Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Комп'ютерної інженерії та управління
(повна назва)

Кафедра Автоматизації проектування обчислювальної техніки
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти перший (бакалаврський)

Використання високопродуктивної HTTP бібліотеки Python у системах IoT

(тема)

Виконав:

Здобувач 4курсу, групи КІУКІ-21-9
Селюков В.Г.
(прізвище,ініціали)

спеціальності 123–Комп'ютерна інженерія
(шифр і назва спеціальності)

Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма Комп'ютерна інженерія
(повна назва освітньої програми)

Керівник ас. Кулак Г.К.
(посада,прізвище,ініціали)

Допускається до захисту

Зав. кафедри ____________ Чумаченко С.В.
(підпис) (прізвище,ініціали)

2025р.

Харківський національний університет радіоелектроніки

Факультет Комп’ютерної інженерії та управління

Кафедра Автоматизації проектування обчислювальної техніки

Рівень вищої освіти перший (бакалаврський)

Спеціальність 123 Комп’ютерна інженерія

Тип програми Освітньо-професійна

Освітня програма Комп’ютерна інженерія

ЗАТВЕРДЖУЮ:
Зав. кафедри

____
(підпис)
«06» 05 2025р.

ЗАВДАННЯ
НАКВАЛІФІКАЦІЙНУРОБОТУ

здобувачеві Селюкову Вадіму Геннадійовичу
(прізвище,ім’я,по батькові)

1. Тема роботи Використання високопродуктивної HTTP бібліотеки Pythonу системах IoT

затверджена наказом університету від 21 05 2025р. №_403Ст

2. Термін подання здобувачем роботи до екзаменаційної комісії_16_ 06 2025р.

3. Вихідні дані до роботи

Протоколи HTTP/3 та QUIC,формат серіалізації MessagePack, віртуальні сервери DigitalOcean «Droplet», домен та DNS-сервіс від GoDaddy

Середовище розробки PyCharmIDE2024.1.3

Мова програмування Python 3.12.

4. Перелік питань, що потрібно опрацювати в роботі

Аналіз та огляд існуючих альтернатив.

Постановка задачі.

Розробка структурної схеми бібліотеки.

Розробка алгоритму роботи бібліотеки.

Аналіз розробленої системи

Тестування бібліотеки.

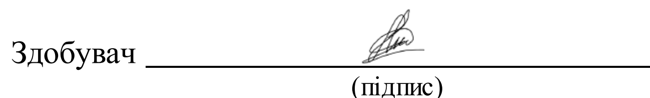
5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри)

14 слайдів

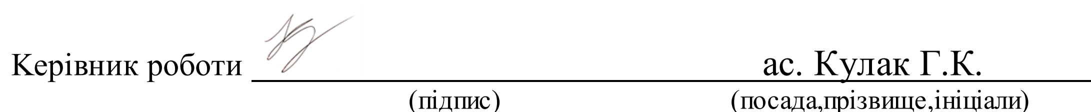
КАЛЕНДАРНИЙПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Видача теми проєкту,узгодження і затвердження теми.	06.05.2025 – 10.05.2025	
2	Аналіз предметної області,постановка задачі, Вибір стеку технологій.	10.05.2024 – 11.05.2024	
3	Розробка алгоритму, загальної логіки та схеми роботи бібліотеки.	11.05.2024 – 14.05.2024	
4	Розробка програмної частини бібліотеки.	14.05.2024 – 21.05.2024	
5	Тестування та аналіз розробленого продукту, у порівнянні з конкурентами.	21.05.2024 – 24.05.2024	
6	Оформлення пояснювальної записки.	24.05.2024 – 28.05.2024	
7	Перевірка проєкту науковим керівником, допуск до захисту роботи.	28.05.2024 – 16.06.2025	

Дата видачі завдання 06.05.2025р.



Здобувач ____________________
(підпис)



Керівник роботи ____________________ ас. Кулак Г.К.
(підпис) (посада,прізвище,ініціали)

РЕФЕРАТ

Пояснювальна записка з кваліфікаційної містить: 48 сторінок, 12 лістингів,7 джерел посилання, 5 рисунків, 1 таблицю, 1 додаток.

БІБЛІОТЕКА, ІНТЕРНЕТ, КЛІЄНТ, ПРИСТРІЙ, ПРОГРАМА, СЕРВЕР, СИСТЕМА, APP, HTTP/3, IOT, MESSAGEPACK, REST, RPC, PYTHON

Метою кваліфікаційної роботи є розроблення високопродуктивної та зручної у застосуванні HTTP бібліотеки мовою програмування Python, та її тестове використання та демонстрація у програмній системі IoT, як інструменту комунікації, як між програмними компонентами в межах серверної частини системи, так і між індивідуальними IoT-приладами та серверною частиною застосунку.

Для написання програмного коду бібліотеки та допоміжних застосунків використано мову програмування Python. У якості середовища розробки (IDE)використано спеціалізоване середовище PyCharm.

Було розроблено два допоміжних застосунки для демонстрації використання бібліотеки на практиці: серверна частина застосунку, та безпосередньо програмна емуляція IoT-пристрою. Реалізована комунікація між програмними компонентами, збереження інформації до бази даних та управління пристроєм з «сервера».

ABSTRACT

The explanatory note of the qualification workcontains: 48pages, 12 listings,7 sources, 5 figures, 1 table,1 appendix.

APP,CLIENT, DEVICE, HTTP/3, INTERNET, IOT, LIBRARY, MESSAGEPACK, PROGRAM, PYTHON, REST, RPC, SERVER, SYSTEM

The objective of the qualification work is the development of a high-performanceand easy-to-use HTTP library using the Python programming language,as well as its testing and a demonstration within an IoT software system, as the communication tool for both the software components within the server-side infrastructure and between individual IoT devices and the server-side system.

The python programming language was used to implement both the library and the auxilary applications. The development environment (IDE) used was PyCharm, specialized on Python.

Two auxilary applications were developed to demonstrate the practical use of the library: the server-side component of the IoT system and the software simulation of a IoT device itself. The implementation includes communication between software components, data storage in the database, and device controling logic from the server.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ 8
ВСТУП 10
1АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ 11
1.1Ознайомлення з популярними методамиінтернет-комунікації 11
1.2 Аналіз існуючих інструментів програмної інтернет-комунікаці 12
1.3 Комунікація вIoT системах 16
1.4Визначення ніші бібліотеки 17
1.5 Вибір мови програмування 18
1.6 Вибір технологій для реалізації проекту 18
1.7 Транспортний протокол QUIC 20
1.8Постановка завдання 20
2ПРОГРАМНА РЕАЛІЗАЦІЯ ПРОЕКТУ 22
2.1 Загальна логіка та алгоритм роботи 22
2.2 Базова RESTклієнт-сервер бібліотека 26
2.2.1REST клієнт 27
2.2.2REST сервер 29
2.3Фінальна RPC бібліотека 30
2.3.1 RPC клієнт 31
2.3.2 RPC сервер 34
2.4 Опис загальної функціональності бібліотеки 38
3ТЕСТУВАННЯ ТА ВИКОРИСТАННЯ БІБЛІОТЕКИ 42
3.1Використання бібліотеки у демонстраційних застосунках 42

3.2.1 Загальна логіка ... 42
3.2.2 IoT-пристрій ... 42
3.2.3 IoT-сервер ... 43
3.2 Порівняння швидкості роботи бібліотеки за альтернативами ... 44
ВИСНОВКИ ... 48
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ ... 49
ДОДАТОК А ГРАФІЧНА ЧАСТИНА ... 51
ДОДАТОК Б РЕПОЗИТОРІЇ ТА ПОСИЛАННЯ ... 58

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

API – Інтерфейс програмування застосунків (англ. application programming interface) – це спосіб взаємодії комп'ютерних програм між собою.

Connection pool– це кеш багаторазових підключень до бази даних або іншого комп’ютера, керований клієнтом або проміжним програмним забезпеченням. Він зменшує накладні витрати на відкриття та закриття підключень, покращуючи продуктивність та масштабованість у застосунках.

GraphQL – мова запитів і маніпуляції даними з відкритим кодом для API і середовище виконання для обслуговування запитів з наявних даних.

gRPC – (англ. Google Remote Procedure Calls) – це система віддаленого виклику процедур (RPC) з відкритим кодом.

HTTP – протокол передачі гіпертекстових документів (англ. Hypertext Transfer Protocol).

IoT –інтернет речей (англ. Internet of things).

JSON – запис об'єктів JavaScript (англ. JavaScriptObjectNotation) — це текстовий формат обміну даними між комп'ютерами. JSON базується на тексті, може бути прочитаним людиною.

MessagePack – компактний комп’ютерний формат обміну даними, призначений для двійкового представлення простих структур даних наподобі масивів і асоціативних масивів.

Protobuf – Protocol Buffers, формат серіалізації структурованих даних. Використовується у сфері розробки систем програмного забезпечення, компоненти якого взаємодіють один з одним через інтернет, або у цілях компактного зберігання даних.

QUIC – Швидкі UDP інтернет-з’єднання (англ. Quick UDP Internet Connections; вимовляється як quick) – транспортний мережевий протокол побудований на основі протоколу UDP. Є сучасною альтернативою зв'язці TCP

та TLS.

REST – передача репрезентативного стану (англ. Representational State Transfer). Найпопулярніший підхід до архітектури розподілених систем, який широко застосовується для проєктування веб-інтерфейсів, які надають доступ до інформаційних ресурсів.

RPC – виклик віддалених процедур (англ. Remote procedurecall). Є одним популярних із альтернатив REST архітектури для реалізації комунікації двох застосунків у замкнутій програмній системі.

TCP – Протокол управління передачею(англ. Transmission Control Protocol) – разом із протоколом IP є стрижневим протоколом Інтернету, який дав назву моделі TCP/IP, призначений для керування передаванням даних у комп'ютерних мережах.

TLS – захист на транспортному рівні(англ. TransportLayerSecurity), як і його попередник SSL – криптографічний протокол, що надає можливості безпечної передачі даних в інтернеті для навігації, отримання пошти, спілкування, обміну файлами, тощо.

UDP – Протокол датаграм користувача (англ. User Datagram Protocol) – один із протоколів в стеку TCP/IP. Від протоколу TCP він відрізняється тим, що працює без встановлення з'єднання.

Ендпоінт – це визначена точка доступу до функціональності серверного застосунку, яка обробляє запити за певною URL-адресою та HTTP-методом.

Мікросервіси – архітектурний стиль, за яким єдиний застосунок будується як сукупність невеличких сервісів, кожен з яких працює у своєму власному процесі та спілкується з рештою, використовуючи прості в реалізації та швидкі у виконанні методи передачі даних, зазвичай HTTP.

ВСТУП

У сучасних умовах стрімкого розвитку технологій Інтернету речей (IoT) зростаєпотреба в ефективних засобах обміну даними між пристроями, сервісамитакористувачами. Автоматизація збирання, передачі та обробки інформації з великої кількості розподілених пристроїв відкриває нові можливості в різних сферах: від побутових систем до промислових комплексів. У цьому контексті ключову роль відіграють мережеві протоколи та програмні бібліотеки, які забезпечують стабільну, швидку й безпечну взаємодію між компонентами IoT-систем.

Особливої актуальності набуває розробка власних високопродуктивних рішень, що можуть забезпечити зручність інтеграції, простоту використання та гнучкість конфігурації. У межах цієї дипломної роботи розроблено спеціалізовану HTTP-бібліотеку мовою програмування Python, яка реалізує концепцію віддаленого виклику процедур (RPC) поверх сучасного транспортного протоколу HTTP/3. На відміну від громіздких рішень, що вимагають складної конфігурації та попередньої генерації коду, запропонована бібліотека орієнтована на розробників, яким потрібен простий інтерфейс для швидкого створення клієнт-серверних IoT-застосунків.

Розроблене рішення дозволяє організовувати ефективну взаємодію між програмними компонентами та фізичними пристроями, зокрема сенсорами, мікроконтролерами та шлюзами. Система може бути використана як у домашніх автоматизованих проєктах, так і в масштабованих розподілених рішеннях. Особливу цінність бібліотека може становити для тих сценаріїв, де необхідна як висока ефективністьу передаванні даних, так і мінімальні зусилля на інтеграцію в систему.

1АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Ознайомлення з популярними методами інтернет-комунікації

У сучасних розподілених інформаційних системах важливу роль відіграють протоколи та архітектурні стилі, які визначають способи обміну даними між клієнтами й серверами. Ефективна інтернет-комунікація дозволяє реалізовувати масштабовані, швидкі та гнучкі системи. У цьому підрозділі буде розглянуто найпоширеніші методи інтернет-комунікації: REST, RPC, а також WebSocket як приклади альтернативних підходів.

REST – це архітектурний стиль взаємодії між компонентами розподіленої системи, який базується на використанні стандартних HTTP-методів (GET, POST, PUT, DELETE тощо). Основною ідеєю REST є робота з ресурсами, кожен з яких має унікальний ідентифікатор (URL). Стиль активно використовується у створенні веб-API, мобільних застосунків та мікросервісів. До переваг стилю відносять: простоту реалізації, широку підтримку у веб-середовищі та зрозумілу й інтуїтивну структуру запитів. До недоліків, у свою чергу, можна віднести обмежену гнучкість при складних запитах та незалежність кожного запиту, що ускладнює підтримку стану (stateless).

RPC –метод програмної інтернет-комунікації, який дозволяє клієнту викликати функції або процедури, що виконуються на віддаленому сервері, наче вони локальні. Один із сучасних прикладів реалізації RPC – це gRPC, який використовує Protocol Buffers (Protobuf) для серіалізації даних. Метод особливо добре підходить для внутрішньої взаємодії між мікросервісами в системах, орієнтованих на продуктивність. До переваг методу відносять: високу швидкодію та ефективність, підтримку суворої типізації, стрімінгу та двосторонньої комунікації (для деяких представників цього методу комунікації). До недоліків відносять складність налаштування, необхідність

відтворення великої кількості шаблонного коду, незручну відладку, порівняно з REST.

WebSocket– це протокол, що забезпечує постійне двостороннє з'єднання між клієнтом і сервером. На відміну від REST або RPC, де комунікація ініціюється клієнтом, WebSocket дозволяє і клієнту, і серверу надсилати повідомлення в будь-який момент часу. Використовується у чатах онлайн відео-іграх,системах моніторингу в реальному часі.

GraphQL – це запитова мова для API, яка дозволяє клієнтам запитувати саме ті дані, які їм потрібні, і нічого зайвого. Це гнучка альтернатива REST у випадках, коли REST-інтерфейс занадто обмежений або створює надмірність даних. На відміну від REST-підходу, GraphQL лише надає клієнтам один ендпоінт для надсилання запитів. До головних переваг відносять можливість отримування складно-структурованих даних лише за один запит, оскільки клієнт сам обирає які дані бажає отримати.

На момент написання, стиль REST досі залишається найпопулярнішим через свою простоту та сумісність. Його сучасніші альтернативи – RPC і GraphQL використовуються для забезпечення вищої продуктивності або гнучкості. WebSocket– незамінний у реальному часі, де потрібна постійна взаємодія.

1.2 Аналіз існуючих інструментів програмної інтернет-комунікації

Найбільш розповсюдженим шляхом комунікації двох комп’ютерних програм є RESTклієнт-серверна взаємодія, нижче приведено загальні етапи виконання комунікації таким методом:

- клієнт ініціалізує зв’язок із сервером;
- відбувається так зване «TLSрукостискання», при якому сторони обмінюютьсяTLS/SSLкрипто даними один з одним для підтвердження безпечного подальшого зв’язку;

- вдале з’єднання підтверджується з обох сторін – сервер готовий приймати запити зі стороні клієнта;
- клієнт надсилає запит на необхідний ендпоінт, доступ до якого надає сервер;
- сервер оброблює запит та надсилає відповідь назад до клієнта.

Майже кожна сучасна мова програмування надає інструменти для реалізації такого виду взаємодії. Загально ці інструменти, а саме, вбудована функціональність обраної мови програмування, або ж, окремо завантаженні бібліотеки та фреймворки, можна поділити на два основні типи: клієнтські та серверні.

Клієнтські HTTP-бібліотеки є фундаментальними інструментами для взаємодії програм із веб-сервісами. Вони забезпечують можливість надсилання HTTP-запитів (GET, POST, PUT, DELETE тощо), обробку відповідей, роботу з заголовками, параметрами, аутентифікацією, сесіями тощо. Такі бібліотеки дозволяють будувати клієнтські частини REST-архітектури, інтегруватися з API сторонніх сервісів або організовувати внутрішню взаємодію компонентів розподіленої системи. Серед популярних представників цього класу інструментів, для мови програмування Python, можна згадати такі бібліотеки, як “requests” або “httpx”, які реалізують HTTP-клієнти у високорівневому стилі з акцентом на зручність використання.

Серверні HTTP-бібліотеки надають користувачу, потенціально, легко та зручно запустити сервер у своєму застосунку та мати можливість отримувати та обробляти HTTP-запити та формувати відповідні відповіді. Такі фреймворки та бібліотеки забезпечують маршрутизацію запитів, підтримку проміжного програмного забезпечення (middleware), роботу з формами, JSON, автентифікацію, валідацію даних та інші функції. Вони є базою для створення RESTful API, веб-додатків і мікросервісів. Як приклади серверних фреймворків та бібліотек, для мови програмування Python, можна назвати Flask, FastAPI та Express.js, Spring Boot для мови програмування JavaScript, які дозволяють

реалізовувати HTTP-сервери з різними рівнями складності та масштабованості.

Зазвичай клієнт – серверна взаємодія відбувається саме використовуючи протокол HTTP версії 1.1, що була випущена у 1997-ому році, та є досить застарілою. Більшість компаній, що надають своїм користувачам взаємодіяти із їхніми сервісами програмно, досі використовують HTTP/1.1 для створення своїх API.

Дані, в свою чергу, зазвичай, передаються у форматі JSON, саме цей формат можна сміливо назвати стандартом індустрії.

Через використання вищеназваних технологій та форматувань стандартна REST комунікація має середні показники швидкості та ефективності, оскільки має ряд обмежень щодо розміру та продуктивності [1]. Тому, у випадках, коли необхідно реалізувати внутрішню комунікації у замкнутих системах, де ви маєте повний контроль над обома сторонами зв'язку, то іноді краще подивитися у сторону альтернативних інструментів програмної інтернет-комунікації, а саме RPC.

RPC – альтернативний інструмент програмної інтернет-комунікації. Для задач, де необхідна структурована взаємодія між компонентами за принципом виклику віддалених процедур, використовуються інструменти, що реалізують відповідні протоколи. Такі бібліотеки забезпечують автоматичну генерацію клієнтських і серверних обгорток, підтримку строгих типів, серіалізацію/десеріалізацію даних у компактних форматах, а також оптимізовану передачу повідомлень. Ядро алгоритму роботи RPC, загалом, засновується на згаданій раніше REST-взаємодії, але зі значними модифікаціями.

Нижче приведений загальний алгоритм роботи RPC бібліотек:

- сторона відправник ініціалізує зв'язок із сервером;
- відбувається так зване «TLS рукостискання», при якому сторони обмінюються TLS/SSL крипто даними один з одним для підтвердження безпечного подальшого зв'язку;

- вдале з’єднання підтверджується з обох сторін – сторона отримувач готова приймати запити зі сторони відправника,зазвичай сторона отримувач відкриває лише один ендпоінт, наприклад “/rpc”, та очікує вхідних запитів лише на нього;

- відправник викликає віддалену процедуру, передаючи її назву та аргументи, як звичайну локальну функцію, зазвичай серіалізовані у компактний формат даних, такі як MessagePack [2]та Protobuf (використовується у gRPC);

- відправник надсилає запит на необхідний ендпоінт, доступ до якого надає сервер;

- сервер отримує запит, десеріалізує дані, отримуючи ім’я процедури та аргументи;

- виконує відповідну процедуру, результат серіалізує у відповідний формат та відправляє відповідь стороні відправнику.

Найбільш популярним представником RPC бібліотек та фреймворків є gRPC, створена командою компанії Google у 2016-ому році. Вона використовує компактний бінарний формат даних Protobuf, що потребує типізованої схеми та відповідного файлу схемою для кожної процедури та сервісу під’єднаного до інстанції gRPC. Для комунікації бібліотека використовує протокол HTTP версії 2, що також вийшла у 2016-ому році,і є значно швидшою за попередню версію HTTP/1.1.

Комбінація компактного форматування даних запитів, та використання сучасних ефективних протоколів надає бібліотеці значну перевагу у швидкості, та продуктивності порівняно з іншими бібліотеками свого типу, та, загалом, іншими типами програмної інтернет-комунікації [3].

Однак, бібліотека також має деякі недоліки. До них відносять:

- значно вищий поріг входу, порівняно з REST;

- обмеженість у використані системи через браузер;

- необхідність створення файлів схем, використання автоматичної генерації коду, вивчення принципів роботи з форматом Protobuf.

Тож, gRPC може не підійти у випадках невеликого досвіду у програмуванні, необхідності використання більш динамічного фреймворку, та випадках коли розробник прагне використовувати більш інтуїтивний та простіший інтерфейс, та уникнути створення зайвого шаблонного коду.

1.3 Комунікація в IoT системах

Інтернет речей (IoT) є галуззю, що стрімко розвивається [4]. Ефективна комунікація між пристроями є одним із ключових аспектів побудови надійної та масштабованоїIoT-системи. Найбільш поширеними засобами передачі даних у таких системах традиційно є легковагові мережеві протоколи, зокрема MQTT,CoAP та AMQP [5], які забезпечують мінімальне споживання ресурсів та підтримку роботи у нестабільних мережевих умовах. Водночас, з розвитком обчислювальних можливостей пристроїв і появою потреби в більш уніфікованих та гнучких рішеннях, усе ширше впроваджуються комунікаційні моделі, що базуються на HTTP-запитах.

Зокрема, дедалі більшого поширення набувають підходи, орієнтовані на використання протоколів та методів комунікації прикладного рівня – таких як REST або RPC, які дають змогу побудувати зрозумілі та розширювані API для взаємодії між компонентами системи. Використання HTTP як транспортного шару спрощує інтеграцію з веб-сервісами, покращує інтер-операбельність та забезпечує кращу підтримку засобів безпеки. RPC-підхід, зокрема, дозволяє зменшити обсяг передаваних даних та реалізувати більш тісну зв'язку між клієнтом і сервером, що особливо важливо для критичних IoT- застосунків, де мають значення швидкість реакції та продуктивність.

1.4 Визначення ніші бібліотеки

У межах цієї дипломної роботи передбачається створення спеціалізованої HTTP-бібліотеки для IoT-систем, яка займе нішу простих, інтуїтивно зрозумілих та зручних у використанні засобів комунікації. Основною метою розробки є забезпечення максимальної доступності для широкого кола розробників – як початківців, так і досвідчених фахівців. Бібліотека повинна стати ефективним інструментом для побудови клієнт-серверних застосунків у сфері IoT, та загалом універсальною у використанні у також і у інших сферах програмування, без необхідності глибокого занурення в складні технічні деталі мережевих протоколів або попередню генерацію коду.

Запропоноване рішення має на меті стати альтернативою поширеним підходам, зокрема бібліотекам для реалізації REST-архітектури та gRPC. Обидва підходи, хоч і широко використовуються, мають певні обмеження: REST-бібліотеки часто вимагають ручної обробки HTTP-запитів і відповіді, що ускладнює структурування коду, а gRPC, навпаки, потребує генерації інтерфейсів та додаткових інструментів, що робить його менш зручним для швидкої розробки та прототипування. У цьому контексті нова бібліотека має поєднувати кращі сторони обох технологій: залишатися легкою та інтуїтивно зрозумілою, водночас забезпечуючи високу продуктивність, швидкодію та ефективність передачі даних.

Крім того, особлива увага в проєкті приділятиметься простоті інтеграції бібліотеки у типові IoT-застосунки. Вона повинна стати стартовим інструментом для новачків у сфері IoT, спрощуючи перші кроки у створенні програмної логіки для мікроконтролерів, сенсорів та серверів. Передбачається, що бібліотека надасть зрозумілий програмний інтерфейс (API) та дозволить швидко організовувати обмін даними між компонентами системи без потреби в зайвій конфігурації або зовнішніх залежностях.

Таким чином, цільовою нішею майбутньої бібліотеки є проєкти, які потребують мінімалістичного, продуктивного та водночас дружнього до користувача рішення для мережевої взаємодії.

1.5 Вибір мови програмування

Для розробки бібліотеки було прийнято рішення використовувати мову програмування Python. Такий вибір обумовлений рядом технічних та практичних переваг.

По-перше, Python є однією з найпоширеніших мов серед початківців у галузі програмної інженерії. Її простий синтаксис, велика кількість документації та активна спільнота значно спрощують вивчення й подальше використання, що особливо актуально для молодих фахівців, які роблять перші кроки у сфері IoT [6].

По-друге, Python демонструє стабільне зростання популярності у сфері Інтернету речей, що обумовлено як розвитком апаратного забезпечення (мікроконтролери, одноплатні комп’ютери на зразок Raspberry Pi), так і загальним підвищенням обчислювальних можливостей вбудованих систем [7]. Завдяки цьому Python дедалі частіше використовується не лише для прототипування, а й у продуктивних рішеннях.

Крім того, Python забезпечує високу швидкість розробки програмного забезпечення порівняно з багатьма іншими мовами. Це дозволяє оперативно створювати, тестувати та інтегрувати нові функціональні компоненти, що є критично важливим у швидкозмінному середовищі розробки IoT-рішень.

1.6 Вибір технологій для реалізації проекту

Для реалізації бібліотеки було обрано низку сучасних технологій, які відповідають вимогам продуктивності, безпеки та зручності використання в контексті IoT-систем. У якості формату обміну даними було вирішено

використовувати MessagePack– компактний бінарний формат, що забезпечує гнучкість і динамічність структурування інформації. На відміну від ProtocolBuffers, який використовується у gRPC і вимагає попереднього визначення схем, MessagePack дозволяє працювати зі структурами даних більш природним для Python способом, що не потребують попередньо згенерованих схем, і, як наслідок, значно спрощує розробку та налагодження продукту [8].

Як транспортний протокол для передачі даних обрано HTTP/3, останню версію прикладного протоколу, офіційно затверджену у 2022 році [9]. HTTP/3 побудовано на базі протоколу QUIC, який забезпечує повну мультиплексність, суттєве підвищення швидкості з'єднання, стабільності та швидкості передачі даних та захищеності порівняно з попередніми версіями, зокрема HTTP/2, що використовується у gRPC [10, 11]. Основою бібліотеки обрано aioquic – низькорівневу асинхронну Python-бібліотеку, яка реалізує ключові механізми протоколу QUIC і підтримує HTTP/3. Вона дозволяє безпосередньо взаємодіяти з мережевими шарами, надаючи розробнику гнучкий контроль над процесом обміну даними [12].

Для автоматичної верифікації та декодування вхідних повідомлень на стороні сервера використовується бібліотека «msgspec», розроблена авторами Python-реалізації MessagePack [13]. Вона дозволяє ефективно описувати структури даних, автоматично перевіряти типи та значення,та їх кількість, що значно знижує ризик помилок при обробці запитів.

Для реалізації асинхронної логіки та обробки одночасних підключень використовується стандартна Python-бібліотека «asyncio», яка забезпечує підтримку паралельних операцій у мережевому середовищі.

Таким чином, вибраний стек технологій орієнтований на сучасні вимоги до швидкодії, безпеки, гнучкості та зручності інтеграції в екосистему Python та IoT.

1.7 Транспортний протокол QUIC

У якості транспортної основи для реалізації бібліотеки було обрано протокол QUIC (HTTP/3) (англ. Quick UDP Internet Connections) – сучасне рішення, що поєднує в собі переваги швидкодії, безпеки та надійності. QUIC розроблений як заміна традиційної зв’язки TCP + TLS і є основою для нового протоколу прикладного рівня HTTP/3 [14].

Однією з головних причин вибору QUIC є його висока продуктивність. Завдяки використанню UDP замість TCP, QUIC дозволяє зменшити затримки при встановленні з’єднання, забезпечує мультиплексування потоків без блокування (head-of-line blocking) і ефективніше працює в умовах нестабільних мереж, що є типовим для IoT-середовища.

Ще однією перевагою є вбудована підтримка шифрування з самого початку з'єднання, що усуває потребу в окремому етапі TLS-handshake. Крім того, QUIC забезпечує автоматичне шифрування всіх пакетів, включаючи заголовки, що підвищує рівень безпеки.

Оскільки QUIC є офіційною основою HTTP/3, затвердженого у 2022 році, його використання гарантує відповідність сучасним стандартам інтернет-комунікацій та забезпечує сумісність з останніми технологічними рішеннями. Це робить його ідеальним кандидатом для побудови нових, ефективних і масштабованих мережевих протоколів у рамках IoT-систем.

1.8Постановка завдання

Метою даної кваліфікаційної роботи є розроблення високопродуктивної та зручної у застосуванні HTTP бібліотеки RPC-типу мовою програмування Python, згідно обраному набору технологій, та рішень.

Для розробки головної бібліотеки RPC бібліотеки, також,передбачено створення допоміжної високорівневої клієнт-серверної HTTP бібліотеки, що слугуватиме основою для створення результуючої RPC бібліотеки

Також, на меті є аналіз розробленої бібліотеки та її тестове використання та демонстрація у програмній системі IoT, як інструменту комунікації, як між програмними компонентами в межах серверної частини системи, так і між індивідуальними IoT-приладами та серверною частиною застосунку.

2ПРОГРАМНА РЕАЛІЗАЦІЯ ПРОЕКТУ

2.1 Загальна логіка та алгоритм роботи

Загальний принцип та алгоритм роботи бібліотеки дуже схожий на алгоритми інших представників класу RPC. Нижче представлено загальну логіку функціонування розроблюваної бібліотеки, а також покроковий алгоритм її роботи в контексті типового IoT-застосунку, на прикладі типового запиту від клієнта серверу. Наведений опис дозволяє зрозуміти основні етапи обробки запитів, організації комунікації між клієнтом і сервером, а також принципи взаємодії з прикладними компонентами системи. Алгоритм побудований з урахуванням асинхронної архітектури та особливостей використаних технологій:

- крок 1 – користувач викликає віддалену процедуру (функцію), використовуючи її назву та необхідні параметри, та інтернет адресу – IP та порт, або домен;
- крок 2 – клієнт отримує вхідні параметри запиту. Форматує тіло запиту у двійковий формат MessagePack, приклад тіла запиту представлений у лістингу 3.1,де перше значення списку містить ім’я процедури, друге – індекс типу виклику, а третє – список/колекцію аргументів до виклику цієї процедури;
- крок 3 – клієнт ініціалізує зв’язок з сервером. Відбувається DNS-пошук адреси домену, з яким треба встановити з’єднання;
- крок 4 – сервер отримує запит на створення з’єднання, від клієнта.
- крок 5 – відбувається обмін сертифікатами та іншими крипто даними між двома сторонами;
- крок 6 – сервер підтверджує створення з’єднання з клієнтом. Об’єкт класу з’єднання додається до пулу підключень (connection pool);
- крок 7 – клієнт отримує підтвердження вдалого встановлення підключення;

- крок 8 – клієнт відкриває декілька UDP потоків (streams), для реалізації мультиплексності, тобто виконання паралельних запитів використовуючи лише одне єдине підключення;

- крок 9 – клієнт починає надсилати дані запиту на стандартний ендпоінт з назвою “/ezrpc”, HTTP-методу POST, відкритий сервером;

- крок 10 – спочатку клієнт надсилає заголовки запиту, такі як “content-type”, “:path”, “authority”, “:method”;

- крок 11 – сервер отримує заголовки, дешифрує їх, перевіряє наявність усіх необхідних полей. У разі правильності зберігає дані заголовків та індекс потоку за якого вони були відправленні;

- крок 12 – клієнт надсилає тіло запиту відформатоване у MessagePack двійковий код, також при відправленні, клієнт надсилає сигнал на закриття потоку;

- крок 13 – сервер отримує тіло запиту, зберігає його, у разі отримані сигналу на закриття потоку, одразу переходить до обробки запиту;

- крок 14 – при обробці запиту сервер перевіряє ендпоінт на який був зроблений запит, тобто, якщо у полі “:path” не було передано стандартне значення “/ezrpc”, то сервер повертає повідомлення про помилку клієнту;

- крок 15 – сервер, одночасно верифікує та дешифрує дані тіла запиту, використовуючи інструменти пакету msgspec (бібліотека одразу перевіряє тип та кількість аргументів,що були надіслані у третьому значенні тіла запиту. У разі помилки верифікації, надсилає клієнту повідомлення про неправильну кількість та/або тип переданих аргументів під час виклику процедури. Стандартна структура відповіді сервера представлена в лістингу 2.2, де перше значення–містить текстпомилки, що виникла під час обробки запиту, а друге – результат роботи процедури – значення, що було повернуто процедурою);

- крок 16 – у разі правильності даних запиту, сервер викликає процедуру ім’я якої було зазначено у першому значенні тіла запиту, разом з аргументами надісланими у третьому значенні;

- крок 17 – у разі вдалого виконання, сервер отримує значення, що було повернуто процедурою, формує структуру відповіді, яку можна побачити в лістингу 2.2, де перший елемент списку отримує значення null, а другий – результат роботи повернутий процедурою (якщо ж, під час виконання процедури виникає помилка, сервер формує структуру відповіді, де у першому значенні елементі списку, передає текст помилки);
- крок 18 – сервер форматує дані тіла відповіді в MessagePack двійковий код, та надсилає відповідь клієнту, після чого закриває потік, та очищає дані збережені про цей запит;
- крок 19 – клієнт отримує відповідь, дешифрує її, і, у разі, якщо перший елемент дешифрованого тіла відповіді не дорівнює null та містить текстове значення, то програма викликає виняток (Exception) з цим зазначеним текстом (у разі, якщо перший елемент не містить текстового значення, і дорівнюєnull, то програма витягує значення зазначене у другому елементі списку, та повертає його користувачу);
- крок 20 – цикл RPCзапит-відповідь завершено.

Лістинг 2.1 – стандартне тіло запиту від RPC-клієнта серверу, до форматування у MessagePack двійковий код

```
["get_sum", 0, [1, 2, 3, 4, 5]]
```

Друге значення у тілі запиту є індексом типу виклику процедури, за допомогою цього значення клієнт «каже» серверу як він бажає, щоб сервер виконав обрану процедуру. Нижче перелічено доступні користувачам типи викликів процедур:

- індекс 0 – стандартний шлях виклику процедури, сервер перевіряє отримані дані виконує вказану процедуру, отримане значення передає клієнту;

- індекс 1 – сервер повертає відповідь перед викликом самої процедури. Як тільки дані отримані та верифіковані, сервер повертає відповідь клієнту про вдалий запит процедури, після чого здійснює виклик вказаної процедури. Даний шлях виклику може стати в нагоді коли процедура має довге очікуване виконання, та клієнту не важливий її результат;
- індекс 2 – односпрямований виклик (англ. Fire-And-Forget) – клієнт здійснює виклик процедури і не дочікується відповіді від сервера, після надсилання запиту на виклик, і продовжує виконання програми.

Коли сервер отримує такий тип запиту він викликає процедуру, та не надсилає клієнту відповіді. Такий шлях виклику може знадобитися у випадках, коли виконання процедури не має важливого значення для клієнта, та у пріоритеті стоїть швидкість, а не надійність.

Лістинг 2.2 – стандартне тіло відповіді від RPC-сервера клієнту, до форматування у MessagePack двійковий код

```
[null, 15]
```

Загальну схему роботи та цикл запит-відповідь у контекстірезультуючої бібліотеки – "ezRPC", що засновується на базовій бібліотеці "ezH3" можна побачити на рисунку 2.1.

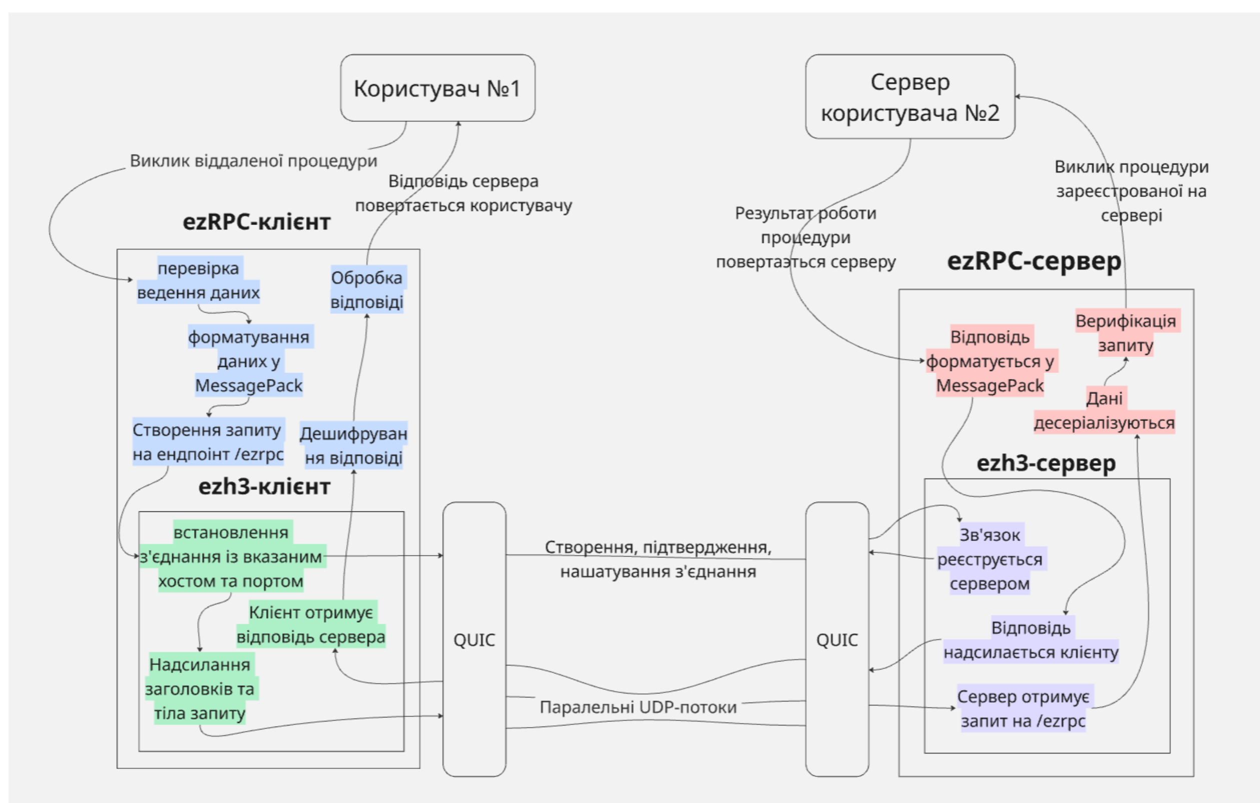


Рисунок 2.1 – Загальна схема алгоритму роботи результуючої бібліотеки, на прикладі стандартного циклу запит-відповідь

2.2 Базова REST клієнт-сервер бібліотека

Для полегшення реалізації логіки фінальної RPC бібліотеки, також буде створена високорівнева REST HTTP бібліотека, яка буде використана як основа для створення першої.За основу базової бібліотеки буде взято функціональність низькорівневої бібліотеки “aioquic”, яка забезпечує роботу з протоколом QUIC та HTTP/3. Завдяки використанню базової бібліотеки, реалізація RPC значно спрощується: код стає модульним, з чітко розділеними зонами відповідальності, а компоненти системи – більш автономними та підтримуваними.

Фінальна RPC бібліотека буде успадковувати функціональність базової REST бібліотеки, з оптимізаціями та адаптаціями, які відповідатимуть

специфіці виклику віддалених процедур. Завдяки цьому повторне використання коду буде максимальним, а структура – більш узгодженою.

Окрім того, базова бібліотека може використовуватись окремо від RPC-рівня як самостійний інструмент для реалізації REST HTTP/3 клієнт-серверної взаємодії в будь-якій сфері: IoT, веб-сервіси, мобільні застосунки тощо.

2.2.1 REST клієнт

REST клієнт базової бібліотеки відповідає за повноцінну реалізацію HTTP/3-клієнтської логіки з урахуванням сучасних вимог до мережевої взаємодії. Його основними можливостями є:

- управління з'єднаннями та пулом з'єднань – клієнт забезпечує повторне використання активних QUIC-з'єднань, що знижує час затримки при повторних запитах і покращує продуктивність;
- підтримка TLS та автоматична генерація ключів, якщо це необхідно – клієнт забезпечує безпечне з'єднання через QUIC/TLS 1.3 без додаткової конфігурації з боку користувача;
- DNS-резолвінг – клієнт самостійно виконує розпізнавання доменного імені в IP-адресу перед встановленням з'єднання, включаючи підтримку IPv6;
- підтримка dual-stack логіки сокетів – навіть якщо запит був ініційований на адресу типу IPv4, клієнт може підключитись до сервера, який працює виключно через IPv6, забезпечуючи високу адаптивність у гетерогенних мережах;
- збереження сесійних тикетів – для мінімізації накладних витрат при повторних з'єднаннях і збереження активного сеансу навіть у разі зміни IP-адреси (наприклад, при зміні мережі на мобільному пристрої);
- підтримка всіх стандартних HTTP методів – клієнт має зручні програмні інтерфейси для виклику GET, POST, PUT, DELETE, PATCH та інших методів згідно зі стандартом HTTP/3;

- обробка мережевих помилок – реалізовано перехоплення, логування та класифікація помилок з можливістю повторної спроби або ескалації;

- конфігурований інтерфейс – REST клієнт дозволяє гнучко задавати таймаути, заголовки, параметри з'єднання, використання проксі та інші ключові параметри взаємодії, що робить його зручним для використання у широкому спектрі застосунків.

Цей клієнт є основою для майбутньої RPC взаємодії, але також може використовуватись як універсальний HTTP/3-клієнт у будь-якому асинхронному Python-застосунку. Приклад використання базового клієнту представлений в лістингу 2.3.

Лістинг 2.3 – Приклад використання базового REST-клієнту

```
from ezh3.client import Client
import asyncio

async def main():
    client = Client("https://127.0.0.1:8000", use_tls=False, timeout=None)
    response = await client.post("/echo",
                                 json={"word": "Hello!"},
                                 headers={"User-Agent": "Vadim Seliukov"})

    if not response.ok:
        print(f"Server error: {response.text}")
        return

    data = response.json()
    print(data)

if __name__ == "__main__":
    asyncio.run(main())
```

2.2.2 REST сервер

REST сервер базової бібліотеки забезпечує ефективну та гнучку реалізацію серверної частини HTTP/3 взаємодії. Його архітектура дозволяє швидко створювати серверні застосунки з мінімальними зусиллями, при цьому зберігаючи високий рівень контролю та налаштування:

- зручні інтерфейси для створення endpoint'ів – сервер надає прості механізми для визначення обробників запитів для всіх стандартних HTTP методів: GET, POST, PUT, DELETE, PATCH тощо, що значно спрощує створення RESTAPI;

- базова верифікація запитів – реалізовано початкову перевірку структури та заголовків вхідного запиту, що дозволяє одразу відсікати некоректні або потенційно небезпечні запити ще до передачі їх до логіки обробки;

- маршрутизація запитів та виклик відповідного обробника – сервер автоматично визначає, який хендлер відповідає конкретному HTTP запиту на основі методу та шляху, забезпечуючи чітке розділення логіки та масштабованість;

- підтримка запуску на IPv4 та IPv6 — сервер може бути запущений як на класичних IPv4-адресах, так і на сучасних IPv6-інтерфейсах, що забезпечує широку мережеву сумісність і готовність до використання у майбутніх інфраструктурах;

- гнучка конфігурація – REST сервер дозволяє гнучко налаштовувати параметри запуску: IP-версію, порт, хост, використання TLS, а також специфікацію сертифікатів та ключів (це робить його придатним як для локальної розробки, так і для розгортання у продакшн-середовищі).

Таким чином, базовий REST сервер забезпечує необхідну основу, а саме створення серверного процесу, можливість отримання, маршрутизації, та обробки запитів, та іншої функціональності переліченої вище. Використовуючи цю основу маємо можливість побудови високопродуктивних HTTP/3 сервісів з

чіткою структурою, безпекою та розширюваністю, що надалі стане базою для реалізації RPC архітектури. Приклад використання базового серверу представлений у лістингу 2.4.

Лістинг 2.4 – Приклад використання базового REST-серверу

```
import asyncio
from ezh3 import Server, ServerRequest

app = Server(
    enable_tls=True,
    custom_cert_file_loc="/app/cert.pem",
    custom_cert_key_file_loc="/app/key.pem"
)

@app.get("/")
async def home():
    return {"message": "Welcome to QUIC Server"}

@app.post("/echo")
async def echo(request: ServerRequest):
    data = request.json()
    return data


if __name__ == '__main__':
    asyncio.run(app.run(port=8000, host="0.0.0.0", enable_ipv6=True))
```

2.3 Фінальна RPC бібліотека

Фінальна RPC бібліотека є завершальним рівнем абстракції в рамках побудови клієнт-серверної системи на основі протоколу HTTP/3. Вона використовує базову REST бібліотеку як основу, наслідуючи її низькорівневу мережеву логіку, обробку з’єднань, роботу з QUIC/TLS, а також механізми маршрутизації й обробки запитів.

На базі цієї фундаментальної інфраструктури, RPC бібліотека реалізує спеціалізовану модель взаємодії, орієнтовану на виклик віддалених процедур. Для цього були переосмислені та частково переозначені деякі класи, структури даних і логіка обробки повідомлень. Таке переозначення дозволяє мінімізувати накладні витрати HTTP-комунікації у контексті RPC, спростити інтерфейси для розробника, зменшуючи кількість коду, необхідного для створення викликаючої або обробної сторони.

Також фінальна бібліотека реалізує нову функціональність унікальну саме для неї та інших представників RPC, такі як виявлення сервісів (servicediscovery), автоматичне налаштування та конфігурація, автоматична верифікація запитів, підтримка використання методів інстанцій класів, та ін..

2.3.1 RPC клієнт

RPC клієнт, наслідує більшість своїх інтерфейсів від базового клієнту. Частина інтерфейсів була повністю, або ж, частково змінена. Так наприклад, фінальний клієнт не приймає параметрів шляху при ініціалізації об'єктів класу клієнту, або при здійсненні виклику віддаленої процедури. Кожен запит робиться на стандартній ендпоінт – "/ezrpc". Також змінена логіка шифрування тіла запитів, та дешифрування тіла відповідей від сервера назад клієнту. Уся комунікація між сторонами здійснюється у форматі MessagePack двійкового коду, який у деяких випадках може зменшити розмір відправлених даних на 80%.

Оскільки заголовки запитів не змінюються після встановлення з'єднання, клієнту доводиться оброблювати однакові значення при кожному виклику процедури. Тож, для оптимізації процесу відправки заголовків, було реалізовано механізм кешування вже оброблених та готових до відправки полей заголовків – клієнт перевіряє актуальність заголовків при кожному виклику, у разі вдалого результату перевірки надсилає серверу збережені кешовані значення, у разі невдалого – заново кешує та зберігає нові заголовки.

Також, RPC-клієнт реалізує функціональність виявлення сервісів (англ. Service Discovery). Для цього одразу на серверній та клієнтській частинах бібліотеки додано підтримку системної процедури з кодовою назвою ":d", що може бути викликана методом "discover". У разі виклику клієнт отримає дані типу хеш-мапа (dictionary), у якій перелічені публічно доступні для виклику процедури, типи та кількість аргументів, що вони приймають, типи даних,які повертають, та описи цих процедур.

Для RPC-клієнта, також, була реалізована підтримка різних типів викликів процедур, що були описані у підрозділі 2.1.

Окрім системного методу виявлення сервісів, у RPC-клієнті ще було реалізовано системну функцію пінг,із кодовою назвою ":p", та метод за допомогою якого виклик цієї процедури може бути здійснений – "ping". Ця процедура не приймає жодних аргументів та повертає значення типу null. Вона може бути використана користувачем для тестування та налаштовування своєї закритої програмної системи. Приклад використання фінального клієнта представлений у лістингах2.5, 2.6, 2.7.

Лістинг 2.5 – Приклад використання фінального клієнта

```
from ezRPC import Producer
import asyncio

asyncdefmain():
    # створення інстанції класу Producer, тобто RPC клієнт
client = Producer("https://127.0.0.1:8080", use_tls=True, timeout=None)

    # отримання всіх доступних процедур сервера
functions = awaitclient.discover()
print(functions)

    # Виклик процедури сервера get_sum, надсилання її назви та аргументів
result = await client.call("get_sum", 1, 2)
    print(result)       # обробка отриманого результату
```

```
if __name__ == "__main__":
    asyncio.run(main())
```

Лістинг 2.6 – Приклад використання системних процедур у фінальному клієнті

```
from ezRPC import Producer
import asyncio

asyncdefmain():
    # створення інстанції класу Producer, тобто RPC клієнт
client = Producer("https://127.0.0.1:8080", use_tls=True, timeout=None)

    # виклик системної процедури виявлення сервісів,
functions = awaitclient.discover()
print(functions)

    # Виклик процедури pingдля перевірки роботи сервера та клієнта
result = await client.ping()

if __name__ == "__main__":
    asyncio.run(main())
```

Лістинг 2.7 – Приклад використання різних видів викликів процедур у фінальному клієнті

```
from ezRPC import Producer, CallType
import asyncio

asyncdefmain():
    # створення інстанції класу Producer, тобто RPC клієнт
client = Producer("https://127.0.0.1:8080", use_tls=True, timeout=None)

    # виклик процедури без очікування відповіді сервера з її результатом,
result = awaitclient.call("get_sum", 1, 2,
call_type=CallType.NOT_AWAITED_RUN_CALL)
print(result)           # None
    # виклик процедури без очікування будь-якої відповіді сервера
result = awaitclient.call("get_sum", 1, 2,
call_type=CallType.FIRE_AND_FORGET_CALL)
```

```
    print(result)       # None

if __name__ == "__main__":
    asyncio.run(main())
```

2.3.2 RPC сервер

Як і фінальний клієнт, сервер, у свою чергу, також наслідує більшість своїх інтерфейсів від відповідної частини базової бібліотеки. Так, наприклад, на відміну від базового сервера, RPC сервер активно не керує маршрутизацією запитів до процедур-хендлерів цих запитів, залежно від мережевого шляху вказаного у заголовках цих запитів. Фінальний сервер здатен лише приймати та оброблювати запити зроблені за шляхом “/ezrpc”, у випадках коли клієнтом зроблений запит не за вищепредставленим шляхом, клієнт отримує повідомлення про помилку. Фінальний сервер реалізує автоматичну верифікацію, дешифрування запитів клієнта.

Також, як і клієнт, сервер імплементує підтримку викликів усіх системних процедур, таких як пінг та виявлення сервісів, клас серверу надає зручні інтерфейси для роботи із цими функціональностями, за допомогою відповідних методів – ping та discover.

Сервер реалізує повний контроль над помилками та їх обробку, у разі виникнення помилок, не сумісностей відправлених даних, винятків що відбуваються в межах виконання самих процедур, тощо, сервер проводить фільтрацію, логування цих подій, та надсилає дані про ці помилки клієнту.

І однією з найголовніших змін що були здійснені у RPC-сервері є реалізація зручних та легко-зрозумілих інтерфейсів, що допомагають новачкам у сфері RPC-взаємодії легко налаштовувати та адаптувати свої програмні системи під створену бібліотеку.

Також, у сервер результуючої бібліотеки було додано підтримку одно-сторонньої комунікації під час викликання процедур, що необхідно для підтримки типу виклику процедури Fire-And-Forget, на стороні RPC-клієнта.

Приклади використання RPC-серверу представлені у лістингах2.8, 2.9, 2.10, 2.11.

Лістинг 2.8 – Приклад базового використання RPC-сервера

```
from ezRPC import Receiver
import asyncio

# створення інстанції RPC-серверу (клас Receiver)

app = Receiver()

# Підключення процедури до сервера
@app.function()
async def get_sum(a: int, b: int) -> int:
    result = a + b
    return result



if __name__ == "__main__":
    asyncio.run(app.run())    # запуск серверу
```

Лістинг 2.9 – Приклад налаштування класу серверу

```
# створення та конфігурація інстанції RPC-серверу (клас Receiver)
app = Receiver(

    enable_tls=False,
    custom_cert_file_loc="/app/cert.pem",
    custom_cert_key_file_loc="/app/key.pem",
host="0.0.0.0"
    port=8080,
    enable_ipv6=True
)
```

Лістинг 2.10 – Приклад налаштування процедур

```
from ezRPC import Receiver
import asyncio

# створення інстанції RPC-серверу (клас Receiver)
app = Receiver()

# Підключення процедури до сервера
@app.function(description="Get the total sum of 2 integers - a and b.")
async def get_sum(a: int, b: int) -> int:
    result = a + b
    return result

# Від'єднання процедури від списку доступних процедур, що передається системною
процедурою виявлення сервісів
@app.function(discovery=False)
async def check(word: str = "") -> int:
    result = a + b

    return result

if __name__ == "__main__":
    asyncio.run(app.run())    # запуск серверу
```

Лістинг 2.11 – Приклад додавання інстанцій класів та непрямого підключення процедур

```
from ezRPC import Receiver
import asyncio

classExample:     # Приклад класу
    def __init__(first_name: str = "", last_name: str = "") -> None:
        self.first_name = first_name
        self.last_name = last_name

    def get_full_name() -> str:
        return f"{self.first_name} {self.last_name}"

# створення інстанції RPC-серверу (клас Receiver)
```

```
app = Receiver()
# створення інстанції тестового класу
example = Example("Petro", "Petrenko")

# Підключення інстанції класу та її методів до сервера
app.add_class_instance(example, description="Example Class methods",
      namespace="Example")
#Створення та непряме підключення процедури до сервера
async def get_sum(a: int, b: int) -> int:
return a + b

app.add_function(get_sum, description="Sum of 2 integers")

if __name__ == "__main__":
    asyncio.run(app.run())    # запуск серверу
```

Важливо буде зазначити, що бібліотека підтримує лише стандартні типи даних, а саме: integer, float, null, string, list/collection, tuple та dictionary/key-value map. У разі, якщо користувач спробує зареєструвати, у інстанції класу RPC-серверу, процедуру, що отримує у якості аргументу, або повертає значення, тип якого не належить до перелічених, то бібліотека викликає програмний виняток, із відповідним повідомленням про несумісність процедури через, використовувані нею типи даних.

Те саме правило стосується й RPC-клієнту – у разі передавання аргументу, під час виклику процедури, тип значення якого не є сумісним з бібліотекою, система здійснює програмний виняток, що містить відповідне повідомлення.

Процедури що були підключені через інстанцію класу, тобто методи класу будуть доступні клієнтам через ім'я класу, або через значення що було передано у параметр namespace методу "add_class_instance". Приклад виклику такої процедури на стороні клієнта наведено у лістингу 2.12.

Лістинг 2.12 – Виклик процедур що були зареєстровані у сервері через передану інстанцію класу

```
from ezRPC import Producer
import asyncio

asyncdefmain():
    # створення інстанції класу Producer, тобто RPC клієнт
client = Producer("https://127.0.0.1:8080", use_tls=True, timeout=None)

    # Виклик процедури сервера get_sum, надсилання її назви та аргументів
result = await client.call("Example.get_full_name")
    print(result)       # "PetroPetrenko"

if __name__ == "__main__":
    asyncio.run(main())
```

2.4 Опис загальної функціональності бібліотеки

Розроблювана RPC-бібліотека орієнтована на зручність використання, простоту інтеграції та високу продуктивність у контексті побудови IoT-систем. Однією з ключових переваг є доступний та інтуїтивно зрозумілий інтерфейс, що дозволяє користувачам швидко розпочати роботу з бібліотекою без потреби у складній конфігурації або додатковому навчанні.

Як на клієнтській, так і на серверній стороні реалізовано системні методи, що підвищують зручність та функціональність. Метод discover дозволяє автоматично отримувати інформацію про доступні методи сервера, що в майбутньому дасть змогу реалізувати генерацію RPC-класів на стороні клієнта безпосередньо з командного рядка, використовуючи лише URL-адресу сервера. Метод ping, у свою чергу, призначений для перевірки з’єднання з сервером і може використовуватись для тестування швидкодії обміну повідомленнями через бібліотеку.

Користувач має можливість легко створювати RPC-сервери, реєструвати функції та об'єкти класів, методи яких відповідають вимогам сумісності з бібліотекою. Система автоматично перевіряє типи аргументів та значень, що повертаються, на наявність підтримуваних типів, а також фіксує можливі невідповідності, пов'язані з використанням нестандартних структур даних. Система, автоматично верифікує запити надіслані клієнтом, та повідомляє клієнта про помилку у разі міс-конфігурації запиту, а також, під час виникнення винятків безпосередньо у ході виконання процедури.

На стороні клієнта, бібліотека, також, реалізує підтримку пулінга підключень, що значно оптимізує швидкість її роботи, та пришвидшує запити що робляться за незмінною адресою. А для гнучкості використання у системах з різними версіями IP-адрес, бібліотека, а саме клієнт, також, кожне створюване підключення до сервера виконує у dual-stack режимі сокету. Цей режим надає користувачу здійснювати підключення як до серверів що запущені на IP-адресі версії 4 (IPv4), так і до серверів що запущені на IP-адресі шостої версії (IPv6).

Загалом, бібліотека надає користувачу гнучкі та зручні інтерфейси на стороні клієнта. користувач має можливість здійснення різних видів виклику віддаленої процедури: стандартний, «fire-and-forget» та «un-awaited», з яких може обирати залежно від його/її потреб. Окрім цього бібліотека надає можливість безпечного та небезпечного виклику процедур:

- небезпечний – у разі виникнення помилки у сервері під час обробки запиту на виклик процедури, текст помилки буде повернутий клієнту, і ця помилка буде пропагована як програмний виняток у середовищі клієнта;
- безпечний – замість повернення безпосередньо тих даних що були надіслані сервером, користувач отримує інстанцію класу "ProducerResponse", який є абстракцією на відповідь сервера, та містить поле з даними що були повернуті сервером, та текстове поле помилки, яка могла виникнути під час обробки запиту.

У разі використання безпечного типу виклику процедури, та отримання відповіді сервера, яка містить текстову помилку, клієнт не буде її пропагувати

як програмний виняток, та просто поверне користувачу інстанцію класу "ProducerResponse".

Під час виклику віддаленої процедури та під час конфігурації інстанції класу RPC-клієнта, користувач, також, може встановити ліміт на кількість часу, у секундах, який клієнт буде чекати на відповідь від сервера, тобто – "requesttimeout". У разі якщо клієнт не бажає встановлювати будь яких лімітів, на цей процес, параметр "timeout"встановлюється до значення None (null).

Для зручної роботи та обробки помилок та винятків, також було створено модуль кастомних програмних винятків, які використовуються бібліотекою, та можуть використовуватись користувачем для обробки помилок у його/її системі. Нижче перелічено доступні класи винятків та їх загальний опис:

- "HTTPError" – базовий клас помилок під час роботи бібліотеки, виникає у разі під час мережевих та системних збоїв;

- "HTTPTimeoutError" – виникає коли клієнт очікує відповіді від сервера більше встановленого значення часу;

- "HTTPStatusError" – виняток пропагується коли викликається метод "raise_for_status" класу відповіді на стороні клієнта, та сервер повернув статус код що більший, або дорівнює 400;

- "ArgumentError" – викликається клієнтом якщо сервер повернув помилку про невідповідність типів або кількості аргументів, що були надіслані під час виклику віддаленої процедури;

- "ProcedureNameError" – виникає коли клієнт надсилає запит на виклик процедури, що не зареєстрована у сервері, тобто процедура з вказаним ім'ям не знайдена;

- "ProcedureRunException" – виняток, який пропагується на стороні клієнта у разі, якщо на сервері, під час виконання викликаної процедури, відбувся виняток.

Крім основної функціональності, бібліотека надає широкі можливості для конфігурації серверного середовища. Зокрема, підтримується автоматичне створення самопідписаних TLS-сертифікатів (self-signed), можливість

використання IPv6, ручне налаштування параметрів мережі (IP-адреса, порт), а також встановлення ідентифікатора та опису сервера. Ще однією суттєвою перевагою є висока швидкість обробки запитів – продуктивність бібліотеки наближається до існуючих аналогів, таких як gRPC, msgpack-rpc та JSON-RPC, що робить її ефективним інструментом для розробки високопродуктивних IoT-рішень.

3ТЕСТУВАННЯ ТА ВИКОРИСТАННЯ БІБЛІОТЕКИ

3.1 Використання бібліотеки у демонстраційних застосунках

Для тестування практичного використання бібліотеки було розроблено два застосунки: IoT-сервер та IoT-пристрій – розумна лампа. У приведеній програмній системі IoTбуло рясно використано розроблену бібліотеку та протестовано її на практиці.

3.1.1 Загальна логіка

Тестова система складається з двох програмних компонентів – IoT-пристрою та IoT-серверу. Сервер запушено в дата-центрі, при включені пристрій під’єднується до серверу, за допомогою розробленої HTTPRPC бібліотеки.

Проблема одностороннього зв’язку була вирішена за допомогою методу “shortpolling” – пристрій інтервально, кожні 60 секунд, робить виклик віддаленої процедури на сервері, яка повертає список «завдань» які треба виконати пристрою, це може бути: зниження яскравості лампи, ввімкнення, вимкнення, перезавантаження, тощо. Також, лампа може робити виклики інших процедур, яким вона буде надсилати оновлення режиму своєї роботи, дані про використання для збору статистики, та ін.

3.1.2 IoT-пристрій

Програмна частина пристрою реалізує клас “Actions”, який містить доступні для виконання команди, у вигляді методів класу, такі як “turn_on”, “turn_off”, “restart”, “update”. Пристрій періодично надсилає запити для отримання нових команд, що зберігаються у базі даних, до якої підключений

сервер. Відбувається це, з використанням віддаленої процедури "get_commands" на сервері, додатково, під час виклику процедури, передається ідентифікатор пристрою.

Керування пристроєм відбувається за допомогою цих періодичних команд. Коли пристрій отримує список із командами, він звертається до інстанції класу Actions, та викликає відповідний метод за допомогою системного методу "getattr" мови програмування Python, який приймає текстове значення ім'я методу, та повертає Callable-об'єкт цього методу.

Отримавши список команд, пристрій виконує їх одна за одною. При виконанні кожної команди пристрій посилає статус цього виконання. 2 – вдало виконано, 3 – помилка в ході виконання. Для реєстрації виконання команди, пристрій виконує виклик віддаленої процедури на сервері під назвою "command_completed", разом із назвою пристрій також надсилає ідентифікатор команди та, безпосередньо, статус-код виконання команди

3.1.3 IoT-сервер

При отриманні виклику, сервер, робить пошук у базі даних, та знаходить інформацію про пристрій, а саме: запланований графік/режим роботи, ім'я пристрою, ідентифікатор володаря пристрою, та ін. У випадку, якщо у хвилину, коли пристроєм був зроблений запит до серверу, його режим роботи повинен заплановано змінитися, то сервером створюється нова команда, яка додається в базу даних, і одразу повертається пристрою разом із аргументами.

Коли пристрій викликає віддалену процедуру "command_completed" разом із ідентифікатором команди та результатом її виконання, у вигляді статус коду, то сервер, робить пошук в базі даних, знаходить команду та оновлює її статус відповідно.

Для зберігання даних була обрана база даних PostgreSQL. Було створено таблиці: users, devices, commands.

Також, до серверної частини були підключені HTTP/1.1 RESTAPI, для спрощеної взаємодії використовуючи браузер, за допомогою яких можна отримувати перелік усіх користувачів, оновлювати, додавати та видаляти індивідуальних користувачів. Відповідно можна робити з командами та приладами. Протестувати API можна скопіювавши URL, та вставивши його у браузер.

Усі посилання для взаємодії представлені у додатку Б, разом з посиланнями на відкриті GitHub репозиторії, що містять код базової REST, та фінальної RPC бібліотек.

3.1 Порівняння швидкості роботи бібліотеки за альтернативами

З метою оцінки ефективності розробленої RPC-бібліотеки було проведено серію тестувань на продуктивність у порівнянні з іншими популярними рішеннями для мережевої взаємодії в Python-середовищі. До порівняння було включено такі технології: gRPC, httpx, requests, а також розроблена бібліотека ezRPC.

Тестування виконувалося в однаковому середовищі з метою забезпечення об'єктивності результатів. Усі системи працювали в єдиному доменному просторі, з DNS-резолверами від GoDaddy, а також на хостингу та віддалених віртуальних машинах, наданих провайдером DigitalOcean. Серверна частина була задеплоєна в датацентрі NYC3, а клієнтська – в NYC1, що дозволило мінімізувати вплив внутрішньої мережевої оптимізації та протестувати системи в умовах, наближених до реального мережевого середовища з доступом до відкритого інтернету.

У тестуванні брали участь два окремі застосунки. Перший – IoT-сервер, який реалізовував одразу кілька серверних рішень: gRPC-сервер (для тестування gRPC), FastAPI-сервер (використовувався для перевірки бібліотек “httpx” та “requests”), а також окрема реалізація RPC-сервера на основі

розробленої ezRPC-бібліотеки. Другий застосунок – тестовий клієнт, який виконував серії запитів до кожного з серверів.

Бібліотека “requests” у середовищі розробки на мові програмування Python є стандартним, та знайомим кожному програмісту, інструментом, саме з якої більшість починають свій шлях у Python. Тому, її використання у тестуванні дає нам об’єктивний погляд на швидкість розробленої бібліотеки у порівнянні зі стандартом індустрії.

Бібліотека “httpx” була обрана для порівняння, через свою оптимізованість, гнучку функціональність, а також, як і у випадку з “requests”, досить велику популярність, що є, звичайно, меншою за популярність останньої. Вона є «наступним кроком» професійного розвитку Python-програміста, після бібліотеки “requests”.

Алгоритм тестування полягав у наступному: спочатку встановлювалося з’єднання з сервером, після чого надсилалося 5 підготовчих (warm-up) запитів для стабілізації сеансу. Потім надсилалася серія з 100 послідовних запитів. Час, необхідний на обробку цих 100 запитів, фіксувався в логах, після чого з’єднання закривалося. Цикл повторювався 30 разів, після чого обчислювалось середнє значення часу виконання серії запитів, а також середній час одного запиту.

Для забезпечення чистоти експерименту кожна система тестувалася окремо – тобто інші серверні реалізації на момент вимірювання вимикались, що виключало сторонній вплив на швидкість роботи систем. Підсумкові результати тестування наведені у таблиці 3.1.

Таблиця 3.1 – результати порівняння швидкості бібліотек у виконанні 100 запитів

Бібліотека	Медіанне значення, с.	Середнє, с.
ezRPC (розроблена)	0.253153	0.271211
gRPC	0.201000	0.204690
requests	0.853700	0.847300
httpx	0.439900	0.436700

Як видно з результатів, “ezRPC” показала швидкодію, близьку до gRPC, і значно перевершила традиційні HTTP-клієнтські рішення (“httpx”, “requests”). З огляду на те, що розроблена бібліотека має простий інтерфейс та не вимагає складної конфігурації, отримані показники свідчать про її придатність до практичного використання у високонавантажених IoT-сценаріях. Також, оскільки швидкість виконання одного циклу запит-відповідь займає менше 3 мілісекунд, бібліотека має підстави вважатися високопродуктивною.

У підтвердження результатів у додатку до роботи подано скріншоти із консолі, що приведені у рисунках 3.1, 3.2, 3.3, 3.4, які містять журнали тестових запусків для кожної з бібліотек.

```
[SUMMARY] 30 batches × 100 requests

[SUMMARY] Collected results: [0.21204713099905348, 0.23012752100112266, 0.22360900999956357, 0.21519485099997837, 0.19
776593200003845, 0.19732011499945656, 0.19364387699897634, 0.20105399800013402, 0.21482517600088613, 0.194527812998785
52, 0.20329066599879297, 0.20461477199933142, 0.210272005000661, 0.21153055499962647, 0.18537657399974705, 0.204733974
00019167, 0.23897316699913063, 0.20342808299938042, 0.2414569449992996, 0.1998397669995029, 0.1930715880007483, 0.1979
358129992761, 0.18886928400024772, 0.1990176170002087, 0.20094788999995217, 0.20315267399928416, 0.1989132190010423, 0
.19584333000057086, 0.18893825600025593, 0.19065729700014344]
[SUMMARY] Total average time per 100 request(s): 0.20469929666651296 seconds
[SUMMARY] Total average time per single request: 0.0020469929666651295 seconds
root@ubuntu-s-1vcpu-1gb-nyc1-01:~/ezrpc-tester#
```

Рисунок 3.1 – Результат тестування бібліотеки gRPC

```
Starting batch #30
Batch 30: 100 requests took 0.854333 seconds. Average time per request: 0.008543 seconds

[SUMMARY] Library REQUESTS 30 batches × 100 requests

[SUMMARY] Collected results: [0.8587596260476857, 0.7813346849288791, 0.7545924870064482, 0.7530899558914825, 0.753870
312939398, 0.7389493549708277, 0.7381866779178381, 0.7821060878923163, 0.815447862027213, 0.7542136430274695, 0.758604
2120819911, 0.7895399730186909, 0.7561364729190245, 0.7560842520324513, 0.7713333381107077, 0.8008855129592121, 0.9154
135960852727, 0.853795669041574, 0.8001563049620017, 0.8802223230013624, 0.9141652910038829, 1.3805950379464775, 1.197
0730850007385, 0.9779788659652695, 0.9572173409396783, 0.8570327380439267, 0.8247905949829146, 0.8536186669953167, 0.7
91600905940868, 0.8543333429843187]
[SUMMARY] Total average time per 100 request(s): 0.847370940555508 seconds
[SUMMARY] Total average time per single request: 0.00847370940555508 seconds
STARTING HTTPX TEST
```

Рисунок 3.2 – Результат тестування бібліотеки “requests”

```
Batch 30: 100 requests took 0.422305 seconds. Average time per request: 0.004223 seconds

[SUMMARY] library HTTPX 30 batches × 100 requests

[SUMMARY] Collected results: [0.46685191604774445, 0.4413348399102688, 0.42694103193935007, 0.41656158003024757, 0.421
56338004861027, 0.4353115650592372, 0.42569098400417715, 0.4157966210041195, 0.4543191799893975, 0.48893737397156656,
0.4870379969943315, 0.4756840029731393, 0.49680631898809224, 0.44243334990460426, 0.4163891839561984, 0.39749409700743
854, 0.42451070505194366, 0.4326554180588573, 0.4567780588986352, 0.41086476400960237, 0.41935708501841873, 0.42209366
301540285, 0.4332657390041277, 0.44405595294665545, 0.4288253500126302, 0.4445071720983833, 0.4384810229530558, 0.4045
111481100321, 0.41040159203112125, 0.4223049229476601]
[SUMMARY] Total average time per 100 request(s): 0.4367255338661683 seconds
[SUMMARY] Total average time per single request: 0.004367255338661683 seconds
root@ubuntu-s-1vcpu-1gb-nyc1-01:~/ezrpc-tester#
```

Рисунок 3.3 – Результат тестування бібліотеки “httpx”

```
[SUMMARY] Collected results: [0.4076601549750194, 0.40587350400164723, 0.35564753494691104, 0.301719073089771, 0.29947
074106894433, 0.29977127397432923, 0.2749135729391128, 0.2617057840107009, 0.29173835390247405, 0.3022837280295789, 0.
297070914064534, 0.3198827590094879, 0.26812850101850927, 0.25315315497573465, 0.24582746403757483, 0.2415907640242949
, 0.23530430102255195, 0.25282372010406107, 0.23656500806100667, 0.2395572119858116, 0.23175801790785044, 0.2186928710
4345858, 0.2299586139852181, 0.22765846503898501, 0.20839469996280968, 0.21692650800105184, 0.22676182992290705, 0.215
4618869535625, 0.21069840888958424, 0.35934218391776085]
[SUMMARY] Total average time per 100 request(s): 0.27121136682884145 seconds
[SUMMARY] Total average time per single request: 0.0027121136682884146 seconds
root@ubuntu-s-1vcpu-1gb-nyc1-01:~/ezrpc-tester#
```

Рисунок 3.4 – Результат тестування розробленої бібліотеки“ezRPC”

Як можна побачити,у багатьох випадках розроблена бібліотека демонструє результати рівня gRPC, але є вони менш стабільними, в окремих серіях тестування, що може бути спричинено недостатньою оптимізацією, або нестабільністю транспортного протоколу UDP, що використовується розробленою бібліотекою, у порівнянні з протоколом TCP, що лежить в основі gRPC.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи було розроблено та протестовано високопродуктивну HTTP бібліотеку мовою програмування Python для організації мережевої взаємодії у системах IoT.

Фінальна версія бібліотеки продемонструвала високий рівень продуктивності, стабільну роботу при тестуванні та зручність інтеграції. Завдяки простому інтерфейсу, бібліотека виявилася інтуїтивно зрозумілою та придатною до використання як початківцями, так і досвідченими розробниками. Її структура дозволяє легко організовувати обмін повідомленнями між IoT-пристроями та сервером, що свідчить про її ефективність як інструменту для використання в розподілених системах, та загалом у випадках, що потребують обміну даними по мережі у замкнутих системах.

У рамках тестування бібліотеки було створено два допоміжних програмних рішення: IoT-сервер та програмну імітацію IoT-пристрою. Ці застосунки дозволили провести практичну перевірку працездатності бібліотеки, підтвердити її ефективність, швидкість обміну даними та зручність впровадження у реальні сценарії. Реалізовано обмін та збереження даних у базі даних, комунікацію використовуючи розроблену бібліотеку та RESTAPI на HTTP/1.1.

Таким чином, розроблена бібліотека є готовим рішенням, що поєднує високу продуктивність, сучасний технологічний стек і простоту у використанні, і може бути успішно застосована в широкому спектрі випадків, та IoT-застосунків.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Almallah O. Analytical Assessment of Binary Data Serialization Techniques in IoT Context [Evaluating Protocol Buffers, FlatBuffers, MessagePack, and BSON for Sensor Nodes] [Електронний ресурс] : thesis / Almallah Obada. – Мілан, 2019. – 67 с. – Режим доступу: https://www.politesi.polimi.it/retrieve/a81cb05d-74f5-616b-e053-1605fe0a889a/Thesis_ObadaAlmallah.pdf. – Назва з екрана.

2. Furuhashi S. MessagePack. Efficient binary serialization format [Електронний ресурс] / Sadayuki Furuhashi // MessagePack. – Режим доступу: https://msgpack.org/. – Назва з екрана.

3. Jackson S. Streaming Technologies and Serialization Protocols: Empirical Performance Analysis [Електронний ресурс] / Samuel Jackson. – [Б. м. : б. в.], 2024. – 14 с. – (Препринт / 2407.13494). – Режим доступу: https://doi.org/10.48550/arXiv.2407.13494. – Назва з екрана.

4. Raghunandan S. P. A Survey of Communication Protocols in IoT: MQTT, COAP, and Beyond [Електронний ресурс] / Shyam Phatak Raghunandan // International Journal of Computer Technology and Electronics Communication (IJCTEC). – 2025. – Т. 8, № 1. – С. 55. – Режим доступу: https://philpapers.org/archive/RAGASO.pdf – Назва з екрана.

5. A Performance Analysis of Internet of Things Networking Protocols: Evaluating MQTT, CoAP, OPC UA [Електронний ресурс] / Daniel Silva [та ін.] // MDPI Applied Science. – 2021. – Т. 11, № 11. – app11114879. – Режим доступу: https://doi.org/10.3390/app11114879 – Назва з екрана.

6. Jyothi P. A Review on Python for Data Science, Machine Learning and IOT [Електроннийресурс] / P.N.Siva Jyothi, Rohita Yamaganti // International Journal of Scientific & Engineering Research. – 2023. – Т. 10, № 12. – Режимдоступу: https://doi.org/10.13140/RG.2.2.18708.48000. – Назвазекрана

7. D ' Urso F. Programming Intelligent IoT Systems with a Python-based Declarative Tool [Електронний ресурс] / Fabio D ' Urso // The 18th International Conference of the Italian Association for Artificial Intelligence : Conference Paper, Rende, 25 листоп. 2019 р. – Catania, 2019. – С. 14. – Режим доступу: https://www.researchgate.net/publication/337496375_Programming_Intelligent_IoT_Systems_with_a_Python-based_Declarative_Tool. – Назва з екрана.

8. Kocaöz H. İ. MessageQueues–MessagePackvsJSONforSerialization [Електронний ресурс] / Halil İbrahim Kocaöz // Medium. – Режим доступу: https://halilibrahimkocaoz.medium.com/message-queues-messagepack-vs-json-for-serialization-749914e3d0bb. – Назва з екрана.

9. HTTP/3 [Електронний ресурс] // Wikipedia. – Режим доступу: https://en.wikipedia.org/wiki/HTTP/3. – Назва з екрана.

10. Tellakula S. https://blog.cloudflare.com/http-3-vs-http-2/ [Електронний ресурс] / Sreeni Tellakula // The CloudFlare Blog. – Режим доступу: https://blog.cloudflare.com/http-3-vs-http-2/. – Назва з екрана.

11. Liu F. Performance Comparison of HTTP/3 and HTTP/2: Proxy vs. Non-Proxy Environments [Електронний ресурс] / Fan Liu. – [Б. м. : б. в.], 2024. – (Препринт / 2409.16267). – Режим доступу: https://doi.org/10.48550/arXiv.2409.

16267. – Назва з екрана.

12. Lainé J. QUIC API – aioquic documentation [Електронний ресурс] / Jonathan Lainé // aioquic – aioquic documentation. – Режим доступу: https://aioquic.readthedocs.io/en/latest/quic.html. – Назва з екрана.

13. msgspec [Електронний ресурс] // Jim Crist-Harif. – Режим доступу: https://jcristharif.com/msgspec/. – Назва з екрана.

14. Corbet J. QUIC as a solution to protocol ossification [Електронний ресурс] / Jonathan Corbet // LWN.net. – Режим доступу: https://lwn.net/Articles/745590/ (дата звернення: 11.06.2025). – Назва з екрана.