

Міністерство освіти і науки України
Державний заклад
«Луганський національний університет імені Тараса Шевченка»

Факультет технологій та інформаційних систем

Кафедра інформаційних технологій та систем

Яценко Михайло Сергійович

**РОЗРОБКА КЛІЄНТ-СЕРВЕРНОЇ СИСТЕМИ ОБМІНУ ФАЙЛАМИ З
ПІДТРИМКОЮ БАГАТОПОТОЧНОСТІ ЗАСОБАМИ PYTHON**

кваліфікаційна робота

здобувача вищої освіти першого (бакалаврського) рівня
освітньої програми «Інженерія програмного забезпечення»
за спеціальністю 121 Інженерія програмного забезпечення

Особистий підпис  Михайло ЯЦЕНКО

Науковий керівник  Світлана ПЕРЕЯСЛАВСЬКА,
кандидат педагогічних наук, доцент
кафедри інформаційних технологій
та систем

Завідувач кафедри  Микола СЕМЕНОВ,
кандидат педагогічних наук, доцент
кафедри інформаційних технологій
та систем

Лубни – 2026

Міністерство освіти і науки України	
Державний заклад «Луганський національний університет імені Тараса Шевченка»	
Факультет (інститут)	Навчально-науковий інститут математики та інформаційних технологій (повна назва)
Кафедра	Інформаційних технологій та систем (повна назва)
Освітній ступень	Бакалавр (код, назва)
Напрямок підготовки	Інженерія програмного забезпечення (код, назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри ІТС
М.А. Семенов

(підпис)

(ініціали, прізвище)

“ ” 2025 р.

**ЗАВДАННЯ
НА ДИПЛОМНИЙ ПРОЕКТ (РОБОТУ) СТУДЕНТУ**

Яценку Михайлу Сергійовичу

(прізвище, ім'я, по батькові)

1. Тема проекту (роботи) **Розробка клієнт-серверної системи обміну
файлами з підтримкою багатопоточності
засобами Python**

Керівник кваліфікаційної роботи

Переяславська С.О. к.п.н., доцент

(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджена наказом по університету

від

2. Строк подання студентом проекту (роботи)

3. Вихідні дані до роботи (проекту) Створено багатопоточну клієнт-серверну систему для обміну файлами, реалізовану мовою програмування Python із використанням сучасного інструментарію розробки

(визначаються кількісні або (та) якісні показники, яким повинен відповідати об'єкт розробки)

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) **НАУКОВО-ДОСЛІДНА ЧАСТИНА. РОЗРОБКА КЛІЄНТ-СЕРВЕРНОЇ СИСТЕМИ**

(визначаються назви розділів або (та) перелік питань, які повинні увійти до тексту ПЗ)

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

6. Консультанти розділів проекту/роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання „_____” _____ 2025р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту (роботи)	Примітка
	Вибір теми роботи, вивчення наукової літератури, затвердження теми та керівника.	До 15 жовтня	
	Аналіз літературних джерел за темою роботи. Розробка та апробація методики дослідно-експериментальної роботи. Подання структури теоретичної частини роботи та плану експериментальних досліджень.	Другий тиждень листопада (10 листопада)	
	Робота над теоретичною частиною. Подання теоретичної частини роботи для першого читання науковим керівником.	До 15 грудня	
	Усунення зауважень, урахування рекомендацій наукового керівника. Подання теоретичної частини роботи на друге читання.	До 28 січня	
	Проведення експериментальної роботи. Поетапний аналіз та обговорення її результатів. Перевірка стану виконання роботи.	Перший тиждень березня	
	Урахування рекомендацій наукового керівника, усунення недоліків, підготовка варіанта роботи до передзахисту. Розробка презентації.	До 31 березня	
	Попередній захист роботи на кафедрі	квітень	
	Доопрацювання роботи з урахуванням рекомендацій після передзахисту. Подання роботи науковому керівникові та рецензентові на підготовку відгуку та рецензії	За 10 днів до державної атестації	
	Подання на кафедру остаточного варіанта роботи, переплетеного та підписаного автором, науковим керівником і рецензентом.	За 5 днів до державної атестації	

Студент

Керівник проекту (роботи)

підпис

підпис

М.С. Яценко

(ініціали, прізвище)

С. О. Переяславська

(ініціали, прізвище)

АНОТАЦІЯ

Яценко М.С.

Тема: Розробка клієнт-серверної системи обміну файлами з підтримкою багатопоточності засобами Python.

Спеціальність: 121 «Інженерія програмного забезпечення»

Установа: ЛНУ імені Тараса Шевченка, 2026 р.

Бакалаврська робота містить: 72 с., 8 рис., 5 табл., 40 джерел.

Об'єкт дослідження - процес мережевої взаємодії та обміну даними між клієнтом і сервером в умовах паралельного опрацювання запитів.

Предмет дослідження - методи, алгоритми та програмні засоби мови Python для реалізації багатопоточної клієнт-серверної архітектури обміну файлами.

Мета роботи - проєктування, розробка та дослідження ефективності клієнт-серверної системи обміну файлами з підтримкою багатопоточності мовою Python для забезпечення швидкої, надійної та паралельної обробки запитів від багатьох користувачів одночасно.

Результати роботи. Було розроблено архітектуру та програмне забезпечення клієнт-серверної системи обміну файлами. Спроектовано структурну та функціональну схеми взаємодії між клієнтом та центральним сервером. Розроблено алгоритми системи: Алгоритм багатопоточного оброблення запитів, який дозволяє серверу обслуговувати кожного клієнта в окремому потоці виконання без блокування загальної черги; Алгоритм синхронізації доступу до файлової системи, що унеможливорює виникнення конфліктів при одночасному читанні чи записі файлу декількома користувачами; Алгоритм порційної передачі даних, який забезпечує завантаження файлів із контролем цілісності пакетів; Алгоритм авторизації та розмежування прав доступу користувачів до файлових каталогів.

Ключові слова: Клієнт-серверна архітектура, Обмін файлами, Багатопоточність, Python, Сокети, Мережева взаємодія, Синхронізація потоків, Паралельна обробка даних.

ABSTRACT

Yatsenko Mykhailo

Theme: Development of a Client-Server File Sharing System with Multithreading Support Using Python.

Speciality: 121 "Software Engineering"

Institution: Luhansk Taras Shevchenko National University (LTSNU), 2026.

Diploma work contains: 72 pages, 8 Figs., 5 Tables, 40 sources.

Object of research is the process of network interaction and data exchange between a client and a server under conditions of parallel request processing.

Subject of research is the methods, algorithms, and software tools of the Python language for implementing a multithreaded client-server file sharing architecture.

The objective of the thesis is the design, development, and efficiency evaluation of a client-server file sharing system with multithreading support in Python to ensure fast, reliable, and parallel processing of requests from multiple concurrent users.

Results of the work. As a result of the conducted research, the architecture and software for a client-server file sharing system were developed. The structural and functional diagrams of interaction between client applications and the central server were designed. The following key system algorithms were developed and implemented: A multithreaded request processing algorithm, which allows the server to serve each client in isolation within a separate thread of execution without blocking the main queue. A file system access synchronization algorithm, which prevents conflicts (race conditions) during concurrent reading or writing of the same file by multiple users. A chunked data transfer algorithm, which ensures stable uploading and downloading of large files with packet integrity control. An authentication and user access control algorithm for managing permissions within file directories.

Keywords: Client-server architecture, File sharing, Multithreading, Python, Sockets, Network interaction, Thread synchronization, Parallel data processing.

Міністерство освіти і науки України
Державний заклад «Луганський національний університет
імені Тараса Шевченка»

Факультет _____ Факультет технологій та інформаційних систем
(інститут) (повна назва)
_____ Інформаційних технологій та систем
(повна назва)

Кафедра

ТЕХНІЧНЕ ЗАВДАННЯ
на виконання програмної розробки (ПР):

"РОЗРОБКА КЛІЄНТ-СЕРВЕРНОЇ СИСТЕМИ ОБМІНУ
ФАЙЛАМИ З ПІДТРИМКОЮ БАГАТОПОТОЧНОСТІ
ЗАСОБАМИ PYTHON"

ПОГОДЖЕНО
Керівник кваліфікаційної роботи

Переславська С.О.
" _____ " _____ 2026 р.

ВИКОНАВЕЦЬ
Студент групи 4 ІПЗ

Яценко М.С.
" _____ " _____ 2026 р.

Лубни – 2026

ЗМІСТ

Вступ.....	3
1. Характеристика об'єкту.....	3
2. Призначення об'єкту.....	4
3. Вимоги системи.....	5
3.1. Загальні вимоги.....	5
3.2. Функціональні вимоги.....	5
3.3. Вимоги до складу та архітектури.....	7
3.4. Вимоги до якості та надійності.....	7
3.5. Вимоги до експлуатації.....	8
3.6. Вимоги до тестування.....	9

ВСТУП

1.1. Найменування: «Клієнт-серверна система обміну файлами».

1.2. Підстава до виконання: Підставою для виконання даної розробки є заява від замовника та завдання на виконання кваліфікаційної роботи за першим (бакалаврським) рівнем освіти.

1.3. Термін розробки:

1.3.1. Початок 20 жовтня 2025 р.

1.3.2. Закінчення 30 травня 2026 р.

1.4. Фінансується за рахунок замовника. Умови фінансування – згідно договору.

1. ХАРАКТЕРИСТИКА ОБ'ЄКТУ

1.1. Розроблювана система повинна забезпечувати обмін файлами між клієнтським застосунком і сервером у комп'ютерній мережі з підтримкою одночасного обслуговування кількох клієнтів. Система має реалізовувати кероване передавання файлів блоками, перевірку цілісності даних, збереження метаданих і журналювання подій.

1.2. До складу об'єкту, що створюється, повинно входити:

1.2.1. Клієнтський застосунок для ініціювання підключення, автентифікації, перегляду доступних файлів, завантаження файлів на сервер і отримання файлів із сервера.

1.2.2. Серверна частина на Python, що приймає TCP-з'єднання, обробляє команди прикладного протоколу, керує файловими операціями, сесіями користувачів і потоками виконання.

1.2.3. Адміністративний вебсервіс для перегляду стану системи, метаданих файлів, активних сесій, історії передавання та журналів подій.

1.2.4. Файлове сховище з розмежуванням тимчасових і завершених файлів, база метаданих SQLite та підсистема журналювання.

1.2.5. Засоби контейнеризованого запуску компонентів системи та набір тестів для перевірки працездатності.

1.3. Вхідна інформація включає вимоги замовника щодо функціональності системи, параметри мережевого підключення, облікові дані користувачів, службові команди прикладного протоколу, файли для передавання, контрольні суми та конфігураційні параметри середовища виконання.

1.4. Вихідна інформація включає передані або отримані файли, службові відповіді сервера, підтвердження приймання блоків, записи про статус операцій, метадані файлів, журнали подій, повідомлення про помилки та результати тестування системи.

2. ПРИЗНАЧЕННЯ ОБ'ЄКТУ

2.1. Призначення: реалізація програмної системи для надійного клієнт-серверного обміну файлами із підтримкою багатопоточності, автентифікації користувачів, рольового розмежування доступу, контролю цілісності переданих даних і відтворюваного журналювання операцій.

2.2. Цільова аудиторія:

2.2.1. Користувачі, яким необхідно передавати та отримувати файли у локальній або тестовій мережевій інфраструктурі.

2.2.2. Адміністратори, які здійснюють контроль стану файлового сервера, перегляд журналів, аналіз сесій і керування доступом до файлів.

2.2.3. Розробники та здобувачі освіти, які використовують систему як навчальний або дослідно-практичний приклад реалізації мережевого програмування, багатопоточності та прикладного протоколу передавання файлів.

2.3. Платформи для роботи системи:

2.3.1. Серверна частина повинна працювати в середовищі, що підтримує Python, TCP/IP-мережеву взаємодію, файлову систему та SQLite.

2.3.2. Контейнеризоване розгортання повинно виконуватися в середовищі Docker або Docker Compose за наявності відповідної конфігурації.

2.3.3. Адміністративний вебінтерфейс повинен бути доступним через сучасний веббраузер у межах налаштованої мережі.

2.3.4. Система не призначена для роботи як публічне промислове хмарне сховище без додаткового аудиту безпеки, масштабування та адміністрування.

3. ВИМОГИ СИСТЕМИ

3.1. Загальні вимоги

3.1.1. Система повинна бути побудована на основі клієнт-серверної архітектури з розмежуванням клієнтської логіки, серверної обробки, файлового сховища, бази метаданих і адміністративного контролю.

3.1.2. Передавання файлів повинно виконуватися через прикладний протокол поверх TCP із використанням службових команд, метаданих, блоків даних і відповідей сервера.

3.1.3. Серверна частина повинна підтримувати одночасну роботу кількох клієнтів без блокування основного циклу приймання підключень.

3.1.4. Файли повинні передаватися блоками без повного завантаження великого файла в оперативну пам'ять.

3.1.5. Система повинна забезпечувати автентифікацію користувачів, сесійні токени, рольове розмежування доступу та захист від некоректних запитів.

3.1.6. Для контролю цілісності переданих даних повинні використовуватися контрольні суми SHA-256 для блоків і повного файла.

3.1.7. Система повинна вести журнал подій, у якому фіксуються підключення, команди, результати операцій, помилки та дії користувачів.

3.2. Функціональні вимоги

3.2.1. Клієнт повинен мати можливість встановлювати з'єднання із сервером і завершувати роботу через службову команду QUIT.

3.2.2. Користувач повинен мати можливість виконувати автентифікацію через команду LOGIN з отриманням сесійного токена у разі коректних облікових даних.

3.2.3. Система повинна підтримувати команду PING для перевірки доступності сервера та коректності активної сесії.

3.2.4. Користувач повинен мати можливість отримувати список доступних файлів через команду LIST та переглядати інформацію про окремий файл через команду INFO.

3.2.5. Завантаження файла на сервер повинно виконуватися за сценарієм UPLOAD_INIT → CHUNK → UPLOAD_FINISH із передаванням метаданих, блоків файла та фінальною перевіркою цілісності.

3.2.6. Для кожного прийнятого блока сервер повинен повертати підтвердження ACK або повідомлення NACK у разі помилки контрольної суми, розміру або порядку блока.

3.2.7. Система повинна підтримувати відновлення перерваного передавання через команду UPLOAD_RESUME без повторного надсилання вже прийнятої частини файла.

3.2.8. Користувач повинен мати можливість завантажувати файл із сервера через команду DOWNLOAD із подальшою локальною перевіркою SHA-256.

3.2.9. Користувач із належними правами повинен мати можливість видаляти файл через команду DELETE.

3.2.10. Адміністративний вебсервіс повинен надавати доступ до відомостей про файли, користувачів, сесії, передачі та журнали подій.

3.2.11. Система повинна обробляти невідомі команди, некоректні JSON-заголовки, неправильні токени, запити до відсутніх файлів і відмови в доступі без аварійного завершення серверного процесу.

3.3. Вимоги до складу та архітектури

3.3.1. Архітектура системи повинна містити клієнтський рівень, серверний рівень, адміністративний рівень і рівень зберігання даних.

3.3.2. Серверна частина повинна включати модуль приймання TCP-з'єднань, диспетчер команд, менеджер сесій, модуль файлових операцій, модуль контролю цілісності, модуль роботи з метаданими та модуль журналювання.

3.3.3. Механізм багатопоточності повинен бути реалізований із використанням керованого пулу потоків, зокрема ThreadPoolExecutor, для відокремлення приймання підключень від обробки клієнтських сесій.

3.3.4. Спільні ресурси, зокрема база метаданих, тимчасові файли, сесії передавання та журнал подій, повинні захищатися засобами синхронізації.

3.3.5. База метаданих SQLite повинна зберігати відомості про користувачів, файли, операції передавання, статуси сесій, контрольні суми та часові мітки.

3.3.6. Файлове сховище повинно відокремлювати завершені файли від тимчасових файлів, які створюються під час незавершеного або відновлюваного передавання.

3.3.7. Адміністративний вебсервіс повинен функціонувати окремо від TCP-каналу передавання файлів і взаємодіяти із серверною логікою через службові механізми доступу до стану системи.

3.4. Вимоги до якості та надійності

3.4.1. Система повинна стабільно працювати під час стандартних сценаріїв передавання, отримання, перегляду та видалення файлів.

3.4.2. Сервер не повинен аварійно завершувати роботу у разі надсилання помилкових команд, некоректних параметрів, неправильних токенів або запитів до відсутніх об'єктів.

3.4.3. Переданий файл повинен розміщуватися в основному сховищі лише після успішної перевірки фактичного розміру та контрольної суми SHA-256.

3.4.4. Тимчасове сховище повинно запобігати потраплянню неповністю переданих або пошкоджених файлів до основної директорії.

3.4.5. Хешування паролів повинно виконуватися без збереження паролів у відкритому вигляді.

3.4.6. TLS-режим повинен використовуватися для захисту транспортного каналу в тестовому або налаштованому середовищі.

3.4.7. Імена файлів повинні очищуватися перед записом у сховище для зменшення ризику path traversal та некоректного доступу до файлової системи.

3.4.8. Журналювання повинно забезпечувати можливість відтворення послідовності дій під час діагностики помилок і перевірки результатів тестування.

3.5. Вимоги до експлуатації

3.5.1. Для експлуатації системи необхідне середовище виконання Python, доступ до мережевого інтерфейсу TCP/IP, файлової системи та каталогу збереження переданих файлів.

3.5.2. У разі контейнеризованого запуску повинні бути налаштовані контейнери серверної частини, адміністративного вебсервісу, база метаданих, змонтований файловий том і конфігураційні параметри портів.

3.5.3. Адміністратор повинен мати доступ до вебінтерфейсу або адміністративного API для контролю стану системи, перегляду журналів і аналізу результатів передавання.

3.5.4. Користувач повинен запускати клієнтський застосунок із коректними параметрами підключення до сервера, обліковими даними та шляхом до локальних файлів.

3.5.5. Перед експлуатацією необхідно перевірити наявність прав доступу до директорій збереження, коректність конфігурації портів, доступність бази метаданих і працездатність мережевого з'єднання.

3.6. Вимоги до тестування

3.6.1. Метод тестування: функціональне, інтеграційне, навантажувальне та безпекове тестування клієнт-серверної системи обміну файлами.

3.6.2. Перевірка повинна охоплювати синтаксичну коректність Python-модулів, роботу основних команд протоколу, передавання файлів блоками, механізм ACK/NACK, відновлення передавання, автентифікацію, рольове розмежування доступу, журналювання та стійкість до некоректних запитів.

3.6.3. Тестування повинно виконуватися в локальному або контейнеризованому середовищі з імітацією типових і граничних сценаріїв використання системи.

Опис умов тестування:

Таблиця 3.6.3

Конфігурація системи	Умови використання	Очікувані результати тестування
Локальне або Docker Compose-середовище: файловий сервер, адміністративний сервіс, SQLite, файловий том	Один клієнт виконує LOGIN, PING, UPLOAD_INIT, CHUNK, UPLOAD_FINISH, LIST, DOWNLOAD і QUIT	Файл успішно передається й отримується, метадані створюються, контрольні суми SHA-256 збігаються
Файловий сервер із ThreadPoolExecutor	Три паралельні клієнти одночасно передають або отримують файли	Усі операції завершуються без блокування основного циклу сервера та без аварійного завершення процесу
Сервер і клієнтський застосунок	Передавання переривається після часткового завантаження, після чого виконується UPLOAD_RESUME	Передавання продовжується з визначеного зміщення без повторного надсилання всього файла
Модуль автентифікації та рольового доступу	Виконується вхід із коректними й некоректними даними; користувач запитує файл, до якого не має прав	Коректний користувач отримує токен, некоректні дані й заборонені дії відхиляються службовими відповідями
Модуль контролю цілісності	Надсилаються блоки з правильним і помилковим значенням SHA-256	Для коректного блока повертається ACK, для некоректного блока повертається NACK
Серверна частина та диспетчер команд	Надсилаються невідомі команди, неправильні JSON-	Сервер формує повідомлення про помилку та продовжує

	заголовки, запити до відсутніх файлів	роботу
TLS-режим і журналювання	Виконується захищене підключення та типова операція передавання файла	Канал устанавлюється, події фіксуються в журналі, результат операції доступний для адміністративного перегляду

3.6.4. Розробка вважається прийнятою після успішного проходження основних сценаріїв тестування, підтвердження коректності передавання файлів, працездатності багатопоточності, відповідності рольових обмежень і наявності журналів подій для перевірених операцій.

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЗ «ЛУГАНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА»

Факультет технологій та інформаційних систем

(назва факультету, інституту)

Інформаційних технологій та систем

(назва кафедри)

Пояснювальна записка

до дипломного проєкту (роботи)

БАКАЛАВРА

(освітньо-кваліфікаційний рівень)

на тему: **РОЗРОБКА КЛІЄНТ-СЕРВЕРНОЇ СИСТЕМИ ОБМІНУ**
ФАЙЛАМИ З ПІДТРИМКОЮ БАГАТОПОТОЧНОСТІ ЗАСОБАМИ
PYTHON

Виконав: студент 4 курсу, групи ____

напряму підготовки (спеціальності)

121 «Інженерія програмного забезпечення»

(шифр і назва напряму підготовки, спеціальності)

Яценко М.С.

(прізвище та ініціали)

Керівник

Переяславська С.О.

(прізвище та ініціали)

Рецензент

Козуб Ю. Г.

(прізвище та ініціали)

Лубни – 2026 року

ЗМІСТ

ВСТУП.....	3
РОЗДІЛ 1. АНАЛІТИЧНИЙ ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ.....	6
1.1 Клієнт-серверна архітектура: засади побудови та варіанти реалізації ..	6
1.2 Типові моделі взаємодії клієнта і сервера в системах передачі даних ..	7
1.3 Мережеві протоколи передавання файлів: функціональні можливості та обмеження.....	10
1.4 Багатопоточність і паралельна обробка: базові поняття та підходи	13
1.5 Огляд наявних рішень для обміну файлами: порівняльна характеристика.....	15
Висновки до розділу 1	17
РОЗДІЛ 2. ПРОЄКТУВАННЯ СИСТЕМИ	18
2.1. Загальна архітектура клієнт-серверної системи	18
2.2. Проєктування структури взаємодії компонентів.....	21
2.3. Застосування протоколів обміну даними.....	25
2.4. Проєктування механізму багатопотокової обробки запитів	30
2.5. Проєктування структури збереження файлів та журналювання	34
2.6. Моделювання системи	38
Висновки до розділу 2	41
РОЗДІЛ 3. ПРОГРАМНА РЕАЛІЗАЦІЯ	42
3.1. Обґрунтування вибору мови Python та використаних бібліотек	42
3.2. Реалізація серверної частини	44
3.3. Реалізація клієнтської частини	47
3.4. Реалізація механізму багатопоточності.....	50
3.5. Забезпечення надійності та безпеки передачі даних	52
3.6. Тестування та аналіз продуктивності системи.....	55
Висновки до розділу 3	58
ВИСНОВКИ.....	59
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	62
ДОДАТОК А.....	62
Порівняльна характеристика підходів до конкурентної обробки в клієнт-серверних системах передавання файлів.....	67
ДОДАТОК Б	69
Результати тестування клієнт-серверної системи обміну файлами	69
ДОДАТОК В.....	71
ДОДАТОК Г	86

ВСТУП

Стрімке зростання обсягів цифрових даних у прикладних інформаційних системах супроводжується підвищенням вимог до швидкості, керованості та надійності передавання файлів між вузлами мережі. Передавання файлів є базовою операцією для резервного копіювання, синхронізації робочих матеріалів, розповсюдження програмних оновлень, обміну медіаданими та транспортування результатів обчислень у розподілених середовищах. У практичних сценаріях передавання відбувається паралельно для багатьох користувачів і процесів, а мережеві з'єднання характеризуються змінною пропускнуою здатністю, затримками, втратами пакетів і ризиками несанкціонованого доступу [17].

Метою бакалаврської роботи є розроблення клієнт-серверної системи обміну файлами з підтримкою багатопоточності засобами Python, орієнтованої на паралельне обслуговування запитів передавання та приймання файлів із забезпеченням узгодженості операцій, цілісності даних і керованого журналювання подій. Для досягнення поставленої мети передбачається виконання комплексу взаємопов'язаних завдань, що охоплюють: аналіз архітектурних принципів клієнт-серверних рішень для передавання даних; вибір способу організації протоколу прикладного рівня (структури повідомлень, підтверджень, службових команд) та визначення вимог до коректного відновлення сесій; проєктування механізмів конкурентної обробки, зокрема моделі керування потоками, синхронізації спільних ресурсів і розподілу запитів у серверній частині; визначення структури зберігання файлів і метаданих, підходів до ідентифікації переданих об'єктів та політики журналювання; програмну реалізацію клієнта й сервера на Python із використанням бібліотек для мережевої взаємодії та конкурентності; перевірку працездатності шляхом функціонального тестування, а також оцінювання продуктивності за метриками пропускнуої здатності, затримки відповіді, стабільності під навантаженням і споживання ресурсів [32].

Об'єктом дослідження є процес організації клієнт-серверного обміну файлами в комп'ютерних мережах із паралельним обслуговуванням множини клієнтських запитів. Предметом дослідження виступають архітектурні та алгоритмічні рішення, що визначають структуру прикладного протоколу, механізми багатопотокової обробки та синхронізації, способи забезпечення надійності й безпеки передавання, а також підходи до контролю цілісності даних і ведення журналів подій [32].

Методологічна основа роботи спирається на поєднання системного підходу до побудови програмних систем, методів програмної інженерії та інженерії мережових застосунків. На етапі аналітичного опрацювання застосовуються порівняльний аналіз архітектур і протоколів передавання даних, узагальнення технічних вимог та формалізація сценаріїв використання. Проєктування системи виконується із застосуванням моделювання, що дозволяє визначити структуру компонентів, зв'язки між ними та поведінку в типових і граничних ситуаціях, включно з паралельним обслуговуванням запитів, помилками мережі та конфліктами доступу до ресурсів. Для обґрунтування багатопоточності використовуються принципи конкурентного програмування: розподіл задач, застосування потокобезпечних структур даних, використання примітивів синхронізації, керування пулом потоків та ізоляція критичних секцій. Реалізаційна частина орієнтована на стандартні можливості Python для мережових з'єднань і паралельного виконання, а також на засоби забезпечення цілісності та конфіденційності (контрольні суми, перевірка повноти передавання, захищені канали зв'язку). Для інтерпретації результатів застосовується аналіз метрик продуктивності й ресурсомісткості, що забезпечує доказовість вибору параметрів конкурентної обробки та технічних компромісів [37].

Практична цінність результатів полягає у створенні придатного до використання прототипу клієнт-серверної системи обміну файлами, який демонструє повний цикл передавання даних у мережі з паралельною обробкою запитів. Реалізовані проєктні рішення можуть бути використані як основа для навчальних і лабораторних робіт з мережевого програмування та конкурентності, а

також як інженерний шаблон для невеликих локальних інфраструктур, де необхідні контрольоване передавання файлів, відтворюване журналювання й підвищена пропускна здатність при одночасній роботі багатьох клієнтів. У підсумку робота спрямована на отримання завершеного програмного рішення з обґрунтованою архітектурою, формалізованими вимогами та підтвердженою працездатністю в умовах паралельного навантаження [32].

РОЗДІЛ 1. АНАЛІТИЧНИЙ ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Клієнт-серверна архітектура: засади побудови та варіанти реалізації

Клієнт-серверна архітектура є моделлю організації розподіленої програмної системи, у якій клієнт ініціює запити та споживає сервіс, а сервер надає функціональність, керує даними й ресурсами та виконує обчислення. Такий поділ відповідальностей формує контракт взаємодії (протокол, формат повідомлень, правила обробки помилок) і забезпечує керованість розвитку системи за рахунок ізоляції клієнтської та серверної логіки [16].

Побудова клієнт-серверних систем спирається на централізацію сервісної функціональності на сервері, стандартизацію інтерфейсів доступу та керування станом сеансу. Для систем обміну файлами керування станом є критичним, оскільки передавання є тривалим, дані надходять фрагментами, а паралельний доступ до сховища створює ризики конфліктів і неконсистентних записів [16].

Розмежування клієнта і сервера конкретизується моделями «тонкого» та «товстого» клієнта. Тонкий клієнт зводить локальну логіку до мінімуму і покладається на сервер у більшості операцій, що полегшує супровід. Товстий клієнт переносить частину логіки (кешування, попередню обробку, буферизацію) на локальний бік, знижуючи мережеві затрати, але ускладнюючи узгодженість версій [23].

Варіанти реалізації архітектури відрізняються кількістю логічних шарів. Дворівнева схема передбачає прямий зв'язок клієнта з сервером, який одночасно виконує прикладну логіку й роботу зі сховищем; її слабким місцем є «вузькі місця» при зростанні навантаження. Трирівнева схема відокремлює представлення, прикладну логіку та рівень даних, підвищуючи керованість, масштабованість і безпеку [16].

Комунікаційні рішення найчастіше базуються на TCP як надійному транспорті для потоків байтів; застосування UDP потребує додаткових механізмів

упорядкування і відновлення втрат. На прикладному рівні використовують або власний протокол поверх TCP, або стандартні підходи на основі HTTP/HTTPS [37].

Керування станом сеансу прямо впливає на масштабованість. Повністю статлес-підхід є обмеженим для файлообміну через потребу зберігати контекст передачі (ідентифікатор, позиція, підтверджені блоки, тимчасові файли). Практично застосовується «м'який стан»: сервер тримає мінімальний контекст активних сесій із очищенням за тайм-аутами, а клієнт зберігає дані, необхідні для відновлення [17].

Безпека інтегрується в архітектуру через захищений канал (TLS або інші механізми), контроль цілісності (хеші/контрольні суми) і підзвітність через журналювання подій із фіксацією параметрів сесії та результатів операцій. Для реалізації на Python архітектурні принципи безпосередньо проявляються у виборі конкурентної моделі на сервері: схема «потік на клієнта» є простою, але ресурсомісткою при масових підключеннях, тоді як пул потоків і черга завдань забезпечують керований паралелізм і стабільніше використання ресурсів, що є критичним для сервера передавання файлів [29].

1.2 Типові моделі взаємодії клієнта і сервера в системах передачі даних

У системах передавання даних взаємодія клієнта і сервера описується через комунікаційний шаблон (interaction pattern), який фіксує ініціатора обміну, порядок повідомлень, правила керування станом сеансу та механізми підтвердження доставки. Вибір моделі взаємодії визначає не лише протокол прикладного рівня, а й вимоги до конкурентної обробки запитів на сервері, оскільки різні моделі генерують відмінні профілі навантаження: короткі атомарні запити, довгі потоки даних, подієві повідомлення або асинхронні підтвердження [13].

Найпоширенішою є синхронна модель «запит-відповідь» (request-response). Клієнт формує запит на операцію (наприклад, отримання переліку файлів, ініціація завантаження або запит на стан), сервер повертає відповідь з результатом або кодом

помилки. Перевага моделі -простота формальної специфікації та тестування: кожному запиту відповідає один детермінований результат, що зручно для протоколів керування (control plane) [15].

Для власне файлообміну типовою є потокова модель (streaming) у межах встановленого з'єднання (переважно TCP): після початкового службового узгодження (метадані файлу, розмір, хеш, режим запису) передавання відбувається як послідовність байтів або блоків фіксованого/змінного розміру. Потоковий режим добре узгоджується з багатопоточним сервером, оскільки кожен активний потік (або завдання в пулі потоків) може обслуговувати окреме з'єднання або окремий канал передавання, перекриваючи очікування вводу/виводу [37].

Модель блокового передавання з підтвердженнями (chunked transfer with acknowledgements) передбачає поділ файла на нумеровані фрагменти з контрольними сумами; сервер надсилає підтвердження приймання для кожного фрагмента, а повторна передача виконується лише для блоків, що не були підтверджені або не пройшли перевірку цілісності. Такий підхід створює передумови для відновлення передавання після розриву з'єднання (resume), оскільки стан прогресу фіксується як множина прийнятих блоків [19].

Коли потрібна двонапрямна взаємодія з ініціативою сервера (наприклад, сповіщення про готовність файлу, завершення обробки, зміну прав доступу), застосовують подієві моделі: «публікація-підписка» (publish-subscribe) або двосторонній канал (WebSocket/подібні механізми поверх TCP). У publish-subscribe клієнти підписуються на теми, а сервер публікує події; клієнт може отримувати повідомлення без постійного опитування [34].

Асинхронна модель «команда-ідентифікатор-результат» (asynchronous job pattern) застосовується, коли операція є тривалою або ресурсоємною: клієнт надсилає команду, сервер повертає ідентифікатор завдання, а результат отримується пізніше (опитуванням або через callback/подію). Для файлообміну цей шаблон доречний у сценаріях із проміжною обробкою файлу (сканування, шифрування, індексація, розпакування), коли передавання є лише частиною конвеєра [38].

Порівняльну характеристику найуживаніших моделей взаємодії наведено (Табл. 1.1).

Таблиця 1.1 – Типові моделі взаємодії клієнта і сервера в системах передавання даних

Модель взаємодії	Типовий транспорт/канал	Становість сеансу	Переваги з позицій передавання даних	Обмеження та ризики	Типова придатність для обміну файлами
Запит-відповідь (request-response)	TCP/HTTP, RPC	Часто статлес або короткоживучий стан	Проста специфікація; детерміновані відповіді; зручна для команд керування та метаданих	Неефективна для великих даних без розбиття; ризик таймаутів; накладні витрати на багато запитів	Команди протоколу (автентифікація, список файлів, створення сесії, статус)
Потокове передавання (streaming)	TCP-сесія	Станова (контекст передавання)	Ефективна для великих файлів; природна буферизація; добра сумісність з конкурентним обслуговуванням підключень	Потрібні правила фреймінгу й завершення; складніша обробка помилок; важливий контроль ресурсів	Основний режим upload/download у межах одного з'єднання
Блокове передавання з підтвердженнями (chunk + ACK)	TCP або прикладний протокол поверх TCP	Станова (прогрес за блоками)	Відновлення після збоїв; повторна передача лише проблемних блоків; контроль цілісності на рівні фрагментів	Вища складність протоколу; потреба синхронізації доступу до прогресу й тимчасових файлів	Надійне передавання; resume; паралельні завантаження великих файлів
Подієва модель (publish-subscribe)	Message broker, WebSocket-подібні канали	Станова (підписки, канали)	Сервер ініціює сповіщення; зменшення опитування;	Додаткова інфраструктура або складніша реалізація;	Сповіщення про завершення/помилки; індикація прогресу;

			оперативні повідомлення про стан	необхідність гарантій доставки подій	керування сесіями
Асинхронне завдання (job: submit-id-result)	HTTP/RPC + черга робіт	Станова (ідентифікатор і статус)	Не блокує клієнта на довгих операціях; придатна для конвеєрів обробки; кероване планування ресурсів	Потрібне сховище стану; узгодження повторів; складніша діагностика без якісного журналу	Передавання + постобробка (сканування, шифрування, індексація), пакетні операції

У межах розроблюваної системи обміну файлами з багатопоточним сервером практично доцільним є комбінування моделей: запит-відповідь для службових команд, потокове або блокове передавання для даних файлу та асинхронний шаблон для операцій, що виходять за межі «передати/отримати» і потребують часу. Таке поєднання дозволяє розділити «керування» і «трафік даних», формалізувати життєвий цикл сесії, а також підтримати конкурентну обробку великої кількості клієнтських підключень без втрати керованості станів і без деградації на операціях вводу/виводу [37].

1.3 Мережеві протоколи передавання файлів: функціональні можливості та обмеження

Мережеві протоколи передавання файлів формують клас прикладних механізмів, які визначають правила встановлення з'єднання, автентифікації сторін, опису файлів і каталогів, передачі вмісту та завершення сеансу із фіксацією результату. У технологічному сенсі протокол задає семантику операцій читання й запису (завантаження, вивантаження, перейменування, видалення, створення каталогів), формат службових повідомлень, коди станів, правила повторів і часові обмеження [22].

Історично базовим протоколом для файлового обміну виступає FTP. Його функціональні можливості охоплюють автентифікацію користувача, навігацію файловою ієрархією, перелік каталогів і передавання файлів у двох режимах (active/passive), що розділяє канал керування і канал даних. Така організація є зручною для реалізації команд керування, проте створює експлуатаційні обмеження: використання окремих портів для даних ускладнює роботу через NAT і міжмережіві екрани, вимагає додаткового узгодження політик безпеки та підвищує ризик помилок конфігурації [22].

Для усунення проблеми конфіденційності у FTP застосовують надбудову TLS, відому як FTPS. Вона додає шифрування каналу, підтримує сертифікатну ідентифікацію та знижує ризик перехоплення даних [38].

Альтернативний напрям розвитку - використання захищених протоколів на основі SSH, насамперед SFTP та SCP. SFTP є підсистемою SSH і забезпечує передачу файлів у межах одного захищеного каналу, зазвичай через стандартний порт 22. Його функціональність охоплює не лише передавання вмісту, а й операції керування файловою системою: перегляд каталогів, створення і видалення об'єктів, зміну атрибутів доступу, що робить його придатним для повноцінного віддаленого файлового управління [26].

SCP також базується на SSH і призначений для копіювання файлів. Його концептуальна простота є перевагою для побутових сценаріїв, однак у порівнянні з SFTP протокол менш гнучкий щодо керування каталогами, має обмежені можливості інтерактивного управління й часто поступається у керованості відновлення передавання, оскільки орієнтований на «одноразову» операцію копіювання [26].

У сучасних мережах значний сегмент передавання файлів реалізується на основі HTTP/HTTPS. Первинно HTTP проєктувався як протокол доступу до гіпертекстових ресурсів, проте його універсальність і наявність усталеної інфраструктури (проксі, кеші, балансувальники, TLS-термінація) зробили його де-факто стандартом для транспортування файлів. Для завантаження файлів використовуються методи GET, а для вивантаження - POST/PUT, при цьому HTTPS

забезпечує конфіденційність та автентичність каналу через TLS. Функціональна перевага HTTP -зручна інтеграція з корпоративними мережами та мінімальні бар'єри проходження через міжмережеві екрани, оскільки порти 80/443 зазвичай доступні. Для великих вивантажень також виникає потреба в протокольних домовленостях щодо блокового надсилання, ідентифікації частин, підтверджень і контролю цілісності, оскільки стандартні HTTP-механізми не задають цього як обов'язкової семантики [38].

Розширенням HTTP у бік «файлового» управління є WebDAV. Воно додає методи і заголовки для операцій із колекціями ресурсів, блокування (locking), версіювання в окремих профілях, що робить WebDAV ближчим до віддаленої файлової системи. Його застосування доцільне, коли потрібна взаємодія із каталогами та метаданими поверх стандартної HTTP-інфраструктури [9].

Паралельно з протоколами прикладного рівня існують протоколи мережеских файлових систем, серед яких поширені SMB/CIFS та NFS. Вони реалізують віддалений доступ до файлів як до ресурсів, інтегрованих у файлову ієрархію операційної системи. Функціональні можливості включають роботу з каталогами, атрибутами, блокуванням файлів, узгодженістю кешів, що створює користувацьке враження «мережевого диска» [17].

Протоколи та алгоритмічні підходи, зорієнтовані на синхронізацію, зокрема rsync, формують спеціалізований клас рішень, у якому пріоритетом є мінімізація переданого обсягу за рахунок обміну лише відмінностями між локальною та віддаленою версіями файлів. Його центральна ідея полягає в передаванні не всього файлу, а лише відмінностей між локальною та віддаленою версіями на основі блокових контрольних сум і «сильних» хешів. Це зменшує трафік і час синхронізації при малих змінах, що є типовим для резервного копіювання та розгортання артефактів [19].

Узагальнюючи функціональні можливості протоколів передавання файлів, до базового ядра належать автентифікація, керування каталогами, передавання вмісту та інформування про результат операції. Для практичної придатності в багатоклієнтському середовищі додаються механізми відновлення передавання,

контроль цілісності (хеші, контрольні суми, валідація розміру), обмеження ресурсів (квоти, ліміти швидкості), а також журналювання для аудиту й діагностики [32].

1.4 Багатопоточність і паралельна обробка: базові поняття та підходи

Багатопоточність у програмних системах визначається як організація виконання, за якої в межах одного процесу одночасно (або квазіодночасно) працюють кілька потоків керування, що розділяють адресний простір і можуть мати спільний доступ до пам'яті, файлових дескрипторів, мережесокетів та інших ресурсів операційної системи. Паралельна обробка є ширшим поняттям і охоплює будь-які підходи, які дозволяють виконувати кілька обчислювальних або ввід/вивідних підзадач одночасно, включно з багатопоточністю, багатопроцесністю, асинхронним введенням/виведенням і подієвими моделями [34].

Виконання програм у сучасних ОС спирається на модель планування, де центральний процесор розподіляє час між потоками або процесами. Квазіпаралельність виникає навіть на одноядерних системах за рахунок швидкого перемикавання контексту, тоді як на багатоядерних системах можливий справжній паралелізм, коли різні потоки/процеси виконуються на різних ядрах одночасно [17].

Фундаментальною проблемою багатопоточності є узгодження доступу до спільних ресурсів. Якщо два потоки виконують операції читання/запису над спільним об'єктом без координації, виникає гонка даних (race condition), яка проявляється недетермінованістю результатів: одна і та сама послідовність зовнішніх подій може призводити до різних станів системи. Для систем обміну файлами типовими є гонки при записі в один і той самий файл, при оновленні метаданих прогресу передачі, при інкременті лічильників, веденні журналів подій, керуванні чергами завдань і пулами з'єднань [32].

Паралельна обробка в мережесерверах зазвичай описується через дві осі: модель конкурентності (threads/processes/async) і характер навантаження (CPU-

bound чи I/O-bound). Для передачі файлів домінує I/O-bound профіль: значна частка часу витрачається на очікування мережових операцій та дискового введення/виведення. За таких умов багатопоточність часто дає приріст не тому, що прискорює обчислення, а тому, що дозволяє перекривати очікування вводу/виводу: поки один потік блокується на прийманні пакета або записі блоку на диск, інший може обслуговувати інше з'єднання [37].

Операційно коректна багатопоточна реалізація передбачає контроль життєвого циклу потоків і ресурсів. Створення «потoku на кожного клієнта» інтуїтивно просте, але при великій кількості одночасних підключень створює значні накладні витрати на планування та пам'ять стеків, підвищує ризик виснаження дескрипторів і деградації через масове перемикання контексту. Більш керованим підходом є пул потоків, де кількість робітників обмежена, а вхідні запити або події надходять у чергу завдань [29].

Подієва (асинхронна) модель конкурентності розглядає паралельність як кооперативне перемикання між корутинами або обробниками подій у межах одного або кількох циклів подій. У цьому підході операції вводу/виводу виконуються неблокувально, а керування повертається в цикл подій до моменту готовності сокета/файлу. Тому на практиці для систем передачі файлів часто застосовують гібридний підхід: мережовий рівень і протокол обробляються асинхронно, а ресурсоємні або блокувальні операції виконуються у виділеному пулі потоків/процесів [34].

Конкурентна обробка потребує контролю двох типових відмовних режимів: взаємного блокування (deadlock), коли потоки циклічно очікують ресурси один одного, і голодування (starvation), коли окремі завдання систематично не отримують процесорного часу або доступу до потрібних ресурсів через пріоритети чи перевантаження. Взаємне блокування виникає, коли потоки утримують взаємозалежні блокування в різному порядку й очікують ресурси один одного. У файловому обміні це може проявлятися, коли один потік утримує блокування метаданих і очікує блокування журналу, а інший -навіпаки [32].

Систематизація підходів до конкурентної обробки в клієнт-серверних системах передавання файлів дає змогу зіставити потокову модель, пул потоків, багатопроцесність, асинхронний цикл подій, гібридну модель і акторний підхід за одиницею виконання, профілем ефективності, перевагами, обмеженнями та придатністю для файлового сервера. Узагальнене порівняння цих підходів винесено до додатків, оскільки таблиця має значний обсяг і виконує допоміжну аналітичну функцію (Додаток А, Табл. А.1) [29].

Висвітлені підходи утворюють концептуальну основу для обґрунтування реалізації багатопоточності в клієнт-серверній системі обміну файлами на Python. Подальше проектування вимагає фіксації цільового профілю навантаження (кількість одночасних клієнтів, розміри файлів, частка операцій читання/запису, потреба у постобробці), після чого модель конкурентності конкретизується як набір технічних рішень: спосіб приймання з'єднань, порядок обробки команд протоколу, організація буферів, політики синхронізації доступу до файлового сховища та правила журналювання для відтворюваності й діагностики помилок [32].

1.5 Огляд наявних рішень для обміну файлами: порівняльна характеристика

Наявні рішення для обміну файлами доцільно зіставляти за сукупністю прикладних і експлуатаційних ознак: наявністю захищеного каналу, підтримкою керування доступом, можливістю відновлення перерваної передачі, засобами аудиту та журналювання, поведінкою в мережах із NAT/файрволами, а також стійкістю до паралельного навантаження. У практиці ці параметри визначають, чи є рішення придатним для багатоклієнтської роботи без деградації якості сервісу та без підвищення операційних ризиків [32].

Класичні протокольні рішення представлені FTP/FTPS і SFTP. FTP забезпечує базовий набір файлових операцій і є історично поширеним, однак у базовому вигляді не гарантує конфіденційність та цілісність трафіку, а також ускладнює експлуатацію через схему з окремим каналом даних і динамічними портами [19].

Другий поширений підхід -передавання файлів через HTTP/HTTPS. Його перевага полягає в універсальності та високій досяжності сервісу, оскільки веб-трафік зазвичай дозволений у корпоративних мережах, а TLS у складі HTTPS забезпечує захист каналу [38].

Інфраструктурні рішення на основі мережових файлових систем (SMB/CIFS, NFS) реалізують модель «віддаленого диска», забезпечуючи прозору інтеграцію з ОС і прикладними програмами. Їхня сильна сторона -користувацька зручність і готові механізми роботи з каталогами та правами [17].

Хмарні сервіси із синхронізацією, версіюванням та механізмами спільного доступу формують самостійний клас рішень для обміну файлами. Вони надають широкий функціонал і знижують потребу в локальному адмініструванні, однак зменшують контроль над протоколом, журналюванням і політиками зберігання, а також залежать від зовнішньої платформи й комплаєнс-вимог. Для навчальної реалізації їх доречно розглядати як еталон користувацьких можливостей, але не як технічний шаблон серверної частини [32].

У підсумку найбільш релевантними для проектованої системи є концепції, що прямо впливають на коректність і продуктивність: робота в межах одного прогнозованого каналу зв'язку, підтримка шифрування або принаймні незалежного контролю цілісності (хеш/контрольна сума), керований життєвий цикл передачі (ініціація, підтвердження, завершення), відтворюваність через журналювання та здатність стабільно обслуговувати паралельні підключення. Саме ці характеристики формують основу для вибору власного прикладного протоколу й моделі багатопотокової серверної обробки в реалізації засобами Python [32].

Висновки до розділу 1

Клієнт-серверна архітектура є доцільною основою для побудови системи обміну файлами, оскільки забезпечує чітке розмежування функцій між клієнтською частиною, яка ініціює запити та виконує локальні операції користувача, і серверною частиною, яка централізовано керує прийманням, збереженням, перевіркою та передаванням файлів. Така модель дає змогу організувати контрольований доступ до файлових ресурсів, підтримувати єдине сховище даних, фіксувати метадані та забезпечувати узгоджене обслуговування кількох клієнтів у межах спільної мережевої інфраструктури [23].

Аналіз моделей взаємодії клієнта і сервера засвідчив, що для задачі файлового обміну найбільш придатним є поєднання службової моделі «запит-відповідь» і потокового або блокового передавання даних. Службові команди зручно використовувати для автентифікації, запиту списку файлів, отримання метаданих і завершення сесії, тоді як передавання вмісту файла потребує блокової організації, фреймінгу повідомлень, контролю розміру й перевірки цілісності [19].

Порівняння наявних протоколів і систем обміну файлами показало, що FTP, FTPS, SFTP, HTTP/HTTPS, WebDAV, SMB/NFS і синхронізаційні рішення мають різні переваги та обмеження щодо безпеки, керування станом, роботи через міжмережеві екрани, відновлення передачі та підтримки метаданих. Для реалізації навчального прототипу найбільш обґрунтованим є не відтворення складного промислового протоколу, а створення власного прикладного протоколу поверх TCP із підтримкою команд, блокового передавання, контролю цілісності, журналювання та багатопотокової обробки [32].

РОЗДІЛ 2. ПРОЄКТУВАННЯ СИСТЕМИ

2.1. Загальна архітектура клієнт-серверної системи

Загальна архітектура клієнт-серверної системи обміну файлами визначається як багаторівнева програмна структура, у межах якої функції передавання файлів, керування мережевими з'єднаннями, збереження даних, ведення метаданих і адміністративного контролю розмежовані між окремими компонентами. Така побудова забезпечує впорядковану взаємодію між вузлами [17].

На концептуальному рівні система складається з клієнтського застосунку, файлового сервера, адміністративного вебсервісу, файлового сховища, бази метаданих і підсистеми журналювання (Рис. 2.1). Клієнтський застосунок призначений для ініціювання запитів на передавання або отримання файлів, перегляду доступних об'єктів і завершення сесії [32].

Фізична структура системи ґрунтується на контейнеризованому розгортанні, у межах якого файловий сервер і адміністративний вебсервіс функціонують як окремі контейнери. Така організація дозволяє ізолювати залежності, стандартизувати середовище виконання та спростити процедури запуску, оновлення і тестування [5].

Архітектура файлового сервера базується на модульній внутрішній організації. Один модуль відповідає за приймання TCP-з'єднань і первинну обробку мережових пакетів, інший інтерпретує прикладні команди й визначає сценарій подальшої обробки, окремий модуль виконує файлові [25].

Клієнтський застосунок взаємодіє з сервером по TCP-з'єднанню через прикладний протокол, який визначає набір команд, формат службових повідомлень, структуру ідентифікації файлів і правила завершення сесій. На відміну від універсальних вебпротоколів, такий підхід забезпечує точне керування життєвим циклом передавання [25].

Особливого значення в загальній архітектурі набуває багатопотокова організація серверної частини. Одночасна робота кількох клієнтів у межах однієї системи передбачає наявність механізму паралельної обробки запитів, який унеможливорює блокування всієї системи під час виконання окремої операції. З цією метою серверна частина поділяє приймання з'єднань і виконання клієнтських запитів [16].

Файлове сховище становить окремий рівень архітектури й реалізується як область постійного збереження даних, змонтована до контейнера файлового сервера через Docker volume. У межах цієї підсистеми розрізняються файли, передавання яких завершено коректно, і тимчасові об'єкти, що перебувають у процесі запису або були збережені після переривання сесії [5].

Адміністративний вебсервіс функціонує як окремий шар системи, призначений для контролю та моніторингу. Його наявність не змінює основної логіки обміну файлами, оскільки передавання даних між користувачем і сервером відбувається через основний TCP-канал. Вебсервіс забезпечує перегляд переліку файлів, історії передач, активних сесій, журналів подій, а також виконання службових дій, пов'язаних із контролем стану системи [32].

Забезпечення цілісності даних інтегроване в загальну архітектуру на рівні кожної файлової операції. Передача супроводжується обчисленням контрольної суми, яка використовується для перевірки відповідності отриманого файлу вихідному об'єкту. Сервер фіксує успішне завершення операції лише після повної перевірки цілісності, що унеможливорює розміщення в основному сховищі пошкоджених або неповністю переданих даних [19].

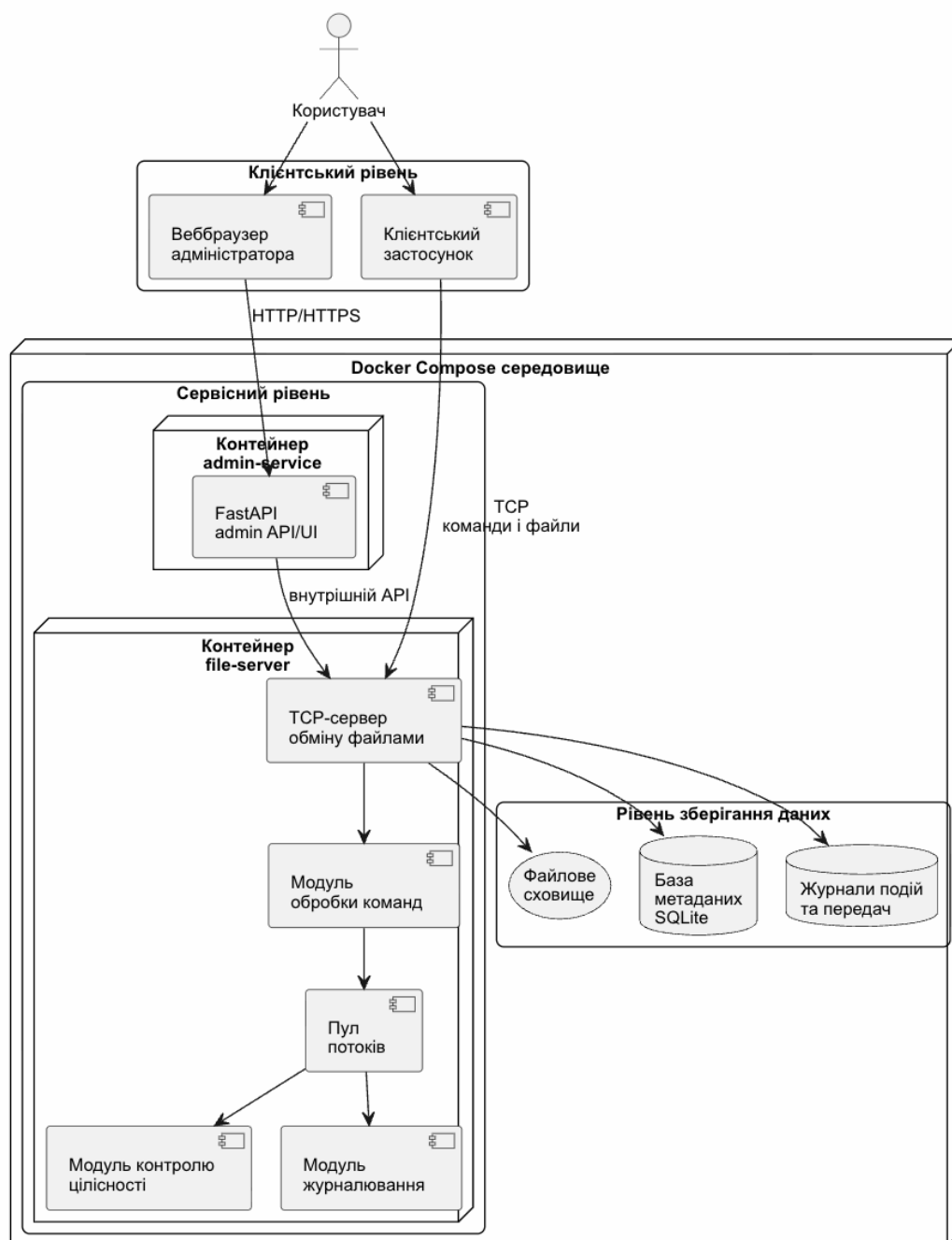


Рис. 2.1 – Загальна архітектура клієнт-серверної системи обміну файлами

Отже, загальна архітектура клієнт-серверної системи обміну файлами визначається як контейнеризована багатокомпонентна структура з централізованим файловим сервером, окремим адміністративним вебсервісом, прикладним протоколом поверх TCP, багатопотоковою серверною обробкою, файловим сховищем, базою метаданих і системою журналювання. Така побудова забезпечує

логічну впорядкованість взаємодії між компонентами, підтримує одночасну роботу кількох [32].

2.2. Проєктування структури взаємодії компонентів

Проєктування структури взаємодії компонентів клієнт-серверної системи обміну файлами ґрунтується на принципі функціонального розмежування, за якого кожний елемент системи виконує чітко визначену роль у загальному циклі обробки запиту. Така організація є необхідною передумовою для забезпечення керованої багатопотокової роботи, коректного передавання [23].

На верхньому рівні взаємодія компонентів охоплює три основні функціональні зони: клієнтську, серверну та адміністративну (Рис. 2.2). Клієнтська зона представлена застосунком, через який користувач ініціює встановлення з'єднання з файловим сервером, надсилає команди прикладного протоколу, передає файли або отримує їх із серверного сховища [13].

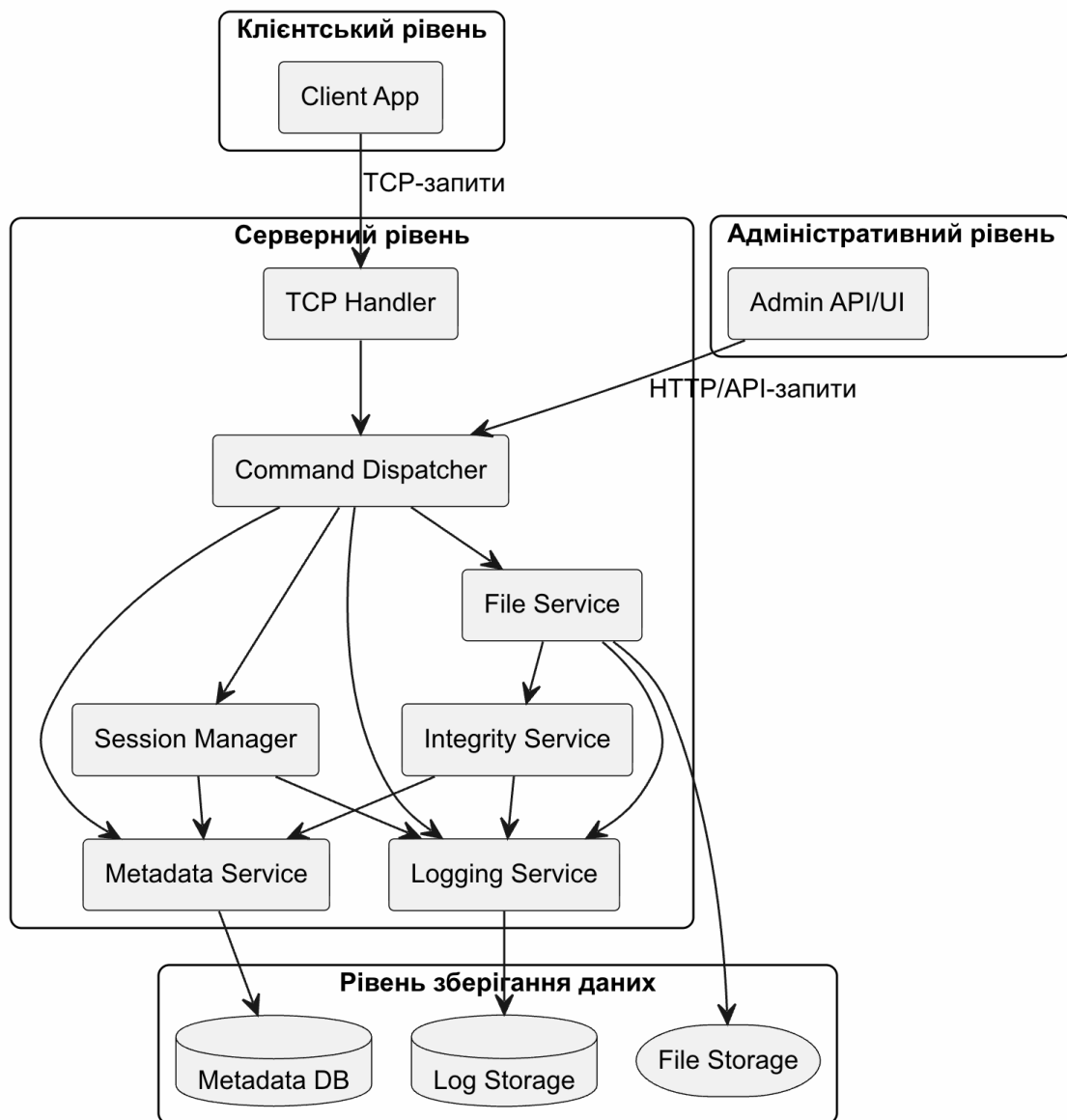


Рис. 2.2 – UML-діаграма компонентів системи обміну файлами

Клієнтський застосунок доцільно розглядати як сукупність кількох внутрішніх модулів, які забезпечують формування зовнішньої поведінки системи. Перший модуль відповідає за встановлення й підтримання TCP-з'єднання із сервером, виконуючи ініціалізацію сесії, відкриття сокета, передавання службових повідомлень і контроль стану каналу зв'язку [34].

Центральне місце у структурі взаємодії належить файловому серверу, оскільки саме через нього проходять усі прикладні дії, пов'язані з передаванням, зберіганням і видачею файлів. На рівні внутрішньої композиції цей сервер містить модуль приймання з'єднань, диспетчер команд, менеджер сесій, модуль файлових [13].

Диспетчер команд є компонентом, що перетворює отриманий потік службових повідомлень на набір прикладних дій. Він аналізує тип команди, перевіряє її коректність, визначає, який саме модуль повинен виконати операцію, і формує відповідний сценарій обробки [13].

Менеджер сесій є окремим логічним компонентом, що відповідає за створення, ідентифікацію, супровід і завершення взаємодії з кожним підключеним клієнтом. У межах його функцій фіксуються ідентифікатор сесії, часові параметри, поточний статус, перелік активних операцій і допоміжні відомості, потрібні для відновлення стану у випадку розриву з'єднання [13].

Виконання реальних дій із файлами покладається на модуль файлових операцій. Саме він створює тимчасові файли, записує блоки отриманих даних, зчитує об'єкти зі сховища для подальшого передавання клієнту, переміщує завершені файли в постійний каталог і видаляє ресурси за відповідним запитом. Його взаємодія з іншими компонентами має двосторонній характер [13].

Модуль перевірки цілісності входить до структури взаємодії як гарантія коректності переданих даних. Його роль не обмежується обчисленням контрольної суми. Таким чином, у структурі взаємодії саме він замикає критичний цикл підтвердження завершеного передавання [19].

Модуль роботи з метаданими виконує функцію внутрішнього інформаційного реєстру системи. Він забезпечує створення та оновлення записів про файли, передачі, сесії, статуси, часові мітки, шляхи зберігання і результати перевірки цілісності [19].

Модуль журналювання забезпечує відтворюваність перебігу всіх операцій у системі. На відміну від метаданих, які переважно описують поточний або підсумковий стан, журнал фіксує саму послідовність подій: встановлення з'єднання, отримання команди, початок завантаження, приймання [32].

Адміністративний вебсервіс взаємодіє з файловим сервером не як паралельне сховище логіки, а як окремий керувальний компонент. Він отримує доступ до відомостей про систему через службові запити до файлового сервера й відображає ці дані у вебінтерфейсі адміністратора [13].

Формалізація внутрішньої організації системи потребує виокремлення основних сутностей, які беруть участь у встановленні сесії, обробці команд, передаванні файлів, перевірці цілісності, збереженні метаданих і фіксації подій (Рис. 2.3). Таке подання дає змогу відобразити не лише перелік об'єктів, а й характер зв'язків між ними, включно з ініціюванням сесій клієнтом, супроводом передачі [19].

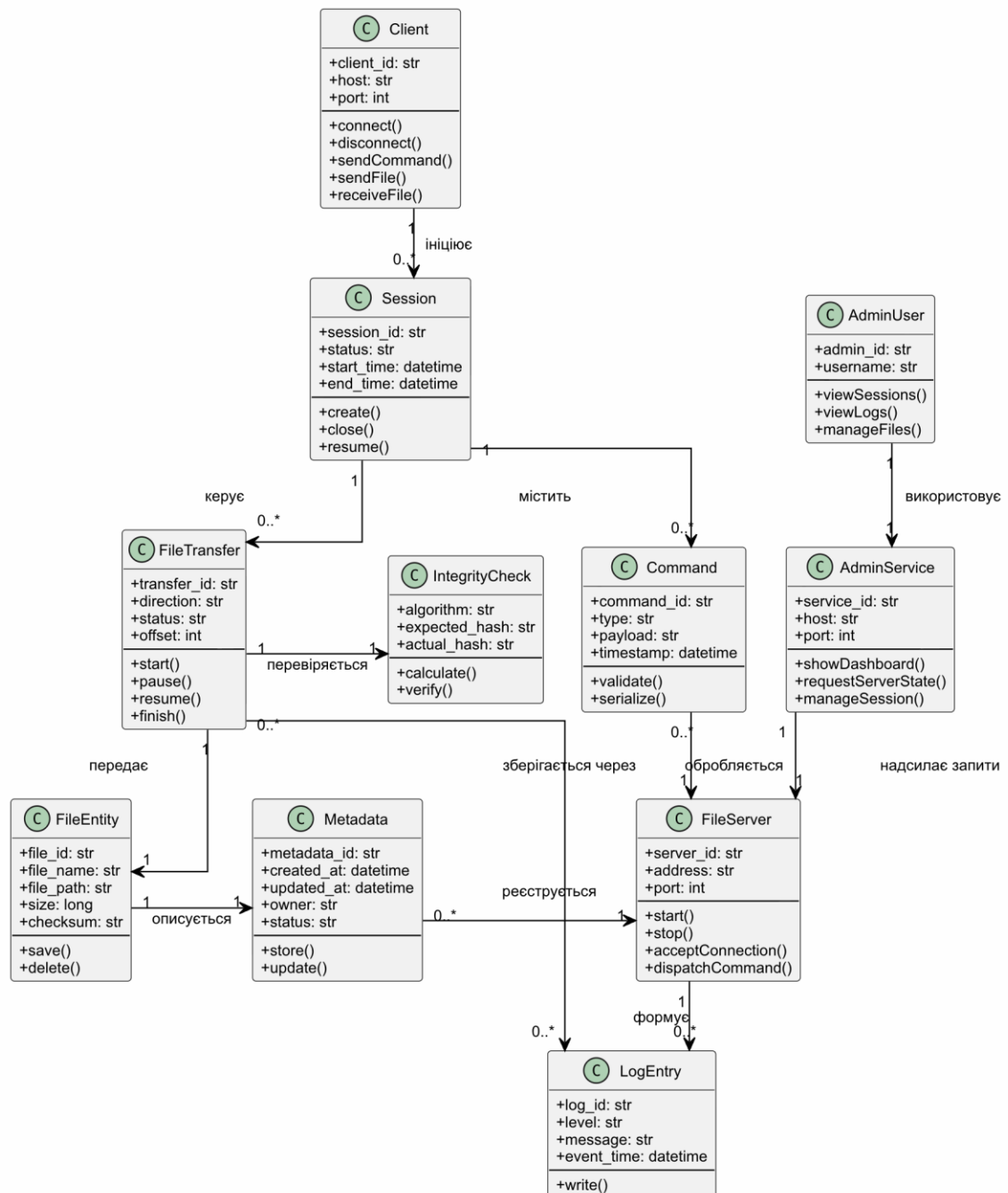


Рис. 2.3 – UML-діаграма класів основних сутностей системи обміну файлами

Отже, структура взаємодії компонентів клієнт-серверної системи обміну файлами формується як впорядкована мережа функціонально спеціалізованих модулів, у межах якої клієнтський застосунок ініціює запити, файловий сервер здійснює їх багатопотокову прикладну обробку, модулі сесій, файлових операцій, метаданих, перевірки цілісності та журналювання забезпечують виконання й контроль усіх дій, а адміністративний вебсервіс надає засоби службового моніторингу та керування. Саме така структура створює логічно завершену модель [32].

2.3. Застосування протоколів обміну даними

Застосування протоколів обміну даними в клієнт-серверній системі обміну файлами визначає не лише технічний спосіб передавання інформації між вузлами, а й логіку встановлення сесії, формування команд, структуру службових повідомлень, порядок передавання блоків файла, механізми підтвердження операцій і правила завершення взаємодії. Для системи, орієнтованої на багатопотокову обробку запитів, контроль стану передачі та роботу з файлами значного [23].

Вибір TCP як базового транспортного механізму зумовлений тим, що передавання файлів потребує гарантованої доставки байтового потоку, збереження порядку сегментів і вбудованих механізмів повторної передачі втрачених пакетів. На відміну від безз'єдневих транспортних [25].

Разом із тим застосування лише транспортного протоколу не забезпечує завершеної моделі обміну даними, оскільки він не задає ні семантики команд, ні способу розмежування службових повідомлень і файлового вмісту, ні формату ідентифікації операцій. З цієї причини в системі [23].

Структура протоколу організується у вигляді двох взаємопов'язаних рівнів: рівня службових повідомлень і рівня передавання файлових блоків. На службовому

рівні клієнт і сервер обмінюються командами, які описують тип дії, параметри файла, часовий контекст, статус сесії, ідентифікатор передачі, обсяг даних і додаткові службові ознаки [13].

Для коректного розпізнавання меж службових повідомлень у TCP-поточі застосовується механізм попереднього зазначення довжини заголовка або повідомлення. Це означає, що перед власне JSON-пакетом передається коротке службове поле фіксованого розміру, яке містить довжину наступного блоку даних [31].

Набір прикладних команд формується відповідно до функціональних задач системи. На базовому рівні використовуються команди перевірки доступності сервера, отримання списку доступних файлів, ініціації завантаження файла на сервер, передавання чергового блока, завершення [24].

Операція передавання файла на сервер починається з ініціальної команди, яка повідомляє серверу про намір клієнта розпочати завантаження (Рис. 2.4). У службовому заголовку такої команди вказуються ім'я файла, його очікуваний розмір, контрольна сума, напрям передавання та допоміжні атрибути сесії [19].

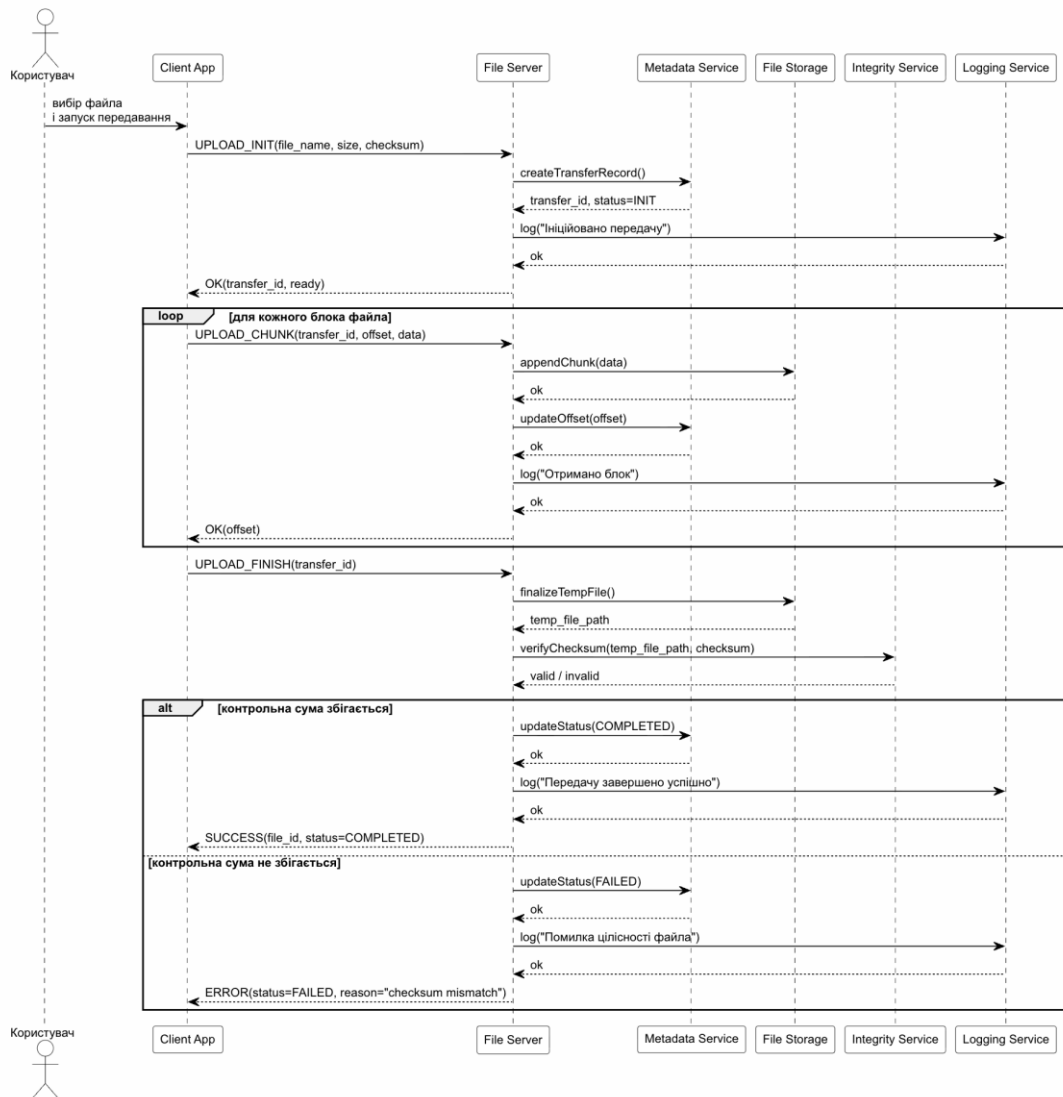


Рис. 2.4 – UML-діаграма послідовності передавання файла на сервер

Отримання файла із серверного сховища організується за схожим принципом, але з інверсією напрямку передавання (Рис. 2.5). Клієнт надсилає службовий запит на отримання конкретного файла, після чого сервер перевіряє наявність такого ресурсу, актуальність його метаданих і доступність для читання [13].

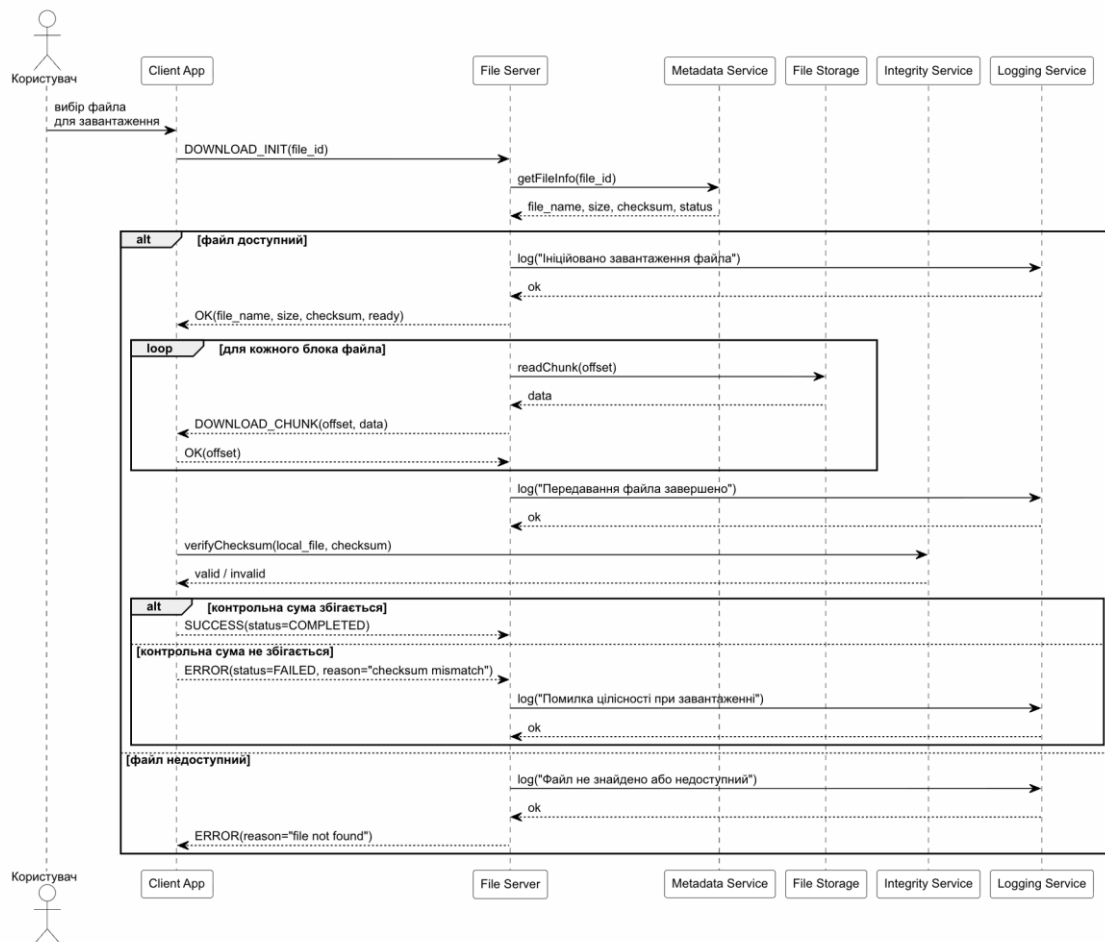


Рис. 2.5 – UML-діаграма послідовності отримання файлу із сервера

Особливе значення для системи має підтримка перерваних передач. У мережевому середовищі з нестабільним з'єднанням або при великому обсязі файлу повне перезапускання операції після втрати каналу є неефективним і створює надмірне навантаження на сервер [17].

У межах протоколу важливу роль відіграють повідомлення про результат виконання команди. Сервер повинен повертати уніфіковані відповіді, які містять статус операції, код результату та пояснювальний текст або службові параметри [23].

Протокольна модель також охоплює взаємодію з адміністративним вебсервісом. Незважаючи на те, що адміністрування системи реалізується через HTTP-орієнтований інтерфейс, сам вебсервіс не функціонує як окреме сховище логіки, а звертається до файлового сервера з використанням службового набору адміністративних запитів [9].

Додатковим протокольним механізмом є контроль цілісності даних. Оскільки TCP гарантує доставку потоку байтів, але не замінює прикладної перевірки коректності повністю зібраного файла, у системі використовується схема порівняння контрольної суми. Клієнт до початку передавання формує хеш-значення файла і надсилає його в службовому повідомленні [19].

Таблиця 2.1 – Основні команди прикладного протоколу обміну даними

Команда	Призначення	Основні параметри	Відповідь сервера
PING	Перевірка доступності сервера та працездатності з'єднання	session_id (за наявності)	OK, службовий статус сервера
LIST	Отримання переліку файлів, доступних у серверному сховищі	session_id, за потреби фільтр або пагінація	OK, список файлів і метаданих
UPLOAD_INIT	Ініціація передавання файла на сервер	file_name, size, checksum, session_id	OK, transfer_id, готовність до приймання блоків
UPLOAD_CHUNK	Передавання чергового блока файла на сервер	transfer_id, offset, chunk_size, data	OK, підтвердження приймання блока
UPLOAD_FINISH	Завершення операції завантаження після передавання всіх блоків	transfer_id	SUCCESS або ERROR, підсумковий статус передачі
DOWNLOAD_INIT	Ініціація отримання файла із серверного сховища	file_id або file_name, session_id	OK, атрибути файла, розмір, checksum, готовність до передавання
DOWNLOAD_CHUNK	Передавання сервером чергового блока файла клієнту	transfer_id, offset, chunk_size	Блок даних або OK із параметрами наступного блока
RESUME	Відновлення перерваної передачі з певного зміщення	transfer_id, offset, session_id	OK, підтвердження відновлення або ERROR
DELETE	Видалення файла із серверного сховища	file_id або file_name, session_id	SUCCESS або ERROR, результат операції
INFO	Отримання детальної інформації про файл	file_id або file_name	OK, метадані файла
OK	Уніфіковане підтвердження успішного виконання проміжної або службової дії	status, службові параметри	Службове повідомлення про успішний стан
SUCCESS	Підтвердження повного успішного завершення операції	status, file_id або transfer_id	Підсумкове повідомлення про завершення

ERROR	Повідомлення про помилку під час виконання команди або передачі	status, reason, код помилки	Пояснення причини відмови або збою
-------	---	-----------------------------	------------------------------------

Застосування протоколів обміну даними в проєктованій системі базується на поєднанні надійного транспортного рівня ТСП із власним прикладним командно-блоковим протоколом, який визначає правила встановлення сесії, структуру службових повідомлень, порядок передавання файлів частинами, уніфікований формат відповідей, механізми відновлення передачі та контроль цілісності даних. Такий підхід забезпечує достатній рівень керованості для багатопотокової клієнт-серверної системи, підтримує роботу з великими [19].

2.4. Проєктування механізму багатопотокової обробки запитів

Проєктування механізму багатопотокової обробки запитів у клієнт-серверній системі обміну файлами визначається необхідністю одночасного обслуговування кількох клієнтських сесій без втрати керованості станом системи, без конфліктів доступу до спільних ресурсів і без суттєвого зростання часу очікування під час виконання файлових операцій. Для системи, у якій клієнти паралельно завантажують файли на сервер, отримують їх із серверного сховища, ініціюють повторні підключення після обриву зв'язку та звертаються до [13].

Основою обраної моделі є поєднання виділеного потоку приймання з'єднань, диспетчеризації вхідних подій та пулу робочих потоків, які виконують прикладні завдання. Така організація забезпечує розмежування трьох різних класів операцій. До першого класу належить короткотривале мережеве реагування: прослуховування порту, встановлення ТСП-з'єднання, приймання початкового пакета та реєстрація нової сесії [37].

У межах проєктованої системи головний потік сервера виконує функцію безперервного приймання нових підключень і первинного створення об'єктів сесій.

Після встановлення з'єднання він не переходить до повного обслуговування конкретного клієнта, а передає пов'язане із ним завдання до внутрішнього механізму диспетчеризації [13].

Центральним елементом механізму багатопоточності виступає пул робочих потоків. Його використання є доцільнішим, ніж створення окремого потоку для кожного клієнта без обмеження їх кількості [37].

Підсистема диспетчеризації виконує функцію проміжного буфера між мережею та пулом потоків. Вона приймає сформовані завдання, класифікує їх за типом операції та передає до черги виконання, що відображено у структурній схемі багатопотокової обробки клієнтських запитів (Рис. 2.6) [37].

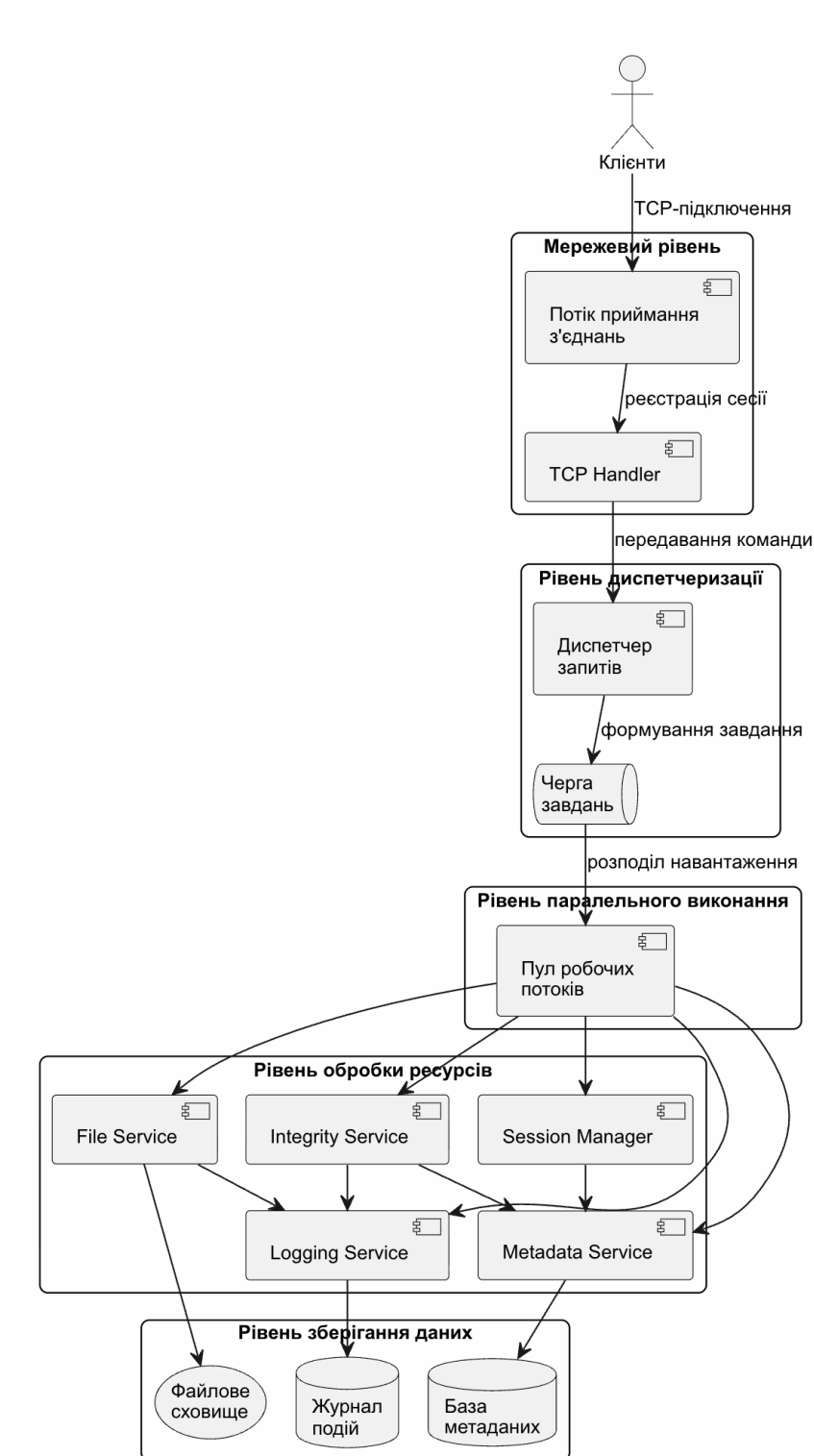


Рис. 2.6 – Структурна схема багатопотокової обробки клієнтських запитів

Проектована модель багатопотокової обробки орієнтована передусім на I/O-залежний характер навантаження. У системі обміну файлами значну частину часу займає не саме обчислення, а очікування мережових пакетів, доступу до диска, запису блоків, читання метаданих і завершення операцій журналювання [32].

Поряд із перевагами багатопотоковість створює ризики, пов'язані з доступом до спільних ресурсів. У проєктованій системі такими ресурсами є файлове сховище, тимчасові файли передачі, записи метаданих, журнали подій і внутрішні об'єкти сесій, тому схема синхронізації подана окремо (Рис. 2.7) [32].

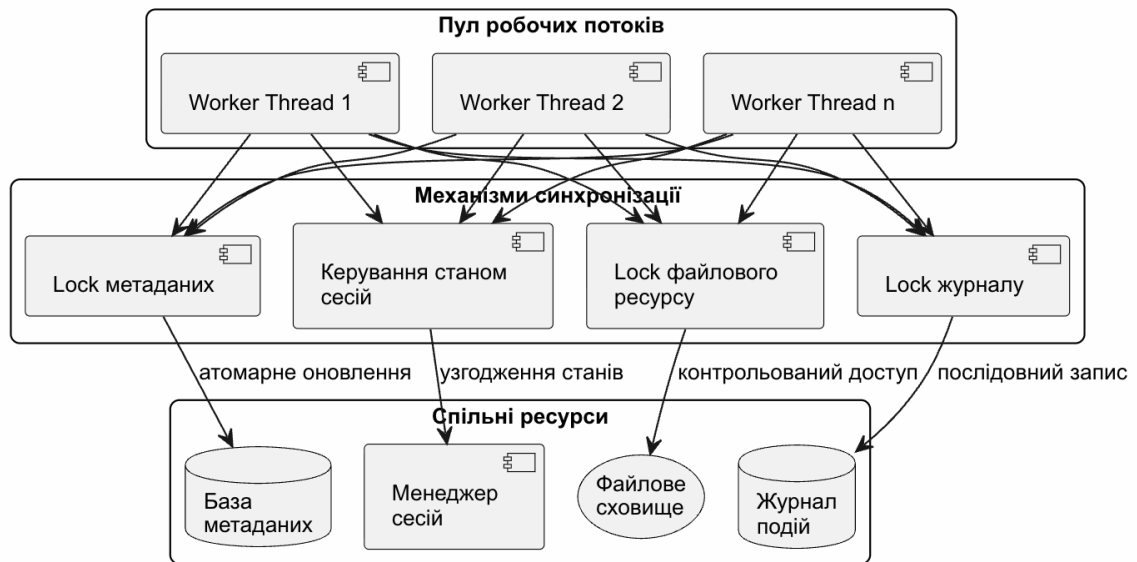


Рис. 2.7 – Схема синхронізації доступу до спільних ресурсів у багатопотоковому сервері

Критичні секції мають бути локалізовані й якомога коротшими. Потік повинен утримувати блокування лише на той час, який потрібний для атомарного виконання конкретної дії: запису блока у файл, оновлення зміщення передачі, зміни статусу сесії, фіксації завершення операції [13].

Суттєве значення для багатопотокового сервера має контроль життєвого циклу сесії. Кожне клієнтське підключення повинно мати власний ідентифікатор, поточний статус, часові мітки початку й завершення, ознаки активності та відомості про незавершені операції. Робочий потік, який обслуговує конкретний запит, звертається до менеджера сесій для читання і зміни відповідних параметрів [23].

Для запобігання перевантаженню сервера механізм обробки повинен підтримувати політики обмеження навантаження. Кількість одночасно активних робітників у пулі не може бути довільною, оскільки надмірна кількість потоків

лише збільшить накладні витрати на перемикання контексту й ускладнить доступ до спільних ресурсів [37].

У механізмі багатопотокової обробки доцільно також передбачити розрізнення службових і файлових операцій за пріоритетністю. Короткі запити, пов'язані з перевіркою доступності сервера, отриманням списку файлів, читанням стану сесії або фіксацією адміністративного [13].

Взаємодія механізму багатопотоковості з журналюванням потребує окремого врахування. Кожен потік генерує технічні та прикладні події, які мають бути послідовно зафіксовані без накладання записів один на один. Якщо журнал не є потокобезпечним, результати одночасного логування стають нерозбірливими, що позбавляє систему діагностичної цінності [32].

Отже, проєктування механізму багатопотокової обробки запитів у системі обміну файлами базується на розділенні приймання з'єднань і виконання прикладних операцій, використанні диспетчеризації завдань, застосуванні пулу робочих потоків, локалізації критичних секцій, синхронізації доступу до спільних ресурсів, підтриманні життєвого циклу сесій і контролі навантаження. Така модель дозволяє забезпечити одночасне обслуговування кількох [37].

2.5. Проєктування структури збереження файлів та журналювання

Проєктування структури збереження файлів та журналювання в клієнт-серверній системі обміну файлами визначається потребою поєднати фізичне розміщення двійкових об'єктів із контрольованим обліком їхнього стану, походження, параметрів передавання та технічних подій, що супроводжують життєвий цикл кожної операції. Для системи, у якій одночасно виконуються завантаження, отримання, [32].

Базовим принципом побудови цієї інфраструктури є розмежування між власне файловим сховищем, у якому розміщується двійковий вміст об'єктів, і

системою метаданих, що описує стан файлів, параметри передачі та службовий контекст (Рис. 2.8). Такий підхід є технічно доцільним з кількох причин [23].

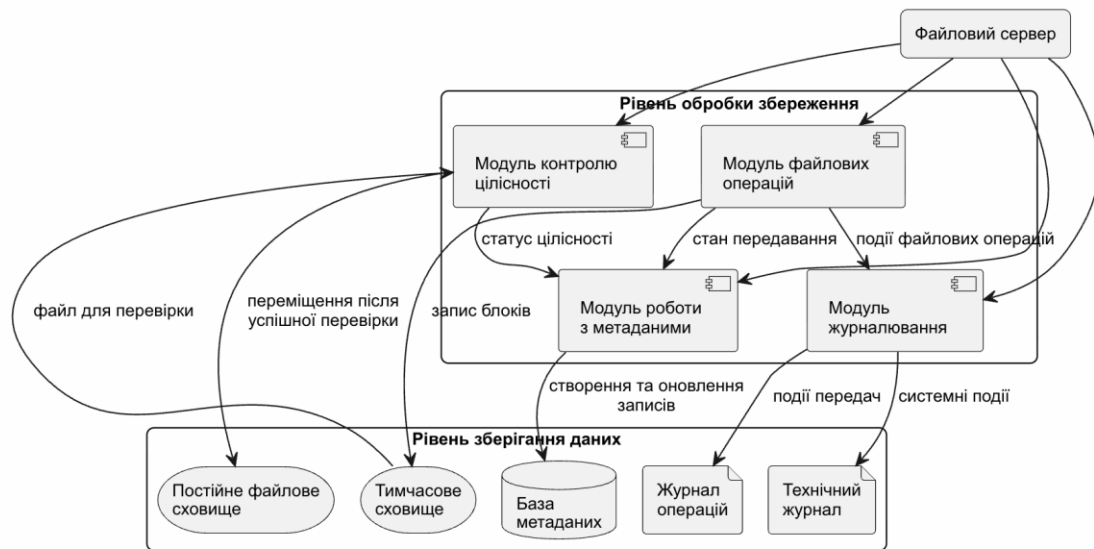


Рис. 2.8 – Структура збереження файлів, метаданих і журналювання в системі обміну файлами

Файлове сховище в проєктованій системі організовується як окрема область постійного збереження даних на стороні файлового сервера. На фізичному рівні воно реалізується через змонтований том Docker, який зберігає вміст незалежно від життєвого циклу контейнера [5].

Особливу роль у структурі збереження відіграють тимчасові файли. Під час передавання великого об'єкта сервер не повинен записувати отримані блоки одразу в остаточний ресурс, оскільки завершення операції може бути перерване через збій мережі, помилку структури пакета, розрив з'єднання або невідповідність контрольної суми [19].

Назви файлів та їхні шляхи збереження не повинні покладатися виключно на початкове ім'я, яке надсилає клієнт. Це пов'язано з ризиком колізій, некоректних символів, спроб перезапису наявних ресурсів або цілеспрямованого використання шкідливих шляхів. Тому у структурі збереження доцільно застосовувати логічні ідентифікатори файлів, які формуються сервером і виступають первинними ключами доступу до ресурсу [13].

Другим ключовим елементом проєктованої структури є база метаданих. Її призначення полягає у фіксації службового опису кожного файлу, стану передачі, параметрів сесії та результатів контрольних перевірок [35].

Вибір SQLite як інструмента для зберігання метаданих є обґрунтованим для контейнеризованої системи середнього масштабу. Це рішення дозволяє зменшити складність розгортання та зберегти структурованість даних. Водночас така модель вимагає контрольованого доступу в умовах багатопотокової роботи [35].

Журналювання в системі виконує іншу функцію, ніж метадані, хоча обидва механізми працюють із службовою інформацією. Метадані відображають поточний або підсумковий стан об'єкта, тоді як журнал фіксує послідовність подій, які призвели до цього стану [32].

Технічне журналювання повинно бути потокобезпечним і підтримувати структурований запис повідомлень. У записі мають фіксуватися дата й час, тип події, рівень критичності, ідентифікатор сесії або передачі, назва компонента, який сформував повідомлення, і текст пояснення [32].

Прикладний журнал операцій може бути реалізований або як окремий структурований лог, або як журналоподібна таблиця у сховищі метаданих. У цій частині доцільно фіксувати атрибути, що прямо стосуються користувацьких або адміністративних дій: створення передачі, отримання підтвердження приймання блока, завершення перевірки цілісності, зміна статусу [32].

Структура збереження й журналювання повинна підтримувати повний життєвий цикл файлу. Цей цикл починається зі створення запису про передачу, після чого сервер створює тимчасовий об'єкт у файловій системі й починає приймання блоків. Паралельно метадані фіксують стан активної передачі, а журнал подій відображає всі критичні етапи процесу. Після завершення передачі модуль контролю цілісності перевіряє контрольну суму [32].

Окремого врахування потребують механізми очищення й обслуговування сховища. Тимчасові файли не можуть залишатися в системі необмежено довго, оскільки це спричинить накопичення службового сміття й неактуальних записів [13].

Таблиця 2.2 – Основні атрибути метаданих файлів і передач

Атрибут	Зміст атрибута	Тип / формат	Призначення в системі
file_id	Унікальний ідентифікатор файла	UUID / str	Однозначна ідентифікація файла в межах системи
file_name	Початкова назва файла, отримана від клієнта	str	Відображення файла в клієнтському й адміністративному інтерфейсі
file_path	Фактичний шлях збереження файла на сервері	str	Локалізація файла у файловому сховищі
temp_path	Шлях до тимчасового файла під час незавершеної передачі	str	Підтримка проміжного стану та відновлення передачі
size	Повний розмір файла в байтах	int / long	Контроль повноти передавання та перевірка відповідності обсягу
checksum	Контрольна сума файла	str	Перевірка цілісності даних після завершення передачі
status	Поточний стан файла або передачі	str	Фіксація стадії життєвого циклу об'єкта (INIT, IN_PROGRESS, COMPLETED, FAILED)
transfer_id	Унікальний ідентифікатор операції передавання	UUID / str	Зв'язок між файлом, сесією та конкретною передачею
session_id	Ідентифікатор клієнтської сесії	UUID / str	Відстеження операцій у межах певного підключення
direction	Напрямок передавання	str	Розрізнення операцій завантаження на сервер і отримання із сервера
offset	Поточне зміщення в передаваному файлі	int	Підтримка блокового передавання та механізму відновлення
chunk_count	Кількість уже оброблених блоків	int	Контроль прогресу передачі
created_at	Дата й час створення запису	datetime	Фіксація початку життєвого циклу файла або передачі
updated_at	Дата й час останнього оновлення запису	datetime	Відстеження останньої зміни стану
completed_at	Дата й час завершення передачі	datetime	Фіксація моменту успішного або аварійного завершення
client_host	Мережева адреса клієнта	str	Ідентифікація джерела запиту
owner	Ідентифікатор користувача або адміністратора, який ініціював дію	str	Прив'язка операції до суб'єкта взаємодії
error_code	Код помилки, якщо операція завершилась невдало	str / int	Формалізоване описання причини збою
error_message	Текстове пояснення помилки	str	Діагностика й журналювання проблемної ситуації
log_ref	Посилання на відповідний запис у журналі подій	str	Узгодження метаданих із технічним або прикладним журналом

Наведений набір атрибутів забезпечує повноцінний опис як фізичних характеристик файла, так і службового стану його передавання. Поєднання ідентифікаторів файла, передачі та сесії створює основу для відстеження повного життєвого циклу операції, тоді як атрибути offset, [13].

Отже, проектування структури збереження файлів та журналювання базується на розмежуванні двійкового вмісту, метаданих і хронології подій, використанні постійного файлового сховища з ізольованою тимчасовою областю, формалізованій моделі метаданих, потокобезпечному технічному журналюванні та прикладному обліку операцій передачі. Така побудова забезпечує контрольований життєвий цикл файлів, підтримує відновлення перерваних передач, підвищує прозорість стану системи для адміністративного вебсервісу й створює придатну для реалізації основу, у якій фізичне збереження [32].

2.6. Моделювання системи

Моделювання клієнт-серверної системи обміну файлами виконує функцію формалізації проєктних рішень, завдяки якій логічна структура, поведінка компонентів, послідовність передавання даних, правила конкурентної обробки та організація стану системи подаються у впорядкованому, узгодженому та придатному до реалізації вигляді. У межах проєктованої системи моделювання не зводиться лише до графічного відображення окремих елементів [16].

На статичному рівні система подається як сукупність взаємопов'язаних компонентів, кожен із яких виконує окрему функціональну роль у межах загального циклу передавання файла. Компонентна модель фіксує існування клієнтського застосунку, файлового сервера, адміністративного вебсервісу, модуля диспетчеризації команд, менеджера сесій, сервісу файлових операцій, механізму контролю цілісності, шару метаданих і підсистеми журналювання [32].

Класова модель деталізує цю структуру вже не на рівні великих функціональних блоків, а на рівні основних сутностей, які формують внутрішній

опис системи. У такому поданні виділяються клієнт, сесія, команда, передавання файла, файловий об'єкт, перевірка цілісності, метадані, файловий сервер, адміністративний сервіс і журнал подій. Діаграма класів (Рис. 2.3) фіксує зв'язки між цими сутностями та їхню участь у файловому обміні [32].

Поведінковий аспект системи відображається через моделювання сценаріїв взаємодії під час завантаження файла на сервер і його отримання із серверного сховища. Ці сценарії є центральними для функціонування всієї системи, оскільки саме в них поєднуються протокольна логіка, сесійне керування, файлові операції, оновлення метаданих і журналювання подій. UML-діаграми послідовності передавання файла на сервер і отримання файла із сервера наведені відповідно на рис. 2.4 і рис. 2.5 [32].

Окрему роль у системному моделюванні відіграє протокольна модель. Її функція полягає в тому, щоб перевести прикладну взаємодію в набір уніфікованих команд і відповідей, придатних до машинної інтерпретації. Командний склад прикладного протоколу, зафіксований у табличній формі (Табл. 2.1), уточнює службові дії клієнта й відповіді сервера [23].

Процесний вимір моделі пов'язаний із багатопотоковою обробкою запитів. Для клієнт-серверної системи обміну файлами цього недостатньо описати на рівні загальних міркувань про паралельність, оскільки одночасна робота кількох клієнтів породжує реальні ризики конфлікту доступу до сховища, журналів, метаданих і стану сесій. Структурна схема багатопотокової обробки клієнтських запитів (Рис. 2.6) показує розподіл приймання з'єднань, диспетчеризації та виконання операцій у пулі потоків [32].

Поглиблення цієї моделі здійснюється через формалізацію доступу до спільних ресурсів. Схема синхронізації доступу робочих потоків до файлового сховища, метаданих, журналів і менеджера сесій (Рис. 2.7) показує, що конкурентна модель функціонує лише за наявності контрольованих точок синхронізації [32].

Інформаційний вимір моделі репрезентується через структуру збереження файлів, метаданих і журналювання (Рис. 2.8), а також через опис атрибутів метаданих у табличній формі (Табл. 2.2) [32].

На рівні розгортання система моделюється як контейнеризована інфраструктура з двома основними сервісами: файловим сервером і адміністративним вебсервісом. Загальна архітектурна схема (Рис. 2.1) відображає розміщення цих сервісів у Docker Compose-середовищі та їхній зв'язок із файловим сховищем, базою метаданих і журналами подій [5].

Таким чином, модель клієнт-серверної системи обміну файлами набуває завершеного вигляду лише за умови узгодження всіх її рівнів. Компонентне моделювання визначає склад функціональних частин, класова модель фіксує основні сутності й зв'язки між ними, діаграми послідовності формалізують динаміку виконання ключових [16].

Висновки до розділу 2

Проектування клієнт-серверної системи обміну файлами дозволило сформуванню архітектурну модель програмного рішення, у якій виділено клієнтський застосунок, файловий сервер, адміністративний сервіс, файлове сховище, базу метаданих і підсистему журналювання. Такий поділ компонентів забезпечує логічну ізоляцію функцій: клієнт відповідає за ініціювання операцій і локальну взаємодію з користувачем, сервер — за приймання команд, багатопоточну обробку, перевірку прав і файлові [32].

Структура взаємодії компонентів побудована навколо власного прикладного протоколу, що використовує службові JSON-повідомлення та бінарні блоки даних. Проектні рішення передбачають фреймінг повідомлень, поділ передавання на етапи ініціалізації, приймання блоків, підтвердження, завершення та фінальної перевірки [31].

Механізм багатопотокової обробки запитів спроектовано з урахуванням потреби одночасного обслуговування кількох клієнтів. Серверна частина має приймати підключення в головному циклі, а обробку клієнтських сесій передавати робочим потокам, що дає змогу не блокувати систему під час тривалих операцій передавання [13].

РОЗДІЛ 3. ПРОГРАМНА РЕАЛІЗАЦІЯ

3.1. Обґрунтування вибору мови Python та використаних бібліотек

Програмна реалізація клієнт-серверної системи обміну файлами потребує такого технологічного стеку, який забезпечує стабільну роботу з мережевими з'єднаннями, підтримку конкурентної обробки клієнтських запитів, коректне виконання файлових операцій, фіксацію службових подій, контроль цілісності даних і можливість тестування окремих модулів. Для реалізації системи обрано мову Python, оскільки вона поєднує достатню виразність синтаксису, розвинену стандартну [1].

Вибір Python обґрунтовується характером задачі. Сервер обміну файлами переважно виконує операції введення-виведення: приймає мережеві пакети, читає й записує файлові блоки, оновлює записи в базі метаданих, формує відповіді клієнту [35].

Мережевий рівень системи реалізовано на основі стандартного модуля `socket`. Цей модуль забезпечує доступ до TCP-сокетів і дозволяє безпосередньо керувати встановленням з'єднання, прийманням клієнтів, передаванням байтових послідовностей і завершенням сеансів. Використання `socket` відповідає архітектурній логіці розробленої системи, оскільки прикладний протокол побудовано не поверх готового HTTP API, а безпосередньо над TCP [34].

Для захисту мережевого каналу використано модуль `ssl`, який входить до стандартної бібліотеки Python і надає засоби створення TLS-контексту. Його застосування дозволяє обгорнути звичайне TCP-з'єднання в захищений канал і зменшити ризики передавання службових повідомлень або файлових даних у відкритому вигляді [36].

Багатопоточна обробка клієнтських підключень реалізована засобами модуля `concurrent.futures`, зокрема через клас `ThreadPoolExecutor`. Цей механізм обрано як керовану альтернативу створенню необмеженої кількості потоків для кожного нового клієнта [29].

Для синхронізації доступу до спільних структур застосовується модуль `threading`, зокрема механізми блокування. Вони потрібні через те, що кілька клієнтських потоків можуть одночасно звертатися до файлового сховища, бази метаданих, журналу подій або службових структур активних сесій [32].

Для збереження службової інформації обрано `SQLite`, доступ до якого здійснюється через стандартний модуль `sqlite3`. Це рішення відповідає масштабу прототипу, оскільки система не потребує окремого серверного компонента бази даних, але має зберігати структуровані записи про користувачів, файли та сесії передавання [35].

Контроль цілісності даних реалізовано за допомогою модуля `hashlib`. Для обчислення контрольних значень використовується алгоритм `SHA-256`, який дозволяє порівняти початковий файл із файлом, отриманим сервером або клієнтом після завершення передавання [30].

Прикладний протокол системи реалізовано з використанням модулів `json` і `struct`. Модуль `json` застосовується для формування службових повідомлень, що містять команди, параметри операцій, ідентифікатори файлів, розміри блоків, хеші, токени сесій і статуси відповідей [31].

Робота з файловою системою реалізована через модулі `pathlib`, `os`, `shutil` і `uuid`. `pathlib` забезпечує зручне й кросплатформене формування шляхів до директорій `storage/tmp`, `storage/files`, `downloads` і `logs`. Модуль `uuid` використовується для створення унікальних службових імен файлів, що запобігає конфліктам між файлами з однаковими початковими назвами [33].

Журналювання подій реалізовано через стандартний модуль `logging`. Він забезпечує фіксацію запуску сервера, підключення клієнтів, виконання команд, результатів автентифікації, початку й завершення передавання, помилок перевірки блоків, відмов у доступі та завершення сесій [32].

Для адміністративного перегляду стану системи використано `FastAPI`. Цей фреймворк дозволяє створити легкий службовий API для отримання інформації про файли, метадані й журнали без ускладнення основної TCP-логіки. `FastAPI` не

замінює файловий сервер і не бере участі в передаванні файлів через основний протокол, а виконує допоміжну функцію моніторингу [32].

Тестування реалізованих модулів виконується за допомогою `pytest`. Цей інструмент дозволяє перевірити окремі функції протоколу, коректність обчислення SHA-256, автентифікацію, базові команди, передавання файлів, обробку помилкових запитів і поведінку системи під паралельним навантаженням [19].

3.2. Реалізація серверної частини

Серверна частина клієнт-серверної системи обміну файлами реалізована як центральний програмний компонент, що забезпечує приймання мережових підключень, оброблення прикладних команд, керування файловими передаваннями, перевірку цілісності даних, збереження метаданих, автентифікацію користувачів і журналювання службових подій. Архітектура серверного компонента побудована за модульним принципом: основна логіка запуску й обслуговування клієнтів зосереджена у файлі `server/server.py`, прикладний протокол винесено до `server/protocol.py`, робота з файловим сховищем реалізована в [32].

Основою серверної частини є клас `FileExchangeServer`, який ініціалізує мережовий сокет, підготовлює середовище виконання, створює або відкриває базу метаданих, перевіряє наявність директорій для тимчасового та постійного збереження файлів, налаштовує журналювання й запускає цикл приймання клієнтських підключень. Сервер використовує TCP як транспортний протокол, оскільки саме TCP забезпечує впорядковану доставку байтового [32].

Для організації конкурентної обробки застосовано механізм `ThreadPoolExecutor`. Така модель дозволяє обмежити кількість одночасних робочих потоків і запобігти неконтрольованому створенню нових потоків при зростанні кількості клієнтів [29].

Мережевий обмін між клієнтом і сервером виконується через власний прикладний протокол, реалізований у модулі `server/protocol.py`. Його призначення полягає у формалізації структури службових повідомлень і відокремленні JSON-заголовків від бінарних даних файла [31].

Сервер підтримує набір прикладних команд, необхідних для повного циклу роботи з файлами. Команда `LOGIN` використовується для автентифікації користувача та отримання сесійного токена. Команда `PING` перевіряє доступність сервера й коректність базового каналу зв'язку [22].

Автентифікаційний механізм серверної частини побудований на таблиці користувачів у `SQLite`. У базі даних зберігаються ідентифікатор користувача, ім'я облікового запису, хеш пароля, роль і час створення запису [35].

Рольова модель доступу реалізована через розмежування користувачів із ролями `admin` і `user`. Адміністратор має розширені права доступу до файлових записів, тоді як звичайний користувач працює переважно з власними файлами [23].

Файлове сховище серверної частини організоване через відокремлення тимчасових і завершених файлів. Під час початку завантаження сервер створює запис про сесію передавання й формує службове ім'я файла, що не залежить безпосередньо від початкової назви, переданої клієнтом. Це потрібно для запобігання конфліктам імен і зменшення ризиків перезапису вже збережених об'єктів [27].

Передавання файла на сервер організоване за блоковою моделлю. Після команди `UPLOAD_INIT` сервер створює сесію передавання, повертає клієнту ідентифікатор сесії та очікує надходження окремих блоків. Кожен блок супроводжується службовим JSON-повідомленням, у якому вказуються ідентифікатор передавання, номер блока, зміщення, розмір і контрольна сума блока [31].

Механізм `UPLOAD_RESUME` забезпечує продовження незавершеного передавання після розриву з'єднання або припинення клієнтської операції. Для цього сервер зберігає службову інформацію про активну або перервану сесію в

таблиці `transfer_sessions`: ідентифікатор передавання, очікуваний розмір, контрольну [19].

Завантаження файлу із сервера реалізовано через команду `DOWNLOAD`. Після отримання запиту сервер перевіряє сесійний токен, права доступу користувача до відповідного файлу, наявність запису в базі метаданих і фізичну наявність файлу у сховищі [35].

Захист файлових операцій доповнюється очищенням назв файлів і перевіркою шляхів. Сервер не використовує передану клієнтом назву як прямий шлях до збереження файлу, а нормалізує її, вилучає небезпечні фрагменти й формує службове ім'я через `UUID` [27].

Для збереження метаданих використовується `SQLite`-база, яка містить таблиці `users`, `files` і `transfer_sessions`. Таблиця `users` забезпечує автентифікацію та зберігання ролей, таблиця `files` фіксує відомості про завершені файли, а таблиця `transfer_sessions` описує активні або перервані передавання. Операції з базою інкапсульовані в окремому модулі, тому серверна логіка не працює з `SQL`-запитами безпосередньо в кожному обробнику команди [35].

Серверна частина також підтримує `TLS`-режим через модуль `ssl`. При відповідному налаштуванні сервер створює `TLS`-контекст, завантажує сертифікат і ключ, після чого обгортає прийняте клієнтське з'єднання в захищений канал. У тестовій конфігурації використовується самопідписаний сертифікат, достатній для локальної перевірки механізму шифрування [36].

Журналювання виконується у файл `logs/server.log`. Сервер фіксує запуск, підключення клієнтів, виконання команд, результати входу, відмови в доступі, створення сесій передавання, отримання блоків, помилки контрольних сум, завершення передавання й аварійні ситуації [32].

Серверна частина доповнена адміністративним `API` на `FastAPI`, який виконує функцію службового перегляду стану системи. Через цей `API` можна отримувати інформацію про файли, метадані та журнали, не втручаючись у основний `TCP`-протокол передавання [32].

Реалізована серверна частина забезпечує завершений цикл обробки файлів: автентифікацію користувача, приймання команди, перевірку прав доступу, створення або пошук сесії передавання, блоковий запис даних, підтвердження коректних фрагментів, фінальну перевірку цілісності, перенесення файла до постійного сховища, збереження метаданих і фіксацію події в журналі. Така структура дозволяє розглядати сервер як функціональне ядро системи, яке поєднує мережеву взаємодію, конкурентну [32].

3.3. Реалізація клієнтської частини

Клієнтська частина системи реалізована як консольний застосунок, призначений для взаємодії користувача із файловим сервером через прикладний протокол поверх TCP-з'єднання. Її функціональне призначення полягає у формуванні запитів до сервера, передаванні службових параметрів, надсиланні й отриманні файлових блоків, перевірці контрольних сум, відображенні результатів операцій і коректному завершенні клієнтської сесії [19].

Основний модуль клієнта розміщено у файлі `client/client.py`. У ньому реалізовано логіку запуску програми з командного рядка, підготовку параметрів підключення, встановлення з'єднання із сервером, виконання автентифікації, формування службових JSON-повідомлень і оброблення відповідей [31].

Клієнтський застосунок працює в режимі командного інтерфейсу. Користувач запускає окрему команду, після чого програма встановлює з'єднання із сервером, за потреби виконує вхід до системи, надсилає запит і повертає результат у текстовому вигляді [27].

Клієнт підтримує команди `login`, `ping`, `list`, `upload`, `resume`, `download`, `info`, `delete`, `raw` і завершення роботи. Команда `login` використовується для автентифікації користувача та отримання сесійного токена. Команда `ping` перевіряє доступність сервера й коректність мережевого з'єднання. Команда `list` повертає перелік файлів, доступних поточному користувачу відповідно до його ролі [22].

Перед виконанням команд, які потребують доступу до серверних ресурсів, клієнт проходить автентифікацію. Під час входу формується JSON-повідомлення з командою LOGIN, іменем користувача та паролем. Сервер перевіряє облікові дані й у разі успішної автентифікації повертає сесійний токен [31].

Встановлення мережевого з'єднання виконується через TCP-сокет. Клієнт зчитує адресу сервера, порт і режим використання TLS із конфігураційних параметрів або аргументів запуску. Якщо активовано захищений режим, звичайне TCP-з'єднання обгортається за допомогою TLS-контексту [34].

Обмін службовими повідомленнями побудовано на тому самому механізмі фреймінгу, що й у серверній частині. Клієнт передає JSON-повідомлення не як довільний текстовий рядок, а як структурований блок: спочатку надсилається 4-байтове поле довжини JSON-заголовка, потім сам JSON-заголовок, після чого, за потреби, передаються бінарні дані файла [31].

Передавання файла на сервер починається з локальної перевірки шляху. Клієнт визначає, чи існує файл, чи є він звичайним файловим об'єктом, чи доступний він для читання. Після цього обчислюється розмір файла та SHA-256 усього вмісту. Ці параметри потрібні серверу для створення сесії передавання й подальшої фінальної перевірки цілісності [19].

Після ініціалізації клієнт відкриває файл у бінарному режимі й починає послідовно читати його блоками. Для кожного блока формується службове повідомлення з командою CHUNK, ідентифікатором сесії передавання, номером або зміщенням блока, розміром фрагмента та SHA-256 конкретного блока. Потім клієнт надсилає JSON-заголовок і відповідну бінарну частину [31].

Після передавання всіх блоків клієнт надсилає команду UPLOAD_FINISH. Ця команда повідомляє серверу, що передавання завершено, і ініціює фінальну перевірку файла на серверному боці [27].

Команда resume реалізує продовження незавершеного передавання. Її використання передбачає, що сервер уже має запис про сесію завантаження й тимчасовий файл із частково прийнятими даними. Клієнт надсилає команду

UPLOAD_RESUME, після чого сервер повертає поточне зміщення, з якого необхідно продовжити надсилання [27].

Отримання файла із сервера виконується командою download. Клієнт передає серверу ідентифікатор або службове позначення файла разом із сесійним токеном. Сервер перевіряє права доступу, знаходить запис у базі метаданих і повертає службову інформацію про файл: початкову назву, розмір і SHA-256 [35].

Команди list, info і delete не передбачають передавання великих бінарних даних, тому їх реалізація зосереджена на формуванні службових JSON-запитів та інтерпретації відповідей. Для list клієнт отримує структурований перелік доступних файлів і виводить його в зручному текстовому вигляді. Для info відображаються основні метадані вибраного файла [31].

Клієнтська частина містить засоби локальної обробки помилок. Вони охоплюють відсутність файла для завантаження, неможливість підключення до сервера, невдалу автентифікацію, відмову в доступі, втрату з'єднання, некоректну відповідь сервера, помилку контрольної суми та відсутність запитуваного файла [19].

Локальна структура клієнта передбачає окрему директорію downloads для файлів, отриманих із сервера. Це дозволяє не змішувати вхідні файли користувача, які передаються на сервер, із файлами, завантаженими назад із віддаленого сховища. Під час збереження отриманого файла клієнт використовує безпечне ім'я й контролює, щоб файл записувався саме в межах визначеної директорії [27].

Реалізація клієнтської частини забезпечує повний цикл взаємодії з сервером: встановлення з'єднання, автентифікацію, виконання службових команд, передавання файла блоками, отримання підтверджень, продовження незавершеної передачі, завантаження файла із серверного сховища, локальну перевірку SHA-256 і виведення результатів користувачу. Консольна форма клієнта зберігає простоту керування та водночас надає весь набір [19].

3.4. Реалізація механізму багатопоточності

Механізм багатопоточності в клієнт-серверній системі обміну файлами реалізовано як серверний засіб паралельного обслуговування клієнтських підключень, який дає змогу одночасно виконувати кілька незалежних операцій передавання, отримання, перегляду та видалення файлів. Для такої системи багатопоточність має прикладне значення, оскільки операції обміну файлами пов'язані з тривалими мережевими й файловими діями: прийманням блоків через TCP-з'єднання, записом тимчасових файлів, перевіркою контрольних сум, оновленням бази метаданих і формуванням відповідей клієнту [37].

У реалізованій системі головний серверний цикл виконує функцію приймання TCP-з'єднань. Після запуску сервер створює сокет, прив'язує його до заданого хоста й порту, переходить у режим прослуховування та очікує на клієнтські підключення. Коли новий клієнт встановлює з'єднання, сервер не починає виконувати всі його команди в головному потоці. Натомість прийняте підключення разом із мережевою адресою клієнта передається до робочого середовища, організованого через `ThreadPoolExecutor` [29].

Використання `ThreadPoolExecutor` обрано як керований спосіб організації багатопоточності. Альтернативна модель, за якої для кожного клієнта створюється новий потік без обмежень, є простішою за структурою, але менш контрольованою щодо використання ресурсів [29].

Змістовно кожен робочий потік обслуговує окрему клієнтську сесію. У межах цієї сесії він приймає службові JSON-команди, перевіряє їхню коректність, визначає тип операції, виконує автентифікацію або перевірку сесійного токена, звертається до файлового сховища, бази метаданих і журналу подій [31].

Багатопоточність у цій системі орієнтована насамперед на задачі введення-виведення. Передавання файла не потребує постійного інтенсивного використання центрального процесора, оскільки більша частина часу витрачається на очікування

даних із мережі або завершення операцій запису на диск. Під час такого очікування інший потік може виконувати команди іншого клієнта [37].

Паралельне виконання клієнтських сесій потребує захисту спільних ресурсів від некоректного одночасного доступу. У системі такими ресурсами є база метаданих SQLite, службова структура активних сесій, тимчасове файлове сховище, директорія завершених файлів і журнал подій [32].

База метаданих використовується для зберігання відомостей про користувачів, файли та сесії передавання. Оскільки різні клієнти можуть одночасно виконувати `UPLOAD_INIT`, `UPLOAD_FINISH`, `LIST`, `INFO`, `DELETE` або `UPLOAD_RESUME`, операції з базою повинні бути узгодженими [35].

Файлове сховище також потребує контрольованого доступу. Під час завантаження файлів сервер працює з директорією `storage/tmp`, де зберігаються незавершені або частково прийняті файли, і з директорією `storage/files`, куди переміщуються успішно перевірені об'єкти. Потік, який отримує блок файла, записує його у визначене зміщення тимчасового файла. Після завершення передавання той самий або інший обробник може виконувати фінальну перевірку SHA-256 і перенесення файла до постійного сховища [19].

Механізм блокового передавання з ACK і NACK також пов'язаний із багатопоточністю. Кожен клієнтський потік отримує блок, перевіряє його SHA-256, записує дані у тимчасовий файл і повертає підтвердження. Підтвердження ACK означає, що блок прийнято коректно, а NACK повідомляє клієнту про потребу повторного передавання [37].

Відновлення перерваного передавання через `UPLOAD_RESUME` додатково демонструє необхідність узгодження потоків. Коли клієнт звертається із запитом на продовження передачі, сервер перевіряє наявність відповідної сесії, визначає кількість уже отриманих байтів і повертає клієнту зміщення для продовження [37].

Журналювання в багатопоточному середовищі виконує не лише інформаційну, а й діагностичну функцію. Оскільки події від різних клієнтів можуть відбуватися одночасно, записи в `server.log` дозволяють простежити, який клієнт виконував певну команду, коли було створено сесію передавання, який блок було

прийнято, чи виникала помилка контрольної суми, чи було відмовлено в доступі [32].

Реалізація багатопоточності впливає і на обробку помилок. Помилка одного клієнта не повинна зупиняти весь сервер або впливати на інші активні сесії [37].

Паралельне обслуговування також враховує авторизаційні перевірки. Кожен робочий потік отримує команду від клієнта разом із сесійним токеном і перед виконанням захищеної операції звертається до механізму перевірки користувача. Це означає, що навіть при одночасній роботі багатьох клієнтів права доступу застосовуються окремо до кожної команди. Користувач із роллю user може бачити й завантажувати лише дозволені файли, тоді як адміністратор має ширший доступ. Багатопоточність не скасовує цих правил, а лише забезпечує паралельне виконання команд за умови індивідуальної перевірки кожної сесії. Контроль кількості робочих потоків дозволяє адаптувати сервер до доступних ресурсів [37].

Під час навантажувального тестування багатопоточна модель перевіряється через одночасну роботу кількох клієнтів, які виконують передавання файлів на сервер. Такий сценарій дозволяє оцінити, чи сервер здатний приймати паралельні з'єднання, чи не виникають помилки при одночасному записі у [37].

Реалізований механізм багатопоточності поєднує простоту потокової моделі з контрольованим використанням ресурсів. Головний потік відповідає за приймання підключень, пул робочих потоків — за обробку клієнтських сесій, блокування — за узгодженість спільних ресурсів, а журналювання — за відтворюваність подій [32].

3.5. Забезпечення надійності та безпеки передачі даних

Надійність і безпека передачі даних у клієнт-серверній системі обміну файлами забезпечуються сукупністю програмних механізмів, що охоплюють захищене мережеве з'єднання, автентифікацію користувачів, рольове розмежування доступу, контроль цілісності файлів, підтвердження приймання

блоків, відновлення незавершених передач, безпечну роботу з файловими шляхами, збереження метаданих і журналювання подій. Такий підхід дає змогу розглядати надійність не лише як здатність системи передати файл від клієнта до сервера, а як властивість збереження коректного стану даних за умов паралельної роботи кількох [32].

Передавання файлів у системі виконується поверх TCP-з'єднання, яке саме по собі забезпечує впорядковану доставку байтів і повторне передавання втрачених сегментів на транспортному рівні. Проте TCP не гарантує прикладної цілісності файла як логічного об'єкта, оскільки серверу необхідно [19].

Контроль цілісності реалізовано за допомогою алгоритму SHA-256. Перед початком завантаження клієнт обчислює контрольну суму всього файла та передає її серверу в службовому повідомленні `UPLOAD_INIT`. Під час подальшого передавання кожен блок також супроводжується власним хеш-значенням [19].

Після отримання всіх блоків клієнт надсилає команду `UPLOAD_FINISH`, яка ініціює фінальну перевірку файла на серверному боці. На цьому етапі сервер перевіряє фактичний розмір тимчасового файла, обчислює SHA-256 для повного вмісту й порівнює результат із контрольним значенням, отриманим під час ініціалізації передавання [19].

Надійність передавання підвищується через розмежування тимчасового й постійного збереження. Під час активної передачі дані записуються не відразу до основного сховища, а до директорії `storage/tmp`. Тимчасовий файл використовується як проміжний буфер для накопичення прийнятих блоків. Лише після завершення передавання, перевірки розміру та контрольної суми файл переноситься до `storage/files` [19].

Механізм відновлення незавершеного передавання реалізовано через команду `UPLOAD_RESUME` і таблицю `transfer_sessions`. Під час ініціалізації завантаження сервер створює службову сесію, у якій фіксуються ідентифікатор передачі, очікуваний розмір файла, контрольна сума, шлях до тимчасового файла, кількість прийнятих байтів, власник операції та поточний статус [19].

Безпека передавання службових повідомлень і файлових даних підсилюється використанням TLS-режиму. Сервер може обгорнути TCP-з'єднання в захищений канал за допомогою модуля `ssl`, завантажуючи сертифікат і приватний ключ із відповідної конфігурації [36].

Автентифікація користувачів реалізована через команду `LOGIN`. Клієнт передає ім'я користувача та пароль, після чого сервер перевіряє відповідний запис у таблиці `users`. Паролі не зберігаються у відкритому вигляді: для них формується похідне хеш-значення за допомогою `PBKDF2-SHA256` [19].

Рольове розмежування доступу побудовано на двох базових ролях: `admin` і `user`. Користувач із роллю `user` працює переважно з файлами, власником яких він є, тоді як адміністратор має розширений доступ до файлових записів. Під час виконання команд `LIST`, `INFO`, `DOWNLOAD` і `DELETE` сервер перевіряє не лише факт автентифікації, а й право користувача на конкретну операцію [22].

Захист файлової системи реалізований через очищення імен файлів, генерацію службових назв і перевірку шляхів. Сервер не використовує початкову назву файла як прямий шлях для запису у сховище. Назва нормалізується, небезпечні символи або послідовності відкидаються, а фактичне службове ім'я формується з використанням `UUID` [27].

Окремий рівень надійності забезпечує база метаданих `SQLite`. У таблиці `files` фіксуються відомості про завершені файли: ідентифікатор, початкова назва, службова назва, розмір, `SHA-256`, час завантаження, статус, власник і шлях у сховищі. У таблиці `transfer_sessions` зберігаються параметри активних або перерваних передач [35].

Журналювання подій виконує функцію діагностики, аудиту й відтворення послідовності дій. У файл `logs/server.log` записуються події запуску сервера, підключення клієнтів, спроби входу, результати автентифікації, створення сесій передавання, отримання блоків, відповіді `ACK` і `NACK`, завершення завантаження, помилки контрольних [32].

Стійкість серверної частини до помилкових сценаріїв забезпечується через контроль вхідних команд і обробку виняткових ситуацій. Якщо клієнт надсилає

невідому команду, некоректний JSON-заголовок, неправильний токен, недійсний ідентифікатор файла або запит на відсутній об'єкт, сервер не завершує роботу аварійно [31].

Надійність клієнтської частини також підтримується локальними перевірками. Перед завантаженням файла клієнт перевіряє його наявність, доступність для читання, розмір і контрольну суму. Після отримання файла із сервера клієнт обчислює SHA-256 локально збереженого об'єкта та порівнює його з контрольним значенням, переданим сервером [19].

Сукупність реалізованих механізмів формує багаторівневу модель надійності та безпеки. TLS зменшує ризики перехоплення трафіку, автентифікація обмежує доступ до функцій системи, рольова модель контролює права користувачів, SHA-256 підтверджує цілісність файлів і блоків, ACK/NACK забезпечує контроль проміжних фрагментів, UPLOAD_RESUME підвищує стійкість до розривів, тимчасове сховище захищає основну директорію від неповних [38].

3.6. Тестування та аналіз продуктивності системи

Тестування клієнт-серверної системи обміну файлами виконувалося з метою перевірки коректності реалізованих функцій, стійкості серверної частини до помилкових запитів, працездатності механізму багатопоточності, правильності автентифікації та рольового [37].

Початковий етап перевірки полягав у контролі синтаксичної коректності програмного коду. Для цього застосовано команду компіляційної перевірки Python-модулів, яка дозволяє виявити синтаксичні помилки, некоректні імпорти на рівні структури файлів і проблеми, що перешкоджають запуску програми [1].

Основний набір автоматизованих тестів реалізовано з використанням pytest. Цей інструмент дозволив перевірити програмну логіку не лише через ручний запуск клієнтських команд, а й через повторювані тестові сценарії [28].

Контрольний приклад функціонального тестування передбачав послідовне виконання типового циклу роботи користувача. Спочатку клієнт встановлював

з'єднання із сервером і виконував автентифікацію через команду LOGIN. Після успішного отримання сесійного токена перевірялася доступність сервера командою PING. Потім клієнт ініціював передавання файла через UPLOAD_INIT, надсилав файл блоками з використанням команди CHUNK, отримував підтвердження ACK для коректно прийнятих блоків і завершував операцію командою UPLOAD_FINISH [19].

Узагальнені результати функціонального, інтеграційного, навантажувального та безпекового тестування винесено в додаток Б (Табл. Б.1). Наведені сценарії охоплюють не лише позитивні операції, а й ситуації, у яких система має відхиляти некоректні запити або обмежувати доступ користувача до ресурсів [15].

Перевірка прикладного протоколу підтвердила коректність обраної моделі фреймінгу. Передавання JSON-повідомлення з попереднім зазначенням довжини дозволяє серверу й клієнту точно визначати межі службового заголовка в TCP-поточі. Це усуває ризик змішування службових параметрів із бінарними даними файла [31].

Перевірка цілісності файлів виконувалася на двох рівнях. На рівні окремого блока тестувалася правильність формування та перевірки SHA-256 для фрагмента, який передається командою CHUNK. Сервер приймав блок, обчислював його хеш і порівнював отримане значення з тим, яке надіслав клієнт [19].

Автентифікаційні тести підтвердили працездатність команди LOGIN і механізму сесійних токенів. При введенні коректних облікових даних клієнт отримував токен, який надалі використовувався для виконання захищених команд. При використанні неправильного пароля або неіснуючого користувача сервер повертав помилку автентифікації [22].

Інтеграційне тестування завантаження файла дозволило перевірити взаємодію одразу кількох модулів: клієнтського застосунку, прикладного протоколу, серверного обробника команд, файлового сховища, SQLite-бази метаданих і

журналювання. Під час успішного сценарію сервер створював сесію передавання, приймав блоки, підтверджував їх, завершував [32].

Окремо перевірялося відновлення незавершеного передавання. Тестовий сценарій передбачав створення сесії завантаження, часткове передавання файлу, фіксацію поточного зміщення та подальше продовження операції через `UPLOAD_RESUME`. Сервер визначав кількість уже прийнятих байтів і повертав клієнту позицію, з якої потрібно продовжити надсилання [27].

Навантажувальне тестування виконувалося для перевірки багатопоточності та здатності сервера працювати з кількома клієнтами одночасно. У контрольному сценарії три клієнти паралельно передавали файли розміром 1 МБ кожен. За результатами тесту отримано 3 успішні операції та 0 помилок [37].

Аналіз продуктивності системи свідчить, що обрана архітектура є придатною для локального й навчального сценарію використання. Передавання файлів виконується блоками, тому сервер не завантажує весь файл в оперативну пам'ять. Це зменшує ризик перевитрати ресурсів під час роботи з великими файлами. Використання `ThreadPoolExecutor` дозволяє паралельно обслуговувати кількох клієнтів, але водночас обмежує кількість робочих потоків, що запобігає неконтрольованому зростанню навантаження [29].

Під час тестування не виявлено помилок, які б призводили до аварійного завершення сервера при стандартних сценаріях роботи. Некоректні команди, запити до відсутніх файлів, помилки автентифікації та відмови в доступі обробляються через службові відповіді, а не через завершення процесу [1].

Результати тестування підтверджують, що реалізована система виконує заявлені функції: забезпечує клієнт-серверний обмін файлами, підтримує багатопоточне обслуговування клієнтів, використовує автентифікацію і рольове розмежування доступу, передає файли блоками з підтвердженнями, перевіряє SHA-256, зберігає метадані в [37].

Висновки до розділу 3

Програмна реалізація клієнт-серверної системи обміну файлами виконана засобами Python із використанням стандартних бібліотек для мережевої взаємодії, багатопоточності, хешування, роботи з файловою системою, SQLite-базою даних, TLS-з'єднаннями та журналюванням. Серверна частина реалізує приймання TCP-підключень, оброблення клієнтських команд, автентифікацію користувачів, перевірку ролей, збереження файлів, оновлення метаданих і ведення журналу подій [32].

Механізм багатопоточності реалізовано через ThreadPoolExecutor, що дозволяє серверу одночасно обслуговувати кілька клієнтських сесій. Основний серверний потік приймає нові підключення, а робочі потоки виконують обробку команд, приймання блоків, перевірку SHA-256, запис у тимчасове сховище, перенесення завершених файлів і оновлення бази метаданих. Така організація забезпечує стабільне виконання паралельних операцій і запобігає некоректному змішуванню даних різних клієнтів [29].

Надійність і безпека передавання забезпечені через TLS-режим, автентифікацію користувачів, хешування паролів, сесійні токени, рольове розмежування доступу, очищення імен файлів, захист від path traversal, тимчасове збереження незавершених передач, підтвердження блоків через ACK/NACK, відновлення через UPLOAD_RESUME і перевірку SHA-256 для блоків та повного файла. Тестування підтвердило працездатність основних функцій системи: синтаксична перевірка завершилася результатом compileall OK, автоматизований набір тестів показав 13 passed in 6.83s, а навантажувальний сценарій із трьома паралельними клієнтами завершився трьома успішними операціями без помилок [38].

ВИСНОВКИ

Розроблення клієнт-серверної системи обміну файлами з підтримкою багатопоточності засобами Python підтвердило практичну придатність обраного архітектурного підходу для організації керованого, контрольованого й відтворюваного передавання даних між клієнтськими вузлами та сервером. Поставлена мета досягнута через поєднання теоретичного аналізу предметної області, проєктування структури системи, формалізації прикладного протоколу, реалізації серверної та клієнтської частин, упровадження механізмів багатопоточності, контролю цілісності, автентифікації, рольового доступу й тестової перевірки працездатності програмного прототипу [37].

Аналіз предметної області засвідчив, що файловий обмін у мережевому середовищі потребує не лише передавання байтів між двома вузлами, а й упорядкованого керування станами операцій, контролю доступу, перевірки цілісності та коректної реакції на помилки. Клієнт-серверна архітектура виявилася придатною для такої задачі, оскільки дозволяє централізувати збереження файлів, метаданих і журналів, а також відокремити клієнтську логіку від серверної обробки. Для розробленого прототипу раціональним рішенням стало створення власного прикладного протоколу поверх TCP, оскільки такий підхід забезпечив точний контроль над структурою службових повідомлень, передаванням блоків, підтвердженнями, метаданими та станами файлових операцій [32].

Проєктування системи дало змогу сформувати цілісну архітектуру, у якій клієнтський застосунок, файловий сервер, адміністративний сервіс, файлова система, база метаданих і підсистема журналювання мають чітко визначені функції. Серверна частина спроєктована як центральний вузол, що приймає підключення, перевіряє користувачів, обробляє команди, керує передаванням і збереженням файлів. Клієнтська частина виконує роль інтерфейсу користувача для ініціювання операцій і локальної перевірки результатів. База метаданих забезпечує облік користувачів, завершених файлів і сесій передавання, а журнал подій дозволяє відтворити послідовність дій під час тестування й діагностики [32].

Програмна реалізація виконана засобами Python, що дозволило використати стандартні бібліотеки для мережевої взаємодії, багатопоточності, хешування, роботи з SQLite, файловою системою, TLS-з'єднаннями та журналюванням. Серверна частина реалізує TCP-сокет, приймання клієнтських підключень, обробку команд LOGIN, PING, LIST, INFO, DELETE, UPLOAD_INIT, UPLOAD_RESUME, CHUNK, UPLOAD_FINISH, DOWNLOAD і QUIT [32].

Механізм багатопоточності реалізовано за допомогою ThreadPoolExecutor, що дало змогу відокремити головний серверний цикл приймання підключень від обробки клієнтських сесій. Така організація дозволяє одночасно обслуговувати кількох клієнтів, не блокуючи сервер під час тривалого передавання файла або очікування мережових даних. Синхронізація доступу до спільних ресурсів забезпечує узгодженість операцій із базою метаданих, тимчасовими файлами, сесіями передавання та журналом подій [29].

Надійність передавання забезпечена блоковою структурою обміну, підтвердженнями ACK/NACK, перевіркою SHA-256 для окремих фрагментів і фінальною перевіркою повного файла. Система не переносить файл до основного сховища, доки не буде підтверджено відповідність фактичного розміру й контрольної суми очікуваним значенням [19].

Безпека системи реалізована на кількох рівнях. TLS-режим захищає транспортний канал у тестовому середовищі, автентифікація через LOGIN обмежує доступ до серверних операцій, PBKDF2-SHA256 унеможливорює збереження паролів у відкритому вигляді, а сесійні токени пов'язують подальші запити з конкретним користувачем. Рольова модель admin/user дозволяє розмежувати доступ до файлів відповідно до прав користувача [38].

Тестування підтвердило працездатність основного функціонального ядра. Синтаксична перевірка завершилася результатом compileall OK, що засвідчило коректність структури програмних модулів. Автоматизований набір тестів показав результат 13 passed in 6.83s, охопивши прикладний протокол, хешування, автентифікацію, рольове розмежування, завантаження та отримання файлів,

підтвердження блоків, відновлення передавання, обробку помилкових команд і запити до відсутніх файлів [19].

Розроблений прототип має завершену функціональну структуру й відповідає поставленим завданням. Він демонструє практичну реалізацію клієнт-серверного обміну файлами з підтримкою багатопоточності, контролем цілісності, базовими механізмами безпеки, метаданими, журналюванням і тестовою перевіркою продуктивності [32].

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Beazley D., Jones B. K. Python Cookbook: Recipes for Mastering Python 3. 3rd ed. Sebastopol : O'Reilly Media, 2013. 706 p. URL: <https://www.oreilly.com/library/view/python-cookbook-3rd/9781449357337/> (Дата звернення 03.12.2025)
2. Bosch J. Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach. Boston : Addison-Wesley, 2000. 354 p. URL: <https://dl.acm.org/doi/abs/10.5555/339362> (Дата звернення 03.12.2025)
3. Burns B. Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services. Sebastopol : O'Reilly Media, 2018. 166 p. URL: <https://www.oreilly.com/library/view/designing-distributed-systems/9781491983638/> (Дата звернення 09.12.2025)
4. Docker Docs. Compose file reference. URL: <https://docs.docker.com/reference/compose-file/> (Дата звернення 17.12.2025)
5. Docker Docs. Docker Compose overview. URL: <https://docs.docker.com/compose/> (Дата звернення 24.12.2025)
6. Docker Docs. Dockerfile reference. URL: <https://docs.docker.com/reference/dockerfile/> (Дата звернення 13.01.2026)
7. FastAPI Documentation. Testing. URL: <https://fastapi.tiangolo.com/tutorial/testing/> (Дата звернення 15.01.2026)
8. Fielding R. T. Architectural Styles and the Design of Network-based Software Architectures : doctoral dissertation. Irvine : University of California, 2000. 180 p. URL: <https://ics.uci.edu/~fielding/pubs/dissertation/top.htm> (Дата звернення 21.01.2026)
9. Fielding R., Nottingham M., Reschke J. RFC 9110. HTTP Semantics. IETF, 2022. URL: <https://www.rfc-editor.org/rfc/rfc9110> (Дата звернення 21.01.2026)

10. Fowler M. Patterns of Enterprise Application Architecture. Boston : Addison-Wesley, 2002. 560 p. URL: <https://martinfowler.com/books/ea.html> (Дата звернення 27.01.2026)
11. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Boston : Addison-Wesley, 1994. 395 p. URL: <https://www.oreilly.com/library/view/design-patterns-elements/0201633612/> (Дата звернення 03.02.2026)
12. IEEE Computer Society. SWEBOK: Guide to the Software Engineering Body of Knowledge. Version 4.0. IEEE Computer Society, 2024. URL: <https://www.computer.org/education/bodies-of-knowledge/software-engineering> (Дата звернення 05.02.2026)
13. ISO/IEC/IEEE 12207:2017. Systems and software engineering — Software life cycle processes. Geneva : ISO, 2017. 147 p. URL: <https://www.iso.org/standard/63712.html> (Дата звернення 11.02.2026)
14. ISO/IEC 25010:2023. Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model. Geneva : ISO, 2023. URL: <https://www.iso.org/standard/78176.html> (Дата звернення 11.02.2026)
15. ISO/IEC/IEEE 29119-1:2022. Software and systems engineering — Software testing — Part 1: General concepts. Geneva : ISO, 2022. URL: <https://www.iso.org/standard/81291.html> (Дата звернення 11.02.2026)
16. Kleppmann M. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. Sebastopol : O'Reilly Media, 2017. 616 p. URL: <https://www.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/> (Дата звернення 19.02.2026)
17. Kurose J. F., Ross K. W. Computer Networking: A Top-Down Approach. 8th ed. Harlow : Pearson, 2021. 800 p. URL: https://gaia.cs.umass.edu/kurose_ross (Дата звернення 24.02.2026)
18. Newman S. Building Microservices: Designing Fine-Grained Systems. 2nd ed. Sebastopol : O'Reilly Media, 2021. 616 p. URL:

<https://www.oreilly.com/library/view/building-microservices-2nd/9781492034018/> (Дата звернення 24.02.2026)

19. NIST. FIPS PUB 180-4. Secure Hash Standard (SHS). Gaithersburg : National Institute of Standards and Technology, 2015. URL: <https://csrc.nist.gov/publications/detail/fips/180/4/final> (Дата звернення 24.02.2026)
20. NIST. SP 800-63B. Digital Identity Guidelines: Authentication and Lifecycle Management. Gaithersburg : National Institute of Standards and Technology, 2017. URL: <https://pages.nist.gov/800-63-3/sp800-63b.html> (Дата звернення 03.03.2026)
21. Nottingham M. RFC 9111. HTTP Caching. IETF, 2022. URL: <https://www.rfc-editor.org/rfc/rfc9111> (Дата звернення 03.03.2026)
22. OWASP Foundation. Authentication Cheat Sheet. URL: https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html (Дата звернення 04.03.2026)
23. OWASP Foundation. Authorization Cheat Sheet. URL: https://cheatsheetseries.owasp.org/cheatsheets/Authorization_Cheat_Sheet.html (Дата звернення 05.03.2026)
24. OWASP Foundation. File Upload Cheat Sheet. URL: https://cheatsheetseries.owasp.org/cheatsheets/File_Upload_Cheat_Sheet.html (Дата звернення 25.03.2026)
25. Postel J. RFC 793. Transmission Control Protocol. DARPA, 1981. URL: <https://www.rfc-editor.org/rfc/rfc793> (Дата звернення 25.03.2026)
26. Postel J., Reynolds J. RFC 959. File Transfer Protocol (FTP). IETF, 1985. URL: <https://www.rfc-editor.org/rfc/rfc959> (Дата звернення 26.03.2026)
27. Pressman R. S., Maxim B. R. Software Engineering: A Practitioner's Approach. 9th ed. New York : McGraw-Hill Education, 2020. 704 p. URL: <https://www.mheducation.com/highered/product/software-engineering-a-practitioners-approach-pressman.html> (Дата звернення 26.03.2026)

- 28.pytest Documentation. URL: <https://docs.pytest.org/en/stable/> (Дата звернення 27.03.2026)
- 29.Python Software Foundation. concurrent.futures — Launching parallel tasks. Python 3 Documentation. URL: <https://docs.python.org/3/library/concurrent.futures.html> (Дата звернення 14.04.2026)
- 30.Python Software Foundation. hashlib — Secure hashes and message digests. Python 3 Documentation. URL: <https://docs.python.org/3/library/hashlib.html> (Дата звернення 16.04.2026)
- 31.Python Software Foundation. json — JSON encoder and decoder. Python 3 Documentation. URL: <https://docs.python.org/3/library/json.html> (Дата звернення 28.04.2026)
- 32.Python Software Foundation. logging — Logging facility for Python. Python 3 Documentation. URL: <https://docs.python.org/3/library/logging.html> (Дата звернення 05.05.2026)
- 33.Python Software Foundation. pathlib — Object-oriented filesystem paths. Python 3 Documentation. URL: <https://docs.python.org/3/library/pathlib.html> (Дата звернення 05.05.2026)
- 34.Python Software Foundation. socket — Low-level networking interface. Python 3 Documentation. URL: <https://docs.python.org/3/library/socket.html> (Дата звернення 06.05.2026)
- 35.Python Software Foundation. sqlite3 — DB-API 2.0 interface for SQLite databases. Python 3 Documentation. URL: <https://docs.python.org/3/library/sqlite3.html> (Дата звернення 06.05.2026)
- 36.Python Software Foundation. ssl — TLS/SSL wrapper for socket objects. Python 3 Documentation. URL: <https://docs.python.org/3/library/ssl.html> (Дата звернення 06.05.2026)
- 37.Python Software Foundation. threading — Thread-based parallelism. Python 3 Documentation. URL: <https://docs.python.org/3/library/threading.html> (Дата звернення 06.05.2026)

38. Rescorla E. RFC 8446. The Transport Layer Security (TLS) Protocol Version 1.3. IETF, 2018. URL: <https://www.rfc-editor.org/rfc/rfc8446> (Дата звернення 21.05.2026)
39. SQLite Documentation. About SQLite. URL: <https://sqlite.org/about.html> (Дата звернення 21.05.2026)
40. SQLite Documentation. Database File Format. URL: <https://sqlite.org/fileformat.html> (Дата звернення 21.05.2026)

ДОДАТОК А

Порівняльна характеристика підходів до конкурентної обробки в клієнт-серверних системах передавання файлів

Таблиця А.1 – Підходи до конкурентної обробки в клієнт-серверних системах передачі файлів

Підхід конкурентності	Одиниця виконання	Типовий профіль ефективності	Основні переваги	Ключові обмеження	Придатність у сервері обміну файлами
Потік на з'єднання (thread-per-connection)	Потік ОС	I/O-bound: помірна; CPU-bound: залежить від середовища	Простота реалізації; ізоляція логіки на рівні з'єднання; прямолінійне дебагування	Масштабованість обмежена кількістю потоків; накладні витрати на контекст; ризик виснаження ресурсів	Невеликі системи або навчальні прототипи з помірною кількістю клієнтів
Пул потоків + черга завдань	Потік ОС + задача	I/O-bound: висока за керованого паралелізму	Контроль кількості паралельних робіт; стабільне використання ресурсів; зручне квотування	Потрібна коректна декомпозиція задач; синхронізація черг і спільних структур	Базовий варіант для багатоклієнтського сервера, зокрема для обробки команд і записів на диск
Багатопрое-сність (process pool / multi-process)	Процес ОС	CPU-bound: висока; I/O-bound: помірна	Реальний паралелізм на ядрах; ізоляція пам'яті; стійкість до помилок в одному процесі	Дорожчі міжпроцесні комунікації; складніша передача стану; дублювання пам'яті	Доцільна для важких обчислень (шифрування, компресія, хешування великих файлів)
Асинхронний цикл подій (async I/O)	Корутина/обробник	I/O-bound: висока при великій кількості з'єднань	Малі накладні витрати; масштабованість за кількістю підключень; природна	Вимога неблокувальності; складніша логіка помилок; блокувальні операції «зупиняють» цикл	Ефективний мережевий рівень; часто комбінується з пулом потоків/процесів для диска й CPU-етапів

			неблокувальна мережа		
Гібрид: async I/O + пул потоків/процесів	Корутина + потік/процес	I/O-bound: висока; CPU-bound: керована	Поєднання масштабованості мережі та паралелізму для блокувальних /CPU- операцій; гнучкий розподіл навантаження	Підвищена складність архітектури; потрібна чітка межа між асинхронними та блокувальними ділянками	Практично придатний для серверів передачі файлів з контролем цілісності, журналюванням і постобробкою
Акторна модель (message-driven actors)	Актор (ізолюваний стан)	Залежить від реалізації; ефективна для складних протоколів	Ізоляція стану, мінімізація гонок; ясна модель обміну повідомленнями	Потребує відповідної інфраструктури/фреймворку; накладні витрати на маршрутизацію повідомлень	Доцільна при складній логіці сесій, квотах, політиках, коли пріоритетом є коректність станів

ДОДАТОК Б

Результати тестування клієнт-серверної системи обміну файлами

Таблиця Б.1 – Результати тестування клієнт-серверної системи обміну файлами

Напрямок тестування	Мета перевірки	Тестовий сценарій	Отриманий результат	Статус
Синтаксична перевірка коду	Встановити, чи всі модулі проекту коректно інтерпретуються Python	Запуск компіляційної перевірки всіх файлів проекту	Отримано результат compileall OK	Успішно
Перевірка прикладного протоколу	Перевірити коректність JSON-фреймінгу та зчитування повідомлень	Передавання службових JSON-заголовків із 4-байтовим полем довжини	Повідомлення коректно кодуються, передаються та зчитуються	Успішно
Перевірка SHA-256	Підтвердити правильність обчислення контрольних сум	Обчислення хешу для тестового файла й окремих блоків	Значення SHA-256 збігаються з очікуваними	Успішно
Автентифікація	Перевірити команду LOGIN і створення токена	Вхід із коректними та некоректними обліковими даними	Коректний користувач отримує токен, неправильні дані відхиляються	Успішно
Рольове розмежування	Перевірити обмеження доступу до чужих файлів	Користувач із роллю user запитує файл іншого власника	Сервер повертає відмову в доступі	Успішно
Завантаження файла на сервер	Перевірити повний цикл UPLOAD_INIT → CHUNK → UPLOAD_FINISH	Передавання файла блоками з контролем SHA-256	Файл збережено, метадані додано до SQLite	Успішно
Підтвердження блоків	Перевірити реакцію сервера на коректні й некоректні блоки	Надсилання блоків із правильним і помилковим хешем	Для коректного блока повертається ACK, для некоректного — NACK	Успішно
Відновлення передавання	Перевірити команду UPLOAD_RESUME	Продовження частково переданого файла з визначеного зміщення	Передавання відновлюється без повторного надсилання всього файла	Успішно
Завантаження файла із сервера	Перевірити команду DOWNLOAD	Отримання файла з подальшим локальним	Отриманий файл відповідає серверному оригіналу	Успішно

		обчисленням SHA-256		
Обробка помилкових команд	Перевірити стійкість сервера до некоректних запитів	Надсилання невідомої команди через тестовий клієнт	Сервер повертає повідомлення про помилку й продовжує роботу	Успішно
Запит відсутнього файла	Перевірити реакцію на некоректний ідентифікатор	Команда DOWNLOAD або INFO для неіснуючого файла	Сервер повідомляє про відсутність об'єкта	Успішно
Автоматизований набір pytest	Перевірити сукупність реалізованих функцій	Запуск усіх автоматизованих тестів	Отримано 13 passed in 6.83s	Успішно
Навантажувальний тест	Перевірити паралельну роботу кількох клієнтів	3 клієнти одночасно передають файли по 1 МБ	3 успішні операції, 0 помилок	Успішно

ДОДАТОК В

Міністерство освіти і науки України
Державний заклад «Луганський національний університет
імені Тараса Шевченка»
Навчально-науковий інститут математики та інформаційних технологій
Кафедра інформаційних технологій та систем

**Програма та методика тестування
на виконання програмної розробки (ПР):
«Розробка клієнт-серверної системи обміну файлами
з підтримкою багатопоточності засобами Python»**

Полтава – 2026

ЗМІСТ

ТЕСТ-ПЛАН	3
1. Вступ	3
1.1. Мета	3
1.2. Цілі тестування	3
1.3. Об'єкти тестування	4
1.4. Тестові сценарії	5
1.5. Методи тестування	5
1.6. План тестування та час тестування	6
2. Тестувальне середовище	7
3. Тест-кейси	8
4. Виконання тестування	11
4.1. Підготовка тестового середовища	12
4.2. Запуск тестових сценаріїв	12
4.3. Реєстрація результатів	13
4.4. Виявлення помилок	13
5. Виправлення помилок	14
6. Повторне тестування	14
7. Висновки тестування	14

ТЕСТ-ПЛАН

1. Вступ

1.1. Мета

Тест-план визначає порядок, методи й критерії перевірки клієнт-серверної системи обміну файлами з підтримкою багатопоточності засобами Python. Документ призначений для формалізації тестових робіт, перевірки відповідності програмної реалізації заявленим вимогам, фіксації результатів перевірки та підтвердження готовності системи до демонстраційного використання.

Метою тестування є підтвердження коректної роботи серверної та клієнтської частин під час передавання, отримання, відновлення й адміністрування файлів у мережевому середовищі. Перевірка охоплює прикладний протокол поверх TCP, блокове передавання даних, підтвердження приймання фрагментів, контроль цілісності за SHA-256, роботу з метаданими у SQLite, журналювання подій, автентифікацію, рольове розмежування доступу, захищене TLS-з'єднання та паралельне обслуговування кількох клієнтів.

1.2. Цілі тестування

- перевірити запуск серверної частини, клієнтського застосунку та адміністративного вебсервісу в контейнеризованому середовищі;
- підтвердити коректність встановлення TCP-з'єднання, обробки службових команд і завершення клієнтської сесії;
- перевірити автентифікацію користувача, видачу сесійного токена та обмеження доступу відповідно до ролі користувача;
- оцінити правильність завантаження файла на сервер через послідовність команд UPLOAD_INIT, CHUNK і UPLOAD_FINISH;
- перевірити механізми ACK/NACK для підтвердження коректних блоків і відхилення пошкоджених або некоректно пронумерованих фрагментів;
- підтвердити відповідність отриманого файла початковому об'єкту шляхом порівняння розміру та контрольної суми SHA-256;

- перевірити відновлення незавершеної передачі через команду `UPLOAD_RESUME` без повторного надсилання всього файлу;
- оцінити стабільність багатопотокової обробки під час одночасної роботи кількох клієнтів;
- перевірити коректність журналювання подій і збереження метаданих про файли, сесії та результати передач;
- визначити, чи обробляються помилкові команди без аварійного завершення серверного процесу.

1.3. Об'єкти тестування

Об'єктами тестування є програмні компоненти, що беруть участь у повному циклі клієнт-серверного обміну файлами. До них належать файловий сервер, клієнтський застосунок, адміністративний вебсервіс, модуль прикладного протоколу, менеджер сесій, модуль файлових операцій, сервіс контролю цілісності, підсистема метаданих, журнал подій і файлове сховище.

- файловий сервер, що приймає TCP-з'єднання та передає клієнтські запити до пулу робочих потоків;
- клієнтський застосунок, який ініціює команди `LOGIN`, `PING`, `LIST`, `INFO`, `UPLOAD_INIT`, `CHUNK`, `UPLOAD_FINISH`, `UPLOAD_RESUME` і `DOWNLOAD`;
- адміністративний API/UI, призначений для перегляду стану системи, історії передач, журналів подій і метаданих;
- SQLite-база метаданих, у якій фіксуються відомості про файли, сесії, користувачів і результати операцій;
- файлове сховище, що містить завершені файли та тимчасові об'єкти незавершених передач;
- механізм журналювання, який забезпечує відтворюваність подій і діагностику помилок;
- механізми безпеки: автентифікація, рольове розмежування доступу, захищене TLS-з'єднання, перевірка файлових шляхів і контроль цілісності.

1.4. Тестові сценарії

Тестові сценарії охоплюють типові та граничні режими роботи системи. Основний сценарій передбачає запуск серверної частини, встановлення TCP-з'єднання клієнтом, автентифікацію, перевірку доступності сервера, завантаження файла блоками, підтвердження приймання фрагментів, завершення передачі, обчислення контрольної суми та подальше отримання файлу з сервера.

- запуск серверної частини, адміністративного вебсервісу та клієнта в підготовленому середовищі;
- перевірка доступності сервера командою PING після встановлення з'єднання;
- автентифікація користувача з коректними й некоректними обліковими даними;
- отримання списку доступних файлів і метаданих окремого файлу;
- завантаження файлу на сервер із поділом на блоки та перевіркою ACK-відповідей;
- імітація пошкодженого блока для перевірки NACK-відповіді та повторного надсилання;
- відновлення незавершеної передачі після розриву з'єднання;
- завантаження файлу із сервера з локальним обчисленням SHA-256;
- перевірка стійкості до невідомих команд, запитів до відсутніх файлів і некоректних параметрів;
- одночасна робота кількох клієнтів із паралельним передаванням файлів;
- перевірка відображення результатів передач і системних подій у журналі та адміністративному інтерфейсі.

1.5. Методи тестування

Для перевірки програмної розробки застосовується поєднання ручного, автоматизованого, інтеграційного, негативного, регресійного та навантажувального тестування. Така комбінація дозволяє оцінити як окремі модулі, так і поведінку системи в повному циклі мережевої взаємодії.

- ручне функціональне тестування — послідовне виконання користувацьких сценаріїв через клієнтський застосунок і контроль відповідей сервера;

- автоматизоване тестування — запуск набору `pytest` для перевірки програмної логіки, команд протоколу, обробки помилок і контрольних сценаріїв;
- інтеграційне тестування — перевірка взаємодії клієнта, сервера, бази метаданих, файлового сховища, журналу подій і адміністративного сервісу;
- негативне тестування — надсилання некоректних команд, помилкових параметрів, неправильних облікових даних і запитів до відсутніх файлів;
- безпекове тестування — перевірка автентифікації, прав доступу, захищеного каналу, заборони небезпечних файлових шляхів і реакції на неавторизовані дії;
- навантажувальне тестування — імітація одночасної роботи кількох клієнтів із передаванням файлів для перевірки багатопотокової моделі;
- регресійне тестування — повторна перевірка сценаріїв після внесення змін до коду або конфігурації.

1.6. План тестування та час тестування

План тестування передбачає послідовне виконання підготовчих, функціональних, інтеграційних, негативних, навантажувальних і повторних перевірок. Тривалість окремих етапів може уточнюватися залежно від обсягу змін у програмному коді, однак загальна логіка тестування залишається сталою: спочатку перевіряється готовність середовища, далі виконуються базові сценарії, після цього перевіряються помилкові та граничні ситуації, а завершальний етап фіксує результати й підтверджує готовність системи.

Таблиця В.1 – План проведення тестування

Етап	Зміст робіт	Метод	Орієнтовний час	Вихідний результат
Підготовка	Перевірка запуску Docker Compose, файлового сервера, адміністративного сервісу, SQLite та каталогів сховища	Ручний контроль	1 год	Середовище готове до тестування
Базові функції	LOGIN, PING, LIST, INFO, UPLOAD_INIT, CHUNK,	Функціональне тестування	2 год	Підтверджено коректність основних команд

Етап	Зміст робіт	Метод	Орієнтовний час	Вихідний результат
	UPLOAD_FINISH, DOWNLOAD			
Цілісність	Перевірка SHA-256, розміру файла, ACK/NACK і повторного надсилання некоректного блока	Інтеграційне тестування	1 год	Підтверджено контроль цілісності
Відновлення	Імітація розриву з'єднання й продовження передачі через UPLOAD_RESUME	Сценарне тестування	1 год	Передавання відновлюється з потрібної позиції
Безпека	Перевірка автентифікації, ролей, некоректних шляхів і неавторизованих операцій	Негативне тестування	1,5 год	Система відхиляє небезпечні дії
Навантаження	Одночасна робота кількох клієнтів із передаванням тестових файлів	Навантажувальне тестування	1 год	Сервер стабільно обробляє паралельні запити
Регресія	Повторне виконання ключових сценаріїв після виправлень або зміни конфігурації	Регресійне тестування	1 год	Підтверджено відсутність повторних дефектів

2. Тестувальне середовище

Тестування виконується у середовищі, наближеному до умов демонстраційної експлуатації програмного продукту. Основою є локальний комп'ютер розробника або тестувальника, на якому запускаються контейнеризовані компоненти системи: файловий сервер, адміністративний вебсервіс, база метаданих і файлове сховище. Клієнтський застосунок підключається до сервера через TCP-канал і виконує службові команди прикладного протоколу.

- операційна система: Windows, Linux або macOS із підтримкою Docker і Python;

- мова та середовище виконання: Python 3.x, стандартні бібліотеки socket, ssl, hashlib, sqlite3, logging, pathlib, concurrent.futures;
- середовище розгортання: Docker Compose для ізольованого запуску серверної та адміністративної частин;
- база метаданих: SQLite із таблицями користувачів, файлів, сесій, передач і журналів;
- файлове сховище: окремі каталоги для завершених файлів, тимчасових блоків і службових журналів;
- інструменти тестування: pytest, тестовий клієнт, журнал подій, контрольні файли різного розміру;
- мережеві умови: локальна мережа або localhost-з'єднання з можливістю імітації розриву сесії та паралельних підключень.

Для перевірки використовуються контрольні файли різного обсягу: малий файл для швидкого функціонального тесту, файл середнього розміру для перевірки блокового передавання та файл більшого розміру для оцінювання стабільності під час тривалішого обміну. Додатково формується файл із навмисно зміненим блоком, що дозволяє перевірити реакцію сервера на порушення цілісності даних.

3. Тест-кейси

Тест-кейси сформовано відповідно до функціональних і нефункціональних характеристик програмної розробки. Кожен тестовий випадок має визначену мету, послідовність дій, очікуваний результат і критерій успішності. Перелік тест-кейсів подано у вигляді послідовних сценаріїв, оскільки кожний сценарій перевіряє окремий аспект роботи системи та може виконуватися незалежно від інших перевірок.

3.1. Запуск системи

Мета тесту полягає у перевірці готовності серверної частини, адміністративного сервісу, SQLite-бази метаданих і файлового сховища. Тестувальник запускає середовище, перевіряє доступність портів, наявність службових каталогів і відсутність помилок ініціалізації. Очікуваним результатом є стабільний запуск усіх компонентів без аварійного завершення процесів.

3.2. Перевірка доступності сервера

Клієнт встановлює TCP-з'єднання із сервером і надсилає команду PING. Сервер має повернути коректну службову відповідь, що підтверджує доступність мережевого каналу, готовність обробника команд і працездатність базового циклу приймання запитів.

3.3. Автентифікація користувача

Перевіряється команда LOGIN із коректними й некоректними обліковими даними. У першому випадку сервер має сформувати сесійний токен, у другому — повернути повідомлення про помилку без створення активної сесії. Додатково перевіряється, що неавторизований клієнт не може виконувати операції передавання файлів.

3.4. Отримання списку файлів

Авторизований клієнт надсилає команду LIST для отримання переліку доступних файлів. Очікуваним результатом є відповідь сервера зі структурованим списком об'єктів, який узгоджується із записами SQLite-бази метаданих і фактичним станом файлового сховища.

3.5. Отримання метаданих файла

Перевіряється команда INFO для наявного файла. Сервер має повернути назву, розмір, контрольну суму SHA-256, дату створення або оновлення та службові параметри, потрібні для подальшого отримання файла. Запит до неіснуючого файла має завершуватися контрольованою помилкою.

3.6. Передавання файла на сервер

Клієнт ініціює завантаження через команду UPLOAD_INIT, після чого передає файл блоками командою CHUNK і завершує операцію через UPLOAD_FINISH. Сервер має створити тимчасовий об'єкт, підтвердити приймання коректних блоків, перемістити завершений файл до основного сховища й оновити метадані.

3.7. Контроль цілісності SHA-256

Після завершення передачі обчислюється контрольна сума отриманого файла та порівнюється з хешем вихідного об'єкта. Тест вважається успішним, якщо значення SHA-256 збігаються, розмір файла не змінюється, а сервер фіксує операцію як завершену.

3.8. Обробка некоректного блока

Клієнт надсилає блок із неправильним номером, неповним вмістом або некоректною контрольною сумою. Сервер має повернути NACK, не включити пошкоджений блок до завершеного файла та залишити сесію в керованому стані для повторного надсилання фрагмента.

3.9. Відновлення незавершеної передачі

Імітується розрив з'єднання під час передавання файла. Після повторного підключення клієнт надсилає команду `UPLOAD_RESUME`, отримує від сервера інформацію про останній підтверджений блок і продовжує передачу без повторного надсилання всього файла.

3.10. Завантаження файла із сервера

Клієнт надсилає команду `DOWNLOAD` для файла, який збережено в основному сховищі. Після отримання об'єкта клієнт локально обчислює SHA-256 і порівнює його із серверним значенням. Очікуваний результат — повна відповідність отриманого файла серверному оригіналу.

3.11. Обробка помилкових команд

Сервер перевіряється на стійкість до невідомих команд, неповних параметрів і некоректних форматів повідомлень. Очікуваним результатом є службова відповідь про помилку без аварійного завершення сервера, втрати активних сесій або пошкодження метаданих.

3.12. Перевірка рольового розмежування доступу

Користувач без адміністративних прав намагається виконати службову або адміністративну дію. Система має відхилити операцію, зберегти незмінним стан файлів і метаданих та зафіксувати подію в журналі.

3.13. Перевірка журналювання

Після виконання базових сценаріїв перевіряється наявність записів про підключення клієнта, автентифікацію, початок і завершення передачі, помилки протоколу, перевірку SHA-256 та результат операції. Журнал має містити достатньо даних для відтворення послідовності подій.

3.14. Паралельна робота клієнтів

Кілька клієнтів одночасно передають файли на сервер. Перевіряється здатність ThreadPoolExecutor розподіляти клієнтські сесії між робочими потоками без конфліктів запису, блокування всього сервера або втрати службових відповідей.

3.15. Автоматизований набір pytest

Запускається автоматизований набір перевірок, що охоплює базові команди протоколу, обробку помилок, контроль цілісності й типові сценарії файлового обміну. Успішним результатом є проходження всіх тестів без помилок і відтворюваність результату під час повторного запуску.

Очікуваним підсумком виконання тест-кейсів є підтвердження того, що система виконує повний цикл обміну файлами, не приймає пошкоджені дані як коректні, не завершує роботу після помилкових запитів і зберігає узгоджений стан метаданих під час паралельної роботи клієнтів.

4. Виконання тестування

Виконання тестування здійснюється за попередньо визначеним порядком, який забезпечує відтворюваність результатів. Перед кожним тестовим запуском перевіряється початковий стан файлового сховища, доступність бази метаданих, очищення тимчасових файлів і готовність журналу подій до запису нової сесії.

4.1. Підготовка тестового середовища

Підготовка тестового середовища починається з перевірки структури проекту, наявності конфігураційних файлів, доступності портів і коректності змінних середовища. Після цього запускаються контейнеризовані компоненти системи, створюються або перевіряються каталоги для збереження файлів,

ініціалізується SQLite-база метаданих і перевіряється доступність адміністративного сервісу.

Перед функціональним тестуванням формується набір контрольних файлів. Для кожного файла заздалегідь визначається розмір і контрольна сума SHA-256, що дозволяє порівняти вихідний об'єкт із результатом передавання. Для негативного тестування готується файл або окремий блок із навмисно зміненими даними, щоб перевірити здатність системи виявляти порушення цілісності.

4.2. Запуск тестових сценаріїв

Запуск тестових сценаріїв відбувається від простих перевірок до складніших інтеграційних ситуацій. Спочатку виконується перевірка запуску сервера та команди PING. Далі тестується автентифікація користувача, отримання списку файлів і запит метаданих. Після підтвердження базової працездатності виконується повний цикл завантаження файла на сервер із поділом на блоки.

Під час тестування передавання файла контролюється кожен етап: ініціація операції, створення тимчасового файла, приймання блоків, формування ACK/NACK, завершення через UPLOAD_FINISH, переміщення файла до постійного сховища, оновлення метаданих і запис події в журнал. Окремо перевіряється завантаження файла із сервера та відповідність локального SHA-256 серверному значенню.

Навантажувальний сценарій передбачає одночасне підключення кількох клієнтів. У цьому режимі перевіряється, чи сервер не блокує приймання нових з'єднань під час активної передачі, чи не виникають конфлікти доступу до спільних ресурсів і чи правильно журналюються паралельні сесії.

4.3. Реєстрація результатів

Результати кожного тестового сценарію фіксуються в тестовому журналі. Для кожного випадку зазначаються дата й час запуску, назва тесту, вхідні умови, послідовність дій, фактичний результат, очікуваний результат, статус виконання та коментар щодо виявлених відхилень. Записи журналу порівнюються з даними SQLite, що дозволяє перевірити узгодженість службових подій і метаданих.

Автоматизовані тести формують окремий протокол виконання, у якому зазначається кількість виконаних перевірок, кількість успішних результатів і час проходження набору. Під час демонстраційного тестування набір `pytest` використовується як засіб підтвердження повторюваності перевірок після змін у програмному коді.

4.4. Виявлення помилок

Виявлені помилки класифікуються за впливом на роботу системи. Критичними вважаються дефекти, що призводять до аварійного завершення сервера, пошкодження файла або втрати метаданих. Високий пріоритет мають помилки автентифікації, некоректне розмежування доступу, помилки контрольних сум і некоректне відновлення передачі. Середній пріоритет надається помилкам інтерфейсу адміністратора, неповному журналюванню або неточним службовим повідомленням. Низький пріоритет отримують дефекти оформлення, які не впливають на цілісність передавання.

Під час контрольного тестування критичних помилок не зафіксовано. Некоректні команди, запити до відсутніх файлів і пошкоджені блоки обробляються через службові відповіді, а не через завершення серверного процесу. Така поведінка відповідає вимогам до стійкості системи в умовах помилкових або неповних клієнтських запитів.

5. Виправлення помилок

Виправлення помилок передбачає встановлення причини дефекту, локалізацію відповідного модуля, внесення змін до коду або конфігурації, повторне збирання середовища та перевірку сценарію, у межах якого була виявлена проблема. Для помилок прикладного протоколу перевіряється правильність парсингу команди, валідації параметрів і формування відповіді. Для помилок передавання файла аналізуються номери блоків, контрольні суми, стан тимчасового файла та коректність оновлення метаданих.

Після виправлення кожного дефекту обов'язково виконується регресійна перевірка суміжних сценаріїв. Наприклад, зміни у механізмі приймання блоків перевіряються не лише через тест передавання файла, а й через сценарії

ACK/NACK, UPLOAD_RESUME, DOWNLOAD і паралельного завантаження кількома клієнтами.

6. Повторне тестування

Повторне тестування виконується після усунення виявлених помилок або зміни конфігурації середовища. Основним завданням повторної перевірки є підтвердження того, що внесені зміни усунули дефект і не спричинили нових відхилень у суміжних компонентах. Для цього повторно запускаються ключові функціональні, негативні та навантажувальні сценарії.

Під час повторного тестування особлива увага приділяється сценаріям, пов'язаним із цілісністю файлів, відновленням передачі та паралельним доступом до спільних ресурсів. Якщо всі контрольні суми збігаються, журнали містять повні записи, а сервер не завершує роботу під час некоректних запитів, зміни вважаються успішно перевіреними.

7. Висновки тестування

Проведене тестування підтверджує працездатність клієнт-серверної системи обміну файлами з підтримкою багатопоточності засобами Python. Система коректно встановлює мережеве з'єднання, виконує автентифікацію, обробляє службові команди, передає файли блоками, підтверджує приймання фрагментів, перевіряє цілісність за SHA-256, зберігає метадані в SQLite і фіксує події в журналі.

Результати функціонального, інтеграційного, негативного та навантажувального тестування свідчать, що серверна частина здатна обслуговувати кілька клієнтських сесій без взаємного блокування й некоректного запису даних. Помилкові команди та запити до відсутніх об'єктів обробляються штатно, а пошкоджені блоки не приймаються як коректні. Застосування автоматизованих тестів підвищує повторюваність перевірки та дає змогу швидко підтверджувати стабільність системи після змін у програмному коді.

Загальний результат тестування дозволяє вважати програмну розробку готовою до демонстрації в межах кваліфікаційної роботи та придатною для подальшого розширення: додавання повнішої політики доступу, розширення

адміністративної панелі, масштабування серверної частини й удосконалення засобів моніторингу продуктивності.

ДОДАТОК Г

Вихідний код скриптів

У додатку наведено вихідний код основних скриптів, які організовують роботу клієнт-серверної системи обміну файлами: серверний TCP-модуль, модулі прикладного протоколу, підсистема роботи з файловим сховищем, база метаданих SQLite, клієнтський застосунок, адміністративний вебсервіс та автоматизовані тестові сценарії.

Лістинг Г.1 – server/config.py

```
from pathlib import Path
import os

BASE_DIR = Path(__file__).resolve().parents[1]

SERVER_HOST = os.getenv("FILE_SERVER_HOST", "0.0.0.0")
SERVER_PORT = int(os.getenv("FILE_SERVER_PORT", "9000"))
MAX_WORKERS = int(os.getenv("FILE_SERVER_MAX_WORKERS",
"20"))
CHUNK_SIZE = int(os.getenv("FILE_SERVER_CHUNK_SIZE",
str(64 * 1024)))
SOCKET_TIMEOUT =
float(os.getenv("FILE_SERVER_SOCKET_TIMEOUT", "120"))
USE_TLS = os.getenv("FILE_SERVER_USE_TLS",
"false").lower() in {"1", "true", "yes", "on"}

STORAGE_DIR = Path(os.getenv("FILE_SERVER_STORAGE_DIR",
BASE_DIR / "storage"))
FILES_DIR = STORAGE_DIR / "files"
TMP_DIR = STORAGE_DIR / "tmp"
DOWNLOADS_DIR =
Path(os.getenv("FILE_CLIENT_DOWNLOADS_DIR", BASE_DIR /
"downloads"))
LOG_DIR = Path(os.getenv("FILE_SERVER_LOG_DIR", BASE_DIR /
"logs"))
LOG_FILE = LOG_DIR / "server.log"
DATABASE_PATH = Path(os.getenv("FILE_SERVER_DB_PATH",
STORAGE_DIR / "metadata.db"))
CERTS_DIR = Path(os.getenv("FILE_SERVER_CERTS_DIR",
BASE_DIR / "certs"))
TLS_CERT_FILE = Path(os.getenv("FILE_SERVER_TLS_CERT",
CERTS_DIR / "server.crt"))
TLS_KEY_FILE = Path(os.getenv("FILE_SERVER_TLS_KEY",
CERTS_DIR / "server.key"))

MAX_JSON_HEADER =
int(os.getenv("FILE_SERVER_MAX_JSON_HEADER", str(1024 *
1024)))

def ensure_directories() -> None:
    for directory in (STORAGE_DIR, FILES_DIR, TMP_DIR,
DOWNLOADS_DIR, LOG_DIR, CERTS_DIR):
        directory.mkdir(parents=True, exist_ok=True)
```

Лістинг Г.2 – server/logger_config.py

```
import logging
from logging.handlers import RotatingFileHandler

from server.config import LOG_FILE, ensure_directories

def setup_logging() -> logging.Logger:
    ensure_directories()
    logger = logging.getLogger("file_exchange_server")
    logger.setLevel(logging.INFO)
    logger.propagate = False

    if logger.handlers:
        return logger

    formatter = logging.Formatter(
        "%(asctime)s | %(levelname)s | %(threadName)s |
%(message)s"
    )

    file_handler = RotatingFileHandler(
        LOG_FILE, maxBytes=2_000_000, backupCount=5,
encoding="utf-8"
    )
    file_handler.setFormatter(formatter)
    logger.addHandler(file_handler)

    console_handler = logging.StreamHandler()
    console_handler.setFormatter(formatter)
    logger.addHandler(console_handler)

    return logger
```

Лістинг Г.3 – server/protocol.py

```
import json
import socket
import struct
from typing import Any

from server.config import MAX_JSON_HEADER

class ProtocolError(Exception):
    pass

def recv_exact(sock: socket.socket, size: int) -> bytes:
    data = bytearray()
    while len(data) < size:
        chunk = sock.recv(size - len(data))
        if not chunk:
            raise ConnectionError("Connection closed while
receiving data")
        data.extend(chunk)
    return bytes(data)

def send_json(sock: socket.socket, payload: dict[str,
Any]) -> None:
    encoded = json.dumps(payload,
ensure_ascii=False).encode("utf-8")
    if len(encoded) > MAX_JSON_HEADER:
        raise ProtocolError("JSON header is too large")
    sock.sendall(struct.pack("!I", len(encoded)))
    sock.sendall(encoded)

def recv_json(sock: socket.socket) -> dict[str, Any]:
    raw_length = recv_exact(sock, 4)
    length = struct.unpack("!I", raw_length)[0]
    if length <= 0 or length > MAX_JSON_HEADER:
        raise ProtocolError(f"Invalid JSON header length:
{length}")
    raw_payload = recv_exact(sock, length)
    try:
        payload = json.loads(raw_payload.decode("utf-8"))
    except json.JSONDecodeError as exc:
        raise ProtocolError("Invalid JSON payload") from
exc
    if not isinstance(payload, dict):
        raise ProtocolError("JSON payload must be an
object")
    return payload
```

Лістинг Г.4 – server/storage.py

```
import hashlib
import os
import re
import shutil
import uuid
from pathlib import Path

from server.config import FILES_DIR, TMP_DIR,
ensure_directories

SAFE_NAME_PATTERN = re.compile(r"^[A-Za-z0-9._-]+$")

def sanitize_filename(filename: str) -> str:
    name = Path(filename).name.strip()
    name = SAFE_NAME_PATTERN.sub("_", name)
    name = name.strip("_")
    if not name:
        return "file"
    return name[:180]

def unique_stored_name(original_name: str) -> str:
    safe = sanitize_filename(original_name)
    return f"{uuid.uuid4().hex}_{safe}"

def tmp_path_for(stored_name: str) -> Path:
    ensure_directories()
    return TMP_DIR / f"{stored_name}.part"
```

```

def final_path_for(stored_name: str) -> Path:
    ensure_directories()
    return FILES_DIR / stored_name

def move_to_storage(tmp_path: Path, stored_name: str) -> Path:
    ensure_directories()
    final_path = final_path_for(stored_name)
    shutil.move(str(tmp_path), final_path)
    return final_path

def remove_quietly(path: Path) -> None:
    try:
        path.unlink(missing_ok=True)
    except OSError:
        pass

def delete_file(path: Path) -> bool:
    try:
        path.unlink()
        return True
    except FileNotFoundError:
        return False

def sha256_file(path: Path, chunk_size: int = 1024 * 1024) -> str:
    digest = hashlib.sha256()
    with path.open("rb") as file_obj:
        while True:
            chunk = file_obj.read(chunk_size)
            if not chunk:
                break
            digest.update(chunk)
    return digest.hexdigest()

def file_size(path: Path) -> int:
    return os.path.getsize(path)

```

Лістинг Г.5 – server/metadata_db.py

```

import hashlib
import os
import sqlite3
import threading
import uuid
from pathlib import Path
from typing import Any

from server.config import DATABASE_PATH,
ensure_directories

PASSWORD_ITERATIONS = 200_000

def hash_password(password: str) -> str:
    salt = os.urandom(16)
    digest = hashlib.pbkdf2_hmac(
        "sha256", password.encode("utf-8"), salt,
        PASSWORD_ITERATIONS
    )
    return
f"pbkdf2_sha256${PASSWORD_ITERATIONS}${salt.hex()}${digest
.hex()}"

def verify_password(password: str, stored_hash: str) ->
bool:
    try:
        algorithm, iterations, salt_hex, digest_hex =
stored_hash.split("$", 3)
        if algorithm != "pbkdf2_sha256":
            return False
        digest = hashlib.pbkdf2_hmac(
            "sha256",
            password.encode("utf-8"),
            bytes.fromhex(salt_hex),
            int(iterations),

```

```

    )
    return digest.hex() == digest_hex
except (ValueError, TypeError):
    return False

class MetadataDB:
    def __init__(self, db_path: Path = DATABASE_PATH):
        ensure_directories()
        self.db_path = db_path
        self._lock = threading.RLock()
        self._connection = sqlite3.connect(
            self.db_path, check_same_thread=False,
            isolation_level=None
        )
        self._connection.row_factory = sqlite3.Row
        self._initialize()

    def _initialize(self) -> None:
        with self._lock:
            self._connection.execute(
                """
                CREATE TABLE IF NOT EXISTS users (
                    id INTEGER PRIMARY KEY AUTOINCREMENT,
                    username TEXT NOT NULL UNIQUE,
                    password_hash TEXT NOT NULL,
                    role TEXT NOT NULL CHECK(role IN
('admin', 'user')),
                    created_at TEXT NOT NULL DEFAULT
CURRENT_TIMESTAMP
                )
                """
            )
            self._connection.execute(
                """
                CREATE TABLE IF NOT EXISTS files (
                    id INTEGER PRIMARY KEY AUTOINCREMENT,
                    original_name TEXT NOT NULL,
                    stored_name TEXT NOT NULL UNIQUE,
                    size_bytes INTEGER NOT NULL,
                    sha256 TEXT NOT NULL,
                    upload_time TEXT NOT NULL DEFAULT
CURRENT_TIMESTAMP,
                    status TEXT NOT NULL,
                    client_address TEXT NOT NULL,
                    path TEXT NOT NULL,
                    owner_id INTEGER REFERENCES users(id)
                )
                """
            )
            self._ensure_column("files", "owner_id",
"INTEGER REFERENCES users(id)")
            self._connection.execute(
                """
                CREATE TABLE IF NOT EXISTS
transfer_sessions (
                    session_id TEXT PRIMARY KEY,
                    original_name TEXT NOT NULL,
                    stored_name TEXT NOT NULL,
                    tmp_path TEXT NOT NULL,
                    expected_size INTEGER NOT NULL,
                    expected_sha256 TEXT NOT NULL,
                    received_bytes INTEGER NOT NULL
DEFAULT 0,
                    status TEXT NOT NULL,
                    owner_id INTEGER NOT NULL REFERENCES
users(id),
                    client_address TEXT NOT NULL,
                    created_at TEXT NOT NULL DEFAULT
CURRENT_TIMESTAMP,
                    updated_at TEXT NOT NULL DEFAULT
CURRENT_TIMESTAMP
                )
                """
            )
            self._connection.execute(
                "CREATE INDEX IF NOT EXISTS
idx_files_original_name ON files(original_name)"
            )
            self._connection.execute(
                "CREATE INDEX IF NOT EXISTS
idx_files_status ON files(status)"
            )
            self._connection.execute(
                "CREATE INDEX IF NOT EXISTS
idx_files_owner_id ON files(owner_id)"
            )

```

```

        )
        self._connection.execute(
            "CREATE INDEX IF NOT EXISTS
idx_transfer_owner_status ON transfer_sessions(owner_id,
status)"
        )
        self._ensure_default_users()

    def _ensure_column(self, table: str, column: str,
definition: str) -> None:
        columns = {
            row["name"]
            for row in self._connection.execute(f"PRAGMA
table_info({table})").fetchall()
        }
        if column not in columns:
            self._connection.execute(f"ALTER TABLE {table}
ADD COLUMN {column} {definition}")

    def _ensure_default_users(self) -> None:
        defaults = [
            ("admin", "admin123", "admin"),
            ("user", "user123", "user"),
            ("user2", "user123", "user"),
        ]
        for username, password, role in defaults:
            row = self._connection.execute(
                "SELECT id FROM users WHERE username = ?",
                (username,)
            ).fetchone()
            if not row:
                self._connection.execute(
                    "INSERT INTO users (username,
password_hash, role) VALUES (?, ?, ?)",
                    (username, hash_password(password),
role),
                )

    def authenticate_user(self, username: str, password:
str) -> dict[str, Any] | None:
        with self._lock:
            row = self._connection.execute(
                "SELECT * FROM users WHERE username = ?",
                (username,)
            ).fetchone()
            if not row or not verify_password(password,
row["password_hash"]):
                return None
            user = dict(row)
            user.pop("password_hash", None)
            return user

    def get_user(self, self, user_id: int) -> dict[str, Any] |
None:
        with self._lock:
            row = self._connection.execute(
                "SELECT id, username, role, created_at
FROM users WHERE id = ?", (user_id,)
            ).fetchone()
            return dict(row) if row else None

    def add_file(
        self,
        original_name: str,
        stored_name: str,
        size_bytes: int,
        sha256: str,
        client_address: str,
        path: str,
        owner_id: int,
        status: str = "uploaded",
    ) -> dict[str, Any]:
        with self._lock:
            cursor = self._connection.execute(
                """
INSERT INTO files (
    original_name, stored_name,
size_bytes, sha256,
    status, client_address, path, owner_id
)
VALUES (?, ?, ?, ?, ?, ?, ?, ?)
""",
                (
                    original_name,
                    stored_name,
                    size_bytes,
                    sha256,
                    status, client_address, path, owner_id
                )
            )

        sha256,
        status,
        client_address,
        path,
        owner_id,
    ),
    )
    return self.get_by_id(cursor.lastrowid,
include_deleted=True)

    def list_files(
        self, include_deleted: bool = False, user:
dict[str, Any] | None = None
    ) -> list[dict[str, Any]]:
        query = "SELECT * FROM files"
        conditions: list[str] = []
        params: list[Any] = []

        if not include_deleted:
            conditions.append("status != ?")
            params.append("deleted")
        if user and user.get("role") != "admin":
            conditions.append("owner_id = ?")
            params.append(user["id"])

        if conditions:
            query += " WHERE " + " AND ".join(conditions)
        query += " ORDER BY upload_time DESC, id DESC"
        with self._lock:
            rows = self._connection.execute(query,
tuple(params)).fetchall()
            return [dict(row) for row in rows]

    def get_by_id(
        self,
        file_id: int,
        include_deleted: bool = False,
        user: dict[str, Any] | None = None,
    ) -> dict[str, Any] | None:
        query = "SELECT * FROM files WHERE id = ?"
        params: list[Any] = [file_id]
        if not include_deleted:
            query += " AND status != ?"
            params.append("deleted")
        if user and user.get("role") != "admin":
            query += " AND owner_id = ?"
            params.append(user["id"])
        with self._lock:
            row = self._connection.execute(query,
tuple(params)).fetchone()
            return dict(row) if row else None

    def find_file(
        self,
        identifier: str | int,
        include_deleted: bool = False,
        user: dict[str, Any] | None = None,
    ) -> dict[str, Any] | None:
        text_identifier = str(identifier)
        if text_identifier.isdigit():
            by_id = self.get_by_id(
                int(text_identifier),
                include_deleted=include_deleted, user=user
            )
            if by_id:
                return by_id

        query = """
SELECT * FROM files
WHERE (original_name = ? OR stored_name = ?)
"""
        params: list[Any] = [text_identifier,
text_identifier]
        if not include_deleted:
            query += " AND status != ?"
            params.append("deleted")
        if user and user.get("role") != "admin":
            query += " AND owner_id = ?"
            params.append(user["id"])
        query += " ORDER BY upload_time DESC, id DESC
LIMIT 1"

        with self._lock:
            row = self._connection.execute(query,
tuple(params)).fetchone()
            return dict(row) if row else None

```

```

def mark_deleted(
    self, identifier: str | int, user: dict[str, Any]
    | None = None
) -> dict[str, Any] | None:
    file_record = self.find_file(identifier,
user=user)
    if not file_record:
        return None
    with self._lock:
        self._connection.execute(
            "UPDATE files SET status = ? WHERE id =
?",
            ("deleted", file_record["id"]),
        )
    return self.get_by_id(file_record["id"],
include_deleted=True)

def create_transfer(
    self,
    original_name: str,
    stored_name: str,
    tmp_path: str,
    expected_size: int,
    expected_sha256: str,
    owner_id: int,
    client_address: str,
) -> dict[str, Any]:
    session_id = uuid.uuid4().hex
    with self._lock:
        self._connection.execute(
            """
            INSERT INTO transfer_sessions (
                session_id, original_name,
stored_name, tmp_path,
                expected_size, expected_sha256,
received_bytes,
                status, owner_id, client_address
            )
VALUES (?, ?, ?, ?, ?, ?, 0, 'receiving',
?, ?)
            """,
            (
                session_id,
                original_name,
                stored_name,
                tmp_path,
                expected_size,
                expected_sha256,
                owner_id,
                client_address,
            ),
        )
    return self.get_transfer(session_id)

def get_transfer(
    self, session_id: str, user: dict[str, Any] | None
= None
) -> dict[str, Any] | None:
    query = "SELECT * FROM transfer_sessions WHERE
session_id = ?"
    params: list[Any] = [session_id]
    if user and user.get("role") != "admin":
        query += " AND owner_id = ?"
        params.append(user["id"])
    with self._lock:
        row = self._connection.execute(query,
tuple(params)).fetchone()
        return dict(row) if row else None

def update_transfer_progress(
    self, session_id: str, received_bytes: int,
status: str = "receiving"
) -> None:
    with self._lock:
        self._connection.execute(
            """
            UPDATE transfer_sessions
            SET received_bytes = ?, status = ?,
updated_at = CURRENT_TIMESTAMP
            WHERE session_id = ?
            """,
            (received_bytes, status, session_id),
        )

def close(self) -> None:

```

```

with self._lock:
    self._connection.close()

```

Лістинг 7.6 – server/server.py

```

from __future__ import annotations

import hashlib
import secrets
import signal
import socket
import ssl
import threading
from concurrent.futures import ThreadPoolExecutor
from pathlib import Path
from typing import Any

from server import storage
from server.config import (
    CHUNK_SIZE,
    MAX_WORKERS,
    SERVER_HOST,
    SERVER_PORT,
    SOCKET_TIMEOUT,
    TLS_CERT_FILE,
    TLS_KEY_FILE,
    USE_TLS,
    ensure_directories,
)
from server.logger_config import setup_logging
from server.metadata_db import MetadataDB
from server.protocol import ProtocolError, recv_exact,
recv_json, send_json

class FileExchangeServer:
    def __init__(
        self,
        host: str = SERVER_HOST,
        port: int = SERVER_PORT,
        max_workers: int = MAX_WORKERS,
        use_tls: bool = USE_TLS,
    ):
        ensure_directories()
        self.host = host
        self.port = port
        self.max_workers = max_workers
        self.use_tls = use_tls
        self.logger = setup_logging()
        self.db = MetadataDB()
        self.shutdown = threading.Event()
        self.sessions: set[str] = set()
        self.sessions_lock = threading.RLock()
        self.storage_lock = threading.RLock()
        self.tokens: dict[str, dict[str, Any]] = {}
        self.tokens_lock = threading.RLock()
        self._server_socket: socket.socket | None = None

    def start(self) -> None:
        tls_context = self._create_tls_context() if
self.use_tls else None
        self.logger.info(
            "Starting server on %s:%s (TLS=%s)",
self.host, self.port, self.use_tls
        )
        with socket.socket(socket.AF_INET,
socket.SOCK_STREAM) as server_socket:
            server_socket.setsockopt(socket.SOL_SOCKET,
socket.SO_REUSEADDR, 1)
            server_socket.bind((self.host, self.port))
            server_socket.listen()
            server_socket.settimeout(1.0)
            self._server_socket = server_socket

        with
ThreadPoolExecutor(max_workers=self.max_workers) as
executor:
            while not self.shutdown.is_set():
                try:
                    client_socket, address =
server_socket.accept()
                except socket.timeout:
                    continue
                except OSError:
                    break

```

```

        if tls_context:
            try:
                client_socket =
tls_context.wrap_socket(
                    client_socket,
server_side=True
            )
            except ssl.SSLError as exc:
                self.logger.warning("TLS
handshake failed from %s: %s", address, exc)
                client_socket.close()
                continue
            executor.submit(self._handle_client,
client_socket, address)

        self.db.close()
        self.logger.info("Server stopped")

    def stop(self) -> None:
        self._shutdown.set()
        if self._server_socket:
            try:
                self._server_socket.close()
            except OSError:
                pass

    def _create_tls_context(self) -> ssl.SSLContext:
        if not TLS_CERT_FILE.exists() or not
TLS_KEY_FILE.exists():
            raise FileNotFoundError(
                f"TLS certificate or key not found:
{TLS_CERT_FILE}, {TLS_KEY_FILE}"
            )
        context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
        context.load_cert_chain(certfile=TLS_CERT_FILE,
keyfile=TLS_KEY_FILE)
        return context

    def _session_name(self, address: tuple[str, int]) ->
str:
        return f"{address[0]}:{address[1]}"

    def _handle_client(
        self, client_socket: socket.socket, address:
tuple[str, int]
    ) -> None:
        session = self._session_name(address)
        client_socket.settimeout(SOCKET_TIMEOUT)
        with self._sessions_lock:
            self._sessions.add(session)
            self.logger.info("Client connected: %s", session)

        try:
            with client_socket:
                while not self._shutdown.is_set():
                    try:
                        request = recv_json(client_socket)
                    except ConnectionError:
                        self.logger.info("Client
disconnected: %s", session)
                        break
                    except (socket.timeout, ProtocolError)
as exc:
                        self.logger.warning("Protocol
error from %s: %s", session, exc)
                        self._safe_send(client_socket,
self._error(str(exc)))
                        break

                    command = str(request.get("command",
"")).upper()
                    self.logger.info("Command from %s:
%s", session, command)

                    if command == "QUIT":
                        send_json(client_socket,
{"status": "OK", "message": "Session closed"})
                        self.logger.info("Client session
closed: %s", session)
                        break
                    self._dispatch(client_socket, request,
session)
                except Exception as exc:
                    self.logger.exception("Unexpected error in
session %s: %s", session, exc)
            finally:

```

```

        with self._sessions_lock:
            self._sessions.discard(session)

    def _dispatch(self, sock: socket.socket, request:
dict[str, Any], session: str) -> None:
        command = str(request.get("command", "")).upper()
        handlers = {
            "LOGIN": self._handle_login,
            "PING": self._handle_ping,
            "LIST": self._handle_list,
            "INFO": self._handle_info,
            "DELETE": self._handle_delete,
            "UPLOAD": self._handle_upload_init,
            "UPLOAD_INIT": self._handle_upload_init,
            "UPLOAD_RESUME": self._handle_upload_resume,
            "DOWNLOAD": self._handle_download,
        }
        handler = handlers.get(command)
        if not handler:
            send_json(sock, self._error(f"Unknown command:
{command or '<empty>'"}"))
            return
        handler(sock, request, session)

    def _handle_ping(self, sock: socket.socket, request:
dict[str, Any], session: str) -> None:
        send_json(sock, {"status": "OK", "message":
"PONG"})

    def _handle_login(self, sock: socket.socket, request:
dict[str, Any], session: str) -> None:
        username = str(request.get("username", ""))
        password = str(request.get("password", ""))
        user = self.db.authenticate_user(username,
password)
        if not user:
            self.logger.warning("Failed LOGIN from %s for
username=%s", session, username)
            send_json(sock, self._error("Invalid username
or password"))
            return

        token = secrets.token_urlsafe(32)
        with self._tokens_lock:
            self._tokens[token] = user
        self.logger.info("Successful LOGIN from %s as %s
(%s)", session, user["username"], user["role"])
        send_json(sock, {"status": "OK", "token": token,
"user": user})

    def _require_user(
        self, sock: socket.socket, request: dict[str,
Any], session: str
    ) -> dict[str, Any] | None:
        token = request.get("token")
        if not isinstance(token, str):
            self.logger.warning("Auth required for %s",
session)
            send_json(sock, self._error("Authentication
required"))
            return None
        with self._tokens_lock:
            user = self._tokens.get(token)
        if not user:
            self.logger.warning("Invalid token from %s",
session)
            send_json(sock, self._error("Invalid or
expired session token"))
            return None
        return user

    def _handle_list(self, sock: socket.socket, request:
dict[str, Any], session: str) -> None:
        user = self._require_user(sock, request, session)
        if not user:
            return
        send_json(sock, {"status": "OK", "files":
self.db.list_files(user=user)})

    def _handle_info(self, sock: socket.socket, request:
dict[str, Any], session: str) -> None:
        user = self._require_user(sock, request, session)
        if not user:
            return
        identifier = request.get("identifier")
        if identifier is None:

```

```

        send_json(sock, self._error("INFO requires
identifier"))
        return
        file_record = self.db.find_file(identifier,
user=user)
        if not file_record:
            self.logger.warning("Access denied or file not
found for %s: %s", user["username"], identifier)
            send_json(sock, self._error("File not found or
access denied"))
            return
        send_json(sock, {"status": "OK", "file":
file_record})

        def _handle_delete(self, sock: socket.socket, request:
dict[str, Any], session: str) -> None:
            user = self._require_user(sock, request, session)
            if not user:
                return
            identifier = request.get("identifier")
            if identifier is None:
                send_json(sock, self._error("DELETE requires
identifier"))
                return
            file_record = self.db.find_file(identifier,
user=user)
            if not file_record:
                self.logger.warning("DELETE denied or file not
found for %s: %s", user["username"], identifier)
                send_json(sock, self._error("File not found or
access denied"))
                return
            with self._storage_lock:
                storage.delete_file(Path(file_record["path"]))
                deleted = self.db.mark_deleted(identifier,
user=user)
            self.logger.info("File deleted by %s (%s): %s",
user["username"], session, file_record["stored_name"])
            send_json(sock, {"status": "OK", "file": deleted})

            def _handle_upload_init(self, sock: socket.socket,
request: dict[str, Any], session: str) -> None:
                user = self._require_user(sock, request, session)
                if not user:
                    return
                try:
                    filename = str(request["filename"])
                    expected_size = int(request["size"])
                    expected_sha = str(request["sha256"]).lower()
                    chunk_size = int(request.get("chunk_size",
CHUNK_SIZE))
                except (KeyError, TypeError, ValueError):
                    send_json(sock, self._error("UPLOAD requires
filename, size and sha256"))
                    return

                if expected_size < 0 or chunk_size <= 0:
                    send_json(sock, self._error("Invalid file size
or chunk size"))
                    return

                original_name =
storage.sanitize_filename(filename)
                stored_name =
storage.unique_stored_name(original_name)
                tmp_path = storage.tmp_path_for(stored_name)
                transfer = self.db.create_transfer(
                    original_name=original_name,
                    stored_name=stored_name,
                    tmp_path=tmp_path,
                    expected_size=expected_size,
                    expected_sha256=expected_sha,
                    owner_id=int(user["id"]),
                    client_address=session,
                )

                self.logger.info(
                    "Upload session created by %s (%s):
transfer_id=%s file=%s size=%s",
                    user["username"],
                    session,
                    transfer["session_id"],
                    original_name,
                    expected_size,
                )
                send_json(

sock,
{
    "status": "READY",
    "transfer_id": transfer["session_id"],
    "offset": transfer["received_bytes"],
    "chunk_size": min(chunk_size, CHUNK_SIZE),
},
)
                self._receive_upload_chunks(sock,
transfer["session_id"], user, session)

            def _handle_upload_resume(self, sock: socket.socket,
request: dict[str, Any], session: str) -> None:
                user = self._require_user(sock, request, session)
                if not user:
                    return
                transfer_id = request.get("transfer_id")
                if not isinstance(transfer_id, str):
                    send_json(sock, self._error("UPLOAD_RESUME
requires transfer_id"))
                    return
                transfer = self.db.get_transfer(transfer_id,
user=user)
                if not transfer or transfer["status"] not in
{"receiving", "failed"}:
                    send_json(sock, self._error("Transfer session
not found or already completed"))
                    return

                tmp_path = Path(transfer["tmp_path"])
                offset = tmp_path.stat().st_size if
tmp_path.exists() else int(transfer["received_bytes"])
                if offset != int(transfer["received_bytes"]):
                    self.db.update_transfer_progress(transfer_id,
offset)
                self.logger.info(
                    "Upload resumed by %s (%s): transfer_id=%s
offset=%s",
                    user["username"],
                    session,
                    transfer_id,
                    offset,
                )
                send_json(
                    sock,
                    {
                        "status": "READY",
                        "transfer_id": transfer_id,
                        "offset": offset,
                        "chunk_size": CHUNK_SIZE,
                    },
                )
                self._receive_upload_chunks(sock, transfer_id,
user, session)

            def _receive_upload_chunks(
                self,
                sock: socket.socket,
                transfer_id: str,
                user: dict[str, Any],
                session: str,
            ) -> None:
                try:
                    while True:
                        request = recv_json(sock)
                        command = str(request.get("command",
"")).upper()
                        if command == "UPLOAD_FINISH":
                            self._finish_upload(sock, transfer_id,
user, session)
                            return
                        if command != "CHUNK":
                            send_json(sock, self._error("Expected
CHUNK or UPLOAD_FINISH"))
                            return
                        self._receive_one_chunk(sock, request,
transfer_id, user, session)
                    except ConnectionError as exc:
                        self.logger.warning(
                            "Upload interrupted: transfer_id=%s
user=%s error=%s",
                            transfer_id,
                            user["username"],
                            exc,
                        )
                except Exception as exc:

```

```

        self.logger.exception("Upload failed:
transfer_id=%s error=%s", transfer_id, exc)
        self._safe_send(sock, self._error(str(exc)))

    def _receive_one_chunk(
        self,
        sock: socket.socket,
        request: dict[str, Any],
        transfer_id: str,
        user: dict[str, Any],
        session: str,
    ) -> None:
        if request.get("transfer_id") != transfer_id:
            send_json(sock, {"status": "NACK", "message":
"Invalid transfer_id"})
            return
        transfer = self.db.get_transfer(transfer_id,
user=user)
        if not transfer:
            send_json(sock, {"status": "NACK", "message":
"Transfer not found"})
            return

        try:
            chunk_index = int(request["chunk_index"])
            offset = int(request["offset"])
            size = int(request["size"])
            chunk_sha = str(request["sha256"]).lower()
        except (KeyError, TypeError, ValueError):
            send_json(sock, {"status": "NACK", "message":
"Invalid chunk header"})
            return

        block = recv_exact(sock, size)
        actual_chunk_sha =
hashlib.sha256(block).hexdigest()
        current_offset = int(transfer["received_bytes"])

        if offset != current_offset:
            send_json(
                sock,
                {
                    "status": "NACK",
                    "message": "Unexpected offset",
                    "expected_offset": current_offset,
                },
            )
            self.logger.warning(
                "NACK offset transfer_id=%s chunk=%s
expected=%s got=%s",
                transfer_id,
                chunk_index,
                current_offset,
                offset,
            )
            return
        if actual_chunk_sha != chunk_sha:
            send_json(sock, {"status": "NACK", "message":
"Chunk checksum mismatch"})
            self.logger.warning("NACK checksum
transfer_id=%s chunk=%s", transfer_id, chunk_index)
            return

        tmp_path = Path(transfer["tmp_path"])
        tmp_path.parent.mkdir(parents=True, exist_ok=True)
        with self._storage_lock:
            with tmp_path.open("ab") as file_obj:
                file_obj.write(block)
            received_bytes = current_offset + len(block)
            self.db.update_transfer_progress(transfer_id,
received_bytes)

        if chunk_index % 32 == 0:
            self.logger.info(
                "ACK transfer_id=%s chunk=%s received=%s",
                transfer_id,
                chunk_index,
                received_bytes,
            )
        send_json(
            sock,
            {
                "status": "ACK",
                "transfer_id": transfer_id,
                "chunk_index": chunk_index,
                "received_bytes": received_bytes,
            },

```

```

        ),
    )

    def _finish_upload(
        self,
        sock: socket.socket,
        transfer_id: str,
        user: dict[str, Any],
        session: str,
    ) -> None:
        transfer = self.db.get_transfer(transfer_id,
user=user)
        if not transfer:
            send_json(sock, self._error("Transfer not
found"))
            return
        tmp_path = Path(transfer["tmp_path"])
        if not tmp_path.exists():
            send_json(sock, self._error("Temporary file
not found"))
            return

        actual_size = storage.file_size(tmp_path)
        actual_sha = storage.sha256_file(tmp_path)
        expected_size = int(transfer["expected_size"])
        expected_sha = str(transfer["expected_sha256"])
        if actual_size != expected_size or actual_sha !=
expected_sha:
            self.db.update_transfer_progress(transfer_id,
actual_size, status="failed")
            self.logger.warning(
                "Final checksum/size mismatch
transfer_id=%s size=%s/%s sha=%s/%s",
                transfer_id,
                actual_size,
                expected_size,
                actual_sha,
                expected_sha,
            )
            send_json(sock, self._error("Final checksum or
size mismatch"))
            return

        with self._storage_lock:
            final_path = storage.move_to_storage(tmp_path,
transfer["stored_name"])
            record = self.db.add_file(
                original_name=transfer["original_name"],
                stored_name=transfer["stored_name"],
                size_bytes=actual_size,
                sha256=actual_sha,
                client_address=session,
                path=str(final_path),
                owner_id=int(user["id"]),
            )
            self.db.update_transfer_progress(transfer_id,
actual_size, status="completed")

            self.logger.info(
                "Upload completed by %s (%s): transfer_id=%s
stored=%s",
                user["username"],
                session,
                transfer_id,
                transfer["stored_name"],
            )
            send_json(sock, {"status": "OK", "file": record,
"transfer_id": transfer_id})

    def _handle_download(self, sock: socket.socket,
request: dict[str, Any], session: str) -> None:
        user = self._require_user(sock, request, session)
        if not user:
            return
        identifier = request.get("identifier")
        if identifier is None:
            send_json(sock, self._error("DOWNLOAD requires
identifier"))
            return

        file_record = self.db.find_file(identifier,
user=user)
        if not file_record:
            self.logger.warning("DOWNLOAD denied or file
not found for %s: %s", user["username"], identifier)

```

```

        send_json(sock, self._error("File not found or
access denied"))
        return

    path = Path(file_record["path"])
    if not path.exists():
        send_json(sock, self._error("File metadata
exists, but file is absent in storage"))
        return

    self.logger.info(
        "Download started for %s by %s (%s)",
        file_record["stored_name"],
        user["username"],
        session,
    )
    send_json(sock, {"status": "OK", "file":
file_record, "chunk_size": CHUNK_SIZE})

    with path.open("rb") as file_obj:
        while True:
            block = file_obj.read(CHUNK_SIZE)
            if not block:
                break
            sock.sendall(block)

    self.logger.info(
        "Download completed for %s by %s (%s)",
        file_record["stored_name"],
        user["username"],
        session,
    )

    def safe_send(self, sock: socket.socket, payload:
dict[str, Any]) -> None:
        try:
            send_json(sock, payload)
        except OSError:
            pass

    def _error(self, message: str) -> dict[str, str]:
        return {"status": "ERROR", "message": message}

def main() -> None:
    server = FileExchangeServer()

    def handle_signal(signum: int, frame: object) -> None:
        server.logger.info("Shutdown signal received: %s",
signum)
        server.stop()

    signal.signal(signal.SIGINT, handle_signal)
    signal.signal(signal.SIGTERM, handle_signal)
    server.start()

if __name__ == "__main__":
    main()

```

Лістинг Г.7 – client/protocol.py

```

import json
import socket
import struct
from typing import Any

MAX_JSON_HEADER = 1024 * 1024

class ProtocolError(Exception):
    pass

def recv_exact(sock: socket.socket, size: int) -> bytes:
    data = bytearray()
    while len(data) < size:
        chunk = sock.recv(size - len(data))
        if not chunk:
            raise ConnectionError("Connection closed while
receiving data")
        data.extend(chunk)
    return bytes(data)

```

```

def send_json(sock: socket.socket, payload: dict[str,
Any]) -> None:
    encoded = json.dumps(payload,
ensure_ascii=False).encode("utf-8")
    if len(encoded) > MAX_JSON_HEADER:
        raise ProtocolError("JSON header is too large")
    sock.sendall(struct.pack("!I", len(encoded)))
    sock.sendall(encoded)

```

```

def recv_json(sock: socket.socket) -> dict[str, Any]:
    raw_length = recv_exact(sock, 4)
    length = struct.unpack("!I", raw_length)[0]
    if length <= 0 or length > MAX_JSON_HEADER:
        raise ProtocolError("Invalid JSON header length:
{length}")
    payload = json.loads(recv_exact(sock,
length).decode("utf-8"))
    if not isinstance(payload, dict):
        raise ProtocolError("JSON payload must be an
object")
    return payload

```

Лістинг Г.8 – client/utils.py

```

import hashlib
from pathlib import Path

def sha256_file(path: Path, chunk_size: int = 1024 * 1024)
-> str:
    digest = hashlib.sha256()
    with path.open("rb") as file_obj:
        while True:
            chunk = file_obj.read(chunk_size)
            if not chunk:
                break
            digest.update(chunk)
    return digest.hexdigest()

def unique_download_path(downloads_dir: Path, filename:
str) -> Path:
    downloads_dir.mkdir(parents=True, exist_ok=True)
    safe_name = Path(filename).name or "downloaded_file"
    candidate = downloads_dir / safe_name
    if not candidate.exists():
        return candidate

    stem = candidate.stem
    suffix = candidate.suffix
    counter = 1
    while True:
        candidate = downloads_dir /
f"{stem}_{counter}{suffix}"
        if not candidate.exists():
            return candidate
        counter += 1

```

Лістинг Г.9 – client/client.py

```

from __future__ import annotations

import argparse
import hashlib
import shlex
import socket
import ssl
import sys
import time
from pathlib import Path
from typing import Any

from client.protocol import recv_exact, recv_json,
send_json
from client.utils import sha256_file, unique_download_path

DEFAULT_HOST = "127.0.0.1"
DEFAULT_PORT = 9000
DEFAULT_CHUNK_SIZE = 64 * 1024
BASE_DIR = Path(__file__).resolve().parents[1]
DOWNLOADS_DIR = BASE_DIR / "downloads"

```

```

DEFAULT_CAFILE = BASE_DIR / "certs" / "server.crt"

class FileExchangeClient:
    def __init__(
        self,
        host: str = DEFAULT_HOST,
        port: int = DEFAULT_PORT,
        timeout: float = 120,
        username: str = "admin",
        password: str = "admin123",
        use_tls: bool = False,
        cafile: Path | None = None,
    ):
        self.host = host
        self.port = port
        self.timeout = timeout
        self.username = username
        self.password = password
        self.use_tls = use_tls
        self.cafile = cafile or DEFAULT_CAFILE
        self.sock: socket.socket | None = None
        self.token: str | None = None
        self.user: dict[str, Any] | None = None

    def connect(self) -> None:
        if self.sock:
            return
        raw_sock = socket.create_connection((self.host,
self.port), timeout=self.timeout)
        if self.use_tls:
            context =
ssl.create_default_context(cafile=str(self.cafile))
            self.sock = context.wrap_socket(raw_sock,
server_hostname=self.host)
        else:
            self.sock = raw_sock
        self.sock.settimeout(self.timeout)
        print(f"Connected to {self.host}:{self.port}
TLS={self.use_tls}")

    def close(self) -> None:
        if not self.sock:
            return
        try:
            send_json(self.sock, {"command": "QUIT"})
            response = recv_json(self.sock)
            print(response.get("message", "Session
closed"))
        except OSError:
            pass
        finally:
            self.sock.close()
            self.sock = None
            self.token = None
            self.user = None

    def _require_socket(self) -> socket.socket:
        if not self.sock:
            self.connect()
        assert self.sock is not None
        return self.sock

    def login(self, username: str | None = None, password:
str | None = None) -> dict[str, Any]:
        sock = self._require_socket()
        username = username or self.username
        password = password or self.password
        send_json(sock, {"command": "LOGIN", "username":
username, "password": password})
        response = recv_json(sock)
        if response.get("status") == "OK":
            self.token = response["token"]
            self.user = response["user"]
            self.username = username
            self.password = password
            print(response)
            return response

    def _ensure_login(self) -> str:
        if not self.token:
            response = self.login()
            if response.get("status") != "OK":
                raise
PermissionError(response.get("message", "Login failed"))
            assert self.token is not None

        return self.token

    def ping(self) -> dict[str, Any]:
        sock = self._require_socket()
        send_json(sock, {"command": "PING"})
        response = recv_json(sock)
        print(response)
        return response

    def list_files(self) -> dict[str, Any]:
        sock = self._require_socket()
        send_json(sock, {"command": "LIST", "token":
self._ensure_login()})
        response = recv_json(sock)
        if response.get("status") == "OK":
            files = response.get("files", [])
            if not files:
                print("No files on server")
            for file_record in files:
                print(
                    f"#{file_record['id']}
{file_record['original_name']} "
                    f"{file_record['size_bytes']} bytes
{file_record['sha256'][:12]}..."
                )
        else:
            print(response)
            return response

    def info(self, identifier: str) -> dict[str, Any]:
        sock = self._require_socket()
        send_json(sock, {"command": "INFO", "identifier":
identifier, "token": self._ensure_login()})
        response = recv_json(sock)
        print(response)
        return response

    def delete(self, identifier: str) -> dict[str, Any]:
        sock = self._require_socket()
        send_json(sock, {"command": "DELETE",
"identifier": identifier, "token": self._ensure_login()})
        response = recv_json(sock)
        print(response)
        return response

    def upload(
        self,
        file_path: str | Path,
        chunk_size: int = DEFAULT_CHUNK_SIZE,
        transfer_id: str | None = None,
    ) -> dict[str, Any]:
        path = Path(file_path)
        if not path.is_file():
            raise FileNotFoundError(f"File not found:
{path}")

        sock = self._require_socket()
        token = self._ensure_login()
        size = path.stat().st_size
        digest = sha256_file(path)
        print(f"Upload started: {path.name}, {size}
bytes")

        if transfer_id:
            send_json(
                sock,
                {
                    "command": "UPLOAD_RESUME",
                    "token": token,
                    "transfer_id": transfer_id,
                },
            )
        else:
            send_json(
                sock,
                {
                    "command": "UPLOAD_INIT",
                    "token": token,
                    "filename": path.name,
                    "size": size,
                    "sha256": digest,
                    "chunk_size": chunk_size,
                },
            )
        response = recv_json(sock)
        if response.get("status") != "READY":

```

```

        print(response)
        return response

    transfer_id = response["transfer_id"]
    offset = int(response.get("offset", 0))
    effective_chunk_size =
int(response.get("chunk_size", chunk_size))
    sent = offset
    chunk_index = offset // effective_chunk_size
    last_report = time.monotonic()
    with path.open("rb") as file_obj:
        file_obj.seek(offset)
        while True:
            chunk =
file_obj.read(effective_chunk_size)
            if not chunk:
                break
            self._send_chunk_with_ack(
                sock=sock,
                token=token,
                transfer_id=transfer_id,
                chunk_index=chunk_index,
                offset=sent,
                chunk=chunk,
            )
            sent += len(chunk)
            chunk_index += 1
            now = time.monotonic()
            if now - last_report >= 0.5 or sent ==
size:
                print(f"Sent {sent}/{size} bytes")
                last_report = now

    send_json(
        sock,
        {
            "command": "UPLOAD_FINISH",
            "token": token,
            "transfer_id": transfer_id,
        },
    )
    final_response = recv_json(sock)
    print(final_response)
    return final_response

def _send_chunk_with_ack(
    self,
    sock: socket.socket,
    token: str,
    transfer_id: str,
    chunk_index: int,
    offset: int,
    chunk: bytes,
    max_retries: int = 3,
) -> None:
    chunk_sha = hashlib.sha256(chunk).hexdigest()
    for attempt in range(max_retries):
        send_json(
            sock,
            {
                "command": "CHUNK",
                "token": token,
                "transfer_id": transfer_id,
                "chunk_index": chunk_index,
                "offset": offset,
                "size": len(chunk),
                "sha256": chunk_sha,
            },
        )
        sock.sendall(chunk)
        response = recv_json(sock)
        if response.get("status") == "ACK":
            return
        print(response)
        if response.get("status") != "NACK":
            raise RuntimeError(response.get("message",
"Unexpected upload response"))
        raise RuntimeError(f"Chunk {chunk_index} was
rejected after {max_retries} attempts")

    def download(self, identifier: str, downloads_dir:
Path = DOWNLOADS_DIR -> dict[str, Any]:
        sock = self._require_socket()
        send_json(sock, {"command": "DOWNLOAD",
"identifier": identifier, "token": self._ensure_login()})
        response = recv_json(sock)

        if response.get("status") != "OK":
            print(response)
            return response

        file_record = response["file"]
        size = int(file_record["size_bytes"])
        target = unique_download_path(downloads_dir,
file_record["original_name"])
        received = 0
        print(f"Download started:
{file_record['original_name']}, {size} bytes")

        with target.open("wb") as file_obj:
            while received < size:
                chunk = recv_exact(sock,
min(DEFAULT_CHUNK_SIZE, size - received))
                file_obj.write(chunk)
                received += len(chunk)
                print(f"Received {received}/{size} bytes")

        actual_sha = sha256_file(target)
        if actual_sha != file_record["sha256"]:
            target.unlink(missing_ok=True)
            result = {"status": "ERROR", "message":
"Checksum mismatch after download"}
            print(result)
            return result

        result = {"status": "OK", "path": str(target),
"sha256": actual_sha}
        print(result)
        return result

    def raw_command(self, command: str) -> dict[str, Any]:
        sock = self._require_socket()
        send_json(sock, {"command": command})
        response = recv_json(sock)
        print(response)
        return response

    def execute_command(client: FileExchangeClient, args:
list[str]) -> bool:
        if not args:
            return True
        command = args[0].lower()
        if command == "ping":
            client.ping()
        elif command == "login" and len(args) == 3:
            client.login(args[1], args[2])
        elif command == "list":
            client.list_files()
        elif command == "upload" and len(args) == 2:
            client.upload(args[1])
        elif command == "resume" and len(args) == 3:
            client.upload(args[2], transfer_id=args[1])
        elif command == "download" and len(args) == 2:
            client.download(args[1])
        elif command == "info" and len(args) == 2:
            client.info(args[1])
        elif command == "delete" and len(args) == 2:
            client.delete(args[1])
        elif command == "raw" and len(args) == 2:
            client.raw_command(args[1])
        elif command in {"quit", "exit"}:
            return False
        else:
            print("Commands: ping, login <username>
<password>, list, upload <path>, resume <transfer_id>
<path>, download <id|name>, info <id|name>, delete
<id|name>, quit")
            return True

    def interactive_shell(client: FileExchangeClient) -> None:
        print("Interactive client. Type: ping, login
<username> <password>, list, upload <path>, resume
<transfer_id> <path>, download <id|name>, info <id|name>,
delete <id|name>, quit")
        while True:
            try:
                line = input("file-client> ").strip()
            except EOFError:
                break
            if not execute_command(client, shlex.split(line)):
                break

```

```

def parse_args(argv: list[str]) -> argparse.Namespace:
    parser = argparse.ArgumentParser(description="TCP file
exchange client")
    parser.add_argument("--host", default=DEFAULT_HOST)
    parser.add_argument("--port", type=int,
default=DEFAULT_PORT)
    parser.add_argument("--username", default="admin")
    parser.add_argument("--password", default="admin123")
    parser.add_argument("--tls", action="store_true")
    parser.add_argument("--cafile", type=Path,
default=DEFAULT_CAFILE)
    parser.add_argument("command", nargs="*")
    return parser.parse_args(argv)

def main(argv: list[str] | None = None) -> int:
    args = parse_args(argv or sys.argv[1:])
    client = FileExchangeClient(
        args.host,
        args.port,
        username=args.username,
        password=args.password,
        use_tls=args.tls,
        cafile=args.cafile,
    )
    try:
        client.connect()
        if args.command:
            execute_command(client, args.command)
        else:
            interactive_shell(client)
    except (OSError, FileNotFoundError, KeyboardInterrupt)
as exc:
    print(f"Error: {exc}")
    return 1
finally:
    client.close()
    return 0

if __name__ == "__main__":
    raise SystemExit(main())

```

Лістинг Г.10 – admin/app.py

```

from pathlib import Path

from fastapi import FastAPI
from fastapi.responses import PlainTextResponse

from server.config import LOG_FILE
from server.metadata_db import MetadataDB

app = FastAPI(title="File Exchange Admin API")

@app.get("/health")
def health() -> dict[str, str]:
    return {"status": "OK"}

@app.get("/files")
def files() -> dict[str, object]:
    db = MetadataDB()
    try:
        return {"status": "OK", "files":
db.list_files(include_deleted=True)}
    finally:
        db.close()

@app.get("/files/{identifier}")
def file_info(identifier: str) -> dict[str, object]:
    db = MetadataDB()
    try:
        record = db.find_file(identifier,
include_deleted=True)
        if not record:
            return {"status": "ERROR", "message": "File
not found"}
        return {"status": "OK", "file": record}
    finally:

```

```

db.close()

@app.get("/logs", response_class=PlainTextResponse)
def logs(lines: int = 100) -> str:
    path = Path(LOG_FILE)
    if not path.exists():
        return ""
    content = path.read_text(encoding="utf-8",
errors="replace").splitlines()
    return "\n".join(content[-max(1, min(lines, 100)):])

```

Лістинг Г.11 – tests/conftest.py

```

import socket
import sys
import threading
import time
from pathlib import Path

import pytest

ROOT_DIR = Path(__file__).resolve().parents[1]
if str(ROOT_DIR) not in sys.path:
    sys.path.insert(0, str(ROOT_DIR))

from client.client import FileExchangeClient
from server.server import FileExchangeServer

def free_port() -> int:
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM)
as sock:
        sock.bind(("127.0.0.1", 0))
        return int(sock.getsockname()[1])

@pytest.fixture(scope="session")
def server_instance():
    port = free_port()
    server = FileExchangeServer(host="127.0.0.1",
port=port, max_workers=10)
    thread = threading.Thread(target=server.start,
name="pytest-file-server", daemon=True)
    thread.start()

    deadline = time.monotonic() + 5
    while time.monotonic() < deadline:
        try:
            client = FileExchangeClient("127.0.0.1", port,
timeout=2)
            client.connect()
            client.ping()
            client.close()
            break
        except OSError:
            time.sleep(0.1)
    else:
        server.stop()
        raise RuntimeError("Server did not start in time")

    yield {"host": "127.0.0.1", "port": port}

    server.stop()
    thread.join(timeout=3)

@pytest.fixture(scope="session")
def tls_server_instance():
    port = free_port()
    server = FileExchangeServer(host="127.0.0.1",
port=port, max_workers=10, use_tls=True)
    thread = threading.Thread(target=server.start,
name="pytest-tls-file-server", daemon=True)
    thread.start()

    deadline = time.monotonic() + 5
    while time.monotonic() < deadline:
        try:
            client = FileExchangeClient(
"127.0.0.1",
port,
timeout=2,
use_tls=True,
cafile=ROOT_DIR / "certs" / "server.crt",

```

```

    )
    client.connect()
    client.ping()
    client.close()
    break
except OSError:
    time.sleep(0.1)
else:
    server.stop()
    raise RuntimeError("TLS server did not start in
time")

yield {"host": "127.0.0.1", "port": port}

server.stop()
thread.join(timeout=3)

@pytest.fixture()
def client(server_instance):
    cli = FileExchangeClient(server_instance["host"],
server_instance["port"], timeout=10)
    cli.connect()
    try:
        yield cli
    finally:
        cli.close()

```

Лістинг Г.12 – tests/test_protocol.py

```

import socket

import pytest

from server.protocol import ProtocolError, recv_json,
send_json

def test_json_framing_roundtrip():
    left, right = socket.socketpair()
    try:
        send_json(left, {"command": "PING", "value": 123})
        assert recv_json(right) == {"command": "PING",
"value": 123}
    finally:
        left.close()
        right.close()

def test_invalid_header_length():
    left, right = socket.socketpair()
    try:
        left.sendall((0).to_bytes(4, "big"))
        with pytest.raises(ProtocolError):
            recv_json(right)
    finally:
        left.close()
        right.close()

```

Лістинг Г.13 – tests/test_hash.py

```

from client.utils import sha256_file

def test_sha256_file(tmp_path):
    path = tmp_path / "sample.txt"
    path.write_text("hello file exchange\n",
encoding="utf-8")

    assert sha256_file(path) ==
"486fcc03927cb3ca2a3f57288313b056d48902b0a363a31d55734165d
47c26cf"

```

Лістинг Г.14 – tests/test_upload_download.py

```

from pathlib import Path
from uuid import uuid4

from client.utils import sha256_file

```

```

def test_ping(client):
    response = client.ping()
    assert response["status"] == "OK"
    assert response["message"] == "PONG"

def test_upload_info_download_and_delete(client,
tmp_path):
    source = tmp_path / f"demo_{uuid4().hex}.txt"
    source.write_text("Bachelor file exchange
prototype\n", encoding="utf-8")
    expected_hash = sha256_file(source)

    upload_response = client.upload(source)
    assert upload_response["status"] == "OK"
    file_id = str(upload_response["file"]["id"])

    info_response = client.info(file_id)
    assert info_response["status"] == "OK"
    assert info_response["file"]["sha256"] ==
expected_hash

    download_response = client.download(file_id,
downloads_dir=tmp_path / "downloads")
    assert download_response["status"] == "OK"
    assert Path(download_response["path"]).is_file()
    assert download_response["sha256"] == expected_hash

    delete_response = client.delete(file_id)
    assert delete_response["status"] == "OK"

def test_unknown_command(client):
    response = client.raw_command("UNKNOWN_TEST_COMMAND")
    assert response["status"] == "ERROR"

def test_download_missing_file(client):
    response = client.download("9999999999")
    assert response["status"] == "ERROR"

```

Лістинг Г.15 – tests/test_security_and_resume.py

```

import hashlib
import time
from pathlib import Path
from uuid import uuid4

from client.client import FileExchangeClient
from client.protocol import recv_json, send_json
from client.utils import sha256_file

def test_login_success_and_failed(client):
    ok = client.login("admin", "admin123")
    assert ok["status"] == "OK"
    assert ok["user"]["role"] == "admin"

    failed_client = FileExchangeClient(client.host,
client.port, timeout=10)
    failed_client.connect()
    try:
        failed = failed_client.login("admin", "wrong-
password")
        assert failed["status"] == "ERROR"
    finally:
        failed_client.close()

def test_delete_requires_auth(server_instance):
    cli = FileExchangeClient(server_instance["host"],
server_instance["port"], timeout=10)
    cli.connect()
    try:
        sock = cli._require_socket()
        send_json(sock, {"command": "DELETE",
"identifier": "1"})
        response = recv_json(sock)
        assert response["status"] == "ERROR"
        assert "Authentication" in response["message"]
    finally:
        cli.close()

```

```

def
test_user_cannot_download_foreign_file_and_admin_can(serve
r_instance, tmp_path):
    source = tmp_path / f"owned_{uuid4().hex}.txt"
    source.write_text("owned by first user\n",
encoding="utf-8")

    owner = FileExchangeClient(
        server_instance["host"], server_instance["port"],
username="user", password="user123"
    )
    stranger = FileExchangeClient(
        server_instance["host"], server_instance["port"],
username="user2", password="user123"
    )
    admin = FileExchangeClient(
        server_instance["host"], server_instance["port"],
username="admin", password="admin123"
    )

    for cli in (owner, stranger, admin):
        cli.connect()
    try:
        upload_response = owner.upload(source)
        assert upload_response["status"] == "OK"
        file_id = str(upload_response["file"]["id"])

        denied = stranger.download(file_id,
downloads_dir=tmp_path / "stranger")
        assert denied["status"] == "ERROR"

        allowed = admin.download(file_id,
downloads_dir=tmp_path / "admin")
        assert allowed["status"] == "OK"
    finally:
        for cli in (owner, stranger, admin):
            cli.close()

def test_chunk_nack_on_wrong_hash(client, tmp_path):
    source = tmp_path / "bad_chunk.txt"
    payload = b"this block will have a wrong declared
checksum"
    source.write_bytes(payload)

    client.login("admin", "admin123")
    sock = client._require_socket()
    send_json(
        sock,
        {
            "command": "UPLOAD_INIT",
            "token": client.token,
            "filename": source.name,
            "size": len(payload),
            "sha256": sha256_file(source),
            "chunk_size": 64,
        },
    )
    ready = recv_json(sock)
    assert ready["status"] == "READY"

    send_json(
        sock,
        {
            "command": "CHUNK",
            "token": client.token,
            "transfer_id": ready["transfer_id"],
            "chunk_index": 0,
            "offset": 0,
            "size": len(payload),
            "sha256": "0" * 64,
        },
    )
    sock.sendall(payload)
    response = recv_json(sock)
    assert response["status"] == "NACK"

def test_upload_resume(server_instance, tmp_path):
    source = tmp_path / "resume.txt"
    source.write_bytes(b"A" * 1024 + b"B" * 1024)
    digest = sha256_file(source)

    first = FileExchangeClient(server_instance["host"],
server_instance["port"], timeout=10)

```

```

first.connect()
first.login("admin", "admin123")
sock = first._require_socket()
send_json(
    sock,
    {
        "command": "UPLOAD_INIT",
        "token": first.token,
        "filename": source.name,
        "size": source.stat().st_size,
        "sha256": digest,
        "chunk_size": 1024,
    },
)
ready = recv_json(sock)
assert ready["status"] == "READY"
transfer_id = ready["transfer_id"]
first_chunk = source.read_bytes()[1024]
send_json(
    sock,
    {
        "command": "CHUNK",
        "token": first.token,
        "transfer_id": transfer_id,
        "chunk_index": 0,
        "offset": 0,
        "size": len(first_chunk),
        "sha256":
hashlib.sha256(first_chunk).hexdigest(),
    },
)
sock.sendall(first_chunk)
ack = recv_json(sock)
assert ack["status"] == "ACK"
sock.close()
first.sock = None
time.sleep(0.1)

second = FileExchangeClient(server_instance["host"],
server_instance["port"], timeout=10)
second.connect()
try:
    response = second.upload(source, chunk_size=1024,
transfer_id=transfer_id)
    assert response["status"] == "OK"
    assert response["file"]["sha256"] == digest
finally:
    second.close()

def test_tls_connection(tls_server_instance):
    cli = FileExchangeClient(
        tls_server_instance["host"],
        tls_server_instance["port"],
        use_tls=True,
        cafile=Path(__file__).resolve().parents[1] /
"certs" / "server.crt",
    )
    cli.connect()
    try:
        response = cli.ping()
        assert response["status"] == "OK"
    finally:
        cli.close()

```

Лістинг Г.16 – tests/stress_test.py

```

from __future__ import annotations

import argparse
import concurrent.futures
import os
import sys
import tempfile
import time
from pathlib import Path

ROOT_DIR = Path(__file__).resolve().parents[1]
if str(ROOT_DIR) not in sys.path:
    sys.path.insert(0, str(ROOT_DIR))

from client.client import FileExchangeClient

def create_payload(path: Path, size_mb: int) -> None:

```

```

remaining = size_mb * 1024 * 1024
chunk = 1024 * 1024
with path.open("wb") as file_obj:
    while remaining > 0:
        block_size = min(chunk, remaining)
        file_obj.write(os.urandom(block_size))
        remaining -= block_size

def upload_once(
    host: str,
    port: int,
    file_path: Path,
    username: str,
    password: str,
    use_tls: bool,
    cafile: Path,
) -> tuple[bool, float, str]:
    started = time.perf_counter()
    client = FileExchangeClient(
        host,
        port,
        timeout=300,
        username=username,
        password=password,
        use_tls=use_tls,
        cafile=cafile,
    )
    try:
        client.connect()
        response = client.upload(file_path)
        elapsed = time.perf_counter() - started
        return response.get("status") == "OK", elapsed,
    str(response)
    except Exception as exc:
        return False, time.perf_counter() - started,
    str(exc)
    finally:
        client.close()

def main() -> int:
    parser = argparse.ArgumentParser(description="Parallel
upload stress test")
    parser.add_argument("--host", default="127.0.0.1")
    parser.add_argument("--port", type=int, default=9000)
    parser.add_argument("--clients", type=int, default=5)
    parser.add_argument("--size-mb", type=int, default=10)
    parser.add_argument("--username", default="admin")
    parser.add_argument("--password", default="admin123")
    parser.add_argument("--tls", action="store_true")
    parser.add_argument("--cafile", type=Path,
default=ROOT_DIR / "certs" / "server.crt")
    args = parser.parse_args()

    with tempfile.TemporaryDirectory() as tmp_dir:
        file_path = Path(tmp_dir) /
f"stress_{args.size_mb}mb.bin"
        create_payload(file_path, args.size_mb)

        started = time.perf_counter()
        with
concurrent.futures.ThreadPoolExecutor(max_workers=args.cli
ents) as executor:
            futures = [
                executor.submit(
                    upload_once,
                    args.host,
                    args.port,
                    file_path,
                    args.username,
                    args.password,
                    args.tls,
                    args.cafile,
                )
                for _ in range(args.clients)
            ]
            results = [future.result() for future in
futures]
            total_time = time.perf_counter() - started

            success_count = sum(1 for ok, _, _ in results if ok)
            error_count = args.clients - success_count
            average_client_time = sum(elapsed for _, elapsed, _ in
results) / len(results)
            total_mb = args.clients * args.size_mb

            speed = total_mb / total_time if total_time else 0

            print("\nStress test report")
            print("=====")
            print(f"Clients: {args.clients}")
            print(f"File size per client: {args.size_mb} MB")
            print(f"Total transfer time: {total_time:.2f}
s")
            print(f"Average time per client:
{average_client_time:.2f} s")
            print(f"Successful operations: {success_count}")
            print(f"Errors: {error_count}")
            print(f"Average transfer speed: {speed:.2f} MB/s")

            if error_count:
                print("\nErrors:")

```