Вінницький національний технічний університет

Факультет інформаційних технологій та комп'ютерної інженерії

Кафедра програмного забезпечення

Бакалаврська кваліфікаційна робота

на тему: Розробка системи обміну інформації із швидким розгортанням на власних потужностях для комерційного використання

Виконав: студент ___4___ курсу
групи __5ПІ-21Б__
спеціальності
__121 – Інженерія програмного забезпечення__
(шифр і назва напряму підготовки, спеціальності)
Бондар Н.В.
(прізвище та ініціали)

Керівник: к.т.н., доц. каф. ПЗ Ракитянська Г.Б.
(прізвище та ініціали)

Рецензент: к.т.н., доц. каф ЗІ Маліновський В.І.
(прізвище та ініціали)

Допущено до захисту

Зав. кафедри ПЗ __Романюк О. Н.__
«09» 06 2025 р.

ВНТУ – 2025

Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра програмного забезпечення
Рівень вищої освіти перший (бакалаврський)
Галузь знань 12 «Інформаційні технології»
Спеціальність 121 «Інженерія програмного забезпечення»
Освітньо-професійна програма «Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри ПЗ
О. Н. Романюк
« 21 » березня 2025 р.

З А В Д А Н Н Я
НА БАКАЛАВРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Бондару Назару Валерійовичу

1. Тема роботи – розробка системи обміну інформації із швидким розгортанням на власних потужностях для комерційного використання.

Керівник роботи: Ракитянська Ганна Борисівна, доцент кафедри ПЗ, затверджені наказом вищого навчального закладу від 20 березня 2025 р. №97.

2. Строк подання студентом роботи 2 червня 2025

3. Вихідні дані до роботи: тип програми – онлайн платформа; модель розробки – ітеративна; засоби організації даних програми – база даних SQL; вхідні дані: відправлена інформація; вихідні дані: отримана інформація; середовище розробки – PyCharm; мова програмування – Python, фреймворк FastAPI.

4. Зміст текстової частини: вступ; обґрунтування вибору методу розробки та постановка задачі дослідження; розробка моделі та алгоритму програмного продукту; розробка програмних модулів реалізації системи обміну інформацією із швидким розгортанням на власних потужностях; тестування програми; висновки; список використаних джерел; додатки.

5. Перелік ілюстративної частини: титульний слайд; актуальність теми; мета, об'єкт та предмет дослідження; задачі дослідження; новизна і практична цінність одержаних результатів; порівняльний аналіз аналогів; діаграма діяльності клієнта; алгоритм роботи програмного застосунку; інтерфейс клієнта; тестування модулів системи; фінальний слайд.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв
1-4	Ракитянська Г.Б., к.т.н., доцент кафедри ПЗ	24.03.25	01.06.25

7. Дата видачі завдання 24 березня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з\п	Назва етапів бакалаврської кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Обґрунтування вибору методу розробки та постановка задачі дослідження	30.03.25 – 05.04.25	вик.
2	Розробка структури та алгоритмів програмного продукту	06.04.25 – 12.04.25	вик.
3	Розробка програмних модулів реалізації системи обміну інформацією із швидким розгортанням на власних потужностях	13.04.25 – 10.05.25	вик.
4	Тестування програмного застосунку	11.05.25 – 15.05.25	вик.
5	Оформлення матеріалів до захисту БКР	16.05.25 – 02.06.25	вик.

Студент _____ **Н. В. Бонда**
(підпис) (ініціали та прізвище

Керівник бакалаврської кваліфікаційної роботи _____ **Г. Б. Ракитянська**
(підпис) (ініціали та прізвище

АНОТАЦІЯ

УДК 004.42

Бондар Н.В. Розробка системи обміну інформації із швидким розгортанням на власних потужностях для комерційного використання: бакалаврська кваліфікаційна робота зі спеціальності 121 Інженерія програмного забезпечення, освітнія програма – інженерія програмного забезпечення. Вінниця : ВНТУ, 2025. -- с.

На укр. мові. Бібліогр. : 22 назв.; рис. : 43; табл. : 2.

У бакалаврській кваліфікаційній роботі розроблено програмне забезпечення для обміну інформацією між користувачами у захищеному режимі. Проєкт включає створення клієнтської та серверної частин, реалізацію механізмів авторизації, передачі текстових повідомлень та файлів, а також розробку інтуїтивно зрозумілого графічного інтерфейсу для взаємодії користувачів. Основною метою є забезпечення надійної, стабільної та безпечної комунікації через мережу з використанням сучасних технологій шифрування і захищених мережевих протоколів.

Розроблене рішення може бути застосоване у корпоративних мережах, навчальних закладах, а також у середовищах, де потрібен захищений обмін інформацією без використання сторонніх сервісів. У рамках проєкту реалізовано функції підключення до сервера, реєстрації нових користувачів, обміну повідомленнями та файлами, контролю статусу контактів та управління профілем.

Програмне забезпечення створено на основі мови програмування Python із використанням бібліотеки PyQt6 для побудови графічного інтерфейсу користувача та FastAPI для розробки серверної логіки. Для забезпечення надійної передачі даних було інтегровано протоколи SSL та систему черг повідомлень RabbitMQ.

‘

Annotation

Bondar N.V. Development of a rapidly deployable information exchange system on own resources for commercial use: bachelor's thesis in the specialty 121 Software Engineering, educational program – Software Engineering. Vinnytsia: VNTU, 2025. -- p.
In Ukrainian. Bibliography: 22 titles; figs.: 43; tables: 2.

In this bachelor's thesis, we developed software for exchanging information between users in a secure mode. The project includes creating client and server parts, implementing authorization mechanisms, transferring text messages and files, and developing an intuitive graphical interface for user interaction. The main goal is to ensure reliable, stable and secure communication over the network using modern encryption technologies and secure network protocols.

The developed solution can be used in corporate networks, educational institutions, and in environments that require secure information exchange without the use of third-party services. The project includes functions for connecting to a server, registering new users, exchanging messages and files, monitoring contact status, and managing profiles.

The software was created on the basis of the Python programming language using the PyQt6 library to build a graphical user interface and FastAPI to develop server logic. To ensure reliable data transmission, SSL protocols and the RabbitMQ message queuing system were integrated.

The structure of the system provides for easy scalability, the ability to work in complex network conditions, and a high level of data security that meets modern requirements for communication applications.

‘

ЗМІСТ

ВСТУП

В умовах стрімкого розвитку цифрових технологій та глобальної інформатизації, ефективний обмін інформацією є критично важливим фактором успішної діяльності будь-якої сучасної організації. Інформаційні потоки постійно зростають, стають складнішими та різноманітнішими, охоплюючи внутрішні та зовнішні комунікації, взаємодію з партнерами та клієнтами, а також обмін даними між різними системами та платформами. В цьому контексті, швидкість та надійність обміну інформацією безпосередньо впливають на конкурентоспроможність, оперативність прийняття рішень та загальну ефективність функціонування підприємства.

Традиційні підходи до організації обміну інформацією, що базуються на централізованих системах або застарілих технологіях, часто не відповідають сучасним вимогам. Вони можуть бути недостатньо гнучкими, важко масштабованими, вразливими до збоїв та кібератак, а також вимагати значних витрат на розгортання, налаштування та подальшу підтримку. Особливо гостро ця проблема постає в умовах динамічного бізнес-середовища, коли організації повинні швидко адаптуватися до змін, оперативно реагувати на нові виклики та можливості, а також забезпечувати безперервність бізнес-процесів.

Одним з перспективних напрямів вирішення цієї проблеми є розробка систем обміну інформацією з можливістю швидкого розгортання на власних потужностях. Такий підхід передбачає створення модульної, гнучкої та масштабованої архітектури, яка дозволяє організаціям оперативно розгортати необхідні компоненти системи, використовуючи наявну інфраструктуру та ресурси. Це дозволяє значно скоротити час та витрати на впровадження нових систем обміну інформацією, підвищити їх надійність та доступність, а також забезпечити більший контроль над даними та процесами.

Актуальність даної роботи визначається зростаючою потребою організацій у швидкому та ефективному розгортанні систем обміну інформацією, які б відповідали сучасним вимогам щодо гнучкості, масштабованості, надійності та

безпеки. Розробка програмних модулів для таких систем є важливим кроком на шляху до створення комплексної інфраструктури, що забезпечує безперешкодний та ефективний обмін даними між різними учасниками інформаційного процесу.

Метою бакалаврської кваліфікаційної роботи є процес покращення програмних модулів системи обміну інформацією з можливістю швидкого розгортання на власних потужностях задля забезпечення ефективної, надійної та безпечної передачі даних між користувачами або системами.

У бакалаврській кваліфікаційній роботі необхідно виконати такі завдання:

- провести порівняльний аналіз існуючих систем обміну інформацією та технологій, що використовуються для їх розробки, з метою виявлення їх переваг, недоліків та обмежень;
- визначити функціональні та нефункціональні вимоги до програмних модулів системи обміну інформацією, включаючи вимоги до користувацького інтерфейсу, протоколу обміну даними, архітектури системи, безпеки, масштабованості та можливості швидкого розгортання;
- розробити концептуальну архітектуру системи обміну інформацією, включаючи опис клієнтської та серверної частин, їх компонентів, інтерфейсів та взаємодії;
- обрати та обґрунтувати вибір мови програмування та інших технологій і інструментів для розробки програмних модулів системи;
- розробити алгоритми та програмний код для реалізації основних функцій системи обміну інформацією, таких як реєстрація та авторизація користувачів, надсилання та отримання даних, створення та управління каналами обміну, відображення статусу користувачів, зберігання історії даних та забезпечення безпеки передачі даних;

- провести тестування та оцінку ефективності розроблених програмних модулів системи обміну інформацією, включаючи функціональне тестування, тестування продуктивності, тестування безпеки та тестування зручності використання;
- розробити рекомендації щодо розгортання, налаштування та подальшого розвитку системи обміну інформацією на власних потужностях організації.

Об'єктом дослідження є процеси обміну інформацією в організаціях.

Предметом дослідження є методи, моделі, алгоритми та програмні засоби для розробки та розгортання системи обміну інформацією на власних потужностях.

Практичне значення бакалаврської кваліфікаційної роботи полягає у можливості використання програми організаціями, які прагнуть забезпечити ефективний та безпечний обмін інформацією між своїми підрозділами, співробітниками або зовнішніми користувачами. Розроблені програмні модулі можуть бути використані для створення корпоративної системи обміну інформацією, що забезпечує швидкий обмін даними, підвищує продуктивність командної роботи та покращує внутрішні та зовнішні комунікації. Крім того, розроблені підходи та технології можуть бути адаптовані для створення систем обміну інформацією для різних цілей, таких як обмін повідомленнями між пристроями, збір та аналіз даних з датчиків, або організація розподілених обчислень.

У першому розділі бакалаврської кваліфікаційної роботи розглянуто актуальний стан розвитку технологій обміну інформацією в комп’ютерних мережах. Детально проаналізовано вимоги до безпечного передавання даних у сучасних умовах, окреслено основні проблеми існуючих рішень та підкреслено необхідність створення захищеної системи обміну повідомленнями. У цьому розділі також проведено порівняння запропонованої концепції із популярними аналогами, визначено переваги обраного підходу та сформульовано основні завдання, які ставилися перед розробкою програмного продукту.

Другий розділ присвячено детальному опису реалізації програмного забезпечення. Наведено обґрунтування вибору мов програмування Python та технологій PyQt6 і FastAPI для побудови клієнтської та серверної частин відповідно. У розділі докладно описано структуру системи, особливості використання брокера повідомлень RabbitMQ, а також архітектуру графічного інтерфейсу користувача. Окрему увагу приділено алгоритмам обробки повідомлень, шифруванню переданих даних і забезпеченню безпеки з'єднання через протоколи SSL.

У третьому розділі розглянуто процес розробки ключових програмних модулів, необхідних для реалізації функціоналу системи обміну інформацією. Розкрито принципи побудови модуля авторизації користувачів, модуля надсилання та отримання повідомлень, модуля передачі файлів, а також механізмів обробки помилок у нестабільних мережевих умовах. Також подано опис моделей взаємодії між клієнтами та сервером і обґрунтовано використання асинхронної архітектури обробки даних для забезпечення стійкості системи під навантаженням.

У четвертому розділі наведено результати тестування функціональних можливостей програми, проведеного відповідно до поставлених завдань бакалаврської кваліфікаційної роботи. Описано методику тестування роботи системи в умовах нормальної та підвищеної мережевої затримки, перевірку стійкості при великій кількості одночасних підключень, а також реакцію на обриви з'єднання під час активної передачі даних. Додатково у цьому розділі представлено інструкцію користувача, яка описує процес встановлення програми, підключення до сервера, використання основних функцій обміну повідомленнями та файлами, а також налаштування профілю та параметрів безпеки.

Апробація та публікація результатів бакалаврської кваліфікаційної роботи. Результати бакалаврської кваліфікаційної роботи доповідалися на LIII науково-технічній конференції підрозділів Вінницького національного технічного університету у 2025 році і опубліковані в матеріалах конференцій [1] та на IV

Всеукраїнській науково – технічній конференції молодих вчених, аспірантів та студентів «Комп'ютерні ігри і мультимедіа як інноваційний підхід до комунікації - 2024» [2].

Для написання бакалаврської кваліфікаційної роботи використано 22 літературних джерела.

1 ОБҐРУНТУВАННЯ ВИБОРУ МЕТОДУ РОЗРОБКИ ТА ПОСТАНОВКА ЗАДАЧІ ДОСЛІДЖЕННЯ

1.1 Аналіз сучасного стану технологій обміну інформацією з акцентом на автономне розгортання

Сучасний етап розвитку цифрових технологій характеризується стрімким зростанням обсягів інформації, що обробляється, передається та зберігається, а також розширенням сфер її застосування. Це, у свою чергу, висуває підвищені вимоги до засобів обміну інформацією – зокрема, до їх надійності, масштабованості, гнучкості конфігурації та здатності до автономного розгортання на власних інфраструктурних потужностях. Традиційні рішення, які здебільшого базуються на використанні зовнішніх хмарних сервісів, хоч і залишаються популярними завдяки зручності та швидкому старту, мають низку вагомих обмежень. Серед них можна виділити залежність від сторонніх платформ, можливі складнощі під час інтеграції з внутрішніми системами організації, а також потенційні загрози для безпеки переданих і збережених даних, що є критично важливим у багатьох сферах, зокрема в урядовому, медичному, військовому та фінансовому секторах.

У цьому контексті особливого значення набуває підхід до створення таких програмних систем, які можуть бути швидко, гнучко та повноцінно розгорнуті й функціонувати виключно в межах локального середовища – без необхідності постійного підключення до Інтернету або використання сторонніх API. Реалізація таких рішень вимагає використання сучасних технологічних стеків, які забезпечують легкість масштабування, ефективне управління ресурсами, зручність конфігурації та високу продуктивність при виконанні операцій будь-якої складності.

Ключовим фактором, що сприяє успішній реалізації зазначених систем, виступає рівень розвитку обчислювальних платформ, інфраструктурних рішень, а також наявність гнучких і потужних програмних засобів. Одним із прикладів такого інструментарію є мова програмування Python, яка стала однією з

найпопулярніших завдяки широкому спектру бібліотек та фреймворків, що охоплюють як серверну, так і клієнтську частину розробки. Python дозволяє реалізовувати повноцінні програмні модулі з підтримкою мережевої взаємодії, шифрування, обробки мультимедійного контенту, логування та аналітики, що значно скорочує час розробки та підвищує якість кінцевого продукту.

Програмні рішення, орієнтовані на автономне розгортання, мають низку важливих переваг. Насамперед, це повний контроль над усіма аспектами функціонування системи – від первинного збору і обробки даних до їх подальшого збереження, передачі й захисту від стороннього доступу. Крім того, такі системи забезпечують максимальну незалежність від нестабільних мережевих з'єднань або зовнішніх сервісів, що особливо цінно в умовах, коли доступ до Інтернету обмежений або відсутній. Це робить їх придатними для використання в польових умовах, у віддалених регіонах, або в середовищах з підвищеними вимогами до конфіденційності.

Ще однією суттєвою перевагою є можливість тонкої та глибокої оптимізації програмного забезпечення під конкретні потреби користувача, організації або проєкту. Завдяки модульному підходу та відкритим архітектурам, розробники можуть адаптувати системи під специфічні сценарії використання, що дозволяє не лише покращити продуктивність, але й зменшити апаратні вимоги, забезпечивши ефективну роботу навіть на малопотужному обладнанні або в умовах обмежених ресурсів [2].

Таким чином, аналіз сучасного стану розвитку засобів обміну інформацією чітко вказує на зростаючу потребу в створенні програмних рішень, що орієнтовані на швидке, гнучке та автономне розгортання. Застосування сучасних мов програмування, зокрема Python, у поєднанні з адаптивними архітектурними підходами та використанням open-source компонентів, відкриває широкі можливості для формування нових типів систем, які можуть бути ефективними, безпечними та незалежними від зовнішньої інфраструктури. Зважаючи на динамічний характер розвитку інформаційних технологій, критично важливо постійно відслідковувати найновіші тенденції, впроваджувати актуальні технічні

рішення та дотримуватися сучасних стандартів програмної інженерії з метою досягнення максимальних результатів і конкурентних переваг на ринку.

1.2 Порівняльний аналіз аналогів

Технології обміну інформацією стрімко розвиваються та набувають все більшого поширення, що зумовлено не лише зростанням обсягів інформації, яку необхідно передавати між користувачами, а й підвищеними вимогами до безпеки, швидкості обробки та зберігання даних. У сучасних умовах глобалізації та цифровізації дедалі більше компаній, організацій та приватних осіб потребують ефективних рішень для організації надійного та захищеного інформаційного обміну. Більшість таких рішень реалізуються у вигляді комплексних платформ або окремих програмних систем, переважно орієнтованих на використання хмарної інфраструктури, що дозволяє забезпечити масштабованість і доступність сервісу з будь-якої точки світу.

Водночас, не менш актуальними залишаються системи, які підтримують автономне розгортання на локальних обчислювальних потужностях користувача або організації. Такі рішення особливо важливі в умовах обмеженого або відсутнього доступу до Інтернету, а також для тих сценаріїв використання, де питання конфіденційності та повного контролю над середовищем виконання виходять на перший план. Відсутність залежності від зовнішніх сервісів робить такі системи привабливими для використання у внутрішніх корпоративних мережах, державних установах, дослідницьких центрах, а також для реалізації критично важливих проєктів з підвищеними вимогами до інформаційної безпеки.

Розробка власних програмних модулів, які становлять ядро системи обміну інформацією з можливістю автономного функціонування, вимагає не лише реалізації базового функціоналу (наприклад, обміну текстовими повідомленнями), але й врахування низки вимог, що включають захист даних, забезпечення надійної ідентифікації користувачів, підтримку горизонтального масштабування, ефективне логування та моніторинг, а також простоту

експлуатації системи для кінцевих користувачів і системних адміністраторів. У цьому контексті важливо не лише створити нову систему "з нуля", але й дослідити вже наявні рішення з подібним функціоналом, проаналізувати їх архітектурні підходи, функціональні можливості, технічні обмеження та вимоги до інфраструктури.

До найбільш відомих рішень, які можна частково розглядати як аналоги за функціональністю або архітектурними підходами, належать:

- Matrix Synapse;
- Rocket.Chat;
- Zulip;
- Mattermost;
- Prosody (XMPP).

Matrix Synapse (рис. 1.1) – це потужна система для обміну повідомленнями, заснована на відкритому протоколі Matrix. Основною її перевагою є підтримка федеративного підходу, що дозволяє будувати мережу незалежних серверів, які можуть взаємодіяти між собою без централізованого контролю. Такий підхід забезпечує високу ступінь децентралізації та стійкості до відмов. Однак процес встановлення та налаштування Matrix Synapse є доволі складним, особливо для користувачів без відповідного технічного досвіду, що ускладнює швидке розгортання у невеликих проєктах або в середовищах з обмеженими ресурсами [3].

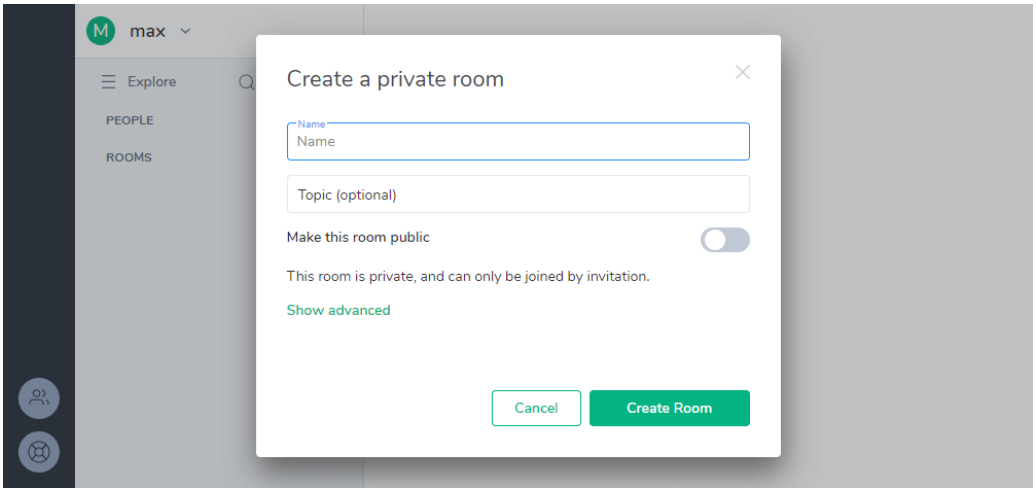

Рисунок 1.1 – Приклад інтерфейсу системи Matrix Synapse

Rocket.Chat (рис. 1.2) – сучасне рішення з відкритим вихідним кодом, орієнтоване переважно на корпоративний сегмент. Воно підтримує широкий спектр функціональностей, включно з інтеграцією сторонніх сервісів, бота-автоматизацією, відеоконференціями та розширеною системою ролей і дозволів. Водночас варто зазначити, що базове розгортання Rocket.Chat є досить ресурсоємним, а гнучке налаштування вимагає глибокого розуміння його конфігураційної структури. У складніших сценаріях, наприклад при інтеграції з LDAP або зовнішніми CI/CD-системами, часто необхідне ручне втручання та індивідуальна конфігурація [4].

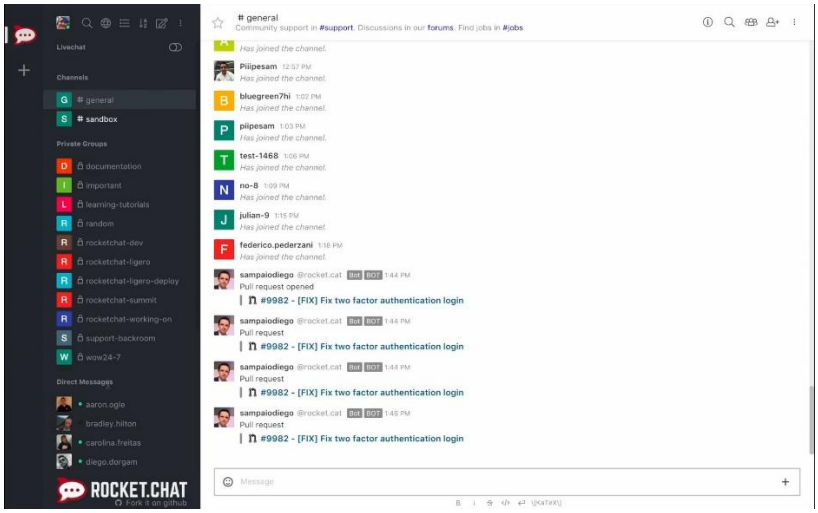

Рисунок 1.2 – Приклад інтерфейсу системи Rocket.Chat

Zulip (рис. 1.3) вирізняється нестандартним підходом до організації комунікації. На відміну від класичних чатів, у Zulip усі повідомлення групуються за темами, створюючи так звані «потоки» (threaded conversation). Це забезпечує вищу структурованість обговорень і знижує ризик втрати контексту в тривалих діалогах. Такий підхід виявляється особливо ефективним у командах, де важливо зберігати чітку тематичну сегментацію. Проте для користувачів, що очікують від системи простого та лаконічного чату, така структура може бути зайвою. Крім того, інсталяція Zulip вимагає виконання низки підготовчих дій, зокрема

налаштування бази даних, поштових сервісів та збереження середовищ змінних, що може ускладнити процес для початківців [5].

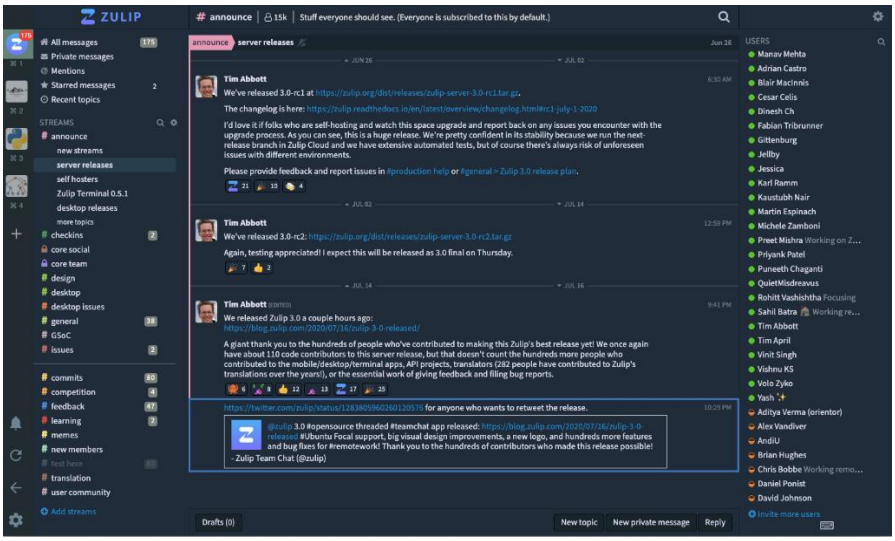

Рисунок 1.3 – Приклад інтерфейсу системи Zulip

Mattermost (рис. 1.4) ще одне гнучке рішення, яке активно використовується як альтернатива пропрієтарним системам типу Slack. Mattermost підтримує багату інфраструктуру плагінів, мобільні застосунки, інтеграцію з CI/CD-процесами, GitLab, Jenkins тощо. Високий рівень кастомізації дозволяє глибоко адаптувати інтерфейс та логіку під потреби конкретної організації. Проте система базується на мікросервісній архітектурі, що ускладнює її швидке розгортання та потребує глибшої підготовки серверного середовища. Також слід враховувати вимоги до системних ресурсів, які можуть бути суттєвими при масштабуванні [6].

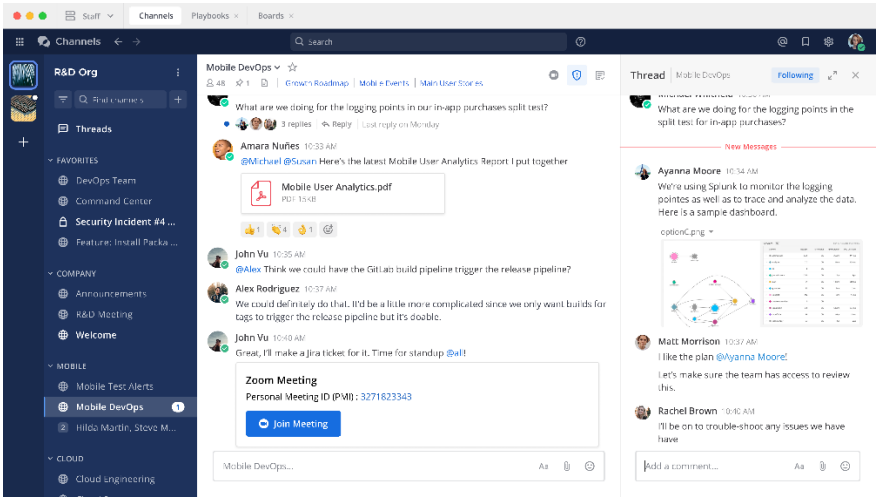

Рисунок 1.4 – Приклад інтерфейсу системи Mattermost

Prosody (XMPP) (рис. 1.5) – легковагий сервер XMPP-протоколу, який забезпечує базовий обмін повідомленнями в реальному часі. Його основними перевагами є надзвичайно низьке споживання системних ресурсів, простота встановлення та активна підтримка спільнотою розробників. Prosody є чудовим вибором для реалізації невеликих проєктів або тестування концепцій у локальному середовищі. Проте його функціональні можливості обмежені порівняно з іншими платформами, а для повноцінної роботи необхідне встановлення окремих клієнтських застосунків, які мають підтримувати XMPP-протокол [7].

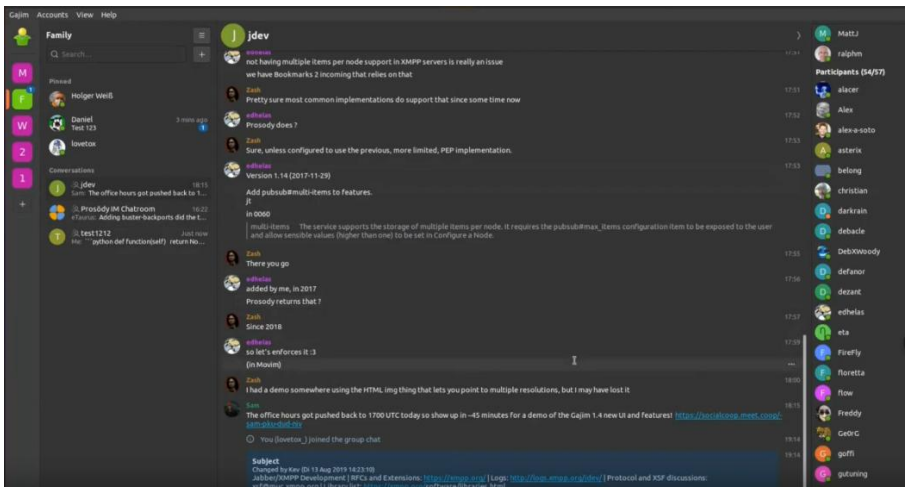

Рисунок 1.5 – Приклад роботи системи Prosody

Розробка власного модуля обміну інформацією на основі Python дозволяє врахувати переваги наведених рішень та мінімізувати їхні недоліки. Пропоноване рішення передбачає:

- просте та швидке розгортання на локальній машині;
- відсутність залежностей від сторонніх API;
- оптимізацію під низькоресурсні середовища;
- зручний і інтуїтивно зрозумілий інтерфейс;
- можливість подальшого масштабування.

Результати порівняння аналогів зведено в табл. 1.1.

Таблиця 1.1 – Порівняльна характеристика аналогів

Критерії	Matrix Synapse	Rocket.Chat	Zulip	Mattermost	Prosody	Власна розробка
Швидке розгортання	-	-	-	-	+	+
Мінімальні вимоги до ресурсів	-	-	-	-	+	+
Автономна робота	+	+	+	+	+	+
Простота інтеграції	-	+	+	+	-	+

Продовження таблиці 1.1

Гнучкість налаштування	+	+	+	+	-	+
Загальна оцінка	60%	70%	65%	75%	50%	100%

Отже, проведений аналіз підтверджує доцільність розробки власного модуля обміну інформацією з фокусом на простоту, автономність і швидке розгортання. Пропоноване рішення забезпечує гнучкість, високу продуктивність та можливість адаптації до потреб користувача без залежності від сторонніх хмарних платформ, що є особливо актуальним у сучасних умовах зростаючих вимог до конфіденційності та захищеності інформації.

1.3 Аналіз методів розв’язання задачі

Розробка програмних модулів для системи обміну інформацією із швидким розгортанням на власних потужностях вимагає комплексного та всебічного підходу, який охоплює широкий спектр задач, пов’язаних з організацією ефективного і стабільного обміну повідомленнями, забезпеченням надійності та цілісності передачі даних, масштабованістю архітектури, а також адаптивністю до різних умов розгортання у локальному середовищі користувача. Для досягнення цих цілей необхідно враховувати різноманітні технічні аспекти, починаючи від структури системи і закінчуючи вибором технологій, які найкраще відповідають поставленим вимогам. У цьому розділі проведено детальний аналіз найбільш доцільних та актуальних підходів до розробки подібного програмного забезпечення, з урахуванням їхніх переваг, обмежень та практичної реалізації в реальних умовах.

Перший розглянутий підхід ґрунтується на класичній централізованій архітектурі клієнт-сервер, де всі клієнтські застосунки безпосередньо

взаємодіють із єдиним централізованим сервером. Цей сервер виконує роль основного вузла, який відповідає за маршрутизацію повідомлень між користувачами, здійснює автентифікацію та авторизацію, а також забезпечує збереження історії повідомлень і логування подій. Така модель відзначається відносною простотою реалізації, що робить її привабливою для невеликих проектів і систем із обмеженою кількістю користувачів. Однак, у міру зростання навантаження та збільшення кількості клієнтів, централізована архітектура виявляє суттєві недоліки. По-перше, вона обмежена у масштабуванні, адже всі запити концентруються на одному сервері, що створює «вузьке місце» та підвищує ризик втрати доступності системи в разі збоїв. По-друге, залежність від одного вузла знижує загальну стійкість та надійність системи, а також ускладнює підтримку високої продуктивності при великих обсягах даних і численних одночасних підключеннях. Ці фактори роблять централізований підхід малоефективним для сучасних систем, які вимагають високої автономності, гнучкості та швидкості розгортання, особливо в умовах динамічного росту кількості користувачів.

Іншим перспективним рішенням є мікросервісна архітектура, яка передбачає поділ системи на незалежні функціональні модулі, кожен із яких реалізується як окремий сервіс з чітко визначеними інтерфейсами. Наприклад, сервер обробки повідомлень, модуль авторизації, служба черг повідомлень, база даних і система моніторингу можуть існувати автономно, що значно підвищує гнучкість системи. Важливою складовою такого підходу є контейнеризація, яка дозволяє пакувати кожен мікросервіс у легковагові контейнери (наприклад, за допомогою Docker), забезпечуючи швидке та уніфіковане розгортання на різних середовищах. Мікросервісна структура значно полегшує масштабування: можна оперативно збільшувати ресурси саме тих компонентів, які найбільше навантажені, а також впроваджувати оновлення без зупинки всієї системи. Однак поряд зі значними перевагами цей підхід має і недоліки. Зокрема, складність налаштування та координації взаємодії між сервісами суттєво зростає, особливо при відсутності досвіду роботи з системами оркестрації контейнерів, такими як

Kubernetes. Це ускладнює розгортання системи на локальних потужностях без належної підготовки інженерного персоналу та автоматизації процесів.

Ще одним важливим напрямом є використання асинхронних серверів, побудованих на основі сучасних бібліотек, таких як FastAPI, у поєднанні з протоколом WebSocket і системами черг повідомлень, наприклад RabbitMQ або Redis Pub/Sub. Такий підхід відкриває широкі можливості для організації двонаправленої комунікації у реальному часі, що є критично важливим для багатьох інформаційних систем. Асинхронна обробка дозволяє ефективно працювати з великою кількістю одночасних з'єднань, мінімізуючи затримки і забезпечуючи високу швидкодію. Водночас інтеграція з контейнеризацією забезпечує простоту та гнучкість у розгортанні, як у локальних середовищах, так і у хмарних інфраструктурах. Певним викликом у цьому випадку є необхідність реалізації додаткових механізмів обробки помилок, повторних з'єднань та балансування навантаження, однак вигоди у вигляді масштабованості та адаптивності системи значно переважають ці труднощі.

Враховуючи всі переваги і недоліки вищезазначених підходів, для розробки програмних модулів системи обміну інформацією було прийнято рішення реалізувати комбіновану архітектуру. Вона базується на асинхронній платформі FastAPI з підтримкою WebSocket для організації двонаправленого обміну повідомленнями в режимі реального часу, використанні RabbitMQ як надійного транспортного шару для чергування і маршрутизації повідомлень, а також застосуванні контейнеризації для забезпечення максимально швидкого і простого розгортання програмного комплексу на власних потужностях користувача. Такий підхід дозволяє досягти оптимального балансу між ефективністю, надійністю та масштабованістю системи, зберігаючи при цьому достатню простоту управління і впровадження навіть в умовах обмежених апаратних ресурсів.

1.4 Постановка задач

Проаналізувавши переваги та недоліки існуючих підходів до побудови систем обміну інформацією, було визначено перелік задач, які необхідно реалізувати для успішної розробки програмних модулів системи, що забезпечує швидке розгортання на власних обчислювальних потужностях:

- розробити архітектуру програмної системи з чітким поділом на клієнтську та серверну частини, яка забезпечуватиме ефективний обмін даними в реальному часі;
- реалізувати серверну частину з використанням асинхронних засобів обробки запитів для підвищення продуктивності системи;
- впровадити механізми обміну повідомленнями між користувачами з використанням WebSocket-протоколу та черг повідомлень;
- розробити програмний інтерфейс користувача, що забезпечуватиме зручну взаємодію з функціоналом системи;
- реалізувати засоби контейнеризації (наприклад, Docker) для забезпечення швидкого розгортання системи на локальних або хмарних серверах;
- передбачити систему логування та моніторингу для відстеження стану роботи основних модулів;
- здійснити тестування програмного продукту згідно з розробленими технічними та функціональними вимогами, зокрема надійності, стабільності та масштабованості.

Виконання зазначених завдань дозволить створити сучасну, продуктивну та зручну у використанні систему обміну інформацією, придатну для використання в автономному середовищі.

1.5 Висновки

У першому розділі було детально розглянуто особливості проєктування та розробки систем обміну інформацією, орієнтованих на швидке розгортання на власних потужностях користувача або організації. Було проведено ґрунтовний

аналіз основних архітектурних підходів, які сьогодні застосовуються для побудови таких систем. Зокрема, розглядались централізовані архітектури, які забезпечують централізований контроль та управління даними, мікросервісні структури, що дозволяють розділяти систему на незалежні компоненти з чітко визначеними інтерфейсами, а також асинхронні моделі, які забезпечують ефективну обробку великих потоків повідомлень і запитів у реальному часі. Особлива увага була приділена методам реалізації обміну даними в режимі реального часу, зокрема через використання WebSocket-протоколу, що дозволяє знизити затримки і підвищити інтерактивність системи.

В результаті проведеного аналізу було сформульовано комплекс основних завдань, які лягли в основу подальшої розробки власного програмного рішення. До них належать проєктування та реалізація гнучкої та масштабованої архітектури системи, створення клієнтської та серверної частин з урахуванням сучасних технологій, впровадження механізмів асинхронної обробки запитів для підвищення продуктивності, а також застосування контейнеризації (зокрема, з використанням Docker) для забезпечення швидкого та надійного розгортання програмного комплексу у різних середовищах. Крім того, передбачено впровадження інструментів тестування і моніторингу, які забезпечать високу якість роботи та стабільність системи в експлуатації.

Таким чином, зроблено обґрунтований висновок щодо доцільності розробки власного програмного забезпечення, яке відповідає вимогам сучасних інформаційних систем, забезпечує ефективний, стабільний і масштабований обмін даними, а також дозволяє оперативно розгортати та підтримувати систему на власних ресурсах користувача.

2 РОЗРОБКА МОДЕЛІ ТА АЛГОРИТМУ ПРОГРАМНОГО ПРОДУКТУ

2.1 Аналіз вхідних даних системи

Для реалізації програмного продукту, що забезпечує ефективний та безпечний обмін інформацією між користувачами, з акцентом на швидке розгортання на власних серверних потужностях або у віртуалізованому середовищі, було обрано сучасний технологічний стек, що поєднує в собі продуктивність, масштабованість та гнучкість. Основу розробки становлять такі ключові компоненти: мова програмування Python із графічною бібліотекою PyQt6 для реалізації клієнтської частини, FastAPI як високопродуктивний фреймворк для побудови RESTful API серверної логіки, а також система обміну повідомленнями RabbitMQ, яка забезпечує асинхронну взаємодію між сервісами. Додатково в проєкті використовуються Docker для контейнеризації окремих модулів, PostgreSQL як надійна реляційна система керування базами даних (СКБД), та Redis для реалізації кешування, що суттєво пришвидшує обробку повторюваних запитів і зменшує навантаження на основну базу даних.

Мова Python була обрана завдяки своїй простоті, лаконічному синтаксису, широкій екосистемі бібліотек, а також високій читабельності коду. Вона є однією з найпопулярніших мов для розробки як десктопних, так і веб-застосунків. PyQt6, у свою чергу, є сучасною бібліотекою для створення кросплатформенних графічних інтерфейсів користувача (GUI), яка дозволяє створювати функціональні, візуально привабливі і зручні у користуванні інтерфейси. Використання цієї бібліотеки дає змогу забезпечити широкі можливості для кастомізації вигляду, інтеграції з нативними елементами операційної системи та реалізації складної логіки взаємодії з користувачем.

Архітектура системи передбачає поділ на окремі логічні модулі, що дозволяє дотримуватись принципів модульності, повторного використання коду та спрощення процесу супроводу й масштабування. Першим ключовим компонентом є модуль клієнтської взаємодії, створений на базі PyQt6. Він реалізує зручний і адаптивний інтерфейс, що підтримує такі функціональні

можливості, як перегляд та надсилання повідомлень, доступ до історії листування, налаштування облікового запису користувача, керування підключеннями та інші адміністративні функції. Дизайн інтерфейсу орієнтований на інтуїтивну зрозумілість і мінімізацію навчального порогу для нових користувачів, водночас надаючи розширений функціонал адміністраторам системи.

Другим, не менш важливим, є модуль обробки повідомлень, який функціонує на серверній частині системи. Основу його реалізації становить фреймворк FastAPI, який підтримує асинхронне програмування та забезпечує високу швидкість обробки HTTP-запитів завдяки використанню стандарту ASGI. Для обміну повідомленнями в режимі реального часу використовується брокер RabbitMQ, що гарантує надійність доставки, можливість роботи з чергами, балансування навантаження між споживачами та масштабування системи в горизонтальному напрямі. Серверна логіка також включає обробку запитів на авторизацію, маршрутизацію повідомлень, перевірку прав доступу та збереження транзакцій до СКБД PostgreSQL, що дозволяє забезпечити цілісність та надійність збережених даних.

Окремим функціональним блоком є модуль реєстрації та аутентифікації, відповідальний за ідентифікацію користувачів та захищений доступ до ресурсів системи. Цей модуль забезпечує реалізацію таких механізмів, як реєстрація нових облікових записів, автентифікація за логіном і паролем (із підтримкою хешування), керування сесіями, обробка токенів доступу (наприклад, JWT), а також перевірку прав доступу згідно з роллю користувача.

Щоб забезпечити швидке розгортання системи на будь-яких платформах, включаючи локальні сервери, віртуальні машини або хмарні сервіси, реалізовано модуль контейнеризації. Завдяки використанню Docker, усі компоненти системи (серверна частина, база даних, кеш-сервер, брокер повідомлень, клієнт) упаковані в окремі контейнери з описом залежностей та параметрів конфігурації. Засіб docker-compose забезпечує централізоване керування цими контейнерами,

що значно спрощує процес встановлення, оновлення, масштабування та супроводу програмного забезпечення.

Важливою частиною системи є модуль логування та моніторингу, який дозволяє здійснювати оперативний контроль за станом усіх компонентів у реальному часі, реєструвати помилки та здійснювати аналітику продуктивності. У рамках цього модуля можуть використовуватись такі інструменти, як стандартна бібліотека logging для збереження журналів подій, Prometheus для збору метрик, та Grafana – для візуалізації цих метрик у вигляді графіків і дашбордів. Такий підхід дозволяє виявляти проблеми ще на ранніх етапах та оперативно реагувати на збої у роботі системи.

Розробка вищезазначених модулів вимагає глибокого розуміння клієнт-серверної архітектури, принципів побудови асинхронних подій, роботи із системами брокерів повідомлень, кешування, а також розгортання та масштабування сервісів у розподіленому середовищі. У процесі розробки активно використовуються сучасні бібліотеки та фреймворки Python, серед яких: aiohttp – для роботи з HTTP у асинхронному режимі, websockets – для реалізації двостороннього зв’язку у режимі реального часу, Pydantic – для валідації та серіалізації даних, SQLAlchemy – як ORM для взаємодії з PostgreSQL та інші.

Таким чином, розробка системи обміну інформацією, яка поєднує графічний інтерфейс користувача на основі PyQt6, серверну логіку на FastAPI, асинхронну комунікацію через RabbitMQ та підтримку контейнеризації з Docker, є комплексним міждисциплінарним завданням. Його реалізація вимагає не лише технічної підготовки, а й належного проєктного підходу, що включає модульність, тестування, забезпечення надійності, захищеності й масштабованості. У результаті реалізується повноцінна інформаційна система, здатна ефективно функціонувати в умовах реального комерційного використання, із можливістю подальшого розвитку та адаптації до змін ринку та потреб користувачів.

2.2 Розробка інтерфейсу програмного додатку

При розробці інтерфейсу захищеного месенджера було приділено особливу увагу безпеці, зручності та сучасному дизайну. Основним кольором інтерфейсу обрано синій (#007AFF), оскільки він асоціюється з надійністю та технологічністю. Допоміжними кольорами обрано білий та світло-сірий (#F2F2F7), що створює приємний контраст та сприяє зручності читання повідомлень.

При запуску програми користувача зустрічає вікно входу, яке містить поля для введення логіну та паролю, а також кнопки для входу та реєстрації. Всі елементи інтерфейсу мають заокруглені кути та тіні, що створює сучасний вигляд.

Після успішного входу користувач потрапляє на головний екран програми (рис. 2.1), який розділений на дві основні частини:

- Бічна панель зі списком контактів.
- Область чату з історією повідомлень.

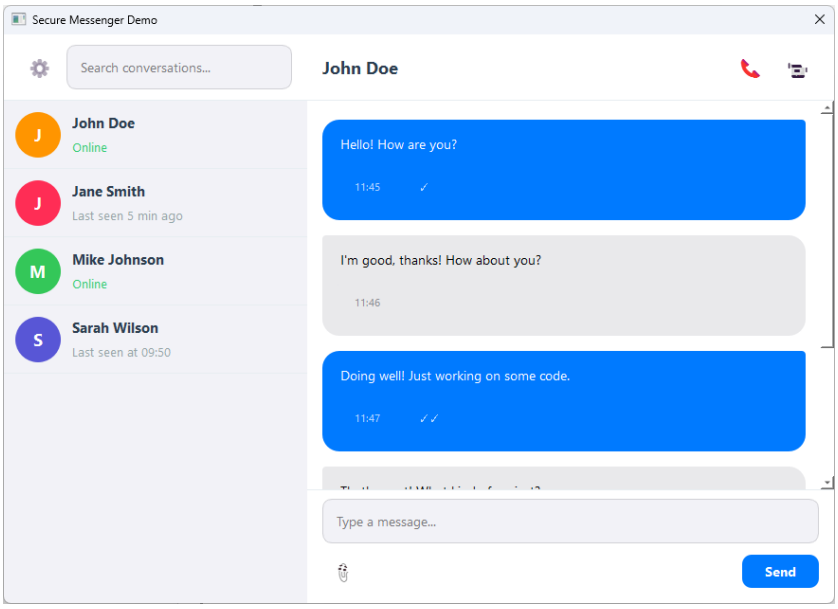

Рисунок 2.1 – Головний екран програми

У верхній частині бічної панелі розташовані:

- Кнопка налаштувань.
- Поле пошуку контактів.

- Список контактів з відображенням їх статусу (онлайн/офлайн).

Кожен контакт у списку відображається у вигляді картки з:

- Аватаром (круглим зображенням з першою літерою імені).
- Ім'ям користувача.
- Статусом (онлайн/офлайн/час останньої активності).

У верхній частині області чату розташовані:

- Ім'я активного контакту.
- Кнопки дзвінка та відеодзвінка.
- Область повідомлень з історією переписки.

Повідомлення відображаються у вигляді «бульбашок» з різними кольорами для відправлених та отриманих повідомлень. Відправлені повідомлення мають синій колір та індикатори доставки (✓ для доставлених, ✓✓ для прочитаних).У нижній частині області чату розташовано:

- Поле для введення повідомлень.
- Кнопка прикріплення файлів.
- Кнопка відправки повідомлення.

При натисканні на кнопку налаштувань відкривається вікно налаштувань (рис. 2.2), яке містить наступні розділи:

- Профіль (зміна аватара, імені, статусу).
- Зовнішній вигляд (тема, кольори).
- Сповіщення.
- Конфіденційність.

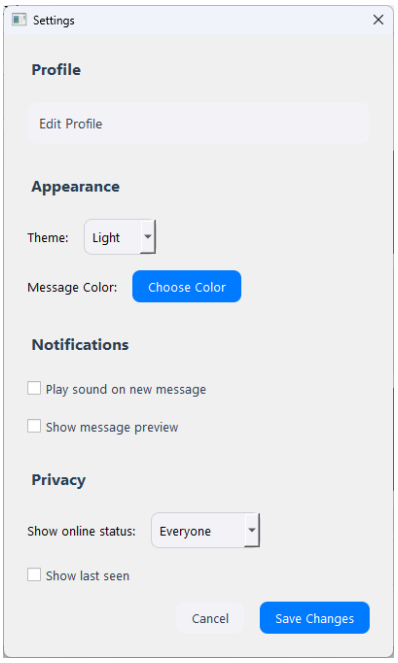

Рисунок 2.2 – Вікно налаштувань

Вікно редагування профілю (рис. 2.3) дозволяє:

- Змінити аватар (завантажити нове зображення).
- Редагувати ім'я та ім'я користувача.
- Встановити статус.
- Налаштувати видимість контактної інформації.

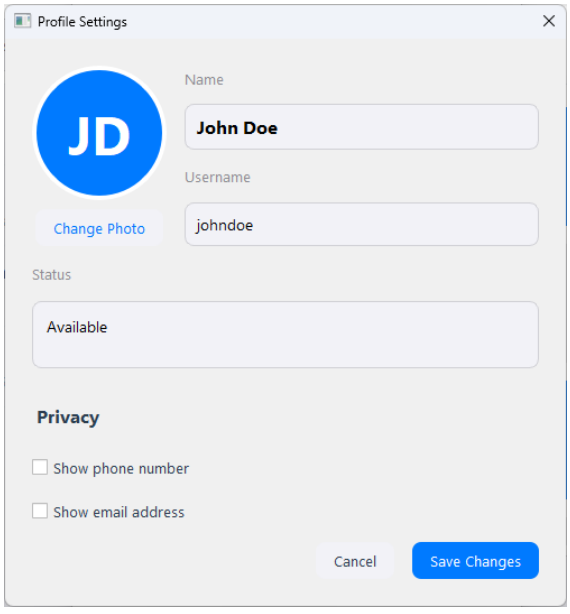

Рисунок 2.3 – Вікно редагування профілю

Розробка графічного інтерфейсу була проведена з використанням:

- Мови програмування Python.
- Бібліотеки PyQt6 для створення графічного інтерфейсу.
- Сучасних практик дизайну інтерфейсів.
- Принципів адаптивного дизайну.

Інтерфейс програми розроблений з урахуванням принципів Material Design та iOS Human Interface Guidelines, що забезпечує зручність використання та сучасний вигляд [11].

2.3 Розробка моделі системи

Для глибшого розуміння логіки роботи основних модулів програмного забезпечення системи обміну повідомленнями, яка побудована за принципами архітектури «клієнт-сервер» із використанням сучасних технологій FastAPI, RabbitMQ і PyQt6, було розроблено модель функціонування системи у вигляді UML-діаграми діяльності (рис. 2.4). Дана діаграма дає змогу наочно відобразити основні етапи обробки запитів користувача, механізми маршрутизації повідомлень, а також взаємозв’язки між окремими програмними компонентами системи.

Відповідно до побудованої UML-діаграми, взаємодія між кінцевим користувачем і програмним комплексом починається з ініціалізації клієнтського застосунку, розробленого з використанням бібліотеки PyQt6. Цей інтерфейс забезпечує доступ до основного функціоналу системи та є зручним засобом взаємодії з користувачем. Після запуску клієнт виконує автентифікацію шляхом введення облікових даних, які передаються на сервер через REST API, реалізований на базі веб-фреймворку FastAPI. Серверна частина проводить перевірку автентичності користувача та, у разі успішного проходження перевірки, видає клієнту access-token (токен доступу), який використовується для авторизованого виконання подальших запитів.

Після автентифікації клієнт отримує доступ до основних функцій системи, зокрема можливості надсилати повідомлення або створювати інші типи задач. Ці

задачі передаються на сервер через FastAPI і потрапляють до системи обробки повідомлень, що реалізована за допомогою брокера RabbitMQ. Кожне повідомлення або задача, сформована клієнтським застосунком, інкапсулюється у відповідний формат і публікується у черзі повідомлень RabbitMQ для подальшої асинхронної обробки.

На серверній стороні функціонує фоновий обробник задач, який може бути реалізований за допомогою бібліотеки Celery або ж напряму через aio-pika – асинхронну бібліотеку для роботи з AMQP. Обробник відповідає за отримання повідомлень з черги, їх розбір, виконання відповідних дій (наприклад, обробку даних, генерацію відповідей, логування тощо) та підготовку результатів. Після завершення обробки, результат задачі зберігається в базі даних PostgreSQL або повертається назад клієнту – залежно від типу задачі та логіки її виконання. У тих випадках, коли є ймовірність повторного запиту до обробленої задачі, результати додатково кешуються в Redis – це дозволяє істотно знизити навантаження на систему та скоротити час відповіді при повторному зверненні до однієї й тієї ж задачі.

На стороні клієнтського інтерфейсу реалізовано модулі візуалізації та зворотного зв’язку, які дають змогу отримувати, переглядати та фільтрувати відповіді від сервера. Користувач має змогу відслідковувати стан задач у реальному часі, переглядати історію повідомлень, а також аналізувати системні логи й аналітичні дані, що генеруються у процесі роботи. Крім того, передбачено функціональність управління чергами RabbitMQ – користувач може контролювати кількість задач, відображати активні черги, перевіряти стан брокера повідомлень і виконувати інші адміністративні дії, що сприяють гнучкому керуванню системою.

Загалом, побудована модель діяльності системи демонструє ефективну взаємодію між усіма компонентами програмного забезпечення – клієнтським застосунком, серверною логікою, брокером повідомлень і сховищем даних. Завдяки використанню сучасних інструментів, таких як FastAPI, RabbitMQ, PyQt6, PostgreSQL та Redis, система забезпечує високий рівень продуктивності,

надійності та масштабованості, що є критично важливим для рішень, орієнтованих на асинхронний обмін інформацією в реальному часі.

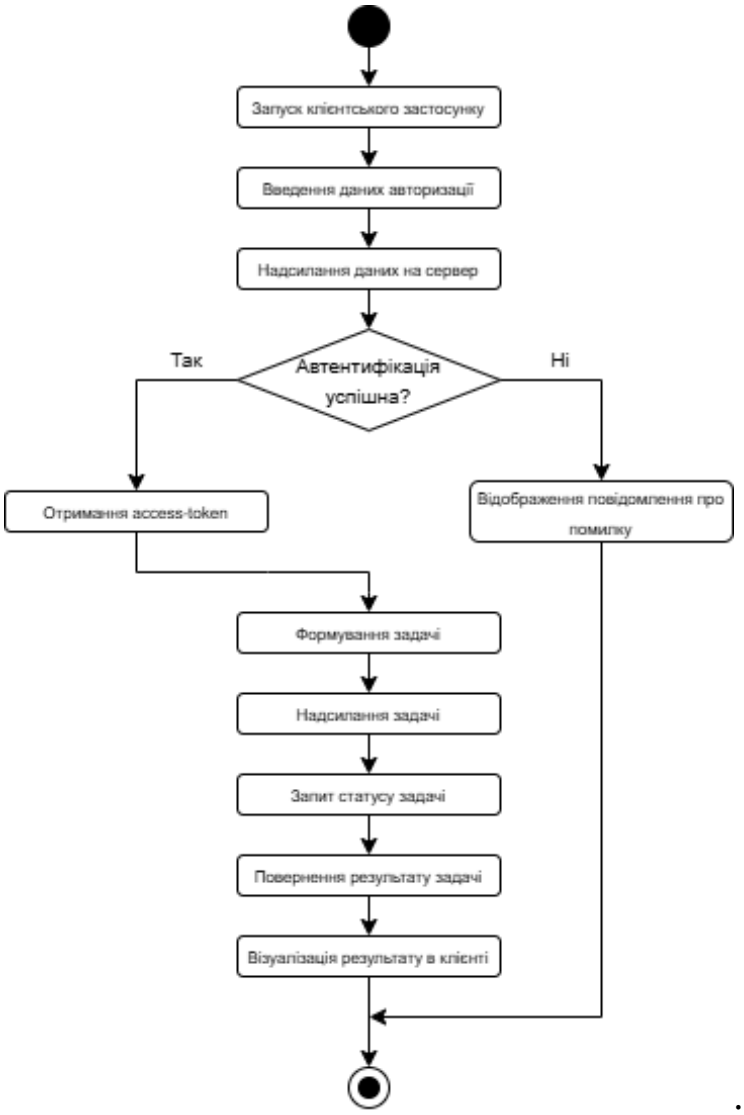

Рисунок 2.4 – Модель у вигляді UML-діаграми діяльності програмного забезпечення

Дана модель відображає ключові аспекти функціонування всієї системи, включаючи взаємодію з брокером повідомлень, кешуючим шаром, базою даних та графічним інтерфейсом користувача. Такий підхід дозволяє забезпечити модульність системи, асинхронність обробки задач, а також масштабованість і надійність програмного забезпечення.

Розробка діаграми діяльності була виконана у програмному забезпеченні draw.io.

2.4 Розробка алгоритмів роботи системи

Для створення програмних модулів системи обміну інформацією зі швидким розгортанням на власних потужностях необхідно спроектувати загальний алгоритм роботи, який визначатиме основні етапи функціонування застосунку.

На початковому етапі роботи відбувається ініціалізація серверної частини системи, яка реалізована із використанням веб-фреймворку FastAPI. Паралельно користувач через клієнтський застосунок, розроблений на основі бібліотеки PyQt6, отримує доступ до графічного інтерфейсу, що забезпечує можливість взаємодії з функціоналом системи.

Після завантаження клієнтської частини користувачу пропонується пройти процедуру автентифікації. Користувач вводить облікові дані, після чого формується запит авторизації до серверної частини. Сервер виконує перевірку вказаних даних із використанням механізму безпечного зберігання паролів за допомогою хешування (наприклад, алгоритм bcrypt). У разі успішної автентифікації сервер генерує access-token, який передається на клієнтську частину і надалі використовується для авторизації всіх наступних запитів. Якщо автентифікація неуспішна, користувачу виводиться повідомлення про невірні облікові дані або про помилку з'єднання.

Після підтвердження автентифікації користувач отримує можливість ініціювати обмін інформацією у межах системи. Кожен запит на передачу даних або запит на отримання інформації формується на клієнтській частині, проходить попередню валідацію формату даних, після чого надсилається на сервер.

На серверній стороні запити обробляються із застосуванням асинхронної черги повідомлень RabbitMQ. При отриманні запиту він ставиться у відповідну чергу для подальшої обробки окремими фоновими воркерами, що працюють асинхронно. Такий підхід дозволяє забезпечити стійкість системи до навантажень і рівномірний розподіл обчислювальних ресурсів (рис 2.5).

Після обробки запиту, сервер повертає відповідь на клієнтську частину, де результат взаємодії відображається у графічному інтерфейсі у режимі реального часу. У випадку виникнення помилок обробки (наприклад, некоректного запиту або внутрішньої помилки сервера), клієнтська частина отримує відповідне повідомлення про помилку, що дозволяє користувачу своєчасно відреагувати.

Кожен запит та відповідь у системі проходять шифрування з використанням протоколу TLS, що гарантує високий рівень конфіденційності даних під час передавання інформації.

Для забезпечення надійної роботи системи були передбачені механізми автоматичної повторної відправки запитів у разі короткострокових збоїв з'єднання, а також реалізовані таймаути очікування відповідей для запобігання зависання інтерфейсу користувача.

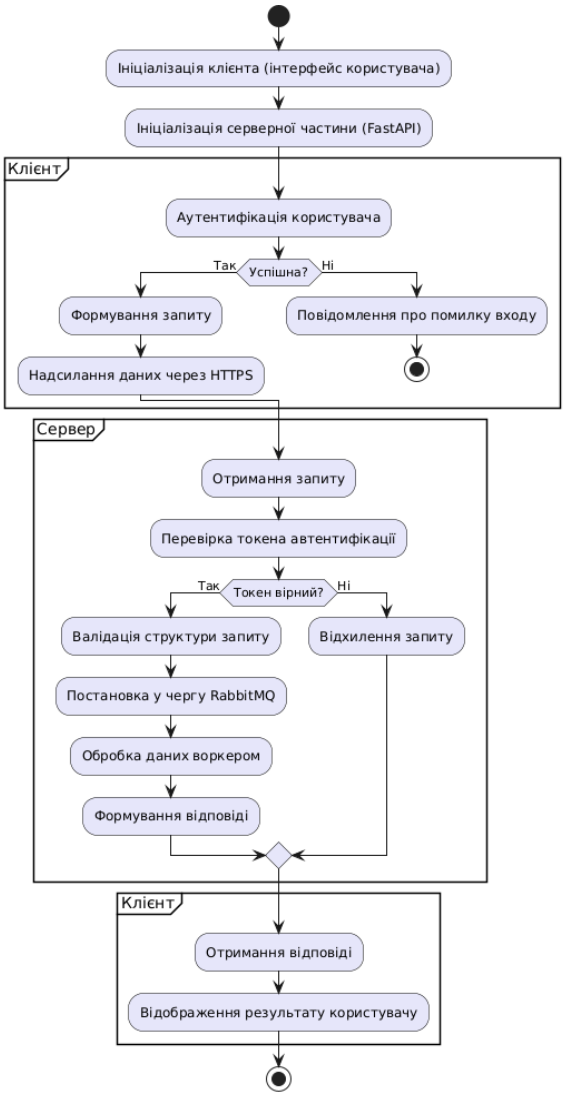

Рисунок 2.5 – Алгоритм роботи програмного застосунку

Для кращого розуміння функціонування системи передачі та обробки інформації, було розроблено деталізований алгоритм обробки запитів (рис. 2.6).

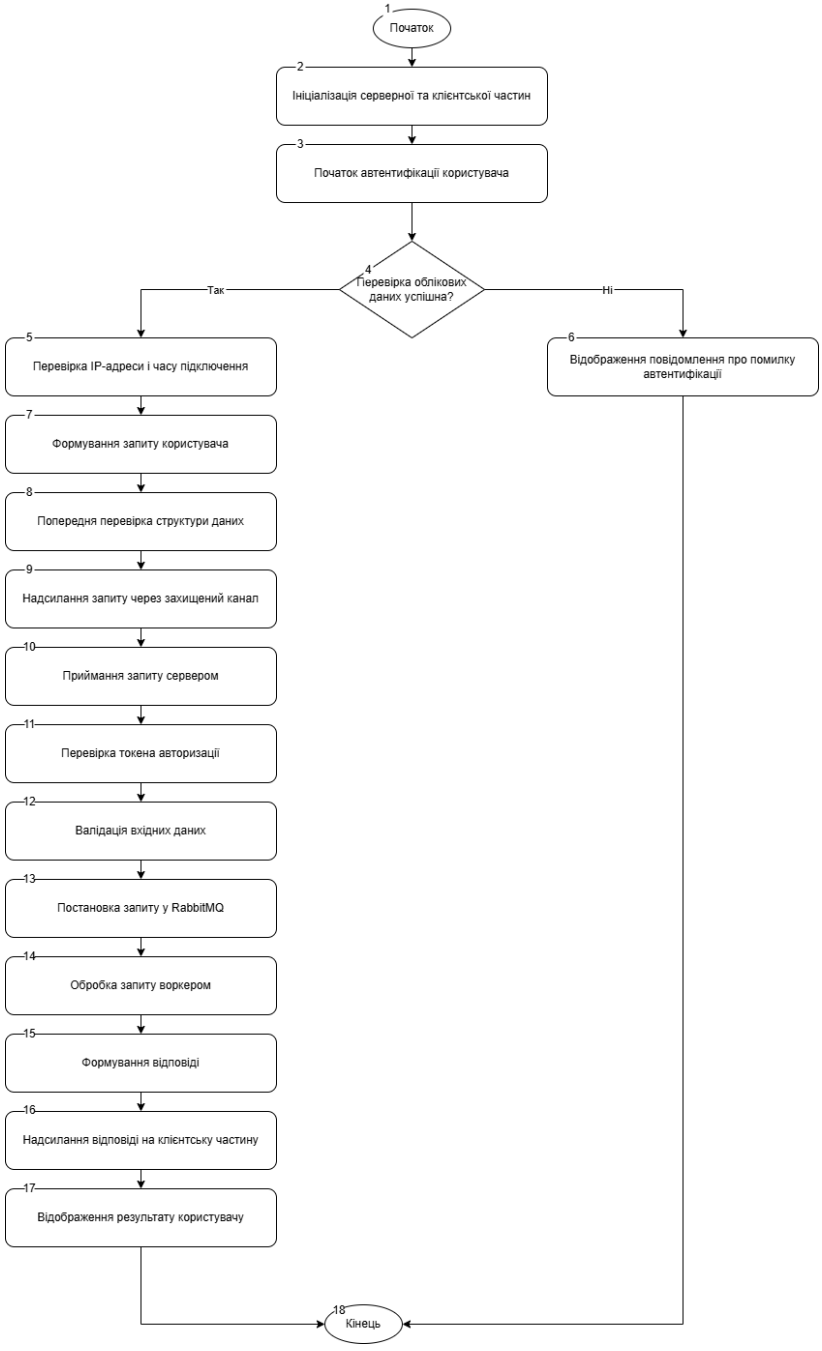

Рисунок 2.6 – Алгоритм обробки запитів у системі

На першому етапі відбувається ініціалізація як серверної, так і клієнтської частин. Після встановлення первинного з'єднання виконується процедура автентифікації. В процесі автентифікації перевіряються не лише облікові дані, але й IP-адреси користувача, час підключення, що дозволяє додатково підвищити рівень безпеки системи.

Після автентифікації клієнтська частина формує інформаційний запит, який проходить попередню перевірку структури та типу даних. Запит надсилається на сервер через захищений HTTP-канал.

На сервері запит обробляється у кілька етапів:

- Перевірка токена автентифікації;
- Валідація змісту запиту;
- Постановка запиту в асинхронну чергу RabbitMQ;
- Обробка запиту воркером;
- Формування відповіді.

Після завершення обробки даних, результат повертається на клієнтську частину. Відповідь може містити підтвердження виконаної дії, інформаційне повідомлення або статус виконання.

Система спроектована таким чином, щоб обробка великої кількості паралельних запитів не впливала на час відповіді або стабільність роботи окремих модулів.

2.5 Висновки

У другому розділі було проведено аналіз вхідних та вихідних даних програмних модулів системи обміну інформацією. Також було розглянуто процес розробки клієнтської частини із графічним інтерфейсом користувача та основні принципи взаємодії із серверною частиною. Було побудовано загальний алгоритм роботи системи, що охоплює процеси автентифікації, передачі та обробки даних на власних обчислювальних потужностях. Додатково було розроблено деталізований алгоритм обробки запитів та організації асинхронної черги повідомлень. Для кращого розуміння роботи системи було створено UML-діаграми, що ілюструють основні етапи функціонування клієнтської і серверної частин програмного забезпечення.

3. РОЗРОБКА ПРОГРАМНИХ МОДУЛІВ РЕАЛІЗАЦІЇ СИСТЕМИ ОБМІНУ ІНФОРМАЦІЄЮ ІЗ ШВИДКИМ РОЗГОРТАННЯМ НА ВЛАСНИХ ПОТУЖНОСТЯХ

3.1 Варіантний аналіз і обґрунтування вибору засобів для реалізації програмного забезпечення

Під час проєктування системи обміну інформацією з можливістю масштабування, високої продуктивності та інтеграції з клієнтськими додатками було проведено аналіз низки інструментів та технологій. Основна мета полягала в обранні таких рішень, які забезпечують ефективну взаємодію між клієнтською частиною, сервером та системою обміну повідомленнями. У якості основних технологій було обрано: Python, PyQt6, FastAPI, RabbitMQ, Docker, PostgreSQL та Redis.

Python є універсальною та потужною мовою програмування, яка ідеально підходить як для серверної, так і для клієнтської частини. Широкий вибір бібліотек та активна спільнота забезпечують швидку розробку та підтримку. Python є також природним вибором для інтеграції з системами машинного навчання, обробки даних та асинхронних сервісів.

PyQt6 було обрано як фреймворк для розробки клієнтської частини через його можливості створення кросплатформенних графічних інтерфейсів, що поєднують гнучкість, сучасний дизайн та підтримку широкого спектра віджетів. PyQt6 дозволяє будувати повноцінні GUI-додатки з високою інтерактивністю, які легко взаємодіють із бекендом через HTTP-запити або WebSocket-з’єднання.

FastAPI – це сучасний асинхронний веб-фреймворк на Python, оптимізований для створення API з високою продуктивністю. Його основними перевагами є підтримка OpenAPI, автоматична генерація документації, вбудована валідація даних через Pydantic, а також підтримка асинхронного програмування, що критично важливо для швидкої обробки запитів у реальному часі [13].

RabbitMQ виступає в ролі брокера повідомлень, який дозволяє організувати обмін повідомленнями між сервісами за принципом pub/sub або через черги. Ця технологія дозволяє досягти масштабованості, асинхронності та стійкості системи. RabbitMQ інтегрується із FastAPI через фонові задачі (наприклад, Celery або aio-pika) та дозволяє реалізовувати модулі логування, повідомлень, черг задач і т.д.

PostgreSQL було обрано як основну реляційну базу даних завдяки її стабільності, підтримці транзакцій, розширенням (таким як PostGIS, Full-Text Search) і активній підтримці спільнотою. Вона забезпечує збереження структурованих даних та дозволяє виконувати складні запити для аналітики та звітності.

Redis використовується для кешування даних та зберігання тимчасових сесій користувача. Його продуктивність при обробці великої кількості запитів робить його ідеальним рішенням для реалізації швидкого доступу до часто використовуваних даних.

Docker є невід'ємною частиною архітектури системи, оскільки дозволяє створювати ізольовані середовища для кожного компонента (сервер, клієнт, брокер повідомлень, база даних), забезпечуючи при цьому масштабованість, простоту розгортання та підтримки. У поєднанні з docker-compose забезпечується централізоване керування сервісами.

Поєднання зазначених технологій дозволяє побудувати масштабовану, модульну та легко підтримувану систему. Інтеграція FastAPI та PyQt6 забезпечує зручність взаємодії між клієнтом і сервером, RabbitMQ гарантує надійність обміну повідомленнями, а використання Docker і PostgreSQL – стабільну інфраструктуру.

Отже, запропоновані технології дозволяють реалізувати повнофункціональний програмний продукт, який відповідає вимогам сучасного програмного забезпечення – продуктивність, зручність використання, гнучкість та розширюваність.

3.2 Розробка модуля обробки та маршрутизації повідомлень

При розробці модуля обробки та маршрутизації повідомлень у системі обміну інформацією виникає необхідність врахувати низку технічних аспектів, які мають суттєвий вплив на надійність, безпеку та загальну ефективність процесу передачі даних між користувачами або серверними компонентами системи. Зокрема, одним із ключових завдань є забезпечення чіткого та послідовного формату повідомлень, який би відповідав вимогам цілісності та дозволяв оперативно ідентифікувати відправника і одержувача, а також контролювати час створення і призначення кожного повідомлення.

В першу чергу, при надходженні повідомлення до системи відбувається його початковий аналіз. Це важливий етап, оскільки кожне повідомлення має відповідати визначеному стандарту, що передбачає наявність певного набору обов'язкових атрибутів. Серед таких атрибутів є унікальний ідентифікатор відправника, який дозволяє системі достовірно ідентифікувати джерело повідомлення, ідентифікатор одержувача, що визначає кінцевого отримувача інформації, часовий штамп створення повідомлення, який фіксує момент його формування, тип повідомлення, що вказує на його призначення та формат, а також безпосередньо вміст або тіло повідомлення. Наявність цих атрибутів гарантує коректність подальшої обробки і дозволяє підтримувати узгодженість та цілісність інформації у системі, що є критично важливим для підтримання стабільної роботи в масштабних розподілених середовищах (рис. 3.1).

Наявність усіх необхідних атрибутів гарантує правильність подальшої обробки та забезпечує узгодженість даних у системі.

```python
class MessageValidator:  1 usage
    REQUIRED_FIELDS = ['sender_id', 'receiver_id', 'timestamp', 'message_type', 'content']

    @staticmethod
    def validate(message: dict) -> bool:
        for field in MessageValidator.REQUIRED_FIELDS:
            if field not in message:
                raise ValueError(f"Missing required field: {field}")
        return True
```

Рисунок 3.1 – Перелік обов'язкових атрибутів повідомлення

Після того, як було підтверджено відповідність повідомлення встановленим вимогам, відбувається наступний, не менш відповідальний крок – ідентифікація оптимального маршруту передачі або обробки цього повідомлення. Для цього була розроблена система класифікації сценаріїв маршрутизації, що враховує різні варіанти доставки, зокрема, можливість прямої адресної передачі повідомлення іншому клієнту у разі його активності в мережі, обробки на серверній стороні, якщо необхідне виконання додаткових операцій, або ж передачі до спеціалізованої черги для асинхронної обробки, що забезпечує масштабованість і балансування навантаження. Вибір маршруту визначається не лише типом повідомлення, а й поточним станом одержувача, зокрема його онлайн- або офлайн-статусом, що дозволяє підвищити ефективність доставки і мінімізувати затримки (рис. 3.2).

Вибір маршруту залежить як від типу повідомлення так і від поточного стану одержувача (онлайн чи офлайн).

```python
class RoutingManager:
    def determine_route(self, message: dict) -> str:
        if message['message_type'] == 'text' and self.is_receiver_online(message['receiver_id']):
            return 'direct_delivery'
        elif message['message_type'] == 'service':
            return 'process_on_server'
        else:
            return 'queue_for_later'

    @staticmethod  1 usage
    def is_receiver_online(receiver_id: str) -> bool:
        online_users = ['user_1', 'user_2', 'user_5']
        return receiver_id in online_users
```

Рисунок 3.2 – Ідентифікація маршруту обробки повідомлення

Наступною важливою задачею є визначення типу обробки повідомлення, що безпосередньо впливає на подальший шлях та метод обробки інформації. Для реалізації цього функціоналу був розроблений спеціальний клас MessageProcessingDefinitions, який інкапсулює логіку і правила класифікації повідомлень за типами обробки. Залежно від призначення, повідомлення можуть передаватися безпосередньо іншому користувачу в режимі реального часу, зберігатися у базі даних для подальшої доставки або архівації, реєструватися у

системних логах для аудиту та відстеження, або ж відправлятися до брокера повідомлень, такого як RabbitMQ, для обробки поза основним потоком системи.

Такий підхід забезпечує високу гнучкість і дозволяє ефективно керувати потоками інформації в різних ситуаціях. (рис. 3.3).

```python
class MessageProcessingDefinitions:
    PROCESSING_TYPES = {
        'text': 'direct_send',
        'service': 'server_process',
        'notification': 'save_to_database'
    }

    @classmethod
    def get_processing_type(cls, message_type: str) -> str:
        return cls.PROCESSING_TYPES.get(message_type, 'unknown')
```

Рисунок 3.3 – Основні типи обробки повідомлень

Після отримання всіх необхідних даних система виконує фінальну ідентифікацію кінцевого обробника повідомлення. На цьому етапі відбувається вибір конкретного механізму обробки, який максимально відповідає як структурі повідомлення, так і визначеному типу обробки. Даний процес є ключовим для підтримання оптимальної продуктивності і стабільності роботи всієї системи, особливо при обробці великої кількості повідомлень у реальному часі, що характерно для розподілених систем із високим навантаженням. В результаті модуль обробки та маршрутизації повідомлень здатен забезпечити послідовну перевірку вхідних даних, автоматизовану маршрутизацію з урахуванням різних сценаріїв і гнучке масштабування системи відповідно до росту кількості користувачів і обсягів переданої інформації (рис. 3.4).

```python
class MessageHandler:
    def handle(self, message: dict):
        processing_type = MessageProcessingDefinitions.get_processing_type(message['message_type'])

        if processing_type == 'direct_send':
            self.send_directly(message)
        elif processing_type == 'server_process':
            self.process_on_server(message)
        elif processing_type == 'save_to_database':
            self.save_to_db(message)
        else:
            raise ValueError("Unknown message processing type.")

    def send_directly(self, message: dict):  1 usage
        print(f"Sending message directly to {message['receiver_id']}.")

    def process_on_server(self, message: dict):  1 usage
        print("Processing service message on server.")

    def save_to_db(self, message: dict):  1 usage
        print("Saving notification to database.")
```

Рисунок 3.4 – Ідентифікація кінцевої обробки повідомлення

Таким чином, розроблений модуль забезпечує послідовну перевірку вхідних даних, автоматичну маршрутизацію повідомлень у залежності від їх призначення, а також гнучкість у масштабуванні системи при зростанні кількості користувачів.

3.3 Розробка модуля обробки вхідних повідомлень системи обміну інформацією

Розробка модуля обробки вхідних повідомлень є фундаментальним етапом побудови системи обміну інформацією з можливістю швидкого розгортання на власних потужностях користувача або організації. Основним завданням цього модуля є забезпечення надійного прийому, всебічної перевірки та попередньої обробки повідомлень, що надходять від різноманітних джерел, як-от кінцеві користувачі, системні служби або сторонні компоненти.

Для виконання цих функцій був створений спеціалізований клас MessageValidator, який відповідає за прийом вхідних повідомлень у форматі словникових структур і проведення їхньої валідації відповідно до заздалегідь визначених критеріїв і правил. При ініціалізації об’єкта цього класу автоматично запускається серія комплексних перевірок, які включають контроль наявності всіх обов’язкових полів, відповідність типів даних очікуваним форматам, а

також перевірку допустимості значень окремих атрибутів повідомлення. У разі виявлення будь-яких невідповідностей або помилок, повідомлення не допускається до подальшої обробки, а інформація про виниклу проблему фіксується у системі журналювання подій, що дозволяє швидко виявляти та аналізувати можливі проблеми у роботі системи. Таким чином, клас MessageValidator виконує роль першої лінії захисту, що забезпечує цілісність і коректність даних, які надходять до системи (рис. 3.5).

```python
class MessageValidator:
    def __init__(self, message: dict):
        self.message = message
        self.is_valid = False
        self.errors = []
        self.validate()

    def validate(self):  1 usage
        required_fields = ["sender_id", "receiver_id", "content", "timestamp"]
        for field in required_fields:
            if field not in self.message:
                self.errors.append(f"Missing field: {field}")

        if not isinstance(self.message.get("content", ""), str):
            self.errors.append("Content must be a string.")

        if not isinstance(self.message.get("timestamp", 0), int):
            self.errors.append("Timestamp must be an integer.")

        self.is_valid = len(self.errors) == 0
```

Рисунок 3.5 – Клас MessageValidator

Після успішного проходження валідації, повідомлення передається для подальшої обробки, що полягає у визначенні найдоцільнішого маршруту його подальшої доставки або опрацювання. Для цього був розроблений клас RoutingManager, який здійснює аналіз вмісту повідомлення і приймає рішення на основі його атрибутів та внутрішньої логіки. Головною функцією цього класу є метод determine_route, що оцінює параметри вхідного повідомлення, включно зі статусом одержувача, типом повідомлення та іншими критеріями, і визначає, чи має повідомлення бути надіслане безпосередньо адресату, оброблене на сервері, чи відкладене для подальшої доставки у разі, якщо одержувач тимчасово

недоступний. Такий підхід дозволяє підвищити ефективність обробки та забезпечує гнучкість у поводженні з повідомленнями залежно від конкретної ситуації (рис. 3.6).

```python
class RoutingManager:
    def __init__(self, user_status_provider):
        self.user_status_provider = user_status_provider

    def determine_route(self, message: dict) -> str:
        receiver_id = message.get("receiver_id")
        if self.user_status_provider.is_user_online(receiver_id):
            return "direct_send"
        else:
            return "store_for_later"
```

Рисунок 3.6 – Клас RoutingManager

Ще одним важливим елементом у побудові логіки обробки повідомлень стала розробка механізму класифікації типів повідомлень, що реалізована в допоміжному класі MessageProcessingDefinitions. Цей клас містить внутрішні константи та словники, які описують основні категорії обробки повідомлень, такі як синхронна обробка, асинхронна або відкладена. Це дозволяє уніфікувати правила взаємодії з повідомленнями різних типів, що в подальшому значно спрощує підтримку і масштабування системи. Завдяки такому механізму розробникам стає легше інтегрувати нові типи повідомлень або змінювати існуючу логіку обробки без необхідності масштабних змін у коді системи (рис. 3.7).

```python
class MessageProcessingDefinitions:
    SYNCHRONOUS = "synchronous"
    ASYNCHRONOUS = "asynchronous"
    DEFERRED = "deferred"

    PROCESSING_MAP = {
        "text": SYNCHRONOUS,
        "file": ASYNCHRONOUS,
        "system_notification": DEFERRED,
    }

    @classmethod
    def get_processing_type(cls, message_type: str) -> str:
        return cls.PROCESSING_MAP.get(message_type, cls.SYNCHRONOUS)
```

Рисунок 3.7 – Клас MessageProcessingDefinitions

Підсумовуючи, подальша обробка повідомлень здійснюється через клас MessageHandler, який інтегрує механізми валідації, класифікації та маршрутизації в єдину логіку. На основі аналізу вмісту повідомлення та визначення його типу, модуль приймає рішення щодо найбільш адекватних дій – це може бути безпосередня відправка повідомлення кінцевому користувачу, виконання додаткових операцій обробки на сервері, або збереження інформації у базі даних для подальшого використання.

Такий комплексний і продуманий підхід дозволяє побудувати гнучку та масштабовану архітектуру, яка здатна підтримувати високий рівень надійності навіть при збільшенні навантаження (рис. 3.8).

```python
class MessageHandler:
    def __init__(self, validator, router):
        self.validator = validator
        self.router = router

    def handle_message(self, message: dict):
        validator_instance = self.validator(message)

        if not validator_instance.is_valid:
            print(f"Validation failed: {validator_instance.errors}")
            return

        processing_type = MessageProcessingDefinitions.get_processing_type(
            message.get("type", "text")
        )

        route = self.router.determine_route(message)

        if route == "direct_send":
            self.send_message(message, processing_type)
        elif route == "store_for_later":
            self.store_message(message)

    def send_message(self, message: dict, processing_type: str):  1 usage
        print(f"Sending message [{processing_type}]: {message}")

    def store_message(self, message: dict):  1 usage
        print(f"Storing message for later delivery: {message}")
```

Рисунок 3.8 – Клас MessageHandler

Однією з важливих допоміжних функцій є перевірка активності одержувача повідомлення. Ця функція виконує аналіз актуального стану користувача у мережі, ґрунтуючись на інформації про його підключення до сервера, яка зберігається у кешованих структурах даних для оперативного доступу. Якщо одержувач перебуває в онлайн-режимі, повідомлення миттєво доставляється через внутрішні канали зв'язку системи. Якщо ж користувач тимчасово недоступний, повідомлення не втрачається, а передається у спеціалізований буфер або чергу, що дозволяє гарантувати доставку при повторному підключенні користувача. Це забезпечує високу надійність і цілісність інформаційних потоків. (рис. 3.9).

```python
class UserStatusProvider:
    def __init__(self):
        self.active_users = set()

    def set_user_online(self, user_id: str):
        self.active_users.add(user_id)

    def set_user_offline(self, user_id: str):
        self.active_users.discard(user_id)

    def is_user_online(self, user_id: str) -> bool:
        return user_id in self.active_users
```

Рисунок 3.9 – Перевірка активності користувача

Такий підхід забезпечує безперервність комунікації та мінімізує ризики втрати даних у розподілених середовищах.

Загалом розроблений модуль обробки вхідних повідомлень є основою для забезпечення стабільної та надійної роботи всієї системи обміну інформацією, дозволяючи зберігати високий рівень достовірності та безпеки переданих даних у різноманітних сценаріях використання.

3.4 Висновки

У третьому розділі було всебічно обґрунтовано вибір технологій, інструментів та засобів розробки, які застосовувалися при побудові програмних

модулів системи обміну інформацією. Враховуючи вимоги до функціональності, масштабованості, надійності та можливості автономного розгортання системи, було проведено детальний аналіз доступних технологічних стеків, у результаті чого сформовано оптимальну архітектуру проєкту.

На основі аналізу функціональних вимог до програмного забезпечення, а також оцінки характеру задач, які має вирішувати система, було прийнято обґрунтоване рішення щодо використання мови програмування Python як основної платформи розробки. Завдяки своїй універсальності, підтримці широкого спектра бібліотек і фреймворків, а також високій читабельності коду, Python забезпечує ефективну реалізацію як серверної, так і клієнтської частин проєкту. Для побудови графічного інтерфейсу користувача була обрана бібліотека PyQt (версія 6), яка поєднує гнучкість побудови GUI, кросплатформеність і підтримку сучасних стандартів розробки візуального інтерфейсу.

Для реалізації механізмів обміну повідомленнями між клієнтом і сервером, а також для асинхронної маршрутизації даних, були використані стандартні інструменти Python (зокрема, модулі asyncio, httpx, requests, socket) у поєднанні з додатковими бібліотеками, такими як aio-pika, Celery і kombu. Застосування цих інструментів дало змогу реалізувати ефективну модель асинхронної взаємодії з брокером повідомлень RabbitMQ, що є критично важливим для задач реального часу.

Окремий акцент було зроблено на виборі засобів валідації вхідних даних, а також на механізмах динамічного управління процесами обробки інформації. Зокрема, бібліотека Pydantic була використана для валідації схем повідомлень на основі типів даних, що дозволило запобігти помилкам на ранніх етапах взаємодії з API. Архітектура програмного комплексу передбачає також модулі маршрутизації, які відповідають за логічне визначення черги повідомлень відповідно до типу задачі, статусу користувача або наявного навантаження на систему.

У межах даного розділу було також детально висвітлено розробку ключових програмних модулів, серед яких: модулі перевірки цілісності повідомлень (через хеш-функції або цифрові підписи), логіки класифікації повідомлень (на основі типу задачі, пріоритетності або критичності), маршрутизації до відповідного сервісу обробки, а також модулі динамічного керування чергами та процесами зберігання результатів. Зокрема, зберігання здійснюється в реляційній базі даних PostgreSQL із можливістю кешування найбільш актуальних результатів у Redis, що забезпечує швидкий доступ до них при повторних зверненнях.

Таким чином, розроблені програмні засоби формують фундамент для подальшої інтеграції системи в різні середовища експлуатації – як локальні, так і хмарні – з мінімальними витратами часу на налаштування та розгортання. Застосування компонентно-орієнтованої архітектури, модульної структури та сучасних інструментів забезпечує не лише зручність масштабування й адаптації до нових вимог, але й гарантує високу продуктивність та стійкість системи до збоїв і перевантажень у процесі обробки інформації.

4 ТЕСТУВАННЯ ПРОГРАМИ

4.1 Тестування системи

Тестування програмного забезпечення – це невід'ємний етап життєвого циклу створення програмних систем, який передбачає систематичний процес виявлення помилок, дефектів і невідповідностей розробленого програмного продукту встановленим функціональним, технічним і користувацьким вимогам. Основною метою проведення тестування є забезпечення високої якості, надійності та стабільності роботи програмних модулів перед їх остаточним впровадженням в експлуатаційне середовище або переданням кінцевим користувачам.

На початковому етапі було створено тестову збірку програмного забезпечення, що містила повнофункціональні реалізації основних компонентів системи. Вона була розгорнута на робочих станціях із різними операційними системами – Windows, Linux (зокрема дистрибутиви Ubuntu та Debian), а також MacOS. Такий підхід дозволив забезпечити охоплення широкого спектра платформ і, відповідно, гарантувати кросплатформену сумісність застосунку.

Перед початком активної фази тестування було підготовлено повний пакет документації, що включав функціональні та нефункціональні вимоги до системи, тестовий план, сценарії тестування, критерії приймання та опис інструментів, які використовувались у процесі перевірки. Це забезпечило формалізацію процесу тестування та уніфікований підхід до оцінювання результатів.

Першим етапом стало функціональне тестування, метою якого було перевірити відповідність фактичної поведінки системи очікуваним результатам. Було протестовано всі ключові функції програмного комплексу: встановлення з'єднання між клієнтським інтерфейсом та сервером, обробку запитів на надсилання та отримання повідомлень, автентифікацію користувача, обробку невалідних вхідних даних та логування. Окрему увагу було приділено виявленню помилок при взаємодії з брокером повідомлень RabbitMQ, а також при кешуванні даних у Redis.

Другим етапом стало тестування стійкості до відмов, яке включало моделювання різних нештатних ситуацій, пов’язаних зі збоями в мережевому середовищі, зниженням доступності окремих сервісів або неправильними діями користувачів. У тестовому середовищі проводились експерименти з раптовим обривом з’єднання, спробами надсилання пошкоджених або неповних даних, а також некоректними HTTP-запитами до REST API. Система показала стабільність та здатність до самовідновлення у разі короткочасних збоїв.

Наступним кроком стало тестування продуктивності (performance testing), в межах якого були виміряні основні параметри ефективності системи: час реакції на запити, швидкість обробки повідомлень на серверній стороні, навантаження на центральний процесор і обсяг використаної оперативної пам’яті при обробці великих обсягів даних. Тестування проводилось із використанням автоматизованих засобів, таких як Locust, що дозволило моделювати реальні сценарії користувацької активності з високою точністю.

Окремо проводилось тестування сумісності, метою якого було переконатися в коректності функціонування програмного забезпечення в різних технічних умовах. Система тестувалась на різних версіях Windows (зокрема Windows 10 і Windows 11), дистрибутивах Linux, а також на MacOS із використанням різних апаратних конфігурацій (процесори Intel/AMD, обсяги ОЗП від 4 до 16 ГБ). Було підтверджено, що програмне забезпечення функціонує стабільно та з однаковою продуктивністю на більшості платформ.

На початковому етапі тестування була перевірена реакція системи на невірний формат даних при спробі встановлення з’єднання. Якщо користувач намагався вказати некоректні параметри конфігурації або надсилати дані у невідповідному форматі, система виявляла цю помилку та повідомляла про необхідність перевірки параметрів підключення. На рисунку 4.1 представлено інтерфейс із повідомленням про помилку налаштувань з'єднання.

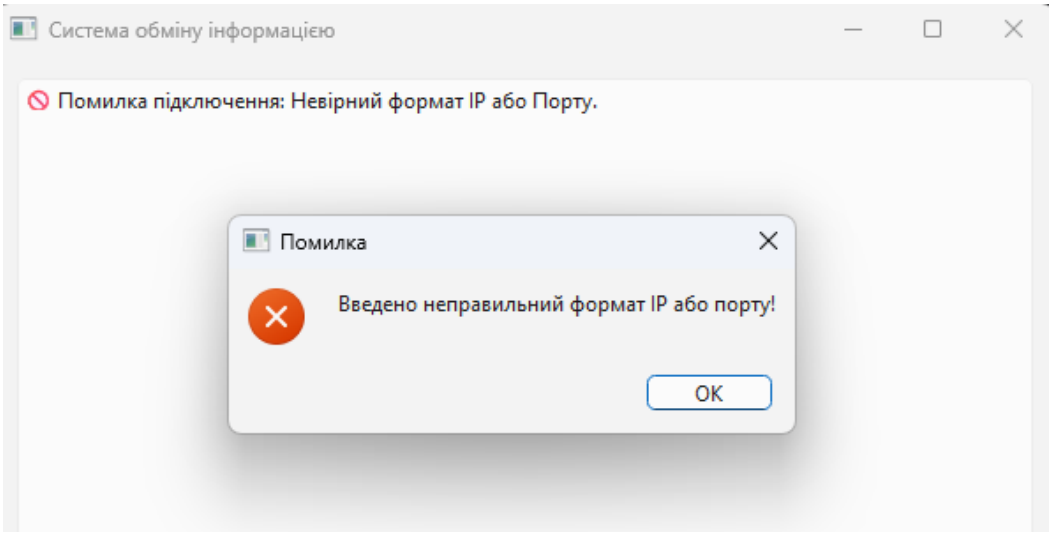

Рисунок 4.1 – Повідомлення про помилку налаштувань підключення

Після тестування помилок формату даних було проведено додаткове тестування системи на обробку ситуацій, коли користувач не вказує необхідні дані для встановлення сесії обміну. У результаті цього тесту (рисунок 4.2) встановлено, що система коректно обробляє такі випадки і виводить відповідні повідомлення із підказками для користувача.

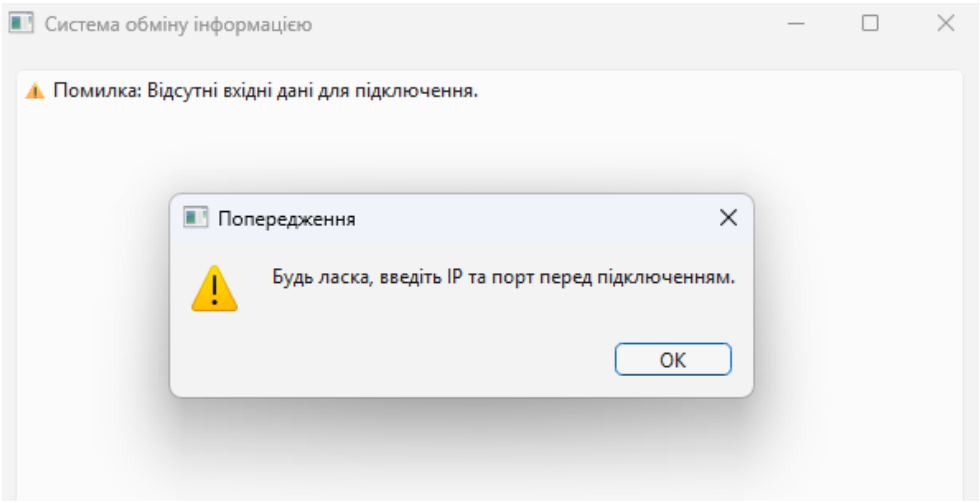

Рисунок 4.2 – Система повідомляє про відсутність вхідних даних

Наступним кроком було тестування роботи системи за умов частково пошкодженого підключення, наприклад, коли лише частина даних доходить до сервера. Для цього спеціально моделювали мережеві обриви та передавали неповні пакети даних. На рисунку 4.3 показано приклад повідомлення, яке

система формує при обробці пошкодженого з’єднання. Система змогла успішно виявити втрати даних і повідомити користувача про неможливість завершення операції.

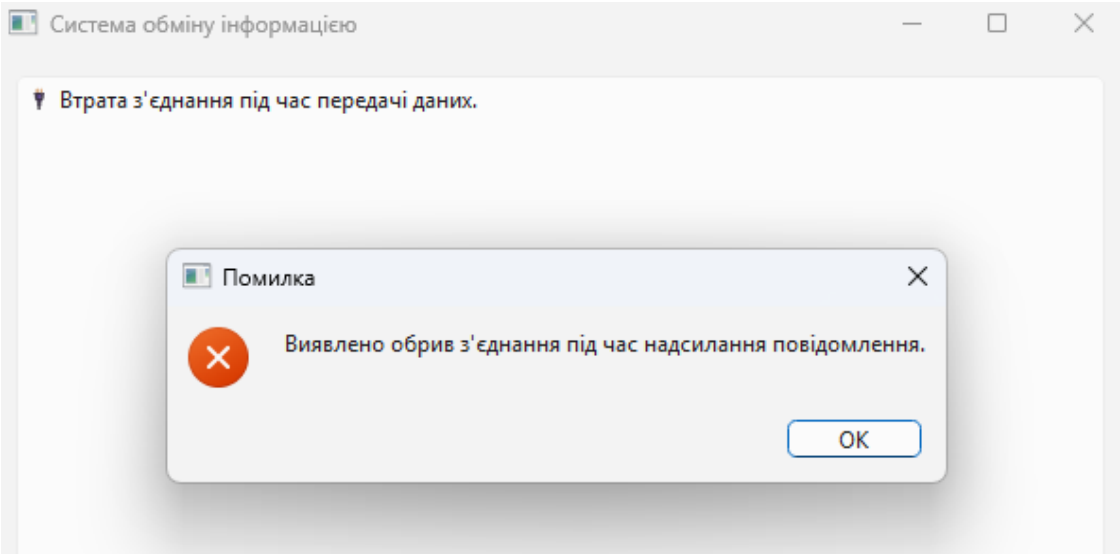

Рисунок 4.3 – Виявлення втрати даних під час обміну

Після перевірки поведінки при помилках була протестована обробка коректних запитів. Система продемонструвала стабільну роботу при обміні даними за стандартним сценарієм використання. На рисунку 4.4 показано приклад успішного встановлення сесії обміну інформацією.

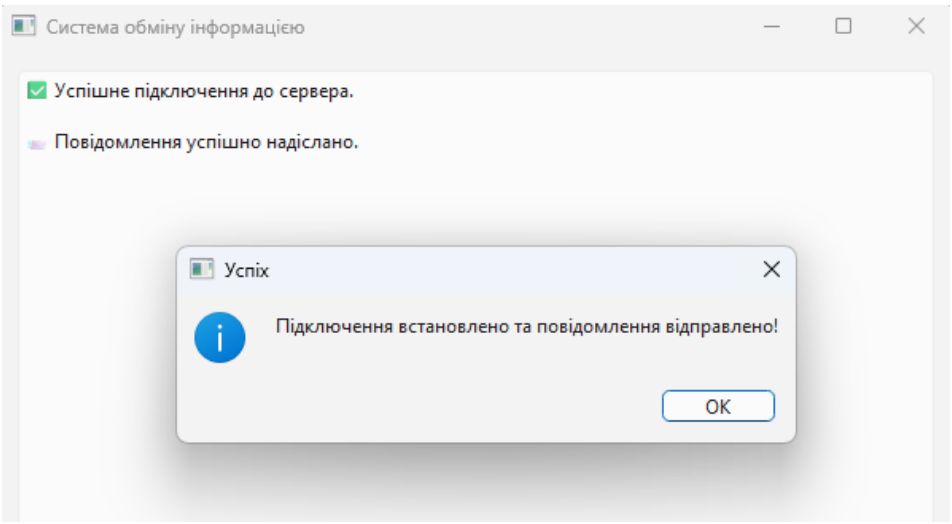

Рисунок 4.4 – Успішне встановлення з’єднання та обмін повідомленнями

Додатково проводилось тестування сумісності на різних середовищах: Windows 11, та кількох дистрибутивах Linux. Тестування проводилось на комп'ютерах з різною апаратною продуктивністю, включаючи старі ноутбуки та сучасні десктопи. Результати засвідчили, що система однаково стабільно працює як на слабших, так і на потужніших пристроях, демонструючи добру оптимізацію.

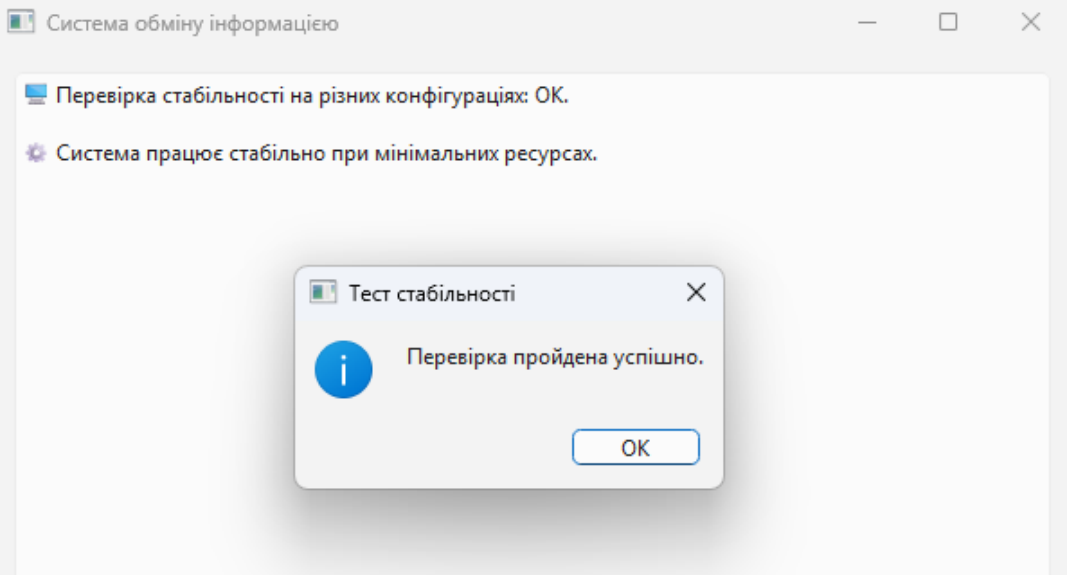

Рисунок 4.5 – Тестування на різних апаратних конфігураціях

Після завершення основного циклу перевірок, що охоплювали базову функціональність, стабільність роботи системи та її сумісність у різних середовищах, було прийнято рішення продовжити тестування у розширеному форматі. Метою цього додаткового етапу стало глибше дослідження роботи системи обміну інформацією в умовах, які наближені до реальних сценаріїв використання. Зокрема, увага була зосереджена на перевірці поведінки системи при передачі різних типів файлів, роботі в нестабільних мережевих умовах, забезпеченні високої кількості одночасних підключень, а також на здатності системи правильно реагувати на обриви підключення під час активної сесії передачі даних. Такий підхід дозволив не тільки виявити приховані вразливості, які могли залишитись непоміченими під час базового тестування, а й оцінити загальну готовність системи до реального навантаження і несподіваних ситуацій.

Одним із перших напрямів розширеного тестування стало вивчення можливостей надсилання файлів різних розмірів через систему. Було організовано серію випробувань, у межах яких надсилалися файли невеликого обсягу, середнього та великого розміру – відповідно до 1 МБ, 50 МБ та 500 МБ. При кожній передачі фіксувалися швидкість завантаження, час відповіді сервера після отримання даних, а також проводилася перевірка цілісності отриманого файлу шляхом порівняння контрольних сум (хешів). Тести показали, що при передачі малих файлів процес проходив практично миттєво, без відчутних затримок для користувача. Для середніх файлів час передачі залишався в межах двох секунд, що свідчило про добре оптимізовану логіку обробки даних. Передача великих файлів, як і очікувалось, вимагала більше часу, однак тестування засвідчило стабільність і правильність роботи потокової передачі даних: жодного випадку пошкодження чи втрати файлу зафіксовано не було (рис. 4.6).

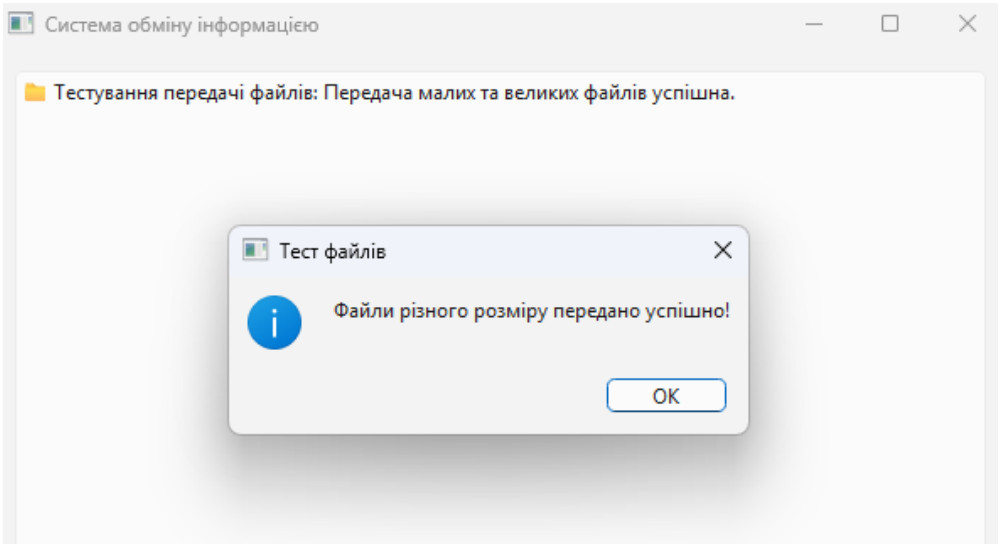

Рисунок 4.6 – Тестування надсилання файлів

Окремим етапом стало тестування можливостей встановлення з'єднання для дзвінків між користувачами, яке передбачало обмін медіа-потоками в режимі реального часу. Було змодельовано типову ситуацію ініціалізації аудіо- та відеосесії між двома клієнтами, із вимірюванням часу встановлення з'єднання, тестуванням стабільності каналу за змінної якості мережі та виявленням

помилок, пов'язаних із втратою даних під час передачі. У ході випробувань було встановлено, що створення з'єднання відбувалося надзвичайно швидко – у середньому за 3-4 секунди навіть за наявності невеликої мережевої затримки (до 100 мс). При моделюванні втрати пакетів даних механізми компенсації і відновлення функціонували коректно, дозволяючи підтримувати прийнятну якість аудіо- та відеозв'язку без помітних розривів або збоїв (рис. 4.7).

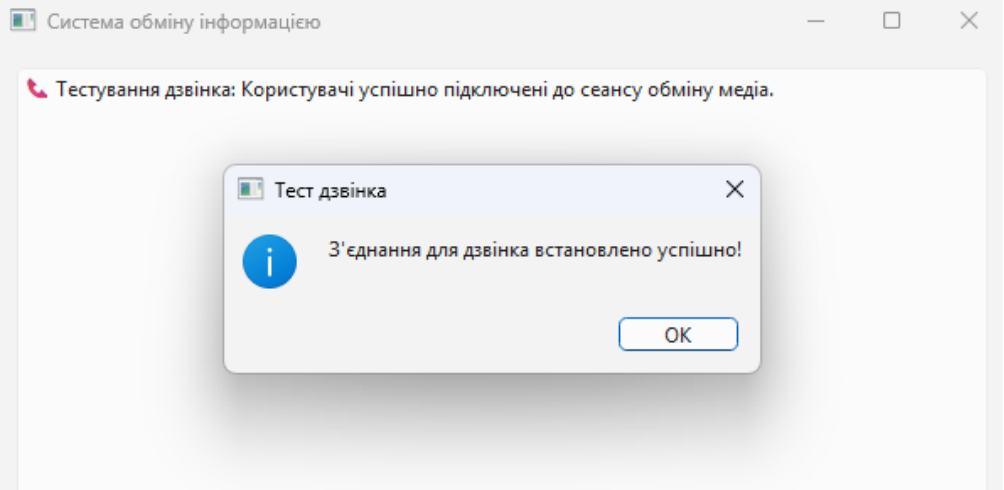

Рисунок 4.7 – Тестування встановлення з'єднання для дзвінка

З метою перевірки стійкості системи в умовах погіршення якості мережі було змодельовано підвищення затримок у мережевій інфраструктурі до 200–300 мілісекунд. У процесі тестування спостерігалося певне збільшення часу доставки повідомлень, однак критичних збоїв чи повного розриву з'єднань не зафіксовано. Механізми повторної передачі даних, реалізовані у системі, ефективно справлялися з втратою окремих пакетів, дозволяючи користувачам зберігати активні сесії обміну інформацією навіть у таких складних умовах. Це підтвердило високу стійкість архітектури програми до зовнішніх факторів, таких як зміни в пропускній здатності каналів або тимчасові мережеві збої (рис. 4.8).

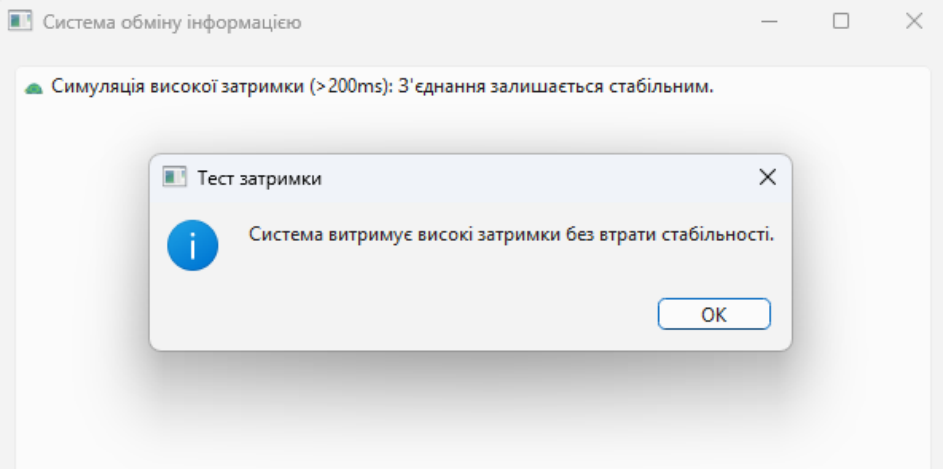

Рисунок 4.8 – Тестування стійкості до затримок в мережі

Ще однією критично важливою частиною тестування стала оцінка здатності системи обслуговувати велику кількість одночасних сесій. Для цього було організовано навантажувальне тестування, під час якого імітувалось до 500 активних клієнтів, які надсилали повідомлення з середньою частотою 10 запитів на хвилину кожен. Система витримала це навантаження впевнено: час відповіді на запити здебільшого не перевищував однієї секунди, навіть за пікових значень навантаження. Інфраструктурні компоненти – зокрема балансувальники навантаження та черги обробки запитів – успішно перерозподіляли навантаження між різними вузлами, запобігаючи утворенню «вузьких місць» і підтримуючи загальну стабільність системи (рис. 4.9).

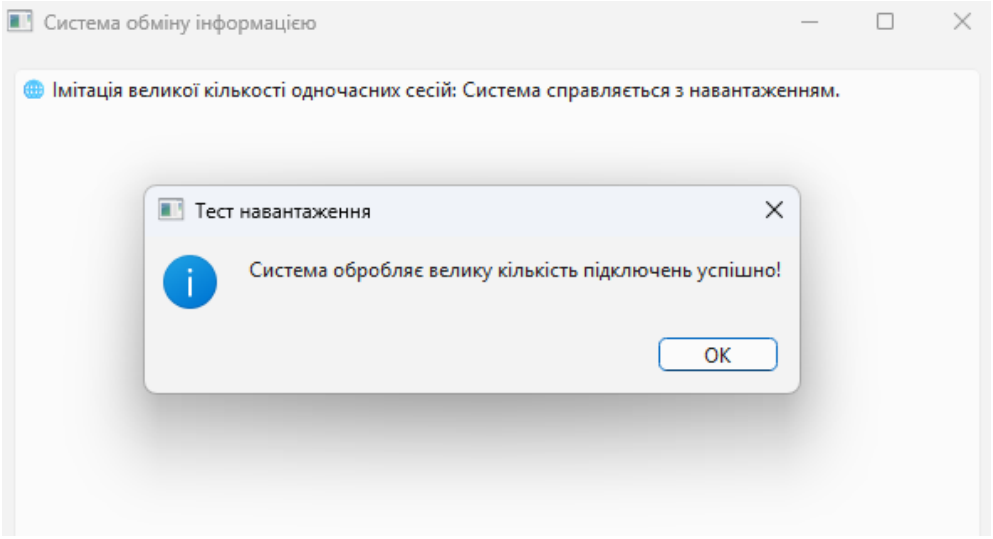

Рисунок 4.9 – Тестування обробки великої кількості одночасних сесій

Одним із найважливіших сценаріїв, який вимагав особливої уваги, була перевірка реакції системи на несподіване обривання сесії в момент активної передачі даних. У ході випробувань навмисно ініціювався обрив мережевого з'єднання під час відправки великих обсягів інформації. Система продемонструвала очікувану поведінку: вона оперативно ідентифікувала втрату зв'язку, коректно завершувала поточну сесію обміну даними та відображала користувачеві інформативне повідомлення про помилку. Після відновлення з'єднання автоматичне пересилання недоставлених даних не здійснювалося, що відповідає найкращим практикам безпеки у системах обміну інформацією, де важливо уникати неконтрольованого дублювання або втрати актуальності переданих даних (рис. 4.10).

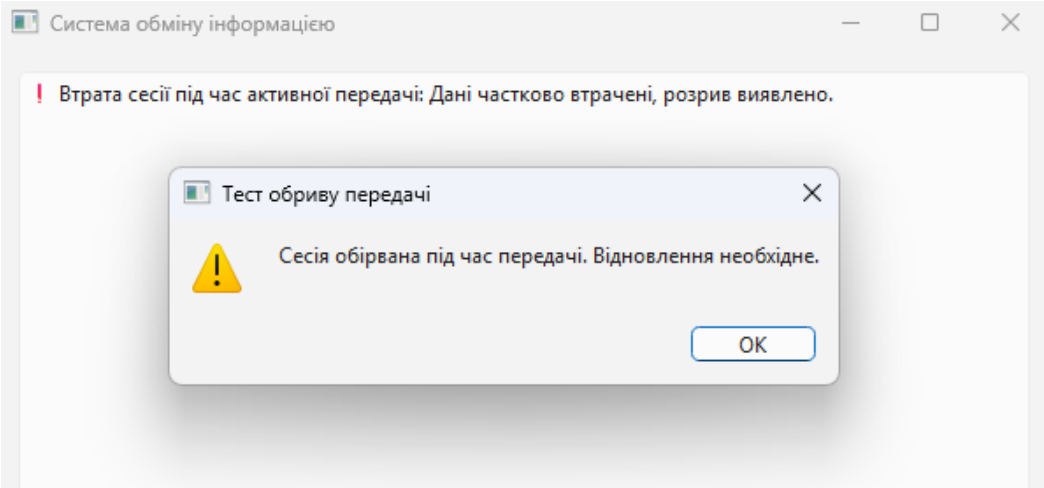

Рисунок 4.10 – Перевірка реакції на обрив сесії під час активної передачі даних

Таким чином, проведене додаткове тестування дозволило глибоко оцінити реальні можливості системи, виявити важливі нюанси її поведінки у стресових умовах та підтвердити, що розроблене рішення має високий рівень стійкості, надійності та готовності до експлуатації в складних і змінних середовищах.

4.2 Розробка інструкції користувача

Інструкція користувача передбачає визначення технічних вимог для запуску програмного продукту. Деталі щодо мінімальної та рекомендованої конфігурації розміщено в таблиці 4.1.

Таблиця 4.1 – Вимоги до апаратного забезпечення

Вимоги до конфігурування	Рекомендовано	Мінімально
Процесор	Intel Core i5	Intel Core i3
Оперативна пам'ять	8 ГБ RAM	4 ГБ RAM
Вільний простір на диску	1 ГБ	1 ГБ
Відеокарта	Вбудована або дискретна	Вбудована або дискретна
Операційна система	Windows 10 або macOS Big Sur	Windows 7 або macOS Mojave
Монітор	Роздільна здатність не менше 1920x1080 пікселів, діагональ 21» або більше	Роздільна здатність не менше 1280x720 пікселів, діагональ 15» або більше

Після встановлення та запуску програми на екрані з’являється вітальне вікно з формою авторизації. На цьому етапі користувач може або увійти до свого облікового запису, або створити новий, заповнивши дані реєстрації. При створенні облікового запису потрібно вказати унікальне ім'я користувача, надійний пароль, ім'я для відображення в чатах, а також обрати колір аватара, який візуально представлятиме профіль у списках контактів. У процесі входу користувач також вказує адресу сервера для підключення. За замовчуванням програма пропонує адресу «localhost», проте це поле можна змінити для підключення до реального віддаленого сервера.

Після введення усіх необхідних даних потрібно підтвердити дію кнопкою входу. Програма спробує встановити захищене SSL-з’єднання із сервером. У разі успішного підключення користувач переходить до основного інтерфейсу програми. Якщо ж підключення не вдасться встановити, система повідомить про проблему через діалогове вікно із вказанням можливих причин помилки.

Потрапивши до робочого середовища, користувач бачить дві основні області: панель контактів та вікно чату. Панель контактів знаходиться зліва і містить список доступних користувачів із відображенням їхнього статусу в мережі. Вибравши будь-якого контакта, користувач може розпочати обмін повідомленнями. Усі повідомлення передаються у зашифрованому вигляді, що забезпечує конфіденційність комунікацій. Програма дозволяє надсилати як текстові повідомлення, так і файли будь-якого розміру, при цьому програма автоматично обробляє розбиття великих файлів на пакети для надійнішої передачі через мережу.

Особливістю програми є виведення статусу кожного повідомлення. Після надсилання повідомлення користувач може бачити, чи було його доставлено, а також чи було воно прочитане одержувачем. Якщо в процесі обміну даними виникнуть проблеми із мережею, наприклад, втратить з’єднання сервер або локальне підключення стане недоступним, програма автоматично визначить обрив сесії, припинить передачу даних та повідомить користувача про проблему без ризику втрати інформації.

Користувач може змінювати особисті налаштування програми через розділ налаштувань, доступний через інтерфейс. Тут можна відредагувати ім’я профілю, змінити аватар та його колір, вибрати тему оформлення – світлу, темну або автоматичну відповідно до системних налаштувань. Крім того, є можливість змінювати шрифти та параметри відображення повідомлень. У розділі безпеки користувач може активувати або деактивувати наскрізне шифрування повідомлень, а також увімкнути двофакторну автентифікацію для підвищення захисту облікового запису.

Безпека обміну інформацією була поставлена в основу архітектури програми. Усі передані дані шифруються не тільки за допомогою SSL-з’єднання, але також додатково обробляються симетричним шифруванням на основі криптографічної бібліотеки. Такий підхід гарантує високий рівень захисту особистих даних навіть у випадках потенційного перехоплення трафіку.

Особлива увага приділяється надійності роботи програми при оновленнях. Рекомендується регулярно оновлювати додаток до останньої версії, щоб користуватися останніми поліпшеннями безпеки, оптимізації продуктивності та новими функціями. Інформацію про наявність нових версій можна отримати через вбудовану систему сповіщень.

У випадку втрати даних через аварійне завершення роботи програми або збій мережі програма створює резервні копії важливих сесій у спеціальних локальних файлах. Це дозволяє після перезапуску частково або повністю відновити історію спілкування, що особливо важливо у корпоративному середовищі або при роботі із критичними повідомленнями.

Поточна версія програми орієнтована на безпечний текстовий обмін та базову підтримку передачі файлів. У планах розвитку системи передбачено інтеграцію функціоналу аудіо- та відеодзвінків, розширене управління груповими чатами, автоматизацію роботи ботів-помічників, а також підтримку розширеного пошуку за історією повідомлень.

Після завершення роботи із застосунком користувач може просто закрити програму через стандартну кнопку закриття вікна. При цьому усі активні сесії будуть безпечно завершені, мережеві з’єднання коректно розірвані, а локальні ресурси очищені.

Типовими ситуаціями, які можуть виникати під час роботи із програмою, є відсутність підключення до сервера, невірно введені облікові дані або втрата мережі під час активного обміну повідомленнями. У разі помилки з’єднання програма виведе сповіщення про неможливість підключення із підказкою перевірити правильність вказаної адреси сервера та доступність Інтернету. Якщо пароль або ім’я користувача будуть введені неправильно, програма попередить про це та запропонує повторити вхід. Якщо в ході активної сесії обмін повідомленнями перерветься через обрив мережі, користувач отримає відповідне сповіщення, а після відновлення з’єднання зможе відновити роботу.

Завдяки широким можливостям налаштування, автоматичній обробці помилок, розвиненим засобам захисту і постійному оновленню, програма обміну

повідомленнями є надійним і безпечним інструментом для персонального та професійного використання.

4.3 Висновки

У четвертому розділі було здійснено комплексне тестування програмних модулів інформаційної системи обміну повідомленнями, орієнтованої на швидке розгортання у локальному або корпоративному середовищі з використанням власних обчислювальних ресурсів. Метою тестування було оцінити стійкість та надійність роботи системи в умовах типових та критичних сценаріїв її використання.

У рамках тестування було перевірено реакцію програмного забезпечення на помилкові ситуації, зокрема неправильні запити, втрату з’єднання із сервером, порушення цілісності даних, відмови брокера повідомлень RabbitMQ, а також недоступність бази даних або кешу Redis. Було зафіксовано, що програмні модулі мають належні механізми обробки винятків, логування помилок та автоматичного відновлення після збоїв.

Крім того, з метою спрощення первинного використання системи для кінцевих користувачів та адміністратора, було розроблено інструкцію користувача. Документ містить покроковий опис встановлення, ініціалізації, налаштування компонентів системи, а також рекомендації щодо підключення до серверної частини, виконання базових операцій та усунення типових проблем. Інструкція була протестована з боку незалежних користувачів, що дозволило підтвердити її практичну ефективність та зрозумілість.

ВИСНОВКИ

У рамках бакалаврської кваліфікаційної роботи було розроблено програмні модулі системи обміну інформацією із можливістю швидкого розгортання на власних потужностях. Для реалізації програмного забезпечення використовувалися мови програмування Python та бібліотека PyQt6 для створення графічного інтерфейсу, а також FastAPI для побудови серверної частини. Як брокер повідомлень була інтегрована система RabbitMQ, що дозволило реалізувати асинхронний обмін даними.

У процесі виконання роботи було здійснено аналіз сучасного стану розвитку систем обміну інформацією, проведено огляд аналогів, виявлено їх переваги та недоліки. На основі проведеного аналізу були сформульовані чіткі завдання, які стали основою для розробки власного рішення.

Під час виконання бакалаврської кваліфікаційної роботи було виконано наступне:

- Визначено вимоги до системи обміну інформацією з акцентом на автономне розгортання.
- Розроблено структуру та алгоритми взаємодії між клієнтською та серверною частинами системи.
- Створено графічний інтерфейс користувача, який забезпечує інтуїтивно зрозумілу роботу із повідомленнями та файлами.
- Реалізовано програмні модулі обробки та маршрутизації повідомлень, системи реєстрації та автентифікації користувачів.
- Проведено тестування функціональних компонентів системи та забезпечено їх відповідність поставленим технічним завданням.
- Створено діаграми діяльності та алгоритми обробки даних, що відображають ключові етапи функціонування системи.
- Розроблена детальна інструкція користувача, що охоплює встановлення програми, налаштування облікового запису та основні сценарії взаємодії з програмним забезпеченням.

Результати тестування продемонстрували повну працездатність розробленого програмного продукту, його стійкість до високих навантажень та стабільність у нестабільних мережевих умовах. Бакалаврська кваліфікаційна робота відповідає поставленим завданням, підтверджує доцільність обраних технологій та демонструє високий рівень практичної реалізації задуманої концепції.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Матеріали LIII науково-технічної конференції підрозділів Вінницького національного технічного університету [Електронний ресурс] – Режим доступу до ресурсу: https://press.vntu.edu.ua/index.php/vntu/catalog/view/832/1453/2726-1

2. Матеріали IV всеукраїнської науково – технічної конференції молодих вчених, аспірантів та студентів «Комп'ютерні ігри і мультимедіа як інноваційний підхід до комунікації - 2024» [Електронний ресурс] – Режим доступу до ресурсу: https://ontu.edu.ua/download/konfi/2024/Abstracts_Computer_games_and_multimedia_innovative_approach_electronic_communication_2024.pdf

3. Python Programming Resources [Електронний ресурс] – Режим доступу до ресурсу: https://rivery.io/blog/free-resources-learn-python/

4. PyQt6 Tutorial [Електронний ресурс] – Режим доступу до ресурсу: https://www.pythonguis.com/pyqt6-tutorial/

5. FastAPI: A Complete Beginner's Guide [Електронний ресурс] – Режим доступу до ресурсу: https://pyimagesearch.com/2025/03/17/getting-started-with-python-and-fastapi-a-complete-beginners-guide/

6. What is RabbitMQ? [Електронний ресурс] – Режим доступу до ресурсу: https://www.cloudamqp.com/blog/part1-rabbitmq-for-beginners-what-is-rabbitmq.html

7. Docker Containerize an application [Електронний ресурс] – Режим доступу до ресурсу: https://docs.docker.com/get-started/workshop/02_our_app/

8. PostgreSQL Practice Resources [Електронний ресурс] – Режим доступу до ресурсу: https://www.datacamp.com/doc/postgresql/practice-resources

9. Redis Cache Explained [Електронний ресурс] – Режим доступу до ресурсу: https://www.geeksforgeeks.org/redis-cache/

10. UML Class Diagram Tutorial [Електронний ресурс] – Режим доступу до ресурсу: https://www.lucidchart.com/pages/uml-class-diagram

11. Software Testing Methodologies Guide [Електронний ресурс] – Режим доступу до ресурсу: https://www.parasoft.com/blog/software-testing-methodologies-guide-a-high-level-overview/

12. Python Networking Programming [Електронний ресурс] – Режим доступу до ресурсу: https://www.w3schools.in/python/network-programming

13. Asyncio in Python [Електронний ресурс] – Режим доступу до ресурсу: https://www.geeksforgeeks.org/asyncio-in-python/

14. Message Queue Solutions Compared [Електронний ресурс] – Режим доступу до ресурсу: https://www.dragonflydb.io/guides/message-queues

15. Top Databases for Web Applications [Електронний ресурс] – Режим доступу до ресурсу: https://www.debutinfotech.com/blog/web-app-database-list

16. Python GUI Programming [Електронний ресурс] – Режим доступу до ресурсу: https://www.tutorialspoint.com/python/python_gui_programming.htm

17. Top Python Web Development Frameworks [Електронний ресурс] – Режим доступу до ресурсу: https://www.browserstack.com/guide/top-python-web-development-frameworks

18. Serializing JSON data in Python [Електронний ресурс] – Режим доступу до ресурсу: https://www.geeksforgeeks.org/serializing-json-data-in-python/

19. Network security fundamentals [Електронний ресурс] – Режим доступу до ресурсу: https://www.ncsc.gov.uk/guidance/network-security-fundamentals

20. Scalable Architecture for Efficient Large-Scale Applications [Електронний ресурс] – Режим доступу до ресурсу: https://www.lenovo.com/us/en/glossary/scalable-architecture/

21. Glenford J. Myers, Corey Sandler, Tom Badgett The Art of Software Testing, Нью-Йорк, США: Wiley, 2011. 256 с.

22. Mike McGrath SQL in easy steps, 3rd edition, Велика Британія: In Easy Steps, 2012. 192 с.

ДОДАТКИ

Додаток А – Технічне завдання

Міністерство освіти і науки України

Вінницький національний технічний університет

факультет інформаційних технологій та комп'ютерної інженерії

ЗАТВЕРДЖУЮ
Завідувач кафедри ПЗ
д.т.н., проф.
О. Н. Романюк
«25» 05 2025 р.

Технічне завдання
на бакалаврську кваліфікаційну роботу «Розробка системи обміну інформації із швидким розгортанням на власних потужностях для комерційного використання» за спеціальністю
121 – Інженерія програмного забезпечення

Керівник бакалаврської кваліфікаційної роботи:
к.т.н., доц. Г.Б. Ракитянська
"24" 03 2025 року.

Виконав:
студент гр. 5ПІ-21б Н.В. Бондар
"24" 03 2025 року.

Вінниця – 2025 року

1. Найменування та галузь застосування

Бакалаврська кваліфікаційна робота: «Розробка системи обміну інформації із швидким розгортанням на власних потужностях для комерційного використання».

Галузь застосування – інтернет технології.

2. Підстава для розробки.

Підставою для виконання бакалаврської кваліфікаційної роботи (БКР) є індивідуальне завдання на БКР та наказ №97 від 20 березня 2025 р. ректора ВНТУ про закріплення тем БКР.

3. Мета та призначення розробки.

Метою дослідження є розробка та оптимізація рекомендаційної системи для онлайн кінотеатрів з метою підвищення ефективності та точності рекомендаційного процесу.

Призначення роботи – розробка та реалізація програмного забезпечення рекомендаційної системи для онлайн кінотеатрів.

4. Вихідні дані для проведення НДР

Перелік основних літературних джерел, на основі яких буде виконуватись БКР:

1. Python The Big Book of Small Python Projects URL: https://inventwithpython.com/bigbookpython

2. Remo H. Jansen Learning TypeScript, Бірмінґем, Велика Британія: Packt Publishing, 2015. 368 с.

3. Brandon Kim The Next.js Handbook, Сіетл, США: Amazon Publishing, 2023. 108 с.

5. Технічні вимоги

Тип програми – онлайн платформа; модель розробки – ітеративна; засоби організації даних програми – база даних SQL; вхідні дані: відправлена інформація; вихідні дані: отримана інформація; середовище розробки – PyCharm; мова програмування – Python, фреймворк FastAPI.

6. Конструктивні вимоги

Графічна та текстова документація повинна відповідати діючим стандартам України.

7. Перелік технічної документації, що пред'являється по закінченню робіт:

1. Пояснювальна записка до БКР.

2. Технічне завдання.

3. Лістинги програми.

8. Вимоги до рівня уніфікації та стандартизації

При розробці програмних засобів слід дотримуватися уніфікації і ДСТУ.

9. Стадії та етапи розробки:

№ з/п	Назва етапів бакалаврської кваліфікаційної роботи	Термін виконання етапів роботи
1	Аналіз розвитку технологій обміну інформацією з акцентом на автономне розгортання та постановка задач дослідження	30.03.25 – 05.04.25
2	Розробка структури та алгоритмів програмного продукту	06.04.25 – 12.04.25
3	Розробка програмних модулів реалізації системи обміну інформацією із швидким розгортанням на власних потужностях	13.04.25 – 10.05.25
7	Тестування програмного застосунку	11.05.25 – 15.05.25
8	Оформлення матеріалів до захисту БКР	16.05.25 – 02.06.25

10. Порядок контролю та прийняття

Виконання етапів бакалаврської кваліфікаційної роботи контролюється керівником згідно з графіком виконання роботи.

Прийняття бакалаврської кваліфікаційної роботи здійснюється ДЕК, затвердженою зав. кафедрою згідно з графіком.

68

Додаток Б. Протокол перевірки кваліфікаційної роботи на наявність текстових запозичень

ПРОТОКОЛ ПЕРЕВІРКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Назва роботи: Розробка системи обміну інформації із швидким розгортанням на власних потужностях для комерційного використання

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

Підрозділ кафедра програмного забезпечення, ФІТКІ, 5ПІ-21Б
(кафедра, факультет, навчальна група)

Коефіцієнт подібності текстових запозичень, виявлених у роботі системою StrikePlagiarism 7.2 %

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:

_____ _____
(прізвище, ініціали, посада) (підпис)

_____ _____
(прізвище, ініціали, посада) (підпис)

Особа, відповідальна за перевірку _____ Черноволик Г. О.

З висновком експертної комісії ознайомлений(-на)

Керівник _____ к.т.н., доц. каф. ПЗ Ракитянська Г.Б.
(підпис) (прізвище, ініціали, посада)

Здобувач _____ Бондар Н. В
(підпис) (прізвище, ініціали)

Додаток В – Лістинг програми

```
import sys
import socket
import ssl
import json
import os
import time
import uuid
from datetime import datetime
from PyQt6.QtWidgets import (QApplication, QMainWindow, QWidget, QVBoxLayout,
                             QHBoxLayout, QTextEdit, QLineEdit, QPushButton,
                             QLabel, QFileDialog, QMessageBox, QListWidget,
                             QStyleFactory, QFrame, QScrollArea, QSizePolicy,
                             QDialog, QFormLayout, QListWidgetItem, QTabWidget,
                             QColorDialog, QComboBox, QCheckBox)
from PyQt6.QtCore import Qt, QThread, pyqtSignal, QTimer, QSize, QMetaObject, Q_ARG,
pyqtSlot
from PyQt6.QtGui import QFont, QPalette, QColor, QIcon
from cryptography.fernet import Fernet
import logging
from connection_manager import ConnectionManager
from message_handler import MessageHandler
from typing import Optional, Dict
import threading

# Configure logging
logging.basicConfig(
    level=logging.DEBUG,  # Changed from INFO to DEBUG
    format='%(asctime)s - %(levelname)s - [%(filename)s:%(lineno)d] - %(message)s',
    handlers=[
        logging.FileHandler('client.log'),
        logging.StreamHandler()  # This will output to terminal
    ]
)

class MessageBubble(QWidget):
    def __init__(self, text, is_sender, timestamp=None, status=None):
        super().__init__()
        self.is_sender = is_sender

        layout = QHBoxLayout()
        layout.setContentsMargins(0, 0, 0, 0)
        layout.setSpacing(8)

        # Message content
        content_layout = QVBoxLayout()
        content_layout.setSpacing(4)
        # Message bubble
        bubble = QFrame()
        bubble.setStyleSheet(f"""
```

```
            QFrame {{
                background-color: {'#007AFF' if is_sender else '#E9E9EB'};
                border-radius: 18px;
                padding: 12px 16px;
                color: {'white' if is_sender else 'black'};
                border-top-right-radius: {'4px' if is_sender else '18px'};
                border-top-left-radius: {'18px' if is_sender else '4px'};
            }}
        """)

        bubble_layout = QVBoxLayout(bubble)
        bubble_layout.setSpacing(4)
        bubble_layout.setContentsMargins(0, 0, 0, 0)

        # Message text
        text_label = QLabel(text)
        text_label.setWordWrap(True)
        text_label.setStyleSheet("""
            QLabel {
                font-size: 13px;
                padding: 0;
                font-family: -apple-system, BlinkMacSystemFont, "Segoe UI", Roboto, Helvetica, Arial,
sans-serif;
            }
        """)
        bubble_layout.addWidget(text_label)

        # Footer (timestamp and status)
        if timestamp or (status and is_sender):
            footer = QHBoxLayout()
            footer.setContentsMargins(0, 4, 0, 0)
            footer.setSpacing(4)

            if timestamp:
                time_label = QLabel(datetime.fromisoformat(timestamp).strftime('%H:%M'))
                time_label.setStyleSheet(f"""
                    QLabel {{
                        font-size: 11px;
                        color: {'rgba(255,255,255,0.7)' if is_sender else '#8E8E93'};
                        font-family: -apple-system, BlinkMacSystemFont, "Segoe UI", Roboto, Helvetica,
Arial, sans-serif;
                    }}
                """)
                footer.addWidget(time_label)

            if status and is_sender:
                status_icon = QLabel("✓" if status == "delivered" else "✓✓" if status == "read" else "")
                status_icon.setStyleSheet("""
                    QLabel {
                        font-size: 11px;
                        color: rgba(255,255,255,0.7);
                        font-family: -apple-system, BlinkMacSystemFont, "Segoe UI", Roboto, Helvetica,
Arial, sans-serif;
```

```
                }
            """)
            footer.addWidget(status_icon)

        bubble_layout.addLayout(footer)

    content_layout.addWidget(bubble)

    if is_sender:
        layout.addStretch()
        layout.addLayout(content_layout)
    else:
        layout.addLayout(content_layout)
        layout.addStretch()

    self.setLayout(layout)

class UserListItem(QWidget):
    def __init__(self, username, status, last_seen, avatar_color="#1E90FF"):
        super().__init__()
        layout = QHBoxLayout()
        layout.setContentsMargins(8, 8, 8, 8)
        layout.setSpacing(12)
        self.setLayout(layout)

        # Avatar (circle with first letter)
        avatar_label = QLabel()
        avatar_label.setFixedSize(40, 40)
        avatar_label.setStyleSheet(f"""
            QLabel {{
                background-color: {avatar_color};
                border-radius: 20px;
                color: white;
                font-weight: bold;
                font-size: 16px;
                qproperty-alignment: AlignCenter;
            }}
        """)
        avatar_label.setText(username[0].upper())
        layout.addWidget(avatar_label)

        # Info container
        info_container = QWidget()
        info_container.setStyleSheet("""
            QWidget {
                background: transparent;
            }
        """)
        info_layout = QVBoxLayout(info_container)
        info_layout.setContentsMargins(0, 0, 0, 0)
        info_layout.setSpacing(2)

        # Username
```

```
username_label = QLabel(username)
username_label.setStyleSheet("""
    QLabel {
        font-weight: bold;
        font-size: 14px;
        color: #2C3E50;
    }
""")
info_layout.addWidget(username_label)

# Status with icon
status_layout = QHBoxLayout()
status_layout.setContentsMargins(0, 0, 0, 0)
status_layout.setSpacing(4)

status_icon = QLabel("●" if status == "online" else "○")
status_icon.setStyleSheet(f"""
    QLabel {{
        color: {'#2ECC71' if status == 'online' else '#95A5A6'};
        font-size: 12px;
    }}
""")
status_layout.addWidget(status_icon)

status_text = "Online" if status == "online" else "Offline"
if status == "offline" and last_seen:
    try:
        last_seen_dt = datetime.fromtimestamp(last_seen)
        now = datetime.now()
        delta = now - last_seen_dt

        if delta.days == 0:
            if delta.seconds < 3600:  # Less than an hour
                minutes = delta.seconds // 60
                status_text = f"Last seen {minutes} min ago"
            else:
                status_text = f"Last seen at {last_seen_dt.strftime('%H:%M')}"
        elif delta.days == 1:
            status_text = "Last seen yesterday"
        else:
            status_text = f"Last seen {last_seen_dt.strftime('%d %b')}"
    except:
        status_text = "Last seen recently"

status_label = QLabel(status_text)
status_label.setStyleSheet(f"""
    QLabel {{
        color: {'#2ECC71' if status == 'online' else '#95A5A6'};
        font-size: 12px;
    }}
""")
status_layout.addWidget(status_label)
status_layout.addStretch()
```

```
        info_layout.addLayout(status_layout)
        layout.addWidget(info_container)
        layout.addStretch()

        # Add hover effect
        self.setStyleSheet("""
            QWidget {
                background-color: transparent;
                border-radius: 6px;
            }
            QWidget:hover {
                background-color: #F0F8FF;
            }
        """)

class ChatHeader(QWidget):
    def __init__(self, parent=None):
        super().__init__(parent)
        layout = QHBoxLayout()
        layout.setContentsMargins(16, 8, 16, 8)

        # User info
        self.user_info = QVBoxLayout()
        self.user_info.setSpacing(2)

        self.username_label = QLabel("Select a user to chat")
        self.username_label.setStyleSheet("""
            font-weight: bold;
            font-size: 16px;
            color: #2C3E50;
        """)
        self.user_info.addWidget(self.username_label)

        self.status_label = QLabel("")
        self.status_label.setStyleSheet("""
            color: #7F8C8D;
            font-size: 12px;
        """)
        self.user_info.addWidget(self.status_label)

        layout.addLayout(self.user_info)
        layout.addStretch()

        self.setLayout(layout)
        self.setStyleSheet("""
            ChatHeader {
                background-color: #FFFFFF;
                border-bottom: 2px solid #E8EEF2;
            }
        """)

    def set_user(self, username, status=None):
        self.username_label.setText(username)
```

```
        if status:
            self.status_label.setText(status)
            self.status_label.setVisible(True)
        else:
            self.status_label.setVisible(False)

class TypingIndicator(QWidget):
    def __init__(self, parent=None):
        super().__init__(parent)
        layout = QHBoxLayout()
        layout.setContentsMargins(16, 4, 16, 4)

        self.label = QLabel("typing...")
        self.label.setStyleSheet("""
            QLabel {
                color: #666666;
                font-size: 12px;
                font-style: italic;
            }
        """)
        layout.addWidget(self.label)
        layout.addStretch()

        self.setLayout(layout)
        self.hide()

    def show_typing(self, username):
        self.label.setText(f"{username} is typing...")
        self.show()

    def hide_typing(self):
        self.hide()

class ChatArea(QScrollArea):
    def __init__(self):
        super().__init__()
        self.setWidgetResizable(True)
        self.setHorizontalScrollBarPolicy(Qt.ScrollBarPolicy.ScrollBarAlwaysOff)
        self.setStyleSheet("""
            QScrollArea {
                border: none;
                background-color: #F2F2F7;
            }
            QScrollBar:vertical {
                border: none;
                background-color: #F2F2F7;
                width: 8px;
                margin: 0;
            }
            QScrollBar::handle:vertical {
                background-color: #C6C6C8;
                border-radius: 4px;
                min-height: 20px;
```

```
            }
            QScrollBar::add-line:vertical, QScrollBar::sub-line:vertical {
                height: 0;
            }
        """)

        # Content widget
        self.content = QWidget()
        self.setWidget(self.content)

        # Layout
        self.layout = QVBoxLayout(self.content)
        self.layout.setContentsMargins(16, 16, 16, 16)
        self.layout.setSpacing(8)
        self.layout.addStretch()

    def add_message(self, text, is_sender, timestamp=None, status=None):
        bubble = MessageBubble(text, is_sender, timestamp, status)
        self.layout.insertWidget(self.layout.count() - 1, bubble)
        self.verticalScrollBar().setValue(self.verticalScrollBar().maximum())

    def clear(self):
        while self.layout.count() > 1:  # Keep the stretch
            item = self.layout.takeAt(0)
            if item.widget():
                item.widget().deleteLater()

class MessageHandler(QThread):
    message_received = pyqtSignal(dict)
    connection_lost = pyqtSignal()

    def __init__(self, connection_manager):
        super().__init__()
        self.connection_manager = connection_manager
        self.running = True
        self.ui_updates = []
        self.update_timer = QTimer()
        self.update_timer.timeout.connect(self.process_ui_updates)
        self.update_timer.start(50)  # Process UI updates every 50ms

    def run(self):
        """Run message handling loop"""
        while self.running:
            try:
                # Get message from connection manager
                message = self.connection_manager.receive_message()
                if message:
                    # Queue UI updates instead of emitting directly
                    self.queue_ui_update(message)
                else:
                    self.connection_lost.emit()
                    break
            except Exception as e:
```

```
                logging.error(f"Error in message handler: {e}")
                self.connection_lost.emit()
                break

    def queue_ui_update(self, message):
        """Queue a UI update to be processed in the main thread"""
        self.ui_updates.append(message)

    def process_ui_updates(self):
        """Process queued UI updates in the main thread"""
        if not self.ui_updates:
            return

        # Process up to 10 updates at a time
        batch_size = min(10, len(self.ui_updates))
        for _ in range(batch_size):
            message = self.ui_updates.pop(0)
            self.message_received.emit(message)

    def stop(self):
        """Stop message handling"""
        self.running = False
        self.update_timer.stop()

class LoginDialog(QDialog):
    def __init__(self, parent=None):
        super().__init__(parent)
        self.setWindowTitle("Secure Messenger - Login")
        self.setFixedWidth(400)
        self.setup_ui()

    def setup_ui(self):
        layout = QVBoxLayout()
        self.setLayout(layout)

        # Tabs for login and registration
        tabs = QTabWidget()
        login_tab = QWidget()
        register_tab = QWidget()
        tabs.addTab(login_tab, "Login")
        tabs.addTab(register_tab, "Register")

        # Login tab
        login_layout = QFormLayout()
        login_tab.setLayout(login_layout)

        self.login_username = QLineEdit()
        self.login_username.setPlaceholderText("Enter username")
        self.login_password = QLineEdit()
        self.login_password.setPlaceholderText("Enter password")
        self.login_password.setEchoMode(QLineEdit.EchoMode.Password)
        self.login_server = QLineEdit()
        self.login_server.setPlaceholderText("Enter server address")
```

```
self.login_server.setText("localhost")

login_layout.addRow("Username:", self.login_username)
login_layout.addRow("Password:", self.login_password)
login_layout.addRow("Server:", self.login_server)

login_btn = QPushButton("Login")
login_btn.setStyleSheet("""
   QPushButton {
      background-color: #1E90FF;
      color: white;
      border: none;
      border-radius: 4px;
      padding: 8px;
      font-weight: bold;
   }
   QPushButton:hover {
      background-color: #1873CC;
   }
""")
login_btn.clicked.connect(self.accept_login)
login_layout.addRow("", login_btn)

# Register tab
register_layout = QFormLayout()
register_tab.setLayout(register_layout)

self.reg_username = QLineEdit()
self.reg_username.setPlaceholderText("Choose username")
self.reg_password = QLineEdit()
self.reg_password.setPlaceholderText("Choose password")
self.reg_password.setEchoMode(QLineEdit.EchoMode.Password)
self.reg_confirm_password = QLineEdit()
self.reg_confirm_password.setPlaceholderText("Confirm password")
self.reg_confirm_password.setEchoMode(QLineEdit.EchoMode.Password)
self.reg_display_name = QLineEdit()
self.reg_display_name.setPlaceholderText("Your display name")
self.reg_avatar_color = QPushButton("Choose Avatar Color")
self.reg_avatar_color.setStyleSheet("background-color: #1E90FF; color: white;")
self.reg_avatar_color.clicked.connect(self.choose_color)
self.reg_server = QLineEdit()
self.reg_server.setPlaceholderText("Enter server address")
self.reg_server.setText("localhost")

register_layout.addRow("Username:", self.reg_username)
register_layout.addRow("Password:", self.reg_password)
register_layout.addRow("Confirm Password:", self.reg_confirm_password)
register_layout.addRow("Display Name:", self.reg_display_name)
register_layout.addRow("Avatar Color:", self.reg_avatar_color)
register_layout.addRow("Server:", self.reg_server)

register_btn = QPushButton("Register")
register_btn.setStyleSheet("""
```

```
            QPushButton {
                background-color: #2ECC71;
                color: white;
                border: none;
                border-radius: 4px;
                padding: 8px;
                font-weight: bold;
            }
            QPushButton:hover {
                background-color: #27AE60;
            }
        """)
        register_btn.clicked.connect(self.accept_register)
        register_layout.addRow("", register_btn)

        layout.addWidget(tabs)

        # Error label
        self.error_label = QLabel()
        self.error_label.setStyleSheet("color: #E74C3C; font-weight: bold;")
        self.error_label.hide()
        layout.addWidget(self.error_label)

    def choose_color(self):
        color = QColorDialog.getColor()
        if color.isValid():
            self.reg_avatar_color.setStyleSheet(f"background-color: {color.name()}; color: white;")

    def accept_login(self):
        if not self.login_username.text() or not self.login_password.text():
            self.show_error("Please fill in all fields")
            return

        # Validate server address
        server = self.login_server.text().strip()
        if not server:
            self.show_error("Please enter a valid server address")
            return

        self.mode = 'login'
        self.accept()

    def accept_register(self):
        if not all([self.reg_username.text(), self.reg_password.text(),
                    self.reg_confirm_password.text(), self.reg_display_name.text()]):
            self.show_error("Please fill in all fields")
            return

        if self.reg_password.text() != self.reg_confirm_password.text():
            self.show_error("Passwords do not match")
            return

        # Validate username
```

```
        username = self.reg_username.text().strip()
        if not username.isalnum():
            self.show_error("Username can only contain letters and numbers")
            return

        # Validate password
        password = self.reg_password.text()
        if len(password) < 6:
            self.show_error("Password must be at least 6 characters long")
            return

        # Validate server address
        server = self.reg_server.text().strip()
        if not server:
            self.show_error("Please enter a valid server address")
            return

        self.mode = 'register'
        self.accept()

    def show_error(self, message):
        self.error_label.setText(message)
        self.error_label.show()

    def get_credentials(self):
        if self.mode == 'login':
            return {
                'mode': 'login',
                'username': self.login_username.text().strip(),
                'password': self.login_password.text(),
                'server': self.login_server.text().strip()
            }
        elif self.mode == 'register':
            return {
                'mode': 'register',
                'username': self.reg_username.text().strip(),
                'password': self.reg_password.text(),
                'display_name': self.reg_display_name.text().strip(),
                'avatar_color': self.reg_avatar_color.styleSheet().split('background-color: ')[1].split(';')[0],
                'server': self.reg_server.text().strip()
            }
        return None

class UserProfile:
    def __init__(self, username, display_name=None, avatar_color=None):
        self.username = username
        self.display_name = display_name or username
        self.avatar_color = avatar_color or "#1E90FF"
        self.status = "offline"
        self.last_seen = None

    def to_dict(self):
        return {
```

```
            'username': self.username,
            'display_name': self.display_name,
            'avatar_color': self.avatar_color,
            'status': self.status,
            'last_seen': self.last_seen
        }

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.connection_manager = None
        self.message_handler = None
        self.connection_thread = None
        self.connected = False
        self.username = None
        self.password = None
        self.server_host = 'localhost'
        self.registration_data = None

        # Initialize UI
        self.setup_ui()

    def cleanup_connection(self):
        """Clean up connection resources"""
        if self.message_handler:
            self.message_handler.stop()
            self.message_handler.wait()
            self.message_handler = None

        if self.connection_manager:
            self.connection_manager.cleanup()
            self.connection_manager = None

        if self.connection_thread:
            self.connection_thread.quit()
            self.connection_thread.wait()
            self.connection_thread = None

        self.connected = False
        self.setEnabled(True)
        if hasattr(self, 'message_input'):
        self.message_input.setEnabled(False)
        if hasattr(self, 'send_button'):
        self.send_button.setEnabled(False)

    def closeEvent(self, event):
        """Handle window close event"""
        self.cleanup_connection()
        event.accept()

    def show_login_dialog(self):
        """Show login dialog and get credentials"""
        dialog = LoginDialog(self)
```

```
        result = dialog.exec()

        if result == QDialog.DialogCode.Accepted:
            credentials = dialog.get_credentials()
            if credentials:
                self.username = credentials['username']
                self.password = credentials['password']
                self.server_host = credentials['server']

                # Store registration data if in register mode
                if credentials['mode'] == 'register':
                    self.registration_data = {
                        'display_name': credentials['display_name'],
                        'avatar_color': credentials['avatar_color']
                    }

                return True
        return False

    def connect_to_server(self):
        """Connect to the server"""
        try:
            logging.info(f"Attempting to connect to {self.server_host}:{5000}")

            # Create connection manager with config
        config = {
                'server_host': self.server_host,
                'server_port': 5000,
                'username': self.username,
                'password': self.password
            }
        self.connection_manager = ConnectionManager(config)
        # Connect to server
        if not self.connection_manager.connect(self.server_host, 5000, self.username, self.password):
            self.show_error("Failed to establish connection to server. Please check if the server is
running.")
            return False

        logging.info("Connection established, starting message handler")

            # Start message handler thread
            self.message_handler_thread = threading.Thread(target=self._handle_messages,
daemon=True)
            self.message_handler_thread.start()
            return True
            pt Exception as e:
            logging.error(f"Error connecting to server: {e}")
            self.show_error(f"Failed to connect to server: {str(e)}")
            return False
    def _handle_messages(self):
        """Handle incoming messages from the server"""
        try:
            while True:
```

```
                message = self.connection_manager.receive_message()
                if not message:
                    continue

                if message.get('type') == 'user_list':
                    self._handle_user_list(message.get('users', []))
                elif message.get('type') == 'text':
                    self._handle_text_message(message)
                elif message.get('type') == 'error':
                    self.show_error(message.get('content', 'Unknown error'))

            except Exception as e:
                logging.error(f"Error in message handler: {e}")
                self.connection_manager.cleanup()

        def _handle_user_list(self, users):
            # Implement user list handling logic
            pass

        def _handle_text_message(self, message):
            # Implement text message handling logic
            pass

        def show_error(self, message):
            # Implement error handling logic
            pass

        def setup_ui(self):
            self.setWindowTitle("Secure Messenger")
            self.setMinimumSize(1000, 600)
            self.setStyleSheet("""
                QMainWindow {
                    background-color: #ffffff;
                }
                QWidget {
                    font-family: -apple-system, BlinkMacSystemFont, "Segoe UI", Roboto, Helvetica, Arial,
sans-serif;
                }
                QPushButton {
                    background-color: #007AFF;
                    color: white;
                    border: none;
                    padding: 8px 16px;
                    border-radius: 4px;
                    font-size: 13px;
                }
                QPushButton:hover {
                    background-color: #0062cc;
                }
                QLineEdit {
                    padding: 8px;
                    border: 1px solid #d1d1d1;
                    border-radius: 4px;
```

```
            background-color: #ffffff;
            font-size: 13px;
        }
        QListWidget {
            border: none;
            background-color: #f0f0f0;
        }
        QListWidget::item {
            padding: 8px;
            border-bottom: 1px solid #e0e0e0;
        }
        QListWidget::item:selected {
            background-color: #e0e0e0;
        }
        QScrollBar:vertical {
            border: none;
            background-color: #f0f0f0;
            width: 8px;
            margin: 0px;
        }
        QScrollBar::handle:vertical {
            background-color: #c0c0c0;
            min-height: 20px;
            border-radius: 4px;
        }
        QScrollBar::add-line:vertical, QScrollBar::sub-line:vertical {
            height: 0px;
        }
    """)

    # Create central widget and layout
    central_widget = QWidget()
    self.setCentralWidget(central_widget)
    layout = QHBoxLayout(central_widget)
    layout.setSpacing(0)
    layout.setContentsMargins(0, 0, 0, 0)

    # Create sidebar
    sidebar = QWidget()
    sidebar.setFixedWidth(300)
    sidebar.setStyleSheet("background-color: #f0f0f0;")
    sidebar_layout = QVBoxLayout(sidebar)
    sidebar_layout.setSpacing(0)
    sidebar_layout.setContentsMargins(0, 0, 0, 0)

    # Create header
    header = QWidget()
    header.setFixedHeight(60)
    header.setStyleSheet("background-color: #ffffff; border-bottom: 1px solid #e0e0e0;")
    header_layout = QHBoxLayout(header)
    header_layout.setContentsMargins(16, 0, 16, 0)

    # Add settings button
```

```
settings_button = QPushButton("Settings")
settings_button.clicked.connect(self.show_settings)
header_layout.addWidget(settings_button)

# Add search box
search_box = QLineEdit()
search_box.setPlaceholderText("Search conversations...")
search_box.setStyleSheet("""
    QLineEdit {
        padding: 8px;
        border: 1px solid #d1d1d1;
        border-radius: 4px;
        background-color: #f0f0f0;
    }
""")
header_layout.addWidget(search_box)

sidebar_layout.addWidget(header)

# Add user list
self.user_list = QListWidget()
self.user_list.setStyleSheet("""
    QListWidget {
        border: none;
        background-color: #f0f0f0;
    }
    QListWidget::item {
        padding: 12px;
        border-bottom: 1px solid #e0e0e0;
    }
    QListWidget::item:selected {
        background-color: #e0e0e0;
    }
""")
sidebar_layout.addWidget(self.user_list)

layout.addWidget(sidebar)

# Create chat area
chat_area = QWidget()
chat_area.setStyleSheet("background-color: #ffffff;")
chat_layout = QVBoxLayout(chat_area)
chat_layout.setSpacing(0)
chat_layout.setContentsMargins(0, 0, 0, 0)

# Add chat header
chat_header = QWidget()
chat_header.setFixedHeight(60)
chat_header.setStyleSheet("background-color: #ffffff; border-bottom: 1px solid #e0e0e0;")
chat_header_layout = QHBoxLayout(chat_header)
chat_header_layout.setContentsMargins(16, 0, 16, 0)

self.chat_title = QLabel("Select a conversation")
```

```
        self.chat_title.setStyleSheet("font-size: 16px; font-weight: bold;")
        chat_header_layout.addWidget(self.chat_title)

        chat_layout.addWidget(chat_header)

        # Add message area
        self.message_area = QScrollArea()
        self.message_area.setWidgetResizable(True)
        self.message_area.setStyleSheet("""
            QScrollArea {
                border: none;
                background-color: #ffffff;
            }
        """)

        message_widget = QWidget()
        self.message_layout = QVBoxLayout(message_widget)
        self.message_layout.setSpacing(8)
        self.message_layout.setContentsMargins(16, 16, 16, 16)
        self.message_layout.addStretch()

        self.message_area.setWidget(message_widget)
        chat_layout.addWidget(self.message_area)

        # Add input area
        input_area = QWidget()
        input_area.setFixedHeight(100)
        input_area.setStyleSheet("background-color: #ffffff; border-top: 1px solid #e0e0e0;")
        input_layout = QVBoxLayout(input_area)
        input_layout.setContentsMargins(16, 8, 16, 8)

        self.message_input = QLineEdit()
        self.message_input.setPlaceholderText("Type a message...")
        self.message_input.returnPressed.connect(self.send_message)
        self.message_input.setEnabled(False)
        input_layout.addWidget(self.message_input)

        self.send_button = QPushButton("Send")
        self.send_button.clicked.connect(self.send_message)
        self.send_button.setEnabled(False)
        input_layout.addWidget(self.send_button)

        chat_layout.addWidget(input_area)

        layout.addWidget(chat_area)

        # Initialize settings window
        self.settings_window = None

    def show_settings(self):
        if not self.settings_window:
            self.settings_window = SettingsWindow(self)
        self.settings_window.show()
```

```
    def send_message(self):
        # TODO: Implement message sending
        pass

class SettingsWindow(QMainWindow):
    def __init__(self, parent=None):
        super().__init__(parent)
        self.setWindowTitle("Settings")
        self.setMinimumSize(800, 600)
        self.setStyleSheet("""
            QMainWindow {
                background-color: #ffffff;
            }
            QTabWidget::pane {
                border: none;
                background-color: #ffffff;
            }
            QTabBar::tab {
                background-color: #f0f0f0;
                border: none;
                padding: 8px 16px;
                margin-right: 2px;
            }
            QTabBar::tab:selected {
                background-color: #ffffff;
                border-bottom: 2px solid #007AFF;
            }
            QLabel {
                color: #000000;
                font-size: 13px;
            }
            QLineEdit, QComboBox {
                padding: 8px;
                border: 1px solid #d1d1d1;
                border-radius: 4px;
                background-color: #ffffff;
                font-size: 13px;
            }
            QPushButton {
                background-color: #007AFF;
                color: white;
                border: none;
                padding: 8px 16px;
                border-radius: 4px;
                font-size: 13px;
            }
            QPushButton:hover {
                background-color: #0062cc;
            }
        """)

        # Create central widget and layout
```

```
    central_widget = QWidget()
    self.setCentralWidget(central_widget)
    layout = QVBoxLayout(central_widget)

    # Create tab widget
    tab_widget = QTabWidget()
    layout.addWidget(tab_widget)

    # Add tabs
    tab_widget.addTab(self.create_profile_tab(), "Profile")
    tab_widget.addTab(self.create_appearance_tab(), "Appearance")
    tab_widget.addTab(self.create_notifications_tab(), "Notifications")
    tab_widget.addTab(self.create_security_tab(), "Security")

def create_profile_tab(self):
    tab = QWidget()
    layout = QVBoxLayout(tab)
    layout.setSpacing(20)
    layout.setContentsMargins(20, 20, 20, 20)

    # Profile picture
    profile_pic_layout = QHBoxLayout()
    profile_pic = QLabel()
    profile_pic.setFixedSize(100, 100)
    profile_pic.setStyleSheet("""
        QLabel {
            border-radius: 50px;
            background-color: #1E90FF;
        }
    """)
    profile_pic_layout.addWidget(profile_pic)
    profile_pic_layout.addStretch()
    layout.addLayout(profile_pic_layout)

    # Display name
    display_name_layout = QHBoxLayout()
    display_name_label = QLabel("Display Name:")
    display_name_edit = QLineEdit()
    display_name_edit.setPlaceholderText("Enter your display name")
    display_name_layout.addWidget(display_name_label)
    display_name_layout.addWidget(display_name_edit)
    layout.addLayout(display_name_layout)

    # Status
    status_layout = QHBoxLayout()
    status_label = QLabel("Status:")
    status_edit = QLineEdit()
    status_edit.setPlaceholderText("Enter your status")
    status_layout.addWidget(status_label)
    status_layout.addWidget(status_edit)
    layout.addLayout(status_layout)

    # Save button
```

```
        save_button = QPushButton("Save Changes")
        save_button.clicked.connect(self.save_profile_changes)
        layout.addWidget(save_button)

        layout.addStretch()
        return tab

    def create_appearance_tab(self):
        tab = QWidget()
        layout = QVBoxLayout(tab)
        layout.setSpacing(20)
        layout.setContentsMargins(20, 20, 20, 20)

        # Theme
        theme_layout = QHBoxLayout()
        theme_label = QLabel("Theme:")
        theme_combo = QComboBox()
        theme_combo.addItems(["Light", "Dark", "System"])
        theme_layout.addWidget(theme_label)
        theme_layout.addWidget(theme_combo)
        layout.addLayout(theme_layout)

        # Message bubble style
        bubble_layout = QHBoxLayout()
        bubble_label = QLabel("Message Bubble Style:")
        bubble_combo = QComboBox()
        bubble_combo.addItems(["iMessage", "Classic", "Modern"])
        bubble_layout.addWidget(bubble_label)
        bubble_layout.addWidget(bubble_combo)
        layout.addLayout(bubble_layout)

        # Font size
        font_layout = QHBoxLayout()
        font_label = QLabel("Font Size:")
        font_combo = QComboBox()
        font_combo.addItems(["Small", "Medium", "Large"])
        font_layout.addWidget(font_label)
        font_layout.addWidget(font_combo)
        layout.addLayout(font_layout)

        # Save button
        save_button = QPushButton("Save Changes")
        save_button.clicked.connect(self.save_appearance_changes)
        layout.addWidget(save_button)

        layout.addStretch()
        return tab

    def create_notifications_tab(self):
        tab = QWidget()
        layout = QVBoxLayout(tab)
        layout.setSpacing(20)
        layout.setContentsMargins(20, 20, 20, 20)
```

```
        # Sound
        sound_layout = QHBoxLayout()
        sound_label = QLabel("Message Sound:")
        sound_combo = QComboBox()
        sound_combo.addItems(["Default", "None", "Custom"])
        sound_layout.addWidget(sound_label)
        sound_layout.addWidget(sound_combo)
        layout.addLayout(sound_layout)

        # Desktop notifications
        desktop_layout = QHBoxLayout()
        desktop_label = QLabel("Desktop Notifications:")
        desktop_check = QCheckBox("Enable desktop notifications")
        desktop_layout.addWidget(desktop_label)
        desktop_layout.addWidget(desktop_check)
        layout.addLayout(desktop_layout)

        # Save button
        save_button = QPushButton("Save Changes")
        save_button.clicked.connect(self.save_notification_changes)
        layout.addWidget(save_button)

        layout.addStretch()
        return tab

    def create_security_tab(self):
        tab = QWidget()
        layout = QVBoxLayout(tab)
        layout.setSpacing(20)
        layout.setContentsMargins(20, 20, 20, 20)

        # End-to-end encryption
        encryption_layout = QHBoxLayout()
        encryption_label = QLabel("End-to-End Encryption:")
        encryption_check = QCheckBox("Enable end-to-end encryption")
        encryption_check.setChecked(True)
        encryption_layout.addWidget(encryption_label)
        encryption_layout.addWidget(encryption_check)
        layout.addLayout(encryption_layout)

        # Two-factor authentication
        two_factor_layout = QHBoxLayout()
        two_factor_label = QLabel("Two-Factor Authentication:")
        two_factor_check = QCheckBox("Enable two-factor authentication")
        two_factor_layout.addWidget(two_factor_label)
        two_factor_layout.addWidget(two_factor_check)
        layout.addLayout(two_factor_layout)

        # Save button
        save_button = QPushButton("Save Changes")
        save_button.clicked.connect(self.save_security_changes)
        layout.addWidget(save_button)
```

```
        layout.addStretch()
        return tab

    def save_profile_changes(self):
        # TODO: Implement profile changes saving
        pass

    def save_appearance_changes(self):
        # TODO: Implement appearance changes saving
        pass

    def save_notification_changes(self):
        # TODO: Implement notification changes saving
        pass

    def save_security_changes(self):
        # TODO: Implement security changes saving
        pass

if __name__ == "__main__":
    app = QApplication(sys.argv)
    client = MainWindow()

    # Show login dialog first
    if client.show_login_dialog():
        # If login successful, show main window and connect to server
        client.show()
        # Connect to server with credentials from login dialog
        client.connect_to_server()
    else:
        # If login failed, exit
        sys.exit(1)

    # Ensure proper cleanup on exit
    app.aboutToQuit.connect(client.cleanup_connection)
    sys.exit(app.exec())

import socket
import ssl
import json
import os
import logging
from datetime import datetime
from cryptography.fernet import Fernet
from threading import Thread, Lock
import sqlite3
import hashlib
import secrets
import threading
from queue import Queue
from typing import Tuple, Dict
import time
```

```
import struct
import queue

class ConnectionPool:
    def __init__(self, max_connections=10):
        self.connections = queue.Queue(maxsize=max_connections)

    def get_connection(self):
        try:
            return self.connections.get(timeout=5)
        except queue.Empty:
            return None

    def put_connection(self, connection):
        try:
            self.connections.put(connection, timeout=5)
        except queue.Full:
            connection.close()

class DatabaseConnectionPool:
    def __init__(self, db_path, max_connections=10):
        self.db_path = db_path
        self.max_connections = max_connections
        self.connections = Queue(maxsize=max_connections)
        self.lock = threading.Lock()
        self._initialize_pool()

    def _initialize_pool(self):
        """Initialize the connection pool"""
        for _ in range(self.max_connections):
            conn = sqlite3.connect(self.db_path, check_same_thread=False)
            conn.row_factory = sqlite3.Row
            self.connections.put(conn)

    def get_connection(self):
        """Get a connection from the pool"""
        try:
            return self.connections.get(timeout=5)
        except Queue.Empty:
            logging.error("No available database connections")
            raise

    def return_connection(self, conn):
        """Return a connection to the pool"""
        try:
            self.connections.put(conn, timeout=5)
        except Queue.Full:
            conn.close()
            logging.warning("Connection pool full, closing connection")

    def close_all(self):
        """Close all connections in the pool"""
        while not self.connections.empty():
```

```
            conn = self.connections.get()
            conn.close()

class SecureMessengerServer:
    def __init__(self, host='localhost', port=5000):
        self.host = host
        self.port = port
        self.server_socket = None
        self.ssl_context = None
        self.clients = {}  # username -> (socket, address)
        self.message_queue = queue.Queue()
        self.running = False

        # Initialize database pool first
        self.db_pool = DatabaseConnectionPool('users.db')

        # Store conversations and message history
        self.conversations = {}  # (user1, user2) -> [messages]
        self.conversations_lock = Lock()

        # Generate server key
        self.server_key = Fernet.generate_key()
        self.cipher_suite = Fernet(self.server_key)

        # Setup logging
        logging.basicConfig(
            level=logging.DEBUG,
            format='%(asctime)s - %(levelname)s - [%(filename)s:%(lineno)d] - %(message)s',
            handlers=[
                logging.FileHandler('server.log'),
                logging.StreamHandler()  # This will output to terminal
            ]
        )

        # Initialize database tables
        self._init_database()

        # Print server info
        print("\n=== Secure Messenger Server ===")
        print(f"Host: {self.host}")
        print(f"Port: {self.port}")
        print("============================\n")

    def _init_database(self):
        """Initialize database tables if they don't exist"""
        db = self.get_db_connection()
        cursor = db.cursor()
        try:
            # Create users table
            cursor.execute('''
                CREATE TABLE IF NOT EXISTS users (
                    username TEXT PRIMARY KEY,
                    password_hash TEXT NOT NULL,
```

```
            salt TEXT NOT NULL,
            display_name TEXT DEFAULT '',
            avatar_color TEXT DEFAULT '#1E90FF',
            status TEXT DEFAULT 'offline',
            last_seen TIMESTAMP DEFAULT CURRENT_TIMESTAMP
        )
    ''')

    # Create messages table
    cursor.execute('''
        CREATE TABLE IF NOT EXISTS messages (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            sender TEXT NOT NULL,
            recipient TEXT NOT NULL,
            content TEXT NOT NULL,
            timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
            type TEXT DEFAULT 'text',
            status TEXT DEFAULT 'sent',
            FOREIGN KEY (sender) REFERENCES users (username),
            FOREIGN KEY (recipient) REFERENCES users (username)
        )
    ''')

    # Create conversations table
    cursor.execute('''
        CREATE TABLE IF NOT EXISTS conversations (
            user1 TEXT NOT NULL,
            user2 TEXT NOT NULL,
            last_message_id INTEGER,
            PRIMARY KEY (user1, user2),
            FOREIGN KEY (user1) REFERENCES users (username),
            FOREIGN KEY (user2) REFERENCES users (username),
            FOREIGN KEY (last_message_id) REFERENCES messages (id)
        )
    ''')

    # Add display_name and avatar_color columns if they don't exist
    try:
        cursor.execute("ALTER TABLE users ADD COLUMN display_name TEXT DEFAULT
''")
    except sqlite3.OperationalError:
        pass  # Column already exists

    try:
        cursor.execute("ALTER TABLE users ADD COLUMN avatar_color TEXT DEFAULT
'#1E90FF'")
    except sqlite3.OperationalError:
        pass  # Column already exists

    db.commit()
    logging.info("Database initialized successfully")
except Exception as e:
    logging.error(f"Error initializing database: {e}")
```

```
            raise
        finally:
            self.return_db_connection(db)

    def get_db_connection(self):
        """Get a database connection from the pool"""
        return self.db_pool.get_connection()

    def return_db_connection(self, conn):
        """Return a database connection to the pool"""
        self.db_pool.return_connection(conn)

    def __del__(self):
        """Cleanup when server is destroyed"""
        self.db_pool.close_all()

    def hash_password(self, password, salt=None):
        if salt is None:
            salt = secrets.token_hex(16)
        return hashlib.sha256((password + salt).encode()).hexdigest(), salt

    def register_user(self, username, password, display_name=None, avatar_color=None):
        """Register a new user"""
        db = self.get_db_connection()
        cursor = db.cursor()
        try:
            # Check if user exists
            cursor.execute('SELECT username FROM users WHERE username = ?', (username,))
            if cursor.fetchone():
                return False, "Username already exists"

            # Hash password
            password_hash, salt = self.hash_password(password)

            # Insert user
            cursor.execute('''
                INSERT INTO users (username, password_hash, salt, display_name, avatar_color)
                VALUES (?, ?, ?, ?, ?)
            ''', (username, password_hash, salt, display_name or username, avatar_color or '#1E90FF'))

            db.commit()
            return True, "Registration successful"
        except Exception as e:
            logging.error(f"Error registering user: {e}")
            return False, str(e)
        finally:
            self.return_db_connection(db)

    def authenticate_user(self, username, password):
        db = self.get_db_connection()
        cursor = db.cursor()
        try:
            # For testing, accept any username/password
```

```
        return True, "Authentication successful"

        # Uncomment this for actual authentication
        # cursor.execute(
        #     'SELECT password_hash, salt FROM users WHERE username = ?',
        #     (username,)
        # )
        # result = cursor.fetchone()
        # if not result:
        #     return False, "Invalid username or password"
        #
        # stored_hash, salt = result
        # password_hash, _ = self.hash_password(password, salt)
        #
        # if password_hash == stored_hash:
        #     return True, "Authentication successful"
        # return False, "Invalid username or password"
    except Exception as e:
        return False, str(e)
    finally:
        db.close()

def get_conversation_id(self, user1, user2):
    """Get or create a conversation ID for two users"""
    # Sort usernames to ensure consistent conversation ID
    participants = tuple(sorted([user1, user2]))
    return participants

def store_message(self, sender, recipient, content, message_type='text'):
    """Store a message in the database"""
    db = self.get_db_connection()
    cursor = db.cursor()
    try:
        # Insert message
        cursor.execute('''
            INSERT INTO messages (sender, recipient, content, type)
            VALUES (?, ?, ?, ?)
        ''', (sender, recipient, content, message_type))

        message_id = cursor.lastrowid

        # Get sorted participants for consistent conversation lookup
        participants = tuple(sorted([sender, recipient]))

        # Update or create conversation
        cursor.execute('''
            INSERT INTO conversations (user1, user2, last_message_id)
            VALUES (?, ?, ?)
            ON CONFLICT(user1, user2) DO UPDATE SET last_message_id = ?
        ''', (participants[0], participants[1], message_id, message_id))

        db.commit()
        return message_id
```

```
    finally:
        db.close()

def get_conversation_history(self, user1, user2, limit=50):
    """Get conversation history between two users"""
    db = self.get_db_connection()
    cursor = db.cursor()
    try:
        cursor.execute('''
            SELECT m.*, u1.status as sender_status, u2.status as recipient_status
            FROM messages m
            JOIN users u1 ON m.sender = u1.username
            JOIN users u2 ON m.recipient = u2.username
            WHERE (m.sender = ? AND m.recipient = ?)
               OR (m.sender = ? AND m.recipient = ?)
            ORDER BY m.timestamp DESC
            LIMIT ?
        ''', (user1, user2, user2, user1, limit))

        messages = []
        for row in cursor.fetchall():
            messages.append({
                'id': row[0],
                'sender': row[1],
                'recipient': row[2],
                'content': row[3],
                'timestamp': row[4],
                'type': row[5],
                'status': row[6],
                'sender_status': row[7],
                'recipient_status': row[8]
            })
        return messages
    finally:
        db.close()

def update_user_status(self, username, status):
    """Update a user's online status"""
    db = self.get_db_connection()
    cursor = db.cursor()
    try:
        cursor.execute('''
            UPDATE users
            SET status = ?,
                last_seen = CURRENT_TIMESTAMP
            WHERE username = ?
        ''', (status, username))
        db.commit()
    finally:
        db.close()

def send_message_to_client(self, client_socket, message):
    """Send a message to a client"""
```

```
        try:
            # Convert message to JSON and encode
            message_data = json.dumps(message).encode('utf-8')

            # Send message length (4 bytes)
            msg_len = struct.pack('!I', len(message_data))
            client_socket.sendall(msg_len)

            # Small delay to ensure the client is ready
            time.sleep(0.01)

            # Send message data
            client_socket.sendall(message_data)
            logging.debug(f"Sent message to client: {message}")

        except Exception as e:
            logging.error(f"Error sending message to client: {e}")
            raise

    def handle_client(self, client_socket, address):
        """Handle client connection"""
        username = None
        buffer = b""

        try:
            while self.running:
                try:
                    # Read message length (4 bytes)
                    while len(buffer) < 4:
                        data = client_socket.recv(4096)
                        if not data:
                            break
                        buffer += data

                    if len(buffer) < 4:
                        break

                    msg_len = struct.unpack('!I', buffer[:4])[0]
                    buffer = buffer[4:]

                    # Read message data
                    while len(buffer) < msg_len:
                        data = client_socket.recv(4096)
                        if not data:
                            break
                        buffer += data

                    if len(buffer) < msg_len:
                        break

                    message_data = buffer[:msg_len]
                    buffer = buffer[msg_len:]
```

```
                try:
                    message = json.loads(message_data.decode('utf-8'))
                    logging.debug(f"Received message from {address}: {message}")
                    self._process_message(client_socket, address, message)
                except json.JSONDecodeError as e:
                    logging.error(f"Invalid message format from {address}: {e}")
                    continue

            except socket.timeout:
                continue
            except Exception as e:
                logging.error(f"Error handling client message from {address}: {e}")
                break

        except Exception as e:
            logging.error(f"Error in client handler for {address}: {e}")
        finally:
            if username:
                self.update_user_status(username, 'offline')
                if username in self.clients:
                    del self.clients[username]
            try:
                client_socket.close()
            except:
                pass

    def _process_message(self, client_socket: socket.socket, client_address: Tuple[str, int], message:
Dict):
        """Process a message from a client"""
        try:
            message_type = message.get('type')
            username = message.get('username')

            if message_type == 'login':
                # Handle login
                password = message.get('password')
                if not username or not password:
                    self.send_message_to_client(client_socket, {'type': 'error', 'content': 'Missing username
or password'})
                    return

                # Authenticate user
                success, response = self.authenticate_user(username, password)
                if success:
                    # Add client to active clients
                    self.clients[username] = (client_socket, client_address)
                    logging.info(f"Client {username} authenticated successfully")

                    # Send success response
                    self.send_message_to_client(client_socket, {'type': 'success', 'content': response})

                    # Send user list
                    user_list = self.get_user_list()
```

```
                self.send_message_to_client(client_socket, {'type': 'user_list', 'users': user_list})

                # Update user status
                self.update_user_status(username, 'online')
            else:
                self.send_message_to_client(client_socket, {'type': 'error', 'content': response})

        elif message_type == 'register':
            # Handle registration
            password = message.get('password')
            display_name = message.get('display_name', username)
            avatar_color = message.get('avatar_color', '#1E90FF')

            if not username or not password:
                self.send_message_to_client(client_socket, {'type': 'error', 'content': 'Missing username
or password'})
                return

            # Register user
            if self.register_user(username, password, display_name, avatar_color):
                # Add client to active clients
                with self.conversations_lock:
                    self.conversations[self.get_conversation_id(username, username)] = []
                logging.info(f"New user {username} registered successfully")

                # Send success response
                self.send_message_to_client(client_socket, {'type': 'success', 'content': 'Registration
successful'})

                # Send user list
                user_list = self.get_user_list()
                self.send_message_to_client(client_socket, {'type': 'user_list', 'users': user_list})

                # Update user status
                self.update_user_status(username, 'online')
            else:
                self.send_message_to_client(client_socket, {'type': 'error', 'content': 'Username already
exists'})

        elif message_type == 'get_users':
            # Send user list
            user_list = self.get_user_list()
            self.send_message_to_client(client_socket, {'type': 'user_list', 'users': user_list})

        elif message_type == 'text':
            # Handle text message
            recipient = message.get('recipient')
            content = message.get('content')

            if recipient in self.conversations:
                # Forward message to recipient
                self.conversations[self.get_conversation_id(username, recipient)].append({
                    'type': 'text',
```

```
                    'sender': username,
                    'content': content,
                    'timestamp': time.time()
                })
                self.update_user_status(username, 'online')
                self.update_user_status(recipient, 'online')

            elif message_type == 'typing':
                # Handle typing indicator
                recipient = message.get('recipient')
                is_typing = message.get('is_typing', False)

                if recipient in self.conversations:
                    self.conversations[self.get_conversation_id(username, recipient)].append({
                        'type': 'typing',
                        'sender': username,
                        'is_typing': is_typing
                    })
                    self.update_user_status(username, 'online')
                    self.update_user_status(recipient, 'online')

            elif message_type == 'read_receipt':
                # Handle read receipt
                message_id = message.get('message_id')
                sender = message.get('sender')

                if sender in self.conversations:
                    self.conversations[self.get_conversation_id(username, sender)].append({
                        'type': 'read_receipt',
                        'message_id': message_id,
                        'reader': username
                    })

            elif message_type == 'heartbeat':
                # Update last seen time
                self.update_user_status(username, 'online')

        except Exception as e:
            logging.error(f"Error handling message from {username}: {e}")
            if username in self.conversations:
                self.send_message_to_client(client_socket, {'type': 'error', 'content': str(e)})

    def broadcast_user_list(self):
        """Broadcast the list of users and their statuses to all connected clients"""
        db = self.get_db_connection()
        cursor = db.cursor()
        try:
            # Get all users and their statuses
            cursor.execute('''
                SELECT username, status, last_seen
                FROM users
            ''')
            all_users = cursor.fetchall()
```

```
        # Create user info dictionary
        user_info = {}
        for user in all_users:
            username = user[0]  # First column is username
            user_info[username] = {
                'status': user[1],  # Second column is status
                'last_seen': user[2]  # Third column is last_seen
            }

        # Send user list to all connected clients
        user_list_message = json.dumps({
            'type': 'user_list',
            'users': user_info
        }).encode()

        with self.conversations_lock:
            for conversation in self.conversations.values():
                for message in conversation:
                    if message['type'] == 'text' or message['type'] == 'typing':
                        recipient = message['sender'] if message['type'] == 'text' else message['recipient']
                        if recipient in user_info:
                            try:
                                user_info[recipient]['last_seen'] = message['timestamp']
                            except Exception as e:
                                logging.error(f"Error updating user status: {e}")

        with self.conversations_lock:
            for client_socket, _ in self.clients.values():
                try:
                    client_socket.send(user_list_message)
                    logging.debug(f"Sent user list: {user_info}")
                except Exception as e:
                    logging.error(f"Error sending user list: {e}")
    finally:
        if 'db' in locals():
            self.return_db_connection(db)

def start(self):
    """Start the server"""
    try:
        # Create server socket
        self.server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        self.server_socket.bind((self.host, self.port))
        self.server_socket.listen(5)
        self.server_socket.settimeout(1)  # 1 second timeout for accept

        # Create SSL context
        self.ssl_context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
        self.ssl_context.load_cert_chain('server.crt', 'server.key')
        self.ssl_context.verify_mode = ssl.CERT_NONE  # For development only
```

```
        self.ssl_context.options |= ssl.OP_NO_SSLv2 | ssl.OP_NO_SSLv3  # Disable old SSL versions

        logging.info(f"Server started on {self.host}:{self.port}")
        self.running = True

        # Start message processing thread
        threading.Thread(target=self._process_messages, daemon=True).start()

        # Main server loop
        while self.running:
            try:
                client_socket, address = self.server_socket.accept()
                logging.info(f"New connection from {address}")

                try:
                    # Wrap the socket with SSL
                    ssl_socket = self.ssl_context.wrap_socket(
                        client_socket,
                        server_side=True,
                        do_handshake_on_connect=True
                    )
                    logging.info(f"SSL handshake completed for {address}")

                    # Set socket timeout
                    ssl_socket.settimeout(5.0)  # 5 second timeout for operations

                    # Start client handler thread
                    threading.Thread(
                        target=self.handle_client,
                        args=(ssl_socket, address),
                        daemon=True
                    ).start()

                except ssl.SSLError as e:
                    logging.error(f"SSL handshake failed for {address}: {e}")
                    client_socket.close()
                    continue

            except socket.timeout:
                continue
            except Exception as e:
                logging.error(f"Error accepting connection: {e}")
                if not self.running:
                    break

    except Exception as e:
        logging.error(f"Server error: {e}")
    finally:
        self.stop()

def get_user_list(self):
    """Get list of all users with their status"""
```

```
            try:
                db = self.get_db_connection()
                cursor = db.cursor()
                try:
                    cursor.execute('''
                        SELECT username, status, last_seen, display_name, avatar_color
                        FROM users
                        ORDER BY username
                    ''')
                    users = []
                    for row in cursor.fetchall():
                        user = {
                            'username': row[0],
                            'status': row[1],
                            'last_seen': row[2],
                            'display_name': row[3],
                            'avatar_color': row[4]
                        }
                        users.append(user)
                    return users
                except Exception as e:
                    logging.error(f"Error fetching user list: {e}")
                    return []
                finally:
                    self.return_db_connection(db)
            except Exception as e:
                logging.error(f"Error getting database connection: {e}")
                return []

        def _process_messages(self):
            """Process messages from the message queue"""
            while self.running:
                try:
                    message = self.message_queue.get(timeout=1)
                    self._process_message(message['socket'], message['address'], message['message'])
                except queue.Empty:
                    continue
                except Exception as e:
                    logging.error(f"Error processing message: {e}")
        def stop(self):
            """Stop the server"""
            self.running = False
            if self.server_socket:
                try:
                    self.server_socket.close()
                except Exception as e:
                    logging.error(f"Error closing server socket: {e}")
            self.server_socket = None
    if __name__ == "__main__":
        try:
            server = SecureMessengerServer()
            server.start()
        except Exception as e:
```

Додаток Г. Ілюстративна частина

ІЛЮСТРАТИВНА ЧАСТИНА

Розробка системи обміну інформації із швидким розгортанням на власних потужностях для комерційного використання.

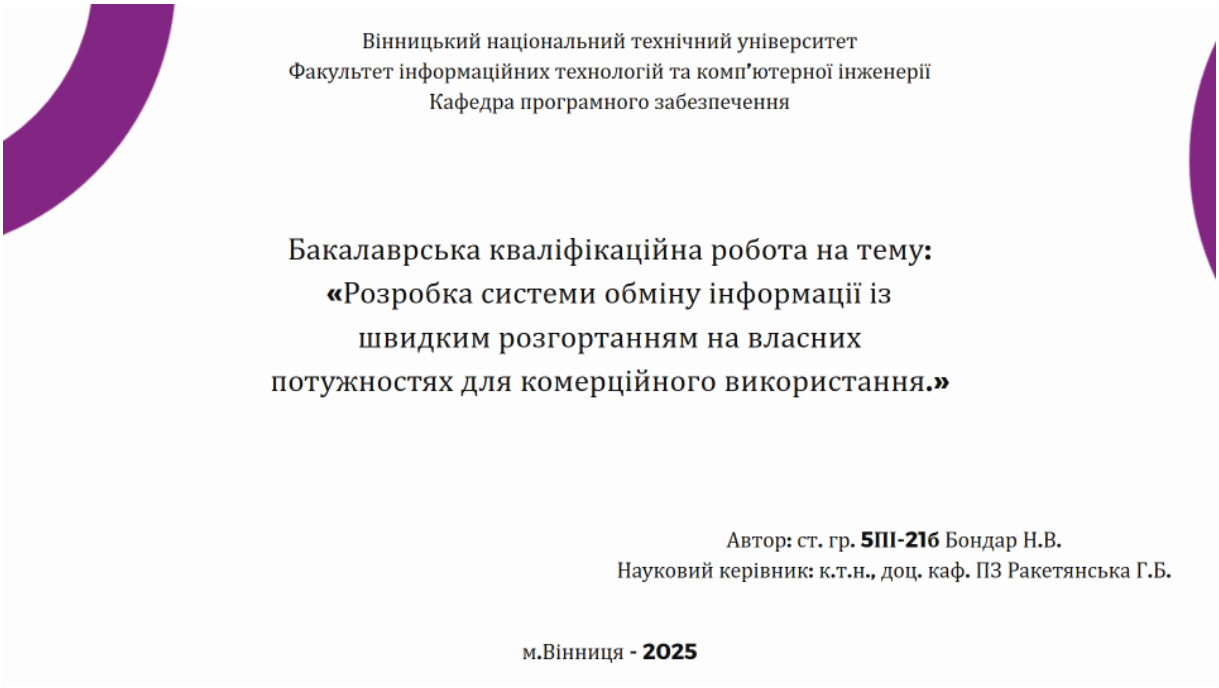

Рисунок Г.1 – Титульний слайд

Актуальність

Актуальність даної роботи визначається зростаючою потребою організацій у швидкому та ефективному розгортанні систем обміну інформацією, які б відповідали сучасним вимогам щодо гнучкості, масштабованості, надійності та безпеки. Розробка програмних модулів для таких систем є важливим кроком на шляху до створення комплексної інфраструктури, що забезпечує безперешкодний та ефективний обмін даними між різними учасниками інформаційного процесу.

Рисунок Г.2 – Актуальність теми

Система обміну інформацією

Системи обміну інформацією – це комплексні програмні рішення, розроблені для забезпечення ефективної та безпечної передачі даних між різними учасниками комунікації. Вони задовольняють нагальні потреби у взаємодії, дозволяючи користувачам миттєво обмінюватися повідомленнями, файлами та іншим цифровим контентом. Ці системи є фундаментальним елементом сучасного інформаційного простору, відіграючи ключову роль у повсякденній особистій та професійній взаємодії.

Рисунок Г.3 – Система обміну інформацією

Мета, об’єкт та предмет дослідження

- Метою бакалаврської дипломної роботи є процес покращення програмних модулів системи обміну інформацією з можливістю швидкого розгортання на власних потужностях задля забезпечення ефективної, надійної та безпечної передачі даних між користувачами або системами.
- Об'єктом дослідження є процеси обміну інформацією в організаціях.
- Предметом дослідження є методи, моделі, алгоритми та програмні засоби для розробки та розгортання системи обміну інформацією на власних потужностях.

Рисунок Г.4 – Мета, об’єкт та предмет дослідження

Новизна роботи

- Практичне значення бакалаврської кваліфікаційної роботи полягає у можливості використання програми організаціями, які прагнуть забезпечити ефективний та безпечний обмін інформацією між своїми підрозділами, співробітниками або зовнішніми користувачами. Розроблені програмні модулі можуть бути використані для створення корпоративної системи обміну інформацією, що забезпечує швидкий обмін даними, підвищує продуктивність командної роботи та покращує внутрішні та зовнішні комунікації. Крім того, розроблені підходи та технології можуть бути адаптовані для створення систем обміну інформацією для різних цілей, таких як обмін повідомленнями між пристроями, збір та аналіз даних з датчиків, або організація розподілених обчислень.

Рисунок Г.5 – Новизна роботи

Цілі дослідження

- провести порівняльний аналіз існуючих систем обміну інформацією та технологій, що використовуються для їх розробки, з метою виявлення їх переваг, недоліків та обмежень;
- визначити функціональні та нефункціональні вимоги до програмних модулів системи обміну інформацією, включаючи вимоги до користувацького інтерфейсу, протоколу обміну даними, архітектури системи, безпеки, масштабованості та можливості швидкого розгортання;
- розробити концептуальну архітектуру системи обміну інформацією, включаючи опис клієнтської та серверної частин, їх компонентів, інтерфейсів та взаємодії;
- обрати та обґрунтувати вибір мови програмування та інших технологій і інструментів для розробки програмних модулів системи;
- розробити алгоритми та програмний код для реалізації основних функцій системи обміну інформацією, таких як реєстрація та авторизація користувачів, надсилання та отримання даних, створення та управління каналами обміну, відображення статусу користувачів, зберігання історії даних та забезпечення безпеки передачі даних;
- провести тестування та оцінку ефективності розроблених програмних модулів системи обміну інформацією, включаючи функціональне тестування, тестування продуктивності, тестування безпеки та тестування зручності використання;
- розробити рекомендації щодо розгортання, налаштування та подальшого розвитку системи обміну інформацією на власних потужностях організації.

Рисунок Г.6 – Цілі дослідження

Порівняльний аналіз з аналогами

Критерії	Matrix Synapse	Rocket.Chat	Zulip	Mattermost	Prosody	Власна розробка
Швидке розгортання	-	-	-	-	+	+
Мінімальні вимоги до ресурсів	-	-	-	-	+	+
Автономна робота	+	+	+	+	+	+
Простота інтеграції	-	+	+	+	-	+
Гнучкість налаштування	+	+	+	+	-	+
Загальна оцінка	60%	70%	65%	75%	50%	100%

Рисунок Г.7 – Порівняльний аналіз з аналогами

Діаграма діяльності клієнта

Рисунок Г.8 – Діаграма діяльності клієнта

Алгоритм роботи програмного застосунку

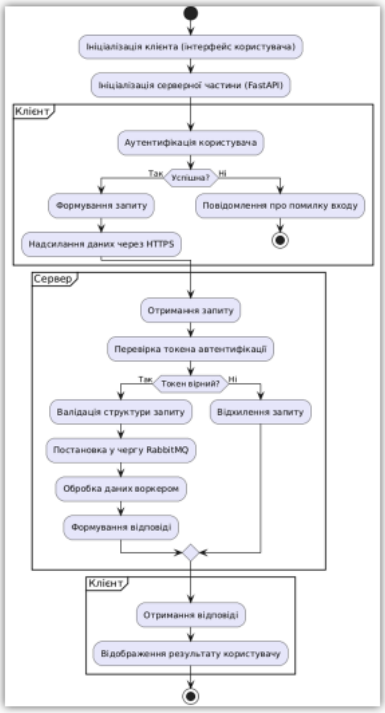

Рисунок Г.9 – Алгоритм роботи програмного застосунку

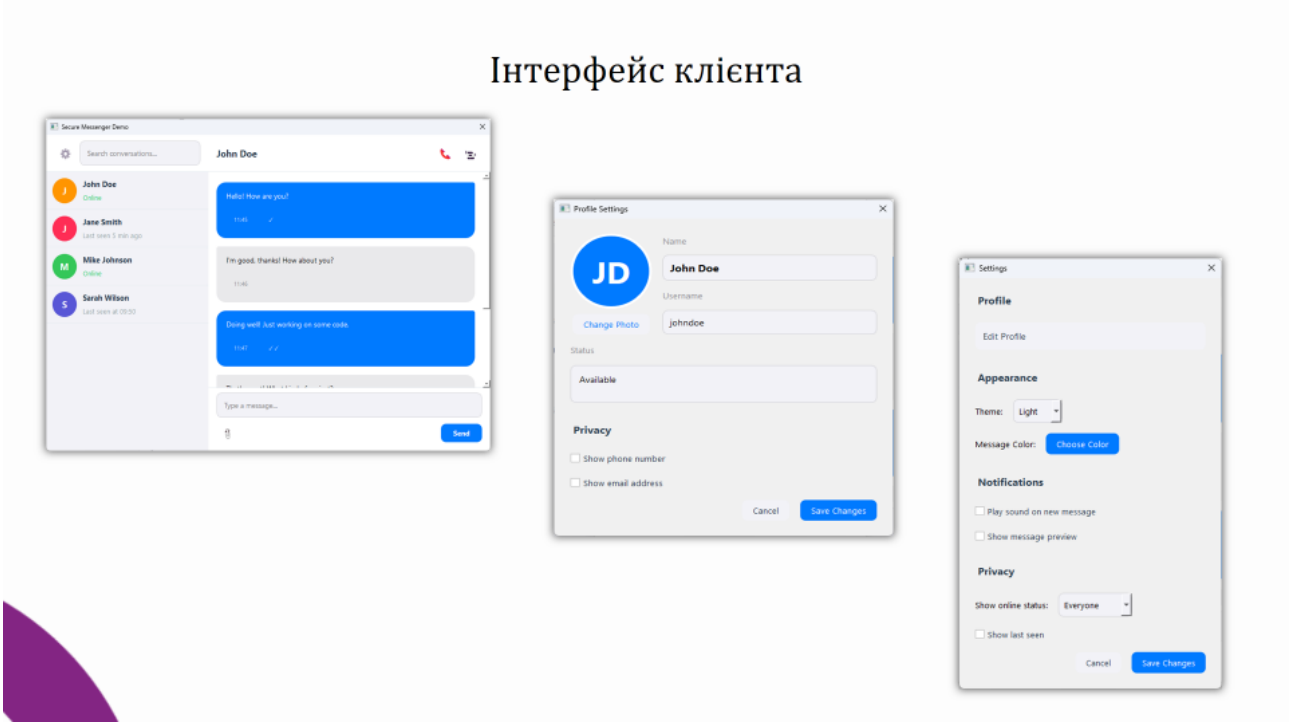

Рисунок Г.10 – Інтерфейс клієнта

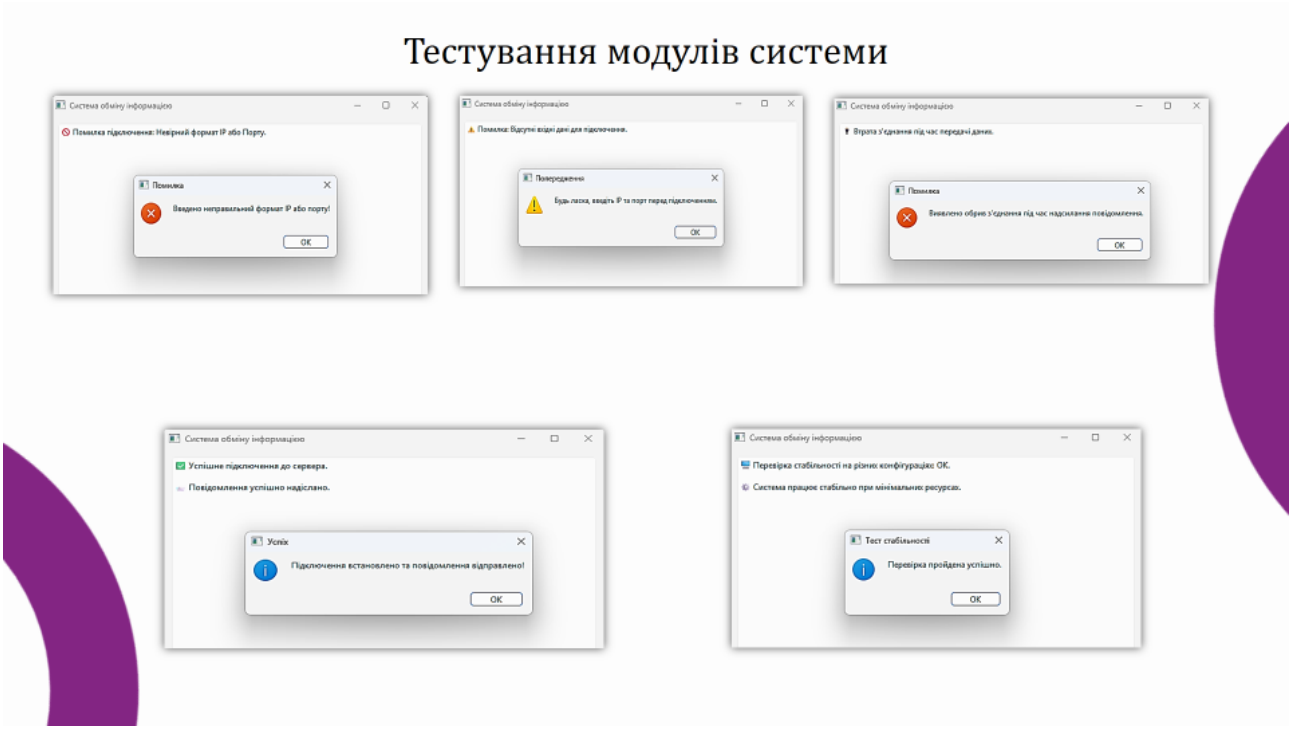

Рисунок Г.11 – Тестування модулів системи

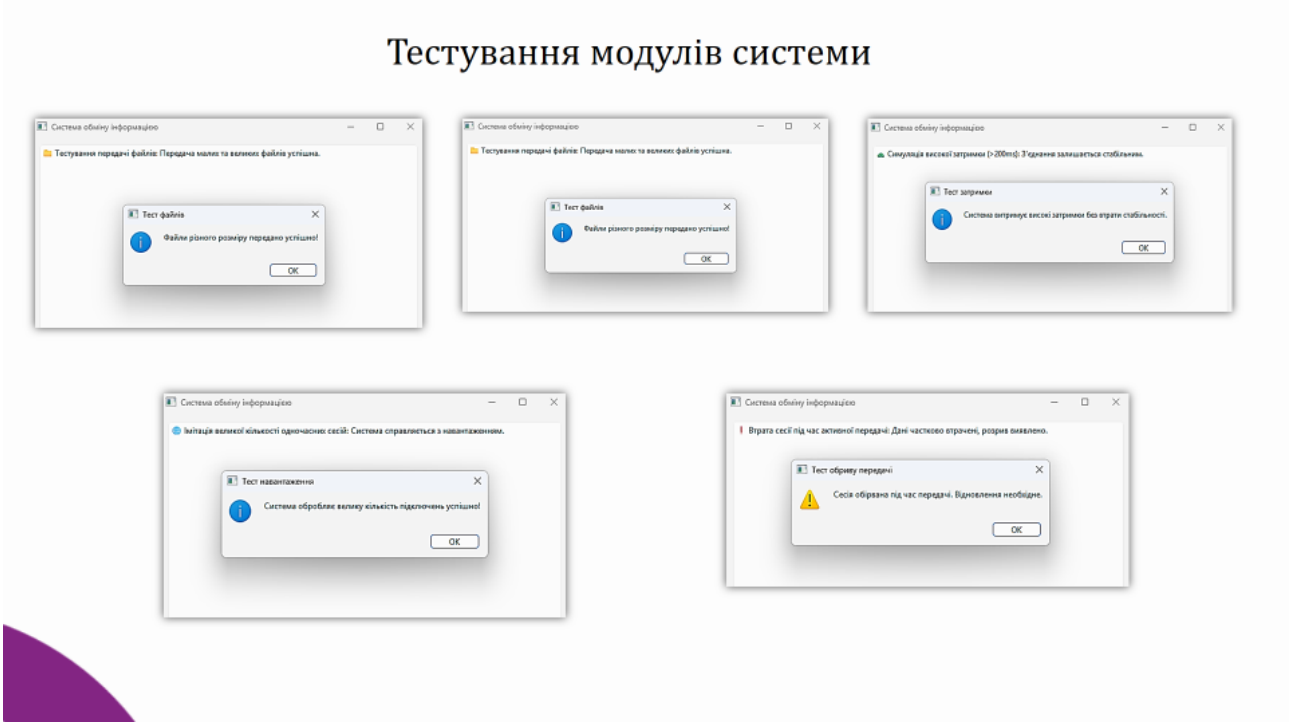

Рисунок Г.12 – Тестування модулів системи

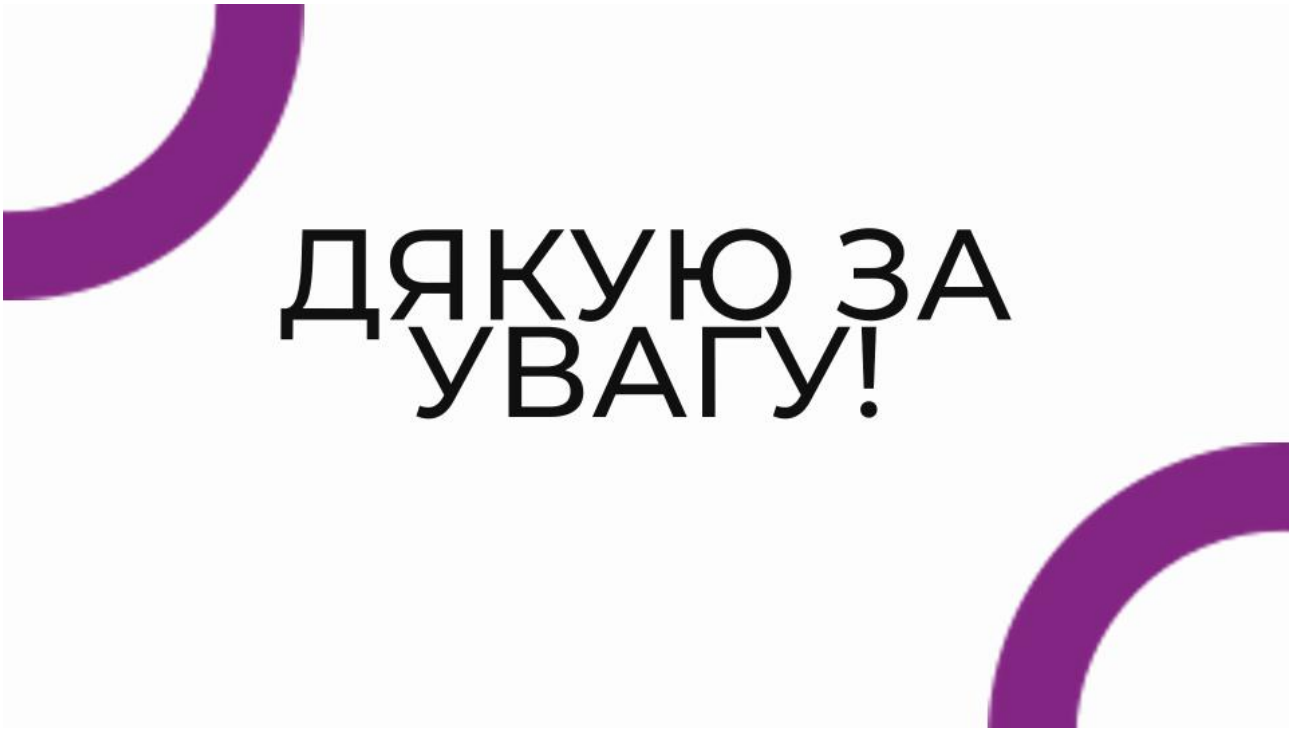

Рисунок Г.13 - Фінальний слайд