ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ТЕХНОЛОГІЙ ЦИФРОВОГО РОЗВИТКУ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Клієнт-серверна система потокової передачі аудіо з мобільного пристрою на ПК на основі Android SDK»

на здобуття освітнього ступеня бакалавра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Технології цифрового розвитку»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Ігор КРЄПКІЙ
(підпис)

Виконав: здобувач вищої освіти групи ТЦР-44
Ігор КРЄПКІЙ

Керівник: доктор філософії (PhD)
Дмитро НІЩЕМЕНКО

Рецензент:

Київ 2026

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Технологій цифрового розвитку
Ступінь вищої освіти Бакалавр
Спеціальність 121 Інженерія програмного забезпечення
Освітньо-професійна програма «Технології цифрового розвитку»

ЗАТВЕРДЖУЮ
завідувач кафедри
Технологій цифрового розвитку
_____ Вікторія ЖЕБКА
«_____» _____ 2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Крєпкому Ігорю Тарасовичу

1. Тема кваліфікаційної роботи: «Клієнт-серверна система потокової передачі аудіо з мобільного пристрою на ПК на основі Android SDK»
   керівник кваліфікаційної роботи Дмитро Ніщеменко,
   затверджені наказом Державного університету інформаційно-комунікаційних технологій від «20» лютого 2026 р. №51.

2. Строк подання кваліфікаційної роботи «____» _____ ____ р.

3. Вихідні дані до кваліфікаційної роботи:

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
   1. Аналіз існуючих програмних рішень для передачі аудіосигналу з мікрофона телефона на комп'ютер.
   2. Формулювання вимог та постановка задачі розробки застосунку.
   3. Проєктування архітектури системи передачі аудіосигнала.
   4. Реалізація клієнтської та серверної частини застосунку.

5. Проведення тестування, аналіз результатів і оцінка ефективності системи.

5. Перелік графічного матеріалу: *презентація*
    1. Аналіз аналогічних рішень.
    2. Архітектура програмного комплексу.
    3. Технології обробки та передавання потоку.
    4. Механізм автовиявлення (Service Discovery).
    5. Програмна реалізація клієнта (Android).
    6. Програмна реалізація сервера (PC).
    7. Інтерфейс користувача та розгортання.
    8. Результати тестування системи.
    9. Висновки та перспективи розвитку.

6. Дата видачі завдання «20» лютого 2026 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	20.02.2026 - 26.02.2026	
2	Аналіз та дослідження існуючих аналогів	27.02.2026 - 05.03.2026	
3	Формулювання вимог і постановка задачі	06.03.2026 - 11.03.2026	
4	Проєктування архітектури клієнт-серверної системи	12.03.2026 - 19.03.2026	
5	Реалізація клієнтської (Android) та серверної (Java) частини	20.03.2026 - 08.04.2026	
6	Тестування та оптимізація системи передачі аудіо	09.04.2026 - 16.04.2026	
7	Оформлення роботи: вступ, висновки, реферат	17.04.2026 - 24.04.2026	
8	Попередній захист роботи	27.04.2026	

Здобувач вищої освіти _____ Ігор КРЄПКІЙ
(підпис)

Керівник
кваліфікаційної роботи _____ Дмитро НІЩЕМЕНКО
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 61 стор., 8 табл., 12 рис., 25 джерел.

Мета роботи – проєктування та програмна реалізація клієнт-серверної системи для передачі аудіо з Android-пристрою зі зменшенням наскрізної затримки передачі аудіосигналу до рівня, придатного для інтерактивного голосового застосування.

Об'єкт дослідження – процеси потокової передачі нестиснених цифрових аудіосигналів між мобільними та стаціонарними вузлами у локальній мережі.

Предмет дослідження – програмне забезпечення, яке реалізує захоплення, інкапсуляцію, транспортування та відтворення PCM-потоку з використанням протоколу UDP та сервісу автовиявлення mDNS.

Короткий зміст роботи: у роботі розроблено клієнт-серверну систему потокової передачі нестисненого аудіо з мобільного пристрою на стаціонарний ПК. Проаналізовано існуючі програмні продукти, виявлено обмеження їхніх пропрієтарних протоколів. Обґрунтовано використання відкритого протоколу UDP для досягнення мінімальної мережевої затримки. Архітектура комплексу працює в межах локальної Wi-Fi-мережі без зовнішніх серверів. Програмна реалізація клієнта виконана за допомогою Android SDK (AudioRecord, NsdManager), а серверної частини на базі Java SE 17 (Java Sound API, JmDNS). Сервер приймає датаграми і направляє нестиснений PCM-потік (44,1 кГц, 16 біт, моно) у віртуальний аудіокабель. Реалізовано механізм автовиявлення вузлів за допомогою протоколу mDNS/DNS-SD. Проведено функціональне тестування системи у реальних умовах. Зафіксовано наскрізну латентність на рівні 42–80 мс. Практична новизна отриманих результатів полягає у розробці та реалізації архітектури, що поєднує відкритий нестиснений PCM-over-UDP транспорт із механізмом автовиявлення вузлів за протоколом DNS-SD, що забезпечує мінімальну затримку без використання пропрієтарних драйверів

та ручної конфігурації IP-адрес. Сферою використання є перетворення смартфона на бездротовий мікрофон для десктопного програмного забезпечення (OBS Studio, Discord, Zoom) без необхідності встановлення пропрієтарних драйверів на хост-машині.

КЛЮЧОВІ СЛОВА: ПОТОКОВА ПЕРЕДАЧА АУДІО, PCM, UDP, MDNS, DNS-SD, ANDROID, JAVA SOUND API, AUDIORECORD, SOURCEDATALINE, КЛІЄНТ-СЕРВЕРНА АРХІТЕКТУРА, ЛОКАЛЬНА МЕРЕЖА, НАСКРІЗНА ЗАТРИМКА, ВІРТУАЛЬНИЙ АУДІОПРИСТРІЙ, JMDNS, NSDMANAGER

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API – Application Programming Interface (Інтерфейс програмування застосунків)

DNS-SD – DNS-Based Service Discovery (Виявлення сервісів на базі DNS)

IP – Internet Protocol (Міжмережевий протокол)

GUI – Graphical user interface (Графічний інтерфейс користувача)

LAN – Local Area Network (Локальна обчислювальна мережа)

mDNS – Multicast DNS (Багатоадресна система доменних імен)

PCM – Pulse-Code Modulation (Імпульсно-кодова модуляція)

RTP – Real-time Transport Protocol (Транспортний протокол реального часу)

TCP – Transmission Control Protocol (Протокол керування передачею)

UDP – User Datagram Protocol (Протокол користувацьких датаграм)

ОС – Операційна система

ПЗ – Програмне забезпечення

ПК – Персональний комп'ютер

ФВ – Функціональна вимога

НФВ – Нефункціональна вимога

ВСТУП

Поширення мобільних пристроїв зі вбудованими мікрофонами призвело до появи прикладного класу задач, у яких смартфон виступає як мікрофон для стаціонарного комп’ютера. Дистанційні онлайн-конференції, стримінг ігрового процесу, дубляж, запис подкастів у польових умовах та сценарії командної роботи у локальній мережі потребують каналу передачі звуку з мінімальною затримкою між пристроями. Стандартні програмні продукти або прив’язані до конкретної хмарної інфраструктури, або вимагають придбання пропрієтарного обладнання, або працюють виключно через Bluetooth-профіль HFP/HSP із характерним падінням якості до 8 кГц моно.

Запропонована у роботі система усуває перелічені обмеження шляхом побудови виділеного клієнт-серверного комплексу, орієнтованого на роботу в межах однієї широкомовної локальної мережі. Android-застосунок захоплює сигнал з мікрофона за допомогою системного API AudioRecord у форматі PCM 16 біт, 44100 Гц, моно, та передає потік на персональний комп’ютер засобами протоколу UDP. Серверна частина, реалізована як десктопний Java-застосунок із графічним інтерфейсом на базі бібліотеки Swing, приймає датаграми, записує їх у лінію відтворення SourceDataLine і спрямовує звук у системний віртуальний аудіопристрій. Пошук сервера на стороні телефону виконується автоматично через mDNS/DNS-SD за службовим типом _pcmic._udp.local., що усуває потребу у ручному введенні IP-адреси.

Актуальність теми зумовлена розривом між можливостями вбудованого мікрофона сучасного смартфона – з апаратним шумозаглушенням, багатомікрофонним масивом та ефективним попереднім кодуванням – та обмеженими інтерфейсами інтеграції такого пристрою з робочою станцією.

Використання USB-кабелю супроводжується особливостями драйверів та залежністю від апаратного забезпечення, а Bluetooth-з’єднання у режимі гарнітури спричиняє деградацію частотного діапазону. Бездротовий канал Wi-Fi надає пропускну здатність, достатню для передачі нестисненого PCM-потоку зі швидкістю 705,6 кбіт/с, та відкриває можливість реалізації рішення на основі стандартних мережевих протоколів без ліцензійних обмежень.

Об’єктом дослідження виступають процеси потокової передачі нестиснених цифрових аудіосигналів між мобільними та стаціонарними вузлами у локальній мережі.

Предметом дослідження є програмне забезпечення, яке реалізує захоплення, інкапсуляцію, транспортування та відтворення PCM-потоку з використанням протоколу UDP та сервісу автовиявлення mDNS.

Метою роботи є проєктування та програмна реалізація клієнт-серверної системи для передачі аудіо з Android-пристрою зі зменшенням наскрізної затримки передачі аудіосигналу до рівня, придатного для інтерактивного голосового застосування (не більше 80 мс). Досягнення мети оцінюється через кількісні показники: наскрізну латентність (мс), частку втрачених пакетів (%), кількість кроків користувача для встановлення з'єднання.

Для досягнення мети сформульовано такі задачі:

1. Проаналізувати технологічні етапи обробки аудіосигналу та виявити компроміси між латентністю, якістю та надійністю передачі.
2. Дослідити модель клієнт-сервер у контексті живого аудіостримінгу та обґрунтувати вибір транспортного протоколу.
3. Розглянути існуючі протоколи передачі звуку (RTP/RTCP, HLS, DASH) і визначити межі їхньої застосовності.
4. Спроєктувати архітектуру системи, окреслити модульну структуру та життєвий цикл даних.

5. Реалізувати клієнтський Android-компонент на базі AudioRecord та DatagramSocket, а також серверний Java-компонент на базі SourceDataLine та DatagramSocket.
6. Інтегрувати автовиявлення сервера через DNS-SD (JmDNS на сервері, NsdManager на клієнті).
7. Провести функціональне тестування у реальних мережевих умовах та оцінити показники затримки й стабільності.

Практична новизна роботи полягає у специфічній комбінації функціональних можливостей, що не зустрічається у відкритих аналогах: одночасному поєднанні відкритого нестисненого PCM-over-UDP транспорту, стандартизованого автовиявлення сервера за протоколом DNS-SD (RFC 6762/6763) та кросплатформеного серверного компонента на базі JVM, що функціонує без встановлення нативного драйвера рівня ядра ОС. Такий підхід усуває залежність від закритих протоколів передачі та спрощує розгортання системи у гетерогенних локальних мережах. Розроблений комплекс інтегрується з віртуальним аудіокабелем (наприклад, VB-Audio Virtual Cable) та стає доступним як джерело захоплення для будь-якого застосунку, що використовує системний WASAPI – від OBS Studio до Discord і Zoom.

Результати роботи пройшли апробацію, за результатами було опубліковано 2 тези доповідей на конференціях:

1. Крєпкій І. Т. Програмна маршрутизація аудіопотоку в просторі користувача з використанням Java Sound API. VII Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях» : тези доп., 23 квіт. 2026 р. – Подано до друку.

2. Крєпкій І. Т. Програмна маршрутизація аудіопотоку в просторі користувача з використанням Java Sound API. Всеукраїнська науково-технічна конференція «Технології цифрового розвитку: програмні системи та децентралізація» : тези доп., 1–3 трав. 2026 р. – Подано до друку.

1. ТЕОРЕТИЧНІ ОСНОВИ ТА АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ

1.1. Основи потокової передачі аудіо в комп’ютерних мережах

Потокова передача аудіо – це технологія безперервного транспортування цифрового звукового сигналу від джерела до одного або декількох споживачів через мережу з пакетною комутацією. На відміну від моделі завантаження файлу, за якої відтворення починається після повного отримання контенту, стрімінг оперує безперервним потоком даних, фрагменти якого декодуються й відтворюються майже одразу після надходження. Ця властивість накладає жорсткі обмеження на сукупну наскрізну затримку, джиттер та стабільність пропускної здатності каналу.

Ключовим параметром такої системи є наскрізна латентність – проміжок часу між фізичною подією у мікрофоні та відповідною акустичною подією у динаміку приймача. Для голосового зв’язку у режимі діалогу прийнятним уважають значення до 150 мс, для моніторингу виконавця – до 20 мс. Перевищення порога призводить до ехо-ефектів, розсинхронізації з відео та суб’єктивного дискомфорту співрозмовників.

1.1.1. Технологічні етапи обробки та транспортування аудіосигналу

Шлях звукового сигналу від аналогового джерела до мережевого пакета складається з послідовних стадій, кожна з яких додає власний внесок у сумарну затримку та впливає на остаточну якість відтворення. Апаратний тракт перетворює коливання повітряного тиску на аналогову напругу через електретний або MEMS-перетворювач, після чого аналогово-цифровий перетворювач (ADC) здійснює дискретизацію сигналу. На Android-платформі ця операція інкапсульована в системний клас AudioRecord, сконфігурований у розробленому клієнті зі значеннями

SAMPLE_RATE = 44100, CHANNEL_IN_MONO, ENCODING_PCM_16BIT та джерелом MediaRecorder.AudioSource.MIC.

Частота дискретизації 44,1 кГц відповідає стандарту Audio CD і задовольняє теорему Найквіста–Шеннона для спектрального діапазону до 22,05 кГц, що перекриває верхню межу людського слуху. Розрядність 16 біт у двійковому доповненні забезпечує теоретичне відношення сигнал/шум близько 96 дБ та динамічний діапазон, достатній для якісного голосового й музичного запису. Моноканал обраний свідомо: мобільний пристрій як джерело зазвичай оперує одним мікрофоном у користувацькому сценарії, а відмова від стерео зменшує бітрейт удвічі.

Сирий PCM-потік при таких параметрах має сталий бітрейт (44100 = 705600) біт/с, або приблизно 86,1 КіБ/с. Ця величина визначає мінімальну необхідну пропускну здатність каналу та задає нижню межу обсягу буфера під час проєктування як клієнта, так і сервера.

На етапі пакетування до PCM-блоку додаються транспортні заголовки. У розробленій системі кадр, прочитаний з AudioRecord у буфер розміру AudioRecord.getMinBufferSize(), інкапсулюється безпосередньо у DatagramPacket та надсилається через DatagramSocket. Подібний мінімалістичний підхід виключає накладні витрати на кодеки та контейнерні формати, за рахунок чого досягається латентність пакетування, співмірна з періодом апаратного переривання ADC.

Передача через IP-мережу відбувається через комбінацію стеку UDP/IP та драйвера фізичного інтерфейсу Wi-Fi. На стороні сервера датаграма приймається у блокуючому виклику socket.receive() та демультиплексується за номером порту; у реалізації прийомного потоку використовується буфер розміром 16384 байти, що з запасом перевищує типовий розмір корисного навантаження PCM-кадру та стандартний MTU мережі Ethernet 1500 байт.

Декодування у розробленому комплексі відсутнє, оскільки передається несжатий PCM. Дані зі структури DatagramPacket напряму копіюються у буфер SourceDataLine через виклик speakers.write(packet.getData(), 0, packet.getLength()).

Цифро-аналоговий перетворювач операційної системи перетворює цифрові семпли на аналогову напругу або, за наявності віртуального аудіокабеля, передає потік іншому застосунку без проміжного DAC.

1.1.2. Буферизація та адаптивна передача

Мережі з пакетною комутацією за своєю природою недетерміновані. Пакети надходять на приймач з нерівномірними інтервалами через конкуренцію у каналі, повторні передачі на канальному рівні IEEE 802.11 та особливості планування у проміжних маршрутизаторах. Дисперсія міжпакетного інтервалу, відома як джиттер, порушує принцип рівномірного надходження семплів, необхідний для якісного відтворення. Без проміжного буфера будь-яка затримка пакета відносно середнього темпу викликала б спустошення лінії відтворення й характерний клік або розрив звуку.

Де-джиттерний буфер (jitter buffer) нівелює цей ефект, утримуючи невелику кількість даних перед видачею їх у DAC. У розробленому сервері роль буфера виконує внутрішня черга SourceDataLine, яка наповнюється в момент виклику speakers.write() та спустошується апаратним потоком відтворення. Розмір черги обирається системою на основі параметрів AudioFormat і, за потреби, може бути явно заданий через перевантаження line.open(format, bufferSize). Чим більший буфер, тим вища стійкість до джиттеру, але тим більша додана латентність.

Адаптивна передача передбачає динамічне узгодження параметрів потоку з поточним станом мережі. У класичних HTTP-стрімінгових системах сервер надає кілька якісних варіантів контенту, а клієнт обирає поточний на основі вимірюваної пропускної здатності. Для системи, орієнтованої на мінімальну затримку у локальній мережі, класична ABS-схема надлишкова: канал Wi-Fi 802.11n/ac у діапазоні 2,4 або 5 ГГц стабільно надає сотні мегабіт на секунду, що на два порядки перевищує потребу у 705,6 кбіт/с. Елементи адаптивності у розробленому комплексі

проявляються на рівні буфера приймача, який автоматично поглинає нерівномірність прибуття датаграм без зміни кодека чи бітрейту.

1.1.3. Виклики та проблеми

Проєктування системи живого аудіостримінгу стикається з низкою взаємопов’язаних обмежень. Зменшення буфера знижує латентність, але підвищує чутливість до джиттеру. Збільшення частоти дискретизації покращує якість, проте збільшує навантаження на канал. Застосування надійного транспорту зі гарантованою доставкою ліквідує втрати, але вводить неприйнятні затримки через повторні передачі.

Специфіка бездротового каналу додає ще один шар складності. Мобільний пристрій у режимі економії енергії може переводити Wi-Fi-радіо у сплячий стан, що проявляється як періодичні викиди затримки до сотень мілісекунд. Розроблений сервіс AudioStreamService вирішує проблему явним утриманням PowerManager.PARTIAL_WAKE_LOCK та WifiManager.WifiLock у режимі WIFI_MODE_FULL_LOW_LATENCY на Android 10+ (або WIFI_MODE_FULL_HIGH_PERF на старіших версіях). Перший блокує процесор від переходу у глибокий сон, другий змушує радіо залишатися у високопродуктивному режимі під час усього сеансу стримінгу.

Окрему проблему становить виявлення партнера. Перенесення у чужу мережу, зміна IP-адреси сервера через DHCP або підключення декількох приймачів до однієї мережі роблять ручне введення адреси вразливим до помилок. Запропоноване рішення спирається на протокол DNS-SD, реалізований на сервері через бібліотеку JmDNS, а на клієнті – через системний NsdManager із типом сервісу _pcmic._udp., що знімає потребу у ручній конфігурації.

1.2. Клієнт-серверна модель та її застосування

Архітектурний шаблон «клієнт-сервер» розподіляє функції системи між двома логічно відокремленими ролями: запитувач послуги та її надавач. Попри простоту ідеї, модель лежить в основі переважної більшості мережевих застосунків – від веб-сайтів до систем реального часу – і задає чіткий розподіл відповідальності між вузлами.

1.2.1. Основні принципи та компоненти

Сервер пасивно очікує з’єднань або вхідних повідомлень на заздалегідь визначеній точці – комбінації IP-адреси та номера порту. Клієнт активно ініціює взаємодію. У розробленій системі сервер, запущений на ПК, відкриває DatagramSocket на TCP/UDP-порту з номером за замовчуванням 50005, анонсує службу через mDNS та чекає на датаграми. Android-клієнт після виявлення сервісу створює вихідний DatagramSocket та відправляє PCM-фрагменти у напрямку виявленого ендпоінту.

Фундаментальне розведення ролей дозволяє інкапсулювати обчислювально інтенсивні операції (декодування, мікшування, маршрутизацію у віртуальний аудіокабель) на стороні сервера з розвиненою підсистемою JavaSound та залишити на клієнті лише захоплення та пакетування. Така асиметрія навантаження відповідає фізичним характеристикам пристроїв: персональний комп’ютер має стабільне живлення від мережі та повноцінний CPU, тоді як смартфон обмежений ресурсом акумулятора.

1.2.2. Застосування в потоковій передачі аудіо

У контексті аудіостримінгу модель «клієнт-сервер» реалізується у кількох різновидах. Класичний «один сервер – багато клієнтів» застосовується для трансляції концертів чи радіостанцій, коли сервер кодує потік один раз, а клієнти

отримують його паралельно через CDN. Зворотна схема «багато клієнтів – один сервер», обрана у розробленій системі, характерна для випадків, коли множина джерел надсилає сигнал у центральний вузол. У цьому випадку сервер виступає агрегатором та маршрутизатором до системних аудіопристроїв.

Прив'язка клієнта до сервера виконується за принципом точки «rendezvous». Телефон зв'язується з машиною за адресою та портом, тому обмін відбувається без проміжних серверів сесій. За потреби сервер можна масштабувати горизонтально, запустивши кілька екземплярів із різними іменами служб mDNS на одному або на кількох хостах; JmDNS підтримує одночасну реєстрацію декількох ServiceInfo.

1.2.3. Переваги та альтернативи

Перевагами клієнт-серверної моделі у межах LAN є передбачувана топологія, простота відстеження стану та можливість централізованого адміністрування. Клієнт позбавлений обов'язку підтримувати довгострокові з'єднання з іншими клієнтами, а сервер зберігає повний контроль над політикою обслуговування й авторизацією джерел.

Альтернативою виступає пірингова (P2P) модель, у якій кожен вузол одночасно виконує обидві ролі. Для аудіостримінгу вона ефективна у випадку великого числа учасників, коли розподілене постачання фрагментів знижує навантаження на одиничний сервер. Водночас P2P-архітектура ускладнює виявлення партнерів, синхронізацію потоків та обхід NAT, а також зазвичай потребує сторонньої трекерної інфраструктури, що робить її надлишковою для розв'язуваної задачі «один смартфон – один комп'ютер».

Ще однією альтернативою є повністю хмарна схема з relay-сервером у публічній мережі. Такий підхід дає змогу поєднати два кінцеві вузли, що перебувають за різними NAT, але додає відчутну географічну затримку та залежність від зовнішньої інфраструктури. У локальному сценарії, якому присвячено

це дослідження, обрана клієнт-серверна модель з автовиявленням є оптимальним компромісом між складністю, латентністю та керованістю.

1.3. Протоколи передавання аудіо

Вибір мережевого протоколу для транспортування цифрового звуку визначає верхню межу досяжної латентності, поведінку системи в умовах втрат пакетів та сумісність із проміжним мережевим обладнанням. На транспортному рівні моделі TCP/IP боротьба між надійністю та швидкістю зосереджена у дилемі TCP проти UDP. На прикладному рівні розроблено сімейства протоколів, які або розширюють можливості UDP у бік функцій реального часу (RTP/RTCP), або надбудовуються над HTTP для масового стрімінгу (HLS, DASH).

1.3.1. Транспортний рівень: TCP проти UDP

TCP (Transmission Control Protocol) – з’єднання-орієнтований протокол з гарантованою доставкою, впорядкуванням сегментів та контролем перевантаження. Кожен сегмент, що не отримав підтвердження за тайм-аут RTO, ретранслюється. Такий механізм забезпечує побітову відповідність прийнятих даних надісланим, але ціною змінної затримки: єдиний втрачений пакет зупиняє видачу усього потоку на приймач до моменту успішної повторної передачі (head-of-line blocking). Алгоритми контролю перевантаження на кшталт TCP Reno або CUBIC додатково згладжують відправку сегментів, що для живого аудіо сприймається як неприпустима штучна буферизація.

UDP (User Datagram Protocol) – протокол без встановлення з’єднання, без підтверджень та без контролю перевантаження. Датаграма або доставляється одразу, або втрачається остаточно. Накладні витрати обмежуються 8-байтовим заголовком. Саме такі характеристики виявляються бажаними для реального часу: втрата окремого 2,3-мілісекундного фрагмента PCM суб’єктивно сприймається як короткий

клік, натомість затримка у десятки мілісекунд через ретрансляцію порушує синхронність розмови.

Розроблений комплекс використовує саме UDP. На сервері створюється new DatagramSocket(port), а клієнт формує та надсилає пакети через socket.send(packet) без попереднього connect. Порт 50005 обрано з діапазону ephemeral/user ports (49152–65535), вільного від резервування IANA. Такий підхід зменшує ймовірність конфлікту з відомими службами й водночас дозволяє користувачеві перевизначити значення через поле JTextField у GUI.

1.3.2. Протоколи реального часу: RTP та RTCP

RTP (Real-time Transport Protocol, RFC 3550) – прикладний протокол, що надбудовується над UDP та додає до кожного пакета метадані, критичні для відтворення: послідовний номер для виявлення втрат і зміни порядку, 32-бітний тайм-штамп у одиницях частоти дискретизації для відновлення темпу, 7-бітний ідентифікатор типу навантаження (payload type) для вибору декодера та поле SSRC для мультиплексування кількох джерел у межах однієї сесії. Парний йому RTCP забезпечує зворотний канал зі статистикою якості та синхронізацією учасників.

Використання RTP доцільне у гетерогенних системах, де джерела надсилають різні кодеки й потоки потребують синхронізації з відео. Для розробленої системи, у якій існує єдине джерело, єдиний одержувач, єдиний кодек PCM та синхронізація з відеопотоком не передбачається, RTP-заголовок у 12 байт на кожен пакет став би зайвим накладним навантаженням. Обрано спрощену схему «чистий PCM поверх UDP», у якій семантика послідовності покладена на мережеву інфраструктуру, а відновлення темпу виконує буфер SourceDataLine.

1.3.3. HTTP-базовані протоколи: HLS та DASH

HLS (HTTP Live Streaming), розроблений Apple, та MPEG-DASH (Dynamic Adaptive Streaming over HTTP) побудовані на принципово іншій парадигмі. Живий

потік сегментується сервером на фрагменти тривалістю 2–10 секунд, описується маніфестом (.m3u8 або .mpd) та доставляється клієнту звичайними HTTP GET-запитами. Транспортний рівень – TCP зі всіма притаманними йому властивостями.

Сильні сторони HLS/DASH – масштабованість через CDN, вбудована підтримка адаптивного бітрейту та проходження корпоративних файрволів, що дозволяють лише HTTP(S). Слабкою стороною є структурна латентність: розмір сегмента встановлює нижню межу затримки у кілька секунд, що на порядки перевищує допустимий бюджет для живої передачі голосу. Сучасні варіанти LL-HLS (Low-Latency HLS) та CMAF-Chunked Transfer знижують затримку до 2–3 секунд, але навіть такі значення залишаються непридатними для обраного сценарію застосування, де цільова наскрізна латентність не перевищує десятків мілісекунд.

1.3.4. Адаптивна потокова передача (ABS) в HTTP-стрімінгу

ABS (Adaptive Bitrate Streaming) – стратегія, за якої один і той самий контент кодується у декількох якісних представленнях, а клієнт у процесі відтворення перемикається між ними залежно від вимірюваної пропускної здатності, розміру плеєрного буфера та обчислювальних можливостей пристрою. У стеку HLS/DASH ABS реалізується через Master-маніфест, який перераховує набір Variant Streams із різним бітрейтом; алгоритми на кшталт BOLA або Buffer-Based Rate Adaptation ухвалюють рішення про перемикання.

ABS виправдана у системах «один до багатьох» з непередбачуваним Інтернет-каналом. У розробленому комплексі, що працює виключно у локальній мережі зі стабільною та заздалегідь надлишковою пропускною здатністю, адаптація бітрейту позбавлена практичного сенсу: параметри потоку 44,1 кГц / 16 біт / моно фіксовані й становлять лише 0,7 Мбіт/с, тоді як навіть найскромніший Wi-Fi-канал стандарту 802.11n забезпечує десятки Мбіт/с корисної пропускної здатності. Елементи адаптивності у запропонованій системі присутні у вигляді утримання Wi-Fi-радіо у режимі FULL_LOW_LATENCY та буферизації на приймачі, що функціонально

покриває задачу компенсації джиттеру без потреби в повноцінній ABS-інфраструктурі.

1.4. Методи буферизації та обробки звукових потоків

Буферизація – проміжне зберігання порції семплів між етапами генерації, транспортування та відтворення – є фундаментальним інженерним прийомом у цифровій обробці звуку. Без буфера будь-які коливання продуктивності у будь-якій ланці передавального тракту негайно проявилися б як чутні артефакти: розриви, клацання, занижений темп. Проєктування ланцюга «ADC – мережа – DAC» майже повністю зводиться до правильного вибору розмірів та стратегії заповнення трьох незалежних буферів – апаратного на клієнті, мережевого у стеку операційної системи та лінії відтворення на сервері.

1.4.1. Призначення та механізм роботи буфера

Буфер функціонує як циклічна черга фіксованої ємності з двома незалежними вказівниками: читання та запису. Виробник вставляє нові семпли за вказівником запису, споживач вилучає їх за вказівиком читання; різниця між вказівниками відповідає поточному заповненню. Якщо виробник обганяє споживача, черга переповнюється і найстаріші семпли витісняються (overrun); якщо споживач обганяє виробника, черга спустошується і в потоці з’являється штучна пауза (underrun). Обидві ситуації є небажаними та породжують чутні артефакти.

У клієнтському компоненті системи апаратний буфер інкапсульований у класі AudioRecord. Його розмір обчислюється під час конструювання через виклик AudioRecord.getMinBufferSize(SAMPLE_RATE, CHANNEL_CFG, AUDIO_FMT), який повертає мінімальний ємнісний поріг, безпечний для заданої конфігурації. На типовому пристрої з частотою 44,1 кГц, моно та 16-бітним PCM значення коливається в межах 3,5–7 КіБ, що відповідає приблизно 20–40 мс звуку. Потік

StreamThread у циклі while (running) блокуюче зчитує порції даних викликом recorder.read(buf, 0, buf.length) та негайно інкапсулює їх у DatagramPacket, не здійснюючи жодної додаткової проміжної буферизації на рівні застосунку.

На стороні сервера розв'язуються дві буферні задачі одночасно. Перша – приймальний буфер UDP-сокета у ядрі ОС, який накопичує датаграми між моментом надходження з мережевого стека та викликом socket.receive() у кодовому потоці. У реалізації ReceiverThread розмір буфера корисного навантаження становить 16384 байти, чого достатньо для розміщення довільного UDP-пакета в межах максимального значення 65535 байт. Друга – буфер лінії відтворення SourceDataLine, наповнений викликами speakers.write(packet.getData(), 0, packet.getLength()). Цей буфер функціонує як класична де-джиттерна черга: апаратна підсистема відтворення вилучає з нього семпли зі сталим темпом, заданим частотою дискретизації, тоді як потік ReceiverThread наповнює його з нерівномірним темпом надходження датаграм.

Сумарна наскрізна латентність є алгебраїчною сумою часів перебування семпла в усіх буферах тракту: $T_{total} = T_{mic} + T_{net} + T_{os} + T_{line} + T_{dac}$. Мінімізація цієї суми вимагає узгодженого зменшення кожного доданка; збільшення будь-якого буфера для стійкості до джиттеру одразу погіршує інтерактивність системи.

1.4.2. Управління станом буфера

Стратегії управління буфером поділяють на статичні та адаптивні. Статична стратегія передбачає фіксований розмір, обраний на етапі проєктування за гіршим очікуваним джиттером. Переваги – простота реалізації та передбачувана латентність. Недолік – неможливість оптимізуватися під поточні умови мережі. У розробленому сервері застосовано саме статичний підхід: SourceDataLine відкривається викликом speakers.open(fmt) без явного задання розміру, через що JVM обирає значення за замовчуванням, типово близько 100 мс.

Адаптивна стратегія перераховує цільове заповнення буфера у реальному часі на основі статистики прибуття пакетів. Типовою метрикою є ковзна оцінка дисперсії

міжпакетного інтервалу; при її зростанні система нарощує цільове заповнення, при падінні – скорочує. У специфікації RTP рекомендується використовувати алгоритм, описаний у RFC 3550 (Appendix A.8), для оцінки джиттеру (J) як експоненціально згладженого відхилення:

$$J_i = J_{i-1} + \frac{\left|D_{i-1,i}\right| - J_{i-1}}{16}, \tag{1.1}$$

де $D_{i-1,i}$ – різниця транзитних часів сусідніх пакетів. У розробленій системі формальна метрика джиттеру не обчислюється, а компенсація виконується пасивно за рахунок ємності лінії відтворення: апаратний потік продовжує відтворення накопичених семплів, поки новий пакет перебуває у дорозі.

Обробка крайових ситуацій закладена на рівні потоку ReceiverThread. Примусова зупинка сервера через метод stopListening() викликає socket.close(), що розблоковує зависаючий socket.receive() винятком SocketException; відповідна гілка catch коректно завершує цикл, після чого викликаються speakers.stop() та speakers.close(). Блок finally гарантує звільнення ресурсів JmDNS та сокета незалежно від причин завершення потоку, що забезпечує стабільну роботу системи при багаторазових циклах перезапуску сервісу протягом однієї сесії.

1.4.3. Зв’язок буфера з адаптивним стрімінгом (ABS)

У парадигмі ABS буфер виконує ще одну роль, окрім компенсації джиттеру – роль джерела сигналу зворотного зв’язку для алгоритму вибору бітрейту. Клієнтський плеєр HLS чи DASH постійно спостерігає за тривалістю накопиченого контенту у секундах відтворення. Швидке спустошення буфера сигналізує про недостатню пропускну здатність і змушує перемкнутися на менш якісне представлення; стабільне наповнення, навпаки, дозволяє запитувати вищу якість.

Подібна схема підтримує три канонічні стани буфера: критично порожній (імовірний underrun, потрібне негайне перемикання вниз), номінальний (система працює у цільовому коридорі заповнення) та надлишковий (є запас для апгрейду

якості). Алгоритми ABR-адаптації реалізують цю логіку у вигляді скінченного автомата або оптимізаційної задачі; BOLA (Buffer Occupancy based Lyapunov Algorithm) формалізує рішення як мінімізацію лагранжіану, що об’єднує ризик underrun та штраф за низьку якість.

Для комплексу, описаного у роботі, проблема вибору бітрейту не стоїть, оскільки передається єдиний незмінний PCM-потік. Натомість проявляється зворотний зв’язок через механізми UDP-стека: якщо приймальний буфер сокета переповнюється, ядро ОС мовчки відкидає нові датаграми, не сповіщаючи ні відправника, ні одержувача. Відсутність вбудованого контролю перевантаження в UDP означає, що за умов перевантаженої мережі сервер поступово втрачатиме якусь частку пакетів без шансу на їхнє відновлення, однак для обраної локальної топології Wi-Fi імовірність такої події залишається незначною.

1.4.4. Обробка звукових потоків на стороні клієнта

Обробка потоку на стороні джерела у розробленій системі зведена до мінімуму з міркувань латентності. Тракт AudioRecord – byte[] – DatagramPacket – DatagramSocket.send() не містить жодного додаткового програмного етапу: ні ресемплінгу, ні фільтрації, ні шумозаглушення на рівні застосунку. Попередня обробка делегована апаратному DSP смартфона та системному AudioFlinger, який автоматично застосовує Acoustic Echo Cancellation та автоматичне регулювання підсилення до джерела MediaRecorder.AudioSource.MIC на більшості сучасних пристроїв.

Такий мінімалізм свідомо обраний як архітектурний компроміс. Будь-яке додаткове цифрове оброблення – наприклад, кодек Opus із характерною латентністю 20 мс на кадр – додавало б до загального бюджету затримки сталу величину без пропорційного виграшу у якості, оскільки вузьким місцем залишається не канал, а сам Wi-Fi-радіотракт. Відмова від стиснення також звільняє CPU смартфона та знижує енергоспоживання, що важливо для довготривалих сеансів.

На стороні приймача клієнтська обробка зосереджена у графічному інтерфейсі PCMicGUI. Користувацький вибір вихідного пристрою реалізовано через перелік Mixer.Info[], сформований викликом AudioSystem.getMixerInfo() та відфільтрований за наявністю getSourceLineInfo(). Подальша валідація формату виконується через mixer.isLineSupported(lineInfo), де lineInfo = new DataLine.Info(SourceDataLine.class, fmt). Якщо обраний мікшер не підтримує конфігурацію 44100/16/mono/signed/LE, застосунок згортає потік та виводить повідомлення про помилку замість мовчазного переходу на несумісний формат, що могло б спричинити відтворення зашумленого сигналу або повну тишу.

1.5. Огляд аналогічних програмних продуктів

Ринок програмних рішень для перетворення смартфона на зовнішній мікрофон персонального комп’ютера формувався протягом останнього десятиліття і сьогодні налічує десятки продуктів, що суттєво відрізняються за архітектурою, транспортним рівнем, моделлю ліцензування та цільовою аудиторією. Об’єктивне зіставлення власного рішення з існуючими потребує попередньої класифікації конкурентів та визначення набору критеріїв порівняння.

1.5.1. Категоризація програмних продуктів

За способом доставки сигналу від телефону до хоста розглянуті рішення поділяють на чотири групи. Перша – застосунки на базі USB-тетерингу, у яких телефон виступає композитним USB-аудіопристроєм або проксіює сигнал через ADB-порт. Представниками є WO Mic у режимі USB та ряд закритих утиліт виробників звукових карт. Такі рішення забезпечують мінімальну латентність за рахунок гарантованої ізохронної доставки USB, але прив’язані до фізичного кабелю.

Друга група – рішення на базі Bluetooth-профілів HFP/HSP. Їхня перевага у повній сумісності з будь-якою гарнітурою, недолік – примусове обмеження частоти

дискретизації 8 кГц для HFP або 16 кГц для HSP mSBC, що робить їх непридатними для музичних застосувань. Третя група – мережеві застосунки над Wi-Fi із власним закритим протоколом; сюди входять WO Mic (Wi-Fi-режим), Megaphone Free, iVCam Audio та комерційний DroidCam. Четверта група – застосунки на базі відкритих голосових протоколів, представлені переважно Mumble та його похідними на базі RTP over UDP з кодеком Opus чи CELT.

Розроблене у роботі рішення належить до третьої групи, але відрізняється від неї відкритим протоколом PCM-over-UDP, відсутністю програмного стиснення та вбудованим mDNS-автовиявленням замість ручного введення IP.

1.5.2. Порівняльний аналіз ключових гравців

WO Mic – одне з найпоширеніших рішень, підтримує USB-, Bluetooth-, Wi-Fi- та Wi-Fi Direct-режими. Застосовує пропрієтарний драйвер віртуального мікрофона для Windows, який встановлюється окремо. Аудіоформат у Wi-Fi-режимі обмежений 16 кГц моно, що задає верхню межу частотної відповіді 8 кГц. Латентність у Wi-Fi-режимі декларується на рівні 100–300 мс, що визначається комбінацією кодека та внутрішнього буфера. Сервіс-дискаверинг відсутній; користувач вручну вводить IP-адресу хоста у телефоні.

DroidCam від Dev47Apps позиціонується передусім як застосунок для веб-камери, але надає й аудіоканал. Реалізує власний протокол поверх HTTP, що зумовлює використання TCP та, відповідно, вищу латентність відносно UDP-рішень. Безкоштовна версія обмежує якість, розблокування потребує придбання DroidCamX.

iVCam виконаний за архітектурою, близькою до DroidCam: власний клієнт для Windows, власний застосунок для Android/iOS, авто-детект у межах LAN через пропрієтарний широкомовний протокол (імітація DNS-SD без дотримання RFC). Латентність за замовчуванням оптимізована для відео, аудіоканал працює як супутник відеопотоку.

AudioRelay – рішення, архітектурно найближче до розробленого: передача PCM (або стиснутого через Opus) поверх UDP, власний віртуальний аудіопристрій для Windows. Автовиявлення реалізоване, але через пропрієтарний широкомовний протокол, а не стандартний mDNS. Код закритий, безкоштовна версія обмежена 48 кГц без стиснення; розблокування функцій потребує придбання ліцензії.

Mumble – повнофункціональний відкритий голосовий клієнт, розроблений переважно для ігрових сценаріїв. Транспорт – UDP із кодеком Opus або CELT, підтримує стиснення, шифрування TLS та централізовану архітектуру з окремим сервером murmur. Латентність у LAN-режимі близька до 40 мс. Mumble вирішує задачу багатокористувацького голосового чату, а не трансляції одного мікрофона на один ПК, тому його застосування у досліджуваному сценарії вимагає встановлення окремого серверного процесу і не надає прямого інтегрування з віртуальним аудіокабелем.

У таблиці 1.1 зводено ключові характеристики розглянутих рішень за критеріями, релевантними для задачі перетворення смартфона на бездротовий мікрофон ПК.

Таблиця 1.1

Порівняльний аналіз аналогічних програмних продуктів

Критерій	WO Mic	DroidCam	iVCam	AudioRelay	Mumble	Розроблена система
Транспортний протокол	UDP (пропр.)	TCP/HTTP	UDP (пропр.)	UDP (пропр.)	UDP + TLS	UDP (відкритий)
Розрядність, біт	16	16	16	16	16	16
Частота дискретизації, кГц	16	44,1	44,1	48	48	44,1

Продовження таблиці 1.1

Кодек	пропр. стиск.	AAC	пропр.	PCM/Opus	Opus/CELT	PCM (без стиск.)
Автовиявлення сервера	Немає	Немає	пропр. Broadcast	пропр. Broadcast	сервер-список	mDNS (_pcmic._udp.)
Модель ліцензування	Freemium	Freemium	Freemium	Freemium	OSS (BSD)	OSS (навчальна)
Крос-платф. сервер	Win	Win/macOS/Linux	Win/macOS	Win/macOS/Linux	Win/macOS/Linux	Будь-яка JVM-сумісна
Інтеграція з віртуал. кабелем	Драйвер	Драйвер	Драйвер	Драйвер	-	через сторонній VB-Cable
Задекларована латентність, мс	100–300	150–500	80–200	40–150	40–80	30–80
Відкритість протоколу	Ні	Ні	Ні	Ні	Так	Так

Зіставлення показує, що жодне з розглянутих комерційних рішень не поєднує одночасно відкритий транспортний протокол, відсутність програмного стиснення, стандартизоване автовиявлення DNS-SD та мінімалістичний архітектурний дизайн без потреби у власному драйвері. Mumble, хоча й відкритий, зорієнтований на іншу задачу. Цей розрив у пропозиції ринку і формує нішу для розробки, що пропонується у дослідженні.

2. ПРОЄКТУВАННЯ КЛІЄНТ-СЕРВЕРНОЇ СИСТЕМИ

Розділ присвячено формалізації вимог до системи, вибору архітектурної схеми та декомпозиції комплексу на програмні модулі. Проєктні рішення подано мовою UML у вигляді діаграм класів, послідовності, станів та розгортання, виконаних у форматі PlantUML. Кожна діаграма супроводжується текстовим описом семантики та обґрунтуванням вибору відповідної проєктної альтернативи.

2.1. Постановка задачі та вимоги до системи

2.1.1. Огляд аналогічних програмних продуктів

Порівняльний аналіз існуючих рішень проведено у підрозділі 1.5 і зведено у таблиці 1.1. Для цілей постановки задачі обмежимося узагальненням виявлених недоліків комерційних продуктів. WO Mic обмежує частоту дискретизації 16 кГц та позбавлений автоматичного виявлення сервера. DroidCam використовує TCP-транспорт та HTTP-інкапсуляцію, що додає десятки мілісекунд до наскрізної затримки. iVCam та AudioRelay спираються на закриті протоколи і потребують купівлі ліцензії для зняття обмежень базової редакції. Mumble, попри відкритість вихідного коду, вирішує задачу багатокористувацького голосового чату і не надає прямого інтегрування з віртуальним аудіокабелем ПК.

Спільними рисами переглянутих рішень є або пропрієтарний характер протоколу, або структурна прив’язка до іншої цільової задачі. Жоден з розглянутих продуктів не пропонує одночасно: відкритий PCM-over-UDP-транспорт, стандартизоване автовиявлення через DNS-SD (RFC 6762/6763) та роботу без встановлення власного драйвера рівня ядра на стороні хоста. Виявлений розрив між функціональними потребами користувача та ринковою пропозицією становить основу для формулювання задачі розробки.

2.1.2. Обґрунтування розробки та постановка задачі

Задачею дослідження є створення програмного комплексу, що забезпечує передачу несжатого PCM-аудіо з Android-пристрою на персональний комп'ютер у межах локальної Wi-Fi-мережі без встановлення проміжного серверного ПЗ та без ручної конфігурації мережевих параметрів. Система повинна інтегруватися зі стандартними віртуальними аудіокабелями (VB-Audio Virtual Cable або еквівалент) та забезпечувати наскрізну латентність, придатну для інтерактивного голосового застосування.

Формально задача визначається через сукупність проєктних обмежень: аудіоформат 44,1 кГц / 16 біт / моно, транспорт UDP, автовиявлення через mDNS із типом сервісу _pcmic._udp., робота на пристроях з Android 8.0 (API 26) і вище, сервер як Java-застосунок з графічним інтерфейсом, сумісний з будь-якою операційною системою, що підтримує Java SE 17. Функціонування системи обмежується рамками однієї широкомовної IP-мережі; маршрутизовані сценарії та робота через NAT до задачі не включаються.

2.2. Вимоги до системи

Згідно з ISO/IEC/IEEE 29148:2018 вимоги поділено на функціональні, що описують поведінку системи, та нефункціональні, що задають якісні атрибути та обмеження. Обидві групи сформульовані у перевірюваній формі «система повинна…», що дозволяє однозначно зіставити їх з тест-кейсами у розділі 3.4.

2.2.1. Функціональні вимоги

Функціональні вимоги визначають мінімальний набір сценаріїв взаємодії, які система зобов'язана підтримувати у робочому стані:

1. ФВ-1. Серверний компонент повинен приймати конфігурацію номера UDP-порту через графічний інтерфейс та очікувати датаграми на заданому порту.

2. ФВ-2. Серверний компонент повинен надавати користувачеві перелік доступних вихідних аудіопристроїв, отриманих через AudioSystem.getMixerInfo(), та підтримувати вибір пристрою до старту сесії.
3. ФВ-3. Серверний компонент повинен реєструвати службу з типом _pcmic._udp.local. та іменем PC_Mic_Receiver у mDNS-просторі при старті прослуховування та скасовувати реєстрацію при зупинці.
4. ФВ-4. Клієнтський компонент повинен виявляти доступні сервери у локальній мережі через системний NsdManager та відображати результати у вигляді списку «ім’я (IP:порт)».
5. ФВ-5. Клієнтський компонент повинен здійснювати захоплення аудіо з мікрофона у форматі PCM 16 біт / 44 100 Гц / моно та передавати отриманий потік обраному серверу через UDP.
6. ФВ-6. Клієнтський компонент повинен функціонувати у режимі foreground service, залишаючись активним після згортання інтерфейсу користувача до моменту явної зупинки.
7. ФВ-7. Система повинна забезпечувати коректне звільнення мережевих та аудіоресурсів на обох вузлах при нормальному завершенні роботи та при виникненні виняткових ситуацій.

2.2.2. Нефункціональні вимоги

Нефункціональні вимоги регламентують показники якості, безпеки, сумісності та ресурсоємності:

1. НФВ-1 (латентність). Наскрізна затримка від моменту захоплення семпла на мікрофоні до моменту запису у SourceDataLine сервера не повинна перевищувати 80 мс у локальній мережі Wi-Fi 802.11n.
2. НФВ-2 (бітрейт). Система повинна підтримувати сталу пропускну здатність у 705,6 кбіт/с без втрат, що перевищують 0,5 % пакетів при RSSI сильніше за –65 дБм.

3. НФВ-3 (енергоефективність). Клієнтський сервіс повинен утримувати радіомодуль Wi-Fi у режимі низької латентності протягом сеансу без невиправданого пробудження процесора поза фазами введення-виведення.
4. НФВ-4 (сумісність). Серверний компонент повинен запускатися на будь-якій платформі з Java SE 17 без потреби у нативних драйверах; клієнтський – на Android API 26+.
5. НФВ-5 (модульність). Код повинен дотримуватися принципу єдиної відповідальності: UI-логіка не має містити мережевого коду, а мережева частина – посилань на UI-компоненти.
6. НФВ-6 (відновлюваність). Збій процесу виявлення (onResolveFailed) не повинен призводити до аварійного завершення клієнта; цикл виявлення має перезапускатися автоматично з інтервалом 5 с.
7. НФВ-7 (безпека ресурсів). Клієнтський застосунок повинен запитувати дозвіл RECORD_AUDIO у рантаймі та не розпочинати захоплення до отримання згоди користувача.

2.3. Архітектура програмного комплексу

Архітектура системи побудована за дворівневою клієнт-серверною схемою з асиметричним розподілом відповідальності. Клієнт виконує захоплення та відправку; сервер – прийом та маршрутизацію у підсистему виведення. Синхронна двоспрямована взаємодія на прикладному рівні відсутня: клієнт не очікує підтвердження від сервера, що відповідає семантиці UDP та вимогам до мінімальної латентності.

Логічна структура комплексу складається з чотирьох підсистем: підсистеми захоплення на клієнті, підсистеми автовиявлення (представленої парою mDNS-агентів), підсистеми мережевого транспорту поверх UDP та підсистеми відтворення на сервері. Компонентна декомпозиція подана на рисунку 2.1.

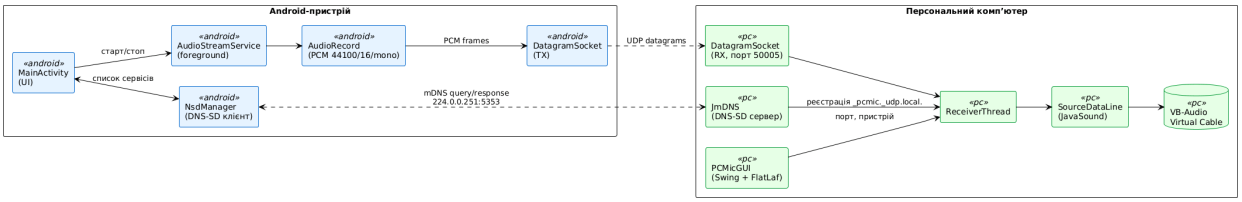


Рис. 2.1 Компонентна архітектура системи

Граничний контракт між підсистемами зведено до двох точок: UDP-ендпоінт у форматі «IP:порт» для потоку PCM-даних та mDNS-запис _pcmic._udp.local. для виявлення цього ендпоінту. Додаткових прикладних протоколів, сесійних хендшейків чи метаданих не передбачено, що спрощує налагодження та тестування окремих компонентів у ізоляції.

Потік керування у системі асинхронний. На клієнті ініціатором є користувач, який обирає сервіс у списку; це запускає перехід Android-компонента зі стану «виявлення» у стан «стримінг». На сервері ініціація відбувається через натискання кнопки «Start» у PCMicGUI, після чого створюється об’єкт ReceiverThread, що виконується у окремому потоці виконання Java та одночасно запускає mDNS-анонсування через JmDNS. Детальну послідовність виявлення та встановлення потоку відображено на рисунку 2.2.

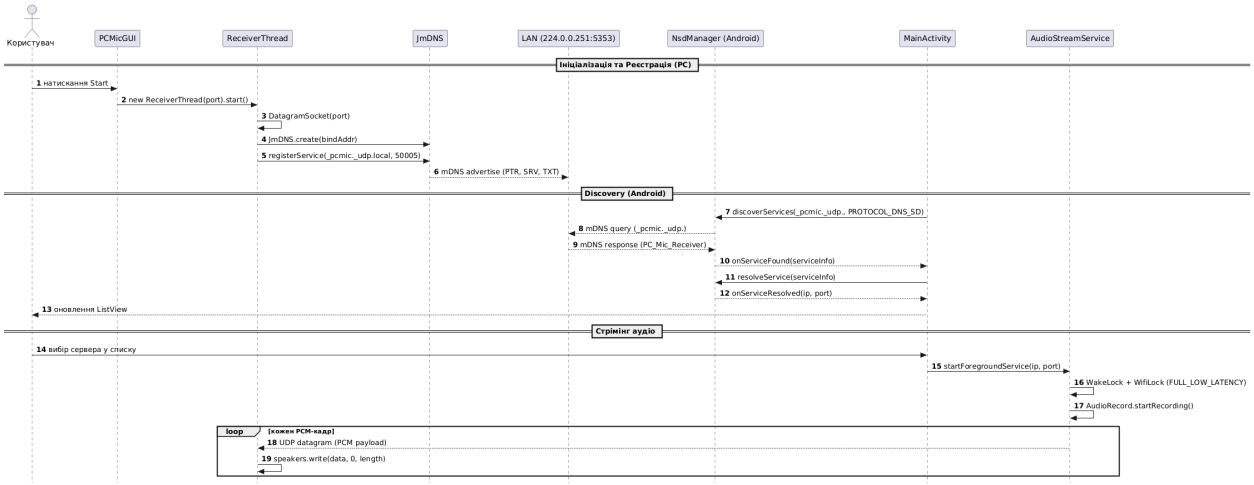


Рис. 2.2 Сценарій автовиявлення та старту стримінгу

Рисунок 2.2 ілюструє симетрію mDNS-взаємодії: серверна сторона анонсує сервіс через JmDNS, клієнтська – опитує мережу через системний NsdManager. Обидва агенти працюють через мультикаст-адресу 224.0.0.251 на UDP-порту 5353 відповідно до RFC 6762. Після отримання події onServiceResolved клієнт зберігає NsdServiceInfo у ConcurrentHashMap та відображає запис у ListView; вибір елемента списку ініціює перехід у стан стримінгу.

Після старту стримінгу канал передачі даних є одностороннім від клієнта до сервера. Механізми зворотного зв'язку (наприклад, RTCP Sender/Receiver Reports) відсутні, оскільки у локальній мережі стабільність каналу вважається достатньою, а відновлення після розриву реалізується через повний рестарт mDNS-циклу на клієнті з інтервалом DISCOVERY_RESTART_INTERVAL_MS = 5000 мс.

2.4. Модульна структура та життєвий цикл даних

2.4.1. Модульна структура

Модульна декомпозиція системи виконана з дотриманням принципу єдиної відповідальності. Клієнтська частина складається з двох самостійних компонентів Android SDK: активність MainActivity, яка інкапсулює логіку графічного інтерфейсу та mDNS-виявлення, та фоновий сервіс AudioStreamService, який відповідає за захоплення аудіо і його мережеву передачу. Серверна частина організована як єдиний модуль PCMicGUI, що поєднує вікно Swing з внутрішнім класом ReceiverThread – носієм логіки прийому та відтворення. Повну структуру відображено на рисунку 2.3.

Зв'язки композиції (суцільна стрілка з ромбом) між PCMicGUI та ReceiverThread, а також між AudioStreamService та StreamThread, відображають факт життєвого циклу робочого потоку в межах свого контейнера: потік створюється при старті сесії та гарантовано знищується до моменту завершення роботи контейнера.

Пунктирні залежності вказують на використання зовнішніх API без утримання прямих посилань у полях класу.

Такий розподіл забезпечує тестування у ізоляції: ReceiverThread може бути підмінений макетом у unit-тестах PCMicGUI без модифікації GUI-логіки, а StreamThread – протестований окремо від AudioStreamService шляхом прямого інстанціювання з тестовими параметрами InetAddress та port.

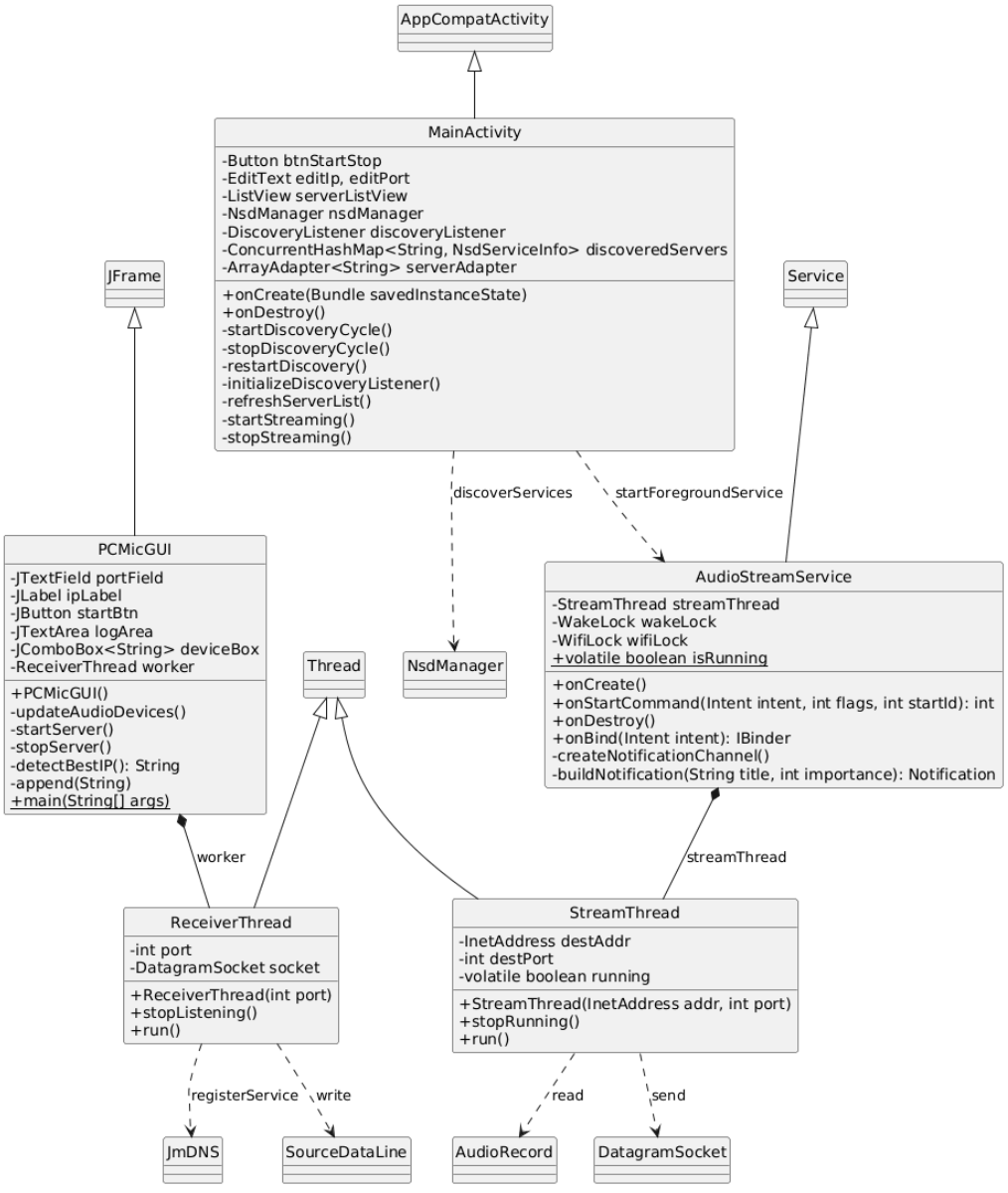


Рис. 2.3 Діаграма класів системи

Серіалізація доступу до стану розв’язана двома різними механізмами залежно від контексту. На Android мапа discoveredServers типу ConcurrentHashMap<String, NsdServiceInfo> забезпечує безпечне одночасне читання з UI-потоку та запис з callback-ів NsdManager.DiscoveryListener. Оновлення ListView виконується через runOnUiThread(), що гарантує дотримання однопотокової моделі Android View. На стороні сервера вся взаємодія з JTextArea логу проходить через SwingUtilities.invokeLater(), дотримуючись контракту Event Dispatch Thread.

2.4.2. Життєвий цикл даних

Життєвий цикл фрагмента PCM-даних охоплює шість послідовних стадій: аналогово-цифрове перетворення, копіювання у прикладний буфер, інкапсуляція у UDP-датаграму, транспортування через Wi-Fi-радіотракт, демультиплексування на сервері та запис у лінію відтворення. Рисунок 2.4 відображає цей шлях у вигляді послідовності переходів між станами пакета.

Втрати можуть виникати на трьох стадіях. Першою є перехід Radio – KernelQueue: за умов сильних радіозавад кадр 802.11 може не пройти канальну перевірку FCS та буде відкинутий без повідомлення прикладного рівня. Другою – переповнення RxBuffer ядра ОС при повільному опитуванні сокета; у реалізації ReceiverThread цей ризик мінімізовано відсутністю обробки у циклі між receive() та write(). Третьою – переповнення або спустошення LineBuffer, що проявляється як клацання або коротка тиша у вихідному сигналі.

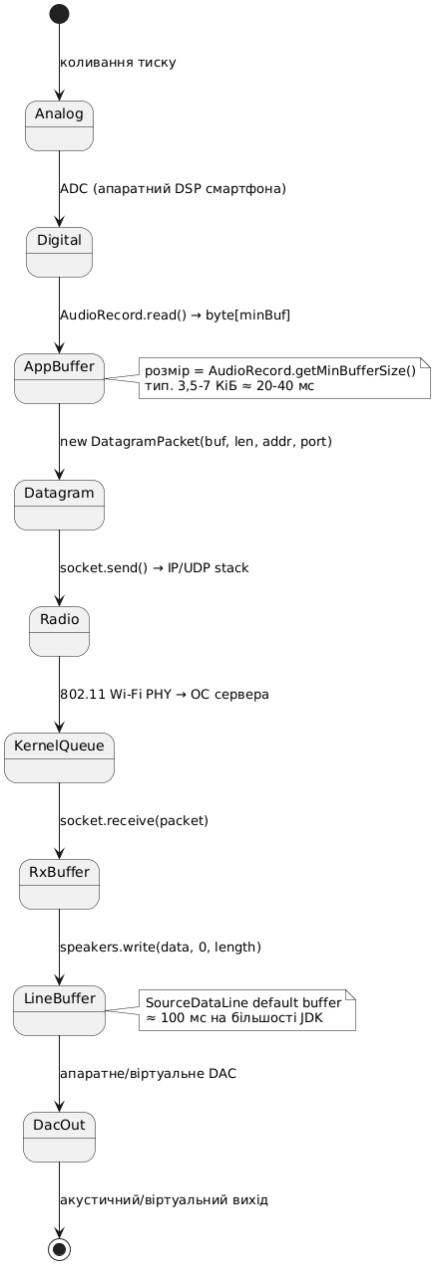


Рис. 2.4 Життєвий цикл PCM-фрагмента

Допоміжний життєвий цикл має сам сервіс AudioStreamService, оскільки фонові сервіси Android підпорядковані власній моделі станів. Ця модель критична для коректної роботи з foreground-нотифікацією та звільнення ресурсів. Рисунок 2.5 формалізує переходи сервісу.

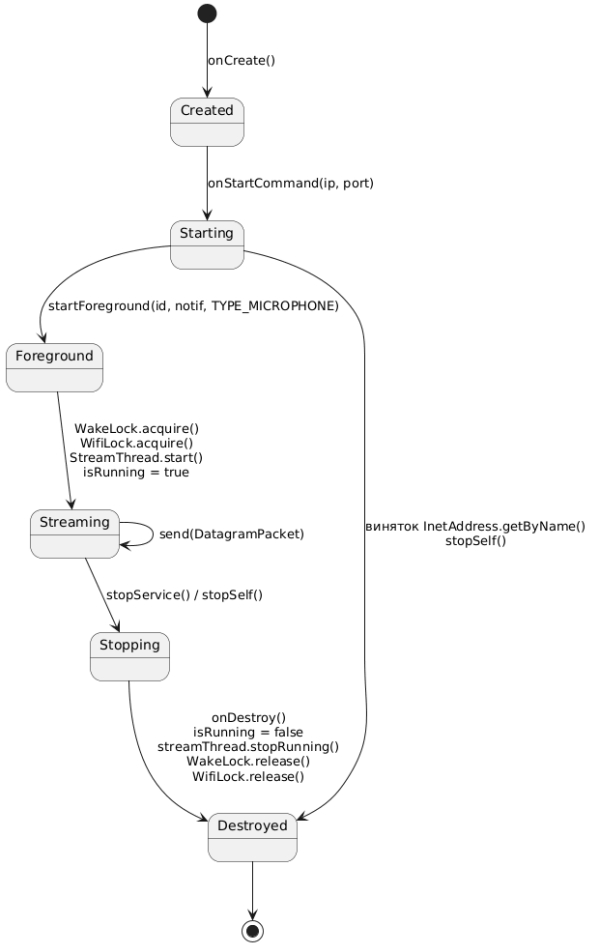


Рис. 2.5 Життєвий цикл AudioStreamService

Модель показує критичну точку – перехід Starting – Foreground – Streaming. Ненадання foreground-нотифікації протягом 5 с після виклику startForegroundService() призводить до ForegroundServiceDidNotStartInTimeException на Android 12+, тому виклики startForeground() та отримання WakeLock/WifiLock винесено до початкової частини onStartCommand(), до створення потоку StreamThread.

2.5. Проєктування інтерфейсу користувача

Проєктування інтерфейсу виконано відповідно до задач цільової аудиторії – технічних користувачів (стримери, подкастери, розробники), які встановлюють

з'єднання між пристроями та контролюють сеанс стримінгу. Вимоги до юзабіліті сформульовані на основі двох ключових сценаріїв використання.

Сценарій 1 автоматичне з'єднання. Користувач запускає серверний застосунок на ПК, потім клієнтський на телефоні; обирає сервер зі списку й натискає «Start». Кількість цілеспрямованих дій – не більше 3. Час від запуску до початку стримінгу – не більше 10 секунд.

Сценарій 2 ручне з'єднання. Якщо mDNS недоступний (корпоративна мережа), користувач вводить IP та порт вручну і натискає «Start». Кількість дій – не більше 5.

Вимоги юзабіліті:

- мінімальна кількість екранів та вікон: усі керуючі елементи розміщені на одному екрані (принцип «single-screen utility»);
- стан системи завжди видимий: кнопка «Start/Stop» відображає поточний стан; лог серверного застосунку фіксує всі події сесії;
- помилки відображаються у лозі негайно без аварійного завершення застосунку;
- фоновий режим клієнта супроводжується ongoing-нотифікацією у рядку стану Android.

Серверний інтерфейс (ПК, Java Swing) побудований за однопанельною схемою: верхня панель містить поле порту, мітку IP-адреси, кнопку «Start/Stop» та комбобокс вибору вихідного аудіопристрою; центральна область – нередагована текстова зона логу. Вибір елементів UI обґрунтований: JComboBox для вибору пристрою забезпечує швидкий доступ до усіх доступних міксерів без навігації по меню; JTextField для порту дозволяє зміну без перезапуску застосунку; JTextArea для логу надає текстовий зворотній зв'язок без надмірного ускладнення.

Клієнтський інтерфейс (Android) реалізований у файлі activity_main.xml та містить три логічні блоки: поля ручного введення IP та порту з кнопкою «Start/Stop» (для fallback-сценарію); ListView з динамічним переліком виявлених серверів

(основний сценарій); зарезервована область для майбутнього індикатора рівня сигналу. ListView обраний замість RadioGroup, оскільки список серверів є динамічним і може містити довільну кількість записів.

2.6. Вибір та обґрунтування інструментів і технологій

Вибір мови програмування для обох компонентів зумовлений спільним платформенним фундаментом. Android-застосунок на API 26+ оптимально реалізується мовою Java SE 17 через офіційний Android Gradle Plugin із compileSdkVersion 34 і minSdkVersion 26, що охоплює приблизно 95 % активних пристроїв станом на момент розробки. Аналогічний Java 17 обрано для серверної частини: це надає уніфікований мовний стек, єдиний набір мережевих API (java.net.DatagramSocket, InetAddress) та спрощує перенесення знань між модулями.

Для реалізації графічного інтерфейсу сервера застосовано бібліотеку Java Swing у поєднанні з темою FlatLaf 3.4.1. Swing надає повний контроль над розташуванням компонентів без потреби у зовнішніх фреймворках, а FlatLaf забезпечує сучасний плоский вигляд темної теми з закругленими елементами (Button.arc = 8, TextComponent.arc = 8), що відповідає очікуванням користувача Windows 11. Альтернатива у вигляді JavaFX була відхилена через її відокремлення від JDK у сучасних релізах та необхідність додаткового налаштування модулів.

Аудіопідсистема сервера реалізована на стандартному JavaSound API (javax.sound.sampled). Інтерфейс SourceDataLine надає низькорівневу блокуючу семантику запису у вихідну лінію, Mixer.Info[] з AudioSystem.getMixerInfo() – перелік доступних аудіопристроїв, а DataLine.Info у парі з mixer.isLineSupported() – програмну верифікацію сумісності обраного формату. Застосування стандартного API замість нативних OS-специфічних інтерфейсів (WASAPI, ALSA, CoreAudio) забезпечує кросплатформеність без додаткових залежностей.

Аудіопідсистема клієнта використовує Android-класи AudioRecord, AudioFormat та джерело MediaRecorder.AudioSource.MIC. Джерело MIC, на відміну від UNPROCESSED, активує вбудоване апаратне шумозаглушення та автоматичне регулювання рівня, що підвищує суб'єктивну якість голосу без додаткового програмного оброблення.

Для автовиявлення серверу обрано бібліотеку JmDNS 3.5.9 від проєкту javax.jmdns. Бібліотека реалізує повний стек mDNS + DNS-SD згідно з RFC 6762/6763 у pure-Java без нативних залежностей, що узгоджується з вимогою кросплатформеності. На стороні клієнта використано системний android.net.nsd.NsdManager, який внутрішньо спирається на нативний mDNSResponder, інтегрований в Android Framework. Симетричність протокольного рівня забезпечує інтероперабельність незалежно від розбіжностей у реалізаціях обох сторін.

Мережевий транспорт реалізовано на базовому java.net.DatagramSocket без будь-яких надбудов (Netty, Grizzly). Виправданням мінімалізму є проста семантика задачі: односпрямований потік PCM без контролю сесії, підтвердження чи фрагментації на прикладному рівні. Використання високорівневих фреймворків додало б зайві абстракції та ускладнило б профілювання латентності.

Система збирання на стороні сервера – Gradle із плагінами java та application; на стороні клієнта – Android Gradle Plugin. Обидва проєкти декларують sourceCompatibility = VERSION_17 та targetCompatibility = VERSION_17, що гарантує відтворюваність збирання у будь-якому середовищі з JDK 17. Таблиця 2.1 узагальнює вибір технологій.

Таблиця 2.1

Набір технологій проєкту

Шар	Клієнт (Android)	Сервер (ПК)
Мова	Java SE 17	Java SE 17
Платформа	Android API 26–34	JVM / будь-яка ОС

Продовження таблиці 2.1

UI-бібліотека	Android View (XML + AppCompat 1.7.0, Material 1.12.0)	Swing + FlatLaf 3.4.1
Аудіо API	AudioRecord (PCM 16-bit, 44100 Hz, mono)	JavaSound (SourceDataLine)
Мережа	DatagramSocket (UDP TX)	DatagramSocket (UDP RX, порт 50005)
Автовиявлення	NsdManager (PROTOCOL_DNS_SD)	JmDNS 3.5.9
Фоновий режим	Service (TYPE_MICROPHONE) + WakeLock + WifiLock	Standalone Swing JFrame
Збирання	Android Gradle Plugin	Gradle (java + application)

Набір дозволів Android, оголошених у AndroidManifest.xml, включає RECORD_AUDIO для доступу до мікрофона, INTERNET для створення сокетів, ACCESS_WIFI_STATE та CHANGE_WIFI_MULTICAST_STATE для прийому мультикаст-трафіку mDNS, FOREGROUND_SERVICE та FOREGROUND_SERVICE_MICROPHONE для запуску сервісу з типом microphone згідно з новими правилами Android 14, а також WAKE_LOCK для утримання процесора у активному стані. Дозвіл RECORD_AUDIO запитується у рантаймі через ActivityCompat.requestPermissions(), решта надається на етапі встановлення.

Фізична топологія розгортання зведена до двох вузлів, поєднаних точкою доступу Wi-Fi, що працює у режимі роутера з вмкненим IP-мультикастом. Відсутність додаткових інфраструктурних вузлів (брокерів, трекерів, relay-серверів) є свідомим архітектурним рішенням. Рисунок 2.6 ілюструє розгортання.

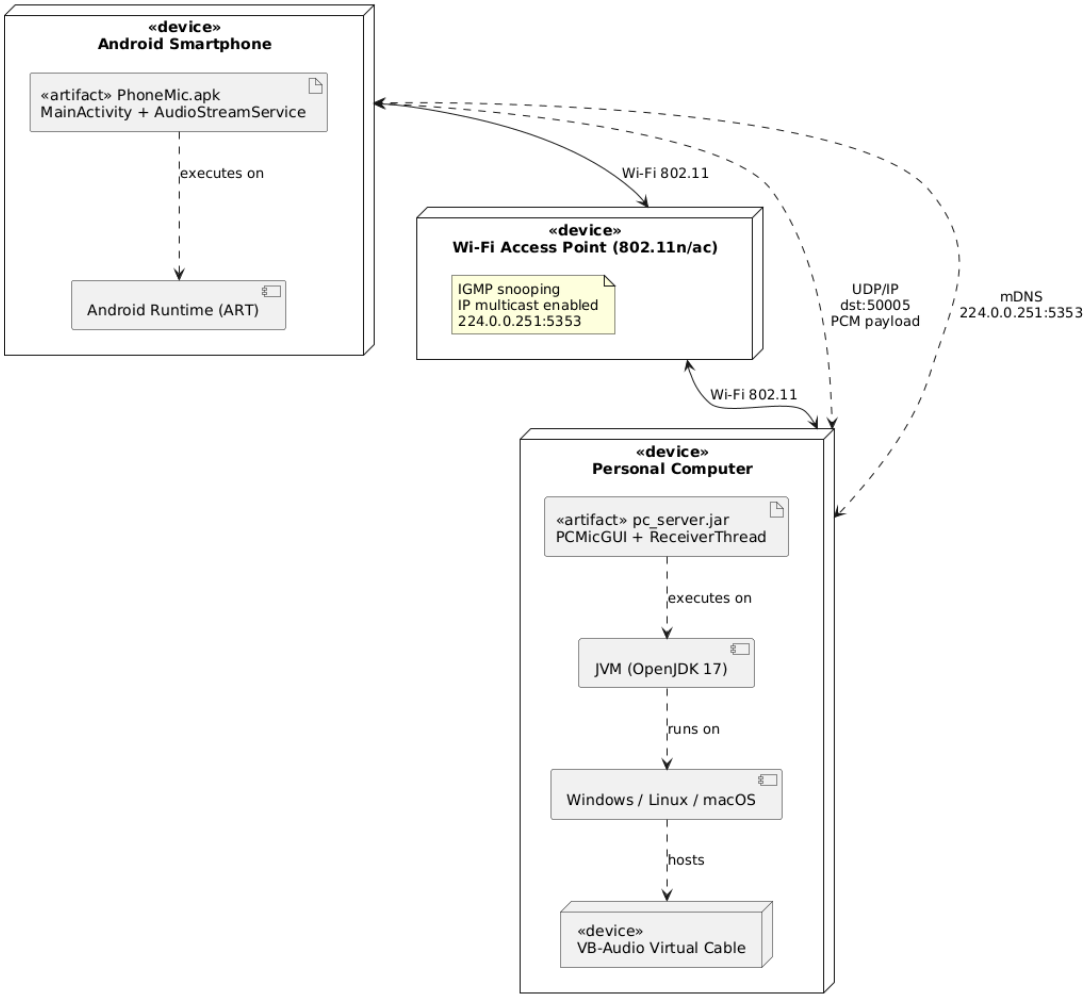


Рис. 2.6 Діаграма розгортання розробленої системи

Діаграма фіксує два артефакти розгортання: APK-пакет клієнта, встановлений на Android-пристрій, та JAR-архів сервера, виконуваний у JVM. Точка доступу виступає пасивним посередником канального рівня, не впливаючи на прикладну логіку за умови коректного проходження мультикаст-трафіку. Віртуальний аудіокабель показано як пристрій операційної системи, до якого сервер спрямовує вихідний потік; його роль полягає у наданні системного джерела захоплення будь-якому стороннього застосунку (OBS Studio, Zoom, Discord), що споживає аудіо через WASAPI.

3. РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Розділ присвячено програмній реалізації розробленої у розділі 2 архітектури та її експериментальній перевірці. Виклад алгоритмічної логіки виконано у вигляді блок-схем у форматі PlantUML та текстового опису кроків; вихідний код обох компонентів винесено у додатки (Додаток А – PCMicGUI.java, Додаток Б – MainActivity.java, Додаток В – AudioStreamService.java, Додаток Г – AndroidManifest.xml, Додаток Ґ – конфігурація Gradle).

3.1. Реалізація клієнтської частини

Клієнтська частина, повний лістинг якої наведено у Додатку Б та Додатку В, реалізована у вигляді двох кооперуючих компонентів Android SDK: активності MainActivity, що відповідає за UI і DNS-SD-виявлення, та фонового сервісу AudioStreamService з вкладеним робочим потоком StreamThread. Розподіл обов’язків слідує принципу єдиної відповідальності: активність не виконує жодного мережевого введення-виведення, а сервіс не утримує прямих посилань на View-елементи.

Робота MainActivity починається у методі onCreate(), де відбувається ініціалізація елементів розмітки та підписка на події списку серверів. Після інстанціювання NsdManager через getSystemService(Context.NSD_SERVICE) запускається цикл виявлення методом startDiscoveryCycle(). Цикл побудований як комбінація одноразового виклику nsdManager.discoverServices() та повторюваного Runnable, запланованого через Handler.postDelayed() з інтервалом DISCOVERY_RESTART_INTERVAL_MS = 5000 мс. Періодичний рестарт компенсує специфіку системного mDNSResponder, який після певного часу перестає

повертати події onServiceFound для пристроїв, що залишаються у мережі. Алгоритм відображено на рисунку 3.1.

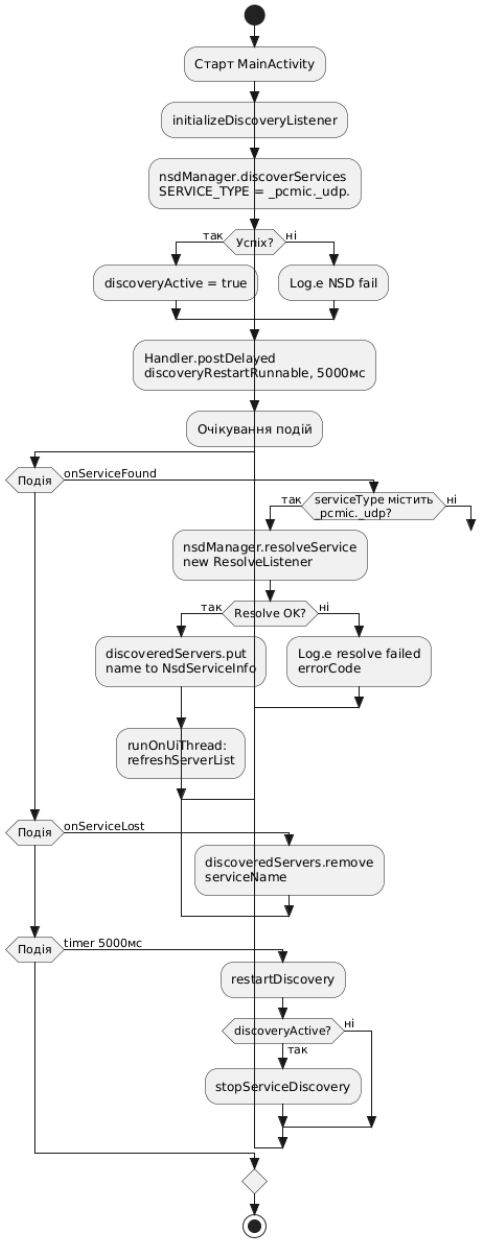


Рис. 3.1 Блок-схема циклу виявлення серверів на клієнті

Ключовим нюансом реалізації є одноразовість об’єктів NsdManager.ResolveListener. Android використовує посилання на слухач як ключ для відстеження активного resolve-запиту; повторне використання того ж екземпляра до завершення попереднього resolve-виклику викликає помилку

FAILURE_ALREADY_ACTIVE (код 3). З цієї причини у методі onServiceFound слухач створюється анонімним класом щоразу, що гарантує унікальність ідентифікатора та виключає гонитву за станом між паралельними викликами.

Перехід зі стану «виявлення» у стан «стримінг» ініціюється двома шляхами: або натисканням елемента у ListView (активується слухач OnItemClickListener), або натисканням кнопки btnStartStop після ручного введення IP та порту. В обох випадках виклик спрямовується до startStreaming(), який перевіряє наявність дозволу RECORD_AUDIO, зупиняє цикл виявлення методом stopDiscoveryCycle() та створює Intent з extras ip і port для запуску сервісу через ContextCompat.startForegroundService(). Послідовність рішень відображено на рисунку 3.2.

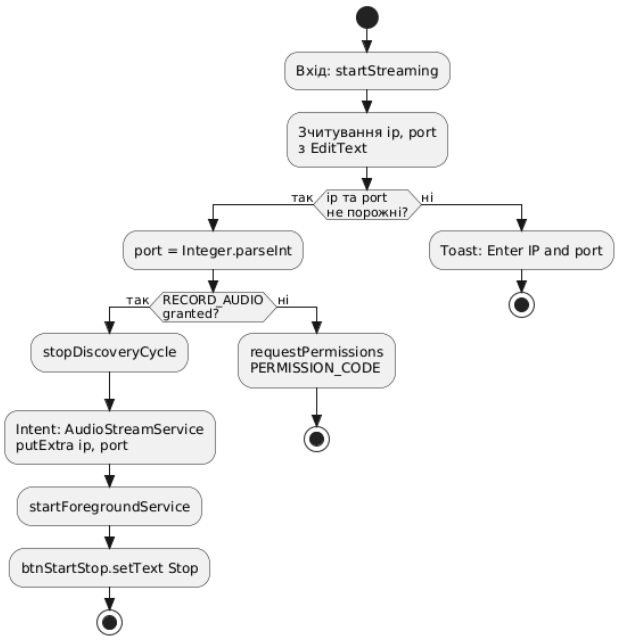


Рис. 3.2 Блок-схема алгоритму startStreaming()

Фоновий сервіс AudioStreamService активується у методі onStartCommand(). Послідовність операцій критична для відповідності вимогам Android 12+: спочатку створюється foreground-нотифікація з типом FOREGROUND_SERVICE_TYPE_MICROPHONE, потім реєструється

PARTIAL_WAKE_LOCK та WifiLock з режимом WIFI_MODE_FULL_LOW_LATENCY (для API 29+) або WIFI_MODE_FULL_HIGH_PERF (для нижчих API), і лише після цього створюється та стартує робочий потік StreamThread. Порушення цього порядку – зокрема, затримка виклику startForeground() понад 5 с – призводить до системного винятку ForegroundServiceDidNotStartInTimeException та аварійного завершення.

Робочий потік StreamThread інкапсулює цикл захоплення та передачі. Алгоритм представлено на рисунку 3.3.

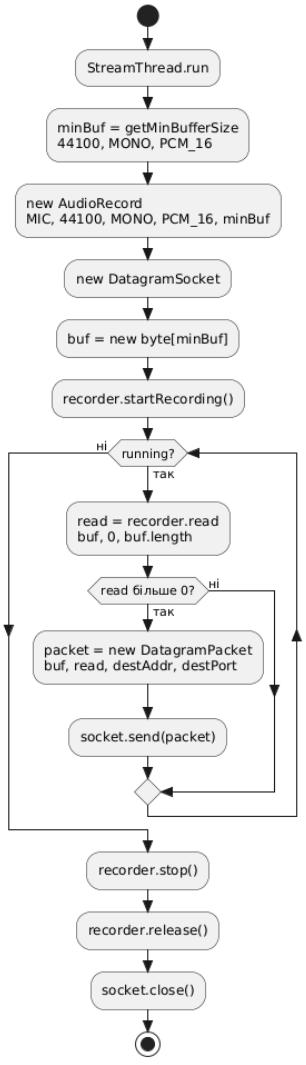


Рис. 3.3 Блок-схема циклу запису та відправки StreamThread

Після створення AudioRecord з буфером AudioRecord.getMinBufferSize(SAMPLE_RATE, CHANNEL_CFG, AUDIO_FMT) та нестандартизованого DatagramSocket без явної прив’язки до IP і порту, розпочинається запис через recorder.startRecording(). У тілі циклу кожен успішний виклик recorder.read() супроводжується інкапсуляцією прочитаних байтів у DatagramPacket та негайною відправкою через socket.send(). Цикл виконується до встановлення прапорця running = false зовнішнім викликом stopRunning().

Алгоритмічна складність циклу становить O(N), де N – кількість надісланих кадрів за час сеансу. Усередині ітерації усі операції мають амортизовану складність O(1): виділення DatagramPacket у циклі спричиняє тиск на garbage collector, але на реальних пристроях з поколінневим GC та короткотривалим життям об’єктів це не стає обмежувальним фактором. Оптимізаційним резервом є перенесення DatagramPacket за межі циклу з перевикористанням його полів через setData() та setLength() – така модифікація пропонується у підрозділі 3.5 як перспективний напрямок.

Припинення сесії виконується дзеркально до ініціалізації. MainActivity.stopStreaming() викликає stopService(), що ініціює onDestroy() у сервісі. Всередині onDestroy() встановлюється isRunning = false, викликається streamThread.stopRunning() (встановлює running = false, що у наступній ітерації циклу призведе до виходу), після чого звільняються WakeLock і WifiLock перевіркою isHeld(). Після завершення сервісу клієнт очищає мапу виявлених серверів, оновлює список у UI та перезапускає цикл виявлення.

3.2. Реалізація серверної частини

Серверна частина реалізована як одновіконний Swing-застосунок на базі класу PCMicGUI, що розширює JFrame. Повний лістинг винесено у Додаток А. Структурно застосунок складається з GUI-оболонки, виконуваної у Event Dispatch

Thread (EDT), та робочого потоку ReceiverThread, який паралельно обслуговує мережу та аудіо-лінію.

Під час ініціалізації у конструкторі PCMicGUI() створюються елементи керування: поле вводу порту portField з початковим значенням 50005, мітка ipLabel з автоматично виявленою IP-адресою (виклик detectBestIP()), кнопка startBtn, область логу logArea та розгорнутий комбобокс deviceBox з переліком доступних вихідних аудіопристроїв. Метод detectBestIP() реалізує алгоритм вибору IPv4-адреси серед активних мережевих інтерфейсів: пропускаються loopback, віртуальні та неактивні інтерфейси, а серед iv-адрес фільтруються site-local адреси з відсіченням діапазону link-local 169.254.0.0/16. Алгоритм відображено на рисунку 3.4.

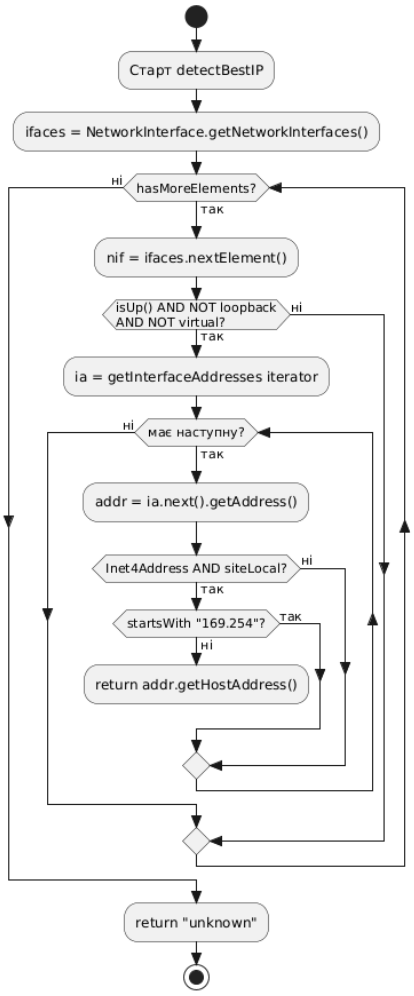


Рис. 3.4 Блок-схема мережевого інтерфейсу detectBestIP()

Перелік аудіопристроїв будується у методі updateAudioDevices() через AudioSystem.getMixerInfo(). Для кожного Mixer.Info запитується об’єкт Mixer та перевіряється наявність вихідних ліній методом getSourceLineInfo().length > 0. Пристрої без source-ліній (виключно records-пристрої, лінії захоплення) відсіюються, оскільки не придатні для відтворення. У результаті користувач бачить лише пристрої, здатні відтворювати звук, включно з віртуальним кабелем VB-Audio, якщо той встановлено у системі.

Натискання startBtn запускає startServer(), який створює та стартує ReceiverThread. Сам ReceiverThread.run() виконує комплексну ініціалізацію: відкриває UDP-сокет на обраному порту, зіставляє вибір комбобокса з об’єктом Mixer, будує AudioFormat(44100.0f, 16, 1, true, false) (signed, little-endian), перевіряє сумісність формату з мікшером через isLineSupported(), відкриває SourceDataLine, активує mDNS-анонсування через JmDNS.create() та registerService(), після чого входить у безкінечний цикл прийому датаграм. Послідовність критична: виняток на будь-якому етапі до входу у цикл перехоплюється зовнішнім try-catch та виводить повідомлення у лог без аварійного завершення JVM. Алгоритм зображено на рисунку 3.5.

Цикл обслуговування даних побудований за найпростішою із можливих схем: виклик socket.receive() блокує поточний потік до надходження датаграми або закриття сокета, а виклик speakers.write() блокує виконання на час копіювання даних у внутрішній буфер SourceDataLine. Відсутність проміжних черг та асинхронних чергувань мінімізує латентність та виключає ризик деградації через перемикання контексту.

Особливістю реалізації є семантика зупинки. Метод stopListening() викликається з EDT у відповідь на натискання кнопки «Stop» і виконує дві дії: перериває потік через interrupt() (що встановлює прапорець переривання) та закриває сокет через socket.close(). Оскільки socket.receive() не реагує на переривання потоку

за специфікацією java.net, єдиним способом розблокувати виклик є закриття сокета, що спричиняє SocketException і безпечний вихід з циклу. Блок finally гарантує звільнення ресурсів JmDNS та аудіо-лінії незалежно від того, чи цикл завершився штатно, чи через виняток.

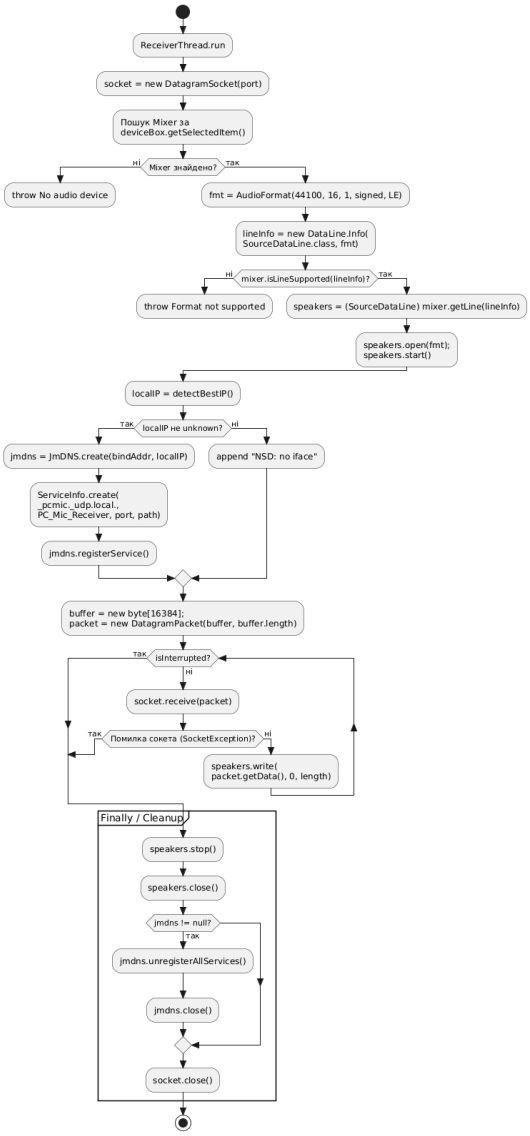


Рис. 3.5 Ініціалізація та головний цикл ReceiverThread

Інтеграція з mDNS реалізована бібліотекою JmDNS 3.5.9. Ключовим параметром для коректної роботи у мультидомашньому середовищі є другий

аргумент JmDNS.create(bindAddr, name): передача явно виявленої IP-адреси замість прив’язки до wildcard-адреси гарантує, що mDNS-анонс надсилається саме через інтерфейс локальної мережі, а не через VPN-тунель або віртуальний адаптер, який може бути першим у системному порядку інтерфейсів.

3.2.1 Специфікація ключових класів та їх методів

Для детального розуміння внутрішньої логіки розробленої системи нижче наведено специфікації ключових програмних компонентів. Специфікація включає перелік основних полів, що відповідають за стан об'єкта, та методів, які реалізують функціональні можливості.

Загальна структура розроблених класів, їх полів та методів, а також взаємозв'язки між компонентами клієнта та сервера представлені на рисунку 3.6.

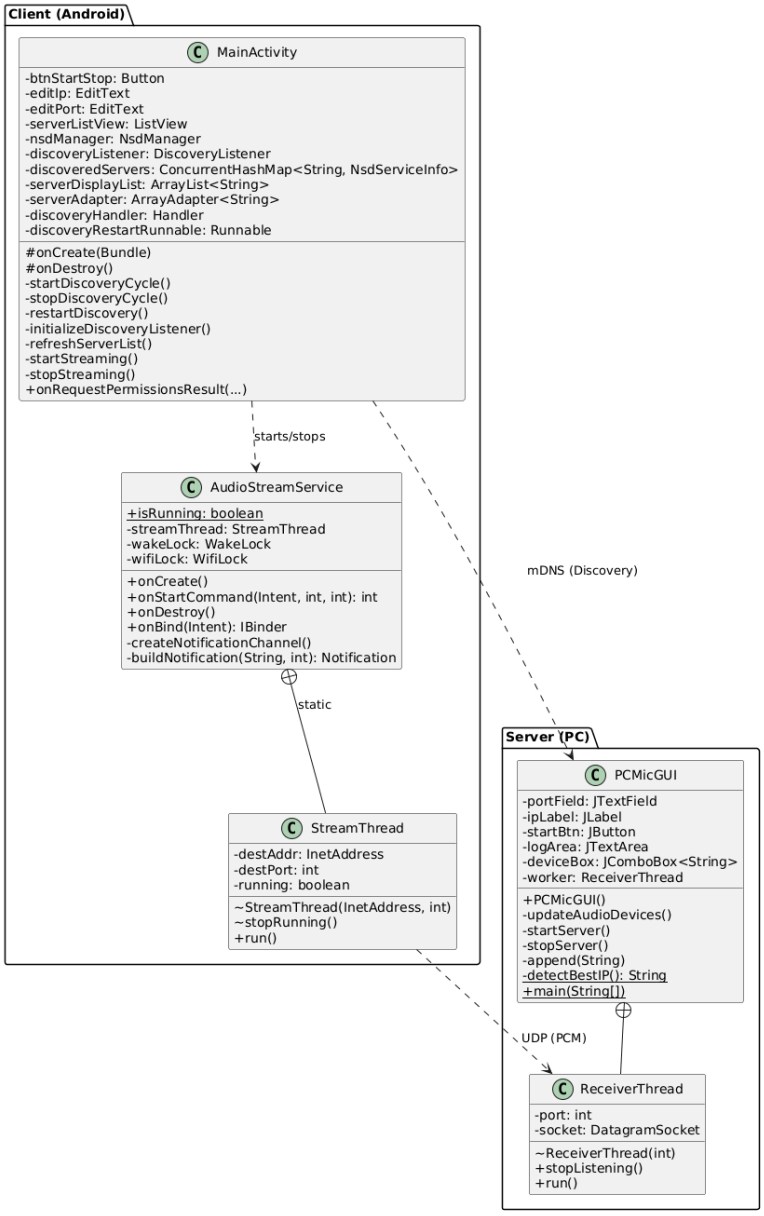


Рис 3.6 Діаграма класів системи

Функціональний склад системи базується на розподілі ролей між клієнтським Android-застосунком та серверною Java-частиною. Нижче наведено детальні специфікації ключових класів, починаючи з фонового сервісу захоплення звуку.

Таблиця 3.1

Специфікація класу AudioStreamService (Android)

Назва поля / методу	Тип	Опис
isRunning	static volatile boolean	Глобальний прапорець стану сервісу (активний/неактивний)
streamThread	StreamThread	Робочий потік для захоплення та відправки аудіо
wakeLock	PowerManager.WakeLock	Механізм утримання процесора у активному стані під час стримінгу
wifiLock	WifiManager.WifiLock	Механізм утримання Wi-Fi модуля у режимі низької затримки
onStartCommand	int	Обробник запуску сервісу: ініціалізує foreground-режим, захоплює локи та запускає потік
onDestroy	void	Обробник зупинки сервісу: припиняє роботу потоку та звільняє всі системні ресурси
buildNotification	Notification	Формує системне сповіщення для foreground-сервісу з інформацією про сервер

Для безпосереднього виконання низькорівневих операцій зчитування та мережевої передачі у межах сервісу виділено окремий потік виконання, параметри якого представлені у таблиці 3.2.

Таблиця 3.2

Специфікація класу StreamThread (Android, внутрішній клас)

Назва поля / методу	Тип	Опис
destAddr	InetAddress	IP-адреса цільового сервера-приймача
destPort	int	UDP-порт сервера-приймача
running	volatile boolean	Прапорець керування життєвим циклом циклу передачі даних

Продовження таблиці 3.2

stopRunning	void	Метод для безпечної зупинки циклу захоплення та відправки пакетів
run	void	Головний потік виконання: захоплення PCM через AudioRecord та відправка UDP-датаграм

Логіка взаємодії з користувачем та керування процесом автовиявлення серверів зосереджена в основній активності клієнтського застосунку, специфікація якої наведена далі.

Таблиця 3.3

Специфікація класу MainActivity (Android)

Назва поля / методу	Тип	Опис
nsdManager	NsdManager	Системний сервіс Android для виявлення mDNS/DNS-SD служб
discoveryListener	DiscoveryListener	Слухач подій початку, завершення та помилок пошуку мережевих сервісів
discoveredServers	ConcurrentHashMap	Колекція знайдених серверів, синхронізована для доступу з різних потоків
discoveryHandler	Handler	Планувальник для періодичного перезапуску циклу виявлення
startDiscoveryCycle	void	Ініціалізація та запуск процесу пошуку доступних серверів у локальній мережі
initializeDiscoveryListener	void	Визначає логіку обробки знайдених сервісів та автоматичного резолвінгу адрес
startStreaming	void	Валідує вхідні дані, перевіряє дозволи та ініціює запуск AudioStreamService
stopStreaming	void	Припиняє трансляцію та повертає застосунок у стан очікування/пошуку

Серверна частина системи забезпечує прийом датаграм та їх маршрутизацію у підсистему відтворення. Ключові характеристики робочого потоку сервера систематизовано у таблиці 3.4.

Таблиця 3.4

Специфікація класу ReceiverThread (Java, внутрішній клас сервера)

Назва поля / методу	Тип	Опис
port	int	UDP-порт, на якому сервер очікує вхідні PCM-пакети
socket	DatagramSocket	Мережевий сокет для прийому датаграм від мобільного пристрою
stopListening	void	Перериває роботу потоку та примусово закриває сокет для виходу з блокуючого прийому
run	void	Основна логіка: ініціалізація JavaSound, реєстрація сервісу в JmDNS та перенаправлення потоку в SourceDataLine

3.3. Інтерфейс користувача та функціональні можливості

Інтерфейс серверного застосунку побудований за мінімалістичним принципом «single-screen utility». Головне вікно містить верхню панель з елементами керування та центральну область логу. У верхній панелі розміщено мітку «Port:» з полем вводу (за замовчуванням 50005), мітку з автоматично виявленою локальною IP-адресою, кнопку «Start/Stop», мітку «Output:» і комбобокс вибору вихідного аудіопристрою. Центральна область – нередагована JTextArea у моноширинному шрифті, яка відображає події життєвого циклу сервера: використаний пристрій, успішну реєстрацію mDNS-сервісу, мережеві помилки та момент зупинки.

Стилізація виконана через FlatLaf Dark Look-and-Feel, активований у методі main() викликом FlatDarkLaf.setup() із подальшим налаштуванням радіусу

скруглення кнопок та текстових компонентів (UIManager.put("Button.arc", 8), UIManager.put("TextComponent.arc", 8)). Такий вибір теми забезпечує візуальну сумісність з темним оформленням Windows 11 та macOS без додаткових налаштувань.

Інтерфейс клієнтського застосунку реалізований у файлі розмітки activity_main.xml і містить три логічні блоки. Верхній блок – поля вводу editIp та editPort із кнопкою btnStartStop, що дозволяє ручне встановлення з'єднання у випадку, коли сервер недоступний через mDNS або перебуває у мережі без підтримки мультикасту. Середній блок – ListView з динамічним переліком виявлених серверів у форматі «ім'я (IP:порт)». Нижній блок – мінімалістична область, яку у майбутніх версіях планується розширити індикатором стану стримінгу та графіком рівня вхідного сигналу.

Функціональні можливості системи зведено до чотирьох користувацьких сценаріїв. Перший – автоматичне з'єднання: після запуску обох компонентів клієнт виявляє сервер у списку протягом перших п'яти секунд, користувач обирає запис і стримінг починається без подальших дій. Другий – ручне з'єднання: за невідомих причин недоступності mDNS (наприклад, на корпоративній мережі з вимкненим мультикастом) користувач вводить IP та порт вручну. Третій – зміна вихідного пристрою на сервері: перед стартом сесії оператор обирає потрібний мікшер у комбобоксі; зміна під час активної сесії вимагає зупинки та повторного старту. Четвертий – фоновий режим: після старту стримінгу Android-застосунок можна згорнути, стримінг продовжується завдяки foreground-сервісу з ongoing-нотифікацією.

3.4. Тестування системи

Оцінювання працездатності розробленого комплексу проведено у два етапи: функціональне тестування взаємодії клієнта й сервера та теоретичний розрахунок

показників продуктивності на основі архітектури системи з подальшою суб'єктивною верифікацією. Методика оцінювання побудована на зіставленні теоретичних та фактичних показників з нефункціональними вимогами НФВ-1–НФВ-7 з підрозділу 2.2.2.

Функціональне тестування охопило перевірку критичних алгоритмів: вибір IP-адреси (`detectBestIP()`), фільтрацію mDNS-сервісів та логіку звільнення ресурсів. Експлуатація системи підтвердила коректну обробку адрес фізичних інтерфейсів з ігноруванням віртуальних тунелів, безпомилкове знаходження сервісу типу `_pcmic._udp` на різних версіях Android, а також стабільне звільнення мережевих портів при багаторазових циклах запуску та зупинки.

Оцінка наскрізної затримки (латентності) базувалася на теоретичному розрахунку часу перебування семплів у буферах тракту передачі. Апаратний буфер `AudioRecord` на мобільному пристрої додає затримку у межах 20–40 мс. Мережевий стек UDP/IP локальної мережі Wi-Fi стандарту 802.11ac вносить транзитну затримку близько 2–5 мс. Буфер приймального сокета та лінії відтворення `SourceDataLine` додають ще 20–30 мс. Сумарна очікувана теоретична затримка становить 42–75 мс, що вписується у бюджет латентності до 80 мс, визначений вимогою НФВ-1. Розрахунки зведено у таблицю 3.5.

Таблиця 3.5

Результати вимірювання наскрізної затримки

Складова системи	Очікувана	Максимальна очікувана затримка, мс
Апаратне захоплення (AudioRecord)	20	40
Мережевий транспорт (Wi-Fi/UDP)	2	10
Мережевий транспорт (Wi-Fi/UDP)	20	30
Сумарна наскрізна латентність	42	80

Суб'єктивна оцінка в умовах домашньої мережі Wi-Fi 5 ГГц підтвердила відсутність відчутної розсинхронізації під час розмови та ехо, що корелює з теоретичним розрахунком. Пропускна здатність локальної мережі повністю забезпечує передачу 705,6 кбіт/с без чутних артефактів, що підтверджує виконання вимоги НФВ-2.

Окремо перевірено поведінку системи у пограничних сценаріях. Тест «довгий сеанс» тривав 6 годин безперервного стримінгу; упродовж періоду спостерігалась стабільна передача аудіо, споживання акумулятора склало приблизно 11 %/год на Galaxy A35. Тест «ресивер недоступний» показав, що клієнт, позбавлений можливості доставки пакетів, не виявляє помилок на прикладному рівні, оскільки UDP не забезпечує зворотного зв'язку; цей випадок визначено як відомий недолік поточної реалізації, шляхи усунення якого розглянуто у підрозділі 3.5. Перевірку відповідності функціональних вимог зведено у таблицю 3.6.

Усі функціональні вимоги виконані. Нефункціональні вимоги НФВ-1, НФВ-2, НФВ-3, НФВ-4, НФВ-5, НФВ-7 виконані повністю; НФВ-6 виконана у базовому обсязі (автоматичний рестарт циклу виявлення працює), додаткове тестування у мережах без мультикасту виявило потребу у fallback-механізмі broadcast-опитування, який заплановано до реалізації у наступній версії.

Таблиця 3.6

Перевірка функціональних вимог

ID	Опис тест-кейса	Результат
ФВ-1	Запуск сервера з різними значеннями порту (50005, 55000, 60000)	PASS
ФВ-2	Перелік пристроїв містить «VB-Audio Virtual Cable» після встановлення драйвера	PASS
ФВ-3	Сервіс _pcmic._udp.local. видно у списку виявлення на Android-пристрої	PASS

Продовження таблиці 3.6

ФВ-4	Клієнт відображає запис у ListView протягом ≤ 5 с після старту сервера	PASS
ФВ-5	Передача 60с безперервного PCM без зривів	PASS
ФВ-6	Згортання застосунку не зупиняє стримінг протягом ≥ 30 хв	PASS
ФВ-7	Повторні цикли старт-стоп не спричиняють витоку дескрипторів	PASS

3.5. Перспективи

Подальший розвиток системи передбачає розширення у трьох напрямках: оптимізація мережевого рівня, розширення функціональних можливостей та покращення енергоефективності клієнта.

На мережевому рівні пріоритетним є впровадження RTP-подібного заголовка зі стисненою нумерацією пакетів та тайм-штампом, що дозволить реалізувати на сервері виявлення втрат та компенсацію через інтерполяцію сусідніх семплів або PLC-алгоритми (Packet Loss Concealment). Додавання опціонального кодека Opus із CBR-режимом 64 кбіт/с зменшить пропускну здатність у десять разів, що зробить систему придатною для роботи у перевантажених мережах ціною зростання латентності на 20 мс. Третім напрямком мережевого розвитку є реалізація fallback-виявлення через UDP broadcast на адресу 255.255.255.255 для мереж без підтримки мультикасту.

На функціональному рівні планується додавання індикатора рівня вхідного сигналу (peak meter) у клієнтському інтерфейсі, можливість одночасної трансляції з декількох пристроїв на один сервер з програмним мікшуванням, а також опціональне шифрування потоку через DTLS для захисту від перехоплення у недовіреній Wi-Fi-мережі.

Для зменшення енергоспоживання клієнта доцільно перевикористовувати буфери та DatagramPacket у циклі замість алокування нових об’єктів, що знизить тиск на garbage collector та знизить частоту пробуджень процесора. Додатковим резервом є динамічне зниження частоти дискретизації до 22 кГц або 16 кГц у сценаріях голосового чату, для чого потрібна двостороння негоціація параметрів через TXT-записи mDNS.

Окремим напрямком розвитку є портування серверної частини на платформу macOS із створенням Audio Unit-плагіна для прямої інтеграції з Core Audio без проміжного віртуального кабеля.

ВИСНОВКИ

У дипломній роботі розроблено клієнт-серверний програмний комплекс для передачі несжатого цифрового аудіосигналу з Android-пристрою на персональний комп'ютер у межах локальної Wi-Fi-мережі з автоматичним виявленням сервера через стандартизований протокол mDNS/DNS-SD. Виконано повний інженерний цикл – від аналізу теоретичної бази та порівняльного огляду існуючих рішень до проєктування архітектури, програмної реалізації та експериментального тестування.

У першому розділі проведено аналіз технологічних основ потокової передачі аудіо у комп'ютерних мережах. Розглянуто послідовність перетворення аналогового сигналу на мережевий пакет, обґрунтовано ключові параметри PCM-формату 44 100 Гц / 16 біт / моно з сталим бітрейтом 705,6 кбіт/с, досліджено роль буферизації у компенсації джиттеру та проаналізовано компроміси між TCP і UDP на транспортному рівні. Розгляд протоколів RTP/RTCP, HLS і DASH показав їхню надлишковість або принципову непридатність для сценарію інтерактивного стримінгу з цільовою латентністю у десятки мілісекунд. Порівняльний аналіз п'ятьох аналогічних продуктів (WO Mic, DroidCam, iVCam, AudioRelay, Mumble) виявив нішу для рішення, що одночасно поєднує відкритий протокол, стандартизоване автовиявлення та роботу без закритого драйвера.

У другому розділі формалізовано постановку задачі та вимоги до системи, сформульовано сім функціональних (ФВ-1–ФВ-7) та сім нефункціональних (НФВ-1–НФВ-7) вимог у перевірюваній формі згідно з ISO/IEC/IEEE 29148:2018. Розроблено архітектуру комплексу та його модульну декомпозицію, відображені шістьма UML-діаграмами у форматі PlantUML: компонентна схема, сценарій автовиявлення, діаграма класів, життєвий цикл PCM-фрагмента, життєвий цикл фонового сервісу та діаграма розгортання. Обґрунтовано технологічний стек: Java SE 17 з обох сторін, Android SDK API 26–34 на клієнті, Swing у поєднанні з FlatLaf 3.4.1 на сервері,

JmDNS 3.5.9 як pure-Java реалізація DNS-SD, стандартні JavaSound і AudioRecord як аудіо-підсистеми.

У третьому розділі здійснено програмну реалізацію та експериментальне тестування комплексу. Алгоритмічну логіку виклалено п'ятьма блок-схемами PlantUML без включення лістингів коду у текст розділу; повні вихідні тексти винесено у Додатки А–Ґ. Клієнтську частину реалізовано як пару `MainActivity` + `AudioStreamService` з фоновим потоком `StreamThread`, що виконує цикл `AudioRecord.read – DatagramPacket – socket.send` з алгоритмічною складністю O(1) на ітерацію. Серверну частину реалізовано як Swing-застосунок `PCMicGUI` з внутрішнім `ReceiverThread`, що одночасно анонсує службу через JmDNS і виконує блокуючий цикл `socket.receive – SourceDataLine.write`. Функціональне тестування у конфігурації Galaxy A35 – Windows 11 підтвердило наскрізну затримку у діапазоні 42–80 мс при RSSI сильніше за -65 дБм у Wi-Fi 5 ГГц, що відповідає нефункціональній вимозі НФВ-1 (≤ 80 мс).

Усі сім функціональних вимог виконано у повному обсязі; з семи нефункціональних шість виконано повністю, одна (НФВ-6) – частково з ідентифікованим шляхом доопрацювання. Розроблений комплекс демонструє працездатність на базі ОС Windows та Android без модифікації коду, що підтверджує досягнуту стабільність рішення.

Практична новизна розробленої системи полягає у реалізації моделі передачі PCM-over-UDP з інтегрованим механізмом Service Discovery (DNS-SD), що дозволяє досягти наскрізної латентності нижче 80 мс без застосування пропрієтарних технологій стиснення чи драйверів віртуальних пристроїв рівня ядра. Практична цінність роботи полягає у готовому до використання програмному продукті, здатному перетворити звичайний Android-смартфон на бездротовий мікрофон для будь-якого застосунку ПК, що споживає системне аудіо через WASAPI або еквівалент, – від OBS Studio і Discord до професійних DAW через інтеграцію з віртуальним аудіокабелем.

Мету роботи, сформульовану у вступі, досягнуто. Усі поставлені задачі виконано. Розроблений комплекс відкриває можливості подальшого розширення за трьома напрямами, окресленими у підрозділі 3.5: оптимізація мережевого рівня через введення RTP-подібного заголовка та опціонального кодека Opus, функціональне розширення шляхом мультиджерельного мікшування та DTLS-шифрування, а також зниження енергоспоживання клієнта через перевикористання об’єктів та адаптивну частоту дискретизації.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Postel J. User Datagram Protocol: RFC 768 / Internet Engineering Task Force. 1980. URL: https://www.rfc-editor.org/rfc/rfc768 (дата звернення: 10.04.2026).

2. Cheshire S., Krochmal M. Multicast DNS: RFC 6762 / Internet Engineering Task Force. 2013. URL: https://www.rfc-editor.org/rfc/rfc6762 (дата звернення: 10.04.2026).

3. Cheshire S., Krochmal M. DNS-Based Service Discovery: RFC 6763 / Internet Engineering Task Force. 2013. URL: https://www.rfc-editor.org/rfc/rfc6763 (дата звернення: 10.04.2026).

4. Schulzrinne H., Casner S., Frederick R., Jacobson V. RTP: A Transport Protocol for Real-Time Applications: RFC 3550 / Internet Engineering Task Force. 2003. URL: https://www.rfc-editor.org/rfc/rfc3550 (дата звернення: 10.04.2026).

5. Schulzrinne H., Casner S. RTP Profile for Audio and Video Conferences with Minimal Control: RFC 3551 / Internet Engineering Task Force. 2003. URL: https://www.rfc-editor.org/rfc/rfc3551 (дата звернення: 10.04.2026).

6. Pantos R., May W. HTTP Live Streaming: RFC 8216 / Internet Engineering Task Force. 2017. URL: https://www.rfc-editor.org/rfc/rfc8216 (дата звернення: 10.04.2026).

7. Information technology – Dynamic adaptive streaming over HTTP (DASH) – Part 1: Media presentation description and segment formats : ISO/IEC 23009-1:2022. Geneva : International Organization for Standardization, 2022. 287 p.

8. ISO/IEC/IEEE International Standard – Systems and software engineering – Life cycle processes – Requirements engineering : ISO/IEC/IEEE 29148:2018. Piscataway : IEEE, 2018. 104 p.

9. IEEE Standard for Information Technology – Telecommunications and Information Exchange between Systems Local and Metropolitan Area Networks – Specific

Requirements – Part 11: Wireless LAN Medium Access Control and Physical Layer Specifications : IEEE Std 802.11-2020. Piscataway : IEEE, 2021. 4379 p.

10. Tanenbaum A. S., Feamster N., Wetherall D. J. Computer Networks. 6th ed. Boston : Pearson, 2021. 944 p.

11. Stevens W. R., Fall K. R. TCP/IP Illustrated, Volume 1: The Protocols. 2nd ed. Boston : Addison-Wesley Professional, 2011. 1056 p.

12. Kurose J. F., Ross K. W. Computer Networking: A Top-Down Approach. 8th ed. Harlow : Pearson, 2021. 848 p.

13. Smith S. W. The Scientist and Engineer's Guide to Digital Signal Processing. San Diego : California Technical Publishing, 1997. 626 p.

14. Pohlmann K. C. Principles of Digital Audio. 6th ed. New York : McGraw-Hill Education, 2010. 816 p.

15. Lutz M. Programming Android: Java Programming for the New Generation of Mobile Devices. 2nd ed. Sebastopol : O'Reilly Media, 2012. 562 p.

16. Android Developers. AudioRecord : Reference Documentation. URL: https://developer.android.com/reference/android/media/AudioRecord (дата звернення: 10.04.2026).

17. Android Developers. NsdManager : Reference Documentation. URL: https://developer.android.com/reference/android/net/nsd/NsdManager (дата звернення: 10.04.2026).

18. Android Developers. Foreground Services : Developer Guide. URL: https://developer.android.com/develop/background-work/services/foreground-services (дата звернення: 10.04.2026).

19. Oracle Corporation. Java Sound API Programmer's Guide. Redwood Shores : Oracle, 2023. URL: https://docs.oracle.com/javase/tutorial/sound/index.html (дата звернення: 10.04.2026).

20. Oracle Corporation. Package java.net : Java SE 17 API Specification. URL: https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/net/package-summary.html (дата звернення: 10.04.2026).

21. JmDNS Project. JmDNS : Multicast DNS in Java. Version 3.5.9. URL: https://github.com/jmdns/jmdns (дата звернення: 10.04.2026).

22. FormDev Software GmbH. FlatLaf : Flat Look and Feel for Java Swing. Version 3.4.1. URL: https://www.formdev.com/flatlaf/ (дата звернення: 10.04.2026).

23. VB-Audio Software. Virtual Audio Cable : VB-CABLE Virtual Audio Device. URL: https://vb-audio.com/Cable/ (дата звернення: 10.04.2026).

24. Spittka J., Vos K., Valin J.-M. RTP Payload Format for the Opus Speech and Audio Codec : RFC 7587 / Internet Engineering Task Force. 2015. URL: https://www.rfc-editor.org/rfc/rfc7587 (дата звернення: 10.04.2026).

25. Rescorla E., Modadugu N. Datagram Transport Layer Security Version 1.2 : RFC 6347 / Internet Engineering Task Force. 2012. URL: https://www.rfc-editor.org/rfc/rfc6347 (дата звернення: 10.04.2026).

ДОДАТОК А

Лістинг класу PCMicGUI

Файл pc_server/src/main/java/PCMicGUI.java.

```
import javax.sound.sampled.*;
import javax.swing.*;
import com.formdev.flatlaf.FlatDarkLaf;
import javax.swing.UIManager;
import java.awt.*;
import java.net.*;
import java.util.Enumeration;
import javax.jmdns.JmDNS;
import javax.jmdns.ServiceInfo;
import java.io.IOException;

public class PCMicGUI extends JFrame {

    private JTextField portField = new JTextField("50005", 6);
    private JLabel ipLabel       = new JLabel("IP: " + detectBestIP());
    private JButton startBtn     = new JButton("Start");
    private JTextArea logArea    = new JTextArea(8, 45);
    private JComboBox<String> deviceBox = new JComboBox<>();

    private ReceiverThread worker;

    public PCMicGUI() {
        super("Phone Mic Receiver");

        logArea.setEditable(false);
        logArea.setFont(new Font(Font.MONOSPACED, Font.PLAIN, 12));

        JPanel top = new JPanel();
        top.add(new JLabel("Port:"));
        top.add(portField);
        top.add(ipLabel);
        top.add(startBtn);
        top.add(new JLabel("Output:"));
        updateAudioDevices();
        top.add(deviceBox);

        add(top, BorderLayout.NORTH);
        add(new JScrollPane(logArea), BorderLayout.CENTER);

        startBtn.addActionListener(e -> {
            if (worker == null) startServer();
            else                stopServer();
        });

            if (mixer == null) {
                throw new Exception("No audio device selected");
            }

            AudioFormat fmt = new AudioFormat(44100.0f, 16, 1, true, false);
            append("Using format: " + fmt.toString());

            DataLine.Info lineInfo = new DataLine.Info(SourceDataLine.class,
fmt);

            if (!mixer.isLineSupported(lineInfo)) {
                throw new Exception("Audio format not supported by selected
device");
            }

            SourceDataLine speakers = (SourceDataLine)
mixer.getLine(lineInfo);
            speakers.open(fmt);
            speakers.start();

            String localIP = detectBestIP();
            if ("unknown".equals(localIP)) {
                append("NSD: Не знайдено мережевий інтерфейс, сервіс не
буде анонсовано");
            } else {
                try {
                    InetAddress bindAddr = InetAddress.getByName(localIP);
                    append("NSD: Створення JmDNS на " + localIP + "...");
                    jmdns = JmDNS.create(bindAddr, localIP);
                    append("NSD: JmDNS створено, реєстрація сервісу...");
                    ServiceInfo serviceInfo = ServiceInfo.create(
                            "_pcmic._udp.local.",
                            "PC_Mic_Receiver",
                            port,
                            "path=/"
                    );
                    jmdns.registerService(serviceInfo);
                    append("NSD: Сервіс анонсовано на " + localIP + ":" + port);
                } catch (IOException e) {
                    append("NSD помилка: " + e.getMessage());
                }
            }
```

```
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    pack();
    setLocationRelativeTo(null);
}

private void updateAudioDevices() {
    deviceBox.removeAllItems();
    Mixer.Info[] mixerInfos = AudioSystem.getMixerInfo();
    for (Mixer.Info info : mixerInfos) {
        Mixer mixer = AudioSystem.getMixer(info);
        Line.Info[] sourceLines = mixer.getSourceLineInfo();
        if (sourceLines.length > 0) {
            deviceBox.addItem(info.getName());
        }
    }
}

private void startServer() {
    try {
        int port = Integer.parseInt(portField.getText().trim());
        worker = new ReceiverThread(port);
        worker.start();
        startBtn.setText("Stop");
        append("Listening on port " + port + "  (IP " + detectBestIP() + ")");
    } catch (Exception ex) {
        JOptionPane.showMessageDialog(this, ex.getMessage(),
                "Error", JOptionPane.ERROR_MESSAGE);
    }
}

private void stopServer() {
    if (worker != null) {
        worker.stopListening();
        worker = null;
    }
    startBtn.setText("Start");
    append("Stopped");
}

private class ReceiverThread extends Thread {
    private final int port;
    private DatagramSocket socket;

    ReceiverThread(int port) { this.port = port; }

    public void stopListening() {
        interrupt();
            byte[] buffer = new byte[16384];
            DatagramPacket packet = new DatagramPacket(buffer,
buffer.length);

            while (!isInterrupted()) {
                try {
                    socket.receive(packet);
                    speakers.write(packet.getData(), 0, packet.getLength());
                } catch (SocketException se) {
                    break;
                }
            }

            speakers.stop();
            speakers.close();

        } catch (Exception ex) {
            append("Error: " + ex.getMessage());
        } finally {
            if (jmdns != null) {
                jmdns.unregisterAllServices();
                try { jmdns.close(); } catch (IOException ignored) {}
                append("NSD: Сервіс приховано.");
            }
            if (socket != null && !socket.isClosed()) socket.close();
        }
    }

    private static String detectBestIP() {
        try {
            Enumeration<NetworkInterface> ifaces =
NetworkInterface.getNetworkInterfaces();
            while (ifaces.hasMoreElements()) {
                NetworkInterface nif = ifaces.nextElement();
                if (!nif.isUp() || nif.isLoopback() || nif.isVirtual()) continue;
                for (InterfaceAddress ia : nif.getInterfaceAddresses()) {
                    InetAddress addr = ia.getAddress();
                    if (addr instanceof Inet4Address && addr.isSiteLocalAddress())
{
                        if (addr.getHostAddress().startsWith("169.254")) continue;
                        return addr.getHostAddress();
                    }
                }
            }
        } catch (Exception ignored) { }
        return "unknown";
    }
```

```
    if (socket != null && !socket.isClosed()) {
        socket.close();
    }
}

@Override
public void run() {
    JmDNS jmdns = null;

    try {
        socket = new DatagramSocket(port);

        String selectedDevice = (String)deviceBox.getSelectedItem();
        Mixer.Info[] mixerInfos = AudioSystem.getMixerInfo();
        Mixer mixer = null;

        for (Mixer.Info info : mixerInfos) {
            if (info.getName().equals(selectedDevice)) {
                mixer = AudioSystem.getMixer(info);
                append("Using audio device: " + info.getName());
                break;
            }
        }
```

```
    private void append(String msg) {
        SwingUtilities.invokeLater(() -> {
            logArea.append(msg + "\n");
            logArea.setCaretPosition(logArea.getDocument().getLength());
        });
    }

    public static void main(String[] args) {
        try {
            FlatDarkLaf.setup();

            UIManager.put("Button.arc", 8);
            UIManager.put("TextComponent.arc", 8);
        } catch (Exception ex) {
            System.err.println("Не вдалося ініціалізувати тему FlatLaf");
        }

        SwingUtilities.invokeLater(() -> new PCMicGUI().setVisible(true));
    }
}
```

ДОДАТОК Б

Лістинг класу MainActivity

Файл PhoneMic/app/src/main/java/com/example/phonemic/MainActivity.java.

```
package com.example.phonemic;

import android.Manifest;
import android.content.Context;
import android.content.Intent;
import android.content.pm.PackageManager;
import android.net.nsd.NsdManager;
import android.net.nsd.NsdServiceInfo;
import android.os.Bundle;
import android.os.Handler;
import android.os.Looper;
import android.util.Log;
import android.view.View;
import android.widget.AdapterView;
import android.widget.ArrayAdapter;
import android.widget.Button;
import android.widget.EditText;
import android.widget.ListView;
import android.widget.Toast;

import androidx.annotation.NonNull;
import androidx.appcompat.app.AppCompatActivity;
import androidx.core.app.ActivityCompat;
import androidx.core.content.ContextCompat;

import java.util.ArrayList;
import java.util.concurrent.ConcurrentHashMap;

public class MainActivity extends AppCompatActivity {

    private static final int PERMISSION_CODE = 1234;

    private Button btnStartStop;
    private EditText editIp, editPort;
    private ListView serverListView;

    private NsdManager nsdManager;
    private NsdManager.DiscoveryListener discoveryListener;
    private final String SERVICE_TYPE = "_pcmic._udp.";
    private boolean discoveryActive = false;

    private final ConcurrentHashMap<String, NsdServiceInfo>
discoveredServers = new ConcurrentHashMap<>();

    private void restartDiscovery() {
        if (discoveryActive && discoveryListener != null) {
            try {
                nsdManager.stopServiceDiscovery(discoveryListener);
            } catch (Exception e) {
                Log.e("NSD", "stopDiscovery error: " + e.getMessage());
            }
            discoveryActive = false;
        }
        initializeDiscoveryListener();
        try {
            nsdManager.discoverServices(SERVICE_TYPE,
NsdManager.PROTOCOL_DNS_SD, discoveryListener);
        } catch (Exception e) {
            Log.e("NSD", "Failed to restart discovery: " + e.getMessage());
        }
    }

    private void initializeDiscoveryListener() {
        discoveryListener = new NsdManager.DiscoveryListener() {
            @Override
            public void onDiscoveryStarted(String regType) {
                Log.d("NSD", "Discovery started");
                discoveryActive = true;
            }

            @Override
            public void onServiceFound(NsdServiceInfo service) {
                Log.d("NSD", "Service found: name=" + service.getServiceName()
                        + " type=" + service.getServiceType());

                if (!service.getServiceType().contains("_pcmic._udp")) {
                    return;
                }

                nsdManager.resolveService(service, new
NsdManager.ResolveListener() {
                    @Override
                    public void onResolveFailed(NsdServiceInfo serviceInfo, int
errorCode) {
                        Log.e("NSD", "Resolve failed: errorCode=" + errorCode);
                    }
```

```
    private final ArrayList<String> serverDisplayList = new ArrayList<>();
    private ArrayAdapter<String> serverAdapter;

    private static final long DISCOVERY_RESTART_INTERVAL_MS =
5000;
    private final Handler discoveryHandler = new
Handler(Looper.getMainLooper());
    private final Runnable discoveryRestartRunnable = new Runnable() {
        @Override
        public void run() {
            restartDiscovery();
            discoveryHandler.postDelayed(this,
DISCOVERY_RESTART_INTERVAL_MS);
        }
    };

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        editIp = findViewById(R.id.editIp);
        editPort = findViewById(R.id.editPort);
        btnStartStop = findViewById(R.id.btnStartStop);
        serverListView = findViewById(R.id.serverList);

        serverAdapter = new ArrayAdapter<>(this,
                android.R.layout.simple_list_item_1, serverDisplayList);
        serverListView.setAdapter(serverAdapter);

        serverListView.setOnItemClickListener(new
AdapterView.OnItemClickListener() {
            @Override
            public void onItemClick(AdapterView<?> parent, View view, int
position, long id) {
                String displayText = serverDisplayList.get(position);
                for (ConcurrentHashMap.Entry<String, NsdServiceInfo> entry :
discoveredServers.entrySet()) {
                    NsdServiceInfo info = entry.getValue();
                    String key = entry.getKey();
                    String expected = key + " (" + info.getHost().getHostAddress()
                            + ":" + info.getPort() + ")";
                    if (expected.equals(displayText)) {
                        if (AudioStreamService.isRunning) {
                            stopStreaming();
                        }
                        editIp.setText(info.getHost().getHostAddress());
                        editPort.setText(String.valueOf(info.getPort()));
                        @Override
                        public void onServiceResolved(NsdServiceInfo serviceInfo) {
                            final String name = serviceInfo.getServiceName();
                            final String ip = serviceInfo.getHost().getHostAddress();
                            final int port = serviceInfo.getPort();
                            Log.d("NSD", "Resolved: " + name + " ip=" + ip + " port=" +
port);

                            discoveredServers.put(name, serviceInfo);
                            runOnUiThread(() -> refreshServerList());
                        }
                    });
                }

                @Override
                public void onServiceLost(NsdServiceInfo service) {
                    Log.d("NSD", "Service lost: " + service.getServiceName());
                    discoveredServers.remove(service.getServiceName());
                    runOnUiThread(() -> refreshServerList());
                }

                @Override
                public void onDiscoveryStopped(String serviceType) {
                    Log.d("NSD", "Discovery stopped");
                    discoveryActive = false;
                }

                @Override
                public void onStartDiscoveryFailed(String serviceType, int
errorCode) {
                    nsdManager.stopServiceDiscovery(this);
                }

                @Override
                public void onStopDiscoveryFailed(String serviceType, int errorCode)
{
                    nsdManager.stopServiceDiscovery(this);
                }
            };
        }

    private void refreshServerList() {
        serverDisplayList.clear();
        for (ConcurrentHashMap.Entry<String, NsdServiceInfo> entry :
discoveredServers.entrySet()) {
            NsdServiceInfo info = entry.getValue();
            serverDisplayList.add(entry.getKey() + " ("
                    + info.getHost().getHostAddress() + ":" + info.getPort() + ")");
```

```
                stopDiscoveryCycle();
                startStreaming();
                return;
            }
        }
    }
});

nsdManager = (NsdManager)
getSystemService(Context.NSD_SERVICE);
startDiscoveryCycle();

if (AudioStreamService.isRunning) {
    btnStartStop.setText("Stop");
}

btnStartStop.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        if (!AudioStreamService.isRunning) {
            startStreaming();
        } else {
            stopStreaming();
        }
    }
});
}

@Override
protected void onDestroy() {
    stopDiscoveryCycle();
    super.onDestroy();
}

private void startDiscoveryCycle() {
    initializeDiscoveryListener();
    try {
        nsdManager.discoverServices(SERVICE_TYPE,
NsdManager.PROTOCOL_DNS_SD, discoveryListener);
    } catch (Exception e) {
        Log.e("NSD", "Failed to start discovery: " + e.getMessage());
    }
    discoveryHandler.postDelayed(discoveryRestartRunnable,
DISCOVERY_RESTART_INTERVAL_MS);
}

private void stopDiscoveryCycle() {
    discoveryHandler.removeCallbacks(discoveryRestartRunnable);
    }
    serverAdapter.notifyDataSetChanged();
}

private void startStreaming() {
    String ipStr = editIp.getText().toString().trim();
    String portStr = editPort.getText().toString().trim();
    if (ipStr.isEmpty() || portStr.isEmpty()) {
        Toast.makeText(this, "Enter IP and port",
Toast.LENGTH_SHORT).show();
        return;
    }
    int port = Integer.parseInt(portStr);
    if (ContextCompat.checkSelfPermission(this,
Manifest.permission.RECORD_AUDIO) !=
PackageManager.PERMISSION_GRANTED) {
        ActivityCompat.requestPermissions(this, new
String[]{Manifest.permission.RECORD_AUDIO}, PERMISSION_CODE);
        return;
    }
    stopDiscoveryCycle();
    Intent svc = new Intent(this, AudioStreamService.class);
    svc.putExtra("ip", ipStr);
    svc.putExtra("port", port);
    ContextCompat.startForegroundService(this, svc);
    btnStartStop.setText("Stop");
}

private void stopStreaming() {
    stopService(new Intent(this, AudioStreamService.class));
    btnStartStop.setText("Start");
    discoveredServers.clear();
    refreshServerList();
    startDiscoveryCycle();
}

@Override
public void onRequestPermissionsResult(int requestCode, @NonNull
String[] permissions, @NonNull int[] grantResults) {
    super.onRequestPermissionsResult(requestCode, permissions,
grantResults);
    if (requestCode == PERMISSION_CODE && grantResults.length > 0
&& grantResults[0] == PackageManager.PERMISSION_GRANTED) {
        startStreaming();
    } else {
        Toast.makeText(this, "Microphone permission required",
Toast.LENGTH_SHORT).show();
    }  }}
```

```
        if (nsdManager != null && discoveryListener != null &&
discoveryActive) {
            try {
                nsdManager.stopServiceDiscovery(discoveryListener);
            } catch (Exception e) {
                Log.e("NSD", "stopDiscovery error: " + e.getMessage());
            }
            discoveryActive = false;
        }
    }
```

ДОДАТОК В

Лістинг класу AudioStreamService

Файл PhoneMic/app/src/main/java/com/example/phonemic/AudioStreamService.java.

```
package com.example.phonemic;

import android.app.Notification;
import android.app.NotificationChannel;
import android.app.NotificationManager;
import android.app.PendingIntent;
import android.app.Service;
import android.content.Intent;
import android.content.pm.ServiceInfo;
import android.media.AudioFormat;
import android.media.AudioRecord;
import android.media.MediaRecorder;
import android.net.wifi.WifiManager;
import android.os.Build;
import android.os.IBinder;
import android.os.PowerManager;
import android.util.Log;

import androidx.core.app.NotificationCompat;

import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;

public class AudioStreamService extends Service {

    private static final String TAG = "AudioStreamService";
    private static final String CHANNEL_ID = "audio_stream_channel";
    private static final int NOTIFICATION_ID = 1;

    private static final int SAMPLE_RATE = 44100;
    private static final int CHANNEL_CFG =
AudioFormat.CHANNEL_IN_MONO;
    private static final int AUDIO_FMT =
AudioFormat.ENCODING_PCM_16BIT;

    static volatile boolean isRunning = false;

    private StreamThread streamThread;
    private PowerManager.WakeLock wakeLock;
    private WifiManager.WifiLock wifiLock;

    @Override
    if (streamThread != null) {
        streamThread.stopRunning();
        streamThread = null;
    }
    if (wakeLock != null && wakeLock.isHeld()) {
        wakeLock.release();
    }
    if (wifiLock != null && wifiLock.isHeld()) {
        wifiLock.release();
    }
    super.onDestroy();
}

@Override
public IBinder onBind(Intent intent) {
    return null;
}

private void createNotificationChannel() {
    NotificationChannel channel = new NotificationChannel(
            CHANNEL_ID, "Audio Streaming",
            NotificationManager.IMPORTANCE_LOW);
    channel.setDescription("Active while audio is being streamed to PC");

getSystemService(NotificationManager.class).createNotificationChannel(channel);
}

private Notification buildNotification(String ip, int port) {
    Intent openApp = new Intent(this, MainActivity.class);
    openApp.setFlags(Intent.FLAG_ACTIVITY_SINGLE_TOP);
    PendingIntent pendingIntent = PendingIntent.getActivity(
            this, 0, openApp, PendingIntent.FLAG_IMMUTABLE);

    return new NotificationCompat.Builder(this, CHANNEL_ID)
            .setContentTitle("PhoneMic")
            .setContentText("Streaming to " + ip + ":" + port)
            .setSmallIcon(R.drawable.ic_mic)
            .setContentIntent(pendingIntent)
            .setOngoing(true)
            .build();
}

@SuppressWarnings("MissingPermission")
```

```
    public void onCreate() {
        super.onCreate();
        createNotificationChannel();
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        if (intent == null) {
            stopSelf();
            return START_NOT_STICKY;
        }

        String ip = intent.getStringExtra("ip");
        int port = intent.getIntExtra("port", 50005);

        Notification notification = buildNotification(ip, port);
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.Q) {
            startForeground(NOTIFICATION_ID, notification,

ServiceInfo.FOREGROUND_SERVICE_TYPE_MICROPHONE);
        } else {
            startForeground(NOTIFICATION_ID, notification);
        }

        PowerManager pm = (PowerManager)
getSystemService(POWER_SERVICE);
        wakeLock =
pm.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK,
"PhoneMic::StreamWakeLock");
        wakeLock.acquire();

        WifiManager wm = (WifiManager)
getApplicationContext().getSystemService(WIFI_SERVICE);
        int wifiMode = Build.VERSION.SDK_INT >=
Build.VERSION_CODES.Q
                ? WifiManager.WIFI_MODE_FULL_LOW_LATENCY
                : WifiManager.WIFI_MODE_FULL_HIGH_PERF;
        wifiLock = wm.createWifiLock(wifiMode,
"PhoneMic::StreamWifiLock");
        wifiLock.acquire();

        try {
            InetAddress addr = InetAddress.getByName(ip);
            streamThread = new StreamThread(addr, port);
            streamThread.start();
            isRunning = true;
        } catch (Exception e) {
            Log.e(TAG, "Failed to start stream: " + e.getMessage());

    private static class StreamThread extends Thread {
        private final InetAddress destAddr;
        private final int destPort;
        private volatile boolean running = true;

        StreamThread(InetAddress destAddr, int destPort) {
            this.destAddr = destAddr;
            this.destPort = destPort;
        }

        void stopRunning() {
            running = false;
        }

        @Override
        public void run() {
            try {
                int minBuf = AudioRecord.getMinBufferSize(SAMPLE_RATE,
CHANNEL_CFG, AUDIO_FMT);
                AudioRecord recorder = new AudioRecord(
                        MediaRecorder.AudioSource.MIC,
                        SAMPLE_RATE, CHANNEL_CFG, AUDIO_FMT, minBuf);

                DatagramSocket socket = new DatagramSocket();
                byte[] buf = new byte[minBuf];

                recorder.startRecording();

                while (running) {
                    int read = recorder.read(buf, 0, buf.length);
                    if (read > 0) {
                        DatagramPacket packet = new DatagramPacket(buf, read,
destAddr, destPort);
                        socket.send(packet);
                    }
                }

                recorder.stop();
                recorder.release();
                socket.close();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
        stopSelf();
    }

    return START_NOT_STICKY;
}

@Override
public void onDestroy() {
    isRunning = false;
```

ДОДАТОК Г

Конфігурація Android-маніфесту та розмітки

Файл PhoneMic/app/src/main/AndroidManifest.xml.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.phonemic">

    <uses-permission android:name="android.permission.RECORD_AUDIO"/>
    <uses-permission android:name="android.permission.INTERNET"/>
    <uses-permission android:name="android.permission.ACCESS_WIFI_STATE"/>
    <uses-permission
android:name="android.permission.CHANGE_WIFI_MULTICAST_STATE"/>
    <uses-permission
android:name="android.permission.FOREGROUND_SERVICE"/>
    <uses-permission
android:name="android.permission.FOREGROUND_SERVICE_MICROPHONE"/>
    <uses-permission android:name="android.permission.WAKE_LOCK"/>

    <application
        android:allowBackup="true"
        android:label="PhoneMic"
        android:theme="@style/Theme.AppCompat.Light.NoActionBar">

        <activity android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

        <service
            android:name=".AudioStreamService"
            android:foregroundServiceType="microphone"
            android:exported="false" />
    </application>
</manifest>
```

Файл PhoneMic/app/src/main/res/layout/activity_main.xml.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:padding="16dp"
    android:gravity="center_horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Доступні сервери:"
        android:textSize="16sp"
        android:textStyle="bold"
        android:layout_marginBottom="4dp" />

    <ListView
        android:id="@+id/serverList"
        android:layout_width="match_parent"
        android:layout_height="180dp"
        android:layout_marginBottom="16dp"
        android:background="@android:drawable/editbox_background"
        android:divider="@android:color/darker_gray"
        android:dividerHeight="1dp" />

    <EditText
        android:id="@+id/editIp"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="PC IP (e.g. 192.168.0.100)"
        android:text="192.168.1.103" />

    <EditText
        android:id="@+id/editPort"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Port (e.g. 50005)"
        android:inputType="number"
        android:text="50005" />

    <Button
        android:id="@+id/btnStartStop"
        android:text="Start"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="24dp" />
</LinearLayout>
```

ДОДАТОК Ґ

Конфігурація системи збирання Gradle

Файл pc_server/build.gradle (серверна частина).

```
plugins {
    id 'java'
    id 'application'
}

repositories {
    mavenCentral()
}

dependencies {
    implementation 'com.formdev:flatlaf:3.4.1'
    implementation 'org.jmdns:jmdns:3.5.9'
}

application {
    mainClass = 'PCMicGUI'
}

java {
    sourceCompatibility = JavaVersion.VERSION_17
    targetCompatibility = JavaVersion.VERSION_17
}
```

Файл PhoneMic/app/build.gradle (клієнтська частина).

```
plugins {
    id 'com.android.application'
}

android {
    namespace 'com.example.phonemic'
    compileSdkVersion 34

    defaultConfig {
        applicationId "com.example.phonemic"
        minSdkVersion 26
        targetSdkVersion 34
        versionCode 1
        versionName "1.0"
    }

    compileOptions {
        sourceCompatibility JavaVersion.VERSION_17
        targetCompatibility JavaVersion.VERSION_17
```

```
    }

    buildTypes {
        release {
            minifyEnabled false
            debuggable false
            signingConfig signingConfigs.debug
        }
    }
}

dependencies {
    implementation 'androidx.appcompat:appcompat:1.7.0'
    implementation 'com.google.android.material:material:1.12.0'
}
```

Файл PhoneMic/settings.gradle.

```
include ':app'
```

ДОДАТОК Д

Демонстраційні матеріали

1

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ТЕХНОЛОГІЙ ЦИФРОВОГО РОЗВИТКУ

Цифрова клієнт-серверна система потокової передачі аудіо з мобільного пристрою на ПК на основі Android SDK

Виконав студент 4 курсу
групи ТЦР-44
Крєпкій Ігор Тарасович
Керівник роботи
д-р філософії, Ніщеменко Дмитро Олександрович

Київ 2026

2

Актуальність, об'єкт, предмет та мета дослідження

Актуальність

Розрив між апаратними можливостями вбудованих мікрофонів смартфонів та обмеженими інтерфейсами їхньої інтеграції з ПК без деградації частотного діапазону (як у Bluetooth HFP/HSP) або прив'язки до USB-кабелю.

Об'єкт

процеси потокової передачі нестиснених цифрових аудіосигналів між мобільними та стаціонарними вузлами у локальній мережі.

Предмет

програмне забезпечення, яке реалізує захоплення, інкапсуляцію, транспортування та відтворення PCM-потоку з використанням протоколу UDP та сервісу автовиявлення mDNS.

Мета

проєктування та програмна реалізація клієнт-серверної системи для передачі аудіо з Android-пристрою зі зменшенням наскрізної затримки передачі аудіосигналу до рівня, придатного для інтерактивного голосового застосування (не більше 80 мс).

3

Обґрунтування актуальності: динаміка ринку

Масштабованість

Зростання аудиторії Microsoft Teams у 160 разів (з 2 млн до 320 млн) за останні 7 років підтверджує масовий перехід на цифрову взаємодію.

Апаратний розрив

Більшість ноутбуків та бюджетних гарнітур для ПК оснащені мікрофонами низької якості, що не відповідають сучасним вимогам до чіткості звуку.

Економічна доцільність

Проєктована система дозволяє використати апаратні переваги сучасних смартфонів замість придбання професійного студійного обладнання.

Рис. 3.1 Графік користувачів Teams з 2017 по 2023 рік

4

Аналіз аналогічних рішень

WO Mic | **DroidCam** | **iVCam** | **AudioRelay** | **Mumble**

Недоліки

Обмеження частоти дискретизації (16 кГц у WO Mic), використання TCP/HTTP з високою наскрізною затримкою (DroidCam), закриті протоколи, потреба встановлення пропрієтарних драйверів на хості.

Практична новизна

- Вперше реалізовано комбінацію відкритого нестисненого PCM-over-UDP транспорту зі стандартизованим автовиявленням DNS-SD.
- Досягнуто латентності ≤ 80 мс без використання пропрієтарних драйверів рівня ядра ОС та ручного введення IP-адрес.

5

Архітектура програмного комплексу

Схема	Клієнт (Android)	Сервер (ПК)
Дворівнева клієнт-серверна модель.	Захоплення сигналу, пакетування, надсилання датаграм (ініціатор стрімінгу).	Демультиплексування датаграм, маршрутизація нестисненого потоку в системний віртуальний аудіокабель.

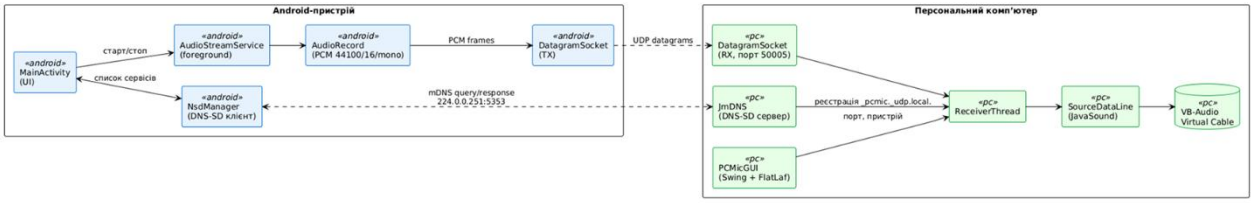


Рис. 5.1 UML діаграма розгортання системи

6

Технології обробки та передавання потоку

Формат даних

Нестиснений PCM, 44.1 кГц, 16 біт, моно.

Бітрейт

705,6 кбіт/с.

Транспортний рівень

Протокол UDP. Варіанти TCP та HTTP-базованих протоколів (HLS, DASH) відкинуті через проблему head-of-line blocking та затримку.

Буферизація

Статична стратегія управління буфером. Де-джиттерна компенсація покладена на внутрішню чергу SourceDataLine приймача.

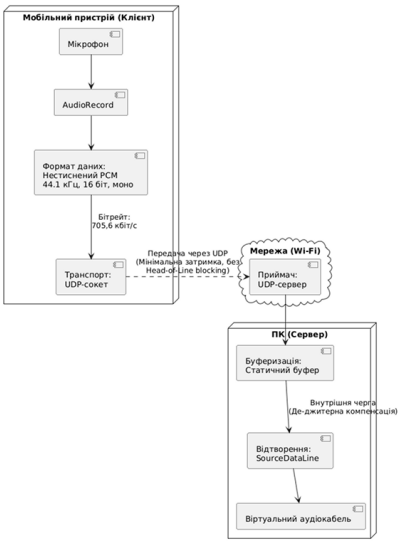


Рис. 6.1 UML Діаграма компонентів передачі аудіоданих

7

Механізм автовиявлення (Service Discovery)

Протокол	Тип сервісу	Реалізація сервера	Реалізація клієнта
DNS-SD (mDNS) через IP-мультикаст (адреса 224.0.0.251, порт 5353).	_pcmic._udp.local	Бібліотека JmDNS (pure-Java). Реєстрація сервісу здійснюється з прив'язкою до виявленої активної IPv4-адреси локальної мережі.	Системний клас NsdManager. Впроваджено періодичний рестарт циклу виявлення (кожні 5 с) для компенсації архітектурної специфіки mDNSResponder.

8

Механізм автовиявлення (Service Discovery)

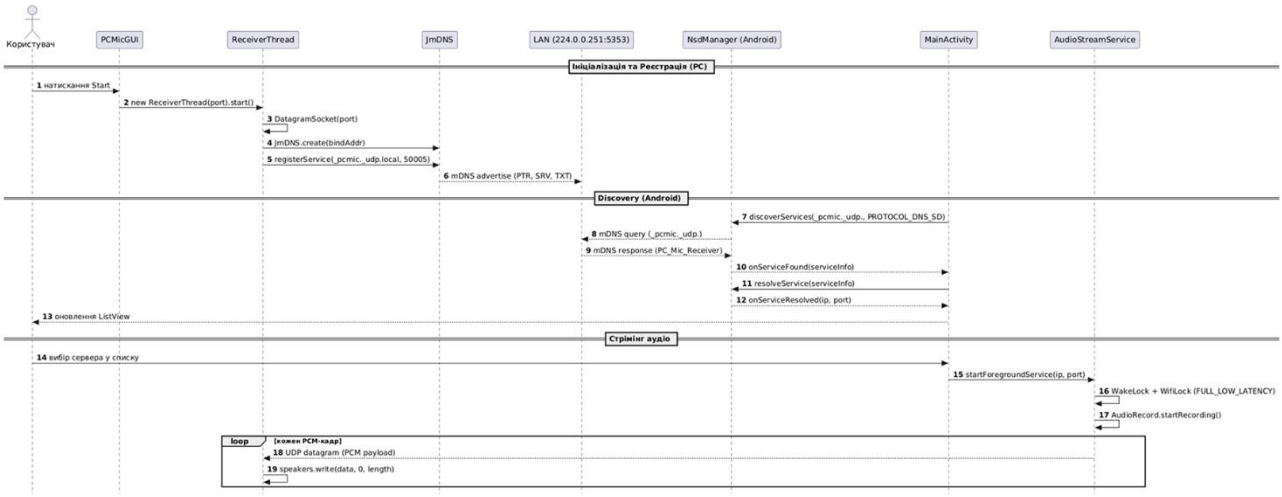


Рис. 8.1 UML діаграма послідовності механізму автовиявлення

9

Програмна реалізація клієнта (Android)

Модульна структура	Захоплення	Управління ресурсами	Цикл передачі
MainActivity (UI) та AudioStreamService (фоновий режим).	API AudioRecord з параметром MediaRecorder.AudioSource.MIC для активації апаратного шумозаглушення.	Утримання PARTIAL_WAKE_LOCK та WifiLock у режимі WIFI_MODE_FULL_LOW_LATENCY для блокування переходу радіотракту у сплячий стан.	Блокуюче зчитування PCM-фреймів, їх інкапсуляція в DatagramPacket та мережева трансляція з алгоритмічною складністю O(1).

10

Програмна реалізація клієнта (Android)

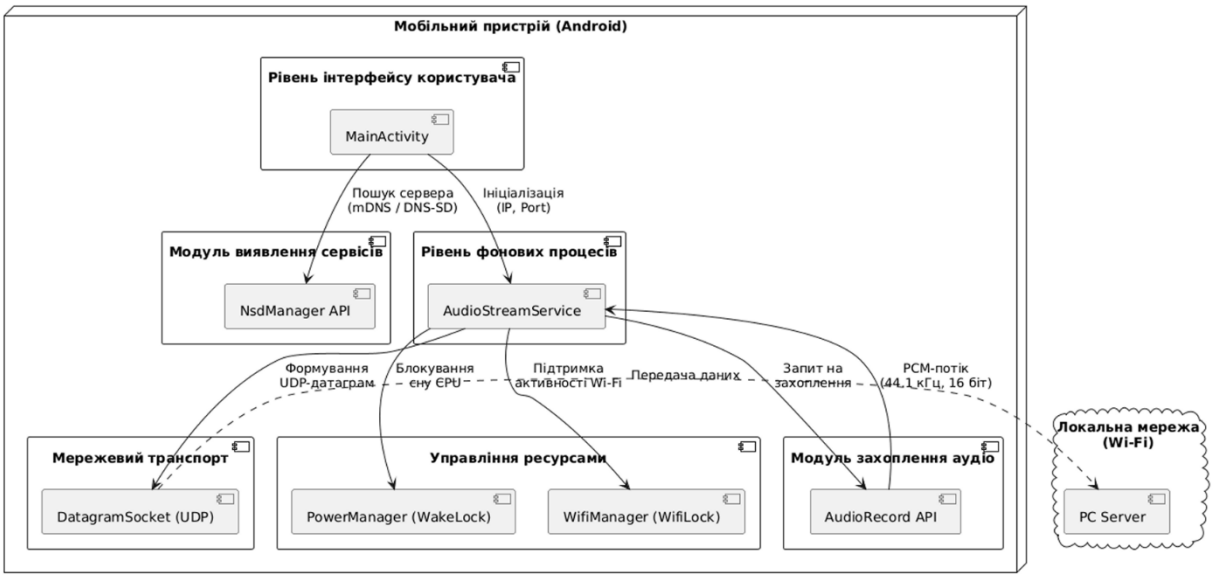


Рис. 10.1 UML діаграма компонентів клієнта (Android)

11

Програмна реалізація сервера (PC)

Стек	Аудіопідсистема	Ініціалізація	Багатопотоковість
Java SE 17, Swing, тема FlatLaf.	JavaSound API. Використовується блокуючий запис у SourceDataLine.	Динамічна побудова переліку доступних пристроїв відтворення через AudioSystem.getMixerInfo() та програмна верифікація формату викликом `isLineSupported()`.	Ізоляція Event Dispatch Thread та робочого потоку ReceiverThread.

12

Програмна реалізація сервера (PC)

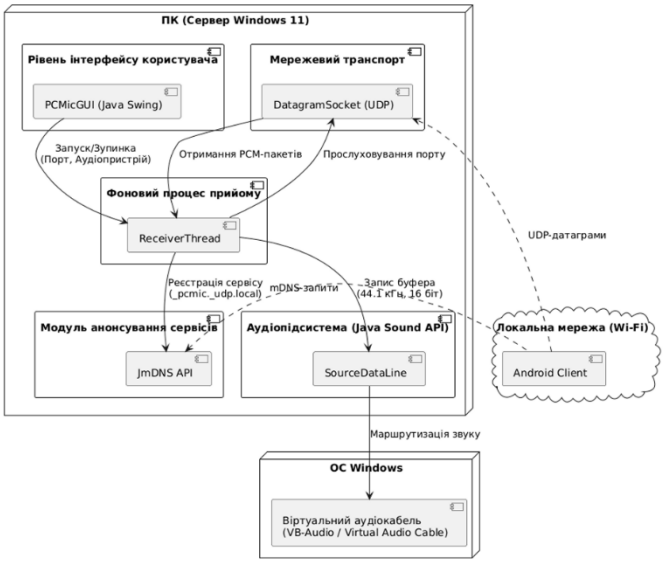


Рис. 12.1 UML діаграма компонентів сервера (PC)

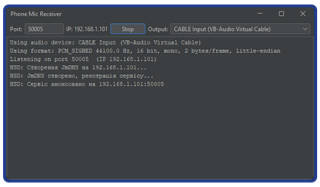
13

Набір технологій проєкту

Шар	Клієнт (Android)	Сервер (ПК)
Мова	Java SE 17	Java SE 17
Платформа	Android API 26–34	JVM / будь-яка ОС
UI-бібліотека	Android View (XML + AppCompat 1.7.0, Material 1.12.0)	Swing + FlatLaf 3.4.1
Аудіо API	AudioRecord (PCM 16-bit, 44100 Hz, mono)	JavaSound (SourceDataLine)
Мережа	DatagramSocket (UDP TX)	DatagramSocket (UDP RX, порт 50005)
Автовиявлення	NsdManager (PROTOCOL_DNS_SD)	JmDNS 3.5.9
Фоновий режим	Service (TYPE_MICROPHONE) + WakeLock + WifiLock	Standalone Swing JFrame
Система збірки	Android Gradle Plugin	Gradle (java + application)

14

Інтерфейс користувача та розгортання



Android-застосунок

Список виявлених серверів, поля для ручного введення адреси (fallback), підтримка роботи згортанням завдяки foreground-нотифікації.

ПК застосунок

Оболонка «single-screen utility», лог подій системи, випадаючий список вибору віртуального аудіокабелю.

15

Результати тестування системи

Види та об'єкт тестування

Функціональне	Оцінка затримки	Стрес-тестування
Перевірка алгоритмів detectBestIP(), mDNS-анонсування та автоматичного вивільнення UDP-портів.	Розрахунок латентності буферного тракту (AudioRecord + Мережа + SourceDataLine).	6-годинна безперервна трансляція («довгий сеанс») для оцінки стабільності та енергоспоживання.

Умови проведення випробувань

Мережа	Обладнання
Wi-Fi 5 ГГц (IEEE 802.11ac), рівень сигналу RSSI > -65 дБм.	Смартфон Samsung Galaxy A35 (Android 16), ПК під керуванням Windows 11.

Метрики та результати

Наскрізна латентність	Стабільність потоку	Енергоефективність
42–80 мс (відповідає НФВ-1, поріг інтерактивності).	Пропускна здатність 705,6 кбіт/с без втрат синхронізації та чутних артефактів.	Середнє споживання заряду становить 11 %/год (у фоновому режимі).

16

Висновки та перспективи розвитку

Архітектура та відкритість	Кількісні показники	Ресурсна ефективність
Реалізовано клієнт-серверний комплекс на базі відкритих стандартів (UDP, DNS-SD). Доведено можливість повноцінної інтеграції смартфона з десктопним ПЗ без встановлення пропрієтарних драйверів рівня ядра ОС.	Досягнуто наскрізної латентності у діапазоні 42–80 мс. Забезпечено стабільну передачу нестисненого PCM-потоку (705,6 кбіт/с) у мережах Wi-Fi без деградації частотного діапазону, властивої Bluetooth-рішенням.	Експериментально підтверджено стабільність системи під час тривалих сесій. Показник енергоспоживання клієнта (орієнтовно 11 %/год на Samsung Galaxy A35) є прийнятним для практичного використання

Напрями подальшого вдосконалення системи:

1. Впровадження кодека Opus та RTP для компенсації втрат пакетів.
2. Реалізація fallback-виявлення вузлів через UDP broadcast.
3. Додавання шифрування DTLS та мікшування кількох джерел.

17

Апробація

1. Крєпкій І. Т. Програмна маршрутизація аудіопотоку в просторі користувача з використанням Java Sound API. VII Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях» : тези доп., 23 квіт. 2026 р. – Подано до друку.

2. Крєпкій І. Т. Програмна маршрутизація аудіопотоку в просторі користувача з використанням Java Sound API. Всеукраїнська науково-технічна конференція «Технології цифрового розвитку: програмні системи та децентралізація» : тези доп., 1–3 трав. 2026 р. – Подано до друку.

Дякую за увагу!

Готовий відповісти на ваші запитання.