

Детектор Плагиата v. 2965 - Отчёт оригинальности: 17.06.2026 12:53:42

Проанализированный документ: Диплом Яценко (1).docx
Лицензия: ВОЛОДИМИР МАТІЄВСЬКИЙ

Тип поиска: Поиск переписанного
Язык: Uk
Тип проверки: Интернет
ТЭЕ и кодировка: DocX n/a

Детальный анализ тела документа:

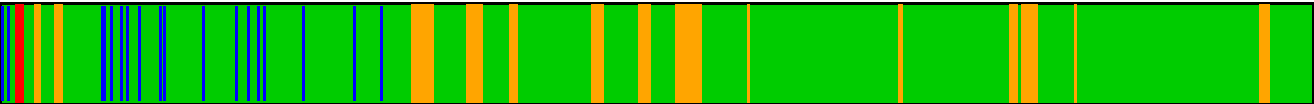
Диаграмма соотношения частей:



Подробное сходство:



Граф распределения зон:



Источники плагиата: 1

0,8% 742 1. <https://ela.kpi.ua/bitstreams/12e4afc5-8602-4df7-b21e-c566ec42d2fc/download>

Детали обработанных ресурсов: 244 - ОК / 16 - Ошибка

Важные замечания:

Википедия:	Google Книги:	Сервисы платных работ:	Античит:
			
[не обнаружено]	[не обнаружено]	[не обнаружено]	[не обнаружено]

 Античит-отчет UACE:

1. Статус: Анализатор Включен Нормализатор Включен сходство символов установлено на 100%
2. Обнаруженный процент загрязнения UniCode: 3,1% с лимитом: 4%
3. Документ не нормализован: процент не достигнут 5%
4. Все подозрительные символы будут отмечены фиолетовым цветом: Abcd...
5. Найдены невидимые символы: 0
Рекомендации по оценке: Никаких особых действий не требуется. Документ в порядке.
Алфавитная статистика и анализ символов:

 Активные ссылки (URL-адреса, извлеченные из документа):

URL не найдены

 Исключённые ресурсы:

URL не найдены

 Включённые ресурсы:

URL не найдены

Детальный анализ документа:

Міністерство освіти і науки України ДЗ

Цитування: 0,01%

id: 1

«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

Цитування: 0%

id: 2

«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 ВСТУПЗ РОЗДІЛ 1. АНАЛІТИЧНИЙ ОГЛЯД ПРЕДМЕТНОЇ
ОБЛАСТІ6 1.1 Клієнт-серверна архітектура: засади побудови та варіанти реалізації6 1.2
Типові моделі взаємодії клієнта і сервера в системах передачі даних7 1.3 Мережеві
протоколи передавання файлів: функціональні можливості та обмеження11 1.4
Багатопоточність і паралельна обробка: базові поняття та підходи13 1.5 Огляд наявних
рішень для обміну файлами: порівняльна характеристика16

Обнаружен Плагиат: 0,1%

id: 3

Висновки до розділу 118 РОЗДІЛ 2. ПРОЄКТУВАННЯ СИСТЕМИ19 2.1. Загальна архітектура
клієнт-серверної системи19 2.2. Проєктування структури взаємодії компонентів22 2.3.
Застосування протоколів обміну даними27 2.4. Проєктування механізму багатопотокової
обробки запитів32 2.5. Проєктування структури збереження файлів та журналювання36
2.6. Моделювання системи41 Висновки до розділу 243 РОЗДІЛ 3. ПРОГРАМНА
РЕАЛІЗАЦІЯ44 3.1. Обґрунтування вибору мови Python та використаних бібліотек44 3.2.
Реалізація серверної частини46 3.3. Реалізація клієнтської частини49 3.4. Реалізація
механізму багатопоточності52 3.5. Забезпечення надійності та безпеки передачі даних54
3.6. Тестування та аналіз продуктивності системи57 Висновки до розділу 360 ВИСНОВКИ
61 СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ64 ДОДАТОК А68 Порівняльна характеристика підходів
до конкурентної обробки в клієнт-серверних системах передавання файлів68 ДОДАТОК
Б70 Результати тестування клієнт-серверної системи обміну файлами70 ДОДАТОК В72
ДОДАТОК Г87 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.

AI generated text detected: ChatGPT 5.3, : 52,08%

id: 4

У практичних сценаріях передавання відбувається паралельно для багатьох користувачів
і процесів, а мережеві з'єднання характеризуються змінною пропускну 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 пропускну здатності, затримки відповіді, стабільності під навантаженням і споживання ресурсів [32].

 AI generated text detected: ChatGPT 5.3, : 61,69%

id: 5

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 мережевих застосунків. На етапі аналітичного опрацювання застосовуються порівняльний аналіз архітектур і протоколів передавання даних, узагальнення технічних вимог та формалізація сценаріїв використання.

Проектування 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 мережевих 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 пропускну 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

 Цитування: 0%

id: 6

«тонкого»

та

 Цитування: 0%

id: 7

«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

Цитування: 0%

id: 8

«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

Цитування: 0%

id: 9

«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

Цитування: 0%

id: 10

«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, події повідомлення або асинхронні підтвердження [13]. Найпоширенішою є синхронна модель

Цитування: 0%

id: 11

«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. Поточковий 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 файлу 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 події моделі:

Цитування: 0% id: 12
«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

Цитування: 0% id: 13
«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 тайм-аутів; 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; оперативні повідомлення про стан	Додаткова інфраструктура або складніша реалізація; необхідність гарантій доставки подій	Сповіщення про завершення/помилки; індикація прогресу; 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

Цитування: 0% id: 14
«передати/отримати»

і потребують часу. Таке поєднання дозволяє розділити

Цитування: 0% id: 15
«керування»

і
Цитування: 0% id: 16
«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 міжмережеві 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

Цитування: 0%

id: 17

«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

Цитування: 0%

id: 18

«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 мережевих 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

Цитування: 0%

id: 19

«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

Цитування: 0%

id: 20

«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, мережевих сокетів 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, багатопроектністю, 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 однопоточних системах за рахунок швидкого перемикавання контексту, тоді як на багатопоточних 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 мережевих серверах 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 мережевих 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

Цитування: 0%

id: 21

«потоку 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 перемикавання 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 перемикавання 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: мережевий 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 - навпаки [32]. Систематизація підходів до конкурентної обробки в клієнт-серверних системах передавання файлів дає змогу зіставити потокову модель, пул потоків, багатопроектність, 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 мережесистем файлових систем ([SMB/CIFS](#), [NFS](#)) реалізують модель

Цитування: 0%

id: 22

«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 проєктованої 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

Цитування: 0%

id: 23

«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].

 AI generated text detected: ChatGPT 5.3, : 61,28%

id: 24

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. Проектування структури взаємодії компонентів

Проектування 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.

 AI generated text detected: ChatGPT 5.3, : 55,55%

id: 25

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 завершення 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.

 AI generated text detected: ChatGPT 5.3, : 54,24%

id: 26

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](#)-потоці 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 файла із сервера

Особливе значення для системи має підтримка перерваних передач. У мережевому середовищі з нестабільним з'єднанням або при великому обсязі файла 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
LIST	Службовий статус сервера	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
STATUS	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
REASON	Пояснення		

причини відмови або збою Застосування протоколів обміну даними в проєктованій системі базується на поєднанні надійного транспортного рівня [TCP](#) 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.

AI AI generated text detected: ChatGPT 5.3, : 59,81%

id: 27

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 [TCP](#)-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 багатопотоковості 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 Проєктована модель багатопотокової обробки орієнтована передусім на [I/O](#)-залежний характер навантаження. У системі обміну файлами значну частину часу займає не саме обчислення, а очікування мережевих пакетів, доступу до диска, запису блоків, читання метаданих і завершення операцій журналювання [32]. Поряд із перевагами багатопотоковості 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].

AI AI generated text detected: ChatGPT 5.3, : 52,11%

id: 28

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.

 AI generated text detected: ChatGPT 5.3, : 60,35%

id: 29

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 файла. 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 файлу в межах системи **file_name** Початкова назва файлу, отримана від клієнта **str** Відображення файлу в клієнтському й адміністративному інтерфейсі **file_path** Фактичний шлях збереження файлу на сервері **str** Локалізація файлу у файловому сховищі **temp_path** Шлях до тимчасового файлу під час незавершеної передачі **str** Підтримка проміжного стану та відновлення передачі **size** Повний розмір файлу в байтах **int** / **long** Контроль повноти передавання та перевірка відповідності обсягу **checksum** Контрольна сума файлу **str** Перевірка цілісності даних після завершення передачі **status** Поточний стан файлу 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 файлу 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 файлу, так і службового стану його передавання. Поєднання ідентифікаторів файлу, передачі та сесії створює основу для відстеження повного життєвого циклу операції, тоді як атрибути **offset**, [13]. Отже, проєктування 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 журналюванню 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.

 AI generated text detected: ChatGPT 5.3, : 54,25%

id: 30

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 файлу. 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 файлу, 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 файлу 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 файлу на сервер і отримання файлу 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 сформулювати 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]. Мережовий 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](#).

AI generated text detected: ChatGPT 5.3, : 51,32%

id: 31

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 файла із сервера реалізовано через команду [DOWNLOAD](#). Після отримання запиту сервер перевіряє сесійний токен, права доступу користувача до відповідного файла, 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 файлу 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.

AI generated text detected: ChatGPT 5.3, : 53,72%

id: 32

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.

AI generated text detected: ChatGPT 5.3, : 56,92%

id: 33

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 виникла 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.

 AI generated text detected: ChatGPT 5.3, : 50,13%

id: 34

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 обгортати [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](#)-потоці. 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 файла, 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, впровадження механізмів багатопоточності, контролю цілісності, автентифікації, рольового доступу й тестової перевірки працездатності програмного прототипу [37].

 AI generated text detected: ChatGPT 5.3, : 52,99%

id: 35

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 сформуванню цілісної архітектури, у якій клієнтський застосунок, файловий сервер, адміністративний сервіс, файлова система, база метаданих і підсистема журналювання мають чітко визначені функції.

Серверна частина спроектована 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](#)-сокети, 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].

Заявление об ограничении ответственности:

Этот отчет должен быть правильно истолкован и проанализирован квалифицированным специалистом, который несет ответственность за оценку!

Любая информация, представленная в этом отчете, не является окончательной и подлежит ручному просмотру и анализу. Пожалуйста, следуйте инструкциям: [Рекомендации по оценке](#)

Детектор Плагиата - Ваше право на оригинальность! ☐ SkyLine LLC