підтверджують, що розроблена в рамках даної кваліфікаційної роботи система p2pml-kotlin є працездатним та ефективним рішенням, яке дозволяє значно знизити залежність від централізованих серверів при доставці медіаконтенту на мобільні платформи.

ВИСНОВКИ

У даній кваліфікаційній роботі було вирішено актуальну науково-практичну задачу підвищення ефективності систем потокової передачі медіа шляхом оптимізації існуючих P2P-технологій та їх адаптації для використання на мобільних платформах. Метою роботи було створення та дослідження ефективної системи P2P-доставки медіаконтенту для Android-додатків на основі бібліотеки P2P Media Loader.

В ході виконання роботи було проведено комплексний аналіз предметної області, що включав огляд сучасних протоколів стрімінгу, принципів роботи P2P-мереж, існуючих комерційних та відкритих P2P-рішень, а також базової бібліотеки P2P Media Loader. Цей аналіз дозволив виявити ключові проблеми та виклики, зокрема низьку ефективність використання ресурсів та відсутність готових гнучких рішень для мобільних платформ, що обґрунтувало необхідність даного дослідження.

На основі аналізу було здійснено низку значних оптимізацій та вдосконалень базової JavaScript-бібліотеки P2P Media Loader.

Головним практичним результатом роботи стала розробка бібліотеки мобільної інтеграції p2pml-kotlin для платформи Android. Наукова новизна роботи полягає в запропонованому архітектурному підході, що поєднує використання прихованого WebView для виконання оригінального JavaScript-ядра P2P Media Loader, та вбудованого Ktor-сервера, що виступає в ролі локального проксі для прозорості інтеграції з нативними медіаплеєрами, як ExoPlayer.

Для підтвердження ефективності розробленої системи було проведено експериментальне дослідження. Результати підтвердили, що мобільна інтеграція p2pml-kotlin є працездатним та ефективним рішенням. Було доведено успішну кросплатформну взаємодію між мобільними та вебклієнтами. Експерименти показали, що система дозволяє досягти значного розвантаження CDN – до 96% в умовах активного P2P-рою, що є суттєвою перевагою. Також було кількісно оцінено додаткове навантаження, який вносить бібліотека на ресурси мобільного

					ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
						71

пристрою, та доведено, що він є помірним і повністю виправдовується значною економією мережевого трафіку.

Практична значущість отриманих результатів полягає у створенні відкритої, готової до використання бібліотеки p2pml-kotlin, яка надає Android-розробникам зручний інструмент для інтеграції P2P-технологій у свої медіа-додатки. Це дозволяє зменшити витрати на CDN-інфраструктуру для постачальників контенту та покращити стабільність стрімінгу для кінцевих користувачів. Оптимізації, внесені в базову бібліотеку P2P Media Loader, також є внеском у розвиток відповідного open-source проекту.

Таким чином, мета дипломної роботи була повністю досягнута. Розроблена та досліджена система P2P-доставки медіа для мобільних платформ продемонструвала свою ефективність та життєздатність.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Демченко Д.О., Савіцький Р.С. Ефективність використання peer-to-peer технологій у стрімінгових сервісах / Д.О. Демченко // Інформаційні технології – 2025: зб. тез XII Всеукраїнської науково-практичної конференції молодих учених, [Київ], 15 трав. 2025, – Київ, 2025. – С.121-122.
2. Cisco Annual Internet Report (2018–2023) White Paper. [Електронний ресурс]. – Режим доступу: <https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html> (дата звернення 21.01.2025).
3. Xu, L. D., He, W., & Li, S. Internet of Things in Industries: A Survey. [Електронний ресурс]. / L.D. Xu // IEEE Transactions on Industrial Informatics. – 2014. Vol. 10, No. 4. – Режим доступу: https://www.researchgate.net/publication/270742269_Internet_of_Things_in_Industries_A_Survey (дата звернення 21.01.2025).
4. Adhikari, V. K., Guo, Y., Hao, F., Varvello, M., Hilt, V., Steiner, M., & Zhang, Z.-L. Unreeling Netflix: Understanding and Improving Multi-CDN Movie Delivery. [Електронний ресурс]. /V. K. Adhikari // Proceedings IEEE INFOCOM. – 25-30 March 2012. – Режим доступу: <https://ieeexplore.ieee.org/document/6195531> (дата звернення 21.01.2025).
5. Androutsellis-Theotokis, S., Spinellis, D. A survey of peer-to-peer content distribution technologies. [Електронний ресурс] / D. A. Spinellis // ACM Computing Surveys. – December 2004, – 36(4), – С. 335-371 – Режим доступу: <https://www.spinellis.gr/pubs/jrnl/2004-ACMCS-p2p/html/AS04.html> (дата звернення 21.01.2025).

6. W3C. Web Real-Time Communications (WebRTC) [Электронный ресурс]. – Режим доступа: <https://www.w3.org/TR/webrtc/> (дата звернения 21.01.2025).
7. Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols [Электронный ресурс]. – Режим доступа: <https://datatracker.ietf.org/doc/html/rfc8445> (дата звернения 20.03.2025).
8. Microsoft Official Blog. Microsoft acquires Peer5 to enhance live video streaming in Microsoft Teams [Электронный ресурс]. – Режим доступа: <https://techcommunity.microsoft.com/blog/microsoftteamsblog/microsoft-acquires-peer5-to-enhance-live-video-streaming-in-microsoft-teams/2628950> (дата звернения 20.03.2025).
9. SwarmCloud: Distributed Live Streaming Solution: [Электронный ресурс]. – Режим доступа: <https://www.swarmcloud.net/en/> (дата звернения 05.04.2025).
10. Novage. P2P Media Loader: Technical Overview [Электронный ресурс]. – Режим доступа: <https://novage.com.ua/p2p-media-loader/technical-overview> (дата звернения 05.04.2025).
11. Android Developers. WebView Api Reference [Электронный ресурс]. – Режим доступа: <https://developer.android.com/reference/android/webkit/WebView> (дата звернения 05.04.2025).
12. Ktor Project. Official Documentation [Электронный ресурс]. – Режим доступа: <https://ktor.io/> (дата звернения 05.04.2025).
13. Square. OkHttp Library [Электронный ресурс]. – Режим доступа: <https://square.github.io/okhttp/> (дата звернения 05.04.2025).
14. Google Developers. ExoPlayer Documentation [Электронный ресурс]. – Режим доступа:

<https://developer.android.com/guide/topics/media/exoplayer> (дата
звернення 05.04.2025).

15. Kotlin Language Documentation [Електронний ресурс]. – Режим
доступу: <https://kotlinlang.org/docs/home.html> (дата звернення
05.04.2025).
16. Elements of Reusable Object-Oriented Software. / Gamma E., Helm R.,
Johnson R., Vlissides J. Design Patterns. – Addison-Wesley, 1994.
17. Android Developers. WebViewCompat class documentation.
[Електронний ресурс]. – Режим доступу:
[https://developer.android.com/reference/androidx/webkit/WebViewCom
pat](https://developer.android.com/reference/androidx/webkit/WebViewCompat) (дата звернення 25.05.2025).

ДОДАТКИ

					ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
						76

```

suspend fun handleSegmentRequest(call: ApplicationCall) {
    val segmentUrlParam =
        call.request.queryParameters["segment"]
        ?: return call.respondText(
            text: "Missing 'segment' parameter",
            status = HttpStatusCode.BadRequest,
        )

    val decodedSegmentUrl = Utils.decodeBase64Url(segmentUrlParam)
    val byteRange = call.request.headers[HttpHeaders.Range]

    try {
        val isCurrentSegment = parserService.isCurrentSegment(decodedSegmentUrl)
        if (p2pEngineStateManager.isEngineDisabled() || !isCurrentSegment) {
            Logger.w(
                TAG,
                message: "P2P engine is disabled or segment is not current," +
                    " falling back to direct fetch for $decodedSegmentUrl.",
            )
            fetchAndRespondWithSegment(call, decodedSegmentUrl, byteRange)
            return
        }

        val deferredSegmentBytes =
            webViewManager.requestSegmentBytes(decodedSegmentUrl)
            ?: throw IllegalStateException("P2P engine is disabled")
        val segmentBytes = deferredSegmentBytes.await()

        Logger.d(TAG, message: "Segment bytes received for $decodedSegmentUrl")
        respondSegment(call, segmentBytes, isPartial: byteRange != null)
    } catch (e: SegmentAbortedException) {
        val message = "Segment $decodedSegmentUrl | ${e.message}"

        Logger.w(TAG, message)
        call.respondText(message, status = HttpStatusCode.RequestTimeout)
    } catch (e: SegmentNotFoundException) {
        Logger.e(
            TAG,
            message: "Segment $decodedSegmentUrl not found: ${e.message}," +
                " falling back to direct fetch.",
        )
        fetchAndRespondWithSegment(call, decodedSegmentUrl, byteRange)
    } catch (e: Exception) {
        Logger.e(
            TAG,
            message: "Error handling segment $decodedSegmentUrl request: ${e.message}," +
                " falling back to direct fetch.",
            e,
        )
        fetchAndRespondWithSegment(call, decodedSegmentUrl, byteRange)
    }
}

```

Рисунок А.1 – Фрагмент коду з класу SegmentHandler: логіка обробки запиту на медіасегмент з P2P-спробою та HTTP-фолбеком

```

/**
 * Initializes and starts P2P media streaming components.
 *
 * @param exoPlayer ExoPlayer instance for media playback
 * @throws IllegalStateException if called in an invalid state
 */
fun start(exoPlayer: ExoPlayer) {
    Logger.d(TAG, message: "Starting P2P Media Loader with ExoPlayer")
    prepareStart(context, ExoPlayerPlaybackProvider(exoPlayer))
}

/**
 * Initializes and starts P2P media streaming components.
 *
 * @param getPlaybackInfo Function to retrieve playback information
 * @throws IllegalStateException if called in an invalid state
 */
fun start(getPlaybackInfo: () -> PlaybackInfo) {
    Logger.d(TAG, message: "Starting P2P Media Loader with playback info callback")
    prepareStart(context, ExternalPlaybackProvider(getPlaybackInfo))
}

private fun prepareStart(
    context: Context,
    provider: PlaybackProvider,
) {
    if (appState == AppState.STARTED) {
        val errorMessage = "Cannot start P2PMediaLoader in state: $appState"
        Logger.e(TAG, errorMessage)
        throw IllegalStateException(errorMessage)
    }

    job = Job()
    scope = CoroutineScope( context: job!! + Dispatchers.Main)
    playbackProvider = provider

    initializeComponents(context, provider)

    appState = AppState.STARTED
}

```

Рисунок Б.1 – Фрагмент коду з класу P2PMediaLoader: ініціалізація
компонентів