Міністерство освіти і науки України
Національний університет «Києво-Могилянська академія»
Факультет інформатики
Кафедра математики

Кваліфікаційна робота

освітній ступінь – бакалавр

на тему: «**РОЗВИТОК ВЗІРЦІВ РЕАКТИВНОГО ПРОГРАМУВАННЯ**»

Виконав: студент 4-го року навчання,
Освітньої програми «Інженерія програмного забезпечення», 121

Клепацький Олег Святославович

Керівник Бублик В.В.,_____
кандидат фіз.-мат. наук, доцент

Рецензент _____
(прізвище та ініціали)

Кваліфікаційна робота захищена

з оцінкою _____

Секретар ЕК _____

«____» _____ 20____ р.

Київ – 2024

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мультимедійних систем факультету інформатики

ЗАТВЕРДЖУЮ

Зав. кафедри інформатики,

к. ф.-м. н. С. С. Гороховський

_____

(підпис)

“____” _____2023 р.

ЗАВДАННЯ

на кваліфікаційну роботу

студента 4-го року навчання Клепацького Олега Святославовича

Тема: Розвиток взірців реактивного програмування

Зміст ТЧ до кваліфікаційної роботи:

Індивідуальне завдання

Календарний план

Зміст

Перелік умовних позначень

Вступ

Розділ 1: Реактивне програмування та взірці реактивного програмування

Розділ 2: Реактивні розширення та бібліотеки імплементації взірців реактивного програмування

Розділ 3: Розробка демонстраційного проекту

Висновки

Перелік використаних джерел

Дата видачі 12.10.2023 р. Керівник _____

(підпис)

Завдання отримав _____

(підпис)

КАЛЕНДАРНИЙ ПЛАН ВИКОНАННЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Тема: Розвиток взірців реактивного програмування

Календарний план виконання роботи:

№ п/п	Назва етапу курсового проекту (роботи)	Термін виконання етапу	Примітка
1.	Отримання завдання на кваліфікаційну роботу.	12.10.2023	
2.	Ознайомлення з існуючою інформацією по темі	13.10.2023	
3.	Проектування прикладів	02.11.2023	
4.	Початок створення практичної частини	21.12.2023	
5.	Подання проміжної версії практичної частини	01.02.2024	
6.	Аналіз практичної частини; її корегування	26.02.2024	
7.	Початок написання теоретичної частини	01.03.2024	
8.	Подання проміжної версії текстової частини	15.03.2024	
9.	Остаточне завершення написання теоретичної частини роботи та корегування практичної частини;	10.05.2024	
10.	Створення презентації	13.05.2024	
11.	Захист кваліфікаційної роботи	29.05.2024	

Студент Клепацький О.С.

Керівник Бублик В. В.

“_____” _____

ЗМІСТ

ПЕРЕЛІК ТЕРМІНІВ ТА УМОВНИХ ПОЗНАЧЕНЬ

Callback – функція зворотного виклику

Асинхронність – це спосіб виконання програми, що дозволяє продовжити виконання інших завдань без очікування закінчення вже початих

Конкурентність – можливість виконання декількох частин програми одночасно

HTTP – HyperText Transfer Protocol – протокол передачі гіпертексту

GRPC – Google Remote Procedure Calls – протокол, що реалізує систему віддаленого виклику процедур

API – Application Programming Interface – програмний інтерфейс, набір класів та функцій, які надає програма/бібліотека

Коміт (commit) – у системах керування версіями об’єкт, який містить знімок стану репозиторію (проекту)

MSVC – Microsoft Visual C++ - компілятор C++ для ОС Windows

ВСТУП

Вимоги до сучасного програмного забезпечення характеризуються необхідністю швидких, відмовостійких і масштабованих програм, здатних обробляти багато завдань та потоків даних одночасно. Досягнення таких властивостей часто вимагає ефективного використання багатопоточного програмування. Хоча багатопотоковість може значно підвищити продуктивність і швидкість виконання програм, програмування у такий спосіб є на порядок складнішим однопоточного програмування. Розробники повинні вільно орієнтуватися в таких термінах, як стан гонитви (з англ. race condition), взаємне блокування (англ. deadblock), ефективно використовувати механізми синхронізації для уникнення таких проблем тощо. Таке навантаження може зробити процес розробки водночас схильним до помилок і складним для відлагодження.

У цьому контексті реактивне програмування є потужною парадигмою, яка спрощує розробку паралельних і асинхронних програм. Зосереджуючись на передачі потоків даних і архітектурі, що керується повідомленнями, реактивне програмування забезпечує абстракцію високого рівня для обробки асинхронних операцій. Ця парадигма абстрагує низькорівневі деталі керування потоками та їх синхронізацію, що дозволяє розробникам легше створювати адаптивні та стійкі системи.

Взірці реактивного програмування інкапсулюють найкращі практики та стратегії впровадження реактивних систем та вирішують притаманні таким системам проблеми. Ці взірці надають структурований підхід до проектування систем, які можуть ефективно передавати та обробляти потоки даних, взаємодіяти з користувачами та фоновими процесами без підводних каменів, з якими так чи інакше зіштовхуються програмісти з використанням традиційних методів багатопоточності.

Мова програмування C++, в особливості нові стандарти цієї мови, з чудовою підтримкою об'єктного, функціонального програмування та шаблонного метапрограмування, є природнім вибором для написання реактивних програм, проте з деяких причин C++ порівняно рідко застосовується розробниками реактивних систем.

Виходячи з цього, за ***мету даної роботи*** було поставлено показати спроможність та доцільність використання взірців реактивного програмування у сучасному програмному забезпеченні написаному мовою C++.

Об'єктом дослідження є реактивні взірці програмування. ***Предметом дослідження*** є реалізація реактивних взірці програмування мовою C++.

Мета роботи зумовила наступне ***наукове завдання*:**

1. Дослідити та класифікувати взірці реактивного програмування.
2. Дослідити існуючі реалізації реактивних взірців мовою C++.
3. Провести порівняльний аналіз реалізацій реактивних взірців.
4. Розробити демонстраційний проект – peer-to-peer чат-застосунок – для демонстрації використання взірців реактивного програмування з використанням бібліотеки, яка буде обрана на основі порівняльного аналізу.

Робота складається з трьох розділів.

У першому розділі проводиться огляд необхідної теорії з парадигми реактивного програмування, наводиться класифікація взірців реактивного програмування та надається концептуальний опис взірців.

Другий розділ присвячено вивченню особливостей існуючих бібліотек, що реалізують взірці реактивного програмування мовою C++, проводиться їх порівняльний аналіз на основі тестування.

У третьому розділі наводяться результати розробки демонстраційного проекту.

РОЗДІЛ 1: РЕАКТИВНЕ ПРОГРАМУВАННЯ ТА ВЗІРЦІ РЕАКТИВНОГО ПРОГРАМУВАННЯ

1.1 Реактивне програмування

Реактивне програмування – це програмування з асинхронними потоками даних. Застосовуючи різні операції до потоку даних, ми можемо розв'язати різні обчислювальні завдання. Основне завдання реактивного програмування полягає у перетворенні даних у потоки, незалежно від джерела цих даних. Наприклад, під час написання програми з графічним інтерфейсом ми маємо обробляти події натискання чи руху миші. Наразі більшість систем мають функції зворотнього виклику (або колбеки з англ. callback) для обробки таких подій у момент їх виникнення. У більшості випадків обробник виконує ряд операцій фільтрації перед тим як викликати метод для обробки даної події. У цьому контексті реактивне програмування допомагає агрегувати події натискання чи руху мишки у деяку колекцію і відфільтровує їх перед сповіщенням логіки обробника. Таким чином код застосунку (логіки обробника подій) не виконується задарма.

Модель обробки потоків даних досить добре вивчена і відома, а також легко реалізується розробниками програмного забезпечення. Практично все може бути конвертоване в потік: повідомлення, журнали виконання, властивості, пости тощо. Техніки функціонального програмування влучно підходять для обробки таких потоків даних. Мова програмування C++, в особливості нові стандарти цієї мови з відмінною підтримкою об'єктного та функціонального програмування, є природнім вибором для написання реактивних програм. Основна ідея реактивного програмування полягає в тому, що існують певні типи даних, які представляють значення у часі. Ці типи даних, а радше послідовності даних, представлені спостережуваними послідовностями у цій парадигмі програмування. Обчислення, пов'язані з значеннями, що змінюються у часі, будуть у свою чергу мати такі змінювані значення, і ми повинні асинхронно отримувати повідомлення про зміну залежних значень.

1.2 Маніфест реактивного програмування

Маніфест реактивного програмування (Reactive Manifesto) – це документ, який визначає мету та цілі парадигми реактивного програмування [1]. У цьому документі описано підхід до розробки сучасних програмних систем, які мають бути надійними, стійкими та масштабованими, та які відповідають сучасним вимогам до програмного забезпечення. Ідеї з цього маніфесту відображають бачення розробників з дуже широкого спектру областей, які використовують схожі моделі для побудови таких реактивних систем. Потреба в таких системах є результатом різких змін у вимогах до додатків, від яких вимагається мінімальний часу відгуку, безперебійна робота сервісів та здатності обробляти терабайти даних.

У цьому маніфесті вказано, що реактивні системи мають чотири ключові характеристики: швидкість реагування, стійкість, еластичність і комунікація на основі повідомлень [1]. Швидкий відклик системи підвищує довіру користувачів і робить взаємодію з системою більш ефективною. Еластичність гарантує, що система здатна адаптуватися до змінних робочих навантажень, уникаючи вузьких місць у системі і прозоро масштабуючись. З іншого боку, стійкість гарантує, що система буде адекватно реагувати на збої, що досягається шляхом реплікації та ізоляції. А завдяки архітектурі на основі повідомлень забезпечується слабка зв’язність, ізоляція та ефективне керування елементами системи за допомогою асинхронного зв’язку.

У сукупності всі ці принципи дизайну забезпечують гармонійний спосіб створення систем, які легко розробляти та підтримувати. Маніфест закликає притримуватись та застосовувати принципи реактивного програмування з початку розробки системи, а не щоразу перевідкривати їх.

1.3 Моделі Push vs Pull у реактивному програмуванні

Реактивне програмування засноване на push та pull моделях, як зазначають автори книги “C++ Reactive Programming” [2]. Система, побудована на pull моделі (модель витягування), очікує на запит на дані, щоб надати потік даних до споживача. Тобто, до джерела даних активно та явно надсилаються запити для отримання (витягування) нової інформації. Для цього доречно використати взірець ітератор, який до отримання нового значення буде блокувати потік виконання програми.

З іншої сторони, система, побудована на push моделі, агрегує та «проштовхує» (push) потік даних через мережу сигналів до кінцевого споживача даних. У цьому випадку, на відміну від pull моделі, оновлення даних передаються споживачу з самого джерела цих даних. Така асинхронність обробки даних досягається тим, що ми не блокуємо оброблювача (споживача) даних, а змушуємо його реагувати на зміни у даних. Для таких цілей використовується взірець Спостерігач (Observer). Використання push моделі є більш вигідним у великих програмах з користувацьким інтерфейсом, де неефективно блокувати основний потік для обробки користувацького інтерфейсу для очікування якихось подій. Цим досягається швидкий відклик реактивних програм на дії користувача.

1.4 Взірці реактивного програмування

У своїй книзі «Reactive Design Patterns» Рональд Кун надає вичерпну класифікацію патернів реактивного програмування [3]. Він поділяє їх на 5 категорій, а саме: взірці відмовостійкості та відновлення, взірці реплікації, взірці управління ресурсами, взірці керування потоком виконання та взірці керування та збереження стану (Рис. 1.1).

Взірці відмовостійкості та відновлення надають спосіб якісно справлятися зі збоями у системі та ефективно відновлюватися після них. Ці взірці спрямовані на ізоляцію обробки помилок, щоб підтримувати загальну стабільність системи та швидкий відклик.

Взірці реплікації фокусуються на реплікації компонентів і даних у реактивних системах для підвищення надійності системи. Ці взірці допомагають підтримувати узгоджений стан між компонентами системи і виконувати відновлення після помилок.

Взірці управління ресурсами пов'язані з ефективним управлінням ресурсами системи, такі як пам'ять, процесорний час і введення/виведення. Вони забезпечують масштабованість системи та її швидкий відклик за високих навантажень.

Взірці керування потоком виконання управляють частотою обробки запитів до системи, щоб запобігти її перевантаженню, а за необхідності кешують, перенаправляють або відкидають запити, що система не змогла обробити у необхідний час.

У наступних пунктах розглянемо деякі з цих взірців.

Рис. 1.1

1.4.1 Error Kernel

Взірець Error Kernel інкапсулює найбільш критичну частину системи для обробки помилок, забезпечуючи те, що помилки з периферійних частин системи не будуть впливати на основну логіку. Цей взірець централізує обробку помилок, що робить систему більш надійною та такою, що легше підтримувати.

Цей взірець зародився з потреби ізолювати критичну логіку в системах, де обробка помилок є надважливою. Початок він бере з моделі акторів, яку використовують у мові Erlang, яку використовували для систем телекомунікацій, де безвідмовність є важливою вимогою [4]. Модель акторів – це математична модель, у якій основною ідеєю є актор – сутність, яка є примітивом паралельного виконання [4]. Актори взаємодіють між собою асинхронно за допомогою повідомлень (замість звичного виклику методів у об'єктно-орієнтованій парадигмі). Кожен актор при отриманні повідомлення може модифікувати свій стан або створити нових акторів для обробки запиту [4]. Змінювати стан інших акторів можливо лише через асинхронні повідомлення.

Для прикладу, візьмемо систему онлайн платежів, де обробка платіжних транзакцій є основним функціоналом (Рис 1.2). Основний компонент буде делегувати обробку транзакцій дочірнім елементам, що забезпечить ізоляцію кожної транзакції. Також усі периферійні елементи, такі як нотифікації чи логування, при помилці не будуть впливати на роботу обробників транзакцій.

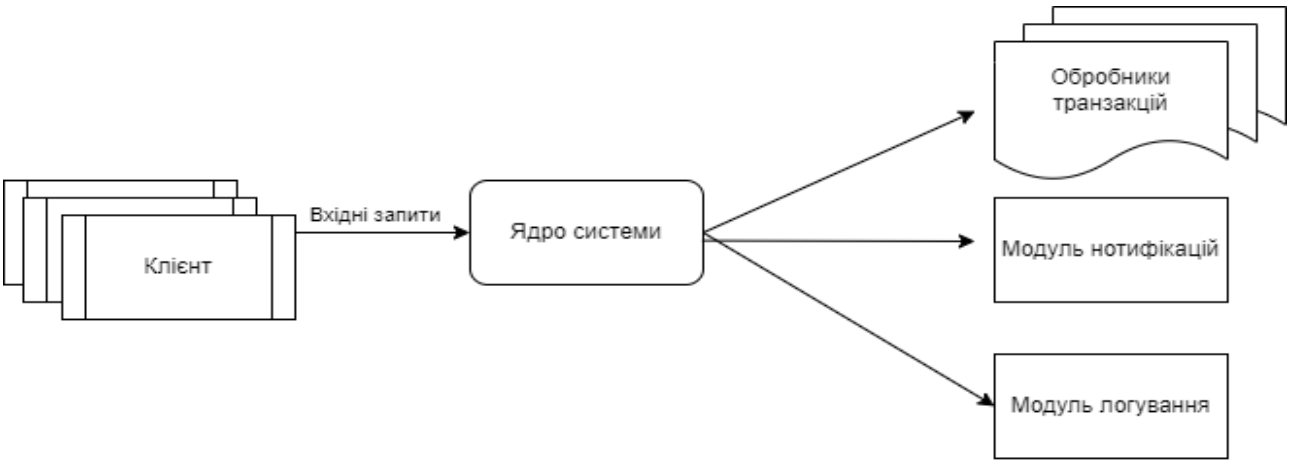

Рис. 1.2

1.4.2 Let-It-Crash

Взірець Let-It-Crash полягає у тому, що під час обробки помилки компонент системи йде коротким шляхом: відразу стає несправним і повертає код помилки (або за необхідності перезавантажується), замість того, щоб запускати довгу процедуру відновлення після помилки. Цей взірець використовує контрольовані ієрархії для ефективної обробки збоїв, сприяючи стійкості системи завдяки простоті.

Беручи початок з філософії мови програмування Erlang, цей взірець був часто застосований у реактивних системах. Одним з прикладів застосування може бути веб-сервер, який під час обробки запиту клієнтів, буде повністю перезапускати компонент, що відповідає за обробку даного запиту, звільняти всі його ресурси, повертати користувачу код помилки і відразу перезапускати процес для того, щоб мати таку ж спроможність обробити наступний запит від користувача. Тут припускається, що після отримання помилки користувач одразу повторить свій запит: саме тому нам потрібно перезапустити процес і перенаправити новий запит туди.

Таким чином, взірець Let-It-Crash вирішує питання складної обробки помилок і стійкості системи. Особливостями є швидка реакція на помилки, контрольований перезапуск компонентів, які мали помилки, та простоту керування обробкою помилок. Цей взірець найкраще підходить для систем, де такі компоненти не мають жодних зв’язаних сутностей (тобто мають лише залежності), тобто їх можна легко видалити та перезапустити. Прикладом таких систем є веб-сервіси, розподілені системи та системи з мікросервісною архітектурою.

1.4.3 Heartbeat

Взірець Heartbeat передбачає надсилання регулярних запитів «heartbeat» між компонентами в системі для відстеження їхнього стану (Рис 1.3). Назва цих запитів є аналогією до відстеження серцебиття у людини для розуміння її поточного стану. Якщо сервісу не вдається отримати відповідь протягом певного періоду часу, вважається, що він не відповідає або вийшов з ладу, що запускає попередньо визначені дії відновлення.

Взірець Heartbeat виник через потребу постійного моніторингу компонентів розподіленої системи. Він широко використовується у системах, де регулярні перевірки статусу мають вирішальне значення. Взірець забезпечує те, що сервіси знають про доступність та стан інших сервісів, запобігаючи тихим збоям (тобто збоям, які не були вчасно поміченими) у системі.

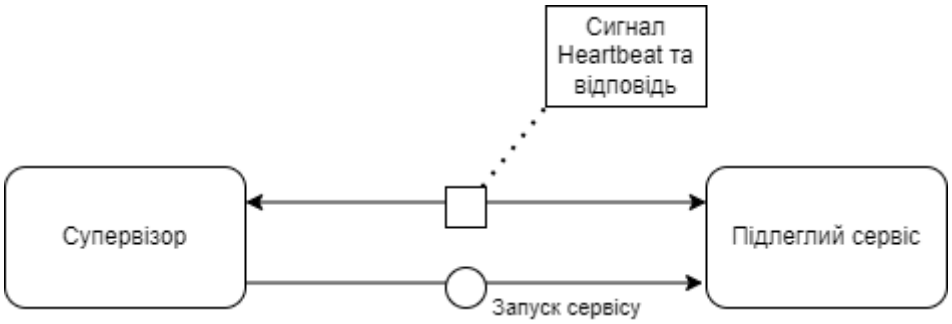

Рис. 1.3

1.4.4 Proactive Failure Signal

Основна ідея взірця Proactive Failure Signal це те, що компоненти системи активно сигналізують залежні компоненти про потенційні несправності, перш ніж ці несправності стануть критичними (Рис 1.4). Це дозволяє системі вживати превентивних заходів для пом’якшення впливу потенційних збоїв, реагуючи на відповідне повідомлення про несправність.

Наприклад, у хмарних системах сервіс, відповідальний за обробку вхідних запитів, може відстежувати різні метрики, такі як використання ресурсів (кількість використаної пам’яті) і показники продуктивності (час відгуку або кількість оброблених запитів на хвилину). Якщо він виявляє невідповідність у якійсь метриці, він надсилає сигнал про збій до центральної системи моніторингу, не чекаючи запиту від неї, через що сигнал і називається проактивним. Вчасно реагуючи на проблему, система забезпечує власну продуктивність і мінімальний час відгуку.

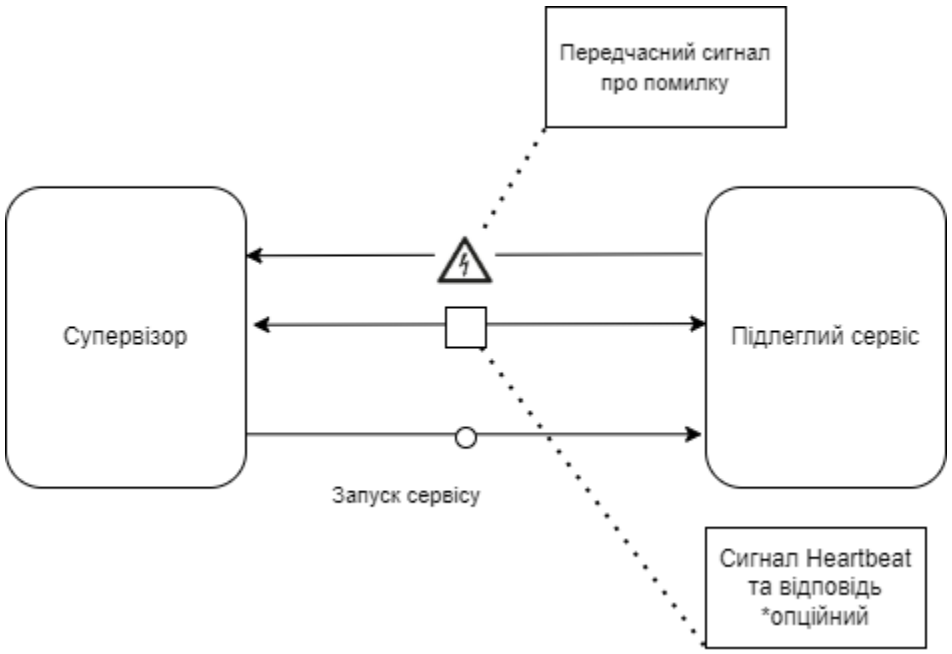

Рис. 1.4

1.4.5 Circuit Breaker

Назва взірця Circuit breaker походить від автоматичних вимикачів мережі (запобіжники), які власне запобігають перевантаженню мережі розмикаючи коло (Рис 1.5). У програмному забезпеченні, цей взірець був популяризований Майклом Найгардом у його книжці “Release It!” як спосіб будувати відмовостійкі системи, які можуть впоратись з тимчасовими збоями без перевантаження системи [5].

Цей взірець може застосовуватись у мікросервісній архітектурі, де сервіси комунікують з іншими сервісами за допомогою HTTP, GRPC або інших протоколів. Якщо сервіс, до якого йшли запити, не відповідає через заданий проміжок часу (timeout) або повертає помилкові статуси, спрацьовує Запобіжник. Наступні запити будуть перервані одразу для того, щоб уникнути додаткового навантаження на проблемний сервіс, аж поки той не відновить нормальну роботу. Після деякого проміжку часу Circuit Breaker дозволить відбутись деякій кількості запитів для того, щоб впевнитись, що сервіс відновився.

Взірець Circuit Breaker вирішує проблеми каскадних відмов і перевантаження системи. У його функції входять виявлення збоїв, тимчасовий розрив зв'язку із проблемним сервісом та контрольовані повторні спроби відновити зв'язок. Цей взірець застосовується у розподілених системах, мікросервісних архітектурах або у будь-яких інших сценаріях, де віддалені сервіси можуть виходити з ладу і запити до них просто погіршать проблему.

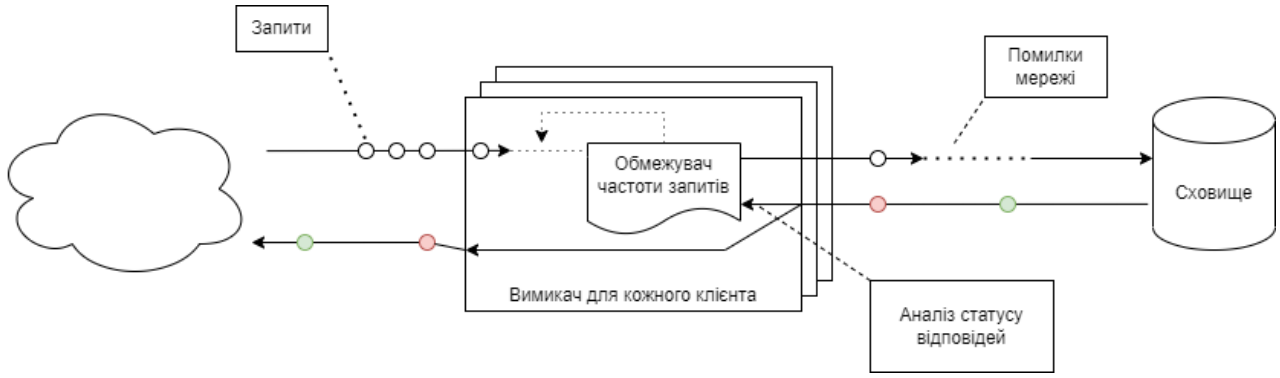

Рис. 1.5

РОЗДІЛ 2: РЕАКТИВНІ РОЗШИРЕННЯ ТА БІБЛІОТЕКИ ІМПЛЕМЕНТАЦІЇ ВЗІРЦІВ РЕАКТИВНОГО ПРОГРАМУВАННЯ

2.1 Функціональне та реактивне програмування

Парадигми функціонального та реактивного програмування, хоч і переслідують різні цілі, але все ж мають спільні риси. Обидві парадигми використовують такі техніки як незмінність даних, декларативне програмування та композиційніть, що сприяє написанню програм з передбачуваною поведінкою, які легко підтримувати та розширяти.

Незмінність даних у функціональному програмуванні пов'язана з тим, що програми (за деякими виключеннями) описуються «чистими» функціями, тобто функціями без побічних наслідків, які не модифікують жодного стану окрім локальних змінних. У реактивному програмуванні це зумовлено необхідністю поширювати потоки даних асинхронно, часто з використанням декількох потоків виконання. Щоб забезпечити цілісність даних та уникнути стану гонитви (з англ. race condition) між потоками, дані копіюються між асинхронними викликами.

Функціональне, як і реактивне програмування, є декларативними за природою. Функціональний дизайн і композиція функцій добре поєднуються з потоками даних, які, перш ніж надати споживачеві даних, треба фільтрувати, групувати, комбінувати, змінювати тощо. Такий список перетворень можна відобразити у декларативному стилі за допомогою ланцюгу операторів. Деякі з цих операторів можуть приймати на вхід функції-обробники, що подібно до функцій вищого порядку у функціональному програмуванні.

Не варто плутати це поєднання з функціонально-реактивним програмуванням, що є окремою парадигмою. У своїй статті «The essence and origins of FRP» Конан Елліотт визначає дві основні властивості функціонально-реактивного програмування: безперервна зміна значень у часі та проста і точна денотація цих змін [6]. Різниця полягає у тому, що класичне реактивне

програмування оперує дискретними значеннями у конкретних проміжках часу, тобто скінченними потоками даних, а ця парадигма фокусується саме на безперервній зміні у часі та формулюванні цього концепту, використовуючи конструкції функціональних мов, такі як Haskell [6].

2.2 Бібліотека std::ranges

Схожі властивості має бібліотека std::ranges зі стандарту C++20, яка використовує декларативний підхід з ланцюгами операцій для обробки структур даних [7]. Представлення даних абстраговано від структури даних – це деяка послідовність, яку ми можемо обробити повністю або частково, що надає змогу оптимізувати програми за допомогою лінивих обчислень.

Це досягається використанням ітераторів для структур даних, а також адаптерам, такі як transform, take, join, zip, які оброблюють ці значення за мірою необхідності. Композиція цих операцій (або адаптерів) імплементується за допомогою перевизначення бінарного оператору або (operator|), що по суті є просто синтаксичним цукром. Продемонструємо простий ланцюг операцій на прикладі алгоритму, який підносить перші 10 непарних чисел до квадрату:

```
 // генерує нескінченну послідовність чисел від 1
for (int num : std::views::iota(1)
                // залишаємо непарні числа
              | std::views::filter([](int n) { return n % 2 != 0; })
                // підносить до квадрату
              | std::views::transform([](int n) { return n * n; })
                // бере перші 10 елементів послідовності
              | std::views::take(10))
{
    std::cout << num << " ";
}
```

2.3 Реактивні розширення (ReactiveX)

ReactiveX (реактивні розширення) – це бібліотека, в основі ідеї якої лежать функціональна композиція та асинхронна обробка потоків даних [8]. Вона розширює взірець Спостерігач, поєднуючи його з послідовностями даних, та додає оператори для декларативної композиції ланцюгів операцій, що полегшує роботу з асинхронними потоками даних. Реалізації доступні різними мовами програмування, такі як JavaScript (RxJs), Java (RxJava), Python (RxPY), C++ (RxCpp) та інші.

Основні концепції ReactiveX – це потоки даних (або подій), які можна спостерігати, спостерігачі, оператори та планувальники (schedulers) [8]. Спостерігати за потоками даних означає отримувати від них сповіщення про нові дані. Існує три типи сповіщень: next – для наступного значення з послідовності, error – для позначення того, що сталась помилка, і complete – послідовність закінчилась і більше не буде надавати сповіщень. Обробниками цих сповіщень і будуть спостерігачі: вони реалізовують методи для обробки усіх типів сповіщень. Оператори дозволяють компонувати спостерігачів у ланцюги операцій у декларативний спосіб. Планувальники контролюють виконання методів спостерігачів, визначають, на якому потоці виконання це буде відбуватись (поточному, новоствореному чи на потоці для обробки операцій вводу-виводу). Планувальники дозволяють ефективно управляти конкурентністю та асинхронними операціями.

Оператори у ReactiveX нагадують аналогічні з бібліотеки std::ranges, але різницею є те, що у ReactiveX значення розподілені у часі і вони обробляються асинхронно, а у std::ranges значення розподілені у просторі (пам’яті) і обробляються зазвичай у один потік виконання.

2.4 Бібліотеки реактивного програмування мовою C++

Як вже було сказано, ReactiveX має безліч імплементацій різними мовами програмування, проте у цій роботі буде розглянуто лише декілька бібліотек реактивного програмування мовою C++, які реалізують API, визначене ReactiveX. Для аналізу було обрано чотири бібліотеки з відкритим кодом.

Спільною рисою усіх імплементацій є те, що вони містять лише заголовні файли (англ. header-only library) через використання шаблонів та різних технік метапрограмування. Це означає, що такі бібліотеки не обов'язково компілювати для включення у інший проект, а також, що компілятор бачить усі визначення функцій, що допомагає оптимізовувати об'єктний код. Недоліками такого підходу є довший час компіляції та необхідність перекомпілювати більшу частину коду при зміні коду бібліотеки.

2.4.1 RxCpp

RxCpp – це офіційна та основна бібліотека реактивного програмування у C++ [9]. Це найстарша з розглянутих бібліотек, вона добре документована і має більше згадувань у онлайн ресурсах, що робить розробку для нових користувачів цієї бібліотеки легшою. З переваг також можна зазначити стабільну реалізацію та повністю реалізоване API, визначене ReactiveX.

Недоліки цієї бібліотеки це застарілий програмний код, який використовує функціонал стандартів C++11 та C++14. У цій реалізації подеколи присутня неефективна логіка та надмірне використання динамічної пам'яті, що впливає на час виконання. Також суттєвим недоліком є довгий час компіляції, що в більшій мірі зумовлено неефективною реалізацією шаблону observable. Цей шаблон містить увесь ланцюг операцій перетворення другим параметром шаблону, що

збільшує обсяг об’єктного коду та час компіляції. Приклад згенерованого компілятором типу наведено нижче:

```
class rxcpp::observable<class std::vector<unsigned char,class std::allocator<unsigned char> >,struct
rxcpp::operators::detail::lift_operator<class std::vector<unsigned char,class std::allocator<unsigned char>
>,struct rxcpp::operators::detail::flat_map<class rxcpp::observable<class rxcpp::observable<unsigned
char,class rxcpp::dynamic_observable<unsigned char> >,struct rxcpp::operators::detail::lift_operator<class
rxcpp::observable<unsigned char,class rxcpp::dynamic_observable<unsigned char> >,struct
rxcpp::operators::detail::flat_map<class rxcpp::observable<int,struct rxcpp::sources::detail::range<int,class
rxcpp::identity_one_worker> >,class `int __cdecl main(void)'::`2'::<lambda_1>,struct
rxcpp::util::detail::take_at<1>,class rxcpp::identity_one_worker>,struct
rxcpp::operators::detail::window<unsigned char> > >,class `int __cdecl main(void)'::`2'::<lambda_2>,struct
rxcpp::util::detail::take_at<1>,class rxcpp::identity_one_worker>,struct rxcpp::operators::detail::tap<class
std::vector<unsigned char,class std::allocator<unsigned char> >,class std::tuple<class `int __cdecl
main(void)'::`2'::<lambda_3> >,struct rxcpp::operators::detail::tap_observer_factory<class
std::vector<unsigned char,class std::allocator<unsigned char> >,class std::tuple<class `int __cdecl
main(void)'::`2'::<lambda_3> > > > > >
```

2.4.2 Another-RxCpp

Another-RxCpp з самої назви вказує на причетність до оригінальної бібліотеки. Вона покликана вирішити проблеми RxCpp та реалізовує усі оператори з API ReactiveX [10].

Основна ідея полягає у нівелюванні проблеми шаблонів з RxCpp використовуючи техніку стирання типу (англ. type erasure). Ця техніка часто використовується у парі з метапрограмуванням у C++ і полягає у поєднанні шаблонів та наслідування. Компілюючи функцію, яка використовує посилання на базовий тип, компіляторові не треба знати про усі деталі класу (в тому числі його тип), який наслідує базовий [11]. Недоліком є втрата часу на віртуальні виклики та використання динамічної пам’яті.

Також ця бібліотека використовує std::function для передачі усіх колбеків для спостерігача, що добре поєднується у концепцію стирання типів. Недоліком є значне збільшення розміру усіх об’єктів під час виконання.

Загалом, Another-RxCpp покращує ситуацію з часом компіляції, але реалізує це у дещо спрощений спосіб, який використовує багато динамічної пам’яті.

2.4.3 cpp-frp

cpp-frp (C++ Functional Reactive Programming) – це бібліотека для реактивного програмування у C++. Вона написана з використанням стандарту C++14 [12], але це можна досі назвати недоліком, оскільки не використовуються нові можливості стандартів C++17 та C++20 для роботи з шаблонним метапрограмуванням.

Має декілька недоліків, а саме не прив’язана до ReactiveX, хоч і має схожу ідею, а також має дуже скупу документацію, що набагато ускладнює роботу з цією бібліотекою. Не реалізовує більшість з ReactiveX API по вже згаданій причині, тому не є сумісною з іншими бібліотеками.

Також помічаємо недолік у назві бібліотеки, а саме використання функціонально-реактивного програмування у назві, хоча це зовсім інша парадигма, не пов’язана з цією бібліотекою. Про різницю між функціонально-реактивним програмуванням та реактивним програмуванням з функціональною композицією я зазначав раніше.

2.4.4 ReactivePlusPlus

ReactivePlusPlus – це найновіша бібліотека реактивного програмування мовою C++ [13]. Написана з використанням стандарту C++20, бібліотека використовує нові інструменти для роботи з шаблонами, що покращує роботу з кодом, наприклад концепти та вивід типових параметрів шаблону (template

argument type deduction). Ця бібліотека використовує схожу техніку як і в Another-RxCpp, але робить тримаючи баланс між стиранням типу та продуктивністю.

Ця бібліотека виправляє майже усі недоліки попередників. Для досягнення більшої гнучкості використовуються стратегії як типовий параметр шаблону для призначення різної поведінки, в тому числі для вибору моделі пам'яті: використання стеку і копіювання значення за необхідності або використання динамічної пам'яті без копіювання значення.

До бібліотеки написана достатня кількість документації для розуміння принципу роботи її компонентів та опису API. Вона добре протестована автоматизованими тестами. Автор також додає графіки часу виконання для кожного тесту у порівнянні з бібліотекою RxCpp, які є підтвердженням покращення продуктивності цієї бібліотеки [14].

Знизу наведено один з таких графіків для трьох тестів: час виконання для кожного коміту, пунктирними лініями позначено RxCpp, суцільною лінією позначено ReactivePlusPlus (Рис 2.1) [14]. З графіку видно, що конкретно для цих тестів перевага у швидкодії вимірюється у сотні наносекунд. Але це стає вирішальним фактором, коли такий код викликається тисячі разів під час виконання програми. Для реактивних систем, у яких час відклику є одним з найважливіших критеріїв, мілісекунди мають значення.

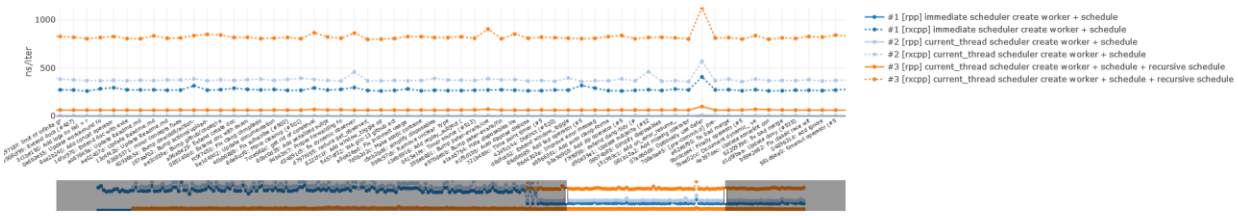

Рис. 2.1

Бібліотека досі у активній розробці, тому недоліками є нестабільність коду та відсутність деяких операторів з ReactiveX API.

2.5 Порівняльний аналіз бібліотек

Висновком з опису усіх вад та переваг бібліотек буде те, що фаворитами є RxCpp через стабільний, відлагоджений та документований код, та ReactivePlusPlus, який виправляє майже усі вади попередника, але вагомим недостатком є відсутність деяких операторів у реалізації.

Для проведення більш ґрунтовного аналізу було написано тестову програму з аналогічною генерацією та подальшим ланцюгом перетворень послідовності даних. Виконував тести на локальній машині, тому для точності вимірювань я проводив декілька запусків під приблизно однаковим навантаженням системи. Недостатність одного тесту я аргументую тим, що планувальник ОС (у моєму випадку Windows) може по-різному пріорітизувати виконання процесу компіляції або виконання тестової програми, тож було проведено ряд випробувань.

На графіках нижче показано середнє значення часу компіляції та виконання за результатами десяти тестів (Рис. 2.2, 2.3).

Рис. 2.2

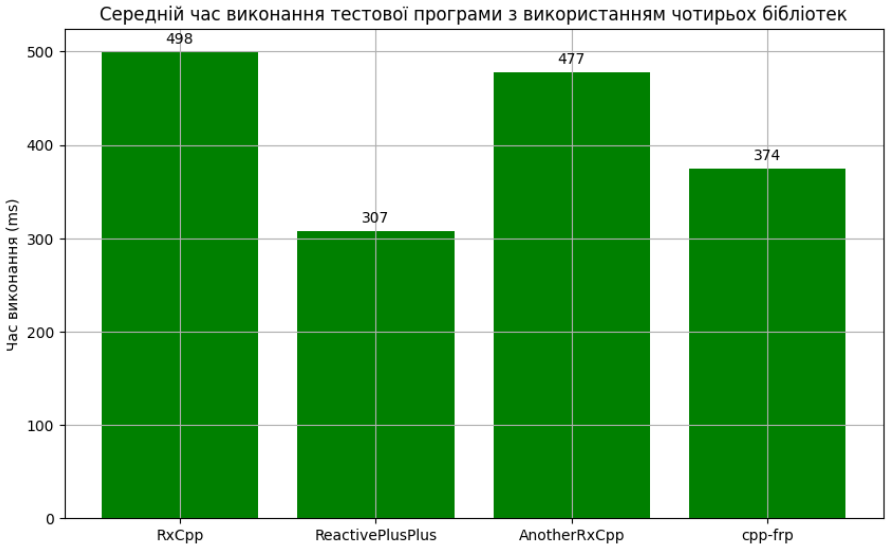

Рис. 2.3

На наступних графіках показано більш точну репрезентацію тих самих даних. Квадратами показано інтерквартильний діапазон, у які входять значення між першим квартилем (перші 25% значень) та третім (75%) квартилем. Додаткові горизонтальні лінії показують значення, що не перевищують мінімальне та максимальне значення у інтерквартильному діапазоні у півтора рази. Точками показано значення, які мають досить велику різницю. Кольорова лінія у квадраті показує медіану даних.

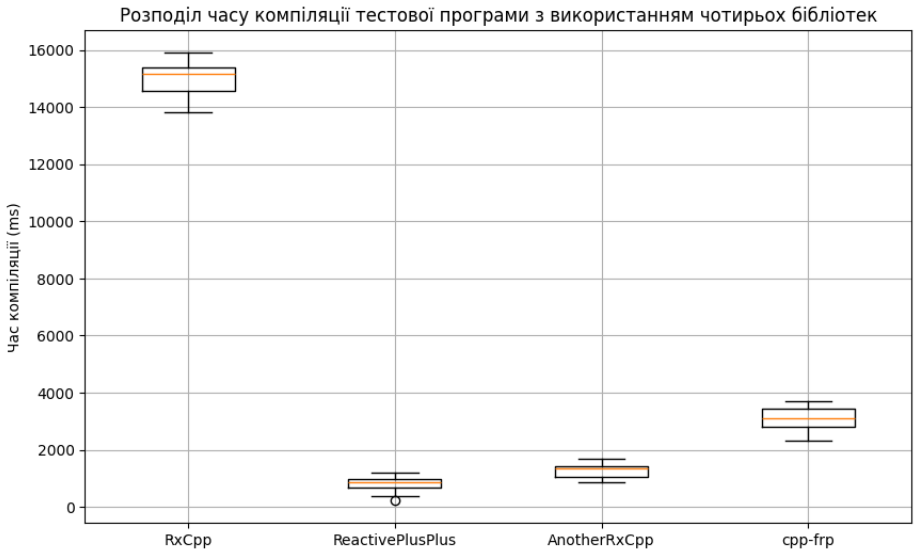

Рис. 2.4

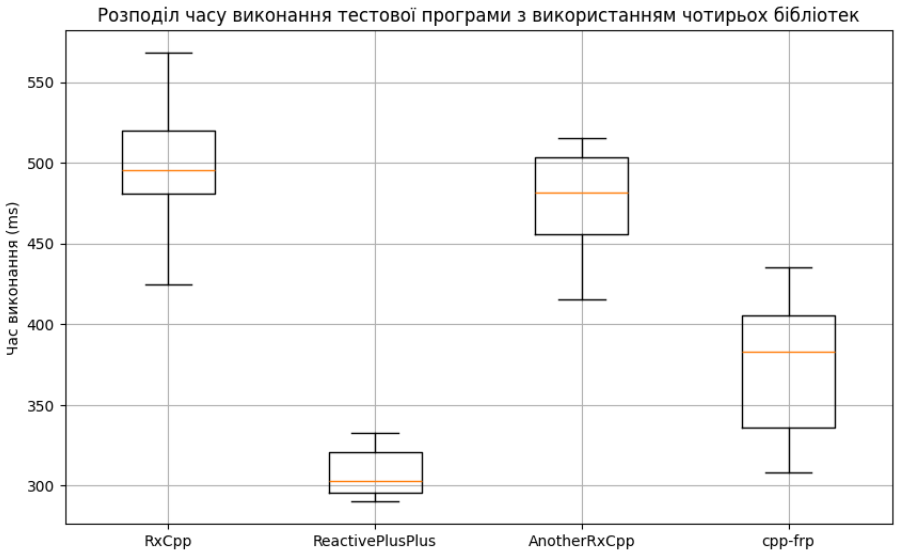

Рис. 2.5

Ці графіки показують достатню точність тесту.

За двома критеріями (час компіляції та час виконання тесту) переважає бібліотека ReactivePlusPlus. Зважаючи на це і на переваги, які я вказував раніше, ця бібліотека була обрана для використання у демонстраційному проекті.

РОЗДІЛ 3: РОЗРОБКА ДЕМОНСТРАЦІЙНОГО ПРОЕКТУ

3.1 Опис проекту

Для демонстрації можливостей інтегрування взірців реактивного програмування було створено демонстраційний проект під назвою “Reactive Chat”. Це peer-to-peer чат застосунок, який дозволяє здійснювати обмін текстовими повідомленнями між кількома клієнтами.

Обмеженням застосунку є те, що окремі клієнти можуть розпізнати один одного лише в локальній мережі. Також застосунок підтримує з’єднання на одній машині для зручності тестування. Це спрошення було навмисним, оскільки цього достатньо для тестування комунікації між клієнтами.

Застосунок тестувався лише на ОС Windows, але за потреби може бути портований на Linux чи MacOS завдяки використанню крос-платформенного фреймворку.

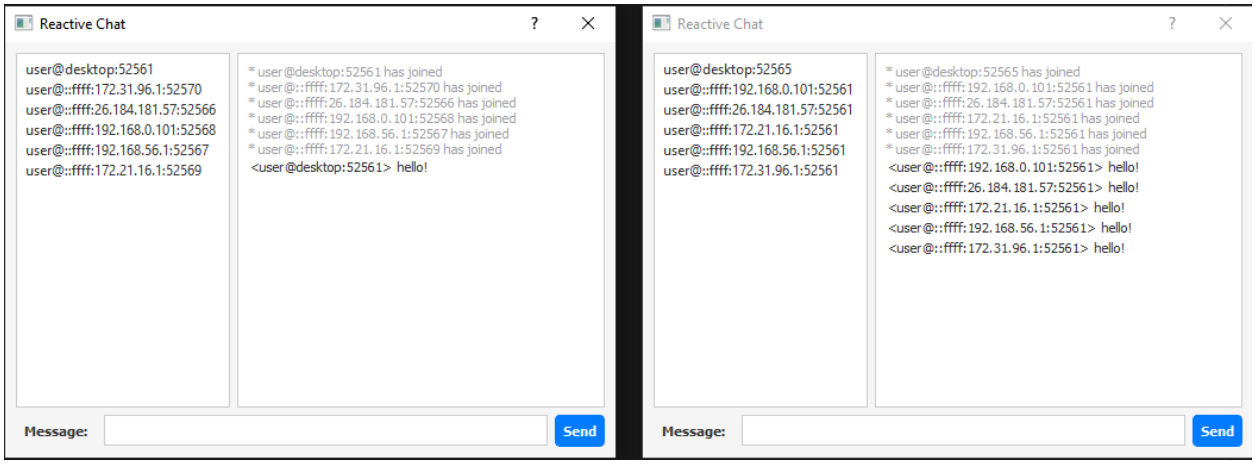

Рис. 3.1

3.2 Опис використаних інструментів розробки

Для виконання цього проекту було обрано кросплатформенний фреймворк Qt для зручності виконання UI частини. Цей фреймворк є крос-платформенним і дозволяє портувати програми майже без змін у програмному коді на Windows, Linux, MacOS та Android. Були використані компоненти Qt Core, Qt Gui, Qt Widgets та Qt Network.

Для розробки застосунку використовувалось інтегроване середовище розробки VS Code, для збірки використовувалась система збірки CMake і компілятор MSVC. Для зручності роботи також використувалась система контролю версій Git.

3.3 Принцип роботи застосунку та інтеграція взірців

Застосунок працює за наступним принципом: відразу після запуску програма запитує у ОС усі відомі адреси у локальній мережі. Після цього, вона надсилає UDP повідомлення на усі відомі адреси домашньої мережі. Один UDP сокет завжди використовується для прослуховувань таких повідомлень. Коли програма отримує таке повідомлення від іншого клієнта, вона додає адресу до свого локального списку і ініціює TCP з’єднання з новим клієнтом. Таким чином кожен клієнт після ініціалізації буде попарно з’єднаний з іншим через TCP з’єднання.

Для кодування та декодування повідомлень використовується формат CBOR (Concise Binary Object Representation) стандарту RFC8949 [15]. Такий формат був обраний через наявність існуючої реалізації у фреймворку Qt, малі накладні витрати, а також через відсутність необхідності задавати схему даних.

Така peer-to-peer архітектура, на відміну від клієнт-серверної архітектури, не вимагає окремого виділеного серверу і не вимагає двох різних реалізацій для серверу і клієнта – програмний код ідентичний для усіх клієнтів.

У цій програмі усі повідомлення від інших клієнтів представляються у вигляді єдиного потоку повідомлень, до якого під'єднано спостерігача, який власне і оброблює вхідні повідомлення та відображає їх на відповідних елементах інтерфейсу.

Для підтримки зв'язку між клієнтами було використано взірець Heartbeat. Таким чином, кожен клієнт у даний момент часу буде отримувати статус інших і за потреби реагувати на зміни повідомленням у інтерфейсі. Для цього кожен клієнт через певний проміжок часу надсилає повідомлення зі своїм статусом, щоб інші клієнти могли це обробити.

Взірець Circuit Breaker було використано для унеможливлення перевантаження окремого клієнту багатьма повідомленнями за раз. Для цього він контролює кількість повідомлень, відправлених з одного клієнту іншому у невеликий період часу. Додатковою властивістю є унеможливлення спаму через застосунок.

Діаграма класів проекту знаходиться у Додатку А.

ВИСНОВКИ

Проаналізовано реактивне програмування як парадигму. Визначена предметна область, де ця парадигма застосовується і поєднується з іншими парадигмами програмування, зокрема об’єктно-орієнтованою, функціональною і узагальненою.

Класифіковано та описано існуючі патерни реактивного програмування.

Проведений аналіз існуючих open-source реалізацій патернів реактивного програмування з використанням C++. Було порівняно вибрані реалізації патернів за допомогою тестової програми за кількома метриками, такі як час компіляці та час виконання.

Розроблено демонстраційний проект “Reactive Chat” – це peer-to-peer чат застосунок з використанням Qt. Було застосовано бібліотеку взірців реактивного програмування ReactivePlusPlus через швидший час компіляції та оптимізацію ресурсів під час виконання.

У підсумку, доведено, що реактивне програмування – це правильний інструмент у ситуаціях, коли вимогами до програмної системи є швидкий відклик, відмовостійкість та гнучкість до навантаження. Розглянуті взірці реактивного програмування дозволяють писати програми, які коректно керують ресурсами системами, адекватно реагують на помилки та забезпечують взаємодію компонентів всередині системи за допомогою правильних абстракцій. Було показано, що мова програмування C++ є доцільним вибором для написання продуктивних програм з використанням реактивної парадигми.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. The Reactive Manifesto [Електронний ресурс]. – 2024. – Режим доступу до ресурсу: https://www.reactivemanifesto.org/.
2. Praseed P. C++ Reactive Programming / P. Praseed, A. Peter., 2018. – 348 с.
3. Kuhn R. Reactive Design Patterns / R. Kuhn, B. Hanafee, J. Allen., 2017. – 392 с.
4. Virding R. Concurrent Programming in Erlang (2nd Edition) / R. Virding, C. Wikstrom, M. Williams., 1996. – 358 с.
5. Nygard M. Release It!: Design and Deploy Production-Ready Software (Pragmatic Programmers) 1st Edition / Michael Nygard., 2017. – 350 с.
6. Elliott C. The essence and origins of FRP / Conal Elliott., 2015.
7. Ranges Library C++20 [Електронний ресурс] – Режим доступу до ресурсу: https://en.cppreference.com/w/cpp/ranges.
8. ReactiveX - Reactive Extensions [Електронний ресурс] – Режим доступу до ресурсу: https://reactivex.io/.
9. RxCpp Library [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/ReactiveX/RxCpp.
10. Another-RxCpp Library [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/CODIANZ/another-rxcpp.
11. C++ type erasure [Електронний ресурс]. – 2010. – Режим доступу до ресурсу: https://cplusplus.com/articles/oz18T05o/.
12. cpp-frp library [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/google/cpp-frp.
13. ReactivePlusPlus library [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/victimsnino/ReactivePlusPlus.
14. ReactivePlusPlus benchmark [Електронний ресурс] – Режим доступу до ресурсу: https://victimsnino.github.io/ReactivePlusPlus/v2/benchmark.
15. RFC 8949 [Електронний ресурс] – Режим доступу до ресурсу: https://datatracker.ietf.org/doc/html/rfc8949.

ДОДАТОК А
(довідниковий)

UML-діаграма класів демонстраційного проекту