КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА

Факультет комп'ютерних наук та кібернетики

Кафедра теорії та технології програмування

Кваліфікаційна робота

на здобуття ступеня бакалавра

за спеціальністю 122 «Комп'ютерні науки»

на тему:

RPC фреймворк орієнтований на розробку на C++

Виконав студент 4-го курсу
Бурцевич Владислав Михайлович

(підпис)

Науковий керівник:
Кандидат фізико-математичних наук
Криволап Андрій Володимирович

(підпис)

Засвідчую, що в цій роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент _____

Роботу розглянуто й допущено до захисту на засіданні кафедри теорії та технології програмування
«_____»_____ 2023 р,
протокол №_____
Завідувач кафедри
Нікітченко М.С. _____
(підпис)

Київ – 2023

РЕФЕРАТ

Обсяг роботи 40 сторінок, 21 ілюстрація, 17 джерел посилань.

Об’єктом даного дослідження є RPC - як клас технологій, створених для організації взаємодії між різними компонентами системи.

Предметом дослідження є RPC фреймворк, спроектований для безпечної комунікації різних компонентів системи.

Метою дослідження є розробка RPC фреймворку на основі аналізу вже існуючих бібліотек.

Методами дослідження є дослідницький аналіз, розробка фреймворку, тестування та порівняння, документування результатів. Інструментами розробки є інтегроване середовище розробки Visual Studio, мова програмування C++, бібліотеки: boost, nlohmann/json, openssl, Sublime merge.

Результати роботи: RPC фреймворк, для розробки на C++. Створений для того, щоб полегшити віддалений виклик процедур між різними системами або компонентами. Використовуючи SSL для аутентифікації, він пропонує безпечний канал зв’язку, який гарантує безпечність та цілісність даних. Фреймворк розроблений таким чином, щоб його налаштування та використання було якнайпростішим для користувача.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ

RPC - Remote Procedure Call.

JSON – JavaScript Object Notation.

SSL – Secure Socket Layer.

TLS – Transport Layer Security.

MinGW - набір інструментів розробки програмного забезпечення для створення додатків під Windows.

g++ - C++ компілятор.

ВСТУП

Оцінка сучасного стану об’єкта дослідження. Фреймворки, що реалізують віддалений виклик процедур (від англ. Remote Procedure Call, RPC) залишаються дуже популярними в сучасній розробці програмного забезпечення для побудови розподілених систем і полегшення зв’язку клієнт-сервер. Незважаючи на наявність альтернативних комунікаційних протоколів і фреймворків, RPC-фреймворки мають переваги, завдяки яким залишаються популярними і сьогодні, наприклад спрощені інструменти комунікації, дозволяють підтримувати комунікацію у розподіленому середовищі, підтримують процесорні та потоково-орієнтовані моделі.

Актуальність роботи та підстави для її виконання. Наразі на просторах інтернет мережі існує багато RPC фреймворків, розроблених для використання з мовою C++. Проте жоден з них не надає зрозумілого та інтуїтивного API для використання у комбінації з надійністю і захищеністю переданих даних.

З розвитком інформаційних технологій розвивається і кіберзлочинність, тому важливо захищати канал спілкування від хакерських атак. Навіть якщо через цей канал не передається конфіденційна інформація, все ще є загроза зловживання, несанкціонованого доступу до ресурсів, підриву цілісності даних та інших кібератак. Тому і було вирішено розробити RPC фреймворк, що є стійким до кібер атак та якнайпростішим у використанні.

Мета і завдання роботи. Метою роботи є розробка RPC фреймворку на основі аналізу вже існуючих бібліотек. Для досягнення цієї мети поставлені такі завдання.

- Здійснити порівняльний аналіз найбільш популярних RPC фрейморків з метою ідентифікації їх недоліків та переваг
- На підставі порівняльного аналізу, сформувати набір технічних вимог для розроблюваної бібліотеки, враховуючи недоліки та переваги розглянутих RPC фрейморків.

- Знайти підходящий інструмент для серіалізації та десеріалізації даних, що використовується у контексті розробки RPC функціональності.
- Вибрати відповідний інструмент для реалізації комунікації між сервером і клієнтом у рамках розробки RPC функціональності
- На основі отриманих даних та вимог, розробити унікальний RPC фреймворк з урахуванням виявлених переваг та недоліків розглянутих фреймворків

Можливе застосування. Дослідження буде корисно людям, що обирають інструмент для реалізації клієнт – серверного додатку та мають за критерії простоту використання обраного інструменту комунікації та його безпечність.

РОЗДІЛ 1. ПОРІВНЯННЯ RPCLIB ТА GRPC ФРЕЙМВОРКІВ.

1.1 Огляд наявних RPC c++ фреймворків

На даний момент в інтернет мережі існує велика кількість rpc фреймворків для мови C++. Найбільш відомі з них – grpc(розроблений Google у 2015 році), nanorpc, rpclib, xmlrpc, Apache Thrift. З цього списку для порівняння було обрано grpc та rpclib фреймворки. grpc – дуже популярний фреймворк від компанії Google, а rpclib приваблює простотою, навчальним матеріалом, гарною документацією у коді, швидкістю та багатим функціоналом.

1.2 Установка бібліотек.

Обидві бібліотеки можна завантажити з гітхабу та зібрати з вихідних файлів. Але через те, що grpc – дуже популярна бібліотека, її підтримують багато пакувальних систем, наприклад, пакет grpc можна завантажити за допомогою msys2 або vcpkg. З rpclib усе складніше, її обов'язково треба збирати з вихідних файлів, та зробивши все правильно можна отримати помилку сегментації на тестуванні прикладу. Проблема в тому, що наразі rpclib некоректно збирається з MinGW g++, щоб бібліотека працювала правильно треба при її збірці додати у кінець CMakeLists.txt такий рядок:

```
if(WIN32) target_compile_definitions(${PROJECT_NAME} PUBLIC -DASIO_HAS_THREADS) endif().
```

Це наразі актуальний та ще не виправлений недолік, автору бібліотеки про це відомо.

1.3 Навчальні матеріали до бібліотек

rpclib: для пояснення основ користування цією бібліотекою її розробник створив сайт rpclib.net де зрозумілою простою мовою знайомить користувача з фреймворком та показує прості приклади використання. З цих прикладів можна отримати дуже багато корисної інформації (Рисунок 1.1).

grpc: на сайті grpc.io створено окремий розділ для користувачів цієї бібліотеки на мові C++ (Рисунок 1.2)**.** Там показано як встановити бібліотеку та деякі базові приклади. Написано теж просто, але потребує занадто більше зусиль для розуміння. Також необхідно мати додаткові знання , що також ускладнює розуміння навіть базових речей.

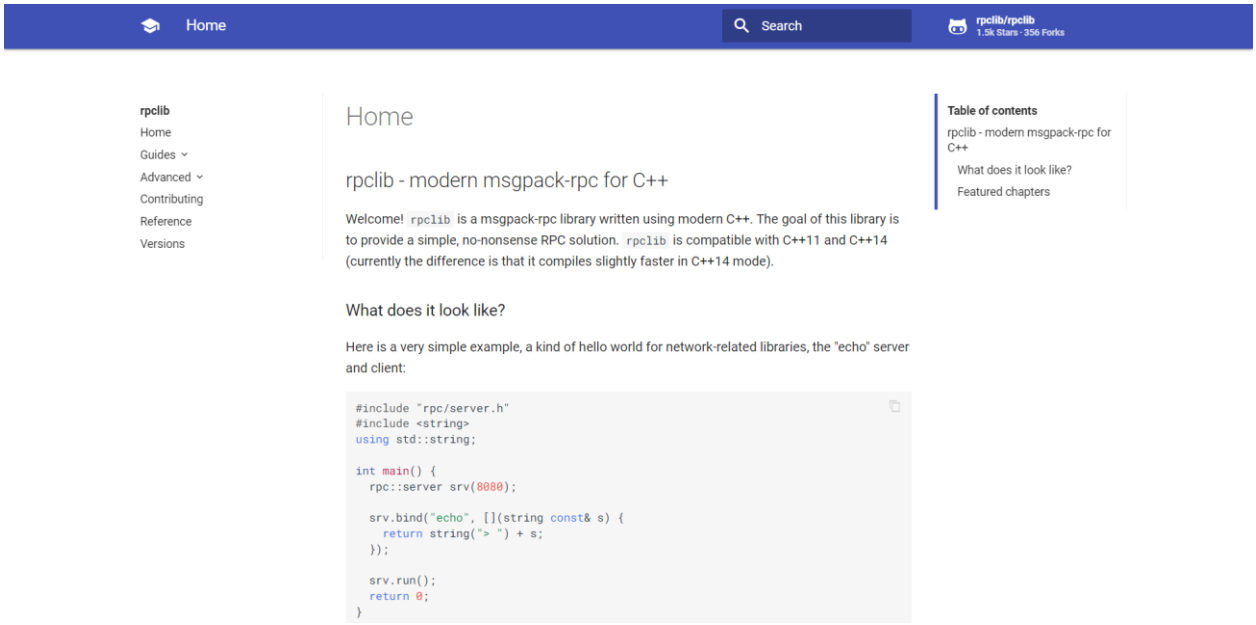

Рисунок 1.1. Домашня сторінка rpclib документації

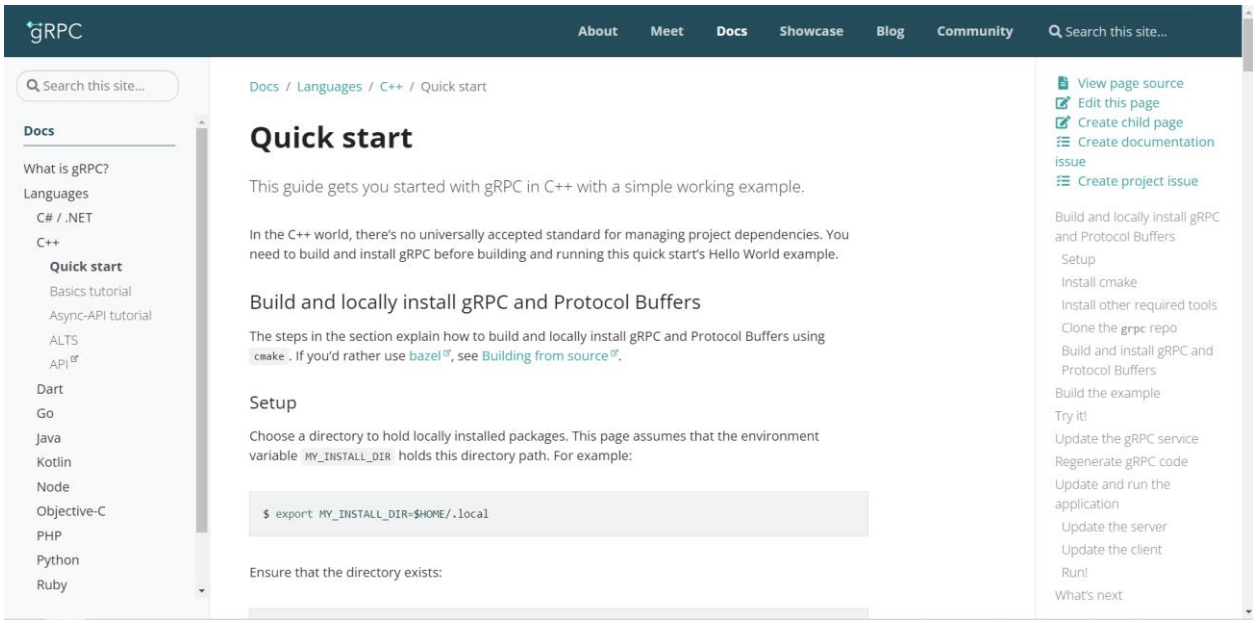

Рисунок 1.2. Сторінка C++ документації grpc фреймворку

1.4 Необхідні знання для використання бібліотек

rpclib: бібліотека дуже проста й інтуїтивна, ніяких специфічних навичок не потребує, щоб почати її використовувати достатньо її просто встановити та протягом 2 хвилин проглянути навчальний матеріал.

grpc: ця бібліотека є більш складною, щоб почати її використовувати потрібно мати знання про протокол серіалізації структурованих даних Protobuf, знати як писати на цій мові файли та як генерувати допоміжні файли для бібліотеки за допомогою Cmake. Це дуже нетривіальна річ, яка потребує деякого часу на опанування. Також необхідно повністю прочитати навчальний матеріал, бо без тих знань зробити щось самостійно неможливо. Тому швидко почати користуватися цією бібліотекою не вийде.

1.5 Складність використання бібліотек

rpclib: як було зазначено раніше, rpclib дуже проста бібліотека, з її допомогою можна швидко написати клієнт та сервер для будь яких цілей. З реалізацією Hello World можна ознайомитися за джерелом[17], директорія rpclib:

Як і зазначалося раніше, усе дуже просто та зрозуміло.

grpc: З цією бібліотекою все відбувається складніше, для кожного можливого запита на клієнті на сервері мають бути реалізовані класи, які в подальшому використовуються для створення запитів. З реалізацією Hello World можна ознайомитися за джерелом [17], директорія grpc/sync:

Як можна побачити, створення навіть Hello world з grpc уже потребує деяких зусиль і навичок. Через те, що grpc має значно більший функціонал ніж rpclib та використовує Protobuf, бачимо, що це приводить до того, що навіть прості програми зроблені за допомогою бібліотеки grpc виглядають складно і зовсім не інтуїтивно, якщо не поглиблюватися в те який функціонал пропонує бібліотека.

Також оглянемо асинхронні реалізації, які можна зробити за допомогою фреймворків. В rpclib для цього замість функцій run() та call() для сервера і клієнта відповідно, зроблено функції async_call() та async_run(). Перша від свого синхронного аналогу відрізняється тим, що вертає std::future, а друга – приймає кількість потоків. А з асинхронною реалізацією того самого Hello World за допомогою grpc можна ознайомитися за джерелом[17], директорія grpc/async.

Як можна побачити, коду стало набагато більше, код незрозумілий, а це тільки на прикладі 1 запита! З’являється багато нетривіальних сутностей, які треба зазначити для того, щоб просто відправляти та отримувати запити асинхронно. З більшою кількістю запитів ситуація буде тільки погіршуватися, у якості прикладу приведено реалізацію за допомогою grpc простого асинхронного калькулятору(джерело[5], директорія calculator, файли з префіксом async).

Маємо майже по 300 строк для кожної реалізації, що здається забагато для такої простої програми.

Дуже простий і зручний у використанні пакет rpclib дозволяє швидко створювати програми на основі RPC. Він простий у вивченні і не потребує якихось специфічних знань чи навичок. Тому використання rpclib не викликає особливих труднощів.

У той же час, незважаючи на широкі можливості gRPC, використання цього API може бути складним. Для кожного типу запиту необхідно розробити окремі класи і використовувати Protobuf для серіалізації даних. Як наслідок, зростає як обсяг коду, так і складність його розуміння. Як наслідок, gRPC складніший за rpclib.

1.6 Продуктивність бібліотек

Оцінка продуктивності бібліотек була взята з репозиторія авторів бібліотеки rpclib(джерело[3]).

Нижче приведено графічну демонстрацію результату бенчмарків:

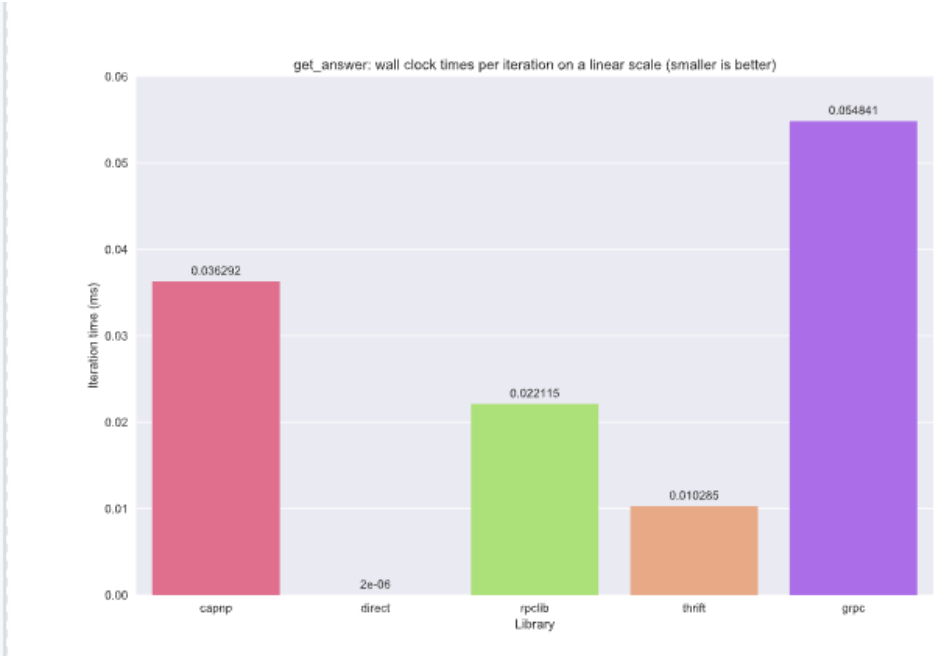

Рисунок 1.3. Порівняння реального часу виконання запиту на функції get_answer

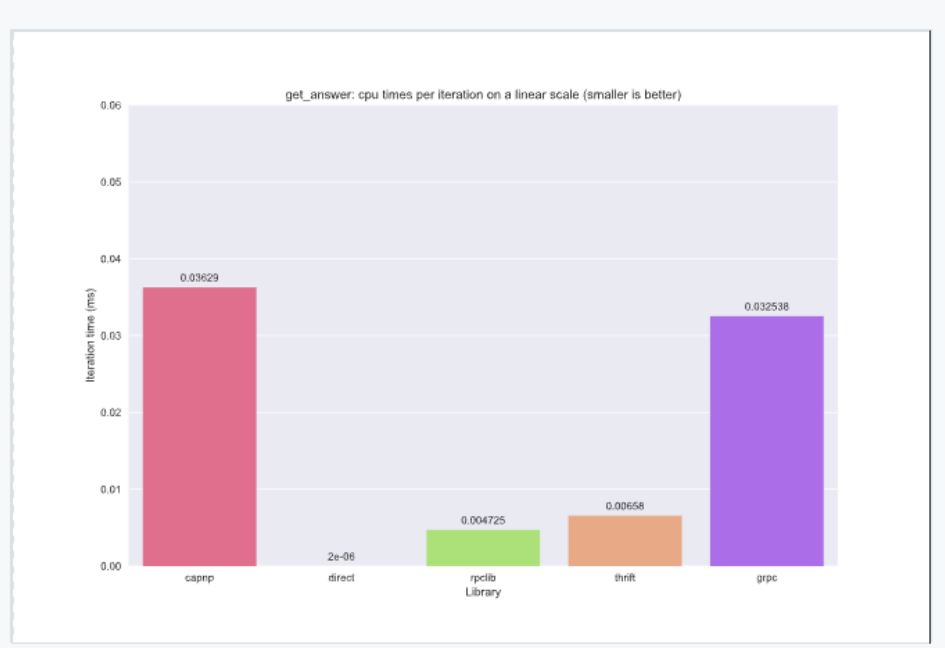

Рисунок 1.4. Порівняння процесорного часу виконання запиту на функції get_answer

Функція get_answer у цьому замірі являю собою функцію, що вертає число та десеріалізує його(число у коді має представлення типу int). З рисунків 1.3 – 1.4 можна побачити, що з виконанням такої простої операції rpclib порається значно скоріше за конкурента. Також можна побачити, що в обох бібліотеках фактичний час на виконання такої операції значно перебільшує процесорний час, при чому в rpclib різниця у цілих 5 разів, коли у grpc – в 1.7 разів.

Функція getStruct являє собою функцію, яка повертає вкладену структуру з великою кількістю полів, великою кількістю даних і навмисно поганим вирівнюванням. З рисунків 1.5 – 1.6 можна побачити, що rpclib виконує цю операцію приблизно у 2 рази швидше за grpc, як зі сторони процесорного часу так і зі сторони реального часу.

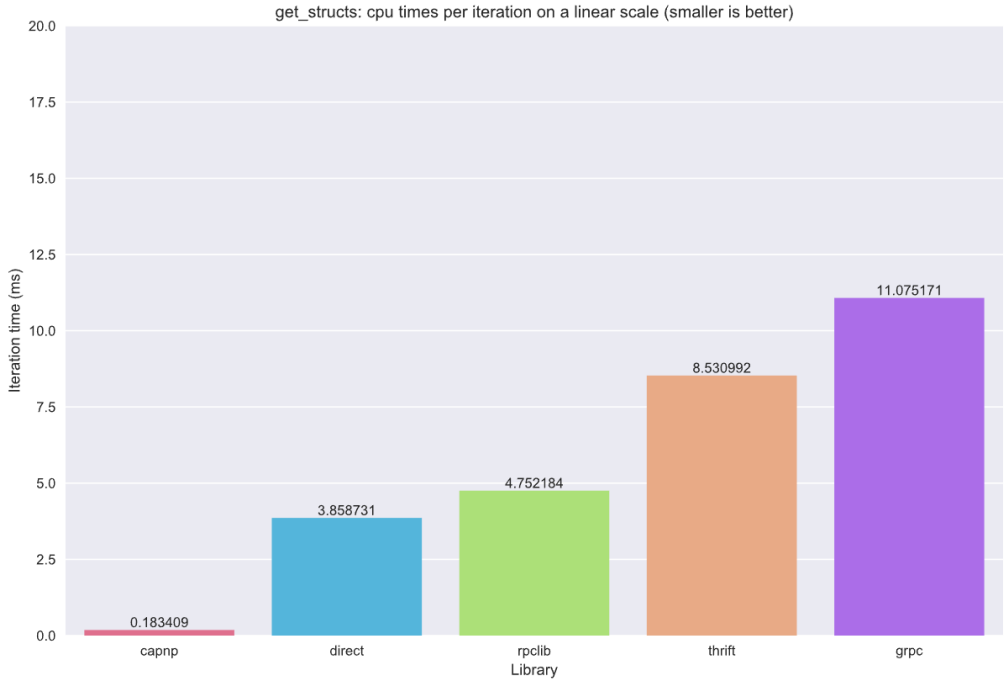

Рисунок 1.5. Порівняння процесорного часу виконання запиту на функції get_struct

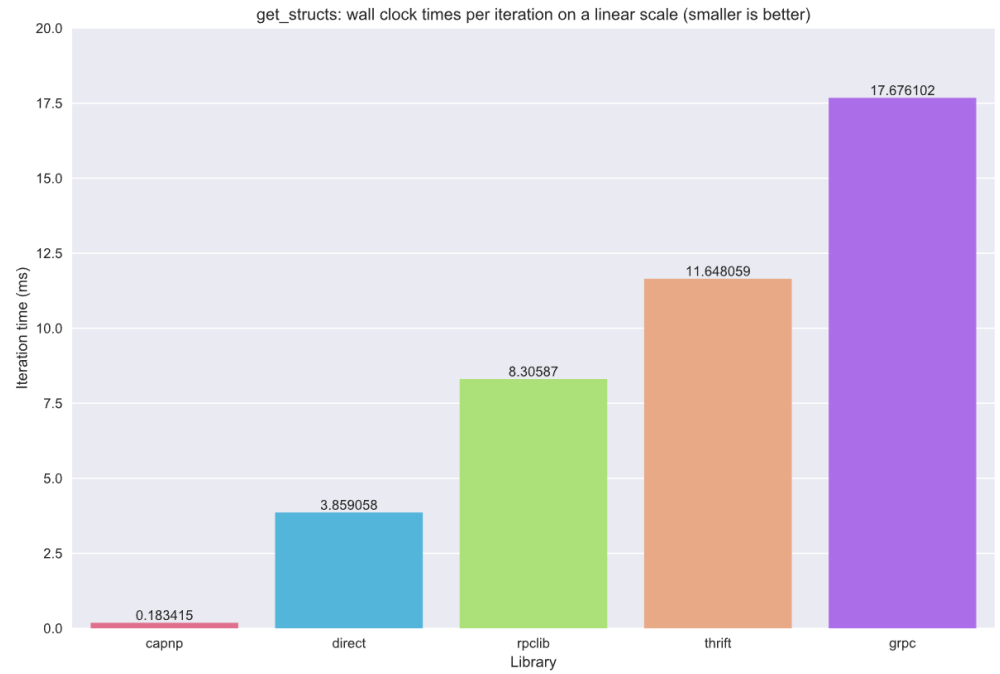

Рисунок 1.6. Порівняння реального часу виконання запиту на функції get_struct

Більш детально з кодом можна ознайомитися в репозиторії, реалізації get_structs, get_answer подано у файлах include/{необхідна бібліотека}/fixture.h.

Швидкість роботи може бути критичною у багатьох проектах, особливо у великих системах з великою кількістю запитів і великим обсягом даних. Таким чином, вибір rpclib може бути вигідним, оскільки вона дозволяє досягти кращої продуктивності і ефективності в обробці запитів.

Проте, варто зауважити, що ефективність і швидкість роботи можуть залежати від конкретних потреб проекту, обсягу даних, мережевої архітектури та інших факторів. Тому важливо провести власне тестування та оцінку продуктивності для конкретного випадку використання.

В загальному, rpclib може бути перевагою, якщо ви шукаєте швидку та ефективну бібліотеку для розробки RPC-серверів та клієнтів, зокрема у випадку, коли швидкість роботи має важливе значення.

1.7 Безпека

gRPC пропонує більше можливостей і засобів захисту, ніж rpclib з точки зору безпеки.

Той факт, що gRPC підтримує автентифікацію та авторизацію, є однією з головних переваг технології з точки зору безпеки. TLS/SSL, JWT (JSON Web Tokens) і OAuth 2.0 - це лише деякі з методів автентифікації, які можна використовувати з gRPC для перевірки ідентичності клієнта і сервера і управління доступом до ресурсів. Це дозволяє забезпечити безпеку даних і захистити комунікацію від несанкціонованого доступу.

Протокол TLS/SSL, який гарантує безпеку та цілісність даних, що надсилаються, також може використовуватися з gRPC для шифрування даних. При роботі з конфіденційними даними або в розподілених системах, де дані можуть передаватися через нестабільні мережі, це надзвичайно важливо.

Крім того, gRPC підтримує обмеження швидкості, що дозволяє регулювати швидкість виконання запитів і уникати перевантаження сервера. Це має вирішальне значення для запобігання зловживань і захисту системи від DDoS-атак.

Однак rpclib не пропонує такого ж рівня інтегрованої підтримки безпеки. Це менш складна і багатофункціональна бібліотека, яка робить акцент на дружності до користувача і швидкому налаштуванні. Через це бібліотека не пропонує ніяких рівнів безпеки.

Отже gRPC пропонує більш досконалий набір методів для захисту комунікацій і даних, якщо безпека є ключовим компонентом вашого проекту, особливо якщо ви працюєте з конфіденційними матеріалами або великими розподіленими системами.

1.8 Ліцензії

Бібліотека rpclib поширюється за ліцензією MIT, яка пропонує широку гнучкість використання. Завдяки ліцензії MIT код бібліотеки rpclib можна використовувати без обмежень як у комерційних, так і у некомерційних програмах. Це надає право використовувати rpclib у будь-яких проектах будь-де і будь яким чином, навіть у приватному програмному забезпеченні або поширюваному програмному забезпеченні.

Для бібліотеки gRPC застосовується ліцензія Apache 2.0. Це надає право використовувати код gRPC під цією ліцензією як для комерційних, так і для некомерційних цілей. Щодо ліцензування зміненого коду існує декілька обмежень. Будь-які зміни до коду, на який поширюється ліцензія Apache 2.0, мають бути зазначені, а відповідна інформація про ліцензування має бути збережена.

Також важливо знати, що ліцензія Apache 2.0 має обмеження на назви проектів та використання торгових марок. Торгові назви, торгові марки, знаки обслуговування або назви продуктів ліцензіара не можуть бути використані, за винятком випадків, коли це абсолютно необхідно для ідентифікації місця походження твору.

1.9 Масштабованість

а) Розмір системи

- rpclib підходить для малих і середніх розподілених систем з обмеженою кількістю клієнтів і серверів.
- gRPC орієнтована для великих розподілених систем з високою масштабованістю. Ця бібліотека має перевагу в широкому масштабі, оскільки може ефективно керувати великою кількістю клієнтів і серверів.

б) Асинхронна обробка

- rpclib має можливість асинхронно обробляти запити та відповіді, що дозволяє серверам обробляти багато запитів одночасно та не чекати на результати запиту зі сторони клієнта.
- grpc також має потужну підтримку асинхронної обробки запитів та відповідей. Більш того, бібліотека дозволяє використовувати потокові сервіси для передачі потокових даних. Потокові запити мож відбуватися як в одному напрямку(unary streaming), так і в обох напрямках(bidirectional streaming).

в) Розподілені транзакції

- Розподілені транзакції не підтримуються бібліотекою rpclib, якщо буде необхідність цей функціонал треба буде реалізувати власноруч.
- grpc підтримує розподілені транзакції. Вони можуть відбуватися за допомогою двофазної фіксації(2PC) або іншими протоколами.

г) Типи запитів

- rpclib підтримує прості типи запитів, а саме простий виклик функції та повернення результату. Це може відбуватися синхронно чи асинхронно.
- grpc підтримує не тільки асинхронний та синхронний виклик функцій та повернення результату, проте ще односторонній та двосторонній потокові запити. Це дозволяє створювати багатоцільові та потокові сервіси, які відповідають вимогам великих розподілених систем.

1.10 Висновок

Враховуючи вищезазначені елементи, робота з rpclib часто буде більш практичною для розгортання та реалізації невеликих проектів. Вона надає простий інтерфейс, який дозволяє швидко налаштувати зв'язок між клієнтом і сервером. Можна почати використовувати бібліотеку одразу, навіть якщо користувач не дуже добре знаєте протокол RPC, оскільки він має менше вимог і його простіше вивчити.

Однак, gRPC пропонує набагато більше корисності та адаптивності. Він більше підходить для складних і розподілених додатків, оскільки підтримує різноманітні запити, включаючи унарні, односторонні та двосторонні потокові запити. Розподілені транзакції, аутентифікація та шифрування даних - це додаткові можливості gRPC, які корисні для побудови складних систем.

У світлі цього, gRPC може бути кращим варіантом, якщо проект потребує більшої гнучкості, розподіленої архітектури, розподілених транзакцій і додаткових можливостей безпеки. Хоча він також вимагає більше зусиль для налаштування і розуміння, варто подумати про те, щоб присвятити час і гроші на освоєння цього інструменту.

Виходячи з проведеного дослідження було вирішено дотримуватись принципів rpclib щодо простоти використання та налаштування та дати можливість установлювати безпечне з’єднання як це робить grpc.

РОЗДІЛ 2. ВИМОГИ ДО СИСТЕМИ ТА ТЕХНОЛОГІЇ РОЗРОБКИ

2.1 Формування вимог

Для забезпечення безпеки було вирішено використовувати OpenSSL, а також дотримуватися принципів гнучкості та зручності використання, які можна порівняти з тими, що надає бібліотека rpclib. Це рішення ґрунтується на ряді речей:

При розробці фреймворку RPC зручність використання є вирішальним фактором. Процес створення та інтеграції фреймворку в проект спрощується завдяки принципам, що дотримуються бібліотекою rpclib, яка надає простий та інтуїтивно зрозумілий інтерфейс. В результаті є можливість зосередитися на функціональності та розвитку фреймворку, а не витрачати час на складні налаштування та вивчення складних інтерфейсів.

Швидкості розгортання сприяє те, що фреймворк менш складний у використанні та налаштуванні. Це особливо корисно, якщо на розробку є короткий проміжок часу або встановлені стислі терміни для завершення роботи. За допомогою підходу rpclib можна швидко створювати прототипи, перевіряти їх ефективність і вносити зміни, як тільки вони будуть потрібні.

Безпека: Підтримка безпеки має вирішальне значення для RPC та розподілених систем. Один з варіантів забезпечення безпеки - використовувати OpenSSL для шифрування та аутентифікації передачі даних між сервером і клієнтом, щоб зробити її більш захищеною. Широко використовувана бібліотека OpenSSL з відкритим вихідним кодом надає фреймворку надійність і безпеку.

Таким чином, було розроблено ефективний та надійний фреймворк RPC, який задовольняє критеріям кваліфікаційної роботи, обравши концепцію простоти використання та модифікації, а також використання OpenSSL для забезпечення безпеки.

2.2 Огляд технологій для розробки бібліотеки

2.2.1 Мова C++

Статично типізована комп'ютерна мова C++ була створена як мова програмування загального призначення. Оскільки вона поєднує в собі риси мов високого та низького рівня, її вважають мовою проміжного рівня. Бйорн Страуструп створив мову в 1979 році в Bell Labs як вдосконалення мови C. Термін "C++" було введено в 1983 році, після того, як спочатку вона була відома як "C з класами".

C++ є однією з найпопулярніших мов програмування з великою кількістю застосувань, включаючи системне програмне забезпечення (наприклад, Microsoft Windows), прикладне програмне забезпечення, драйвери пристроїв, вбудоване програмне забезпечення, високопродуктивні серверні та клієнтські додатки, а також розважальні програми, такі як відеоігри.

C++ також використовується для апаратного забезпечення, де дизайн спочатку описується на C++, а потім архітектурно обмежується і планується створити рівень передачі опису апаратури мовою з використанням високорівневого синтезу.

Покращення C++ розпочалося з додавання класів, віртуальних функцій, перевантаження операторів, багатократного успадкування, шаблонів та обробки винятків серед інших функцій. Після кількох років розробки стандарт мови програмування C++ був ратифікований у 1998 році як стандарт ISO/IEC 14882:1998.

Наразі підтримується 7 стандартів мови: C++98, C++03, C++11, C++14, C++17, C++20 та C++23.

2.2.2 Boost

Boost - зібрання бібліотек класів, що використовують функціональність мови C++ і надають зручний кросплатформовий високорівневий інтерфейс для лаконічного кодування різноманітних повсякденних підзадач програмування (робота з даними, алгоритмами, файлами, потоками, регулярними виразами, лінійна алгебра, генерація псевдовипадкових чисел, обробка зображень, модульне тестування і т. ін.). Версія 1.82 містить 164 окремі бібліотеки.

Вільно поширюються за ліцензією Boost Software License, розробленою для того, щоб Boost можна було використовувати як із вільними, так і з пропріє тарними програмними проектами, разом із вихідним кодом. Проект було створено після ухвалення стандарту C++, коли багато хто був незадоволений відсутністю деяких бібліотек у STL. Проект є своєрідним "випробувальним полігоном" для різних розширень мови і частини бібліотек, які є кандидатами на включення в наступний стандарт C++. Багато хто із засновників Boost входять до комітету зі стандартизації C++, і кілька бібліотек Boost було прийнято для включення до C++ Technical Report 1, стандарту C++11 (наприклад, інтелектуальні покажчики, потоки, регулярні вирази, random, ratio, tuple) і стандарту C++17 (наприклад, filesystem, any, optional, variant, string_view).

Спільнота Boost з'явилася приблизно 1998 року, коли було випущено першу версію стандарту. Відтоді вона постійно зростала і тепер відіграє важливу роль у стандартизації C++. Незважаючи на те що між спільнотою Boost і комітетом зі стандартизації немає офіційних відносин, деякі розробники беруть активну участь в обох групах.

Boost має помітну спрямованість на дослідження і розширюваність (метапрограмування і узагальнене програмування з активним використанням шаблонів). Завдяки ретельному підбору і контролю якості бібліотеки, включені в Boost, мають високу надійність і продуктивність. Думки щодо використання різняться. Деякі вважають його стандартом де-факто і необхідним доповненням до STL. Деякі, навпаки, уникають всякого

використання бібліотеки в проектах, оскільки це зайва залежність в проекті і для використання цих бібліотек програмістові необхідно добре знати C++, оскільки деякі частини Boost вимагають досить хорошої підготовки програміста і є вельми складними.

У розроблюваній бібліотеці використовується Boost.Asio, кросплатформова C++ бібліотека для мережевого та низькорівневого програмування вводу/виводу, яка надає розробникам послідовну асинхронну модель з використанням сучасного підходу C++.

2.2.3 Visual Studio

Visual Studio - це незамінне середовище розробки для створення бібліотек на мові програмування C++. Завдяки розширеним можливостям та інтуїтивному інтерфейсу, Visual Studio допомагає розробникам ефективно створювати, налагоджувати та тестувати бібліотеки.

Visual Studio підтримує останні стандарти мови, підтримує технологію IntelliSense, яка дозволяє більш швидко і зручно писати код, має налагоджувач для більш зручного пошуку помилок, дозволяє налаштовувати залежності та особливості компіляції. Усе це робить Visual Studio ідеальним вибором для розробки бібліотек на мові програмування C++.

2.2.4 Sublime Merge

Sublime Merge - це графічний клієнт для систем контролю версій, зокрема Git, який може використовуватися під час розробки бібліотек. Його можна використовувати як самостійний клієнт або як розширення до текстового редактора Sublime Text.

Використання Sublime Merge під час розробки дозволяє розробникам зосередитися на важливих аспектах коду та його версіонуванні. Його простий та потужний інтерфейс сприяє продуктивності та забезпечує зручне середовище для роботи з Git, що є необхідним для ефективної розробки.

2.2.5 Nlohmann/json

nlohmann/json є бібліотекою у мові C++, яка надає потужні інструменти для роботи з форматом обміну даними JSON. Ця бібліотека використовується під час розробки бібліотек для обробки та розбору даних JSON.

Основні особливості та переваги nlohmann/json:

- Простота використання: nlohmann/json має дружній та інтуїтивно зрозумілий інтерфейс, що робить його легким у використанні для розробників. Він пропонує прості методи та функції для зчитування, запису та маніпулювання даними JSON.
- Підтримка стандарту JSON: бібліотека nlohmann/json повністю відповідає стандарту JSON і забезпечує повну функціональність для роботи з JSON-документами. Вона підтримує різні типи даних JSON, включаючи об'єкти, масиви, числа, рядки, булеві значення та значення null.
- Гнучкість та розширені можливості: nlohmann/json надає розробникам гнучкість у роботі з JSON-даними. Вона дозволяє серіалізувати об'єкти C++ у формат JSON та десеріалізувати JSON-дані у відповідні об'єкти C++. Крім того, вона підтримує різні опції серіалізації та десеріалізації, такі як приховування певних полів, обробка дат, керування пробілами та інші.
- Висока продуктивність: nlohmann/json працює швидко та ефективно, що є важливим аспектом для розробки бібліотек. Вона використовує оптимізовані алгоритми та структури даних, що дозволяють швидко обробляти великі обсяги JSON-даних без значного зниження продуктивності.
- Кросплатформеність: nlohmann/json підтримує роботу на різних платформах, включаючи Windows, macOS та різні дистрибутиви Linux. Це дозволяє розробникам використовувати цю бібліотеку у проектах, незалежно від платформи розгортання.

Бібліотека nlohmann/json є популярним інструментом для розробників у сфері обробки та аналізу даних JSON. Вона надає простість використання, гнучкість та продуктивність, що робить її відмінним вибором для розробки бібліотек, що взаємодіють з JSON-даними.

2.2.6 TLS/SSL

TLS (Transport Layer Security) і SSL (Secure Sockets Layer) - це криптографічні протоколи, що забезпечують безпеку передачі даних через мережу. Вони широко використовуються в розробці бібліотек для забезпечення захищеного з'єднання між клієнтом та сервером.

Основні особливості і переваги TLS/SSL:

- Конфіденційність даних: TLS/SSL застосовує механізми шифрування для захисту переданих даних від прослуховування та перехоплення. Використовуючи симетричне або асиметричне шифрування, вони забезпечують конфіденційність даних, що передаються між клієнтом та сервером.
- Аутентифікація: TLS/SSL надають механізми аутентифікації, що дозволяють перевірити, що комунікуючі сторони є дійсними ідентифікованими суб'єктами. Це дозволяє запобігти атакам на маніпулювання та підробку ідентичності.
- Цілісність даних: TLS/SSL забезпечують механізми контролю цілісності даних, що гарантує, що дані не були змінені під час передачі. Використовуючи хеш-функції або цифрові підписи, вони перевіряють цілісність даних і сприяють виявленню будь-яких змін або порушень.
- Довірені сертифікати: TLS/SSL використовують сертифікати для підтвердження довіри до сервера. Ці сертифікати видані довіреними органами, що гарантується аутентичність сервера і запобігає атакам "людина посередині".

- Кросплатформеність: TLS/SSL підтримуються на різних платформах, що дозволяє використовувати їх у різних середовищах розробки. Це забезпечує широку сумісність і доступність для розробників.

У розробці бібліотек TLS/SSL використовується для створення захищених з'єднань між клієнтом та сервером, забезпечення конфіденційності, аутентифікації та цілісності даних. Використання TLS/SSL в бібліотеці допомагає забезпечити безпеку комунікації та захист від різних типів атак, що можуть загрожувати передачі даних через мережу.

2.2.7 OpenSSL

OpenSSL - це відкрите програмне забезпечення, яке надає реалізацію протоколів криптографії, таких як TLS (Transport Layer Security) і SSL (Secure Sockets Layer). Ця бібліотека надає розширені можливості забезпечення безпеки, які можуть бути використані в розробці різноманітних програм і бібліотек.

Деякі особливості і переваги використання OpenSSL для розробки бібліотек:

- Криптографічні протоколи: OpenSSL надає реалізацію широкого спектру криптографічних протоколів, включаючи TLS, SSL, асиметричне шифрування, хеш-функції, цифрові підписи та багато інших. Це дозволяє розробникам використовувати потужні технології безпеки в розроблюваних проектах.
- Підтримка різних платформ: OpenSSL є кросплатформеною бібліотекою, яка працює на багатьох операційних системах, включаючи Windows, Linux, macOS і більшість UNIX-подібних систем. Це дозволяє розробникам використовувати одну і ту ж криптографічну бібліотеку на різних платформах.
- Багатофункціональність: OpenSSL надає широкий набір функцій і інструментів, що дозволяють виконувати різноманітні криптографічні

операції, обробку сертифікатів, керування ключами і багато іншого. Це дозволяє розробникам зручно працювати з криптографією і забезпечувати потрібну рівень безпеки у розроблюваних проектах.

- Зрілість і надійність: OpenSSL існує протягом багатьох років і має широку спільноту розробників, яка підтримує його розвиток і вдосконалення. Ця бібліотека пройшла значну кількість тестувань і має велику кількість вирішених проблем безпеки. Використання такої відомої і перевіреної бібліотеки може сприяти підвищенню безпеки програмного продукту.

Використання OpenSSL в розробці бібліотек дозволяє розробникам забезпечити захист передачі даних, аутентифікацію та забезпечення конфіденційності. Крім того, ця бібліотека дозволяє розробникам використовувати різноманітні криптографічні функції і протоколи для реалізації різних сценаріїв безпеки в розроблюваних проектах.

2.3 Технічні вимоги

- Функціональні вимоги:
  1. Бібліотека повинна надавати механізм виклику функцій на віддаленому сервері з використанням мережевого протоколу.
  2. Базові типи, такі як int, double, unsigned, string, bool, повинні підтримуватись як параметри та повернені значення функцій.
  3. Бібліотека повинна вміти працювати як з незахищеними каналами зв’язку, так і з захищеними за допомогою TLS/SSL технології.
- Простота використання:
  1. Бібліотека повинна легко встановлюватись та використовуватись.
  2. Розробка клієнтських і серверних додатків повинна бути простою і інтуїтивно зрозумілою.
  3. Необхідно забезпечити документацію, що пояснює процес встановлення, налаштування та використання бібліотеки.

4. Наявність прикладів коду та документації, що пояснюють, як використовувати різні функціональності бібліотеки.

- Надійність та безпека:
  1. Бібліотека повинна бути стійкою до помилок і забезпечувати коректну обробку виняткових ситуацій та помилок під час виклику функцій.
  2. Необхідно забезпечити захист передачі даних, використовуючи шифрування та інші механізми безпеки, такі як TLS/SSL.
  3. Бібліотека повинна бути стійкою до атак і забезпечувати конфіденційність та цілісність даних.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ І ДОКУМЕНТАЦІЯ

3.1 Серіалізація/Десеріалізація

Щоб сервер та клієнт могли обмінюватися даними їм потрібно вміти перетворювати дані у формат, який може передаватися за допомогою інструментів комунікації між різними компонентами системи. Перетворення даних у такий формат та навпаки і називається серіалізацію та десеріалізацією відповідно. Для цих цілей було використано формат json та бібліотеку nlohmann/json. У якості об’єкта, що серіалізується – клас Message(Рисунок 3.1), у якого є 2 поля – назва методу та набір параметрів, що приймає метод. На клієнті ж поле методу інтерпретується як статус виконання запиту та набір параметрів як повернене значення з серверу.

```
struct Message {
    std::string method;
    std::vector<std::any> parameters;

    static Message deserialize(const std::string& json);
    std::string serialize() const;
};
```

Рисунок 3.1. Реалізація класу Message

Вивід типів у функціях десеріалізації (Рисунок 3.2) та серіалізації(Рисунок 3.3) реалізовані за допомогою вбудованих інструментів мови та бібліотеки для ідентифікації типів.

```
for (const auto& param : paramsArray) {
    if (param.is_null()) {
        params.push_back(std::any(nullptr));
    }
    else if (param.is_boolean()) {
        params.push_back(std::any(param.get<bool>()));
    }
    else if (param.is_number_integer()) {
        params.push_back(std::any(param.get<int>()));
    }
    else if (param.is_number_unsigned()) {
        params.push_back(std::any(param.get<unsigned int>()));
    }
    else if (param.is_number_float()) {
        params.push_back(std::any(param.get<double>()));
    }
    else if (param.is_string()) {
        params.push_back(std::any(param.get<std::string>()));
    }
    else {
        throw std::runtime_error("Unsupported data type in JSON");
    }
}
```

Рисунок 3.2. Формування набору параметрів при зчитуванні json

```
nlohmann::json paramsArray;
for (const auto& param : parameters) {
    if (param.has_value()) {
        const std::type_info& paramType = param.type();
        if (paramType == typeid(nullptr)) {
            paramsArray.push_back(nullptr);
        }
        else if (paramType == typeid(bool)) {
            paramsArray.push_back(std::any_cast<bool>(param));
        }
        else if (paramType == typeid(int)) {
            paramsArray.push_back(std::any_cast<int>(param));
        }
        else if (paramType == typeid(unsigned int)) {
            paramsArray.push_back(std::any_cast<unsigned int>(param));
        }
        else if (paramType == typeid(double)) {
            paramsArray.push_back(std::any_cast<double>(param));
        }
        else if (paramType == typeid(std::string)) {
            paramsArray.push_back(std::any_cast<std::string>(param));
        }
        else {
            throw std::runtime_error("Unsupported data type in params");
        }
    }
    else {
        paramsArray.push_back(nullptr);
    }
}
```

Рисунок 3.3. Запис параметрів у json.

3.2 Збереження функції на сервері

Щоб зберігати функції, що визначили користувачі, на сервері та викликати їх по запиту з клієнта був обраний тип std::function<std::any(std::vector<std::any>&&>) . Зберігання таких функцій робиться за допомогою функції addFunction(Рисунок 3.4).

```
template<typename Callable>
inline void Server::addFunction(std::string name, Callable i_function)
```

Рисунок 3.4. Вигляд методу, що зберігає функції за ключем у пам’яті

Як працює сутність, яку створює і зберегає addFunction(сутність типу std::function<std::any(std::vector<std::any>&&>)):

1. Приймає вектор елементів std::any.
2. Конвертує отримані дані у структуру std::tuple<Args…>, де Args… - типи, що приймає користувацька функція.
3. За допомогою std::apply() викликає користувацьку функцію від переданих параметрів.

Щоб реалізувати логіку кроку 2 було використано структуру func_traits(Рисунок 3.5), яка являє собою ознаку типу викликаємого об’єкта:

```
template <typename T>
struct func_traits : func_traits<decltype(&T::operator())> {};

template <typename C, typename R, typename... Args>
struct func_traits<R(C::*)(Args...)> : func_traits<R(*)(Args...)> {};

template <typename C, typename R, typename... Args>
struct func_traits<R(C::*)(Args...) const> : func_traits<R(*)(Args...)> {};

template <typename R, typename... Args> struct func_traits<R(*)(Args...)> {
    using result_type = R;
    static constexpr std::size_t input_size = sizeof...(Args);

    template <std::size_t ... I>
    static std::tuple<Args...> getArgsFromVector(std::vector<std::any> input, std::index_sequence<I...>)
    {
        return std::make_tuple(std::any_cast<Args>(input[I])...);
    }

    static std::tuple<Args...> getArgsFromVector(std::vector<std::any> input)
    {
        return getArgsFromVector(input, std::make_index_sequence<sizeof...(Args)>{});
    }
};
```

Рисунок 3.5. Мета структура для роботи з функціями.

За допомогою цієї структури, передаючи лише тип викликаємого об’єкта є можливість дізнатися тип вхідних, вихідних даних та навіть конвертувати вектор від std::any у кортеж з типами вхідних даних викликаємого об’єкта. Після цього реалізувати функцію addFunction уже є тривіальною задачею.

3.3 Обробка запиту на сервері

Функції, що додає користувач на сервер, зберігаються у словнику, який за ключ має тип std::string, який символізує за яким іменем клієнт може звернутися до цієї функції, та, як було сказано раніше, значення типу std::function<std::any<std::vector<std::any>&&>. Щоб обробити запит, що прийшов з сервера спочатку він десеріалізується, як було описано у темі серіалізації, а потім передається до функції processRequest(Рисунок 3.6).

```
std::string json_data(received_data.begin(), received_data.end());
Message request = Message::deserialize(json_data);

// Process the request and get the response
Message response = processRequest(request);
```

Рисунок 3.6. Формування повної строки запиту, десеріалізація та використання processRequst.

Функція processRequst повертає помилку, якщо функції с отриманим ключом у неї не визначено, якщо клієнт передав невірну кількість аргументів функції або аргументи невірного типу та якщо у процесі виконання функції було викинуте виключення. У інших випадках повертається статус “success” та результат виконання функції, якщо він є, та nullptr, якщо функція мала тип void.

3.4 Ініціалізація сервера

Як було зазначено, для мережевого спілкування було обрано бібліотеку Boost.Asio. Це бібліотека, яка складається тільки з файлів заголовків, тому вона не потребує встановлення ніяких додаткових залежностей.

Щоб налагодити сервер для використання спочатку потрібно створити об’єкт io_context. Цей елемент - центральним в асинхронному програмуванні з використанням Boost.Asio. Він відповідає за управління потоками виконання та планування асинхронних операцій вводу-виводу. Незважаючи на те, що сервер працює синхронно, ця сутність усе одно має бути створена для використання подальшого функціоналу Boost.Asio. Це також залишає можливість для подальшої асинхронної реалізації сервера.

Наступним створюється елемент boost::asioo::ip::tcp::endpoint. Цей елемент відповідає за те яка версія інтернет протоколу буде використовуватись, яка буде адреса домену та який порт. У реалізації бібліотеки цей об’єкт виглядає таким чином(Рисунок 3.7)

```
tcp::endpoint(tcp::v4(), m_port)
```

Рисунок 3.7. Об’єкт, що визначає адресу сервера.

Це означає, що використовується 4 версія інтернет протоколу, сервер розгорнений на localhost з портом, указаним користувачем.

Наступним іде створення boost::asio::ip::tcp::acceptor. Цей об’єкт надає можливість програмі слухати на певному порті і адресі, чекаючи на з’єднання від клієнтів. Коли з’єднання надходить цей об’єкт створює новий сокет для цього з’єднання та повертає його серверу.

На цьому етапі після створення ip::tcp::acceptor сервер уже готовий приймати з’єднання.

3.5 Ініціалізація клієнта

Для ініціалізації клієнта, по аналогії з сервером, так само має бути створений boost::asio::io_context.

Далі, для створення нового з’єднання знадобиться boost::asio::ip::tcp::resolver. Ця сутність дозволяє з імені хоста отримати потрібну ip адресу для підключення. Окрім цих сутностей сервер ще має створити гніздо для комунікації з сервером, що буде показано у наступному розділі.

3.6 TLS/SSL захист

Сервер

Для використання TLS/SSL технології для захищеного з’єднання було обрано використовувати частину бібліотеки Boost.Asio, а саме сутності, що визначені у просторі імен boost::asio::ssl. У boost зроблено обгортку над бібліотекою OpenSSL, яка забезпечує зручне користування усіма перевагами цієї бібліотеки. Зокрема, потрібно зробити передачу даних захищеною від хакерських атак.

Через те, що сервер має вміти працювати у захищеному та незахищеному режимі, у конструкторі об’єкту сервера, окрім порту, додано ще структуру ServerSSLData(Рисунок 3.8), яка містить у собі шлях до TLS/SSL сертифікатів. Цей аргумент не є обов’язковим і сервер запускається у незахищеному режимі якщо ця структура не була передана та у захищеному якщо сервер все ж таки отримав шляхи до потрібних сертифікатів.

```
struct ServerSSLData
{
    std::string certificate;
    std::string privateKey;

    bool empty() const noexcept {
        return certificate.empty() && privateKey.empty();
    }
};
```

Рисунок 3.8. Структура, що визначає шляхи до сертифікатів сервера.

Використовуючи цю інформацію фреймворк уже вирішую чи створювати boost::asio::ssl:context та яке використовувати гніздо для зв’язку з клієнтом.

Якщо сервер отримав сертифікати, створюється об’єкт boost::asio::ssl::context та у конструктор цього елемента передається аргумент boost::asio::ssl::context::sslv23. Це встановлює контекст SSL/TLS для використання протоколу SSLv3 або більш пізніх версій, таких як TLSv1.0, TLSv1.1 та TLSv1.2.

Далі викликаються функції, що налагоджують цей ssl context(Рисунок 3.9).

```
sslContext->use_certificate_file(m_sslData.certificate, ssl_context::pem);
sslContext->use_private_key_file(m_sslData.privateKey, ssl_context::pem);
sslContext->set_options(ssl_context::default_workarounds | ssl_context::no_sslv2 | ssl_context::single_dh_use);
```

Рисунок 3.9. Налаштування ssl контексту

Це вказує контексту які саме сертифікати слід використовувати та який формати вона мають. Через те, що pem – найпоширеніший формат SLL сертифікатів, його і використано у розробленому фреймворку. Далі встановлюю параметри:

1. **ssl_context::default_workarounds**: Ця опція включає різні робочі засоби, які можуть бути потрібні для вирішення різних проблем або неповних реалізацій SSL/TLS в певних реалізаціях бібліотеки. Включення цієї опції гарантує більшу сумісність і надійність в різних середовищах.

2. **ssl_context::no_sslv2**: Ця опція вимикає підтримку протоколу SSLv2. SSLv2 є застарілим і небезпечним протоколом, який вважається незахищеним. Вимкнення його встановлює більш сучасні та безпечні протоколи, такі як SSLv3 або TLS.
3. **ssl_context::single_dh_use**: Ця опція вказує на використання окремого набору параметрів Diffie-Hellman (DH) для кожного з'єднання. DH є протоколом для безпечного обміну ключами і використовується для забезпечення конфіденційності та цілісності даних. Використання окремих наборів параметрів DH для кожного з'єднання підвищує безпеку, оскільки унеможливлює можливість підбору ключа через повторне використання параметрів DH.

Надалі цей контекст буде використовуватися для створення захищеного гнізда для підключення.

Клієнт

На клієнті все відбувається по такому самому принципу, якщо отримується путь до сертифікату –створюється ssl_context, який надалі використовується у створенні захищеного гнізда та безпечного підключення(Рисунок 2.10).

```
if (!pathToSertificate.empty()) {
    sslContext = ssl_context{ ssl_context::tlsv12_client };
    sslContext->set_verify_mode(boost::asio::ssl::verify_peer);
    sslContext->load_verify_file(pathToSertificate);
}
```

Рисунок 3.10. Створення ssl контексту на клієнті.

Контекст створюється з параметром ssl_context::tlsv12_client. Цей аргумент означає що ssl_context буде використовувати TLSv1.2 для клієнтського з'єднання.

Після цього виклик функції set_verify_mode встановлює режим перевірки, де клієнт вимагає перевірки сертифіката сервера. Це означає, що

перед встановленням з'єднання клієнт буде перевіряти сертифікат сервера, щоб переконатися, що він дійсний і виданий довіреною стороною.

Далі ми задаємо файл, який буде використовуватися для верифікації та після цього налагодження клієнта завершене.

3.7 Комунікація

Сервер

Для того, щоб встановити з’єднання, окрім того, що потрібен раніше описаний acceptor також треба створити або звичайне гніздо (об’єкт типу boost::asio::ip::tcp::socket) або захищене гніздо (об’єкт типу boost::asio::stream<boost::asio::ip::tcp::socket>). Яке гніздо створювати вирішується на основі того чи був створений та налагоджений boost::asio::ssl::context об’єкт для безпечного підключення. Підключення відбувається за допомогою acceptor’а, який повертає гніздо з установленим з’єднанням(Рисунок 3.11).

```
std::optional<tcp_socket> tcpSocket;
std::optional<ssl_socket> sslSocket;
if (sslContext.has_value()) {
    sslSocket = ssl_socket{ ioContext, *sslContext };

    acceptor.accept(sslSocket->lowest_layer());
    sslSocket->handshake(asio::ssl::stream_base::server);
}
else {
    tcpSocket = tcp_socket{ ioContext };
    acceptor.accept(*tcpSocket);
}
```

Рисунок 3.11. Установлення з’єднання з клієнтом

Можна побачити, що на відміну від звичайного гнізда, гніздо с ssl захистом викликає ще функцію handshake(). Саме у цій функції відбувається аутентифікація, а саме перевірка сертифікатів, обмін параметрами шифрування, наприклад, симетричним ключем за допомогою якого буде відбуватися шифрування та дешифрування даних. Після виходу з цієї функції

між сервером та клієнтом уже встановлений безпечний канал і сервер готовий приймати і обробляти захищені повідомлення від клієнта.

Читання данних з гнізда проходить поступово, щоб забеспечити можливість передавати велику кількість даних читання з гнізда відбувається у циклі, за кожну ітерацію считується 1024 байта інформації та заповнюється вже динамічно розширюваний буфер отриманою інформацією. Коли досягається кінець повідомлення – цикл переривається.

Далі отримане повідомлення десеріалізується, обробляється, серіалізується для отправки та, нарешті відправляється клієнту(Рисунок 3.12). Після усіх описаних дій сокет закривається та знищується.

Якщо під час усього описаного процесу виникає помилка – сервер виводить цю помилку у термінал та починає чекати наступного повідомлення.

```
std::string json_data(received_data.begin(), received_data.end());
Message request = Message::deserialize(json_data);

// Process the request and get the response
Message response = processRequest(request);

// Serialize the response
std::string json_response = response.serialize();

// Send the response back to the client
auto writeToSocket = [&](auto& socket)
{
    asio::write(socket, asio::buffer(json_response));
};

sslSocket.has_value() ? writeToSocket(*sslSocket) : writeToSocket(*tcpSocket);

if (sslSocket.has_value()) {
    sslSocket->lowest_layer().close();
}
else {
    tcpSocket->close();
}
```

Рисунок 3.12. Десеріалізація отриманих даних, обробка, відправка відповіді на сервер та коректе знищення сокета.

Клієнт

Для створення нового з’єднання на клієнті створюється нове гніздо, та використовується раніше визначений ip адрес та порт домену, що у

розробленій бібліотеці представляє собою змінна endpoints. Далі за допомогою функції boost::connect відбувається підключення до сервера та у випадку захищеного з'єднання ще викликається handshake, яка працює за аналогією з сервером(Рисунок 3.13).

Далі клієнт може відправити повідомлення, це робиться за допомогою передачі об'єкту Message у функцію sendMessage. Ця функція серіалізує повідомлення та відправляє його на сервер. Далі після відправки повідомлення клієнт отримує відповідь з сервера, десеріалізує її та повертає користувачу. Читання повідомлення працює за таким самим принципом як і на сервері.

```
void Client::connect()
{
    if (m_pImpl->sslContext.has_value()) {

        m_pImpl->sslSocket = Impl::ssl_socket{ m_pImpl->ioContext, *m_pImpl->sslContext };

        boost::asio::connect(m_pImpl->sslSocket->lowest_layer(), m_pImpl->endpoints);

        try {
        // Perform SSL handshake
        m_pImpl->sslSocket->handshake(boost::asio::ssl::stream_base::client);
        }
        catch (...) {
            throw std::runtime_error("Certificate is not accepted");
        }
    }
    else {
        m_pImpl->socket = Impl::tcp_socket{ m_pImpl->ioContext };
        boost::asio::connect(*m_pImpl->socket, m_pImpl->endpoints);
    }
}
```

Рисунок 3.13. Установлення з'єднання з сервером

3.8 Налаштування

Щоб використовувати описаний фреймворк для початку потрібно скопіювати його репозиторій. Бібліотеки boost та json уже знаходяться у репозиторії, проте бібліотеку openssl треба встановити власноруч. Щоб використовувати бібліотеку треба у корінь репозиторія фреймворку скопіювати бібліотеку https://github.com/openssl/openssl, та зібрати цю бібліотеку. Після цього можете використовувати CMakeLists.txt у корені

проекту для того, щоб зібрати бібліотеку. Для цього треба прописати у команду строку такі команди:

mkdir build

cd build

cmake ..

cmake –build

cmake –install

Після цього у папці output можна буде знайти .lib файли які і треба використовувати у Вашому проекті. Файли заголовків знаходяться у папці include.

3.9 Використання

Приклад простої програми з використанням бібліотеки наведено на Рисунках 3.14 – 3.15.

```cpp
#include <iostream>
#include "Client.h"

int main()
{
    Client client("localhost", 8080, "good.pem");

    auto response = client.sendMessage({ .method = "add", .parameters = {4, 5} });
    std::cout << std::any_cast<int>(response.parameters.front()) << std::endl;

    std::cin.ignore(1);
    return 0;
}
```

Рисунок 3.14. Приклад клієнта

```cpp
#include "Server.h"
#include <iostream>


int main()
{
    Server server{ 8080, ServerSSLData{.certificate = "server.crt", .privateKey = "server.key"} };

    server.addFunction("add", [](int a, int b) { return a + b; });
    server.start();

    return 0;
}
```

Рисунок 3.15. Приклад сервера

ВИСНОВОК

У результаті порівняні фреймворки rpclib та grpc, виявлено їх слабкі та сильні сторони. Проаналізовані такі критерії як складність налаштування, складність використання, швидкість роботи, безпека з’єднання, ліцензування та масштабованість. На основі дослідження сформовано вимоги до власного фреймворку. За основні критерії взяті безпечність з’єднання та простота використання. На основі цього обрано технології та інструменти для розробки фреймворку, сформовано технічні вимоги. З усіма отриманими даними розроблено RPC фреймворк, що задовольняє поставленим задачам.

Плани для подальшої розробки:

- Підтримка асинхронного виконання.
- Підтримка користувацьких типів.
- Підтримка різних транспортних протоколів.
- Скріпт для створення сертифікатів.

Використані джерела

1. Сайт з документацією по використанню бібліотеки rpclib [Електронний ресурс]. – Режим доступу до ресурсу: rpclib.net
2. Репозиторій rpclib [Електронний ресурс]. – Режим доступу до ресурсу https://github.com/rpclib/rpclib
3. Репозиторій з порівнянням швидкості бібліотеки rpclib та її конкурентів [Електронний ресурс]. – Режим доступу до ресурсу: https://github.com/rpclib/benchmarks
4. Документація використання grpc з C[Електронний ресурс] – Режим доступу до ресурсу: https://grpc.io/docs/languages/cpp/
5. Приклади використання бібліотеки grpc [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/chungphb/grpc-cpp
6. Документація cmake [Електронний ресурс] – Режим доступу до ресурсу: https://cmake.org/documentation/
7. Бібліотека, використана для роботи з json [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/nlohmann/json
8. Bjarne Stroustrup The C++ Programming Language (4th edition), 2013 – 1368 p.
9. Документація до використаної json бібліотеки [Електронний ресурс] – Режим доступу до ресурсу: https://json.nlohmann.me/
10. Документація використаної бібліотеки boost [Електронний ресурс] – Режим доступу до ресурсу: https://www.boost.org/
11. Репозиторій OpenSSL [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/openssl/openssl
12. Історія бібліотеки boost [Електронний ресурс] – Режим доступу до ресурсу: https://docs.cppalliance.org/user-guide/boost-history.html
13. Dmytro Radchuk Boost.Asio C++ Network Programming Cookbook, 2016 – 248 p.
14. Історія мови C++ [Електронний ресурс] – Режим доступу до ресурсу: https://www.geeksforgeeks.org/history-of-c/

15. Пояснення принципу роботи сертифікатів [Електронний ресурс] – Режим доступу до ресурсу: https://www.cloudflare.com/learning/ssl/how-does-ssl-work/
16. Nicolai M. Josuttis C++ 17 The Complete Guide: First Edition, 2019 – 454 p.
17. Реалізації простих програм за допомогою rpclib та grpc [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/Cycxyz/Appendixes/tree/main/Appx_1