ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНИХ СИСТЕМ ТА ТЕХНОЛОГІЙ

КВАЛІФІКАЦІЙНА РОБОТА
на тему: «ЗАСТОСУНОК ДЛЯ ОБМІНУ ПОВІДОМЛЕННЯМИ В РЕАЛЬНОМУ ЧАСІ НА ОСНОВІ IRC»

на здобуття освітнього ступеня бакалавр
зі спеціальності 126 Інформаційні системи та технології
(код, найменування спеціальності)
освітньо-професійної програми Інформаційні системи та технології
(назва)

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

_____ Тимофій ЯВДЮК
(підпис) *Ім'я, ПРІЗВИЩЕ здобувача*

Виконав: здобувач вищої освіти гр. ІСД-42
Тимофій ЯВДЮК
Ім'я, ПРІЗВИЩЕ

Керівник:
доктор філософії Віктор САГАЙДАК
Ім'я, ПРІЗВИЩЕ

Рецензент:
Ім'я, ПРІЗВИЩЕ

Київ 2026

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут Інформаційних технологій

Кафедра Інформаційних систем та технологій

Ступінь вищої освіти Бакалавр

Спеціальність 126 Інформаційні системи та технології

Освітньо-професійна програма Інформаційні системи та технології

ЗАТВЕРДЖУЮ

Завідувач кафедрою ІСТ

_____ Каміла СТОРЧАК

«____» _____ 2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Тимофій Миколайович Явлюк

(прізвище, ім'я, по батькові здобувача)

1. Тема кваліфікаційної роботи: Застосунок для обміну повідомленнями в реальному часі на основі IRC

керівник кваліфікаційної роботи Віктор САГАЙДАК, доктор філософії,

(Ім'я, ПРІЗВИЩЕ, науковий ступінь, вчене звання)

затверджені наказом Державного університету інформаційно-комунікаційних технологій від «20» лютого 2026 р. №51.

2. Строк подання кваліфікаційної роботи «1» червня 2026 р.

3. Вихідні дані до кваліфікаційної роботи:

Технічна документація та специфікації мережевих протоколів передачі даних у реальному часі.

Програмні та технологічні інструментарії: мова програмування Python з бібліотеками, фреймворк Capacitor.

Нормативні та методичні матеріали з інформаційної безпеки, підходи до криптографічного захисту інформації.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Теоретичні засади побудови систем обміну повідомленнями та огляд чинних аналогів.
2. Проєктування архітектури системи, структури бази даних та криптографічного захисту.
3. Програмна реалізація клієнт-серверного застосунку, розробка інтерфейсу та тестуванн.

5. Перелік ілюстративного матеріалу: *презентація*

6. Дата видачі завдання «20» лютого 2026

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Формування мети, завдань дослідження та написання вступу до кваліфікаційної роботи	13.03.26 - 24.03.26	Виконано
2	Теоретико-методичні засади проєктування систем обміну повідомленнями в реальному часі	25.03.26 - 28.03.26	Виконано
3	Системний аналіз вимог, вибір та обґрунтування технологічного стека розробки	29.03.26 - 09.04.26	Виконано
4	Дослідження та проєктування архітектури мобільного застосунку на базі протоколу irc	10.04.26 - 19.04.26	Виконано
5	Проєктування бази даних, алгоритмів маршрутизації, механізмів безпеки та інтерфейсу користувача	20.04.26 - 25.04.26	Виконано
6	Практична реалізація та тестування системи обміну повідомленнями	26.04.26 - 28.04.26	Виконано
7	Оформлення загальних висновків за результатами дослідження	29.04.26 - 02.05.26	Виконано
8	Завершальне тестування застосунку, перевірка функціональності, підготовка демонстраційних матеріалів	03.05.26 - 21.05.26	Виконано

Здобувач вищої освіти _____ Тимофій ЯВДЮК
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

Керівник
кваліфікаційної роботи _____ Віктор САГАЙДАК
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 60 стор., 21 рис., 2 табл., 20 джерел.

Мета роботи – розробка сучасного захищеного мобільного застосунку для ОС Android з функціоналом обміну повідомленнями в реальному часі на базі протоколу IRC з використанням платформних веб технологій та сервера на Python з метою забезпечення надійної, швидкої та конфіденційної комунікації користувачів.

Об'єктом дослідження є процеси інформаційного обміну в реальному часі в межах мобільних клієнт-серверних систем.

Предметом дослідження є методи та програмні засоби розробки мобільного месенджера на базі протоколу IRC з використанням мови Python та фреймворку Capacitor.

Короткий зміст роботи: У роботі досліджено принципи функціонування протоколу IRC та сучасні підходи до розробки застосунків для обміну повідомленнями. Проаналізовано існуючі рішення та виявлено їхні переваги й недоліки. Розроблено архітектуру клієнт-серверної системи зв'язку: серверну частину реалізовано мовою програмування Python, а клієнтський мобільний застосунок з сучасним дизайном створено за допомогою веб технологій (JavaScript) та інтегровано під ОС Android через фреймворк Capacitor. Реалізовано алгоритми наскрізного шифрування для забезпечення безпеки та конфіденційності даних користувачів. Проведено комплексне тестування розробленої системи на швидкодію, безпеку та стабільність роботи.

КЛЮЧОВІ СЛОВА: МЕСЕНДЖЕР, ОБМІН ПОВІДОМЛЕННЯМИ, ПРОТОКОЛ IRC, ANDROID-ЗАСТОСУНОК, PYTHON-СЕРВЕР, CAPACITOR, JAVASCRIPT, ШИФРУВАННЯ ДАНИХ, КЛІЄНТ-СЕРВЕРНА АРХІТЕКТУРА.

ABSTRACT

Text part of the qualification work for the bachelor's degree: 60 pages, 21 pictures, 2 tables, 20 sources.

The objective of the work is the development of a modern, secure mobile application for Android OS with real-time messaging functionality based on the IRC protocol, using platform web technologies and a Python server to ensure reliable, fast, and confidential communication between users.

The object of research is the processes of real-time information exchange within mobile client-server systems.

The subject of research is the methods and software tools for developing a mobile messenger based on the IRC protocol using the Python programming language and the Capacitor framework.

Summary: The paper investigates the operational principles of the IRC protocol and modern approaches to developing messaging applications. Existing solutions are analyzed, and their advantages and disadvantages are identified. The architecture of a client-server communication system has been developed: the server side is implemented using the Python programming language, while the client mobile application with a modern design is created using web technologies (JavaScript) and integrated into Android OS via the Capacitor framework. End-to-end encryption algorithms have been implemented to ensure the security and confidentiality of user data. Comprehensive testing of the developed system for performance, security, and stability has been conducted.

KEYWORDS: MESSENGER, MESSAGING, IRC PROTOCOL, ANDROID APPLICATION, PYTHON SERVER, CAPACITOR, JAVASCRIPT, DATA ENCRYPTION, CLIENT-SERVER ARCHITECTURE.

ЗМІСТ

Стор.

ВСТУП

У сучасних умовах глобальної цифровізації та стрімкого розвитку інфокомунікаційних технологій, що охоплюють усі сфери людської діяльності — від повсякденного приватного спілкування до побудови складних корпоративних мереж — особливого значення набуває питання забезпечення стабільного та ефективного обміну даними в реальному часі. Стрімкий перехід користувачів до мобільних платформ і потреба у відкритих, масштабованих архітектурах спонукають до переосмислення класичних мережевих протоколів та їх адаптації до вимог сучасних екосистем. З огляду на зростаючу потребу в інструментах, що поєднують високу швидкість передачі повідомлень із гнучкістю розгортання на різних пристроях, розробка спеціалізованих мобільних застосунків на базі перевірених часом протоколів стає надзвичайно актуальним завданням для IT-індустрії

Актуальність дослідження зумовлена низкою проблем, що притаманні сучасному сегменту систем миттєвого обміну повідомленнями та інструментам їх розробки. Серед них — висока залежність користувачів від закритих пропрієтарних платформ, які часто мають надлишковий функціонал і складну архітектуру; складність адаптації класичних мережевих протоколів до вимог сучасних мобільних операційних систем; а також відсутність уніфікованих і водночас легковажних рішень для створення крос платформних застосунків, що здатні ефективно взаємодіяти з кастомними серверними рішеннями на мові Python. Окремим викликом є потреба у створенні гнучких інтерфейсів, які б забезпечували комфортну роботу з текстовими протоколами в умовах обмежених ресурсів мобільних пристроїв.

Мета роботи – розробка сучасного захищеного мобільного застосунку для ОС Android з функціоналом обміну повідомленнями в реальному часі на базі протоколу IRC з використанням крос платформних веб технологій та сервера на Python з метою забезпечення надійної, швидкої та конфіденційної комунікації користувачів.

Об'єктом дослідження є процеси інформаційного обміну в реальному часі в межах мобільних клієнт-серверних систем.

Предметом дослідження є методи та програмні засоби розробки мобільного месенджера на базі протоколу IRC з використанням мови Python та фреймворку Capacitor.

Методи дослідження. Під час написання кваліфікаційної роботи були використані методи системного аналізу та порівняльного узагальнення, методи архітектурного моделювання клієнт-серверних систем, принципи об'єктно-орієнтованого проєктування програмного забезпечення, а також методи функціонального та навантажувального тестування.

Наукові завдання:

1. Проаналізувати сучасні технології, мережеві протоколи та архітектурні підходи до побудови клієнт-серверних систем обміну повідомленнями в реальному часі;
2. Дослідити можливості та переваги використання текстового протоколу IRC у порівнянні з сучасними аналогами (WebSocket, HTTP Polling) для організації оптимізованої та швидкої передачі даних;
3. Сформувати функціональні та нефункціональні вимоги до мобільного застосунку;
4. Розробити клієнт-серверну архітектуру системи, спроєктувати структуру реляційної бази даних та визначити математичні моделі (алгоритми) управління життєвим циклом сесій і маршрутизації повідомлень;
5. Дослідити та спроєктувати механізми багаторівневої інформаційної безпеки: захист транспортного каналу (TLS/SSL) та криптографічний захист даних у стані спокою (Data-at-Rest) методами гібридного шифрування;
6. Програмно реалізувати серверну частину месенджера мовою Python із застосуванням багатопотоковості та клієнтський мобільний застосунок за допомогою кросплатформного фреймворку Capacitor;

7. Провести комплексне тестування розробленого програмного рішення для оцінки його продуктивності, стійкості до розривів з'єднання та рівня безпеки.

Наукова новизна одержаних результатів. У ході дослідження розроблено та оптимізовано підхід до побудови мобільних систем зв’язку на базі протоколу IRC із застосуванням фреймворку Capacitor. Запропонована модель поєднання високорівневої мови Python для серверної частини та веб технологій для мобільного клієнта забезпечує високу швидкість обробки повідомлень та легку адаптивність інтерфейсу під ОС Android без втрати продуктивності, притаманної класичним нативним рішенням.

Практична значущість одержаних результатів. Запропонована архітектура та розроблений мобільний застосунок забезпечують ефективне рішення для організації конфіденційного обміну повідомленнями в реальному часі. Створений програмний комплекс може бути використаний як автономна система корпоративного або приватного зв’язку, що легко масштабується та не залежить від сторонніх пропрієтарних сервісів.

Апробація результатів та публікації. Основні положення і результати бакалаврської роботи доповідались на науково практичних конференціях, що проходили на базі Державного університету інформаційно-комунікаційних технологій:

1. Явдюк Т.М. - «ВИКОРИСТАННЯ ШТУЧНОГО ІНТЕЛЕКТУ ДЛЯ ОПТИМІЗАЦІЇ РУХУ ТРАНСПОРТУ В "РОЗУМНОМУ МІСТІ"». Тези доповіді на III Всеукраїнській науково-технічній конференції «Технологічні горизонти: дослідження та застосування інформаційних технологій для технологічного прогресу України і світу».

2. Явдюк Т.М. - «ВИКОРИСТАННЯ ПРОТОКОЛУ IRC ЯК СТІЙКОГО РЕЗЕРВНОГО КАНАЛУ ЗВ'ЯЗКУ ДЛЯ КРИТИЧНОЇ ІТ-ІНФРАСТРУКТУРИ». Тези доповіді на VII Всеукраїнській науково-технічній конференції «Сучасний стан та перспективи розвитку IoT».

1. ТЕОРЕТИКО-МЕТОДИЧНІ ЗАСАДИ ПРОЄКТУВАННЯ СИСТЕМ ОБМІНУ ПОВІДОМЛЕННЯМИ В РЕАЛЬНОМУ ЧАСІ

1.1. Аналіз сучасних технологій та протоколів комунікації в реальному часі

При проєктуванні систем обміну повідомленнями (месенджерів) фундаментальним питанням є вибір транспортного рівня, який визначає механізм обміну даними між клієнтським застосунком та сервером. Класична архітектура вебзастосунків побудована на базі протоколу HTTP, що функціонує за парадигмою «запит-відповідь». Проте для систем, що вимагають комунікації в реальному часі (Real-Time Communications), такий підхід є вкрай неефективним. Для своєчасного отримання нових повідомлень клієнтський застосунок змушений використовувати техніку Long Polling - безперервне циклічне опитування сервера. Це призводить до швидкого виснаження заряду акумулятора мобільних пристроїв та перевантаження каналу зв'язку надлишковим службовим трафіком (HTTP-заголовками), навіть за умови відсутності активного обміну корисними даними(рис 1.1).

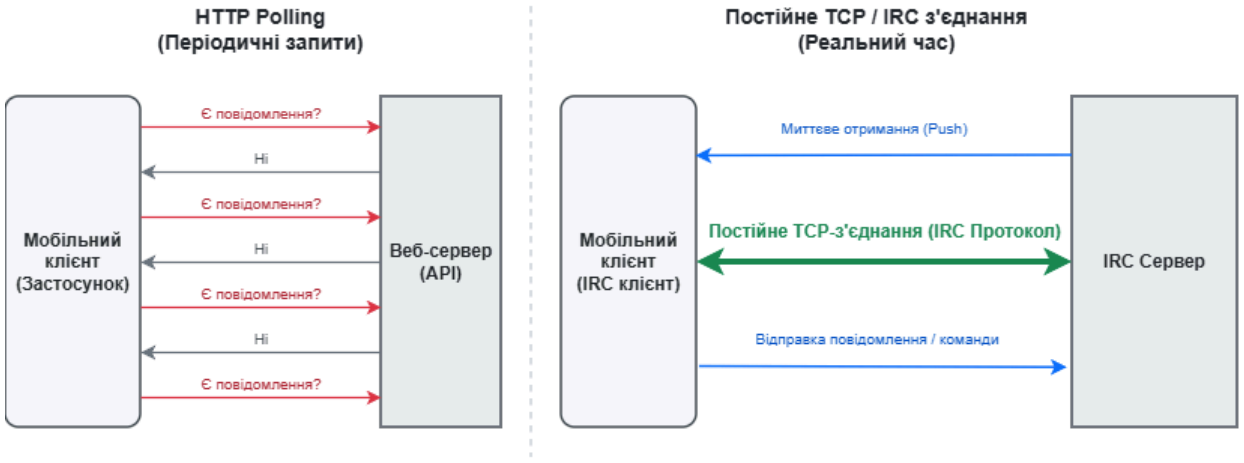


Рис 1.1 - Різниця між HTTP Polling та постійним TCP-з'єднанням

Для вирішення цієї проблеми сучасна індустрія програмного забезпечення перейшла до використання технології WebSockets (стандарт RFC 6455). Цей

протокол є значно ефективнішим, оскільки після початкового «рукотискання» (handshake) він підтримує TCP-з'єднання відкритим. Однак у контексті розробки месенджерів WebSocket виступає лише низькорівневим транспортним каналом, позбавленим вбудованої бізнес-логіки. На рівні протоколу відсутні концепції «каналів», «користувачів» чи «приватних повідомлень». Відповідно, розробнику необхідно проєктувати складну логіку маршрутизації та інкапсулювати кожну транзакцію в об'ємні JSON-структури, що збільшує накладні витрати на серіалізацію та парсинг даних.

Ефективною альтернативою в даному контексті є використання текстового протоколу IRC (Internet Relay Chat, RFC 1459), що функціонує безпосередньо поверх базових TCP-сокетів. Оскільки архітектура IRC від початку створювалася для систем багатокористувацького чату, протокол має стандартизовану та оптимізовану систему команд. Наприклад, клієнт ініціює команду JOIN, і сервер автоматично додає його до відповідного каналу; команда PRIVMSG забезпечує безпосередню маршрутизацію тексту до отримувача. Накладні витрати при цьому є мінімальними: замість багаторівневих JSON-об'єктів передається стандартизований текстовий рядок, що завершується символами \r\n(рис 1.2)[3].

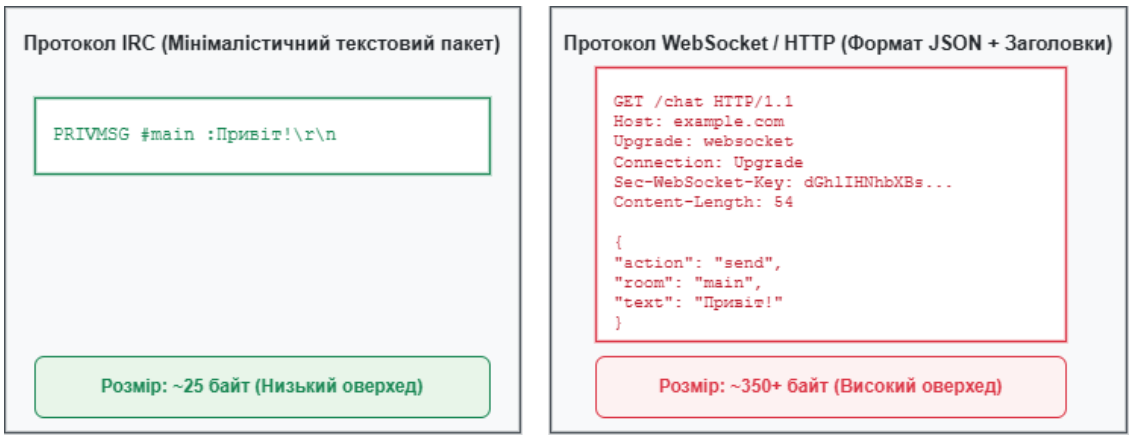


Рис. 1.2 - Порівняння розміру пакетів IRC та WebSocket

Використання прямих TCP-з'єднань у мобільному застосунку забезпечує суттєвий приріст продуктивності. Обчислювальні ресурси пристрою мінімально залучаються до обробки пакетів, а затримка передачі даних залишається низькою навіть в умовах нестабільного мобільного зв'язку (наприклад, у мережах 3G). Крім того, така архітектура забезпечує гнучкість управління безпекою: адміністратор

системи має можливість активувати криптографічний захист трафіку (SSL/TLS на порту 6697) для роботи у відкритих мережах, або ж деактивувати його для досягнення максимальної пропускної здатності в ізольованих корпоративних середовищах.

Основним критичним недоліком класичного протоколу IRC є відсутність механізмів буферизації повідомлень для користувачів, які перебувають поза мережею (офлайн). Однак у межах даного кваліфікаційного дослідження ця проблема вирішується шляхом проєктування авторської підсистеми Memo та її інтеграції з реляційною базою даних SQLite, що забезпечує асинхронну доставку відкладених повідомлень одразу після авторизації отримувача в системі.

1.2. Огляд та порівняльний аналіз існуючих мобільних месенджерів

У процесі розгортання корпоративних або приватних систем комунікації організації найчастіше розглядають можливість використання готових комерційних рішень, лідерами серед яких є месенджери Telegram та Signal. Проте детальний аналіз їхньої архітектури виявляє низку суттєвих обмежень, що ускладнюють їх застосування в умовах підвищених вимог до конфіденційності та контролю над даними.

Месенджер Telegram характеризується високою швидкістю роботи та ергономічним інтерфейсом. Однак серверна частина цієї системи базується на закритому вихідному коді (proprietary software). Це унеможливлює розгортання автономного бекенду на власних апаратних потужностях підприємства (on-premise), особливо в ізольованих мережах без доступу до глобальної мережі Інтернет. Зберігання історії комунікацій на хмарних серверах компанії-розробника за замовчуванням створює потенційні ризики компрометації конфіденційних даних корпоративного сектору.

З іншого боку, месенджер Signal позиціонується як еталон приватного спілкування завдяки використанню протоколу наскрізного шифрування Double Ratchet[9]. Проте такий рівень безпеки супроводжується функціональними

обмеженнями. По-перше, система вимагає обов'язкової прив'язки облікового запису до номера мобільного телефону, що нівелює можливість повної анонімності користувачів. По-друге, криптографічні алгоритми жорстко інтегровані в архітектуру застосунку без можливості їх деактивації. У випадках, коли метою є створення максимально швидкого каналу зв'язку всередині довіреної локальної мережі підприємства, надлишкові математичні обчислення, пов'язані з шифруванням, призводять до невиправданого навантаження на процесори мобільних пристроїв та пришвидшеного розряду акумуляторів[10].

Альтернативним сегментом ринку є класичні клієнтські застосунки для протоколу IRC (наприклад, AndroIRC). Їхньою безперечною перевагою є здатність підключатися до будь-якого автономного сервера. Проте більшість таких рішень володіють застарілим користувацьким інтерфейсом і, що найбільш критично, не підтримують коректну обробку втрати мережевого з'єднання. При короткочасних розривах зв'язку всі вхідні повідомлення безповоротно втрачаються.

Наведена порівняльна діаграма архітектури месенджерів у рис 1.3

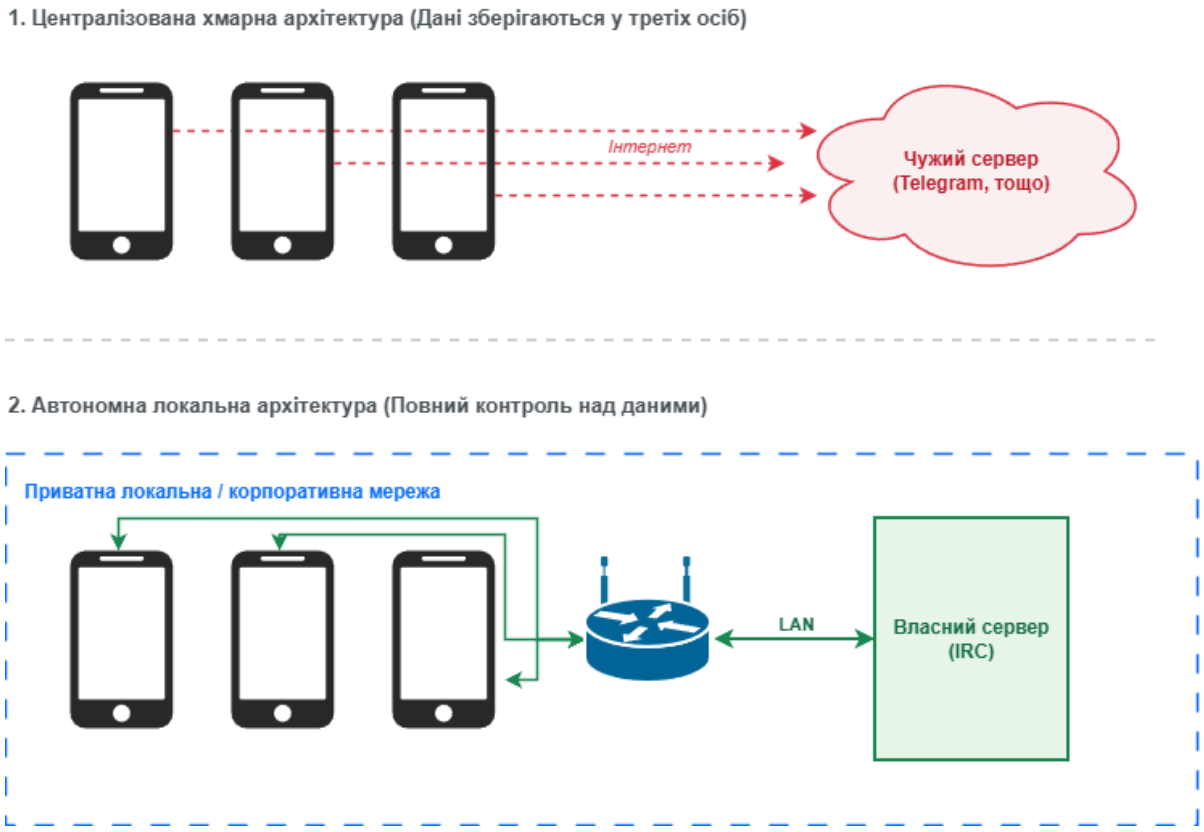


Рис. 1.3 - Порівняльна діаграма архітектури хмарних месенджерів та автономного сервера

Критичний аналіз існуючих рішень дозволяє сформулювати основну науково-практичну проблему даного дослідження: наявність потреби у розробці мобільнго клієнт-серверного застосунку, який би поєднував децентралізовану та незалежну природу класичного протоколу IRC із сучасним функціоналом збереження повідомлень та зручним інтерфейсом. Порівняння існуючих систем наведено в табл. 1.1.

Таблиця 1.1

Порівняння функціоналу існуючих систем та розробленого месенджера

Функція	Telegram	Signal	Наш IRC-месенджер
Повністю власний сервер	Ні	Ні	Так
Анонімність без номера телефону	Так	Ні	Так
Офлайн повідомлення	Так	Так	Так

Визначальною парадигмою розроблюваної системи є архітектурна гнучкість. Замість примусового нав'язування ресурсомістких алгоритмів безпеки, механізми захисту винесено в опціональний конфігураційний блок. Адміністратор системи самостійно визначає необхідність активації модулів автентифікації (NickServ), транспортного шифрування (TLS/SSL) та криптографічного захисту бази даних.

1.3. Обґрунтування вибору технологічного стека та програмних засобів розробки

Проєктування ефективної інформаційної системи вимагає ретельного підбору технологічних інструментів та методів їх застосування. Перед розроблюваним застосунком стояло завдання створення високопродуктивної серверної частини для підтримки постійних TCP-з'єднань, а також оптимізованого

мобільного клієнта, здатного стабільно функціонувати на пристроях із низькими апаратними характеристиками.

Для реалізації серверного ядра було обрано об'єктно-орієнтовану мову програмування Python. Методичною перевагою такого вибору є наявність потужних вбудованих модулів низькорівневої мережевої взаємодії (socket, ssl) та підтримка багатопотоковості (threading). Замість використання великовагових вебсерверів, архітектура передбачає пряме прослуховування TCP-порту. Обробка кожного клієнтського з'єднання ізолюється в окремому системному потоці. Цей метод усуває проблему блокування вводу-виводу: нестабільність з'єднання одного клієнта не впливає на процеси маршрутизації повідомлень інших користувачів.

Для розв'язання задачі персистентності даних (зберігання історії, облікових записів, налаштувань) застосовано реляційну СУБД SQLite. Цей вибір обґрунтований вимогами до автономності системи: SQLite не потребує розгортання окремого серверного процесу, що максимально спрощує імплементацію продукту. Для забезпечення безпеки та зручності взаємодії програмного коду з базою даних використано методологію об'єктно-реляційного відображення (ORM) за допомогою бібліотеки SQLAlchemy[5]. Це дозволяє абстрагуватися від SQL-синтаксису, оперуючи таблицями як об'єктами класів, що значно знижує ризик виникнення вразливостей типу SQL-injection.

У розробці клієнтської частини застосовано парадигму кросплатформної розробки на базі фреймворку Capacitor[4]. Цей методичний підхід дозволяє трансформувати стандартні вебтехнології (HTML5, CSS3, JavaScript) у нативні мобільні застосунки (APK для Android). Робота інтерфейсу забезпечується через оптимізований компонент WebView, який зберігає доступ до апаратних функцій пристрою.

Принциповим архітектурним рішенням стала відмова від використання важких реактивних фреймворків (React, Angular). Технологія Virtual DOM, яка лежить в їх основі, потребує значних обсягів оперативної пам'яті[12]. Натомість, керування інтерфейсом розроблено на базовому (ванільному) JavaScript із застосуванням методів прямої маніпуляції DOM-деревом. Такий підхід мінімізує

обчислювальні витрати, гарантуючи миттєвий відгук елементів управління та стабільну частоту оновлення кадрів під час перегляду об'ємної історії повідомлень.

Для реалізації завдань криптографічного захисту даних у стані спокою (Data-at-Rest) інтегровано бібліотеку cryptography. Захист архівних повідомлень на диску реалізується методами гібридного шифрування (комбінація симетричного алгоритму AES-256 та асиметричного RSA-4096), що гарантує цілісність та конфіденційність інформації навіть у разі фізичної компрометації сервера.

У рис 1.4 наведено обраний технологічний стек.

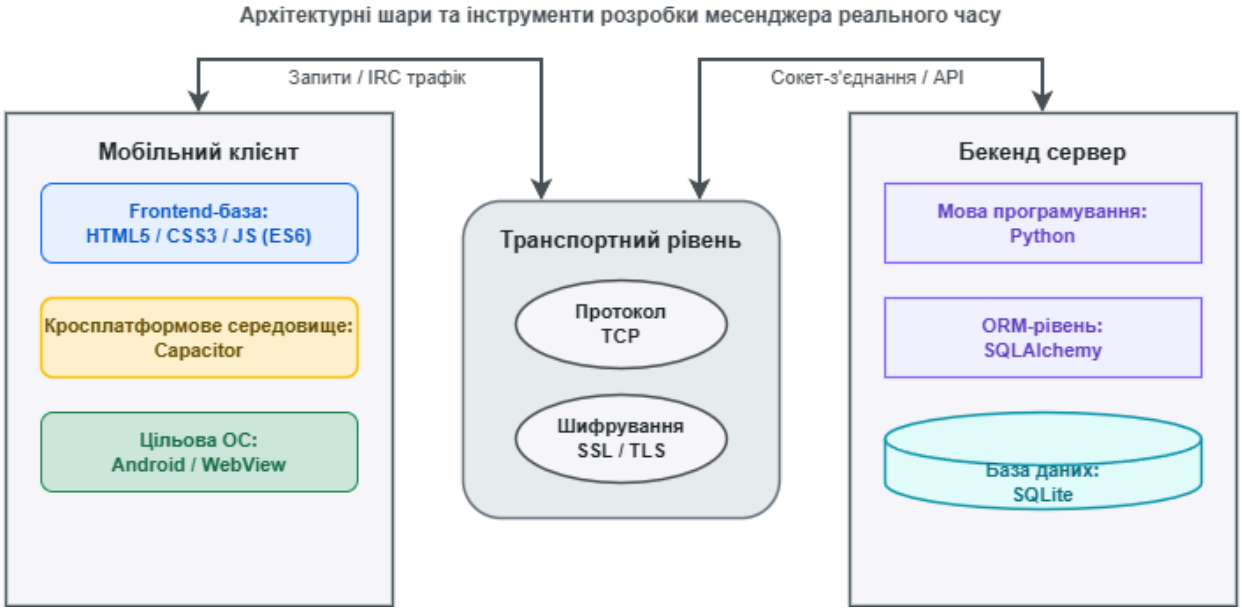


Рис. 1.4 - Схема обраного технологічного стека клієнт-серверної архітектури

Висновки до розділу 1

У першому розділі кваліфікаційної роботи досліджено теоретико-методичні засади проєктування клієнт-серверних систем обміну повідомленнями в реальному часі. Проведений аналіз транспортних протоколів показав, що використання текстового протоколу IRC поверх прямих TCP-сокетів забезпечує значно вищу продуктивність та менші накладні витрати порівняно з традиційними HTTP-запитами чи технологією WebSockets, оскільки виключає потребу у складній серіалізації даних.

Критичний огляд існуючих комерційних рішень (Telegram, Signal) виявив низку концептуальних недоліків, пов'язаних із централізацією зберігання даних, закритістю серверного коду та жорсткою прив'язкою до ідентифікаторів користувачів (номерів телефонів). На основі цього було сформульовано проблематику дослідження та обґрунтовано необхідність створення автономного, децентралізованого месенджера з гнучкою системою безпеки.

Сформовано методологічний підхід та обрано технологічний стек для реалізації системи. Для розробки бекенду обрано мову програмування Python із застосуванням багатопотокової обробки сокетів, бази даних SQLite та технології ORM (SQLAlchemy). Для розробки клієнтської частини обґрунтовано використання кросплатформної парадигми (фреймворк Capacitor) та методу прямої маніпуляції DOM-деревом без використання реактивних фреймворків. Обрана сукупність методів та технологій створює надійне підґрунтя для подальшого системного проєктування та практичної реалізації застосунку у наступних розділах роботи.

2. ДОСЛІДЖЕННЯ ТА ПРОЄКТУВАННЯ АРХІТЕКТУРИ МОБІЛЬНОГО ЗАСТОСУНКУ НА БАЗІ ПРОТОКОЛУ IRC

2.1. Системний аналіз функціональних та нефункціональних вимог до системи

Проєктування будь-якої складної інформаційної системи розпочинається з етапу системного аналізу предметної області та формалізації вимог[19]. На основі виявлених у першому розділі архітектурних та безпекових недоліків існуючих комерційних рішень, було сформовано концептуальну мету дослідження: проєктування та розробка мобільного клієнт-серверного месенджера на базі TCP-сокетів та протоколу IRC, що забезпечує децентралізований обмін повідомленнями в реальному часі, гнучке управління каналами зв'язку та гарантовану доставку відкладених повідомлень.

У результаті декомпозиції головної мети було визначено наступний комплекс ключових функціональних вимог до програмного продукту:

— ініціалізація клієнтської сесії та підтримка стійкого сокетного підключення;
— опціональна реєстрація та криптографічна авторизація користувачів через підсистему NickServ;
— створення загальних кімнат (каналів) та реалізація рольової моделі управління привілеями;
— маршрутизація текстових пакетів у багатокористувацькі групи та приватні діалоги;
— автоматизована буферизація та збереження повідомлень для офлайн-користувачів;
— наявність панелі адміністрування для конфігурації параметрів безпеки сервера.

Критично важливою функціональною особливістю системи є її архітектурна адаптивність, що дозволяє адміністратору змінювати конфігурацію транспортного

рівня (перемикання між захищеним та відкритим режимами підключення) залежно від рівня довіри до інфраструктури мережі розгортання.

Окрім функціональних вимог, під час дослідження було сформульовано низку нефункціональних вимог (технічних обмежень та метрик якості):

— мінімізація затримок (latency) під час обробки та маршрутизації мережевих запитів;
— стійкість (fault tolerance) до короткочасних розривів мобільного інтернет-з'єднання;
— забезпечення багаторівневої безпеки (можливість шифрування транспортного каналу TLS/SSL та локальної бази даних);
— оптимізація споживання оперативної пам'яті для забезпечення плавності роботи кросплатформного інтерфейсу на пристроях базового рівня;
— здатність серверного ядра до масштабування шляхом виділення системних потоків (threads) для нових підключень.

Основні функціональні можливості системи структуровано та представлено в табл. 2.1.

Таблиця 2.1

Основні функціональні можливості IRC-месенджера

Функція	Опис
Підключення до сервера	Встановлення TCP-з'єднання та вибір вільного нікнейма
Авторизація користувача	Захист нікнеймів паролем за допомогою бази даних (NickServ)
Маршрутизація даних	Ретрансляція текстових пакетів між підключеними клієнтами
Управління привілеями	Надання прав доступу від читача до власника каналу
Офлайн-доставка (Memo)	Буферизація повідомлень у базі для користувачів поза мережею
Налаштування безпеки	Зміна параметрів шифрування транспортного та прикладного рівнів

Для успішного проєктування застосунку було побудовано абстрактну модель взаємодії між клієнтом та системою. Взаємодія здійснюється через графічний мобільний інтерфейс, де дії користувача програмно конвертуються у стандартизовані команди протоколу IRC, після чого сервер виконує їх синтаксичний аналіз та маршрутизацію(рис 2.1).

Рис. 2.1 - Загальна логіка взаємодії клієнта з IRC-сервером

Логіка функціонування передбачає, що після запуску мобільний клієнт ініціює підключення. За умови активації механізмів безпеки, система проводить SSL-рукотискання. Інформація про стан системи (профілі, канали, привілеї та архіви повідомлень) синхронізується з реляційною базою даних. Це забезпечує збереження контексту комунікації при перепідключенні та формує фундамент для аналізу загальної активності(рис. 2.2).

Рис. 2.2 - Діаграма станів (State Machine) сесії користувача

Окрім цього, дослідження виявило необхідність проєктування підсистеми відкладеної доставки. Якщо адресат знаходиться поза межами мережі, пакет перехоплюється модулем Memo та записується до бази даних. Автоматизований процес вивантаження гарантує доставку цих даних одразу після наступної успішної автентифікації клієнта.

Проведений системний аналіз вимог дозволив чітко окреслити архітектурні межі майбутньої системи та сформувати концептуальну модель її функціонування, що є базисом для подальшого програмного проєктування структурних компонентів.

2.2. Розробка клієнт-серверної архітектури та структури бази даних

На основі результатів попереднього аналізу було обґрунтовано доцільність використання класичної дворівневої клієнт-серверної архітектури. У цій парадигмі мобільний додаток виконує роль "тонкого клієнта", відповідаючи виключно за рендеринг графічного інтерфейсу та первинну санітизацію вводу

користувача. Уся бізнес-логіка, маршрутизація пакетів та авторизаційні перевірки інкапсульовані виключно на серверній стороні. Такий архітектурний патерн унеможливлює розсинхронізацію станів та захищає систему від компрометації з боку модифікованих клієнтських застосунків(рис. 2.3).

Рис. 2.3 - Логічна архітектура клієнт-серверної взаємодії

Серверне ядро спроєктовано на базі пулу ізольованих обробників. При надходженні нового клієнтського підключення система динамічно виділяє для цього сокета окремий системний потік (Thread). Такий багатопотоковий підхід усуває проблему блокування вводу-виводу (I/O blocking): низька пропускна здатність мережі одного клієнта не спричиняє затримок у маршрутизації повідомлень для інших користувачів.

Для розв'язання задачі персистентності даних було обрано реляційну СУБД SQLite. Цей вибір є оптимальним для автономних систем зв'язку, оскільки усуває потребу у розгортанні та адмініструванні окремого серверного процесу бази даних. Управління даними формалізовано через рівень абстракції — технологію об'єктно-реляційного відображення (ORM) SQLAlchemy. Це мінімізує ризики ін'єкційних атак (SQL Injection) шляхом автоматизованої параметризації запитів.

У процесі проєктування сховища даних було проведено нормалізацію та розроблено реляційну схему, що відображає сутності предметної області(рис. 2.4).

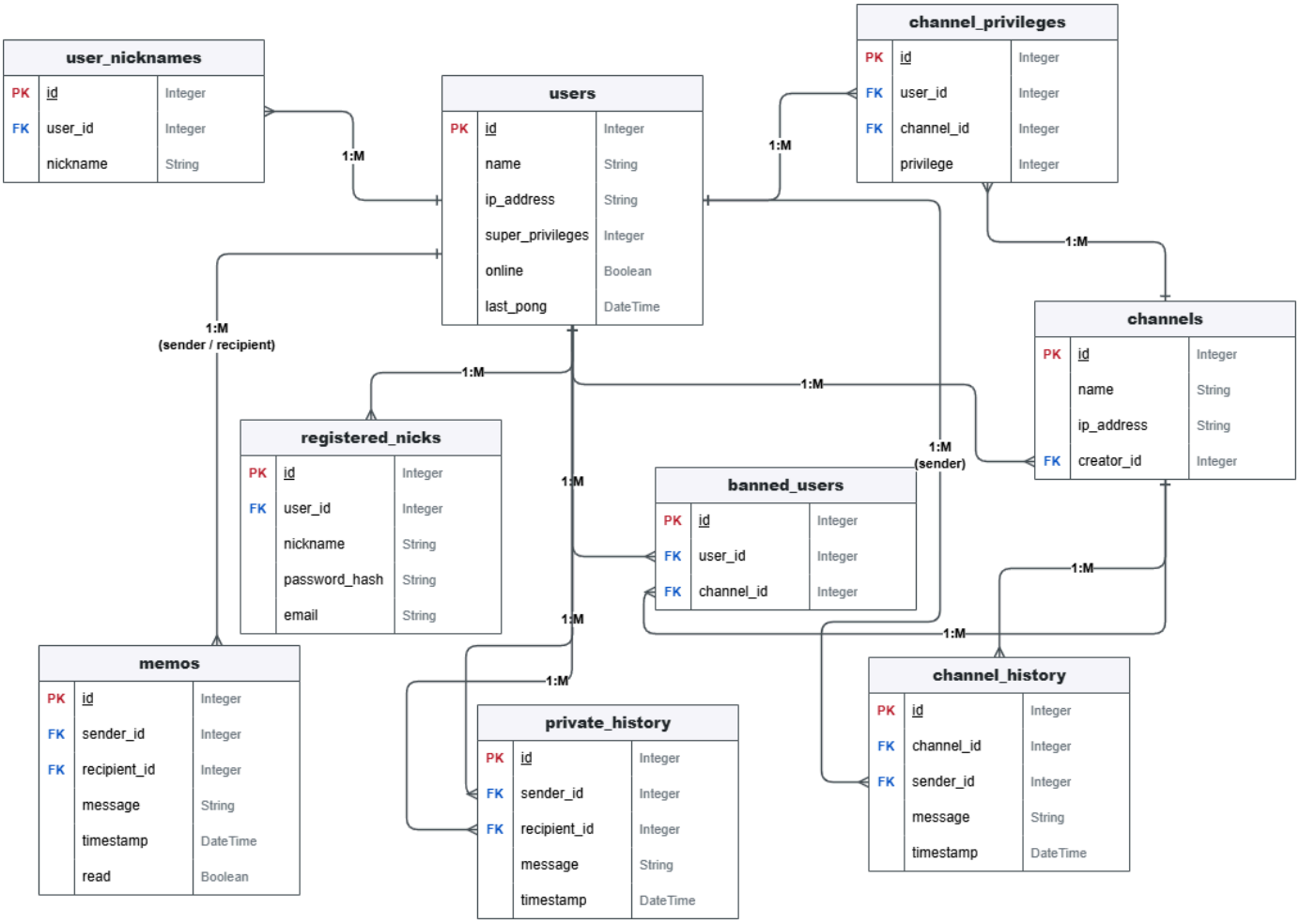


Рис. 2.4 - ER-діаграма (Entity-Relationship) реляційної бази даних месенджера

Логічна структура спроєктованої бази даних декомпонована на 9 взаємопов'язаних моделей:

1. Модель User (таблиця users) - центральна сутність, що акумулює ідентифікатор, ім'я, IP-адресу, статус у мережі та мітку активності. Утворює зв'язки "один-до-багатьох" з більшістю інших таблиць.
2. Модель UserNickname (таблиця user_nicknames) - реалізує підтримку множинних альтернативних псевдонімів.
3. Модель RegisteredNick (таблиця registered_nicks) - забезпечує роботу підсистеми NickServ, зберігаючи ідентифікатори та криптографічні SHA-256 хеші паролів.
4. Модель Channel (таблиця channels) - описує віртуальні кімнати з фіксацією творця (creator_id) та зв'язками для управління доступом.

5. Модель ChannelPrivilege (таблиця channel_privileges) - таблиця зв'язку ("багато-до-багатьох"), що реалізує багаторівневу рольову модель (від 0 до 6 рівнів привілеїв).
6. Модель BannedUser (таблиця banned_users) - модуль обмеження доступу для формування "чорних списків".
7. Модель Memo (таблиця memos) - сутність підсистеми офлайн-доставки, що зв'язує відправника та отримувача з текстом і статусом прочитання.
8. Модель ChannelHistory (таблиця channel_history) - журнал публічних комунікацій.
9. Модель PrivateHistory (таблиця private_history) - архів конфіденційних діалогів між двома вузлами мережі.

2.3. Моделювання процесів маршрутизації повідомлень та управління сесіями

Ефективність сервера обміну повідомленнями визначається оптимізованістю його ядра маршрутизації. Для алгоритмізації цього процесу життєвий цикл сесії клієнта було змодельовано як математичну модель скінченного автомата (State Machine).

При ініціалізації з'єднання автомат переходить у початковий стан. Після базового TCP-рукотискання (та опціонального встановлення TLS-тунелю) сесія набуває статусу очікування авторизації. На цьому етапі спрацьовує програмний блокувальник маршрутизації: сервер ігнорує будь-які команди окрім ідентифікаційних (NICK, USER). За умови активованої підсистеми NickServ, відбувається верифікація криптографічного хешу з бази даних. Тільки після валідації автомат переводить сесію в активний стан(рис. 2.5).

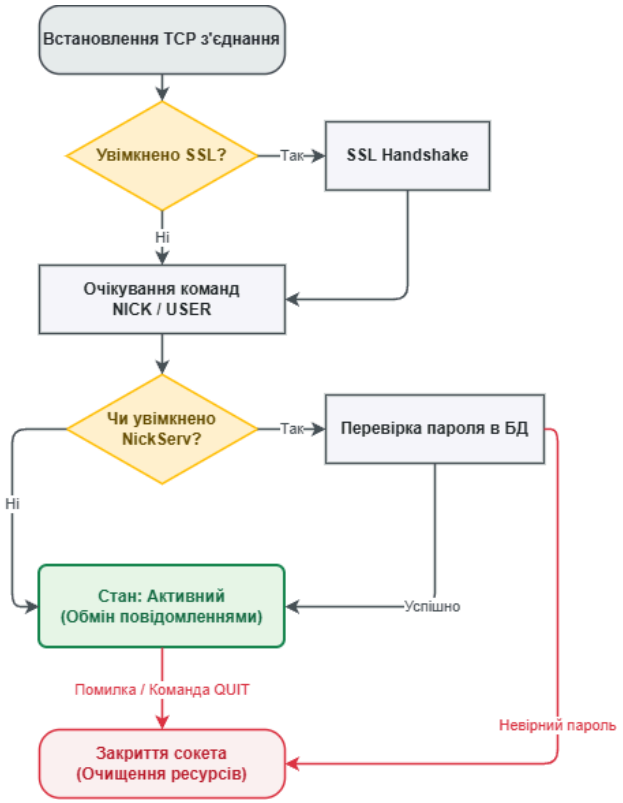


Рис. 2.5 - Алгоритм управління життєвим циклом сесії користувача

В активному стані запускається парсер вхідного байтового потоку. Оскільки протокол IRC є текстовим, сервер сегментує потік за маркерами завершення рядка (\r\n) та класифікує кожну команду за префіксом.

Модель маршрутизації ділиться на два базові сценарії: Multicast (групова розсилка) та Unicast (приватна доставка). Якщо адресат є каналом, алгоритм проводить перевірку ролі відправника в таблиці channel_privileges. У разі успіху формується список цільових сокетів, і пакет ітеративно транслюється кожному учаснику. У випадку Unicast маршрутизації, ядро виконує пошук цільового сокета в оперативній пам'яті сервера(рис 2.6).

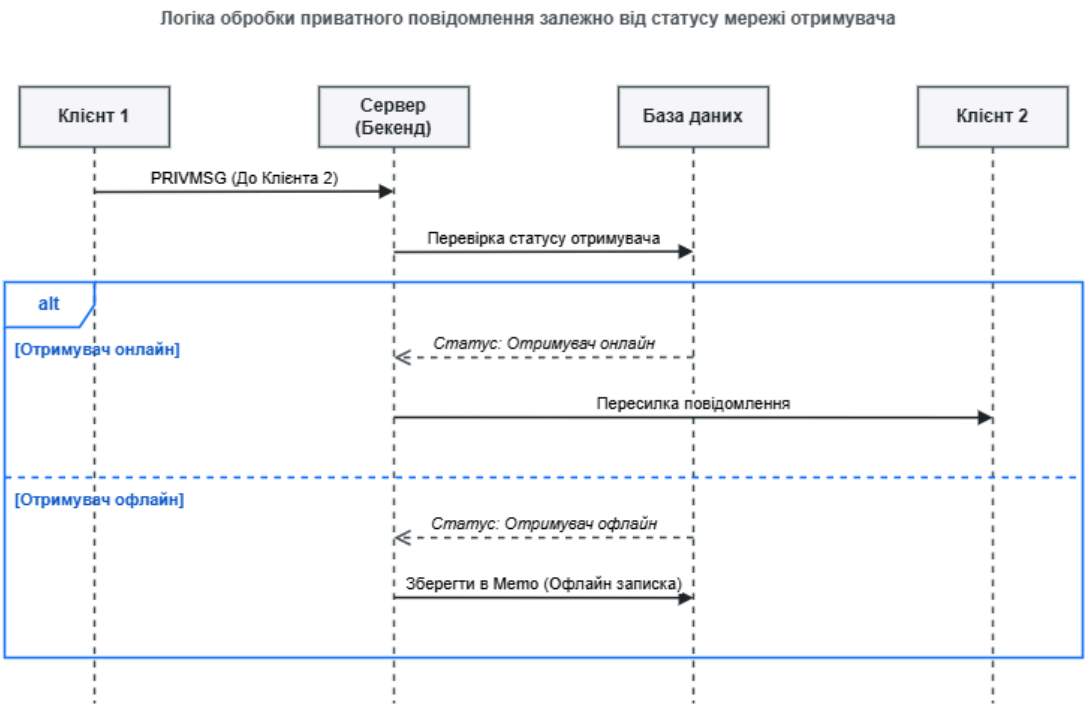


Рис. 2.6 - UML Діаграма послідовностей (Sequence Diagram) маршрутизації повідомлення

Для нівелювання проблеми втрати пакетів при відсутності отримувача в мережі, розроблено алгоритм буферизації. Якщо цільовий сокет недоступний, процес маршрутизації не переривається помилкою, а делегується підсистемі Memo. Текст повідомлення перенаправляється до бази даних і переходить у режим очікування події входу адресата (Login Event), після чого гарантовано доставляється.

Ще одним важливим алгоритмом є моніторинг "завислих" з'єднань (Heartbeat). Для виявлення несанкціонованих обривів мережі (наприклад, перехід смартфона в "авіарежим") планувальник кожні 30 секунд генерує сервісний пакет PING. Якщо впродовж 60 секунд сокет не відповідає пакетом PONG, спрацьовує процедура "Garbage Collection": сокет примусово закривається, статус у базі оновлюється на offline, а залишковий системний потік знищується.

2.4. Дослідження та проєктування механізмів транспортної безпеки та шифрування даних у стані спокою (Data-at-Rest)

Зважаючи на зростання кількості кіберзагроз, процес проєктування включав моделювання векторів атак та розробку комплексної архітектури безпеки. Захист інформації реалізовано у двох незалежних контурах: на рівні мережевої передачі (Data-in-Transit) та на рівні фізичного зберігання (Data-at-Rest).

Для унеможливлення пасивного перехоплення трафіку (Sniffing) та атак типу "Людина посередині" (Man-in-the-Middle) спроєктовано опціональний захист транспортного рівня через протокол SSL/TLS[16]. Під час з'єднання сервер та клієнт генерують симетричні сеансові ключі, створюючи захищений криптографічний тунель поверх TCP-з'єднання. Опціональність цього компонента дозволяє деактивувати його у довірених закритих мережах (наприклад, у VPN-тунелях підприємства) для зниження процесорних витрат(рис 2.7).

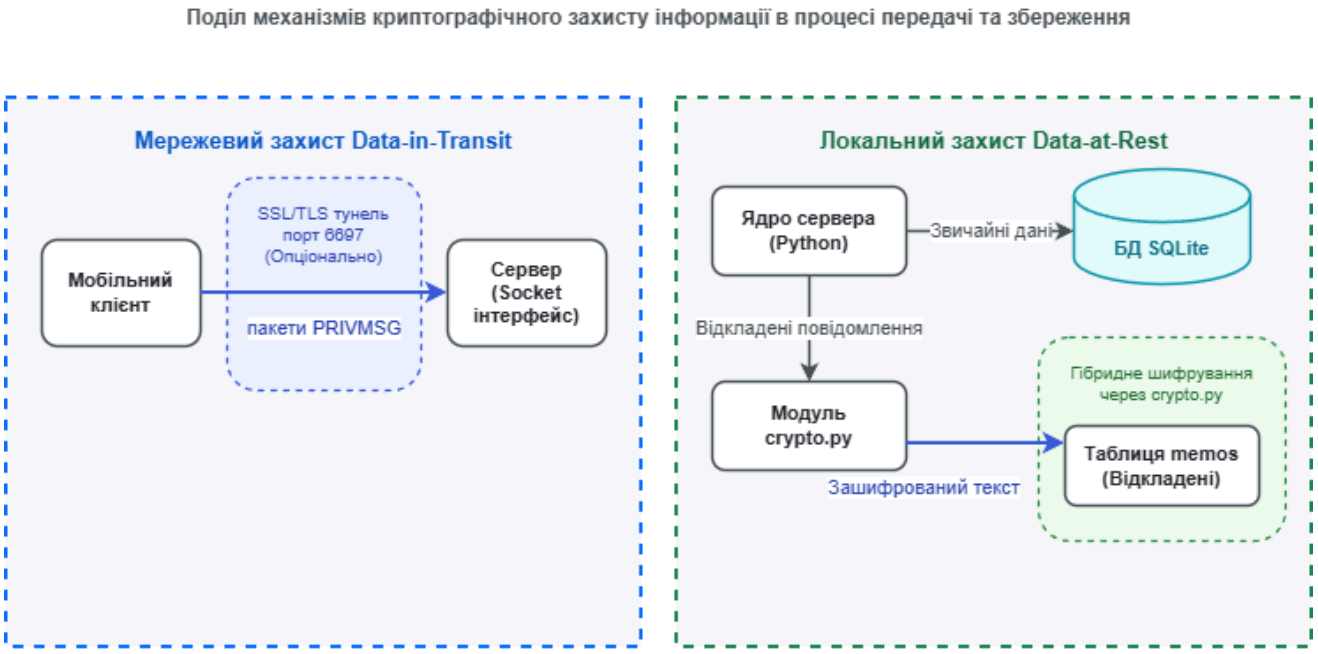


Рис. 2.7 - Схема розподілу транспортної та серверної безпеки

Другий контур фокусується на захисті конфіденційності архівів. Пряме збереження повідомлень у таблицях SQLite створює ризик масового витоку даних у випадку фізичної компрометації сервера[13]. Для нейтралізації цієї загрози було

спроєктовано серверний криптографічний модуль на базі гібридного шифрування (поєднання швидкості симетричної та надійності асиметричної криптографії).

Модель захисту відкладених повідомлень (Memo) передбачає наступний алгоритмічний ланцюг(рис 2.8):

1. Генерація 256-бітного симетричного ключа (AES) та вектора ініціалізації (IV) для кожного окремого повідомлення.
2. Шифрування тексту в режимі Cipher Block Chaining (CBC).
3. Асиметричне шифрування самого AES-ключа за допомогою публічного RSA-4096 сертифіката сервера.
4. Конкатенація RSA-блоку, вектора IV та зашифрованого тексту у єдиний бінарний контейнер, який кодується в Base64 для коректного збереження в БД.

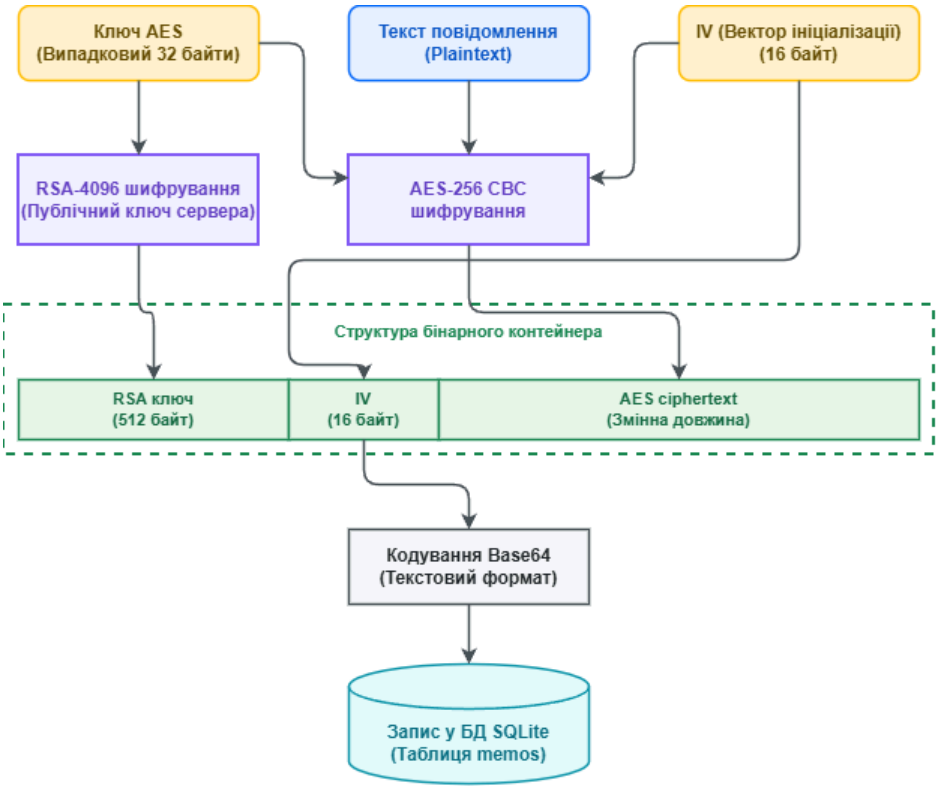


Рис. 2.8 - Концептуальна схема формування зашифрованого контейнера відкладених повідомлень

Цей підхід дозволяє перенести всі важкі обчислення на процесор сервера, залишаючи клієнтські мобільні пристрої ресурсоефективними. Навіть маючи

повний доступ до файлу irc_server.db, зловмисник не зможе розшифрувати історію без наявності приватного RSA-ключа, що зберігається окремо.

2.5. Проєктування користувацького інтерфейсу (UI/UX) мобільного клієнта

Під час проектування архітектури мобільного інтерфейсу одним із пріоритетних критеріїв виступала продуктивність рендерингу на пристроях з обмеженими апаратними ресурсами. Цей фактор став визначальним при виборі технологічного стека: клієнтську частину реалізовано як односторінковий додаток (Single Page Application, SPA) на базі стандартного веб-стека (HTML5/CSS3/JavaScript), що функціонує всередині нативного контейнера WebView за допомогою кросплатформного середовища Capacitor.

Аналіз ефективності рендерингу показав, що інтеграція популярних реактивних фреймворків (зокрема React або Vue) призводить до надлишкового навантаження на підсистему пам'яті через накладні витрати технології Virtual DOM під час динамічного скролінгу великих обсягів текстових даних (історії повідомлень у чатах). З метою оптимізації споживання оперативної пам'яті та забезпечення стабільної частоти кадрів (FPS), архітектуру UI було побудовано на парадигмі прямої маніпуляції DOM-деревом (Direct DOM Manipulation) засобами Vanilla JS, що мінімізує кількість абстракцій між логікою додатка та шаром відображення.

Просторова композиція інтерфейсу оптимізована для мобільної ергономіки. Перемикання між глобальними контекстами додатка («Чати», «Сервер», «Профіль») реалізовано за допомогою нижньої навігаційної панелі (Bottom Navigation Bar), що забезпечує швидкий доступ до основних функцій великим пальцем однієї руки. Для управління каналами та діалогами застосовано систему верхніх експрес-фільтрів (Chips), що дозволяє динамічно сортувати контент без перевантаження екрана зайвими ментальними шарами та прихованими елементами меню(рис. 2.9)[20].

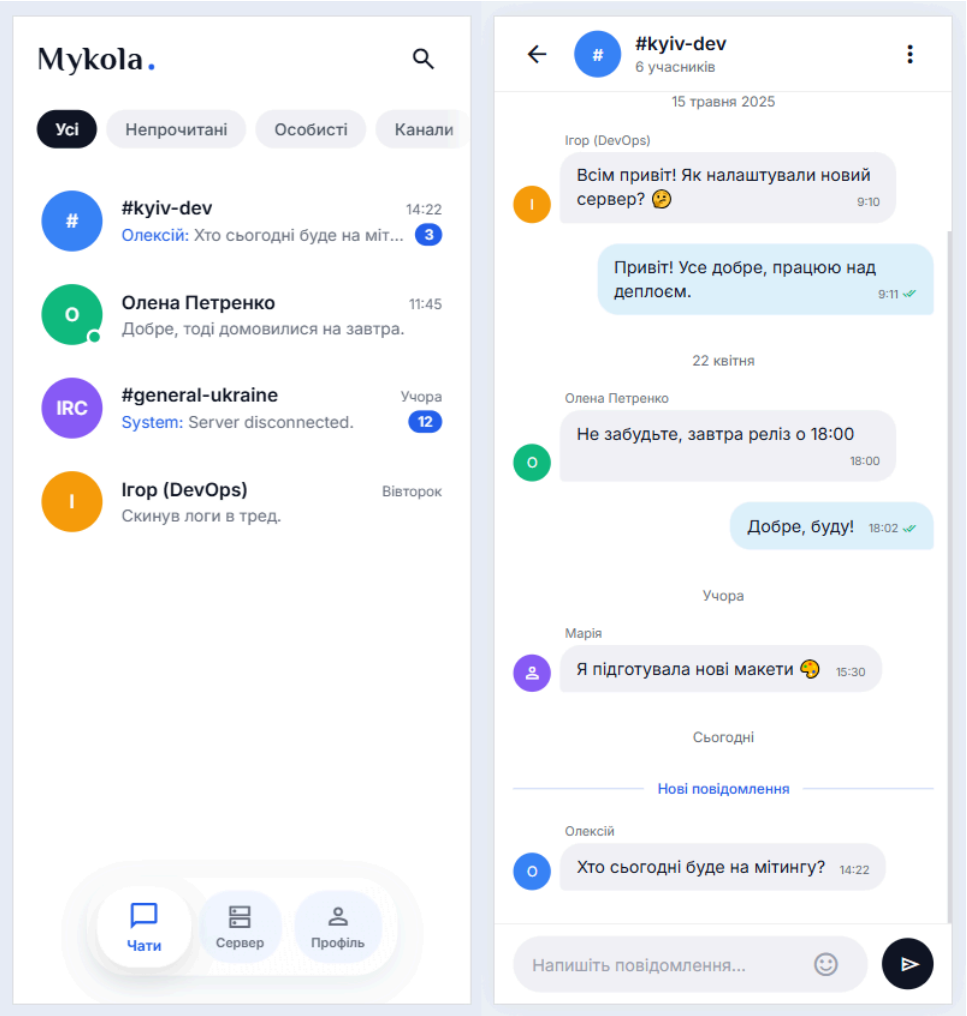


Рис. 2.9 - Макет інтерфейсу користувача

Висновки до розділу 2

У другому розділі проведено системне дослідження та проєктування архітектури мобільного клієнт-серверного месенджера. Сформовано комплекс функціональних та нефункціональних вимог, що деталізують параметри швидкодії, масштабованості та стійкості програми до розривів мережі.

Обґрунтовано доцільність застосування класичної дворівневої архітектури з багатопотоковим сервером на базі Python та тонким мобільним клієнтом. Спроєктовано та нормалізовано структуру реляційної бази даних SQLite (9 сутностей), яка за допомогою ORM технологій забезпечує цілісність даних та управління обліковими записами, правами доступу і журналами історії.

Сформовано математичну модель скінченного автомата для управління життєвим циклом підключень та розроблено алгоритми маршрутизації команд, що включають механізми групової розсилки (Multicast), приватної доставки (Unicast) та відкладеної доставки (Memo) для користувачів поза мережею.

Спроєктовано комплексну модель безпеки системи: для захисту каналу зв'язку застосовано технологію TLS/SSL (Data-in-Transit), а для захисту баз даних на жорсткому диску розроблено гібридний криптографічний алгоритм на базі AES-256 та RSA-4096 (Data-at-Rest). Також обґрунтовано UI/UX рішення на базі прямої маніпуляції DOM-деревом без використання реактивних фреймворків, що гарантує високу продуктивність інтерфейсу на мобільних пристроях базового рівня.

3. ПРАКТИЧНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ СИСТЕМИ ОБМІНУ ПОВІДОМЛЕННЯМИ

3.1. Програмна реалізація серверного ядра та конфігурація мережевої взаємодії (Python)

Для забезпечення зрозумілості та структурованості опису, програмний код серверного ядра месенджера, реалізованого мовою Python, було декомпоновано на логічні блоки відповідно до їхньої функціональності. Кожен наведений фрагмент коду супроводжується поясненням алгоритму його роботи з фокусом на механізми мережевої взаємодії.

Блок 1: Лістинг конфігурації безпеки (Security Config)

```
1 use_encryption = config.get("use_encryption", True)
2 context = None
3 if use_encryption:
4     context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
5     try:
6         context.load_cert_chain(certfile="server.crt", keyfile="server.key")
7     except FileNotFoundError:
8         print("[!] ERROR: 'server.crt' or 'server.key' not found.")
```

Перший блок демонструє процес ініціалізації параметрів транспортної безпеки. Система зчитує конфігураційний параметр use_encryption. За умови його активації, формується об'єкт ssl_context, який завантажує криптографічний сертифікат (server.crt) та приватний ключ (server.key). Такий підхід гарантує готовність сервера до обробки зашифрованих SSL/TLS підключень[2].

Блок 2: Лістинг ініціалізації TCP-сокету (Socket Initialization)

```
1 self.server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
2 self.server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
3 self.server_socket.bind((self.host, self.port))
4 self.server_socket.listen(5)
5 self.server_socket.settimeout(1.0)
```

У другому блоці реалізовано створення головного мережевого вузла. З використанням стандартної бібліотеки socket ініціалізується підтримка протоколу IPv4 (AF_INET) та транспортного протоколу TCP (SOCK_STREAM). Застосування опції SO_REUSEADDR дозволяє системі миттєво звільняти порт після зупинки, уникаючи помилок блокування адреси з боку ОС. Далі сокет прив'язується до визначеної IP-адреси та переходить у стан прослуховування (listen) з визначеною чергою вхідних підключень(рис. 3.1)[1].

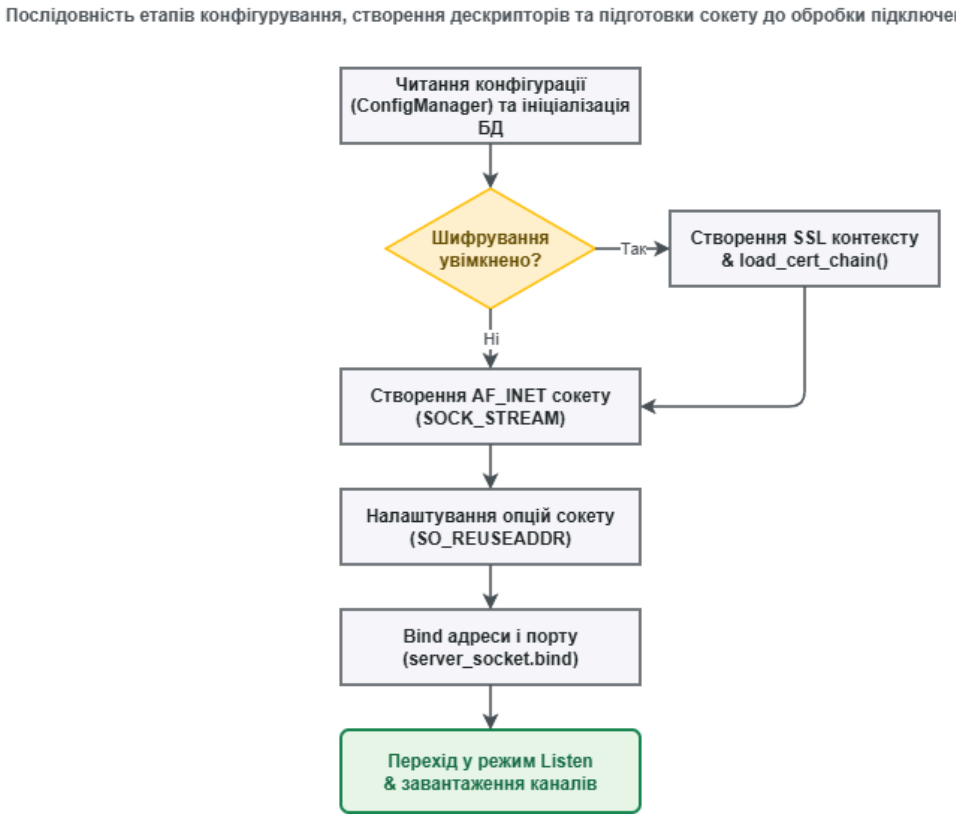


Рис. 3.1 - Алгоритм ініціалізації та запуску TCP-сервера

Блок 3: Лістинг багатопотокового прийому підключень (Multithreaded Accept Loop)

```
1 conn, addr = self.server_socket.accept()
2 if self.ssl_context:
3     try:
4         conn = self.ssl_context.wrap_socket(conn, server_side=True)
5     except ssl.SSLError as e:
6         conn.close()
7         continue
8 client = IRCClient(conn, addr, self)
9 with self.lock:
10     self.clients.add(client)
11 threading.Thread(target=client.handle, daemon=True).start()
```

Третій блок ілюструє основний цикл обробки нових клієнтів. Виклик методу accept() переводить потік у режим очікування. При появі нового з'єднання, за наявності ssl_context, сирий TCP-сокет обгортається у захищений TLS-тунель. Після успішного встановлення з'єднання, створюється екземпляр класу IRCClient, який додається до загального пулу активних сесій з використанням механізму блокування (threading.RLock) для забезпечення потокобезпеки. Обробка клієнтських запитів делегується окремому системному потоку (threading.Thread)[6].

Блок 4: Лістинг обробки вхідного потоку даних (Data Handling)

```
1 def handle(self):
2     while True:
3         if self.closed:
4             break
5         try:
6             raw_data = self.sock.recv(4096)
7             if not raw_data:
8                 break
9             data = raw_data.decode("utf-8", errors="ignore")
10              for line in data.split("\r\n"):
11                  if not line: continue
12                  self.server.handle_command(self, line)
13          except Exception as e:
14              break
```

У четвертому блоці наведено алгоритм зчитування байтового потоку (метод handle класу IRCClient). Цей процес виконується циклічно, зчитуючи пакети до 4096 байт через метод recv. Декодування даних здійснюється у форматі UTF-8. Відповідно до специфікації IRC, отриманий буфер поділяється на окремі команди за допомогою символів розриву рядка \r\n. Кожна розпізнана команда маршрутизується до центрального обробника handle_command(рис. 3.2).

Рис. 3.2 - Модель багатопотокової обробки клієнтських сокетів

Ця багатопотокова архітектура забезпечує високу стійкість сервера: затримки або втрата пакетів на одному клієнтському вузлі не впливають на процеси маршрутизації в інших потоках.

Блок 5: Лістинг управління станом авторизації клієнта (Client Auth State)

```
1 def start_auth_timer(self, seconds=60):
2     self.cancel_auth_timer()
3     self.auth_timer = threading.Timer(seconds, self._auth_timeout)
4     self.auth_timer.daemon = True
5     self.auth_timer.start()
6
7 def _auth_timeout(self):
8     if not self.closed and self.auth_state == AuthState.PENDING:
9         reason = "Registration/Identification timeout"
10         self.send_msg(f"ERROR :Closing Link: {self.nick or '*'}[{self.addr[0]}]
({reason})")
11         self.server.handle_disconnect(self)
```

П'ятий блок демонструє механізм контролю таймаутів під час процесу автентифікації. Для запобігання виснаженню ресурсів сервера неавторизованими сесіями, ініціюється таймер на 60 секунд. Якщо за цей період користувач не проходить валідацію через NickServ (залишаючись у стані AuthState.PENDING), викликається метод _auth_timeout, який примусово розриває з'єднання.

Блок 6: Лістинг безпечного закриття з'єднання (Client Disconnection)

```
1 def close(self):
2     self.closed = True
3     self.cancel_auth_timer()
4     try:
5         self.sock.shutdown(socket.SHUT_RDWR)
6         self.sock.close()
7     except Exception:
8         pass
```

Шостий блок описує процедуру коректного завершення сесії. Метод close оновлює прапорці стану, скасовує активні таймери та ініціює безпечне закриття мережевого каналу за допомогою socket.shutdown(socket.SHUT_RDWR), що гарантує звільнення системних ресурсів.

Блок 7: Лістинг динамічної маршрутизації команд (Command Routing)

```
1 def handle_command(self, client, line):
2     parts = line.split()
3     if not parts: return
4     cmd = parts[0].upper()
5     method_name = f"cmd_{cmd}"
6     if hasattr(self, method_name):
7         getattr(self, method_name)(client, parts, line)
```

Сьомий блок реалізує паттерн динамічної маршрутизації запитів. Система аналізує вхідний рядок, виділяє ключову команду (наприклад, PRIVMSG) та за допомогою рефлексії (getattr) перенаправляє виконання до відповідного методу-обробника.

Блок 8: Лістинг групової ретрансляції в каналах (Channel Broadcasting)

```
1 def broadcast(self, message, exclude_client=None):
2     with self.lock:
3         for client in self.clients:
4             if client != exclude_client:
5                 client.send_msg(message)
```

Восьмий блок визначає логіку групової розсилки (multicast). В межах потокобезпечного блоку with self.lock, сервер ітерує колекцію активних підключень кімнати та ретранслює пакет усім учасникам, за винятком самого ініціатора повідомлення.

3.2. Розробка підсистеми управління даними та інтеграція SQLAlchemy ORM

Для забезпечення постійного зберігання даних (акаунти, історії, налаштування) інтегровано реляційну СУБД SQLite через технологію Object-Relational Mapping (ORM) SQLAlchemy. Це забезпечує високий рівень абстракції при роботі з базою даних.

Блок 1: Лістинг ініціалізації бази даних та управління сесіями (Database Setup)

```
1 engine = create_engine('sqlite:///irc_server.db', connect_args={'check_same_thread': False})
2 SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
3
4 def init_db():
```

```
5    Base.metadata.create_all(bind=engine)
6    with get_session() as db:
7        main_channel = db.query(Channel).filter_by(name="#main").first()
8        if not main_channel:
9            db.add(Channel(name="#main", ip_address="127.0.0.1", creator_id=None))
10           db.commit()
```

У першому блоці показано процес налаштування підключення до irc_server.db. Ключовим параметром є check_same_thread: False, необхідний для коректної роботи багатопотокового серверного ядра. Функція init_db автоматизує створення таблиць на основі ORM-моделей.

Блок 2: Лістинг механізму автентифікації NickServ (Password Hashing)

```
1 def _hash_password(password: str) -> str:
2    salt = b'irc_nickserv_salt'
3    return hashlib.pbkdf2_hmac('sha256', password.encode('utf-8'), salt,
100000).hex()
4
5 def verify_nick(nickname: str, password: str) -> bool:
6    with get_session() as db:
7        reg = db.query(RegisteredNick).filter_by(nickname=nickname).first()
8        if reg and reg.password_hash == _hash_password(password):
9            return True
10       return False
```

Другий блок ілюструє роботу підсистеми криптографічного захисту паролів. Алгоритм PBKDF2 з функцією SHA-256 та сіллю забезпечує стійкість до атак методом перебору (brute-force).

Блок 3: Лістинг логіки обробки логіну клієнта (User Login Logic)

```
1 def handle_db_user_login(username, nick, addr, privilege=None):
2     with get_session() as db:
3         db_user = db.query(User).filter_by(name=username).first()
4         if not db_user:
5             db_user = User(name=username, ip_address=addr[0], online=True,
6             super_privileges=privilege or 0)
7             db.add(db_user)
8         else:
9             db_user.ip_address = addr[0]
10              db_user.online = True
11          db.commit()
12          return db_user.id
```

Третій блок відповідає за оновлення статусу сесії в базі даних. При кожному підключенні система фіксує поточну IP-адресу клієнта та встановлює прапорець online=True, після чого виконує коміт транзакції (db.commit()).

Блок 4: Лістинг збереження відкладених повідомлень Memo (Offline Messages)

```
1 def add_memo(sender_nick: str, recipient_nick: str, message: str, encryptor=None):
2     with get_session() as db:
3         sender_id = get_user_id_by_nickname(sender_nick)
4         recipient_id = get_user_id_by_nickname(recipient_nick)
5         db_message = encryptor.encrypt(message) if encryptor else message
6         memo = Memo(
7             sender_id=sender_id,
8             recipient_id=recipient_id,
9             message=db_message
10        )
```

```
11        db.add(memo)
12        db.commit()
```

Четвертий блок розкриває механізм буферизації повідомлень. Алгоритм визначає ідентифікатори учасників обміну. У разі наявності об'єкта encryptor, текст повідомлення підлягає шифруванню (Data-at-Rest) перед записом до таблиці memos.

Блок 5: Лістинг доставки відкладених повідомлень (Deliver Memos)

```
1 def deliver_memos(self, client):
2     memos = crud.get_unread_memos(client.nick)
3     if not memos: return
4     client.send_msg(f":MemoServ NOTICE {client.nick} :You have {len(memos)}
unread memo(s).")
5     for m in memos:
6         message_text = self.encryptor.decrypt(m.message) if self.encryptor else
m.message
7         sender_nick = crud.get_user_primary_nickname(m.sender_id)
8         client.send_msg(f":MemoServ PRIVMSG {client.nick} :From {sender_nick}:
{message_text}")
9     crud.mark_memos_read(client.nick)
```

П'ятий блок реалізує процес вивантаження збережених даних. Алгоритм отримує всі непрочитані записи з бази, розшифровує їх (за потреби) та відправляє у мережевий сокет клієнта, після чого позначає їх як прочитані (mark_memos_read).

Далі наведено визначення ключових ORM-моделей бази даних:

Блок 6: Лістинг моделі користувача (User Model)

```
1 class User(Base):
2     __tablename__ = 'users'
3     id = Column(Integer, primary_key=True)
4     name = Column(String, unique=True, nullable=False)
5     ip_address = Column(String, nullable=True)
6     super_privileges = Column(Integer, default=0)
7     online = Column(Boolean, default=False)
```

Шостий блок описує базову сутність User. Вона містить унікальне ім'я, останню IP-адресу, глобальні привілеї та мережевий статус користувача.

Блок 7: Лістинг моделі реєстрації нікнеймів (RegisteredNick)

```
1 class RegisteredNick(Base):
2     __tablename__ = 'registered_nicks'
3     id = Column(Integer, primary_key=True)
4     user_id = Column(Integer, ForeignKey('users.id'), nullable=False)
5     nickname = Column(String, unique=True, nullable=False)
6     password_hash = Column(String, nullable=False)
```

Сьомий блок реалізує таблицю підсистеми автентифікації. Тут зберігаються криптографічні хеші паролів з прив'язкою до ідентифікатора користувача.

Блок 8: Лістинг моделі каналів зв'язку (Channel Model)

```
1 class Channel(Base):
2     __tablename__ = 'channels'
3     id = Column(Integer, primary_key=True)
4     name = Column(String, unique=True, nullable=False)
5     creator_id = Column(Integer, ForeignKey('users.id'))
```

Восьмий блок визначає структуру для збереження даних про комунікаційні канали та їхніх творців.

Блок 9: Лістинг моделі відкладених повідомлень (Memo Model)

```
1 class Memo(Base):
2     __tablename__ = 'memos'
3     id = Column(Integer, primary_key=True)
4     sender_id = Column(Integer, ForeignKey('users.id'), nullable=False)
5     recipient_id = Column(Integer, ForeignKey('users.id'), nullable=False)
6     message = Column(String, nullable=False)
7     read = Column(Boolean, default=False)
```

Дев'ятий блок описує структуру для зберігання офлайн-повідомлень, пов'язуючи відправника, отримувача та текст.

Блок 10: Лістинг моделі історії комунікації (History Models)

```
1 class ChannelHistory(Base):
2     __tablename__ = 'channel_history'
3     channel_id = Column(Integer, ForeignKey('channels.id'), nullable=False)
4     sender_id = Column(Integer, ForeignKey('users.id'), nullable=False)
5     message = Column(String, nullable=False)
6     timestamp = Column(DateTime, default=datetime.datetime.now)
```

Десятий блок ілюструє підхід до архівування групових діалогів із фіксацією точного часу відправлення (timestamp).

3.3. Програмна реалізація інтерфейсу користувача та керування глобальним станом

Клієнтський застосунок розроблено як Single Page Application (SPA), інкапсульований у нативний контейнер Capacitor. Архітектура відмовляється від використання важких фреймворків (на зразок React) на користь ванільного JavaScript для мінімізації споживання пам'яті[11].

Блок 1: Лістинг базової структури документа (HTML Foundation)

```
1 <meta name="viewport" content="width=device-width,
2     initial-scale=1.0, maximum-scale=1.0, user-scalable=no, viewport-fit=cover">
3 <meta name="theme-color" id="themeColorMeta" content="#ffffff">
4 <!-- ... -->
5 <script type="module" src="js/fundamentals-global-state.js"></script>
6 <script type="module" src="js/ui-renderers.js"></script>
```

Перший блок містить базові налаштування WebView. Мета-тег viewport адаптує відображення до мобільних екранів, а підключення JavaScript у вигляді модулів (type="module") забезпечує інкапсуляцію логіки.

Блок 2: Лістинг адаптивної стилізації (CSS Theming & Layout)

```
1 :root {
2     --bg-main: #ffffff;
3     --text-primary: #111827;
4     --accent-color: #2563eb;
5     --transition-fast: 0.2s ease;
6 }
7 .app-container.chat-open .sidebar {
8     display: none;
9 }
10 .app-container.chat-open .chat-area {
```

```
11     display: flex;
12     width: 100%;
13 }
```

У другому блоці наведено застосування CSS-змінних для реалізації теми оформлення[14]. Клас .chat-open демонструє підхід до керування видимістю елементів (бічного меню та чату) без перезавантаження сторінки.

Блок 3: Лістинг управління глобальним станом (Global State)

```
1 let TcpSocket = null;
2 let socketId = null;
3 let connectionMode = "none";
4 let chats = {}; // { id: { name, type, messages: [], unreadCount } }
5 let activeChatId = null;
6
7 if (window.Capacitor && window.Capacitor.isNativePlatform()) {
8     TcpSocket = Capacitor.Plugins.TcpSocket;
9 }
```

Третій блок визначає структуру глобального стану програми. Об'єкт chats виконує роль "Single Source of Truth", акумулюючи історію всіх активних сесій[17]. Ініціалізація TcpSocket відбувається виключно у середовищі Capacitor.

Блок 4: Лістинг логіки сокетного підключення (Socket Connection Logic)

```
1 const connectToIRC = async (ip, port, serverName, encryption, username,
password) => {
2     try {
3         connectionMode = "tcp";
4         const result = await TcpSocket.connect({
5             address: ip,
```

```
6           port: port,
7           ssl: encryption
8       });
9       socketId = result.client;
10        sendRaw(`NICK ${username}`);
11        sendRaw(`USER ${username} 0 * :Mykola Client`);
12    } catch (e) {
13        connectionMode = "none";
14        console.error("Connection failed", e);
15    }
16 };
```

Четвертий блок містить асинхронну функцію ініціалізації з'єднання. Вона викликає плагін TcpSocket, передаючи конфігураційні параметри. Після з'єднання клієнт автоматично відправляє сервісні команди NICK та USER.

Блок 5: Лістинг прямого рендерингу елементів (DOM Manipulation)

```
1 const addMessageToChat = (chatId, messageObj) => {
2    const chat = chats[chatId];
3    if (!chat) return;
4    chat.messages.push(messageObj);
5    if (activeChatId === chatId) {
6        const msgEl = document.createElement('div');
7        msgEl.className = `message-wrapper ${messageObj.type}`;
8        msgEl.innerHTML = `${escapeHTML(messageObj.text)}`;
9        document.getElementById('chatMessages').appendChild(msgEl);
10    }
11 };
```

П'ятий блок ілюструє метод оптимізованого рендерингу (Direct DOM Manipulation). Створюється новий вузол документа, застосовується санітизація тексту (escapeHTML) і вузол додається безпосередньо у контейнер chatMessages, що значно швидше за обрахунки Virtual DOM[7].

Блок 6: Лістинг обробки дій користувача (Event Handling)

```
1 const sendMessage = (inputEl) => {
2     const text = inputEl.value.trim();
3     if (!text || !activeChatId) return;
4     inputEl.value = '';
5     sendRaw(`PRIVMSG ${activeChatId} :${text}`);
6     addMessageToChat(activeChatId, {
7         sender: myNick || "Я",
8         text: text,
9         type: "outgoing"
10    });
11 };
```

Шостий блок описує алгоритм відправлення даних. Зчитаний текст перетворюється у команду PRIVMSG, надсилається у сокет, а потім локально відображається у UI, що забезпечує миттєвий відгук інтерфейсу.

3.4. Впровадження механізмів локального збереження даних та офлайн-роботи

Для забезпечення відмовостійкості застосунку у середовищах із нестабільним підключенням до Інтернету було реалізовано підсистему локального кешування на базі Capacitor Preferences.

Блок 1: Лістинг локального збереження метаданих користувачів (User Data Persistence)

```
1 const saveUserDataToStorage = async () => {
2     try {
3         await Preferences.set({ key: 'irc_user_colors', value:
JSON.stringify(userColors) });
4         await Preferences.set({ key: 'irc_user_last_seen', value:
JSON.stringify(userLastSeen) });
5         await Preferences.set({ key: 'irc_user_ips', value: JSON.stringify(userIps) });
6     } catch (e) { console.error("Error saving user data", e); }
7 };
```

Перший блок показує процес серіалізації конфігураційних об'єктів (кольори, мітки часу) у формат JSON із подальшим їх записом до локального сховища пристрою. Це зменшує кількість запитів до сервера під час ініціалізації.

Блок 2: Лістинг кешування та відновлення історії повідомлень (Chat Caching & Recovery)

```
1 const saveChatsToStorage = async () => {
2     try {
3         await Preferences.set({ key: 'irc_chats_data', value: JSON.stringify(chats) });
4     } catch (e) { console.error("Error saving chats", e); }
5 };
6
7 const loadChatsFromStorage = async () => {
8     try {
9         const saved = await Preferences.get({ key: 'irc_chats_data' });
10         if (saved.value) {
11             chats = JSON.parse(saved.value);
12         } else {
```

```
13          chats["IRC"] = { id: "IRC", name: "System Messages", type: "system",
messages:
14          [], unreadCount: 0, lastTimestamp: Date.now() };
15      }
16  } catch (e) { console.error("Error loading chats", e); }
17 };
```

Другий блок ілюструє алгоритми збереження (save) та відновлення (load) історії діалогів. Під час "холодного старту" додаток зчитує масив chats з пам'яті, що дозволяє користувачу переглядати повідомлення до встановлення мережевого підключення(рис. 3.3).

Рис. 3.3 - Схема архітектури локального збереження та відновлення даних клієнта

Блок 3: Лістинг контролю фізичного обсягу сховища (Storage Space Calculation)

```
1 const updateStorageInfo = async () => {
2     try {
3         const chatsData = await Preferences.get({ key: 'irc_chats_data' });
4         let totalBytes = 0;
5         if (chatsData.value) totalBytes += chatsData.value.length;
6         let displaySize = "";
7         if (totalBytes > 1024 * 1024) {
8             displaySize = (totalBytes / (1024 * 1024)).toFixed(1) + " MB";
```

```
9       } else if (totalBytes > 1024) {
10          displaySize = (totalBytes / 1024).toFixed(1) + " KB";
11      } else {
12          displaySize = totalBytes + " B";
13      }
14      if (usedSpaceEl) usedSpaceEl.textContent = displaySize;
15  } catch (e) { console.error("Error updating storage info", e); }
16 };
```

Третій блок містить логіку моніторингу використаної пам'яті. Розмір збережених JSON-об'єктів розраховується в байтах і конвертується у людиночитабельний формат (КБ/МБ) для відображення в налаштуваннях програми.

Блок 4: Лістинг алгоритму фонового перепідключення (Auto-Reconnect Loop)

```
1 setInterval(async () => {
2   if (connectionMode === 'none') {
3     const savedData = await Preferences.get({ key: 'irc_connection_data' });
4     if (savedData.value) {
5       try {
6         const data = JSON.parse(savedData.value);
7         if (data.ip) {
8           connectToIRC(data.ip, data.port || 6667, data.serverName || "none",
9           data.encryption || false, data.username || "User", data.password || "",
10            null, true);
11        }
12      } catch (e) { console.error("Auto-reconnect parse error:", e); }
13    }
14  }
```

```
15 }, 60000);
```

Четвертий блок демонструє фоновий процес (Watchdog timer), який кожні 60 секунд аналізує статус з'єднання. У разі виявлення втрати зв'язку (none), система автоматично витягує збережені параметри та ініціює приховане перепідключення до сервера(рис. 3.4).

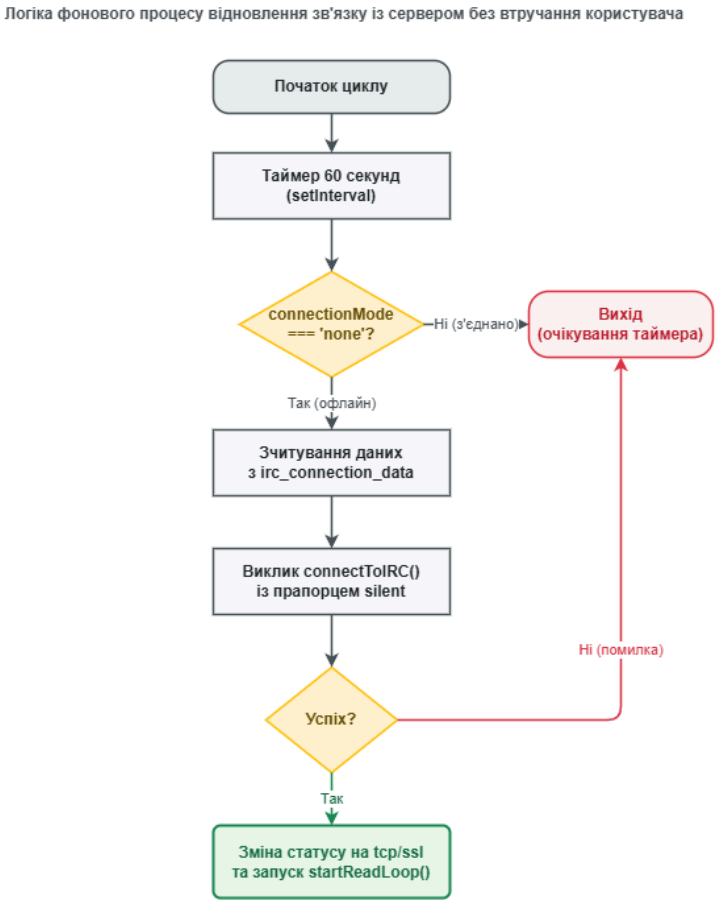


Рис. 3.4 - Блок-схема алгоритму автоматичного відновлення з'єднання (Auto-Reconnect)

3.5. Комплексне тестування програмного продукту та оцінка його ефективності

На завершальному етапі було проведено комплексне тестування розробленої системи (метод "чорної скриньки") з метою верифікації її працездатності та відповідності заданим вимогам. Для перевірки стабільності взаємодії між серверним ядром та мобільним клієнтом у середовищі ОС Android було

розроблено та виконано кілька ключових тестових сценаріїв. Результати тестування візуалізовано за допомогою скриншотів робочих екранів програми та логів сервера.

Сценарій 1: Ініціалізація серверного ядра та моніторинг підключень

Перший сценарій спрямований на перевірку коректності запуску бекенду. При старті скрипта сервера система успішно ініціалізує базу даних SQLite, генерує SSL-контекст та переходить у режим очікування на визначеному порту. На скриншоті логів (рис. 3.5) зафіксовано подію успішного підключення нового клієнта, де сервер реєструє IP-адресу пристрою та виділяє для нього ізольований системний потік обробки.

```
[*] DB: Super-admin '1mba' already present — privileges verified.
[*] Console manager active. Type 'status' for debug, 'exit' to stop.
Server Admin > [*] Secure IRC Server started on 0.0.0.0:6697 (TLS/SSL)
```

Рис. 3.5 - Скриншот вікна термінала з логами запуску сервера

Сценарій 2: Авторизація клієнта та робота підсистеми NickServ

Другий сценарій тестує процес реєстрації та автентифікації на стороні мобільного пристрою. Після запуску APK-файлу на Android-смартфоні, користувачу відображається стартовий екран вводу параметрів[8]. Після введення облікових даних клієнт формує TCP-з'єднання. Як видно з інтерфейсу (рис. 3.6), система успішно валідує криптографічний хеш у базі даних та пропускає користувача до головного меню, змінюючи статус з'єднання на активний.

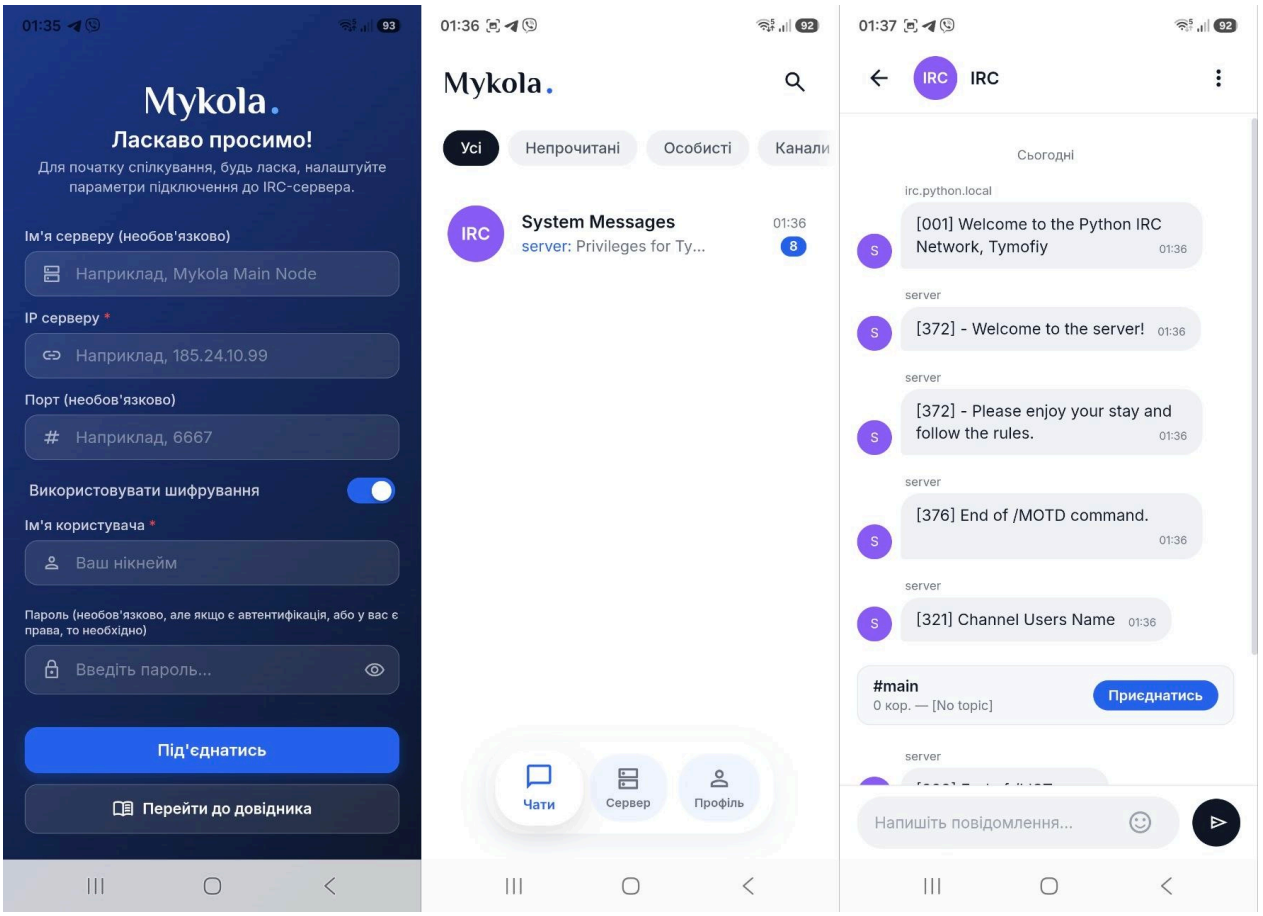


Рис. 3.6 - Скриншот екрана авторизації мобільного додатка та підключення до серверу

Сценарій 3: Маршрутизація повідомлень та офлайн-доставка (MemServ)

Третій сценарій перевіряє основну бізнес-логіку месенджера. Під час тестування перевірено обмін пакетами між двома клієнтами в єдиному каналі #main (Multicast). Окремо було проведено стрес-тест функції офлайн-доставки: одне з повідомлень відправлено користувачу зі статусом offline. На рис. 3.7 продемонстровано екран отримувача після його повторної авторизації в системі — повідомлення успішно розшифровано і відображено з позначкою системного сервісу MemoServ.

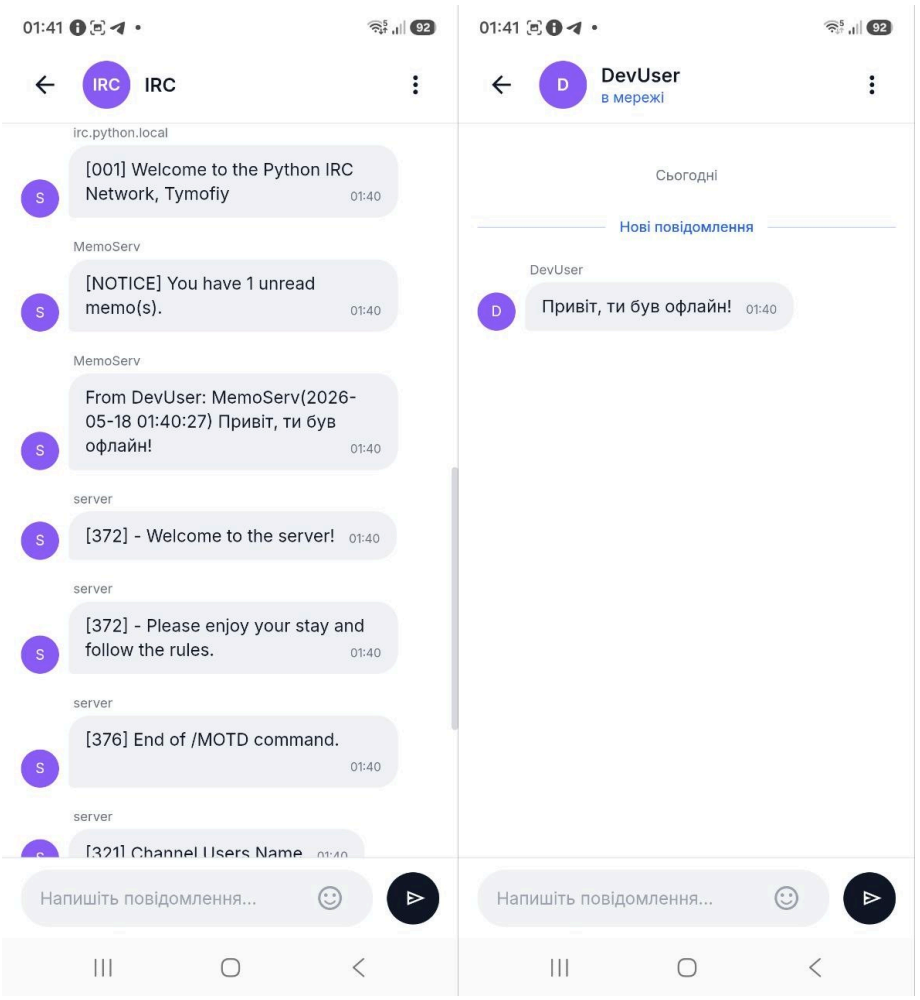


Рис. 3.7 - Скриншот інтерфейсу чату з отриманими відкладеними повідомленнями

Сценарій 4: Тестування відмовостійкості та автоматичного перепідключення

Четвертий сценарій імітує несанкціоновану втрату мережевого сигналу. Серверний моніторинг (Heartbeat) коректно виявляє тайм-аут сокета і закриває зависле з'єднання. Після відновлення доступу до мережі на смартфоні, фоновий процес Capacitor (Watchdog) автоматично зчитує конфігурацію з Preferences та відновлює сесію без втручання користувача (рис. 3.8).

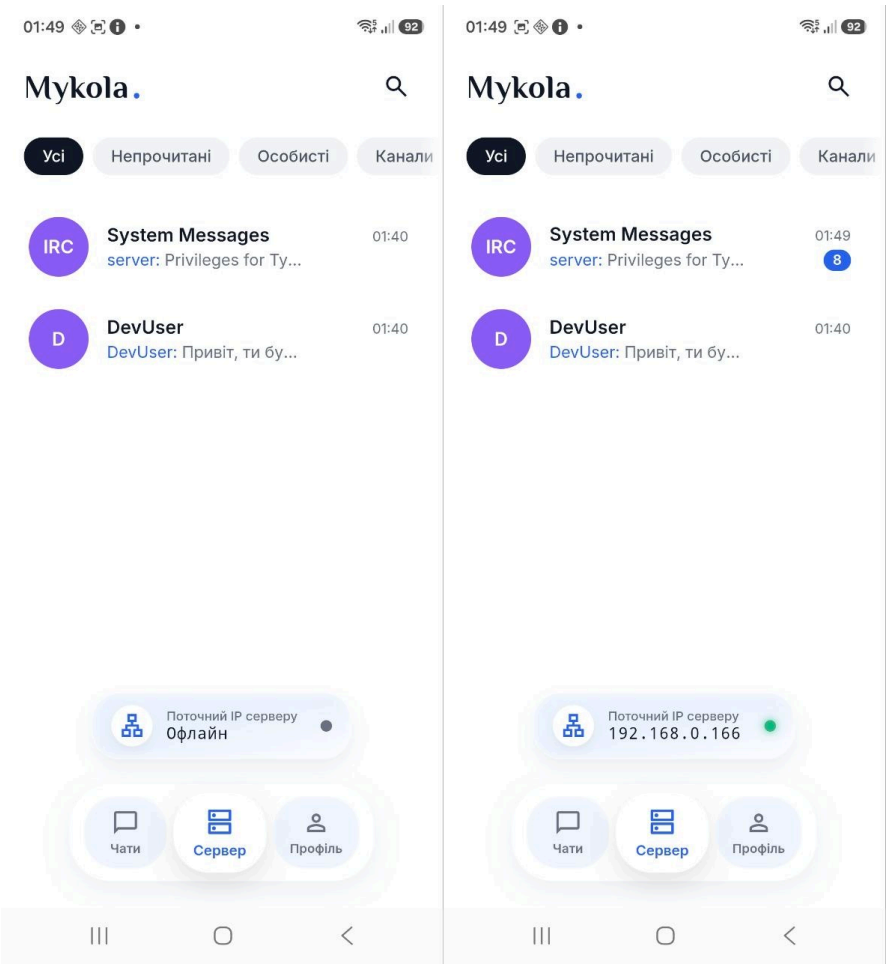


Рис. 3.8 - Скриншот відновлення з'єднання після втрати мережі

Висновки до розділу 3

У третьому розділі описано процес практичної реалізації компонентів спроєктованої системи обміну повідомленнями. Програмний код серверного ядра, розробленого мовою Python, продемонстрував ефективність використання багатопотокової обробки сокетів, динамічної маршрутизації та обробки сесій користувачів.

Інтеграція реляційної бази даних SQLite за допомогою ORM SQLAlchemy дозволила реалізувати безпечну підсистему автентифікації користувачів, рольову модель керування каналами та надійне збереження історії комунікацій і відкладених повідомлень.

Програмна реалізація клієнтської частини у вигляді Single Page Application у середовищі Capacitor підтвердила обґрунтованість відмови від важких реактивних

фреймворків. Використання методів прямої маніпуляції DOM-деревом у поєднанні з оптимізацією керування глобальним станом забезпечило високу продуктивність застосунку на мобільних пристроях.

Впровадження алгоритмів фонового перепідключення та локального кешування даних значно підвищило стійкість системи до умов нестабільного мережевого підключення. Результати тестування довели відповідність розробленої системи заявленим функціональним та нефункціональним вимогам.

ВИСНОВКИ

Підводячи підсумки проведеного кваліфікаційного дослідження, варто зазначити, що у результаті виконання роботи було проведено комплексне дослідження методів проєктування систем обміну повідомленнями в реальному часі. Детальний аналіз предметної області та існуючих комерційних рішень, таких як Telegram і Signal, дозволив виявити низку критичних обмежень, зокрема щодо централізації даних та обов'язкової прив'язки до особистих ідентифікаторів користувача. Усвідомлення цих недоліків стало підґрунтям для розробки власного децентралізованого мобільного застосунку. Було обґрунтовано доцільність використання текстового протоколу IRC поверх прямих TCP-сокетів як більш легкої та ефективної альтернативи традиційним HTTP-запитам. На основі цього спроєктовано надійну клієнт-серверну архітектуру, де серверне ядро реалізоване мовою Python з використанням багатопотоковості, а клієнтська частина — як кросплатформний застосунок на базі Capacitor. Принципова відмова від важких реактивних фреймворків на користь прямої маніпуляції DOM-деревом дозволила досягти високої швидкодії навіть на мобільних пристроях базового рівня.

Важливим етапом роботи стала реалізація надійної підсистеми управління даними та маршрутизації. Інтеграція реляційної бази даних SQLite за допомогою технології об'єктно-реляційного відображення (SQLAlchemy ORM) забезпечила цілісність інформації про користувачів, безпечну автентифікацію, підтримку рольової моделі керування каналами та надійне збереження історії комунікацій. Водночас було успішно вирішено одну з фундаментальних проблем класичного протоколу IRC — відсутність вбудованого механізму збереження офлайн-повідомлень. Для цього розроблено підсистему MemServ, яка автоматично буферизує вхідні пакети для користувачів, що перебувають поза мережею, та гарантовано доставляє їх одразу після успішної авторизації в системі.

Особлива увага під час розробки приділялася питанням інформаційної безпеки та відмовостійкості. У систему було впроваджено комплексну багаторівневу модель захисту: для убезпечення даних під час передачі

(Data-in-Transit) застосовано опціональне транспортне шифрування TLS/SSL, а для захисту інформації у стані спокою на диску сервера (Data-at-Rest) реалізовано гібридний криптографічний алгоритм на основі симетричного шифру AES-256 та асиметричного RSA-4096. На завершальному етапі було проведено комплексне тестування розробленої системи методом "чорної скриньки". Результати випробувань підтвердили безперебійну роботу алгоритмів фонового перепідключення, коректність локального кешування даних та високий загальний рівень відмовостійкості застосунку в умовах нестабільного мобільного зв'язку.

Практична цінність виконаної роботи полягає у створенні повністю автономного та готового до впровадження програмного продукту. Розроблений месенджер може використовуватися як незалежний корпоративний або приватний засіб комунікації, який дозволяє організаціям та окремим користувачам зберігати абсолютний контроль над власною інфраструктурою зв'язку, мінімізуючи залежність від сторонніх хмарних сервісів.

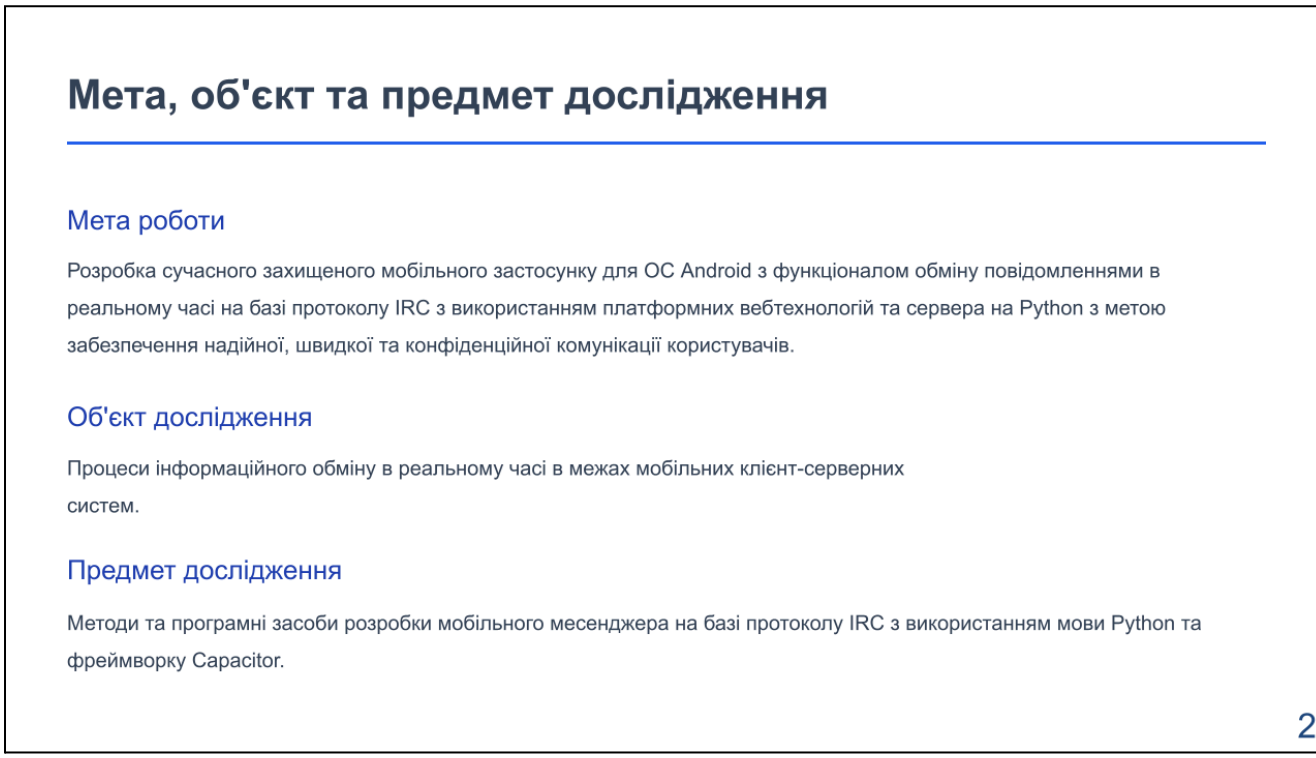
Перспективами подальшого розвитку проєкту є розширення функціоналу системи, зокрема інтеграція механізмів передачі мультимедійних файлів (зображень та голосових повідомлень), розширення можливостей адміністративної панелі для моніторингу навантаження на сервер у реальному часі, а також впровадження напів-автоматичних клієнтських скриптів або інтелектуальних локальних чат-ботів для модерації каналів. Використання сучасних API (наприклад, WebGPU для виконання легких моделей обробки тексту безпосередньо на пристрої) дозволить автоматично фільтрувати спам або категоризувати згадки без надсилання даних на сторонні сервери.

ПЕРЕЛІК ПОСИЛАНЬ

1. Python Software Foundation. The Python Standard Library: socket — Low-level networking interface [Електронний ресурс]. – Режим доступу: https://docs.python.org/3/library/socket.html (дата звернення: 18.05.2026).
2. Python Software Foundation. The Python Standard Library: ssl — TLS/SSL wrapper for socket objects [Електронний ресурс]. – Режим доступу: https://docs.python.org/3/library/ssl.html (дата звернення: 18.05.2026).
3. IRCv3 Working Group. IRCv3 Specifications and Architecture [Електронний ресурс]. – Режим доступу: https://ircv3.net/irc/ (дата звернення: 18.05.2026).
4. Ionic Framework. Capacitor Documentation: Cross-platform Native Runtime [Електронний ресурс]. – Режим доступу: https://capacitorjs.com/docs (дата звернення: 18.05.2026).
5. SQLAlchemy. SQLAlchemy 2.0 Documentation: The Database Toolkit for Python [Електронний ресурс]. – Режим доступу: https://docs.sqlalchemy.org/en/20/ (дата звернення: 18.05.2026).
6. Python Software Foundation. The Python Standard Library: threading — Thread-based parallelism [Електронний ресурс]. – Режим доступу: https://docs.python.org/3/library/threading.html (дата звернення: 18.05.2026).
7. Mozilla Developer Network. Document Object Model (DOM) [Електронний ресурс]. – Режим доступу: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model (дата звернення: 18.05.2026).
8. Android Developers. Download Android Studio & App Tools [Електронний ресурс]. – Режим доступу: https://developer.android.com/studio (дата звернення: 18.05.2026).
9. A formal security analysis of the signal messaging protocol / K. Cohn-Gordon et al. *2017 IEEE european symposium on security and privacy (euros&p)*, Paris, 26–28 April 2017. 2017. URL: https://doi.org/10.1109/eurosp.2017.27 (date of access: 18.05.2026).

10. A formal security analysis of the signal messaging protocol / K. Cohn-Gordon et al. *Journal of cryptology*. 2020. Vol. 33, no. 4. P. 1914–1983. URL: https://doi.org/10.1007/s00145-020-09360-1 (date of access: 18.05.2026).
11. Eloquent javascript: a modern introduction to programming. 3rd ed. San Francisco, California, United States of America : No Starch Press, 2018. 472 p.
12. High performance mobile web: best practices for optimizing mobile web apps. O'Reilly Media, Incorporated, 2016.
13. Katz J., Lindell Y. Introduction. *Introduction to modern cryptography*. 2020. P. 1–22. URL: https://doi.org/10.1201/9781351133036-2 (date of access: 18.05.2026).
14. Meyer E. A., Weyl E. CSS : the definitive guide: web layout and presentation. O'Reilly Media, Incorporated, 2023.
15. Michael K., Chou E., Whaley M. Mastering Python Networking: Your one-stop solution to using Python for network automation, programmability, and DevOps, 3rd Edition. Packt Publishing, 2020. 576 p.
16. National Institute of Standards and Tech. Guidelines for the selection, configuration, and use of transport layer security implementations: draft nist sp 800-52 R2. Independently Published, 2018.
17. Osmani A. Learning javascript design patterns: a javascript and react developer's guide. O'Reilly Media, Incorporated, 2023. 286 p.
18. Touch design for mobile interfaces. Smashing Media AG, 2022. 346 p.
19. Xu A. System design interview - an insider's guide. Independently Published, 2020. 276 p.
20. Yadav M. J., Banubakode D. A. Android design patterns. *Multidisciplinary journal of research in engineering and technology*. 2020. Vol. 7, no. 3. P. 170–175. URL: https://doi.org/10.65521/mjret.v7i3.1236 (date of access: 18.05.2026).

ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ (Презентація)



Задачі кваліфікаційної роботи

1. Проаналізувати сучасні технології, мережеві протоколи та архітектурні підходи до побудови систем обміну повідомленнями.
2. Дослідити можливості та переваги текстового протоколу IRC у порівнянні з аналогами (WebSocket, HTTP Polling).
3. Сформувати функціональні та нефункціональні вимоги до мобільного застосунку.
4. Розробити клієнт-серверну архітектуру системи, спроєктувати структуру БД та алгоритми управління сесіями і маршрутизації.
5. Дослідити та спроєктувати механізми багаторівневої інформаційної безпеки (TLS/SSL, AES-256 + RSA-4096).
6. Програмно реалізувати серверну частину (Python) та клієнтський застосунок (Capacitor/Vanilla JS).
7. Провести комплексне тестування розробленого рішення для оцінки продуктивності та безпеки.

3

Аналіз аналогів

Функція / Характеристика	Telegram	Signal	**Розроблений IRC-месенджер**
Повністю власний сервер (On-premise)	Ні (Хмарна архітектура)	Ні (Залежність від інфраструктури)	**Так (Повний контроль над даними)**
Анонімність (без номера телефону)	Ні	Ні	**Так**
Офлайн-доставка повідомлень	Так	Так	**Так (Через підсистему MemServ)**
Оверхед шифрування у локальній мережі	Високий (завжди увімкнено)	Критично високий	**Опціональне шифрування**
Транспортний протокол	MTProto	Signal Protocol	**TCP / IRC (Низькі накладні витрати)**

4

Вимоги до програмного забезпечення

Функціональні вимоги

- Встановлення стійкого TCP-з'єднання та вибір нікнейма.
- Криптографічна авторизація через базу даних (підсистема NickServ).
- Створення каналів та рольова модель управління привілеями.
- Маршрутизація даних (Multicast для груп, Unicast для приватних діалогів).
- Буферизація повідомлень для користувачів поза мережею (Memo).

Нефункціональні вимоги

- **Мінімізація затримок:** миттєва обробка команд.
- **Відмовостійкість:** стійкість до короткочасних розривів з'єднання (Auto-Reconnect).
- **Безпека:** опціональне транспортне шифрування (TLS/SSL) та шифрування бази даних.
- **Оптимізація пам'яті:** плавна робота на пристроях базового рівня (Direct DOM Manipulation).

5

Програмні засоби реалізації

Python 3
Багатопотокове серверне ядро, обробка TCP-сокетів

SQLite & SQLAlchemy
СУБД та ORM для збереження профілів і історії

Capacitor
Кросплатформне середовище для ОС Android

Vanilla JS
SPA-інтерфейс, пряма маніпуляція DOM (Direct DOM)

Cryptography
Гібридне шифрування AES-256 + RSA-4096

TLS/SSL
Опціональний криптографічний захист транспортного каналу

6

Логічна архітектура клієнт-серверної взаємодії

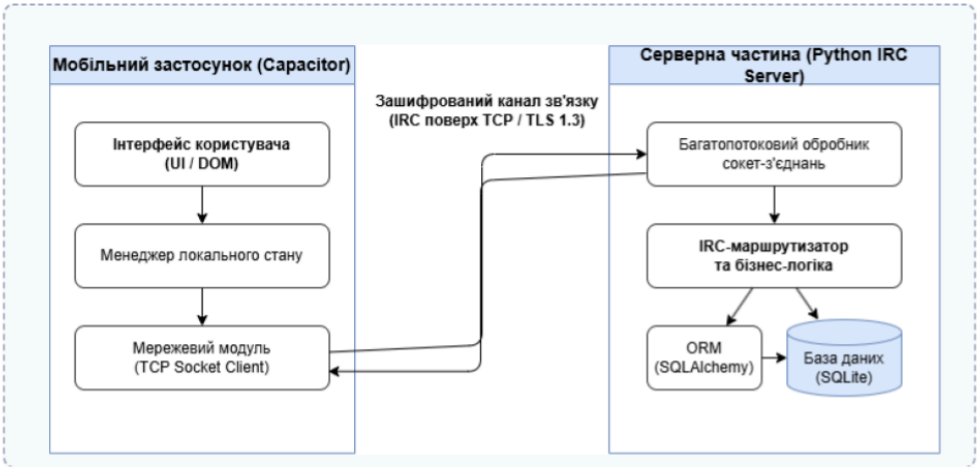


Рис. 1 - Логічна архітектура клієнт-серверної взаємодії застосунку

7

Схема реляційної бази даних месенджера

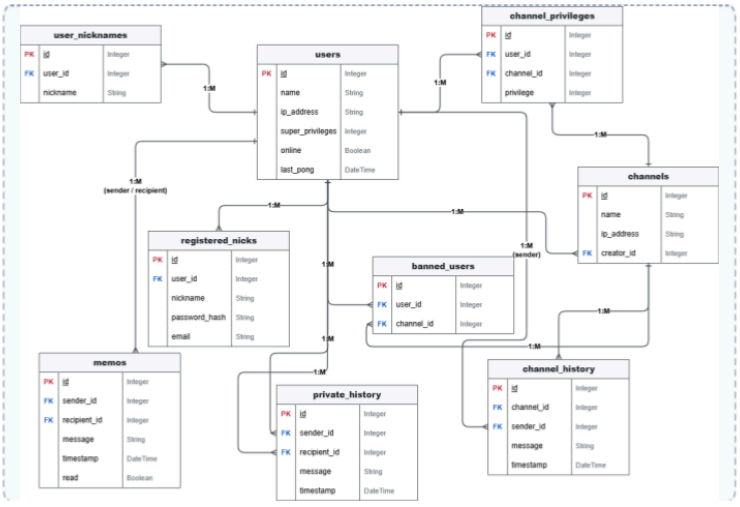


Рис. 2 - ER-діаграма (Entity-Relationship) реляційної бази даних застосунку

8

Механізми інформаційної безпеки

Рис. 3 - Контур мережевого захисту (Data-in-Transit)

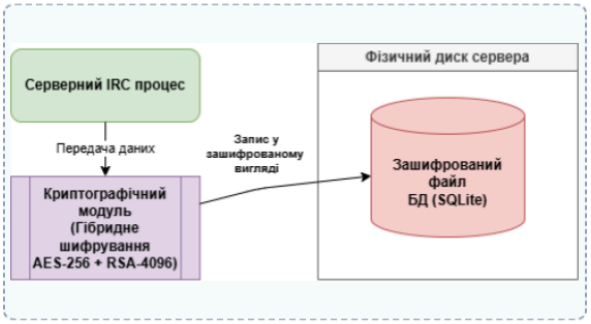


Рис. 4 - Контур локального захисту сервера (Data-at-Rest)

Екранні форми мобільного застосунку

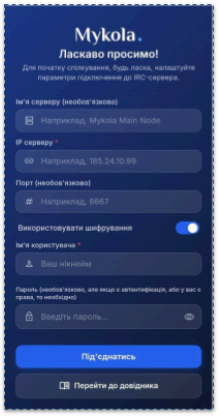


Рис. 5 - Екран конфігурації та логіну

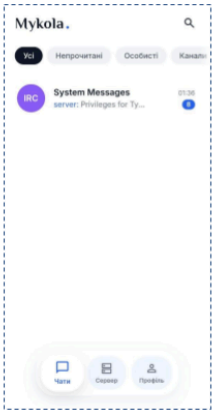


Рис. 6 - Головний екран (Список чатів)

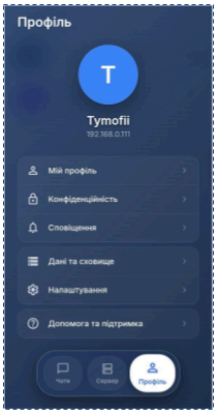


Рис. 7 - Екран профілю

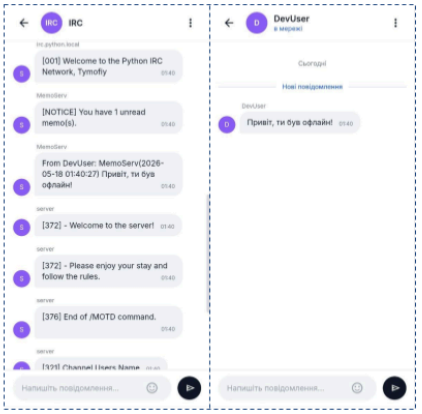


Рис. 8 - Вікно діалогу та офлайн-доставка

Висновки

- ✓ Доведено перевагу протоколу IRC над WebSockets для створення швидких та ресурсоефективних систем обміну повідомленнями.
- ✓ Спроєктовано та реалізовано автономну клієнт-серверну архітектуру, що забезпечує повний контроль над даними (на відміну від Telegram/Signal).
- ✓ Успішно реалізовано підсистему MemServ з інтеграцією SQLite, що вирішує проблему класичного IRC — доставку повідомлень користувачам в офлайн.
- ✓ Впроваджено багаторівневу модель безпеки: опціональне TLS/SSL шифрування каналу та гібридне шифрування бази даних (AES-256 + RSA-4096).
- ✓ Створено мобільний застосунок за допомогою фреймворку Capacitor та Vanilla JS, що забезпечує стабільну роботу та мінімальне споживання пам'яті смартфона завдяки прямій маніпуляції DOM.
- ✓ Комплексне тестування підтвердило стійкість системи: алгоритми Auto-Reconnect автоматично відновлюють сесію при втраті зв'язку без втрати даних.

11

Апробація результатів дослідження

Основні положення і результати кваліфікаційної роботи доповідались та були опубліковані у збірниках тез наступних науково-практичних конференцій:

1. Явдюк Т.М. - «ВИКОРИСТАННЯ ШТУЧНОГО ІНТЕЛЕКТУ ДЛЯ ОПТИМІЗАЦІЇ РУХУ ТРАНСПОРТУ В "РОЗУМНОМУ МІСТІ"». Тези доповіді на III Всеукраїнській науково-технічній конференції «Технологічні горизонти: дослідження та застосування інформаційних технологій для технологічного прогресу України і світу».
2. Явдюк Т.М. - «ВИКОРИСТАННЯ ПРОТОКОЛУ IRC ЯК СТІЙКОГО РЕЗЕРВНОГО КАНАЛУ ЗВ'ЯЗКУ ДЛЯ КРИТИЧНОЇ ІТ-ІНФРАСТРУКТУРИ». Тези доповіді на VII Всеукраїнській науково-технічній конференції «Сучасний стан та перспективи розвитку IoT».

12

Дякую за увагу!