Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра обчислювальної техніки

КОМПЛЕКСНА БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«Апаратно-програмний комплекс для дистанційного передавання аудіо. Частина 2. Застосунок для управління»

08-54. КБКР.037.00.000 ПЗ

Виконав: студент 4 курсу, групи 1КІ-21б
спеціальності 123 — «Комп'ютерна інженерія»
освітня програма — «Комп'ютерна інженерія»
Дацюк В. В. _____

Керівник д.т.н., проф. каф. ОТ
_____ Мартинюк Т.Б.

Рецензент к.т.н., доц. каф. ПЗ
_____ Катєльніков Д. І.

Допущено до захисту
Завідувач кафедри ОТ
д.т.н., проф. Азаров О. Д.

«17» 06 2025 р.

Вінниця ВНТУ – 2025 рік

Вінницький національний технічний університет

Факультет інформаційних технологій та комп'ютерної інженерії

Кафедра обчислювальної техніки

Освітньо—кваліфікаційний рівень бакалвр

Галузь знань — 12 Інформаційні технології

Спеціальність — 123 Комп'ютерна інженерія

Освітня програма — Комп'ютерна інженерія

ЗАТВЕРДЖУЮ

Завідувач кафедри ОТ

д.т.н.проф._____Азаров О. Д.

«21» березня 2025 року

З А В Д А Н Н Я

НА КОМПЛЕКСНУ БАКАЛАВРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ

Дацюку Вадиму Васильовичу

1 Тема роботи: Апаратно-програмний комплекс для дистанційного передавання аудіо. Частина 2. керівник роботи Мартинюк Тетяна Борисівна, д.т.н., професор кафедри ОТ, затверджено наказом вищого навчального закладу від «20» березня 2025 року №97.

2 Строк подання студентом роботи 13.05.2025

3 Вихідні дані до роботи: технічний опис технологій для дистанційної передачі аудіо в реальному часі, опис технології розробки програм з використанням .NET MAUI, опис алгоритмів дистанційної передачі аудіо.

4 Зміст розрахунково-пояснювальної записки: вступ; огляд існуючих рішень для дистанційного передавання аудіо, аналіз функціональних можливостей аналогів, вибір технологій передачі даних в реальному часі, вибір технологій передачі файлів, вибір технологій для розробки мобільних застосунків, розгляд та вибір середовищ розробки, опис архітектури мобільного застосунку, опис алгоритмів передачі та отримання аудіопотоку, опис передачі аудіофайлів,

реалізація інтерфейсу застосунку, реалізація передачі та прийому аудіопотоку, реалізація передачі аудіофалів, тестування застосунку, висновки.

5 Перелік графічного матеріалу: діаграма класів в Visual Studio, архітектура, діаграма застосунку, блок-схема алгоритму передачі аудіо в реальному часі, блок-схема алгоритму передачі аудіофайлів.

6 Консультанти розділів роботи приведені в таблиці 1.

Таблиця 1 — Консультанти роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1-4	к.т.н., доц. каф. ОТ Мартинюк Т. Б.	21.03.25	13.05.25
Нормоконтроль	ас. каф. ОТ Швець С. І.	14.05.25	30.05.25

7 Дата видачі завдання 21.03.2025 р.

8 Календарний план приведений в таблиці 2.

Таблиця 2 — Календарний план

№ з/п	Назва етапів дипломної роботи	Строк виконання	Підпис
1	Постановка задачі роботи	21.03.25	вик
2	Пошук та огляд існуючих рішень	21.03.25—5.04.25	вик
3	Пошук та аналіз сучасних технологій розробки мобільних додатків	6.04.25—10.04.25	вик
4	Визначення алгоритму передачі даних в мережі	11.04.25—20.04.25	вик
5	Розробка й тестування застосунку	21.04.25—1.05.25	вик
7	Оформлення пояснювальної записки та ілюстративного матеріалу	2.05.25—12.05.25	вик
8	Перевірка якості виконання бакалаврської кваліфікаційної роботи та усунення недоліків	13.05.25	вик

Студент Вадим ДАЦЮК

Керівник роботи д.т.н., проф. Тетяна МАРТИНЮК

АНОТАЦІЯ

УДК 004.93

Дацюк В. В. Апаратно-програмний комплекс для дистанційного передавання аудіо. Частина 2. Застосунок для управління. Комплексна бакалаврська робота зі спеціальності 123 — Комп'ютерна інженерія, освітня програма — Комп'ютерна інженерія. Вінниця: ВНТУ, 2025. 91 с.

На укр. мові. Бібліогр.: 19 назв; рис.: 25.

У комплексній бакалаврській кваліфікаційній роботі розроблено апаратно-програмний комплекс для дистанційного передавання аудіо. Друга частина роботи включає розробку застосунку для управління мікроконтролером ESP32, двосторонньої передачі аудіопотоку та дистанційної передачі аудіофайлів. Було реалізовано можливість здійснювати підключення до пристрою через локальну мережу, приймати та відтворювати або передавати аудіо в реальному часі, надсилати аудіофайли і надавати можливість керування списком файлів на SD-карті мікроконтролеру. Реалізовано підтримку UDP-протоколу для потокової передачі та TCP-протоколу для обміну файлами.

Для реалізації застосунку було використано платформу MAUI, що дозволяє створювати кросплатформені програми з використанням мови програмування C# та платформи .NET.

Робота дозволяє розглянути розробку програм для керування засобами IoT, зокрема створення застосунків для отримання інформації з мікроконтролерів і датчиків.

Ключові слова: IoT, .NET, MAUI, C#, ESP32, дистанційна передача аудіо, мережеві протоколи, застосунок.

ABSTRACT

Datsyuk, V. V. Hardware and Software System for Remote Audio Transmission. Part 2. Control Application. Complex Bachelor's Thesis in Specialty 123 — Computer Engineering, Educational Program — Computer Engineering. Vinnytsia: VNTU, 2025. 91 p.

In Ukrainian. Bibliogr.: 19 titles; figures: 25.

This complex bachelor's thesis describes the development of a hardware and software system for remote audio transmission. The second part of the project focuses on creating a control application for the ESP32 microcontroller, enabling two-way audio streaming and remote audio file transfer. The application allows for connection to the device via a local network, real-time audio reception and playback or transmission, sending audio files, and managing the file list on the microcontroller's SD card. Support for the UDP protocol for streaming and the TCP protocol for file exchange was implemented.

The MAUI platform was used to develop the application, allowing for the creation of cross-platform programs using the C# programming language and the .NET platform.

This work demonstrates the development of software for controlling IoT devices, specifically creating applications for obtaining information from microcontrollers and sensors.

Keywords: IoT, .NET, MAUI, C#, ESP32, remote audio transmission, network protocols, application.

ЗМІСТ

ВСТУП

На сьогоднішній час сфери застосування мікроконтролерів стрімко зростаюсь. Вони активно використовуються в промисловості, будівництві, охоронних системах, медицині, екологічному моніторингу, автомобільній електроніці та багатьох інших галузях. Зі збільшенням кількості систем з вбудованими мікроконтролерами тому також зростає і необхідність в зручному та ефективному управлінні цими системами, що призводить до появи нових і удосконаленні вже існуючих рішень щодо обміну інформацією та розробки програмного забезпечення.

Актуальність теми полягає у створенні сучасного застосунку для можливості передачі інформації у локальній мережі з використанням таких технологій, як протоколів UDP, TCP, HTTP, використання платформи .NET MAUI та розробці сучасних та зручних інтерфейсів з використанням мови розмітки XAML для можливості ефективного керування та обміну інформації з мікроконтролерами, зокрема ESP32.

Метою роботи є створення застосунку для керування ESP32, який дозволяє забезпечити двосторонній аудіостримінг через протокол UDP, обмін аудіофайлами по TCP, а також управління пристроєм через HTTP-запити.

Об'єктом дослідження є процес взаємодії між клієнтським застосунком та мікроконтролером ESP32 у локальній мережі.

Предмет дослідження — програмні засоби реалізації аудіообміну, передачі файлів та керування мікроконтролером.

З**адачі роботи**:

— дослідження архітектури обміну даними між MAUI-застосунком і ESP32 через HTTP, UDP і TCP протоколи;

— реалізація механізму підключення до ESP32 та надсилання HTTP-запитів для управління режимами роботи;

— розроблення функціональності для прийому та передачі аудіопотоку по UDP у режимі реального часу;

— реалізація можливості передачі аудіофайлів на ESP32 через TCP та керування файлами (відтворення, видалення).

Апробація результатів комплексної бакалаврської квалафікаційної роботи була проведена на конференції "LIV Всеукраїнська науково-технічна конференція факультету інформаційних технологій та комп'ютерної інженерії (2025)" [1]:

Дацюк В. В., Бондарчук О. Б., Богомолов С. В., Мартинюк Т. Б. Дистанційна передача аудіо в IoT — Режим доступу: https://conferences.vntu.edu.ua/index.php/all-fitki/all-fitki-2025/paper/view/24413

1 АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ТА ПІДХОДІВ

1.1 Огляд сучасних рішень для дистанційного передавання аудіо

В сучасному світі дистанційне передавання аудіо використовується усюди — від бізнесу й освіти, до радіозв’язку, охоронних систем, побутового застосування. Тому на сьогоднішній час існує велика кількість рішень, як комерційних, так і відкритих, які пропонують свій набір функціональних можливостей для таких задач.

Для розробки зручного, ефективного та працездатного застосунку для передачі аудіоданих між телефоном та мікроконтролером ESP32, буде доцільно розглянути схожі за функціональністю рішення та проаналізувати їх можливості, роботу та реалізацію головної задачі — потокової передачі аудіо.

Серед комерційних продуктів, що підтримують передачу аудіо в реальному часі, одним із найвідоміших є TeamViewer. Хоча основна функція програми — віддалене керування, вона також дозволяє передавати аудіо з віддаленого комп’ютера на локальний. Це корисно, наприклад, під час технічної підтримки, коли необхідно чути системні звуки або звуковий супровід програм. Функція доступна переважно на Windows-платформах, вимагає активації відповідних параметрів, зокрема опції «Play computer sounds and music» (рис. 1.1) [2].

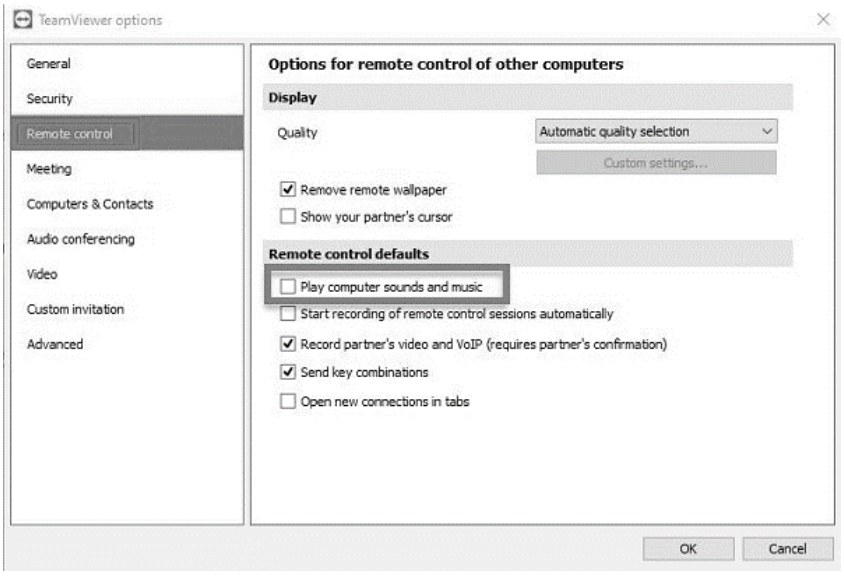

Рисунок 1.1 — Вікно налаштувань TeamViewer

Іншим комерційним рішенням для передачі аудіо даних між пристроями є програма AudioRelay [3], яка дозволяє передавати аудіо між ПК та Android-пристроями через Wi-Fi або USB в реальному часі. Для роботи програми її потрібно встановити на комп’ютер на телефон, після чого зв’язати їх (рис. 1.2).

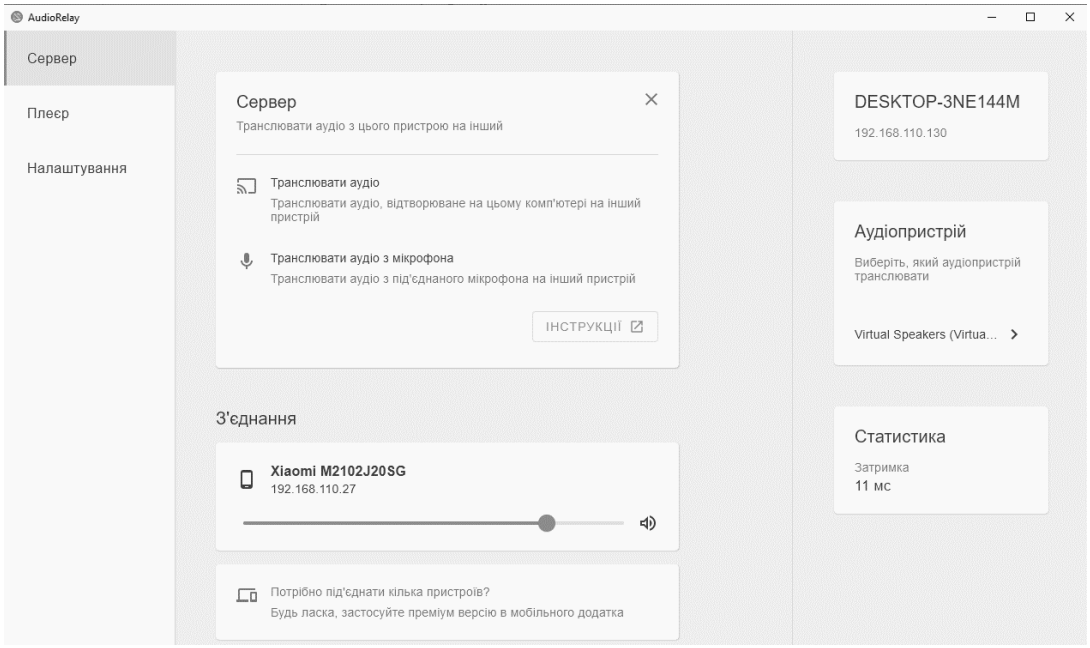

Рисунок 1.2 — Відображення під’єднаного пристрою

Програма дозволяє використовувати телефон для віддалений динамік для відтворення усього захопленого аудіо з комп’ютеру, а також використання телефону, як віддаленого мікрофону. При використанні AudioRelay комп’ютер використовується у ролі сервера, а телефон — клієнта.

До відкритих та кросплатформенних рішень належить SonoBus — застосунок з відкритим вихідним кодом, який дозволяє передавати аудіо з низькою затримкою між пристроями через локальну мережу або Інтернет. Програма підтримує платформи Windows, macOS, Linux, iOS та Android [4] та використовується для спілкування. Для цього спочатку потрібно створити групу (рис. 1.3) та під’єднатись до неї з іншого пристрою (рис. 1.4).

Прикладом окремого мобільного застосунку для передачі аудіо по мережі є LiveVoice [5] — програма, яка використовується для трансляції аудіо та орієнтована на конференції або масові заходи. Для цього потрібно у вікні браузера

створити подію (рис. 1.5), після чого створити окремий голосовий канал з потрібною назвою (рис. 1.6).

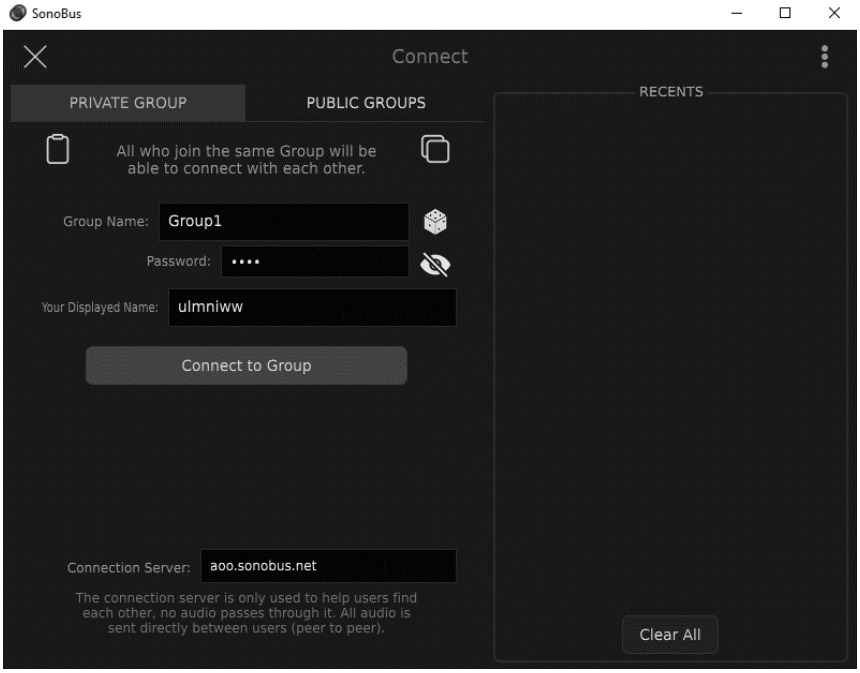

Рисунок 1.3 — Вікно створення групи

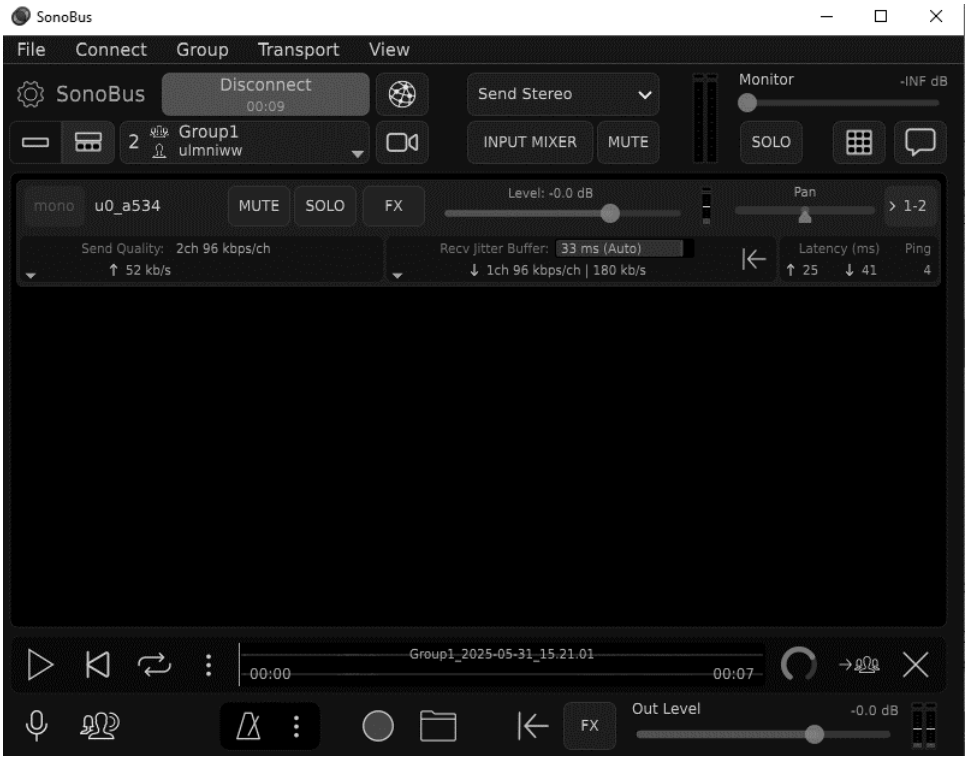

Рисунок 1.4 — Вікно групи

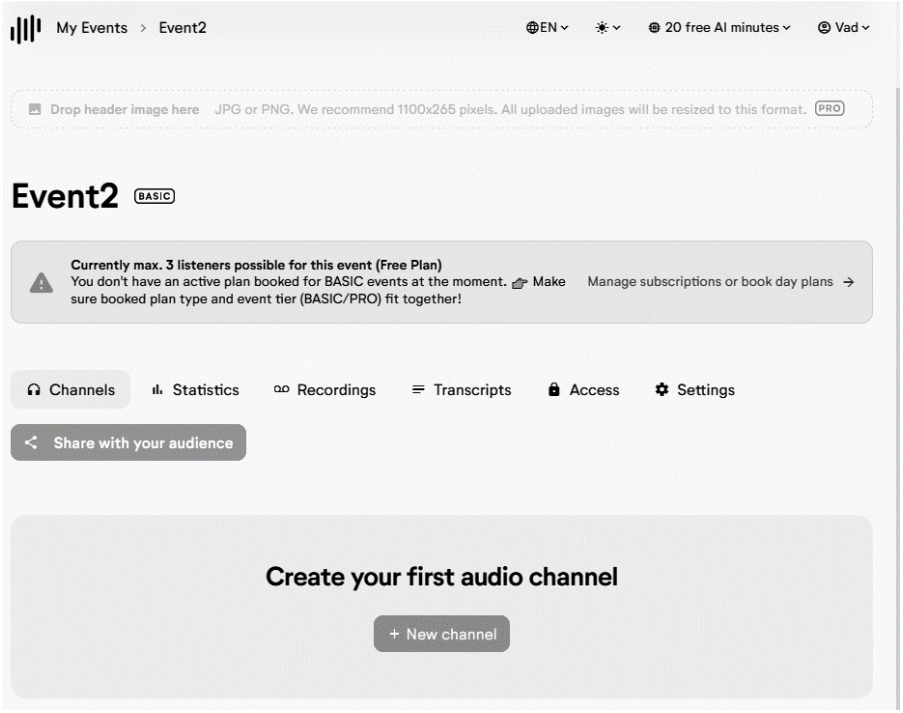

Рисунок 1.5 — Вікно з створенною подією

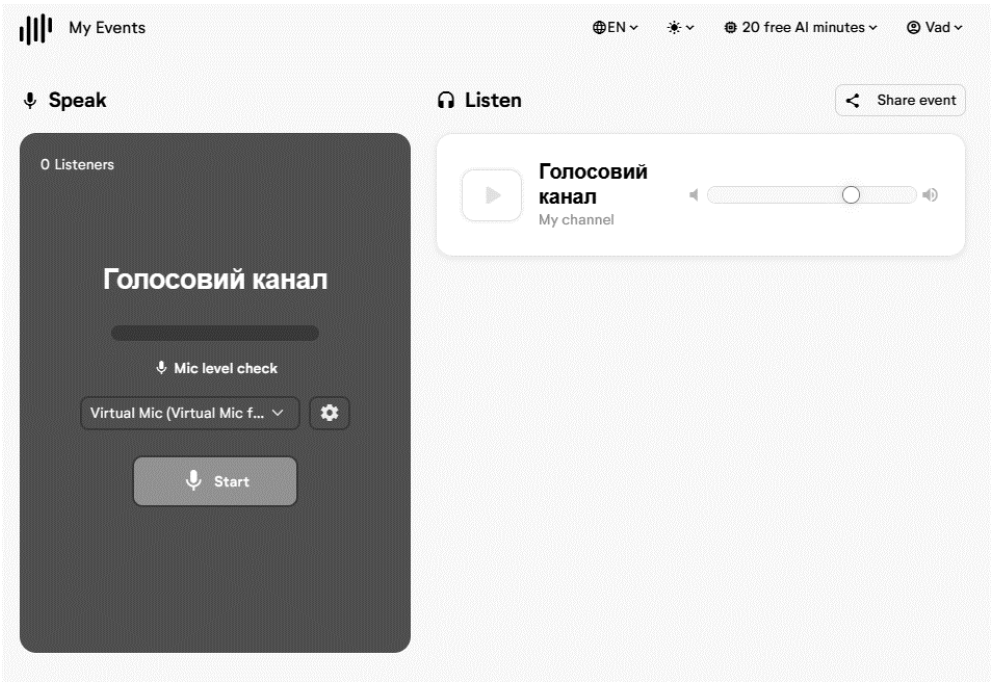

Рисунок 1.6 — Вікно голосового каналу

Після створення голосового каналу до нього можна під'єднатись використовуючи код події або QR-код. Вигляд вікна мобільного додатку показано на рисунку 1.7.

Рисунок 1.7 — Вікно голосового каналу в мобільному додатку

Окрім готових додатків прикладом рішення для дистанційного передавання аудіо також може бути функція Live Listen на операційній системі iOS. Live Listen дозволяє використовувати iPhone як віддалений мікрофон, який передає аудіо на сумісні пристрої для прослуховування аудіоданих, таких як AirPods. Ця функція переважно орієнтована на людей із порушенням слуху, але її також можна застосовувати в побутових ситуаціях [6].

Крім того, існує велика кількість програмних засобів як для мобільних пристроїв, так і для комп’ютерів, що забезпечують голосове спілкування на відстані. Наприклад, одним із найпоширеніших сервісів є Discord, основною функцією якого є передавання аудіо в реальному часі з мінімальними затримками між пристроями. Також можливість віддаленого спілкування надає більшість програм для обміну текстовими повідомленнями, наприклад Telegram або Viber.

1.2 Аналіз функціональних можливостей та обмежень

Розглянуті рішення для дистанційного передавання аудіо показують широкий спектр функціональних можливостей, проте кожне з них має як переваги, так і обмеження, які визначають сферу їх застосування.

TeamViewer, хоча й не є спеціалізованим аудіосервісом, підтримує передачу системного аудіо під час сеансів віддаленого керування. Основна перевага цього підходу це можливість кращої технічної підтримки або демонстрації програмного забезпечення. Водночас аудіофункціональність є допоміжною, не підтримує налаштування звуку, обробку сигналу чи режим реального часу з низькою затримкою.

Інший комерційний застосунок, AudioRelay, орієнтований на передачу аудіо між комп’ютером і мобільним пристроєм, тому має набагато більші можливості для керування передачею аудіо. Окрім статистичної інформації (рис. 1.8) він також має перелік налаштувань, таких як якість звуку, розмір буферу та налаштування управління додатком (рис. 1.9).

Як згадувалось раніше, застосунок може використовуватись, як віддалений мікрофон і в такому випадку передавати звук з телефону на комп’ютер, або як віддалений динамік, отримуючи захоплений з екрану комп’ютера звук. При роботі у режимі динаміку додаток на телефоні відтворює звук з програм на комп’ютері, а у режимі мікрофону — передає захоплений звук з телефону на комп’ютер. Обидва режими також мають свої налаштування звуку. Наприклад при передачі звуку з мікрофону телефона можна змінювати режими мікрофону, його гучність, шумоподавлення. Застосунок SonoBus, на відміну від попередніх, є гнучким аудіосервером peer-to-peer типу, тобто такий, в якому кожен користувач має рівні права та не потребує серверу для зв’язування між собою, що дозволяє створювати групові аудіосесії з високою якістю та низькою затримкою.

Додаток є кросплатформеним, підтримує можливість як монокальної, так і багатоканальної передачі звуку, налаштування підсилення звуку, можливість запису розмови. Окрім цього, SonoBus має вбудований еквалайзер з великою

кількістю налаштувань (рис. 1.10).

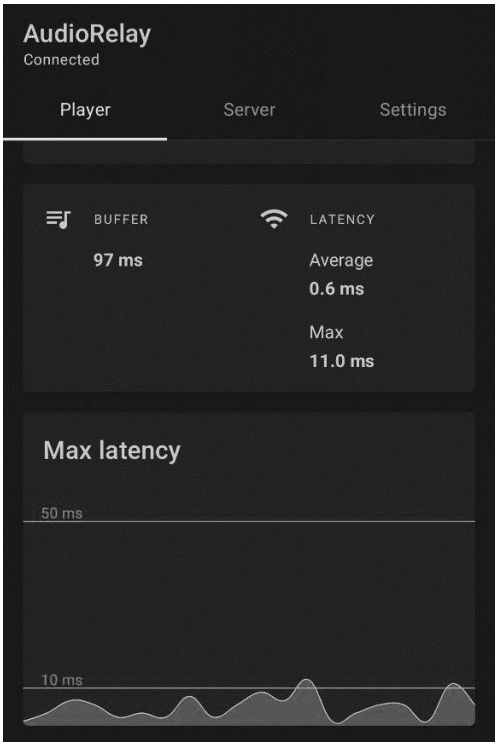

Рисунок 1.8 — Відображення статистики у вікні додатку

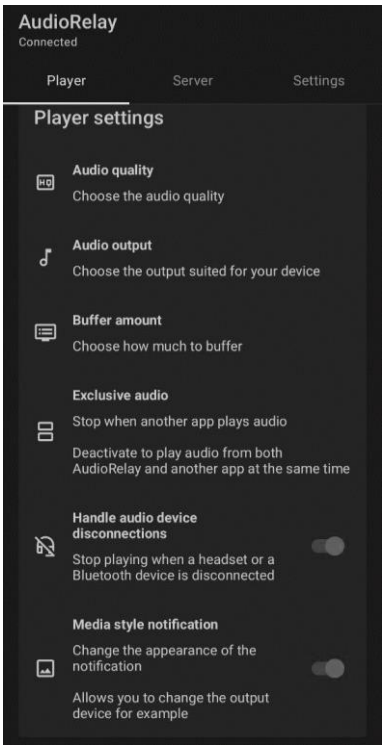

Рисунок 1.9 — Налаштування додатку

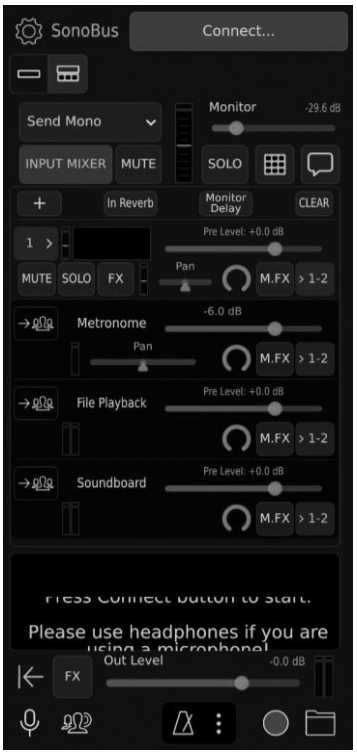

Рисунок 1.10 — Еквалайзер SonoBus

Проте SonoBus не підтримує шифрування, що є важливим аспектом при роботі через Інтернет.

LiveVoice є прикладом професійного, але вузькоспеціалізованого рішення. Програма орієнтована на аудіотрансляції під час подій, конференцій або масових заходів, тому не має гнучкого налаштування звуку. Також програма працює виключно через внутрішні сервери, що вказує на необхідність постійного підключення до Інтернету.

Інші застосунки, такі як Discord, або програми для текстового обміну повідомленнями з можливістю двінку, не мають великої кількості налаштувань звуку, оскільки орієнтовані на масового користувача.

Також інше рішення від компанії Apple, розглянута функція Live Listen операційної системи iOS, має вузьке цільове призначення — допомогу людям із порушенням слуху. Вона дозволяє транслювати аудіо з мікрофона iPhone на навушники типу AirPods у режимі реального часу. Головним обмеженням є апаратна залежність, оскільки доступна така функція тільки на пристроях фірми Apple.

1.3 Недоліки рішень та можливості їх покращення

Аналізуючи переваги та сфери використання наведених у попередніх підрозділах застосунків можна дійти висновку, що існуючі рішення для дистанційної передачі аудіо також мають і низку недоліків.

Одним з недоліків є обмеженість платформ, які підтримуються додатками. Деякі програми працюють виключно в межах певної операційної системи або екосистеми. Наприклад, Live Listen доступний лише на пристроях Apple, а TeamViewer має повну підтримку функцій аудіопередачі переважно на Windows. Це ускладнює використання таких рішень в ситуаціях, де необхідна взаємодія між пристроями на різних платформах.

Крім того, більшість програм потребує обов’язкового підключення до зовнішнього сервера або постійного доступу до Інтернету. Наприклад, такі рішення, як LiveVoice або TeamViewer вимагають для своєї роботи інтернет з’єднання. Таке обмеження виключає автономну роботу в умовах локальної мережі без зовнішнього з’єднання.

Отже, аналіз сучасних рішень для дистанційної передачі аудіо, таких як TeamViewer, AudioRelay, SonoBus, LiveVoice, Live Listen та інших, показує, що вони пропонують велику кількість функціональних можливостей, але мають певні обмеження, які впливають на їх універсальність та ефективність. Основними недоліками є прив’язка до конкретних платформ, як у випадку з Live Listen, що обмежена продукцією компанії Apple, або TeamViewer. Крім того, багато рішень, таких як LiveVoice чи TeamViewer, потребують постійного підключення до Інтернету, що не дозволяє їх використання в автономних локальних мережах.

Для покращення існуючих рішень потрібно взяти до уваги кілька ключових аспектів. По-перше, забезпечення доступності програм на більшості існуючих платформ, включаючи Windows, macOS, Linux, iOS та Android. По-друге, підтримка роботи рішень в локальних мережах без необхідності підключення до зовнішнього сервера, що підвищить автономність і зручність використання в умовах обмеженого доступу до Інтернету. По-третє, впровадження налаштувань

якості звуку, таких як регулювання якості звуку, розміру буфера та шумоподавлення, а також, при необхідності, можливість забезпечення шифрування даних для захисту конфіденційності.

2 РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Вибір технологій та середовища розробки

На сьогоднішній день існує велика кількість технологій для роботи з мережою, розробки як кросплатформених, так і платформозалежних програм, передачі файлів і даних різного обсягу з різними затримками. Аналіз існуючих рішень для дистанційної передачі аудіо в реальному часі показав, що головними функціональними особливостями подібних застосунків є забезпечення мінімальної затримки передачі аудіо в реальному часі, якість звуку при відтворенні або записі, на яку впливають частота дискретизації та бітність кодування звуку, а також можливість обробки затримок отримання пакетів при нестабільному інтернет з’єднанні. Отже, при реалізації застосунку для дистанційної передачі та прийому аудіо в реальному часі слід обрати технології, які б могли забезпечити можливість стабільної передачі даних, обробку помилок, могли б надати доступ до низькорівневих API відповідних платформ для управління якістю звуку та його кодування. Також при обранні технологій потрібно враховувати обмеженість обчислювальних потужностей мікроконтролера, який повинен буде приймати аудіодані та відтворювати їх. Для цих цілей було обрано ESP32, оскільки він має вбудовані модулі Wi-Fi, можливість створення вебсерверу та достатні апаратні характеристики для можливості прийому та відтворення звуку і управління SD-картою.

Для реалізації застосунку, спочатку потрібно розглянути технології, безпосередньо пов’язані з передачею інформації в мережі, зокрема локальній, оскільки при розробці апаратно-програмного комплексу використовуються вбудовані в ESP32 модулі та апаратні можливості з відкриттям вебсерверу.

Для передачі будь-яких даних в мережі, зокрема аудіофайлів або аудіопотоку, потрібно визначити чіткі правила, за якими дані відправляються, приймаються та обробляються. На сьогодні основою для передачі даних в мережі є модель TCP/IP, яка стала стандартом для більшості мережевих технологій. Вона дозволяє різним пристроям, незалежно від географічного місця перебування та

платформи, обмінюватися даними. Загалом TCP/IP складається з чотирьох рівнів:

— прикладний, що використовується для роботи з програмами користувача;

— транспортний, що призначений для передачі даних між пристроями;

— міжмережевий, який відповідає за маршрутизацію пакетів в мережі;

— рівень доступу до середовища передачі, який використовується для фізичної передачі даних [7].

Для реалізації передавання аудіоданих в першу чергу потрібно розглянути транспортний рівень моделі TCP/IP, оскільки протоколи цього рівня використовуються для передачі інформації між пристроями. На трнаспортному рівні основними протоколами є UDP (User Datagram Protocol) та TCP (Transmission Control Protocol).

Протокол UDP — це протокол, який не встановлює з’єднання перед передачею даних і не перевіряє, чи дійшли пакети до отримувача. Його робота фактично зводиться до мультиплексування, тобто об’єднання даних у потік, і демультиплексування — розподілу даних на приймальному боці [8]. Завдяки відсутності перевірки помилок та доставки до отримувача він, найчастіше, використовується при потоковій передачі, де втрата декількох пакетів не є критичною.

Протокол TCP — це протокол, що забезпечує надійне передавання даних у правильній послідовності [8]. Для передачі інформації він встановлює з'єднання між відправником та отримувачем і гарантує, що дані буде доставлено. Завдяки своїм характеристикам, цей протокол використовується у ситуаціях, коли важливо, щоб дані дійшли до отримувача в тому вигляді, в якому вони були надіслані, тобто він використовується для відправки файлів, текстових повідомлень.

Також на основі протоколів UDP та TCP було побудовано низку протоколів вищих рівнів, які використовують відповідні транспортні проколи. Наприклад, такі широко поширені протоколи, як WebRTC та HTTP побудовані на UDP та TCP відповідно.

Протокол WebRTC — це ще один протокол, який часто використовується в сценаріях потокової передачі даних та що базується на протоколі UDP. Він

дозволяє вебдодаткам та сайтам передавати аудіо або медіа-потоки без обов’язкового використання посередників [9]. Проте його недоцільно використовувати при роботі з мікроконтролерами, оскільки він потребує досить суттєвих обчислювальних потужностей, що не підходить для обмежених апаратних ресурсів мікроконтролерів, зокрема ESP32, особливо при потоковій передачі даних. Окрім самої передачі даних він включає в себе складні алгоритми шифрування, узгодження з'єднання, повнорної передачі пакетів, що потребує великої кількості оперативної пам'яті та часу процесора.

Отже, для обміну аудіоданими в реальному часі доцільно використовувати протокол UDP, який, як згадувалось вище, не потребує встановлення з’єднання, що забезпечує високу швидкість передачі даних завдяки відсутності механізмів підтвердження доставки пакетів і контролю їхньої послідовності, що робить його оптимальним протоколом для потокової передачі аудіо, де головним показником є низька затримка, а втрата окремих пакетів не критично впливає на якість сприйняття звуку, оскільки втрачені дані можуть бути компенсовані буферизацією, а також не вимагає складних алгоритмів обробки даних, що дозволяє використовувати його в ситуаціях обмежених апаратних можливостей.

Окрім потокової передачі аудіо в реальному часі, згідно технічного завдання, застосунок повинен також відправляти аудіофайли, для їх подальшого збереження на SD-карті, під’єднаній до мікроконтролера. Для цього буде використовуватись протокол TCP, який, як зазначалось вище, підходить для сценарії відправки файлів, оскільки підтримує повторну відправку пакетів при їх втраті та самостійно відслідковує з’єднання відправника та отримувача.

Оглянувши протоколи для передачі аудіо в реальному часі та аудіофайлів, доцільно буде проаналізувати технології для обміну повідомленнями між додатком та мікроконтролером. Одним з найбільш популярних протоколів, що використовуються в IoT для обміну повідомленнями між пристроями є MQTT (Message Queuing Telemetry Transport). MQTT — це легкий, з точки зору навантаження на апаратну частину, протокол, спеціально розроблений для IoT сфери, де важлива ефективність передачі даних у мережах із низькою пропускною

здатністю [10]. Він працює за моделлю "публікація-підписка", де пристрої обмінюються повідомленнями через брокер, який організовує передачу даних. MQTT дозволяє надсилати невеликі повідомлення в умовах обмежених ресурсів, що робить його популярним для систем із малою обчислювальною потужністю, таких як сенсори чи мікроконтролери. Наприклад, MQTT широко використовується в розумних будинках для керування пристроями, такими, як освітлення чи датчики температури [10]. Проте в даному проєкті використання MQTT є недоцільним із кількох причин. По-перше, MQTT потребує брокера — окремого сервера, який обробляє повідомлення між клієнтами, що додає складність до системи, адже необхідно налаштувати та підтримувати брокер, що не доцільно робити в межах локальних мереж без доступу до Інтернету. По-друге, для простих команд, таких як увімкнення/вимкнення чи зміна параметрів, MQTT не дає значних переваг порівняно з іншими протоколами, адже його сильна сторона — асинхронна обробка багатьох клієнтів.

Іншим протоколом, яким можна використовувати для обміну повідомленням є HTTP (Hypertext Transfer Protocol). Оскільки ESP32 має можливість створювати вебсервер це надає варіант обміну повідомленнями через HTTP-запити, використовуючи GET, POST або DELETE методи. Тому HTTP був обраний як основний протокол для передачі команд завдяки своїй простоті та вбудованій підтримці на мікроконтролері, що використовується у комплексі.

Ще одним аспектом, який потрібно враховувати при виборі технологій, є вибір цільової платформи та технологій для написання програмного забезпечення.

Основною платформою, яка підтримується застосунком, було обрано мобільні пристрої з операційною системою Android. Обрання цільовою платформою мобільних пристроїв зумовлено декількома факторами. По перше, мобільні пристрої забезпечують високу мобільність та зручність використання, що є важливим для систем, що пов’язані з віддаленим моніторингом і які потребують віддаленого керування. Іншим аспектом стало те, що усі сучасні смартфони оснащені вбудованими мікрофонами та динаміками високої якості, що є важливими критеріями для захоплення та відтворення звуку. Окрім наведених

вище аспектів слід зазначити, що сучасні мобільні пристрої мають достатній рівень продуктивності для можливості одночасного прийому аудіо і його відтворення в реальному часі та запису і відправки.

Для розробки мобільних додатків на сьогоднішній день використовується величезна кількість різних технологій, тому розглянемо лише найбільш поширені — Java, Kotlin, Flutter, React Native, .NET MAUI та Xamarin.

Мова програмування Java — це одна з найстаріших і найпоширеніших мов для розробки Android-додатків. Вона використовується в середовищі розробки Android Studio і має величезну кількість бібліотек, інструментів і документації для побудови застосунків на операційну систему Android. Java дозволяє створювати додатки, які максимально використовують можливості Android-пристроїв, наприклад, для обробки аудіо чи роботи з мережею. Вона забезпечує доступ до всіх API Android, що важливо для роботи з мікрофонами, динаміками чи Wi-Fi. Однак Java має суттєвий недолік — вона обмежена платформою Android, тому якщо в майбутньому знадобиться підтримка інших платформ, доведеться писати окремі додатки для кожної з них.

Більш сучасною альтернативою Java є Kotlin, яку Google офіційно підтримує як основну мову для операйної системи Android з 2017 року [11]. Kotlin має сучасний синтаксис та кращу підтримку сучасних парадигм програмування. Проте недоліком, як і в Java, є платформена залежність мови.

Окрім більш «традиційних» підходів до розробки додатків, існує декілька сучасних технологій, які дозволяють швидко будувати мобільні додатки з зручними інтерфейсами. До них відносяться Flutter та React Native.

Flutter — це кросплатформений фреймворк, розроблений компанією Google, який використовує мову Dart для створення додатків для таких операційних систем, як Android, iOS, Windows і macOS. Flutter дозволяє створювати додатки з зручним та сучасним інтерфейсом [12]. Проте, він не має доступу до низькорівневих платформозалежних API, що не підходить для створення застосунків, які повинні працювати з аудіо потоками в реальному часі.

Іншою сучасною технологією для розробки додатків на мобільні пристрої є

React Native — кросплатформений фреймворк, створений компанією Facebook, який використовує мову програмування JavaScript та побудований на іншому популярному фреймворкові — React, що призначений для створення динамічних односторінкових сайтів. React Native використовує вбудовані компоненти, що забезпечує кращу продуктивність порівняно з Flutter у деяких ситуаціях, проте все ще має з проблемами оптимізації для складних аудіозастосунків через залежність від JavaScript. Загалом React Native є популярною технологією для розробки мобільних додатків з використанням JavaScript, але для проєктів із високими вимогами до обробки аудіо в реальному часі може вимагати написання додаткових модулів, що значно ускладнює розробку.

Альтернативою до використання Java та Kotlin є платформа .NET, яка надає низку технологій, які можна використовувати для побудови мобільних застосунків.

Основною платформою для побудови кросплатформених застосунків був і залишається Xamarin — платформа від компанії Microsoft, яка дозволяє створювати додатки за допомогою C# і .NET . Перевагою Xamarin є те, що він забезпечує доступ до вбудованих API Android та iOS, що робить його придатним для роботи з аудіо та мережею. Проте, його продуктивність обмежена через застарілу архітектуру, а інтеграція з сучасними API не завжди ефективна. Окрім цього сама підтримка Xamarin була завершена 1 травня 2024 року. Натомість було випущено .NET MAUI, як еволюцію Xamarin [13].

Інша технологія платформи .NET — MAUI, що є еволюцією Xamarin. MAUI, як і його попередник, дозволяє створювати кросплатформені додатки для Android, iOS, Windows і macOS із єдиною кодовою базою, використовуючи C# та бібліотеки платформи .NET [14]. Також MAUI має доступ до вбудованих функцій платформ, що є важливим критерієм для роботи з аудіо та мережевими операціями, такими як потокова передача через UDP та відтворення аудіо в реальному часі. Він має кращу продуктивність порівняно з Xamarin і підтримує сучасні інструменти розробки, а також дозволяє створювати додатки, які легко масштабуються для використання на різних платформах без значних змін у коді.

Також, окрім офіційних технологій від компанії Microsoft, .NET має й

додаткові технології створення додатків, наприклад фреймворк Avalonia. Проте, на відмінну від офіційних платформ, вони мають гіршу можливість звертатись до API відповідних платформ або взагалі не підходять для побудови додатків, яким потрібно обмінюватись інформацією в реальному часі.

Отже, для реалізації застосунку для дистанційної передачі аудіо було обрано .NET MAUI. По-перше, як було розглянуто вище, він дозволяє створювати кросплатформені додатки з єдиною кодовою базою, що надає можливість розширення підтримуваних платформ на iOS, Windows чи macOS. По-друге, MAUI забезпечує прямий доступ до вбудованих API кожної з операційних систем, що важливо для роботи з аудіопотоками та мережевими операціями, такими як робота з сокетами. По-третє, MAUI — це сучасний фреймворк, який активно підтримується Microsoft, що гарантує регулярні оновлення, велику спільноту та доступ до сучасних бібліотек, на відміну від Xamarin. Порівняно з Flutter і React Native, MAUI дозволяє звертатись до вбудованих компонентів платформ. Також, хоча застосунок розробляється для операційної системи Android, можливість швидкої адаптації до інших платформ надає переваги використанню MAUI. До того ж, при використанні MAUI також використовується і платформа .NET, що дає змогу використовувавати величезну кількість вбудованої функціональності платформи з бібліотеки базових класів. Також важливим аспектом при обранні MAUI стало використання мови програмування C#. C# — це сучасна об’єктно-орієнтована кросплатформена мова програмування з відкритим вихідним кодом, що є основною мовою платформи .NET та має велику підтримку розробників та спільноти [15]. C# активно розвивається та оновлюється компанією Microsoft і має зручний та сучасний синтаксис.

Оскільки для розробки застосунку було обрано .NET MAUI, доцільно розглянути середовища розробки, що є сумісними з цією платформою. Основними IDE, що підтримують C# та MAUI, є Visual Studio та Rider.

Visual Studio — це офіційне середовище розробки від Microsoft, яке тісно пов’язано з платформою .NET та надає можливості редагування, відлагодження та збірки коду [16]. Воно забезпечує повноцінну підтримку MAUI включаючи

дизайнер інтерфейсів, інструменти налагодження, зручну роботу з емуляторами Android та інтеграцію з NuGet для керування залежностями. Крім того, Visual Studio дозволяє виконувати збірку і розгортання застосунку на фізичних пристроях та емуляторах, що важливо при тестуванні застосунків під час розробки.

Аналогом Visual Studio, який забезпечує схожу функціональність, є середовище розробки Rider від компанії JetBrains. Він також орієнтований на .NET-розробку та включає в себе усі переваги Visual Studio, проте підтримка .NET MAUI у Rider на теперішній час є обмеженою. Для роботи з MAUI проєктами потрібні додаткові налаштування, а підтримка Android-емуляторів та дизайнера інтерфейсів є менш стабільною порівняно з Visual Studio.

Отже, з огляду на вищезазначене, для розробки застосунку було обрано середовище розробки Visual Studio, оскільки воно забезпечує кращу сумісність з MAUI ніж Rider.

2.2 Архітектура програмного забезпечення

При розробці будь-яких застосунків першочергово необхідно визначити з яких компонентів повинна складатись програма та як ці компоненти повинні взаємодіяти між собою, тобто визначити архітектуру програми. В загальному, архітектура програмного забезпечення — це структура додатку, яка включає визначення взаємодії усіх компонентів програми.

Для реалізації застосунку було вирішено використовувати патерн MVVM (Model-View-ViewModel). Головною ідеєю шаблону MVVM є розділення бізнес-логіки програми від логіки користувацького інтерфейсу. Такий підхід дозволяє вирішити ряд проблем, зокрема спрощує тестування, обслуговування та подальший розвиток програми [17].

Шаблон MVVM виділяє три основні компоненти: модель, представлення і модель представлення, кожен з яких виконує певні, чітко окреслені задачі. Схема зв’язку між компонентами MVVM показана на рисунку 2.1.

Компонент представлення відповідає за структуру, макет і зовнішній вигляд

інтерфейсу користувача та виділяється в окремі XAML файли. XAML — це декларативна мова розмітки інтерфейсів, яка використовується в таких фреймворках, як MAUI та WPF [18].

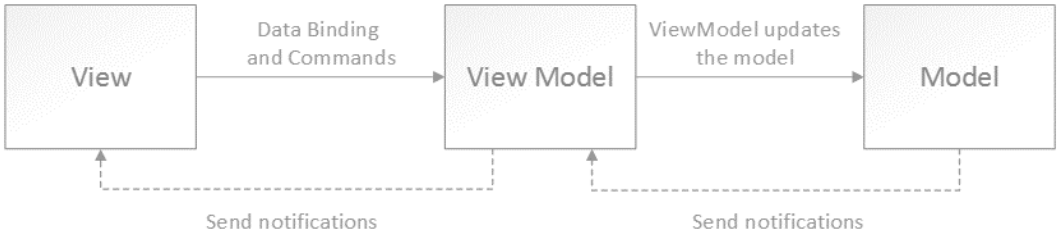

Рисунок 2.1 — Зв’язок між компонентами MVVM

Компонент представлення відповідає за структуру, макет і зовнішній вигляд інтерфейсу користувача та виділяється в окремі XAML файли. XAML — це декларативна мова розмітки інтерфейсів, яка використовується в таких фреймворках, як MAUI та WPF [18].

Отже, при розробці застосунку потрібно кожну сторінку додатку, згідно MVVM, описувати в окремому XAML файлі. Усього таких сторінок в застосунку виділено шість — AboutView, AppShell, AudioStreamingView, ESPView, FileTransferView та LogView. Кожен з цих представлень відповідє за відповідний користувацький інтерфейс та описує сторінку про додаток, основну оболонку інтерфейсу, сторінки для стримінгу, відправки файлів, підключення до ESP32 та сторінку для перегляду історії операцій.

Наступним компонентом в шаблоні MVVM є компонент моделі представлення. Він в собі реалізує властивості і команди, Тобто команди, які надаються моделлю представлення, визначають функціональні можливості, які надаються інтерфейсом користувача, а саме представлення визнає відображення цієї функціональності. Також саме цей компонент виступає в ролі моста між представленням та моделлю, оскільки кожна модель представлення надає данні з моделі в формі, яку може використовувати представлення.

З врахуванням вищеописаного, було створено окрему модель представлення для кожного компоненту представлення. Так, було реалізовано шімть моделей

представлення — AboutViewModel, AppShellViewModel, AudioStreamingViewModel, ESPViewModel, FileTransferViewModel та LogViewModel. Кожна з цих моделей представлення відповідає за надання даних та функціональності інтерфейсу програми.

Останнім компонентом шаблону MVVM є модель. Моделі — це класи, які інкапсулюють дані та бізнес-логіку програми. В рамках додатку було створено дві моделі — AudioFiles та ESP, які відповідають за файли та представлення ESP32 в коді. Також, було окремо виділено сервіси, які надають функціональність для логування виконаних операцій від час роботи застосунку, передачі аудіо в реальному часі та передачі аудіофайлів — AudioStreamingService, FileTransferService та LogService.

Використання патерну MVVM дозволяє зробити програму легко маштабованою, оскільки можна змінювати та розширювати функціональність кожного компонента програми, ніяк не впливаючи на інші.

Також слід детальніше оглянути спосіб реалізації мережевої взаємодії з ESP32. Оскільки застосунок повинен забезпечувати прийом та відтворення аудіо в реальному часі, важливо, щоб він підтримував асинхронну або паралельну обробку UDP-пакетів, що дозволяє уникнути затримок при передачі аудіоданих і забезпечити безперервне відтворення звуку.

Крім того, графічний інтерфейс користувача не повинен блокуватись під час виконання операцій прийому, передачі або відтворення аудіо. Тому для цього у відповідних сервісах потрібно застосовувати асинхронне програмування із використанням механізмів, які надає мова програмування C#.

2.3 Алгоритм роботи з аудіо

Основною функціональністю застосунку, згідно технічного завдання дипломного проєкту, є відправка та прийом аудіо в реальному часі та відправка аудіофайлів.

Для початку будь-якої передачі даних застосунок повинен отримати IP-адресу та порт ESP32, які потрібні для подальшого надсилання команд через HTTP.

Отримати їх можна двома шляхами — через ручне налаштування або автоматичне. При ручному налаштуванні користувач сам повинен вписати у відповідні поля IP-адресу та порт, після чого додаток спробує з’єднатись по ним з мікроконтролером. При автоматичному з’єднанні програма надсилає широкомовний UDP-пакет на порт 12345, який повинен прослуховуватись мікроконтролером. Після отримання відповіді, застосунок автоматично отримає усю необхідну інформацію для подальшої роботи.

Оскільки при обранні технологій розробки застосунку було обрано використання протоколу UDP для роботи з аудіо в реальному часі та протоколу TCP для передачі аудіофайлів, увесь алгоритм буде будуватись на використанні цих двох технологій мережевої передачі інформації.

Після встановлення з’єднання з ESP32 користувач може на вибір обрати або прийом захопленого звуку з мікрофону, під’єднаного до мікроконтролеру, або відправку звуку, захопленого з мікрофону пристрою, на якому запущений застосунок. При прийомі або передачі звуку використовуються багатопотоковість та асинхронність, для усунення зависань інтерфейсу та коректного отримання або відправки звуку.

При виборі опції отримання звуку з мікроконтролеру, на ESP32 надсилається HTTP-запит, який вказує про необхідність початку захоплення звуку з мікрофону. При отриманні HTTP відповіді про успіх, застосунок ініціалізує апаратне забезпечення пристрою для можливості відтворення отриманих пакетів з аудіо.

Після ініціалізації усіх необхідних для відтворення звуку модулів пристрою, починається зчитування UDP пакетів на порту 5555 та одночасно відбувається спроба відправити отримані дані на відтворення. Якщо буфер з звуком не порожній — з нього зчитуються усі наявні в даний час дані та відправляються на відтворення, видаляючись з буферу. Окрім цього, під час отримання UDP-пакетів відбувається перевірка їх застарілості. У випадку, коли пакет прийшов на 100 мс пізніше запланованого часу — він відкидається, що забезпечує захист від відставання звуку, особливо помітного в випадках поганого з’єднання.

При виборі опції відправки звуку, на мікроконтролер надсилається HTTP-

запит про початок відправки UDP-пакетів, отримавши який ESP32 повинен налаштуватись на прослуховування порту 5556 та проініціалізувати свої системи відтворення звуку. У випадку отримання HTTP-відповіді про успіх від мікроконтролеру, застосунок ініціалізує мікрофон пристрою та починає захоплення звуку та його відправку.

При зчитуванні звуку з мікрофону, отриманий звук записується в масив та в подальшому відправляється по UDP-сокету на мікроконтролер.

Вказані операції не можуть виконуватись одночасно. Тому, для можливості перемикання між ними, було реалізовано вимкнення поточної операції. При натисканні користувачем на відповідну кнопку, поточна операції завершується, надсилаючи токен завершення до потоків з відповідними задачами.

Окрім цього, застосунок повинен передавати аудіофайли. Для цього користувач повинен обрати відповідний файл, збережений на пристрої та натиснути на відповідку кнопку для його відправки.

Перед відправкою файлу до мікроконтролера надсилається HTTP-запит, який містить назву файла, що сповіщає про необхідність початку зчитування відповідного TCP порту. Після надходження відповіді від ESP32, вказаний файл відкривається та починається його потокова передача.

Для отримання списку вже збережених на SD-карті аудіофайлів, можливості їх видалення та відтворення, також використовуються відповідні HTPP-запити.

3 ІНТЕРФЕЙС І ФУНКЦІОНАЛЬНА РЕАЛІЗАЦІЯ

3.1 Розробка інтерфейсу застосунку

Інтерфейс користувача є важливою складовою застосунку, оскільки через нього відбувається взаємодія між користувачем та програмою. Тому при його розробці важливо врахувати зручність, інтуїтивність та логічність розміщення елементів, а також швидкий доступ до основних функцій програми.

Для створення інтерфейсу було використано мову розмітки XAML, яка дозволяє описувати зовнішній вигляд елементів декларативно, без потреби реалізовувати їх програмно. Така особливість значно спрощує структуру проєкту, оскільки бізнес-логіка інтерфейсу та його відображення чітко розділені, що повністю відповідає принципам патерну MVVM, який було обрано як архітектурну основу застосунку.

Інтерфейс програми поділено на три основні сторінки: головну сторінку, яка відповідає за підключення до ESP32, сторінку передачі або отримання потокового аудіо та сторінку керування файлами. Також було реалізовані дві додаткові сторінки — сторінку з описом програми та сторінку журналу подій. Кожна сторінка окрім свого опису в XAML форматі має також і відповідну їй модель представлення, яка містить логіку сторінки — обробники подій та дані, що відображаються на сторінці.

Кожна сторінка застосунку представлена багатоплатформеним користувацьким інтерфейсом ContentPage, який відображає одне вікно та являє собою певний макет, наприклад Grid або StackLayout [19], тому усі реалізовані вікна є елементами <ContentPage>.

Розробка інтерфейсу .NET MAUI починається з визначення візуальної ієрархії застосунку, для чого використовується клас Shell. Він призначений для створення основної навігації в додатку, для чого містить три об'єкта ієрархії — FlyuotItem, Tab та ShellContent.

Клас Shell застосунку містить кнопки верхньої стрічки додатку (рис. 3.1) та кнопки навігації між основними сторінка, які розташовані знизу (рис. 3.2).

Рисунок 3.1 — Кнопки верхньої стрічки додатку

Рисунок 3.2 — Кнопки навігації

В верхній стрічці додатку розташовані кнопки, які дозволяють перейти до допоміжних сторінок застосунку — сторінки про додаток та сторінки з журналом операцій. Для цього, до кожної з кнопок додано властивість Command, яка прив’язана до відповідних методів класу AppShellViewMode та додано елемент <ShellContent> з властивостями, які вказують на розташування відповідного представлення:

```
<Shell.ToolbarItems>
   <ToolbarItem Text="About" Command="{Binding About}"/>
   <ToolbarItem Text="Logs" Command="{Binding Logs}"/>
</Shell.ToolbarItems>
<ShellContent Title="About ESP Handler" ContentTemplate="{DataTemplate local:AboutView}" Route="AboutViewPage" />
<ShellContent Title="Logs" ContentTemplate="{DataTemplate local:LogView}" Route="LogViewPage" />
```

В нижній стрічці додатку розташовані кнопки, які відповідають за навігацію між основними сторінками застосунку. Вони дозволяють перейти до сторінки потокової передачі аудіо, сторінки відправки файлу та головної сторінки, для чого до кожного елементу нижніх вкладок навігації вказаний шлях до файлу представлення:

```
<TabBar Route="Main">
   <ShellContent Title="ESP" ContentTemplate="{DataTemplate local:ESPView}" Route="ESPView" />
```

```
  <ShellContent Title="FileTransfer" ContentTemplate="{DataTemplate local:FileTransferView}" Route="FileTransfer"/>
  <ShellContent Title="AudioStreaming" ContentTemplate="{DataTemplate local:AudioStreamingView}" Route="AudioStreaming" />
</TabBar>
```

Модель представлення оболонки інтерфейсу містить два методи для навігації по кнопкам верхньої панелі — AboutClick() та LogClick(), які призначені для переходу до відповідної сторінки:

```
public async Task AboutClick() {
  await Shell.Current.GoToAsync("//AboutViewPage"); }
public async Task LogClick() {
  await Shell.Current.GoToAsync("//LogViewPage"); }
```

Окрім цього, модель представлення також містить дві властивості — About та Logs типу ICommand, кожна з яких ініціалізується в конструкторі класу AppShellViewModel:

```
public AppShellViewModel() {
  About = new Command(() => _ = AboutClick());
  Logs = new Command(() => _ = LogClick()); }
```

Реалізувавши навігацію по сторінкам можна почати розробку інтерфейсу основних та допоміжних сторінок. Кожна з сторінок містить елемент <ContentPage.BindingContext>, який вказує про прив’язку моделі представлення до відповідного представлення та містить вкладений елемент, який вказує шлях, до відповідної моделі представлення.

Також, додатково створено допоміжний клас ESP, який відповідає за керування взаємодією з мікроконтролером ESP32 через HTTP-запити та автоматичне виявлення пристрою в локальній мережі за допомогою UDP-протоколу. Клас реалізує шаблон сінглтон для забезпечення єдиного екземпляра сервісу, доступного в усьому застосунку. Він зберігає конфігурацію з’єднання, таку як IP-адреса, порт і базовий URI, а також забезпечує методи для перевірки з’єднання, завантаження конфігурації пристрою та автоматичного пошуку ESP32.

Клас містить три приватні поля: _ipAddress типу string для зберігання IP-адреси ESP32, _port типу string для зберігання порту веб-сервера та _espHttpClient

типу HttpClient для виконання HTTP-запитів. Також клас містить п’ять публічних властивостей: Config типу ESPConfiguration для зберігання конфігураційних даних пристрою, Instance типу ESP, що призначена для отримання єдиного екземпляру класу, IpAddress, що повертає або встановлює значення _ipAddress, Port, що повертає або встановлює значення поля _port, BaseURI, яка повертає або встановлює базовий URI для HTTP-запитів у форматі "http://{IpAddress}:{Port}", EspConnection типу bool, що вказує на стан з’єднання з ESP32 та EspHttpClient, яка повертає ініціалізований об’єкт HttpClient.

Окрім цього, клас містить властивості AudioEndpoint, AudioFilesEndpoint і StreamingEndpoint, що повертають кінцеві точки API, додаючи відповідні шляхи до BaseURI.

Клас містить п’ять методів: приватний конструктор ESP(), метод UpdateBaseUri для оновлення базового URI, метод TestConnection для перевірки з’єднання, метод LoadESPConfiguration для завантаження конфігурації та метод AutoTestConnection для автоматичного пошуку ESP32. Детальніше розглянемо реалізації методів TestConnection, LoadESPConfiguration та AutoTestConnection, оскільки вони визначають основну функціональність класу.

Конструктор ESP є приватним і відповідає за початкову ініціалізацію сінглтону та викликає метод UpdateBaseUri для встановлення початкового значення BaseURI на основі значень _ipAddress і _port.

Метод TestConnection відповідає за перевірку з’єднання з ESP32 через HTTP-запит. Спочатку метод виконує GET-запит до кінцевої точки /api/status і якщо запит успішний, властивість EspConnection встановлюється в true:

```
var response = await EspHttpClient.GetAsync($"{BaseURI}/api/status");
EspConnection = response.IsSuccessStatusCode;
```

Метод LoadESPConfiguration відповідає за завантаження конфігураційних даних із ESP32. Спочатку метод викликає TestConnection для перевірки з’єднання і якщо з’єднання встановлено, метод виконує GET-запит до кінцевої точки /api/config:

```
var response = await EspHttpClient.GetAsync($"{BaseURI}/api/config");
```

Якщо запит успішний, вміст відповіді десеріалізується в об’єкт ESPConfiguration і присвоюється властивості Config:

```
if (response.IsSuccessStatusCode) {
   var content = await response.Content.ReadAsStringAsync();
   Config = JsonSerializer.Deserialize<ESPConfiguration>(content); }
```

Метод AutoTestConnection відповідає за автоматичне виявлення ESP32 у локальній мережі за допомогою UDP-протоколу. Спочатку метод створює UdpClient і вмикає широкомовну передачу:

```
using var udpClient = new UdpClient();
udpClient.EnableBroadcast = true;
```

Далі відправляється широкомовне повідомлення "ESP32 scanning" на порт 12345:

```
byte[] message = Encoding.UTF8.GetBytes("ESP32 scanning");
await udpClient.SendAsync(message, message.Length, "255.255.255.255", 12345);
```

Метод створює токен скасування з тайм-аутом 10 секунд і очікує відповідь від ESP32 і якщо відповідь отримана, вона декодується як рядок UTF-8:

```
using var cts = new CancellationTokenSource(TimeSpan.FromSeconds(10));
var receiveTask = udpClient.ReceiveAsync();
if (await Task.WhenAny(receiveTask, Task.Delay(Timeout.Infinite, cts.Token)) ==
receiveTask) {
   var result = receiveTask.Result;
string response = Encoding.UTF8.GetString(result.Buffer);
```

Метод розбиває відповідь на частини, беручи останню частину, яка, містить IP-адресу та порт у форматі IP:Port:

```
string[] parts = response.Split(" ");
string addressPart = parts.LastOrDefault();
```

Якщо частина адреси містить двокрапку, метод розбиває її на IP-адресу та порт:

```
if (!string.IsNullOrEmpty(addressPart) && addressPart.Contains(":")) {
```

```
string[] addressSplit = addressPart.Split(':');
IpAddress = addressSplit[0];
int.TryParse(addressSplit[1], out int port);
Port = port.ToString();
```

Далі викликається метод UpdateBaseUri для оновлення BaseURI. Метод встановлює EspConnection у true, викликає LoadESPConfiguration для завантаження конфігурації та повертає true. Якщо відповідь не містить валідної адреси, метод встановлює EspConnection у false і повертає false.

Після реалізації додаткового класу ESP, який буде використовуватись як представлення реального мікроконтролеру, потрібно створити сторінки додатку та їх моделі представлення.

Головна сторінка застосунку призначена для встановлення з’єднання з ESP32 (рис. 3.3). Інтерфейс сторінки включає два поля введення для IP-адреси та порту, кнопки для ініціалізації ручного або автоматичного підключення та текстовий індикатор стану з’єднання, який змінює колір залежно від стану підключення. При виникненні будь-яких помилок користувачу будуть відображатись сповіщення у вигляді спливаючих вікон. Окрім основних елементів, сторінка також відображає додаткову інформацію про пристрій після успішного підключення — ім’я, статус SD-карти та обсяг вільного місця.

Інтерфейс головної сторінки побудований з використання макету сітки — елемента <Grid>, який дозволяє розміщувати елементи у вигляді таблиці з визначеними рядками та стовпцями. Основні візуальні блоки сторінки, такі як інформація про статус підключення, поля вводу, та відображення конфігурації пристрою, обгорнуті в елементи <Frame>, які призначені для візуального виділення на сторінці.

У верхній частині сторінки розміщені назва сторінки, рамка з статусом поточного з’єднання та URI ESP32, які оновлюються динамічно під час роботи програми.

По середині сторінки розміщено рамку з налаштуваннями з’єднання, яка містить інформаційні текстові поля та два поля для вводу IP-адреси і порту ESP32, які необхідні при ручному підключенню до мікроконтролера. Усі елементи в рамці

розміщені в вертикально, завдяки елементу позиціонування <VerticalStackLayout>.

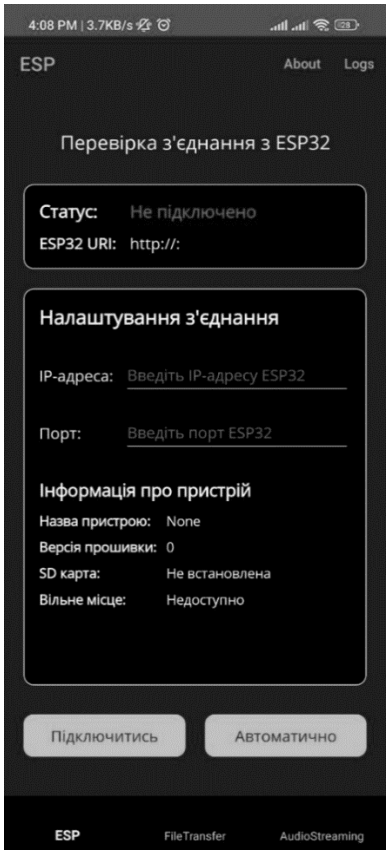

Рисунок 3.3 — Головна сторінка застосунку

У нижній частині сторінки розміщені дві кнопки <Button>, які прив’язані до відповідних команд у ViewModel:

```
<Button Grid.Column="0" Text="Підключитись" Command="{Binding ConnectCommand}" CornerRadius="10" FontSize="16" />
<Button Grid.Column="1" Text="Автоматично" Command="{Binding AutoConnectCommand}" CornerRadius="10" FontSize="16" />
```

Дані про ESP32 також змінюються динамічно після підключення, завдяки прив’язці до відповідних властивостей у ViewModel:

```
<Label Grid.Row="0" Grid.Column="0" Text="Назва пристрою:" FontSize="14" FontAttributes="Bold" />
<Label Grid.Row="0" Grid.Column="1" Text="{Binding ESPConfig.DeviceName}" FontSize="14" />
<Label Grid.Row="2" Grid.Column="0" Text="SD карта:" FontSize="14" FontAttributes="Bold" />
<Label Grid.Row="2" Grid.Column="1" Text="{Binding SDCardStatusText}" FontSize="14"
```

```
/>
        <Label Grid.Row="3" Grid.Column="0" Text="Вільне місце:" FontSize="14" FontAttributes="Bold" />
        <Label Grid.Row="3" Grid.Column="1" Text="{Binding SDCardFreeSpaceFormatted}" FontSize="14" />
```

Модель представлення головного вікна представлена класом ESPViewModel та реалізує інтерфейс INotifyPropertyChanged, який призначений для сповіщення представлення про зміну даних і необхідність оновлення. Клас містить чотири приватні поля — _connectionStatus типу стрічки для відображення статусу з’єдання, _connectionStatusColor типу Color для задання кольору статусу з’єднання, ESP типу ESP, яке призначене для зберігання сінглтону класу ESP, що відповідає за HTTP-взаємодію з мікроконтролером та _espConfig типу ESPConfiguration для збереження конфігурації пристрою.

До вищезгаданих полів реалізовано публічні властивості з підтримкою виклику OnPropertyChanged(), щоб забезпечити зв'язок із представленням. Окрім базових властивостей IpAddress, Port, URI, також створено такі властивості, як IpAddress типу стрічки для збереження IP-адреси ESP32, Port типу стрічки, для збереження порту вебсервера, URI типу стрічки для збереження повної адреси серверу, та SDCardFreeSpaceFormatted і SDCardStatusText для детального відображення інформації про SD-карту. Також у класі реалізовано дві команди типу ICommand — ConnectCommand та AutoConnectCommand, які призначені для обробки натискань на кнопки.

При натисканні на кнопки підключення, ручного чи автоматичного, буде асинхронно викликатись відповідний метод. При ручному підключенні викликається метод ConnectToESPByEndPoint, який призначений для підключення до ESP32. У ньому викликається метод LoadESPConfiguration() сінглтону класу ESP, який відправляє HTPP-запит для отримання конфігурації мікроконтролеру. У разі відповіді про успішну обробку запиту викликається метод UpdateConnectionStatus(), що призначений для зміни статусу на головному екрані, а також викликається метод OnPropertyChanged(), щоб сповістити про зміну властивостей SDCardStatusText та SDCardFreeSpaceFormatted:

```
private async Task ConnectToESPByEndPoint() {
   ConnectionStatus = "Підключення...";
   ConnectionStatusColor = Colors.Orange;
   bool success = await ESP.LoadESPConfiguration();
   UpdateConnectionStatus(success);
   if (success) {
      ESPConfig = ESP.Config;
      OnPropertyChanged(nameof(SDCardStatusText));
      OnPropertyChanged(nameof(SDCardFreeSpaceFormatted)); }
```

При автоматичному підключенню буде викликано метод AutoConnectToEsp(), який викличе метод AutoTestConnection() сінглтону класу ESP та виконає аналогічні дії до методу ConnectToESPByEndPoint().

Також клас ESPViewModel містить метод UpdateConnectionStatus(), який приймає параметр isConnected булевого типу, та призначений для встановлення надписів на головній сторінці застосунку.

Сторінка передачі та отримання потокового аудіо дозволяє користувачу керувати двонаправленою передачею аудіо в реальному часі. Інтерфейс сторінки містить кнопки для запуску та зупинки запису й відтворення, а також текстове поле для відображення поточного стану передачі. Елементи керування представлені трьома кнопками — «Почати прийом аудіо», «Почати передачу аудіо» та «Зупинити». Доступність кнопок залежить від поточної роботи додатку. При прийомі аудіопотоку кнопка «Почати передачу аудіо» блокується і активується кнопка «Зупинити». Після зупинки передачі дві інші кнопки стають доступними, а кнопка «Зупинити» блокується. Сторінка показана на рисунку 3.4.

Інтерфейс сторінки побудований з використанням вертикального макету <VerticalStackLayout>, а для візуального виділення блоків управління використано елементи <Frame>, аналогічно до головної сторінки застосунку. Основні елементи сторінки розміщено у двох блоках: статус трансляції та кнопки управління.

У верхній частині сторінки розміщено заголовок та текстовий індикатор поточного стану трансляції, прив’язаний до властивостей відповідної моделі представлення:

```
<Label Text="Керування аудіо стримінгом"  FontSize="24"  HorizontalOptions="Center"
Margin="0,10,0,10"/>
```

```
<Frame BorderColor="Gray" Padding="10" Margin="0,0,0,10">
  <HorizontalStackLayout>
    <Label Text="Статус:" FontSize="18" Margin="0,0,0,10"/>
    <Label Text="{Binding StatusText}" FontSize="18" Margin="10,0,0,10"
        TextColor="{Binding ConnectionStatusColor}" />
  </HorizontalStackLayout>
</Frame>
```

Нижче розміщено три кнопки <Button>, кожна з яких пов’язана з відповідною командою-обробником:

```
<Button Text="Почати прийом аудіо" Command="{Binding StartReceivingCommand}" IsEnabled="{Binding CanStartReceiving}" Margin="0,10,0,5" HeightRequest="50"/>
<Button Text="Почати передачу аудіо" Command="{Binding StartTransmittingCommand}" IsEnabled="{Binding CanStartTransmitting}" Margin="0,5,0,5" HeightRequest="50"/>
<Button Text="Зупинити" Command="{Binding StopCommand}" IsEnabled="{Binding CanStop}" Margin="0,5,0,10" HeightRequest="50"/>
```

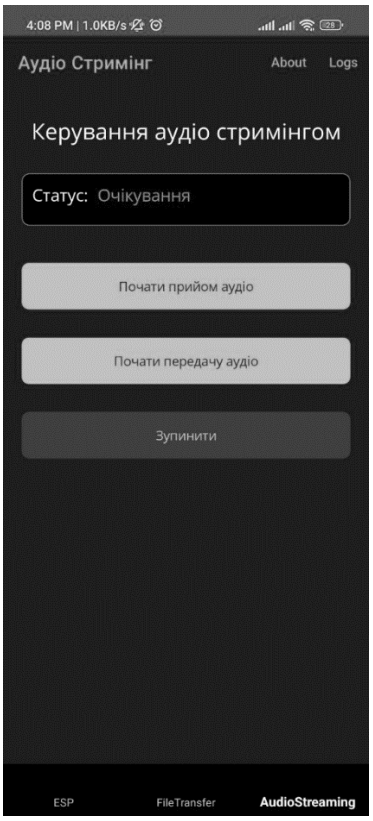

Рисунок 3.4 — Сторінка передачі аудіо в реальному часі

Сторінка потокової передачі аудіо пов’язана з моделлю представлення AudioStreamingViewModel, яка керує станом текстових полів та визначає функціональність відповідних кнопок.

Клас AudioStreamingViewModel працює з сервісом AudioStreamingServices,

використовуючи його функціональність для можливості запису, передачі, отримання та відтворення аудіо по мережі.

Загалом клас містить п’ять приватних полів — _audioStreamingServices типу AudioStreamingServices, _isReceiving булевого типу, _isTransmitting булевого типу, _statusText типу стрічки та _connectionStatusColor типу Color. Поле _audioStreamingServices призначене для розкриття функціональності сервісу AudioStreamingServices, поля _isReceiving та isTransmitting використовуються для інформування представлення про поточний стан роботи, поле _statusText використовується для відображення поточного режиму роботи в текстовому вигляді, а поле _connectionStatusColor — для зміни кольору тексту статусу. До усіх полів, окрім _audioStreamingServices також створено властивості, для можливості інформування представлення про зміни в роботі.

Окрім цього, створено три приватні поля булевого типу CanStartReceiving, CanStartReceiving та CanStop, які відповідають за доступність кнопок на сторінці та три команди, які прив’язані до представлення сторінки.

Кожній з створених команд в моделі представлення було передано відповідний метод, який повинен викликатись при натисненні на певну кнопку.

Також використано конструктор класу, в якому виконується прив’язка методів до команд та ініціалізації поля _audioStreamingServices:

```
public AudioStreamingViewModel() {
   _audioStreamingServices = new AudioStreamingServices();
   StartReceivingCommand = new Command(async () => await StartReceiving());
   StartTransmittingCommand = new Command(async () => await StartTransmitting());
   StopCommand = new Command(async () => await Stop()); }
```

Загалом клас містить шість методів, три з яких виступають обробниками кнопок. Метод StartReceiving() призначений для обробки отримання аудіо, асинхронно викликаючи метод UdpEspRequest() класу AudioStreamingServices. При успішному запуску операції він встановлює полю IsReceiving значення true та викликає методи UpdateButtonsState() та UpdateStatusText().

Метод StartTransmitting() використовується для обробки початку відправки звуку до мікроконтролера та працює аналогічно до обробника отримання аудіо,

викликаючи метод StartMicrophoneStreaming() класу AudioStreamingServices.

Метод Stop() використовується для обробник кнопки зупинки виконання поточної операції асинхронно зупиняючи її.

Інші три методи виступають допоміжними. Так, метод ShowError() приймає параметром стрічку помилки та призначений для відтворення її у спливаючому вікні в разі необхідності, а метод UpdateStatusText() призначений для оновлення тексту поля статусу операцій та його кольору.

Метод UpdateButtonsState() призначений для сповіщення представлення про зміну режиму роботи програми, викликаючи метод OnPropertyChanged().

Наступна сторінка — сторінка керування файлами, яка призначена для вибору аудіофайлів з пристрою, їх передачі на ESP32, а також керування файлами на SD-карті мікроконтролера, а саме можливість їх перегляду, відтворення та видалення. Інтерфейс включає кнопку для вибору файлу, вивід назви обраного файлу, а також таблицю зі списком файлів, де кожна позиція має кнопки для відтворення або видалення. Сторінку керування файлами зображено на рисунку 3.5.

Інтерфейс сторінки побудований з використанням макету сітки — елемента <Grid>, який дозволяє розміщувати елементи у вигляді таблиці з визначеними рядками.

Основні блоки інтерфейсу, такі як статус з’єднання, область вибору файлу, кнопка відправки та список файлів, обгорнуті в елементи <Frame> для візуального виділення.

У верхній частині сторінки, як і в попередніх, розміщено заголовок сторінки

Нижче розташовано рамку зі статусом з’єднання, яка містить текстові поля для відображення поточного стану передачі файлів.

У середній частині сторінки розміщено рамку для вибору файлу, яка містить текстові поля для відображення імені обраного файлу та кнопку для виклику вікна вибору файлу, а нижче рамки вибору файлу розміщено кнопку для відправки обраного файлу.

Рисунок 3.5 — Сторінка керування файлами

У нижній частині сторінки розташовано рамку зі списком файлів, що зберігаються на ESP32, реалізованим через елемент <CollectionView>. Кожен елемент списку відображає ім’я файлу та містить кнопки для відтворення та видалення:

```
        <Frame Grid.Row="4" BorderColor="LightGray" CornerRadius="10" Padding="15" VerticalOptions="FillAndExpand">
            <Grid>
                <Grid.RowDefinitions>
                    <RowDefinition Height="Auto" />
                    <RowDefinition Height="*" />
                </Grid.RowDefinitions>
                <Label Grid.Row="0" Text="Файли на ESP" FontSize="18" FontAttributes="Bold" Margin="0,0,0,10" />
                <ScrollView Grid.Row="1">
                    <CollectionView ItemsSource="{Binding ESPFiles}">
                        <CollectionView.ItemTemplate>
                            <DataTemplate>
                                <Frame BorderColor="LightGray" Margin="0,5" Padding="10" CornerRadius="5">
                                    <Grid ColumnSpacing="10">
```

```
                        <Grid.ColumnDefinitions>
                            <ColumnDefinition Width="*" />
                            <ColumnDefinition Width="Auto" />
                            <ColumnDefinition Width="Auto" />
                        </Grid.ColumnDefinitions>
                        <Label Grid.Column="0" Text="{Binding FileName}"
VerticalOptions="Center" FontSize="16" />
                        <Button Grid.Column="1" Text="▶" Command="{Binding
Source={RelativeSource AncestorType={x:Type local:FileTransferViewModel}},
Path=PlayCommand}" CommandParameter="{Binding .}" WidthRequest="40" HeightRequest="40"
CornerRadius="20" Padding="0" FontSize="16" />
                        <Button Grid.Column="3" Text="🗑" Command="{Binding
Source={RelativeSource AncestorType={x:Type local:FileTransferViewModel}},
Path=DeleteCommand}" CommandParameter="{Binding .}" WidthRequest="40"
HeightRequest="40" CornerRadius="20" Padding="0" FontSize="16" />
                    </Grid>
                </Frame>
            </DataTemplate>
        </CollectionView.ItemTemplate>
    </CollectionView>
</ScrollView>
</Grid>
</Frame>
```

Модель представлення сторінки передачі аудіофайлів представлена класом FileTransferViewModel, який реалізує інтерфейс INotifyPropertyChanged для сповіщення представлення про зміну даних. Клас містить п’ять приватних полів: _audioService типу FileTransferServices для взаємодії з сервісом передачі файлів, _esp типу ESP для зберігання сінглтону класу, що відповідає за HTTP-взаємодію з мікроконтролером, _connectionStatus типу стрічки для відображення статусу з’єднання, _fileName типу стрічки для зберігання імені обраного файлу, _result типу FileResult для зберігання результату вибору файлу та _espFiles типу ObservableCollection<AudioFile> для зберігання списку файлів на ESP32.

До полів _connectionStatus, _fileName та _espFiles реалізовано публічні властивості з підтримкою виклику OnPropertyChanged() для забезпечення зв’язку з представленням. Також реалізовано властивості IPAddressESP та PortESP, які прив’язані до відповідних полів сінглтону ESP. Властивість ESPFiles повертає колекцію файлів, що відображається у <CollectionView>.

Клас містить шість команд типу ICommand: SelectFileCommand для вибору файлу, UploadFileCommand для відправки файлу, PlayCommand для відтворення

файлу та DeleteCommand для видалення файлу. Для ініціалізації полів і команд використовується конструктор класу.

Метод SelectFile відповідає за вибір аудіофайлу з пристрою користувача. Спочатку метод створює об’єкт FilePickerFileType для фільтрації аудіофайлів:

```
var customAudioFileType = new FilePickerFileType(new Dictionary<DevicePlatform, IEnumerable<string>> {
    { DevicePlatform.Android, new[] { "audio/*" } }, });
```

Далі викликається діалог вибору файлу через FilePicker.PickAsync:

```
_result = await FilePicker.PickAsync(new PickOptions {
   PickerTitle = "Виберіть аудіофайл",
   FileTypes = customAudioFileType });
```

Якщо файл обрано, метод встановлює властивість FileName значенням імені файлу:

```
if (_result != null) { FileName = _result.FileName; }
```

Одним з головних методів є метод UploadFile(), що відповідає за відправку обраного файлу на ESP3. Спочатку метод перевіряє наявність з’єднання:

```
if (!_esp.EspConnection) {
   await Application.Current.MainPage.DisplayAlert("Помилка", "Немає з'єднання з ESP", "OK");
   return;}
```

Потім перевіряється, чи обрано файл:

```
if (_result == null) {
   await Application.Current.MainPage.DisplayAlert("Помилка", "Файл не обрано", "OK");
   return; }
```

Далі метод встановлює статус "Відправка файлу..." і викликає метод SendFileAsync сервісу _audioService:

```
ConnectionStatus = "Відправка файлу...";
var success = await _audioService.SendFileAsync(_result.FullPath);
```

Якщо відправка успішна, метод встановлює статус "Файл успішно надіслано" і оновлює список файлів:

```
if (success) {
   ConnectionStatus = "Файл успішно надіслано";
   await RefreshFileList(); }
```

Якщо відправка не вдалася, метод встановлює статус "Помилка відправки файлу":

```
else {
   ConnectionStatus = "Помилка відправки файлу";}
```

У разі виникнення виключення метод встановлює статус із текстом помилки та відображає спливаюче вікно з помилкою.

Метод RefreshFileList відповідає за оновлення списку файлів, збережених на ESP32. Метод перевіряє наявність з'єднання та встановлює статус "Отримання списку файлів..." і викликає метод GetAudioFileListAsync сервісу _audioService у разі наявності підключення до мікроконтролеру. Потім він очищає поточний список файлів і додає нові файли до колекції ESPFiles, після чого встановлює статус із кількістю знайдених файлів.

Метод PlayAudioFile відповідає за відтворення аудіофайлу на ESP32. Спочатку метод перевіряє наявність з'єднання. Далі викликається метод PlayAudioFileAsync сервісу _audioService та у випадку, якщо відтворення не вдалося, метод відображає спливаюче вікно з повідомленням про помилку.

Метод DeleteAudioFile відповідає за видалення аудіофайлу з SD-карти ESP32. Приймає параметр file типу AudioFile, що представляє відповідний файл. Спочатку метод перевіряє наявність з'єднання та у разі його наявності відображається діалог підтвердження видалення і якщо користувач підтвердить видалення, викликається метод DeleteAudioFileAsync сервісу _audioService, в який передається ім'я аудіофайлу, та відбувається оновлення списку файлів:

```
if (confirm) {
   var success = await _audioService.DeleteAudioFileAsync(file.FileName);
if (success) {
   await RefreshFileList(); }
```

Якщо видалення не вдалося, метод відображає спливаюче вікно з

повідомленням про помилку.

Дві інші сторінки — сторінку "Про додаток" та сторінку з журналом операцій, зображено на рисунках 3.6 та 3.7 відповідно. Вони призначені для відображення інформації про сам додаток або його роботу. Сторінка про додаток описує основні функції програми та короткий опис її роботи. Сторінка журналу містить інформацію про поточний стан програми та відображає історію виконання усіх операцій пов’язаних з передачею або прийомом аудіо в реальному часі, з’єднанням з ESP32 та передачею файлів. Ця сторінка може використовуватись при тестуванні додатку та його відлагодженню, а також є корисною для користувачів, оскільки дає детальну інформацію про можливі помилки під час виконання певних функцій.

Рисунок 3.6 — Сторінка з описом додатку

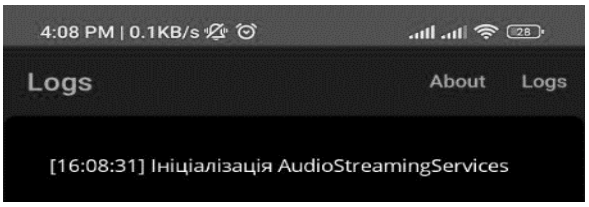

Рисунок 3.7 — Сторінка журналу операцій

Сторінка журналу операцій призначена для відображення історії операцій, які генеруються застосунком під час взаємодії з мікроконтролером ESP32. Інтерфейс сторінки включає кнопку для повернення на головну сторінку та історію виконання операцій, яка представлена у вигляді прокручуваного списку.

Інтерфейс сторінки побудований з використанням елемента <VerticalStackLayout>, який розміщує елементи послідовно зверху вниз. Список історії обгорнутий в елемент <Frame> для візуального виділення та міститься всередині <ScrollView> для забезпечення прокручування.

У верхній частині сторінки розміщено кнопку для повернення на головну сторінку, а нижче розташовано рамку, яка містить прокручуваний список історії операцій, реалізований через елемент <CollectionView>, що дозволяє прокручувати список історії.

Елемент <CollectionView>, що відповідає за відображення історії операцій, прив’язаний до властивості Logs моделі представлення, а кожен елемент списку відображається як текстовий рядок через <Label>.

Модель представлення сторінки журналу подій представлена класом LogViewModel. Клас містить дві властивості: Logs типу ReadOnlyObservableCollection<string> для відображення історії та команду GoBack типу ICommand для обробки повернення на головну сторінку.

Властивість Logs повертає колекцію логів, отриману з сервісу LogService.Instance.Logs, а команда GoBack призначена для виклику асинхронного методу GoBackToMain():

```
public LogViewModel() {
   GoBack = new Command(() => _ = GoBackToMain()); }
```

Метод GoBackToMain() відповідає за перехід до головної сторінки застосунку:

```
public async Task GoBackToMain() {
   await Shell.Current.GoToAsync("//Main"); }
```

Для цієї моделі представлення було створено додатковий сервіс LogService, який забезпечує основну функціональність для можливості збереження та додавання історії. Клас містить чотири приватні поля: _logs типу ObservableCollection<string> для зберігання списку історії, _maxLogCount типу int для обмеження максимальної кількості логів, _lock типу object для синхронізації доступу до колекції при багатопотоковості та статичне поле Instance типу LogService для реалізації сінглтона. Також клас містить одну публічну властивість Logs типу ReadOnlyObservableCollection<string> для надання доступу до логів лише для читання. Також він містить три методи: приватний конструктор LogService(), метод Log для додавання повідомлень до журналу та внутрішній метод LogInternal для безпосереднього оновлення колекції.

Основний метод сервісу — метод Log() — відповідає за додавання нового повідомлення до журналу. Приймає один параметр message типу string, який містить текст повідомлення, і не повертає значення.

```
public void Log(string message) {
   if (MainThread.IsMainThread) {
      LogInternal(message); }
   else {
      MainThread.BeginInvokeOnMainThread(() => LogInternal(message)); }}
```

Метод LogInternal() відповідає за безпосереднє додавання повідомлення до колекції _logs. Приймає один параметр message типу string і нічого не повертає. Виконання методу синхронізується через блокування об’єкта _lock для забезпечення потокобезпечності. Додатково метод перевіряє, чи досягнуто максимальної кількості записів і якщо колекція містить 100 або більше елементів, найстаріший запис видаляється:

```
private void LogInternal(string message){
   lock (_lock) {
```

```
if (_logs.Count >= _maxLogCount)
    _logs.RemoveAt(0);
_logs.Add($"[{DateTime.Now:HH:mm:ss}] {message}"); }}
```

Сторінка "Про додаток" призначена для надання користувачу загальної інформації про застосунок — його функціональність та інструкцію з використання. Інтерфейс сторінки включає кнопку для повернення на головну сторінку та текстові блоки з описом додатку, його основних функцій і покроковою інструкцією користування.

Інтерфейс сторінки побудований з використанням елемента <ScrollView>, який містить вертикальний макет <VerticalStackLayout> для послідовного розміщення елементів зверху вниз. Основні текстові блоки обгорнуті в елемент <Frame> для візуального виділення.

У верхній частині сторінки розміщено кнопку для повернення на головну сторінку, а нижче аналогічно до інших сторінок розташовано заголовок сторінки.

Далі розміщено рамку, яка містить інформацію про функції застосунку та інструкції з використання, оформлені у вигляді вертикального списку за допомогою елемента <VerticalStackLayout>, а елемент <ScrollView> надає можливість прокрутки вмісту, якщо текст перевищує розмір екрана.

Модель представлення сторінки "Про додаток" представлена класом AboutViewModel, що містить одну команду GoBack типу ICommand для обробки повернення на головну сторінку, яка містить метод GoBackToMain().

Метод GoBackToMain() відповідає за перехід до головної сторінки застосунку через Shell-навігацію:

```
public async Task GoBackToMain() {
    await Shell.Current.GoToAsync("//Main"); }
```

Оскільки сторінка "Про додаток" не взаємодіє з динамічними даними, модель представлення не реалізує інтерфейс INotifyPropertyChanged і не містить логіки для обробки змін у даних. Уся інформація, яка відображається, є статичною та визначена безпосередньо в XAML.

3.2 Реалізація передачі аудіопотоку

Передача аудіопотоку в реальному часі є однією з ключових функцій застосунку, що дозволяє записувати аудіо з мікрофона пристрою користувача та передавати його на ESP32, а також відтворювати аудіопотік, отриманий із мікроконтролера. Для реалізації цих функцій потрібно реалізувати сервіс, що представлений класом AudioStreamingServices, який призначений для керування передачею аудіо в реальному часі, включаючи ініціалізацію, прийом, передачу та зупинку аудіопотоків. Клас використовує UDP-протокол для взаємодії з мікроконтролером, а також HTTP-запити для керування і включає можливості обробки помилок та збереження історії до журналу операцій.

При реалізації класу було визначено, що він повинен містити поля для збереження об’єктів UDP-клієнтів, кінцевої точки підключення, черги аудіобуферів, токенів скасування, задач, а також константи для частоти дискретизації та розміру буфера. Для платформи Android клас додатково містить поля для об’єктів AudioTrack і AudioRecord. Клас включає сім основних методів — UdpEspRequest(), StartReceivingAudio(), ReceivingUDP(), StopReceiving(), StartMicrophoneStreaming(), StartCapturingMicrophone(), StopMicrophoneStreaming(), а також два методи для ініціалізації аудіо на Android – InitializeAndroidAudio() і InitializeAndroidMicrophone(), і два методи для обробки аудіо — AndroidPlayStreamingAudio() і AndroidCaptureMicrophoneAudio().

Метод UdpEspRequest відповідає за ініціалізацію стримінгу аудіо від ESP32 через HTTP-запит. Він не приймає параметри та повертає Task<bool> – асинхронний результат, який вказує на успішність запуску стримінгу.

Спочатку метод вносить до журналу операцій запис про початок підключення до ESP32, а далі виконує HTTP GET-запит до кінцевої точки /startsending для повідомлення ESP32 про початок передачі аудіопотоку:

```
var http = ESP.Instance.EspHttpClient;
LogService.Instance.Log($"Запит на {ESP.Instance.StreamingEndpoint}/startsending");
using HttpResponseMessage httpMessageHandler = await
http.GetAsync(ESP.Instance.StreamingEndpoint + "/startsending");
```

Якщо запит успішний, тоді метод вносить до журналу запис про успішне отримання команди, викликає метод StartReceivingAudio() для початку прийому аудіо, встановлює властивість IsReceiving у true та повертає значення true.

Якщо запит неуспішний, метод вносить до журналу код відповіді, відображає повідомлення користувачу через DisplayAlert і повертає false.

Метод StartReceivingAudio відповідає за запуск процесу прийому аудіопотоку. Спочатку метод вносить до журналу операцій запис про початок прийому аудіо і для платформи Android ініціалізує об’єкт AudioTrack через виклик методу InitializeAndroidAudio, після чого запускає задачу відтворення аудіопотоку в окремому потоці:

```
#if ANDROID
LogService.Instance.Log("Ініціалізація Android аудіо");
InitializeAndroidAudio();
_ = Task.Run(AndroidPlayStreamingAudio, _receiveCts.Token);
#endif
```

Далі налаштовується UDP-клієнт для прослуховування на порту 5555 із тайм-аутом прийому 5000 мс та запускається задача прослуховування UDP-пакетів через метод ReceivingUDP(),

Метод ReceivingUDP() відповідає за асинхронне прослуховування UDP-пакетів із аудіоданими від ESP32. На початку роботи метод вносить до журналу операцій запис про запуск задачі прослуховування та у циклі, доки не скасовано токен _receiveCts, асинхронно отримує UDP-пакети через _udpListener.ReceiveAsync:

```
var result = await _udpListener.ReceiveAsync(_receiveCts.Token);
```

Якщо пакет отримано за менш ніж 100 мс, він додається до черги _audioBufferQueue, а лічильник отриманих пакетів збільшується, інакше пакет відкидається, що дозволяє виключити затримку в відтворенні звуку.

Додатково кожні 5 секунд метод вносить до журналу статистику отриманих і відкинутих пакетів, для можливості отримання інформації про стабільність з’єднання та передачі.

У разі помилок метод вносить до журналу деталі помилки, перевіряє токен скасування та, якщо його не скасовано, чекає 100 мс перед наступною ітерацією.

Метод StopReceiving відповідає за зупинку прийому аудіопотоку. Спочатку, аналогічно до попередніх методів, він вносить до журналу операцій запис про початок зупинки, а далі скасовує токен _receiveCts, якщо він ще не скасований, та чекає завершення задачі _receiveTask. Після цього закривається та звільняється _udpListener, якщо він існує, а для платформи Android зупиняється та звільняється об’єкт AudioTrack:

```
#if ANDROID
if (_audioTrack != null)  {
   LogService.Instance.Log("Зупинка відтворення аудіо на Android");
   _audioTrack.Stop();
   _audioTrack.Release();
   _audioTrack = null; }
#endif
```

Далі створюється новий токен _receiveCts і скидається _receiveTask:

```
_receiveCts = new CancellationTokenSource();
_receiveTask = null;
```

В кінці виконання метод відправляє HTTP GET-запит до кінцевої точки /stopsending для зупинки передачі аудіо на ESP32.

Метод StartMicrophoneStreaming відповідає за ініціалізацію передачі аудіопотоку з мікрофона на ESP32. Параметри не приймає, повертає Task<bool> — асинхронний результат, який вказує на успішність запуску стримінгу. Спочатку метод запитує дозвіл на використання мікрофона, а далі виконує HTTP GET-запит до кінцевої точки /startlistening для активації прослуховування на ESP32:

```
var http = ESP.Instance.EspHttpClient;
LogService.Instance.Log($"Запит на {ESP.Instance.StreamingEndpoint}/startlistening");
using HttpResponseMessage httpMessageHandler = await http.GetAsync(
   ESP.Instance.StreamingEndpoint + "/startlistening");
```

Якщо запит успішний, метод налаштовує _udpSender для відправлення на IP-адресу ESP32 і порт 5556, викликає метод StartCapturingMicrophone, встановлює властивість IsTransmitting у true і повертає значення true.

Метод StartCapturingMicrophone відповідає за запуск захоплення аудіо з мікрофона. Спочатку метод вносить до журналу операцій запис про початок захоплення аудіо, а потім для платформи Android ініціалізує об’єкт AudioRecord через виклик методу InitializeAndroidMicrophone і запускає задачу захоплення аудіо в окремому потоці, зберігаючи її в поле _senderTask:

```
#if ANDROID
LogService.Instance.Log("Ініціалізація мікрофона Android");
InitializeAndroidMicrophone();
_senderTask = Task.Run(() => AndroidCaptureMicrophoneAudio(), _senderCts.Token);
#endif
```

Метод StopMicrophoneStreaming відповідає за зупинку передачі аудіопотоку з мікрофона. У разі виклику метод вносить до журналу операцій запис про початок зупинки та скасовує токен _senderCts, якщо він ще не скасований. Після цього метод чекає завершення задачі _senderTask із тайм-аутом 2000 мс та закриває і звільняється _udpSender, якщо він існує:

```
if (_udpSender != null) {
   LogService.Instance.Log("Закриття UDP відправника");
   try {
      _udpSender.Close();
      _udpSender.Dispose();
      LogService.Instance.Log("UDP відправник успішно закрито"); }
   catch (Exception ex) {
      LogService.Instance.Log($"Помилка закриття UDP відправника: {ex.Message}"); }
   _udpSender = null; }
```

Для платформи Android зупиняється та звільняється об’єкт AudioRecord:

```
#if ANDROID
if (_audioRecord != null)  {
   LogService.Instance.Log("Зупинка запису з мікрофона Android");
   _audioRecord.Stop();
   _audioRecord.Release();
   _audioRecord = null; }
#endif
```

В кінці виконання створюється новий токен _senderCts, скидається _senderTask та відправляється HTTP GET-запит до кінцевої точки /stoplistening для зупинки прослуховування на ESP32.

Метод InitializeAndroidAudio відповідає за ініціалізацію об’єкта AudioTrack для відтворення аудіопотоку на платформі Android. Спочатку метод вносить до журналу операцій запис про початок ініціалізації та визначає мінімальний розмір буфера для AudioTrack використовуючи метод GetMinBufferSize() класу AudioTrack:

```
int minBufferSize = AudioTrack.GetMinBufferSize(SampleRate, ChannelOut.Mono, Android.Media.Encoding.Pcm16bit);
```

Після цього створюється об’єкт AudioTrack із параметрами для моно-аудіо, частоти 44100 Гц і 32-бітного PCM-формату та запускається його відтворення.

Метод InitializeAndroidMicrophone відповідає за ініціалізацію об’єкта AudioRecord для запису аудіо з мікрофона на Android. Спочатку метод вносить до журналу операцій запис про початок ініціалізації, визначає мінімальний розмір буфера для AudioRecord та створює об’єкт AudioRecord із параметрами для мікрофона, а саме частоти 44100 Гц і 32-бітного PCM-формату та запускає запис звуку.

Метод AndroidPlayStreamingAudio відповідає за відтворення аудіопотоку з черги _audioBufferQueue. Спочатку метод вносить до журналу операцій запис про початок відтворення та у циклі, доки не буде скасовано токен _receiveCts, метод отримуватиме буфер із черги _audioBufferQueue і записуватиме його в _audioTrack:

```
if (_audioBufferQueue.TryDequeue(out byte[] buffer)) {
   _audioTrack?.Write(buffer, 0, buffer.Length);
   totalBytesPlayed += buffer.Length;
   bufferCount++; }
```

Кожні 5 секунд виконання метод вносить до журналу виконання операцій статистику оброблених буферів і байтів.

Метод AndroidCaptureMicrophoneAudio відповідає за захоплення аудіо з мікрофона та відправлення його через UDP на ESP32. Спочатку метод вносить до журналу операцій запис про початок запису, далі створює буфер розміром RecordBufferSize, а саме 1024 байти, і в циклі, доки не скасовано токен _senderCts, читає аудіодані з _audioRecord: Якщо дані прочитано успішно, вони

відправляються через _udpSender на _espEndPoint:

```
if (bytesRead > 0 && _udpSender != null) {
   await _udpSender.SendAsync(buffer, bytesRead, _espEndPoint);
   totalBytesSent += bytesRead;
   packetsCount++; }
```

Кожні 5 секунд метод вносить до журналу статистику відправлених пакетів і байтів.

3.3 Реалізація передачі аудіофайлів

Іншою ключовою функцією застосунку для дистанційної передачі аудіо є передача аудіофайлів на мікроконтролер. Окрім безпосередньої передачі файлів з пристрою користувача на ESP32, застосунок повинен мати можливість отримання вже збережених на SD-карті пристрою аудіозаписів та реалізувати можливість їх відображення у вигляді списку, з якого користувач зможе обрати конкретний файл для його відтворення або видалення.

Для реалізації передачі файлів потрібно реалізувати клас FileTransferServices, який буде призначений для керування відправкою, отриманням списку, відтворенням та видаленням. Клас повинен використовувати протоколи HTTP та TCP для взаємодії з мікроконтролером та включати можливості обробки помилок та збереження історії до журналу операцій.

При реалізації класу було визначено, що він повинен містити поле цілочисельного типу для збереження номеру порту, який необхідний при TCP-з’єднанні з ESP32, та п’ять методів — SendFileAsync(), GetAudioFileListAsync(), PlayAudioFileAsync(), DeleteAudioFileAsync() і SimpleSanitizeFileName().

Метод SendFileAsync є головним методом відправки файлів та відповідає за передачу аудіофайлу на пристрій ESP32. Процес включає ініціалізацію передачі через HTTP-запит, встановлення TCP-з’єднання та передачу даних файлу через потік. Приймає один параметр – шлях до файлу, який потрібно передати на мікроконтролер, та повертає Task<bool> — асинхронний результат виконання методу.

Спочатку метод отримує ім’я файлу з методу GetFileName() класу Path та вносить до журналу операцій запис про початок передачі відповідного файлу, після чого виконує HTTP GET-запит до сервера ESP32 для ініціалізації передачі файлу, використовуючи URL із базової адреси ESP сінглтону та імені файлу, очищеного за допомогою методу SimpleSanitizeFileName() від символів, що не передаються через HTTP. До журналу операцій вноситься запис про відправлення запиту та отриману відповідь.

Якщо HTTP-запит виявився успішним, метод чекає 1 секунду для очікування обробки мікроконтролером відповідних дій на запит, а далі встановлює TCP-з’єднання з мікрононтролером через об’єкт TcpClient, передаючи йому порт TCP, який за замовчуванням визначений як 8080, та IP-адресу мікроконтролера. Якщо протягом 5 секунд з’єднання не було встановлено, викликається виключення.

В випадку успішного встановлення TCP-з’єднання метод відкриває потік NetworkStream і читає файл через FileStream, передаючи його частинами, використовуючи буфер розміром 1024 байти, на пристрій. Прогрес передачі відстежується та вноситься до історії операцій кожні 10% або при завершенні:

Метод GetAudioFileListAsync відповідає за отримання списку аудіофайлів, які зберігаються на SD-карті пристрою ESP32. Параметри не приймає, повертає Task<List<string>> — асинхронний результат, який містить список імен аудіофайлів або порожній список у разі помилки. Спочатку метод вносить до журналу операцій запис про початок отримання списку аудіофайлів, а далі виконує HTTP GET-запит до кінцевої точки (ESP.Instance.AudioFilesEndpoint), яка повертає список файлів у форматі JSON.

Якщо запит успішний метод логує успішне отримання списку, десеріалізує JSON-вміст у список рядків за допомогою JsonSerializer і повертає його:

```
if (response.IsSuccessStatusCode) {
  LogService.Instance.Log("Список аудіо файлів успішно отримано");
  var content = await response.Content.ReadAsStringAsync();
  var list = JsonSerializer.Deserialize<List<string>>(content);
  LogService.Instance.Log($"Знайдено {list?.Count ?? 0} аудіо файлів");
  return JsonSerializer.Deserialize<List<string>>(content) ?? new List<string>(); }
```

Якщо запит неуспішний, метод вносить до журналу операцій код відповіді та повертає порожній список.

Метод PlayAudioFileAsync відповідає за ініціалізацію відтворення аудіофайлу на пристрої ESP32. Він приймає один параметр — ім’я файлу, який потрібно відтворити, та повертає Task<bool> — асинхронний результат, який вказує на успішність операції.

Спочатку метод вносить до журналу операцій запис про початок відтворення файлу, після чого виконує HTTP GET-запит до кінцевої точки /api/audio/files/play із параметром імені файлу, очищеного за допомогою методу SimpleSanitizeFileName(), а сам запит вноситься до журналу операцій:

```
        string url =
$"{ESP.Instance.BaseURI}/api/audio/files/play?name={SimpleSanitizeFileName(fileName)}";
        LogService.Instance.Log($"Запит на відтворення: {url}");
        var response = await ESP.Instance.EspHttpClient.GetAsync(url);
```

Якщо запит успішний метод вносить до історії повідомлення про успішне відтворення та повертає true.

Метод DeleteAudioFileAsync відповідає за видалення аудіофайлу з SD-карти пристрою ESP32. Приймає один параметр — ім’я файлу, який потрібно видалити, та повертає Task<bool> — асинхронний результат, який вказує на успішність операції. Спочатку метод вносить до журналу операцій запис про початок видалення файлу, а далі виконується HTTP DELETE-запит до кінцевої точки /api/audio/files/delete із параметром імені файлу, очищеного за допомогою методу SimpleSanitizeFileName():

```
        string url =
$"{ESP.Instance.BaseURI}/api/audio/files/delete?name={SimpleSanitizeFileName(fileName)}";
        LogService.Instance.Log($"Запит на видалення: {url}");
        var response = await ESP.Instance.EspHttpClient.DeleteAsync(url);
```

Якщо запит успішний, метод записує до журналу повідомлення про успішне видалення та повертає значення true.

Метод SimpleSanitizeFileName відповідає за очищення імені файлу від недопустимих символів для використання в URL-запитах. Приймає один параметр

– ім’я файлу та повертає стрічку, що містить очищене ім’я файлу.

Якщо вхідне ім’я файлу порожнє або null, метод повертає його без змін:

```
if (string.IsNullOrEmpty(fileName))
   return fileName;
```

Якщо ім’я файлу передано коректро, воно перевіряється та посимвольно заміняє всі недопустимі символи, отримані з Path.GetInvalidFileNameChars(), на символ підкреслення:

```
foreach (char c in Path.GetInvalidFileNameChars()) {
   fileName = fileName.Replace(c, '_'); }
```

Також замінює пробіли на підкреслення для забезпечення коректного формату:

```
fileName = fileName.Replace(' ', '_');
```

4 ОЦІНКА РЕЗУЛЬТАТІВ ТА ПЕРСПЕКТИВИ РОЗВИТКУ

4.1 Тестування в реальних умовах

Після розробки застосунок був ретельно протестований в реальних умовах з підключенням та передаванням і отриманням інформації з ESP32 по локальній мережі Wi-Fi, де ESP32 виступав як сервер. Для аналізу випробування використовувались записи у журналі. Результати роботи програми показані на рисунках 4.1 — 4.3.

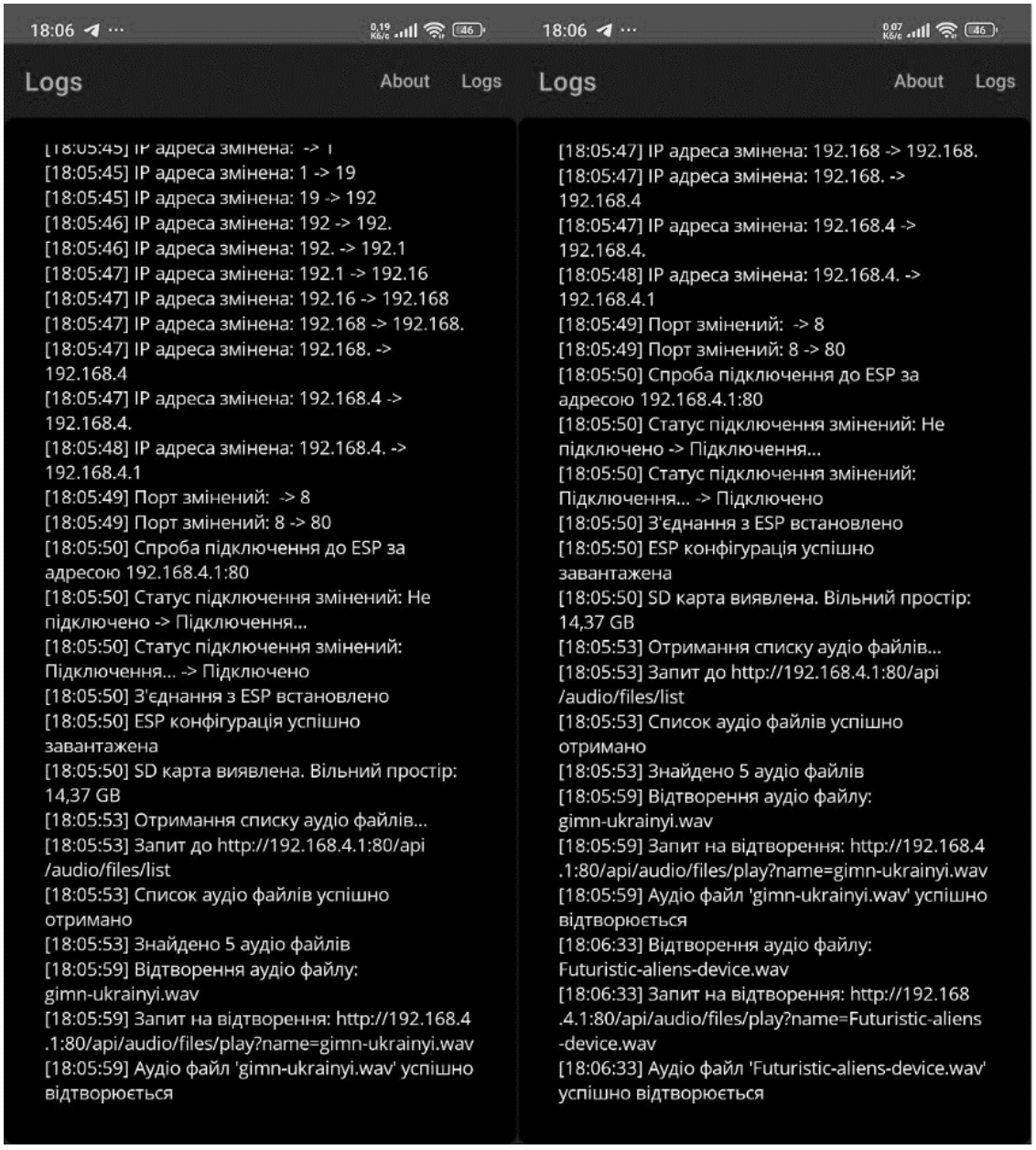

Рисунок 4.1 — Журнал підключення до ESP32 та передачі аудіофайлів

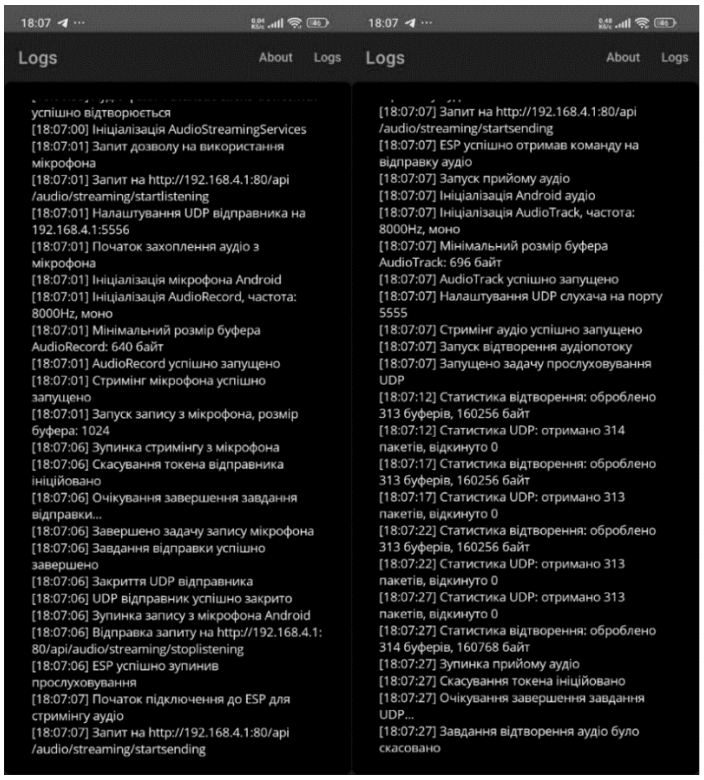

Рисунок 4.2 — Журнал передавання аудіо в реальному часі

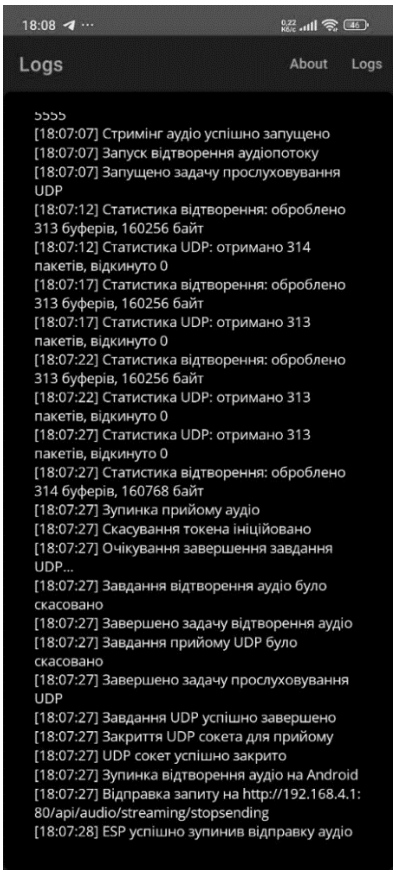

Рисунок 4.3 — Журнал передавання аудіо в реальному часі

Проаналізувавши результати роботи застосунку опираючись на рисунки 4.1 — 4.3 можна дійти висновку, що програма працює коректно. Спочатку перевірялось з’єднання з ESP32 та після успішного встановлення отримувалась інформація про список аудіофайлів, конфігурацію мікроконтролера та вільне місце на SD-карті. Після цього відбулась перевірка на коректність відтворення файлів. Було надіслано запити на відтворення аудіофайлів "gimn-ukrainyi.wav" і "Futuristic-aliens-device.wav" та отримано відповідь про успіх.

Також з журналу видно, що усі операції на потокову передачу аудіоданих проходили без затримок більших за 1 секунду, що свідчить про відсутність значних затримок у обробці запитів та запуску передачі даних. Додатково у журнал виводилась інформація про кількість отриманих та відкинутих пакетів, що показує наскільки успішно обробляється передача звуку по мережі. З рисунку 4.3 видно, що за весь час прийому аудіопотоку не було відкинуто жодного UDP-пакету, що свідчить про стабільне з’єднання між додатком та мікроконтролером та відсутність затримок у передачі пакетів більших за 100 мс.

Отже, програма працює успішно та реалізує усю необхідну функціональність, до якої відноситься потокова передача аудіоданих в реальному часі як з ESP32 до застосунку, так і від телефону до мікроконтролера, управління файлами на SD-карті та керування їх програванням і видаленням.

4.2 Аналіз результатів

Отже, після розробки та тестування застосунку для передачі аудіопотоків і файлів між мобільним додатком на операційній системі Android і мікроконтролером ESP32 можна дійти висновку, що він демонструє високу функціональність в рамках поставленого завдання. Система успішно реалізує двосторонню передачу аудіо в реальному часі з використанням протоколу UDP, забезпечуючи прийом даних з ESP32 в PCM форматі з частотою дискретизації 8 кГц та кодуванням в 32 біта, а також відправку з телефону в PCM форматі з частотою дискретизації 8 кГц та кодуванням в 16 біт.

Передача аудіофайлів, з використанням протоколу TCP також показала свою ефективність та надала можливість надійної передачі даних з підтвердженням доставки для їх подальшого запису на SD-карту.

Також програма продемонструвала коректну відпраку команд з використанням протоколу HTTP, оскільки уся функціональність застосунку базується на спілкуванні через HTTP між пристроєм та телефоном. Журнали подій підтвердили стабільну роботу інтерфейсу, побудованого на патерні MVVM, із динамічним оновленням статусів і обробкою помилок, таких як втрата з’єднання чи переривання. Загалом, застосунок виявився гнучким інструментом для IoT-розробок, здатним адаптуватися до умов локальної мережі з мінімальними затримками до 100 мс під час передачі пакетів.

Також створений застосунок може бути інтегрований до апаратно програмного комплексу у вигляді робота, здатного пересуватись. Він може розширити його функціональність, наприклад бути центральним елементом управління роботом, забезпечуючи обробку аудіопотоків у реальному часі для навігації та взаємодії з оточенням. Наприклад, аудіодані, отримані через ESP32, можуть використовуватися для розпізнавання голосових команд, таких як "рухатися вперед" чи "зупинитися", що дозволить керувати рухом робота без додаткових контролерів. Також застосунок може передавати збережені аудіофайли з SD-карти ESP32 до системи робота для відтворення сповіщень або аналізу звукового оточення, наприклад, для виявлення перешкод чи аномалій під час руху.

Отже, можна підтвердити ефективність системи її здатністю забезпечувати стабільне дистанційне передавання аудіо та керування файлами за умов обмежених ресурсів ESP32 і змінної якості мережі. Аналіз результатів показав, що затримки на етапі налаштування становили кілька секунд, тоді як основна передача даних відбувалася з мінімальними втратами пакетів, що гарантувало високу якість звуку. Механізми буферизації та відкидання застарілих пакетів сприяли стабільності роботи, хоча залежність від якості Wi-Fi з’єднання залишаться головним компонентом цього показника.

4.3 Можливості розширення функціоналу

Розроблений апаратно-програмний комплекс для дистанційного передавання аудіо має значний потенціал для розширення функціональних можливостей, що дозволить адаптувати його до нових завдань і підвищити зручність використання. Розширення базується на вдосконаленні архітектури додатку та інтеграції додаткових технологій, які можуть значно розширити його застосування.

Використання фреймворку .NET MAUI для кросплатформеності відкриває перспективи перенесення додатку на інші операційні системи, такі як iOS і Windows, крім існуючої платформи Android. Завдяки .NET MAUI можна зберегти єдину кодову базу, що спростить підтримку та оновлення, а також адаптувати інтерфейс під особливості кожної платформи, зберігаючи при цьому базову логіку роботи з ESP32 через UDP і TCP.

Інтеграція зі сторонніми сервісами для створення централізованого сервера, наприклад, з використанням Amazon Web Services (AWS), значно розширить можливості системи. AWS може бути використаний для організації хмарного сховища аудіофайлів, що дозволить створювати резервні копії записів із SD-карти ESP32 і забезпечувати доступ до них із будь-якої точки світу. Крім того, AWS Lambda може бути застосована для обробки аудіопотоків у реальному часі, наприклад, для аналізу звукових даних із використанням алгоритмів машинного навчання, що виявлятимуть аномалії чи класифікуватимуть звуки. AWS IoT Core може слугувати для управління кількома пристроями ESP32, забезпечуючи централізовану координацію їхньої роботи та передачу даних через захищені канали. Така інтеграція зробить систему більш масштабованою та придатною для складних IoT-систем, де потрібна централізована обробка та аналіз даних.

Синхронізація з іншими пристроями дозволить створити мережу взаємопов’язаних вузлів, що працюють із аудіоданими. Наприклад, кілька ESP32, підключених до різних джерел звуку, можуть передавати інформацію до одного додатку, який координуватиме їхню роботу, або синхронізуватися з іншими розумними пристроями, такими як камери чи датчики, для створення

мультимодальної системи моніторингу. Для реалізації цього необхідно розробити протокол обміну даними між пристроями та адаптувати додаток для обробки множинних потоків, що вимагатиме оптимізації мережевих ресурсів і підвищення стабільності з’єднання. Такий підхід зробить систему гнучкою платформою для розподілених IoT-рішень, зокрема в автоматизації та робототехніці.

ВИСНОВКИ

У ході виконання комплексної бакалаврської дипломної роботи було створено та протестовано в реальних умовах кіберфізичну систему для дистанційного передавання аудіо. Функціональність системи представлена можливістю обміну аудіо в реальному часі за допомогою протоколу UDP, відправлення аудіофайлів з мобільного пристрою до мікроконтролера з використанням протоколу TCP, а також керування мікроконтролером через HTTP-запити.

Під час розробки застосунку для управління системою було проведено огляд сучасних рішень для дистанційного передавання аудіо і аналіз їх можливостей та обмежень. На основі цього аналізу було сформовано вимоги до функціональності майбутнього застосунку, після чого розпочато реалізацію архітектури взаємодії клієнта та мікроконтролера через локальну мережу.

Було реалізовано механізм підключення до ESP32, створено систему керування режимами його роботи за допомогою HTTP-запитів, розроблено передачу аудіопотоку у режимі реального часу через протокол UDP, а також реалізовано передачу та обробку аудіофайлів через TCP.

Застосунок був реалізований за допомогою платформи .NET MAUI та мови розмітки XAML, що дозволило використовувати вбудувані API Android для запису і відтворення аудіо в реальному часі та використовувати бібліотеку стандарних класів, яка надає зручні високорівневі можливості для роботи з мережею.

Під час тестування системи було доведено, що вона успішно виконує поставлені задачі та демонструє стабільну роботу в умовах локальної мережі. Результати роботи, досягнуті в процесі розробки системи, можуть бути використані для подальшого розвитку мобільних застосунків керування IoT-пристроями, зокрема в системах моніторингу, охорони або дистанційного керування.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Дацюк В. В. Бондарчук О. Б., Богомолов С. В., Мартинюк Т. Б. Дистанційна передача аудіо в IoT. "LIV Всеукраїнська науково-технічна конференція факультету інформаційних технологій та комп'ютерної інженерії (2025)". URL: https://conferences.vntu.edu.ua/index.php/all-fitki/all-fitki-2025/paper/view/2441

2. A Comprehensive Guide to TeamViewer Audio Sharing. URL: https://www.airdroid.com/remote-support/teamviewer-audio-sharing/ (дата звернення: 03.05.2025).

3. Use your phone as a mic for Windows. URL: https://docs.audiorelay.net/instructions/windows/use-your-phone-as-a-mic-for-windows-10 (дата звернення: 05.05.2025).

4. SonoBus. URL: https://sonobus.net/ (дата звернення: 05.05.2025).

5. LiveVoice. URL: https://livevoice.io/en (дата звернення: 05.05.2025).

6. Функція «Слухати наживо». URL: https://support.apple.com/uk-ua/102479 (дата звернення: 05.05.2025).

7. Tanenbaum A., Wetherall D. Computer networks (5th Edition). New Jersey : Prentice Hall, 2010. 960 с.

8. Комп’ютерні мережі : підручник / Азаров О. Д., Захарченко С. М., Кадук О. В., Орлова М. М., Тарасенко В. П. Вінниця : ВНТУ, 2020. 378 с.

9. WebRTC API. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API (дата звернення: 06.05.2025).

10. MQTT: The Standard for IoT Messaging. URL: https://mqtt.org/ (дата звернення: 06.05.2025).

11. Kotlin. URL: https://uk.wikipedia.org/wiki/Kotlin (дата звернення: 06.05.2025).

12. Flutter. URL: https://github.com/flutter/flutter (дата звернення: 07.05.2025).

13. Xamarin. URL: https://dotnet.microsoft.com/en-us/apps/xamarin (дата звернення: 07.05.2025).

14. .NET Multi-platform App UI. URL: https://dotnet.microsoft.com/en-

us/apps/maui (дата звернення: 07.05.2025).

15. A tour of the C# language. URL: https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/overview (дата звернення: 07.05.2025).

16. What is Visual Studio?. URL: https://learn.microsoft.com/en-us/visualstudio/get-started/visual-studio-ide?view=vs-2022 (дата звернення: 07.05.2025).

17. Michael Stonis. Enterprise Application Patterns using .NET MAUI. Redmond, Washington : Microsoft Developer Division, .NET, and Visual Studio product teams, 2022. 99 p.

18. XAML overview. URL: https://learn.microsoft.com/en-us/dotnet/desktop/wpf/xaml/ (дата звернення: 08.05.2025).

19. ContentPage. URL: https://learn.microsoft.com/en-us/dotnet/maui/user-interface/pages/contentpage?view=net-maui-9.0 (дата звернення: 08.05.2025).

ДОДАТОК А

Технічне завдання

Міністерство освіти та науки України

Вінницький національний технічний університет

Факультет інформаційних технологій та комп’ютерної інженерії

ЗАТВЕРДЖУЮ
Завідувач кафедри ОТ
д.т.н. проф._____Азаров О. Д.
«21» березня 2025 року

ТЕХНІЧНЕ ЗАВДАННЯ

на виконання комплексної бакалаврської кваліфікаційної роботи
«Апаратно-програмний комплекс для дистанційного передавання аудіо. Частина 2. Застосунок для управління»

08-54.КБКР.037.00.000 ТЗ

Керівник роботи д.т.н., проф. каф. ОТ

Мартинюк Т. Б.

Студент групи 1КІ-21б

Дацюк В. В.

ВНТУ 2025

1 Підстава для виконання комплексної бакалаврської кваліфікаційної роботи (КБКР)

1.1 Актуальність теми полягає у створенні сучасного застосунку для можливості передачі інформації у локальній мережі з використанням таких технологій, як протоколів UDP, TCP, HTTP, використання платформи .NET MAUI та розробці сучасних та зручних інтерфейсів з використанням мови розмітки XAML для можливості ефективного керування та обміну інформації з мікроконтролерами.

1.2 Наказ ВНТУ №97 про затвердження теми БКР від 20.03.2025.

2 Мета БКР і призначення розробки

2.1 Мета роботи — створення застосунку для керування апаратно-програмним комплексом, побудованим з використанням мікроноктролеру ESP32, який дозволяє забезпечити двосторонній аудіостримінг через протокол UDP, обмін аудіофайлами по TCP, а також управління пристроєм через HTTP-запити.

2.2 Призначення розробки — створення програмного засобу для можливості передачі аудіоданих в реальному часі та аудіофайлів для їх подальшого відтворення, що може використовуватись в охоронних системах та системах сповіщення.

3 Вихідні дані для виконання БКР

3.1 Базові знання технологій розробки програмного забезпечення.

3.2 Принципи мережевого передавання інформації.

3.3 Алгоритми дистанційної передачі аудіо.

3.4 Середовище розробки програмного забезпечення.

3.5 Застосування мобільних застосунків для управління та обміну інформації з мікроконтролерами.

4 Вимоги до виконання БКР

4.1 Провести аналіз існуючих рішень та підходів дистанційної передачі аудіо в реальному часі;

4.2 Провести аналіз технологій розробки програмного забезпечення та алгоритму роботи застосунку;

4.3 Реалізувати інтрейфес та функціональні можливості;

4.4 Провести тестування додатку в реальних умовах.

5 Етапи БКР та очікувані результати

Етапи роботи та очікувані результати приведено в Таблиці А.1.

Таблиця А.1 — Етапи БКР

№ з/п	Назва етапу	Термін виконання		Очікувані результати
		початок	кінець	
1	Постановка задачі роботи	21.03.25	24.03.25	Огляд джерел, висновки із взаємодії викладачів та студентів
2	Пошук та огляд існуючих рішень	25.03.25	5.04.25	Розділ 1
3	Пошук та аналіз сучасних технологій розробки мобільних додатків	6.04.25	10.04.25	Розділ 2
4	Визначення алгоритму передачі даних в мережі	11.04.25	20.04.25	Розділ 2
5	Розробка й тестування застосунку	21.04.25	1.05.25	Розділ 3 та 4
6	Оформлення пояснювальної записки та ілюстративного матеріалу	2.05.25	12.05.25	ПЗ, графічний матеріал, презентація

6 Матеріали, що подаються до захисту КБКР

До захисту подаються: пояснювальна записка КБКР, графічні і ілюстративні матеріали, протокол попереднього захисту КБКР на кафедрі, відгук наукового керівника, відгук рецензента, анотації до КБКР українською та іноземною мовами.

7 Порядок контролю виконання та захисту КБКР

Виконання етапів графічної та розрахункової документації КБКР контролюється науковим керівником згідно зі встановленими термінами. Захист КБКР відбувається на засіданні Екзаменаційної комісії, затвердженої наказом ректора.

8.1 При оформлюванні БКР використовуються:

— ДСТУ 3008: 2015 «Звіти в сфері науки і техніки. Структура та правила оформлювання»;

— ДСТУ 8302: 2015 «Бібліографічні посилання. Загальні положення та правила складання»;

— методичні вказівки до виконання бакалаврських кваліфікаційних робіт зі спеціальності 123 — «Комп’ютерна інженерія»;

— документи на які посилаються у вище вказаних.

8.2 Порядок виконання КБКР викладено в «Положення про кваліфікаційні роботи на першому (бакалаврському) рівні вищої освіти СУЯ ВНТУ-03.02.02-П.001.01:21.

ДОДАТОК Б

ПРОТОКОЛ ПЕРЕВІРКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Назва роботи: Апаратно-програмний комплекс для дистанційного передавання аудіо. Частина 2. Застосунок для управління

Тип роботи: бакалаврська кваліфікаційна робота
(бакалаврська кваліфікаційна робота / магістерська кваліфікаційна робота)

Підрозділ кафедра обчислювальної техніки
(кафедра, факультет, навчальна група)

Коефіцієнт подібності текстових запозичень, виявлених у роботі системою StrikePlagiarism 3,5 %

Висновок щодо перевірки кваліфікаційної роботи (відмітити потрібне)

- ✓ Запозичення, виявлені у роботі, є законними і не містять ознак плагіату, фабрикації, фальсифікації. Роботу прийняти до захисту

- ☐ У роботі не виявлено ознак плагіату, фабрикації, фальсифікації, але надмірна кількість текстових запозичень та/або наявність типових розрахунків не дозволяють прийняти рішення про оригінальність та самостійність її виконання. Роботу направити на доопрацювання.

- ☐ У роботі виявлено ознаки плагіату та/або текстових маніпуляцій як спроб укриття плагіату, фабрикації, фальсифікації, що суперечить вимогам законодавства та нормам академічної доброчесності. Робота до захисту не приймається.

Експертна комісія:

Азаров О.Д., зав. каф. ОТ _____ _____
(прізвище, ініціали, посада) (підпис)

Крупельницький Л.В., доц. каф ОТ _____ _____
(прізвище, ініціали, посада) (підпис)

Особа, відповідальна за перевірку _____ Захарченко С.М.
(прізвище, ініціали) (підпис)

З висновком експертної комісії ознайомлений(-на)

Керівник _____ д.т.н., проф. каф. ОТ Мартинюк Т. Б.
(підпис) (прізвище, ініціали, посада)

Здобувач _____ Дацюк В. В.
(підпис) (прізвище, ініціали)

ДОДАТОК В

Графічна частина

АПАРАТНО-ПРОГРАМНИЙ КОМПЛЕКС ДЛЯ ДИСТАНЦІЙНОГО ПЕРЕДАВАННЯ АУДІО. ЧАСТИНА 2. ЗАСТОСУНОК ДЛЯ УПРАВЛІННЯ

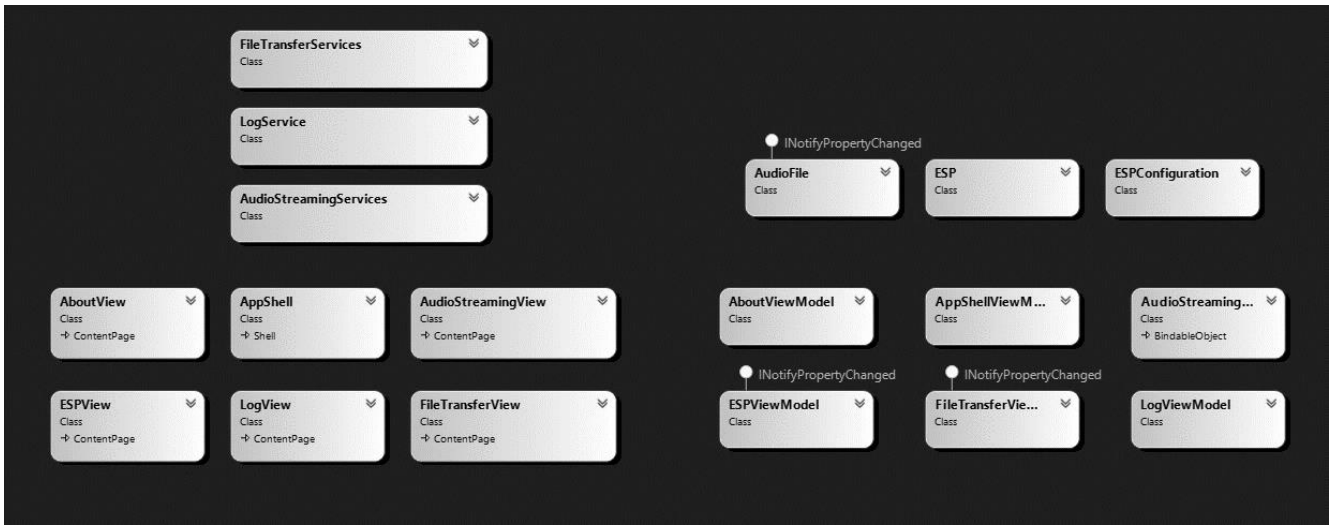

Рисунок В.1 — Діаграма класів в Visual Studio

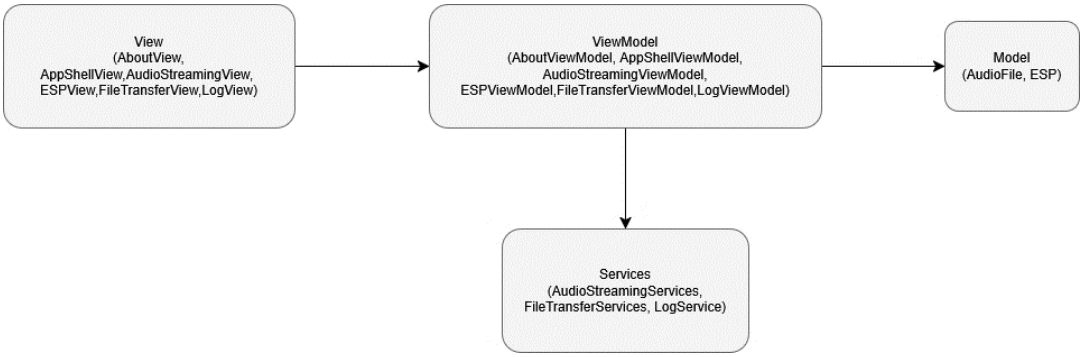

Рисунок В.2 — Архітектурна діаграма застосунку

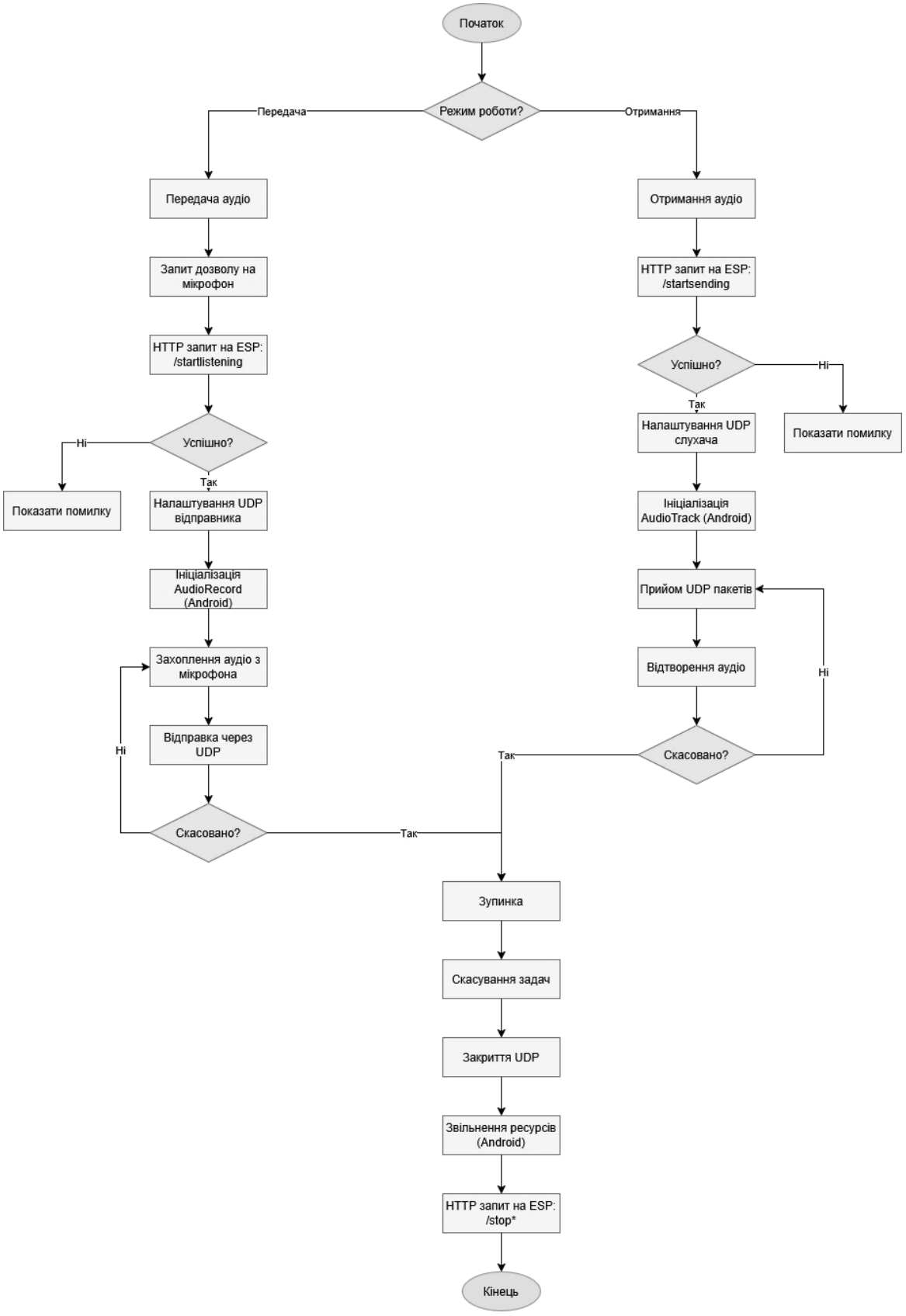

Рисунок В.3 — Блок-схема алгоритму передачі аудіо в реальному часі

Рисунок В.4 — Блок-схема алгоритму передачі аудіофайлів

ДОДАТОК Г

Лістинг програмного забезпечення

AudioStreamingServices.cs

```
#if ANDROID
using Android.Media;
#endif
namespace EspHandler.Services {
   class AudioStreamingServices {
#if ANDROID
      private AudioTrack _audioTrack;
      private AudioRecord _audioRecord;
#endif
      private const int SampleRate = 44100;
      private const int RecordBufferSize = 1024;
      private UdpClient _udpListener;
      private UdpClient _udpSender;
      private IPEndPoint _espEndPoint;
      private CancellationTokenSource _receiveCts;
      private CancellationTokenSource _senderCts;
      private Task _receiveTask;
      private Task _senderTask;
      private ConcurrentQueue<byte[]> _audioBufferQueue = new ConcurrentQueue<byte[]>();
      public bool IsReceiving { get; private set; }
      public bool IsTransmitting { get; private set; }
      public AudioStreamingServices() {
         LogService.Instance.Log("Ініціалізація AudioStreamingServices");
         _receiveCts = new CancellationTokenSource();
         _senderCts = new CancellationTokenSource();
      }
      public async Task<bool> UdpEspRequest() {
         LogService.Instance.Log("Початок підключення до ESP для стримінгу аудіо");
         try {
            var http = ESP.Instance.EspHttpClient;
            LogService.Instance.Log($"Запит на {ESP.Instance.StreamingEndpoint}/startsending");
            using HttpResponseMessage httpMessageHandler = await
http.GetAsync(ESP.Instance.StreamingEndpoint + "/startsending");
            if (httpMessageHandler.IsSuccessStatusCode) {
               LogService.Instance.Log("ESP успішно отримав команду на відправку аудіо");
               await StartReceivingAudio();
               IsReceiving = true;
               LogService.Instance.Log("Стримінг аудіо успішно запущено");
               return true;
            }
            else {
               LogService.Instance.Log($"Помилка запиту на стримінг:
{httpMessageHandler.StatusCode}");
               await Application.Current.MainPage.DisplayAlert("Стримінг аудіо",
                  $"Помилка підключення: {httpMessageHandler.StatusCode}", "ОК");
               return false;
```

```
            }
        }
        catch (Exception ex) {
            LogService.Instance.Log($"Критична помилка підключення до ESP: {ex.Message}");
            await Application.Current.MainPage.DisplayAlert("Помилка з'єднання",
                $"Не вдалося підключитися до ESP: {ex.Message}", "OK");
            return false;
        }
    }
    private async Task StartReceivingAudio() {
        LogService.Instance.Log("Запуск прийому аудіо");
#if ANDROID
        LogService.Instance.Log("Ініціалізація Android аудіо");
        InitializeAndroidAudio();
        _ = Task.Run(AndroidPlayStreamingAudio, _receiveCts.Token);
#endif
        try {
            LogService.Instance.Log("Налаштування UDP на порту 5555");
            _udpListener = new UdpClient(5555);
            _udpListener.Client.ReceiveTimeout = 5000;
            _receiveTask = Task.Run(() => ReceivingUDP(), _receiveCts.Token);
        }
        catch (Exception ex) {
            LogService.Instance.Log($"Не вдалося запустити UDP прослуховування: {ex.Message}");
            await Application.Current.MainPage.DisplayAlert("Помилка UDP",
                $"Не вдалося запустити UDP прослуховування: {ex.Message}", "OK");
        }
    }
    public async Task ReceivingUDP() {
        LogService.Instance.Log("Запущено задачу прослуховування UDP");
        int packetsReceived = 0;
        int packetsDropped = 0;
        DateTime lastLogTime = DateTime.Now;
        try {
            while (!_receiveCts.Token.IsCancellationRequested) {
                try {
                    long startTime = DateTime.Now.Ticks / TimeSpan.TicksPerMillisecond;
                    var result = await _udpListener.ReceiveAsync(_receiveCts.Token);
                    if ((DateTime.Now.Ticks / TimeSpan.TicksPerMillisecond) - startTime < 100) {
                        _audioBufferQueue.Enqueue(result.Buffer);
                        packetsReceived++;
                    }
                    else {
                        packetsDropped++;
                    }
                    if ((DateTime.Now - lastLogTime).TotalSeconds >= 5) {
                        LogService.Instance.Log($"Статистика UDP: отримано {packetsReceived} пакетів,
відкинуто {packetsDropped}");
                        packetsReceived = 0;
                        packetsDropped = 0;
                        lastLogTime = DateTime.Now;
```

```
            }
        }
        catch (OperationCanceledException) {
            LogService.Instance.Log("Завдання прийому UDP було скасовано");
            break;
        }
        catch (Exception ex) {
            LogService.Instance.Log($"Помилка отримання UDP: {ex.Message}");
            if (_receiveCts.Token.IsCancellationRequested) {
                LogService.Instance.Log("Завдання скасовано після помилки");
                break;
            }
            await Task.Delay(100);
        }
    }
  }
  catch (Exception ex) {
    LogService.Instance.Log($"Критична помилка в циклі UDP: {ex.Message}");
  }
  finally {
    LogService.Instance.Log("Завершено задачу прослуховування UDP");
  }
}
public async Task StopReceiving() {
  LogService.Instance.Log("Зупинка прийому аудіо");
  try {
    if (!_receiveCts.IsCancellationRequested) {
        _receiveCts.Cancel();
        LogService.Instance.Log("Скасування токена ініційовано");
    }
    if (_receiveTask != null) {
        LogService.Instance.Log("Очікування завершення завдання UDP...");
        try {
            var completedTask = await Task.WhenAny(_receiveTask, Task.Delay(2000));
            if (completedTask != _receiveTask) {
                LogService.Instance.Log("Тайм-аут очікування завершення завдання UDP");
            }
            else {
                LogService.Instance.Log("Завдання UDP успішно завершено");
            }
        }
        catch (Exception ex) {
            LogService.Instance.Log($"Помилка при очікуванні завдання UDP: {ex.Message}");
        }
    }
    if (_udpListener != null) {
        LogService.Instance.Log("Закриття UDP сокета для прийому");
        try {
            _udpListener.Close();
            _udpListener.Dispose();
            LogService.Instance.Log("UDP сокет успішно закрито");
```

```
            }
            catch (Exception ex) {
                LogService.Instance.Log($"Помилка закриття UDP сокета: {ex.Message}");
            }
            _udpListener = null;
        }
#if ANDROID
        if (_audioTrack != null) {
            LogService.Instance.Log("Зупинка відтворення аудіо на Android");
            _audioTrack.Stop();
            _audioTrack.Release();
            _audioTrack = null;
        }
#endif
        _receiveCts = new CancellationTokenSource();
        _receiveTask = null;
        LogService.Instance.Log($"Відправка запиту на
{ESP.Instance.StreamingEndpoint}/stopsending");
        using HttpResponseMessage response = await ESP.Instance.EspHttpClient.GetAsync(
            ESP.Instance.StreamingEndpoint + "/stopsending");
        if (!response.IsSuccessStatusCode) {
            LogService.Instance.Log($"Помилка зупинки стримінгу на ESP:
{response.StatusCode}");
            await Application.Current.MainPage.DisplayAlert("Увага",
                "Не вдалося передати команду зупинки на ESP", "ОК");}
        else  {
            LogService.Instance.Log("ESP успішно зупинив відправку аудіо");
        }
        IsReceiving = false;
      }
      catch (Exception ex)  {
        LogService.Instance.Log($"Помилка при спробі зупинити прийом аудіо: {ex.Message}");
      }
    }
    public async Task<bool> StartMicrophoneStreaming()  {
      LogService.Instance.Log("Запит дозволу на використання мікрофона");
      await Permissions.RequestAsync<Permissions.Microphone>();
      try  {
        var http = ESP.Instance.EspHttpClient;
        LogService.Instance.Log($"Запит на {ESP.Instance.StreamingEndpoint}/startlistening");
        using HttpResponseMessage httpMessageHandler = await http.GetAsync(
            ESP.Instance.StreamingEndpoint + "/startlistening");
        if (httpMessageHandler.IsSuccessStatusCode)   {
            LogService.Instance.Log($"Налаштування UDP відправника на
{ESP.Instance.IpAddress}:5556");
            _espEndPoint = new IPEndPoint(IPAddress.Parse(ESP.Instance.IpAddress), 5556);
            _udpSender = new UdpClient();
            _udpSender.Client.SendTimeout = 5000;
            await StartCapturingMicrophone();
            IsTransmitting = true;
            LogService.Instance.Log("Стримінг мікрофона успішно запущено");
```

```
                return true;   }
            else  {
                LogService.Instance.Log($"Помилка активації мікрофона:
{httpMessageHandler.StatusCode}");
                await Application.Current.MainPage.DisplayAlert("Мікрофон",
                    $"Помилка активації мікрофона: {httpMessageHandler.StatusCode}", "ОК");
                return false;
            }
        }
        catch (Exception ex) {
            LogService.Instance.Log($"Критична помилка запуску стримінгу мікрофона:
{ex.Message}");
            await Application.Current.MainPage.DisplayAlert("Помилка",
                $"Помилка запуску стримінгу мікрофона: {ex.Message}", "ОК");
            return false;
        }
    }
    private async Task StartCapturingMicrophone()  {
        LogService.Instance.Log("Початок захоплення аудіо з мікрофона");
#if ANDROID
        LogService.Instance.Log("Ініціалізація мікрофона Android");
        InitializeAndroidMicrophone();
        _senderTask = Task.Run(() => AndroidCaptureMicrophoneAudio(), _senderCts.Token);
#endif
    }
    public async Task StopMicrophoneStreaming()  {
        LogService.Instance.Log("Зупинка стримінгу з мікрофона");
        try  {
            if (!_senderCts.IsCancellationRequested) {
                _senderCts.Cancel();
                LogService.Instance.Log("Скасування токена відправника ініційовано");
            }
            if (_senderTask != null) {
                LogService.Instance.Log("Очікування завершення завдання відправки...");
                try {
                    var completedTask = await Task.WhenAny(_senderTask, Task.Delay(2000));
                    if (completedTask != _senderTask) {
                        LogService.Instance.Log("Тайм-аут очікування завершення завдання відправки");
                    }
                    else  {
                        LogService.Instance.Log("Завдання відправки успішно завершено");
                    }
                }
                catch (Exception ex) {
                    LogService.Instance.Log($"Помилка при очікуванні завдання відправки:
{ex.Message}");
                }
            }
            if (_udpSender != null) {
                LogService.Instance.Log("Закриття UDP відправника");
                try {
```

```
                _udpSender.Close();
                _udpSender.Dispose();
                LogService.Instance.Log("UDP відправник успішно закрито");
            }
            catch (Exception ex)  {
                LogService.Instance.Log($"Помилка закриття UDP відправника: {ex.Message}");
            }
            _udpSender = null;
        }
#if ANDROID
        if (_audioRecord != null) {
            LogService.Instance.Log("Зупинка запису з мікрофона Android");
            _audioRecord.Stop();
            _audioRecord.Release();
            _audioRecord = null;
        }
#endif
        _senderCts = new CancellationTokenSource();
        _senderTask = null;
        LogService.Instance.Log($"Відправка запиту на
{ESP.Instance.StreamingEndpoint}/stoplistening");
        using HttpResponseMessage response = await ESP.Instance.EspHttpClient.GetAsync(
            ESP.Instance.StreamingEndpoint + "/stoplistening");
        if (!response.IsSuccessStatusCode) {
            LogService.Instance.Log($"Помилка зупинки прослуховування на ESP:
{response.StatusCode}");
            await Application.Current.MainPage.DisplayAlert("Увага",
                "Не вдалося передати команду зупинки мікрофона на ESP", "OK");
        }
        else {
            LogService.Instance.Log("ESP успішно зупинив прослуховування");
        }
        IsTransmitting = false;
    }
    catch (Exception ex) {
        LogService.Instance.Log($"Помилка при зупинці стримінгу мікрофона: {ex.Message}");
        await Application.Current.MainPage.DisplayAlert("Помилка",
            $"Помилка при зупинці стримінгу мікрофона: {ex.Message}", "OK");
    }
}

#if ANDROID
    private void InitializeAndroidAudio()  {
        try {
            LogService.Instance.Log($"Ініціалізація AudioTrack, частота: {SampleRate}Hz, моно");
            int minBufferSize = AudioTrack.GetMinBufferSize(SampleRate, ChannelOut.Mono,
Android.Media.Encoding.Pcm16bit);
            LogService.Instance.Log($"Мінімальний розмір буфера AudioTrack: {minBufferSize}
байт");
            _audioTrack = new AudioTrack(
                Android.Media.Stream.Music,
```

```
                SampleRate,
                ChannelOut.Mono,
                Android.Media.Encoding.Pcm32bit,
                minBufferSize,
                AudioTrackMode.Stream);
            _audioTrack.Play();
            LogService.Instance.Log("AudioTrack успішно запущено");
        }
        catch (Exception ex) {
            LogService.Instance.Log($"Критична помилка ініціалізації AudioTrack: {ex.Message}");
            Console.WriteLine($"Помилка ініціалізації AudioTrack: {ex.Message}");
            MainThread.BeginInvokeOnMainThread(async () => {
                await Application.Current.MainPage.DisplayAlert("Помилка аудіо",
                    $"Не вдалося ініціалізувати аудіовідтворення: {ex.Message}", "ОК");
            });
        }
    }
    private void InitializeAndroidMicrophone()  {
        try {
            LogService.Instance.Log($"Ініціалізація AudioRecord, частота: {SampleRate}Hz, моно");
            int minBufferSize = AudioRecord.GetMinBufferSize(SampleRate, ChannelIn.Mono,
Android.Media.Encoding.Pcm16bit);
            LogService.Instance.Log($"Мінімальний розмір буфера AudioRecord: {minBufferSize}
байт");
            _audioRecord = new AudioRecord(
                AudioSource.Mic,
                SampleRate,
                ChannelIn.Mono,
                Android.Media.Encoding.Pcm32bit,
                minBufferSize);
            _audioRecord.StartRecording();
            LogService.Instance.Log("AudioRecord успішно запущено");
        }
        catch (Exception ex) {
            LogService.Instance.Log($"Помилка ініціалізації AudioRecord: {ex.Message}");
            Console.WriteLine($"Помилка ініціалізації AudioRecord: {ex.Message}");
            MainThread.BeginInvokeOnMainThread(async () =>  {
                await Application.Current.MainPage.DisplayAlert("Помилка мікрофона",
                    $"Не вдалося ініціалізувати запис з мікрофона: {ex.Message}", "ОК");
            });
        }
    }
    private async Task AndroidPlayStreamingAudio() {
        LogService.Instance.Log("Запуск відтворення аудіопотоку");
        int totalBytesPlayed = 0;
        int bufferCount = 0;
        DateTime lastLogTime = DateTime.Now;
        try {
            while (!_receiveCts.Token.IsCancellationRequested) {
                try {
                    if (_audioBufferQueue.TryDequeue(out byte[] buffer)) {
```

```
                    _audioTrack?.Write(buffer, 0, buffer.Length);
                    totalBytesPlayed += buffer.Length;
                    bufferCount++;
                    if ((DateTime.Now - lastLogTime).TotalSeconds >= 5) {
                        LogService.Instance.Log($"Статистика відтворення: оброблено {bufferCount}
буферів, {totalBytesPlayed} байт");
                        totalBytesPlayed = 0;
                        bufferCount = 0;
                        lastLogTime = DateTime.Now;
                    }
                }
                else {
                    await Task.Delay(5, _receiveCts.Token);
                }
            }
            catch (OperationCanceledException) {
                LogService.Instance.Log("Завдання відтворення аудіо було скасовано");
                break;
            }
            catch (Exception ex) {
                LogService.Instance.Log($"Помилка відтворення аудіо: {ex.Message}");
                if (_receiveCts.Token.IsCancellationRequested) {
                    LogService.Instance.Log("Завдання відтворення скасовано після помилки");
                    break;
                }
                await Task.Delay(100);
            }
        }
    }
    catch (Exception ex) {
        LogService.Instance.Log($"Критична помилка відтворення аудіо: {ex.Message}");
    }
    finally {
        LogService.Instance.Log("Завершено задачу відтворення аудіо");
    }
}
private async Task AndroidCaptureMicrophoneAudio() {
    LogService.Instance.Log($"Запуск запису з мікрофона, розмір буфера: {RecordBufferSize}");
    byte[] buffer = new byte[RecordBufferSize];
    int totalBytesSent = 0;
    int packetsCount = 0;
    DateTime lastLogTime = DateTime.Now;
    try {
        while (!_senderCts.Token.IsCancellationRequested) {
            try {
                if (_audioRecord == null) {
                    LogService.Instance.Log("_audioRecord = null, завершення запису");
                    break;
                }
                int bytesRead = _audioRecord.Read(buffer, 0, buffer.Length);
                if (bytesRead > 0 && _udpSender != null) {
```

```
                    await _udpSender.SendAsync(buffer, bytesRead, _espEndPoint);
                    totalBytesSent += bytesRead;
                    packetsCount++;
                    if ((DateTime.Now - lastLogTime).TotalSeconds >= 5) {
                        LogService.Instance.Log($"Статистика мікрофона: відправлено {packetsCount}
пакетів, {totalBytesSent} байт");
                        totalBytesSent = 0;
                        packetsCount = 0;
                        lastLogTime = DateTime.Now;
                    }
                }
            }
            catch (OperationCanceledException) {
                LogService.Instance.Log("Завдання запису мікрофона було скасовано");
                break;
            }
            catch (Exception ex)  {
                LogService.Instance.Log($"Загальна помилка запису мікрофона: {ex.Message}");
                if (_senderCts.Token.IsCancellationRequested)  {
                    LogService.Instance.Log("Завдання запису мікрофона скасовано після помилки");
                    break;
                }
                await Task.Delay(100);
            }
        }
    }
    catch (Exception ex)  {
        LogService.Instance.Log($"Критична помилка запису мікрофона: {ex.Message}");
    }
    finally {
        LogService.Instance.Log("Завершено задачу запису мікрофона");
    }
  }
#endif

  }
}
```

FileTransferServices.cs

```
namespace EspHandler.Services {
    class FileTransferServices{
        int TcpPort = 8080;
        public async Task<bool> SendFileAsync(string filePath) {
            string fileName = Path.GetFileName(filePath);
            LogService.Instance.Log($"Початок передачі файлу: {fileName}");
            try{
                using (HttpClient httpClient = new HttpClient()){
                    httpClient.Timeout = TimeSpan.FromSeconds(300);
                    string url =
$"{ESP.Instance.BaseURI}/api/audio/files/upload?filename={SimpleSanitizeFileName(fileName)}";
                    LogService.Instance.Log($"Відправка запиту на: {url}");
                    HttpResponseMessage response = await httpClient.GetAsync(url);
```

```
                    string responseContent = await response.Content.ReadAsStringAsync();
                    LogService.Instance.Log($"Отримано відповідь: {response.StatusCode}, Вміст:
{responseContent}");
                    if (!response.IsSuccessStatusCode) {
                        LogService.Instance.Log($"Помилка ініціалізації передачі файлу:
{responseContent}");
                        await Application.Current.MainPage.DisplayAlert("Помилка", $"Помилка
ініціалізації передачі файлу: {responseContent}", "OK");
                        return false;
                    }
                }
                LogService.Instance.Log("Очікування перед встановленням TCP з'єднання...");
                await Task.Delay(1000);
                using (TcpClient tcpClient = new TcpClient()){
                    var serverEndpoint = new IPEndPoint(IPAddress.Parse(ESP.Instance.IpAddress),
TcpPort);
                    LogService.Instance.Log($"Підключення до TCP:
{ESP.Instance.IpAddress}:{TcpPort}");
                    var connectTask = tcpClient.ConnectAsync(ESP.Instance.IpAddress, TcpPort);
                    if (await Task.WhenAny(connectTask, Task.Delay(5000)) != connectTask) {
                        LogService.Instance.Log("Помилка: час очікування TCP з'єднання
вичерпано");
                        await Application.Current.MainPage.DisplayAlert("Помилка", $"Час
очікування TCP з'єднання вичерпано", "OK");
                        throw new TimeoutException("TCP connection timed out");
                    }
                    if (!tcpClient.Connected) {
                        LogService.Instance.Log("Помилка: не вдалося встановити TCP з'єднання");
                        await Application.Current.MainPage.DisplayAlert("Помилка", $"Не вдалося
встановити TCP з'єднання", "OK");
                        throw new Exception("Failed to establish TCP connection");
                    }
                    LogService.Instance.Log("TCP з'єднання успішно встановлено");
                    using (NetworkStream stream = tcpClient.GetStream())
                    using (FileStream fileStream = File.OpenRead(filePath)) {
                        byte[] buffer = new byte[1024];
                        int bytesRead;
                        long fileSize = fileStream.Length;
                        long totalBytesSent = 0;
                        int progressPercentage = 0;
                        LogService.Instance.Log($"Початок передачі файлу розміром {fileSize}
байт");
                        while ((bytesRead = await fileStream.ReadAsync(buffer, 0, buffer.Length)) > 0)
{
                            await stream.WriteAsync(buffer, 0, bytesRead);
                            totalBytesSent += bytesRead;
                            await Task.Delay(10);
                            int currentProgress = (int)((totalBytesSent * 100) / fileSize);
                            if (currentProgress >= progressPercentage + 10 || currentProgress == 100) {
                                progressPercentage = currentProgress;
                                LogService.Instance.Log($"Прогрес передачі файлу:
```

```
{progressPercentage}% ({totalBytesSent}/{fileSize} байт)");
                        }
                    }
                    await stream.FlushAsync();
                    LogService.Instance.Log($"Файл '{fileName}' успішно передано ({fileSize}
байт)");
                    await Application.Current.MainPage.DisplayAlert("Успіх", $"Файл
'{fileName}' успішно передано", "OK");
                }
                tcpClient.Close();
                LogService.Instance.Log("TCP з'єднання закрито");
            }
            return true;
        }
        catch (Exception ex) {
            LogService.Instance.Log($"Помилка передачі файлу: {ex.GetType().Name}:
{ex.Message}");
            await Application.Current.MainPage.DisplayAlert("Успіх", $"Помилка передачі
файлу: {ex.GetType().Name}: {ex.Message}", "OK");
            return false;
        }
    }
    public async Task<List<string>> GetAudioFileListAsync(){
        LogService.Instance.Log("Отримання списку аудіо файлів...");
        try {
            LogService.Instance.Log($"Запит до {ESP.Instance.AudioFilesEndpoint}");
            var response = await
ESP.Instance.EspHttpClient.GetAsync(ESP.Instance.AudioFilesEndpoint);
            if (response.IsSuccessStatusCode) {
                LogService.Instance.Log("Список аудіо файлів успішно отримано");
                var content = await response.Content.ReadAsStringAsync();
                var list = JsonSerializer.Deserialize<List<string>>(content);
                LogService.Instance.Log($"Знайдено {list?.Count ?? 0} аудіо файлів");
                return JsonSerializer.Deserialize<List<string>>(content) ?? new List<string>();
            }
            LogService.Instance.Log($"Помилка отримання списку аудіо файлів. Код
відповіді: {response.StatusCode}");
            return new List<string>();
        }
        catch (Exception ex) {
            LogService.Instance.Log($"Помилка отримання списку аудіо файлів:
{ex.Message}");
            return new List<string>();
        }
    }
    public async Task<bool> PlayAudioFileAsync(string fileName) {
        LogService.Instance.Log($"Відтворення аудіо файлу: {fileName}");
        try{
            string url =
$"{ESP.Instance.BaseURI}/api/audio/files/play?name={SimpleSanitizeFileName(fileName)}";
            LogService.Instance.Log($"Запит на відтворення: {url}");
```

```
                var response = await ESP.Instance.EspHttpClient.GetAsync(url);
                if (response.IsSuccessStatusCode) {
                    LogService.Instance.Log($"Аудіо файл '{fileName}' успішно відтворюється");
                }
                else {
                    LogService.Instance.Log($"Помилка відтворення аудіо файлу. Код відповіді:
{response.StatusCode}");
                }

                return response.IsSuccessStatusCode;
            }
            catch (Exception ex) {
                LogService.Instance.Log($"Помилка відтворення аудіо файлу: {ex.Message}");
                return false;
            }
        }
        public async Task<bool> DeleteAudioFileAsync(string fileName) {
            LogService.Instance.Log($"Видалення аудіо файлу: {fileName}");
            try{
                string url =
$"{ESP.Instance.BaseURI}/api/audio/files/delete?name={SimpleSanitizeFileName(fileName)}";
                LogService.Instance.Log($"Запит на видалення: {url}");
                var response = await ESP.Instance.EspHttpClient.DeleteAsync(url);
                if (response.IsSuccessStatusCode) {
                    LogService.Instance.Log($"Аудіо файл '{fileName}' успішно видалено");
                }
                else{
                    LogService.Instance.Log($"Помилка видалення аудіо файлу. Код відповіді:
{response.StatusCode}");
                }
                return response.IsSuccessStatusCode;
            }
            catch (Exception ex) {
                LogService.Instance.Log($"Помилка видалення аудіо файлу: {ex.Message}");
                return false;
            }
        }
        public static string SimpleSanitizeFileName(string fileName) {
            if (string.IsNullOrEmpty(fileName))
                return fileName;
            foreach (char c in Path.GetInvalidFileNameChars()){
                fileName = fileName.Replace(c, '_');
            }
            fileName = fileName.Replace(' ', '_');
            return fileName;
        }
    }
}
```

ESP.cs

```
namespace EspHandler.Model{
    class ESP{
```

```
        private string _ipAddress = "";
        private string _port;
        private HttpClient _espHttpClient;
        public ESPConfiguration Config { get; private set; }
        public static ESP Instance { get; } = new ESP();
        public string IpAddress { get => _ipAddress; set { _ipAddress = value; UpdateBaseUri();
} }
        public string Port { get => _port; set { _port = value; UpdateBaseUri(); } }
        public string BaseURI { get; set; }
        public bool EspConnection { get; set; }
        public HttpClient EspHttpClient{
            get{
                if (_espHttpClient == null) {
                    _espHttpClient = new HttpClient();
                    _espHttpClient.Timeout = TimeSpan.FromSeconds(100);
                }
                return _espHttpClient;
            }
        }
        public string AudioEndpoint => $"{BaseURI}/api/audio";
        public string AudioFilesEndpoint => $"{BaseURI}/api/audio/files/list";
        public string StreamingEndpoint => $"{BaseURI}/api/audio/streaming";
        private ESP(){
            UpdateBaseUri();
        }
        private void UpdateBaseUri(){
            BaseURI = $"http://{IpAddress}:{Port}";
        }
        public async Task TestConnection(){
            try{
                var response = await EspHttpClient.GetAsync($"{BaseURI}/api/status");
                EspConnection = response.IsSuccessStatusCode;
            }
            catch{
                await Application.Current.MainPage.DisplayAlert("Помилка", "Помилка
підключення", "OK");
            }
        }
        public async Task<bool> LoadESPConfiguration(){
            await TestConnection();
            if (EspConnection) {
                try{
                    var response = await EspHttpClient.GetAsync($"{BaseURI}/api/config");
                    if (response.IsSuccessStatusCode) {
                        var content = await response.Content.ReadAsStringAsync();
                        Config = JsonSerializer.Deserialize<ESPConfiguration>(content);
                    }
                }
                catch (Exception e) {
                    await Application.Current.MainPage.DisplayAlert("Помилка", "Помилка
отримання конфігурації ESP", "OK");
```

```
                }
            }
            return EspConnection;
        }
        public async Task<bool> AutoTestConnection(){
            try{
                using var udpClient = new UdpClient();
                udpClient.EnableBroadcast = true;
                byte[] message = Encoding.UTF8.GetBytes("ESP32 scanning");
                await udpClient.SendAsync(message, message.Length, "255.255.255.255", 12345);
                using var cts = new CancellationTokenSource(TimeSpan.FromSeconds(10));
                var receiveTask = udpClient.ReceiveAsync();
                if (await Task.WhenAny(receiveTask, Task.Delay(Timeout.Infinite, cts.Token)) ==
receiveTask) {
                    var result = receiveTask.Result;
                    string response = Encoding.UTF8.GetString(result.Buffer);
                    string[] parts = response.Split(" ");
                    string addressPart = parts.LastOrDefault();
                    if (!string.IsNullOrEmpty(addressPart) && addressPart.Contains(":")){
                        string[] addressSplit = addressPart.Split(':');
                        IpAddress = addressSplit[0];
                        int.TryParse(addressSplit[1], out int port);
                        Port = port.ToString();
                        UpdateBaseUri();
                        EspConnection = true;
                        await LoadESPConfiguration();
                        return true;
                    }
                }
                EspConnection = false;
                return false;
            }
            catch (Exception ex) {
                await Application.Current.MainPage.DisplayAlert("Помилка", $"Помилка
автоматичного підключення до ESP: {ex.Message}", "ОК");
                EspConnection = false;
                return false;
            }
        }
    }
}
```