Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра мультимедійних систем

Магістерська робота

освітній ступінь – магістр

на тему: «**ЗАСТОСУВАННЯ БАГАТОПОТОКОВИХ МОДЕЛЕЙ ВЗІРЦІВ ПРОЕКТУВАННЯ ДЛЯ СТВОРЕННЯ МАСШТАБОВАНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ**»

Виконав: студент 2-го року навчання,

Спеціальності
121 Інженерія програмного забезпечення

Божко Владислав Вадимович

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

Рецензент _____
(прізвище та ініціали)

Курсова робота захищена
з оцінкою _____

Секретар ЕК _____
«____» _____ 20____ р.

Київ – 2023

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)

“____”_____2022 р.

ЗАВДАННЯ

на курсову роботу

студента 2-го року навчання Божка Владислава Вадимовича

Тема: Застосування багатопотокових моделей взірців проектування для створення масштабованого програмного забезпечення

Зміст ТЧ до кваліфікаційної роботи:

Індивідуальне завдання

Календарний план

Зміст

Перелік умовних позначень

Вступ

Розділ 1: Огляд взірців проектування масштабованого програмного забезпечення

Розділ 2: Багатопотокові моделі взірців проектування у масштабованому програмному забезпеченні

Розділ 3: Реалізація масштабованого програмного забезпечення на основі отриманих моделей

Висновки

Перелік використаних джерел

Дата видачі “___” _____ 2022 р. Керівник _____

Завдання отримав _____

КАЛЕНДАРНИЙ ПЛАН ВИКОНАННЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Тема: Розробка застосування для проектування, відображення та аналізу ER-моделей

Календарний план виконання роботи:

№ п/п	Назва етапу курсового проекту (роботи)	Термін виконання етапу	Примітка
1.	Отримання завдання на курсову роботу.	12.10.2022	
2.	Ознайомлення з існуючою інформацією по темі	13.10.2022	
3.	Проектування прикладів	02.11.2022	
4.	Початок створення практичної частини	21.12.2022	
5.	Подання проміжної версії практичної частини	01.03.2023	
6.	Аналіз практичної частини; її корегування	26.04.2023	
7.	Початок написання теоретичної частини	01.05.2023	
8.	Подання проміжної версії текстової частини	15.05.2023	
9.	Остаточне завершення написання теоретичної частини роботи та корегування практичної частини;	26.05.2023	
10.	Створення презентації	01.06.2023	
11.	Захист курсової роботи	13.06.2023	

Студент Божко В. В.

Керівник Бублик В. В.

“_____” _____

ЗМІСТ

ПЕРЕЛІК ТЕРМІНІВ ТА УМОВНИХ ПОЗНАЧЕНЬ

- API (англ. Application Programming Interface) – механізм надання однією програмою сервісів іншим програмам
- HTTP (англ. HyperText Transfer Protocol) – протокол передачі даних прикладного рівня, що широко використовується для передачі веб-матеріалів та комунікації програм
- Сокет – структура даних, яка описує мережеве з’єднання
- Доменний сокет (англ. domain socket) – структура даних, специфічна для UNIX та інших POSIX-сумісних систем, що описує з’єднання між процесами, що виконуються на одній машині
- Чанк (англ. chunk – шматок) – фрагмент певної інформації або файлу
- Кластер – система з окремих комп’ютерів, що об’єднані мережею та виконують спільну задачу
- Зв’язність – ступінь взаємопов’язаності компонентів програмного забезпечення або системи
- Користувацьке середовище (англ. user space) – виділена ділянка пам’яті та код, що виконується поза ядром операційної системи
- Спекулятивне виконання – поширена оптимізація швидкодії центральних процесорів, що полягає у обчисленні результатів обох ланок умовного оператора, замість очікування на обчислення умови. Після обчислення умови результати обчислення хибної ланки видаляються, і виконання продовжується у звичному режимі
- Дескриптор – цілочисельний ідентифікатор файлу або мережевого чи іншого з’єднання, унікальний в межах одного процесу
- Датаграма – базове окреме повідомлення, що надсилається мережею

ВСТУП

Обсяг даних, який генерується та обробляється інформаційними системами усього світу вже давно вийшов за межі того, що бодай теоретично могло би бути оброблене на одній, навіть найпотужнішій, машині.

Розподілені обчислення, які раніше були доступні лише на суперкомп’ютерах, із розповсюдженням хмарних сервісів, стали широко доступними. Можливість в будь-який момент та з мінімальними зусиллями розгорнути кластер машин робить масштабовані обчислення та обробку даних доступнішими ніж будь-коли, і вдосконалення механізмів роботи масштабованого програмного забезпечення є нагальною проблемою інженерії програмного забезпечення. Одним із основних напрямків такого вдосконалення є оптимізація комунікації між компонентами масштабованого ПЗ.

Мета магістерської роботи – виявити, яким чином асинхронні моделі взірців проектування можуть бути застосовані у створенні масштабованого багатопотокового ПЗ та створити відповідні спеціалізації взірців для вдосконалення механізмів комунікації компонентів у масштабованому програмному забезпеченні

Завдання магістерської роботи – дослідити існуючі взірці проектування масштабованого ПЗ; проаналізувати вплив класичних взірців проектування та їх асинхронних моделей на швидкодію масштабованого ПЗ; розробити власну бібліотеку для комунікації між компонентами масштабованого ПЗ та на її основі змоделювати спеціалізацію взірця Проактор [1]; порівняти отриману реалізацію із існуючими рішеннями

Об’єкт дослідження – масштабоване програмне забезпечення

Предметом дослідження є налаштування асинхронних моделей взірців проектування на ефективну комунікацію компонентів масштабованого програмного забезпечення

У рамках магістерської роботи було досліджено та імплементовано власну реалізацію механізму комунікації «Черга повідомлень», та на її основі було

вивчено вплив використання класичних взірців проектування, таких як «Проактор» у сучасних масштабованих програмах, що інтенсивно працюють із механізмами введення-виведення. Були розглянуті недоліки та потенційні покращення створеної моделі та було виконане її порівняння із існуючими стандартними рішеннями.

Використане програмне забезпечення

Для розробки прикладів було використано:

- Текстовий редактор коду Visual Studio Code
- Заголовні файли системних викликів ядра ОС Linux
- Бібліотека liburing
- Набір інструментів збірки CMake

Робота складається з трьох розділів.

У першому розділі роботи було розглянуто проблеми, специфічні для масштабованих та розподілених систем і способи та особливості їх вирішення.

У другому розділі, на прикладі взірця проектування «Проактор», засновуючись на нових API операційної системи Linux, було розглянуто застосування та вплив класичних взірців проектування на масштабоване програмне забезпечення.

У третьому розділі було описано власну реалізацію механізму комунікації «Черга повідомлень» на основі нової поліпшеної реалізації взірця «Проактор» та модельної задачу минулого року – HTTP-сервера. Обидві реалізації було протестовано та проведено їх порівняння із еталонними стандартними імплементаціями.

РОЗДІЛ 1: ОГЛЯД ВЗІРЦІВ ПРОЕКТУВАННЯ МАСШТАБОВАНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Протягом більш ніж п’ятдесяти років, зріст обчислювальної потужності інтегрованих мікросхем керувався експоненційним «Законом Мура» [2]. І все ж, не зважаючи на ці неймовірні інженерні досягнення, обсяг інформації, яку можна обробити на одній машині, є замалим для багатьох актуальних проблем. Найсучасніші взірці центральних процесорів можуть контролювати 12 терабайт оперативної пам’яті та мають до 96 ядер [3], і цих об’ємів недостатньо для багатьох проблем пошуку даних, їх агрегації, машинного навчання та інших. Але, навіть для задач, для яких таких обчислювальних можливостей вистачає, з багатьох причин доцільніше може бути розподілити обчислення між декількома меншими комп’ютерами. За умови правильної реалізації таке розділення може дозволити підвищити ступінь паралелізму обчислень, зекономити кошти за рахунок використання менш передового обладнання та покращити стійкість системи до збоїв. Проте саме правильна імплементація розподілених обчислень є однією з важливих проблем сучасної інженерії програмного забезпечення.

Визначення взірців проектування такого програмного забезпечення, які можуть бути перевикористані для багатьох задач, дозволяє зручніше проектувати розподілені системи та обмінюватись знаннями й досвідом про них. Це, у свою чергу, дозволяє розробникам уникати типових помилок та концентруватись на проблемах, специфічних для задач, які вони вирішують.

1.1 Структура розподілених обчислень

У розподілених системах найменшою одиницею виконання зазвичай є вузол (англ. node). Вузол може бути виділеною окремою машиною, віртуальною машиною або контейнером виконання. Декілька вузлів, об’єднаних між собою в одну систему для виконання спільних обчислень, називаються кластером.

Усі вузли кластеру зазвичай об’єднані між собою фізичною або віртуальною локальною мережею (англ. VLAN – Virtual Local Area Network). Топологія цієї

локальної мережі може різнитись у залежності від характеру виконуваних обчислень. Для використання кластеру необхідно мати змогу поставити перед ним задачу та розподілити її між вузлами. Для цих цілей серед вузлів кластера можуть бути обрані один або декілька керуючих вузлів, які виконують функції керування паралельно із або замість обчислень.

Сучасні кластери, особливо ті, які виконують не обчислення, а обробку запитів користувачів в реальному часі, доволі часто включають в себе вузли, які можуть виконувати різні додаткові виокремлені ролі. Серед поширених ролей вузлів можна виділити:

- Вузли кешування, які працюють із програмним забезпеченням Memcached, Redis або подібними. Ці вузли виконують тимчасове збереження результатів обчислень або пошуку даних;
- Вузли балансування навантаження, які можуть розподіляти запити або завдання між іншими вузлами, контролюючи їх працездатність та поточну завантаженість;
- Вузли обміну повідомлень, які агрегують та пересилають повідомлення між вузлами кластеру, контролюючи їх доставку та обробку. Такі вузли зазвичай використовують програмне забезпечення на кшталт Apache Kafka та RabbitMQ;

Кінцева структура кластеру проектується з урахуванням конкретних задач, які цей кластер буде виконувати, вимог до швидкості їх виконання, ступеню стійкості до відмов та інших. Сучасні хмарні сервіси та засоби контейнеризації та віртуалізації дозволяють змінювати структуру або розмір кластеру в реальному часі, адаптуючись до поточних потреб. Отже, сучасні взірці масштабованого програмного забезпечення мають адаптуватись до змін у середовищі виконання та ефективно використовувати усі наявні обчислювальні ресурси.

1.2 Перехід від багатопотокових до розподілених обчислень

Принципових відмінностей між багатопотоковими та розподіленими обчисленнями небагато – велика обчислювальна задача розбивається на менші частини, які можуть бути виконані паралельно. Ці менші частини, у свою чергу, розподіляються між потоками програми або вузлами кластеру, і результати цих паралельних обчислень об’єднуються для обчислення фінального результату. Проте у розподілених обчисленнях є одне суттєве обмеження в порівнянні із паралельними – це комунікація між вузлами.

Навіть найшвидші мережеві інтерфейси фізично не можуть порівнятись із тією пропускною здатністю та рівнем затримок, із якими взаємодіють ядра одного процесору. Для порівняння, ядра звичайного сучасного центрального процесору можуть комунікувати в середньому за 50 наносекунд. У той же час, найшвидші у світі мережеві інтерфейси у найкращих випадках пропонують затримки близько мікросекунди, до 20 разів повільніше. Більше того, для розрахунку кінцевої затримки комунікації, це значення необхідно помножити на два (адже в комунікації беруть участь як мінімум два інтерфейси) і додати затримку внутрішніх компонентів обох систем. Навіть у такому випадку збільшення часу, необхідного для комунікації між вузлами ставить суттєві обмеження на обсяг інформації, що передається між вузлами для проведення обчислень. А більшість розподілених систем працює зі звичним мережевим обладнанням, із затримками на кілька десяткових порядків вище.

Такі обмеження покладають на розробників додаткову відповідальність за відповідальне використання обмеженої пропускної здатності мережевих інтерфейсів. Адже у масштабованому програмному забезпеченні неконтрольована або надлишкова комунікація між вузлами системи може призвести до критичного її сповільнення, ставлячи під сумнів всю доцільність переходу до розподіленої схеми обчислень. Це також обмежує перелік задач, які доцільно вирішувати на розподілених системах.

Другою значною відмінністю розподілених обчислень від паралельних є питання надійності системи. Коли всі обчислення виконуються в межах однієї машини, забезпечення надійності її роботи або відновлення результатів її роботи у разі помилки є достатньо прямолінійним – можна додати джерело безперебійного живлення для коректного завершення роботи під час відключення електроенергії, додати резервні блок живлення та жорсткий диск для надійного збереження, та інше. Проте якщо таких машин одночасно працюють десятки або більше, ми вже не можемо зафіксувати єдиний стан системи в будь-який момент часу, аби повернутись до нього в разі краху. Ба більше, починаючи з певної кількості вузлів системи, ми взагалі не можемо сподіватись на їх коректну роботу впродовж усього виконання обчислень. Незважаючи на низьку ймовірність відмови одиничних елементів системи, на великих масштабах, помилки будуть траплятись часто і непередбачувано [4]. Єдиний спосіб запобігти цьому – заздалегідь впровадити стійкість до відмов на рівні всієї системи, маючи змогу передати обчислення, які виконувалися на вузлах, що вийшли з ладу, на інші, вцілілі вузли. Також розподілені системи мають бути стійкими до помилок комунікації між вузлами, загубленим чи пошкодженим повідомленням.

Усі ці проблеми значно ускладнюють реалізацію розробниками розподілених алгоритмів «із нуля». Кожна з перелічених проблем потребує глибокої експертизи у відповідних галузях для її надійного вирішення, і зазвичай їх повторне вирішення є недоцільним навіть для команди інженерів. Відповідно, розробники потребують готових рішень для швидкої та надійної комунікації між вузлами системи. Такі рішення дозволяють розробникам сконцентруватись безпосередньо на алгоритмі обчислень, залишаючи низькорівневі та повторювані деталі реалізації готовим рішенням.

1.3 Комунікація всередині кластеру

Для комунікації всередині кластеру можна визначити два основні механізми: вузол-до-вузла, за якого вузли комунікують напряму один з одним та централізований, за якого повідомлення між вузлами кластеру агрегуються та передаються через один або декілька вузлів-брокерів. Хоча комунікація напряму є найефективнішою з точки зору затримки та пропускної здатності, існує декілька причин, через які надсилання повідомлення через виділені вузли може бути більш привабливим для розробників.

Першою такою причиною є знаходження сервісів (англ. service discoverability) – для правильного визначення, якому вузлу має бути надіслане повідомлення та за якою адресою знаходиться цей вузол, кожен вузол кластеру має або зберігати в себе актуальну модель структури кластеру, або звертатись до інших вузлів, які мають таку інформацію. Актуалізація даних про кластер на всіх вузлах є нетривіальною задачею [5], адже структура кластеру може змінюватись часто та непередбачувано. А якщо для відправки повідомлення на інші вузли все одно необхідно виконувати запити на додаткову інформацію, накладні витрати на передачу повідомлення через посередника вже не є такими суттєвими.

Другою причиною, через яку використання брокерів повідомлень у кластері може працювати краще, ніж пряме надсилання, є відмовостійкість кластеру. Якщо уся комунікація відбувається централізовано, у вузлів-брокерів в будь-який момент часу є актуальна інформація про структуру кластеру та стан його вузлів. Маючи інформацію про розподіл обчислень між вузлами та їх завантаження, вузли-посередники можуть більш ефективно розподіляти повідомлення між вузлами. А інформація про відмови окремих вузлів дозволяє перенаправляти нові повідомлення на працюючі вузли, водночас перенадсилаючи повідомлення, які один з вузлів не зміг обробити через помилку виконання або системи. За рахунок вищеперерахованих можливостей, комунікація через посередників також краще працює в динамічних середовищах, в яких конфігураця кластеру може постійно змінюватись. Обмеження прямої

комунікації між вузлами також дозволяє зменшити зв’язність всередині кластеру, спрощуючи його модифікацію та налагодження.

У межах цієї роботи ми розглянемо лише механізми централізованої комунікації, як більш універсальні та залежні від ефективного виконання операцій введення-виведення на вузлах-посередниках. Класичним прикладом механізму централізованої комунікації в масштабованому програмному забезпеченні є черга повідомлень.

1.3.1. Черга повідомлень

Черга повідомлень – один із найпоширеніших компонентів сучасних кластерів. Основна ідея черги полягає в тому, що будь-який вузол кластеру (або навіть зовнішній користувач) може додати повідомлення в чергу. Потім, у залежності від обраних налаштувань черги, повідомлення буде надіслане одному або декільком вузлам кластеру. Сучасні імплементації черг повідомлень дозволяють глибоке налаштування правил розподілу, фільтрації та пересилки повідомлень. За потреби багато таких систем можуть виконувати функції збереження, дедуплікації та повторної відправки повідомлень, дозволяючи значно покращити стійкість кластеру до відмов [4].

Варто зазначити, що в цілях підвищення надійності, багато реалізацій черг повідомлень і самі є прикладами масштабованого програмного забезпечення і можуть виконуватись одночасно на декількох вузлах кластеру. Прикладами таких систем є Amazon SQS, Apache Kafka та RabbitMQ.

Узагальнюючи, більшість імплементацій черг повідомлень можна розглядати як реалізації взірця проектування Pub-Sub (англ. Publish-Subscribe, публікація-підписка).

1.3.2. Взірець Pub-Sub

Взірець Pub-Sub – архітектурний взірець проектування, що описує спосіб комунікації між компонентами масштабованої програми. Його ідея полягає у розмежуванні відправників (англ. publisher – видавець) та підписників (англ. subscriber), аби знизити зв’язність між компонентами системи (Рис. 1.1). У взірці

Pub-Sub, відправники лише позначають клас повідомлення (в термінології черг повідомлень, вони часто позначаються як «теми» (англ. topic)), але не обирають конкретного отримувача. Після того, як брокер отримує повідомлення, він перенаправляє це повідомлення усім підписникам, які підписалися на цю тему, або за іншими правилами пересилання.

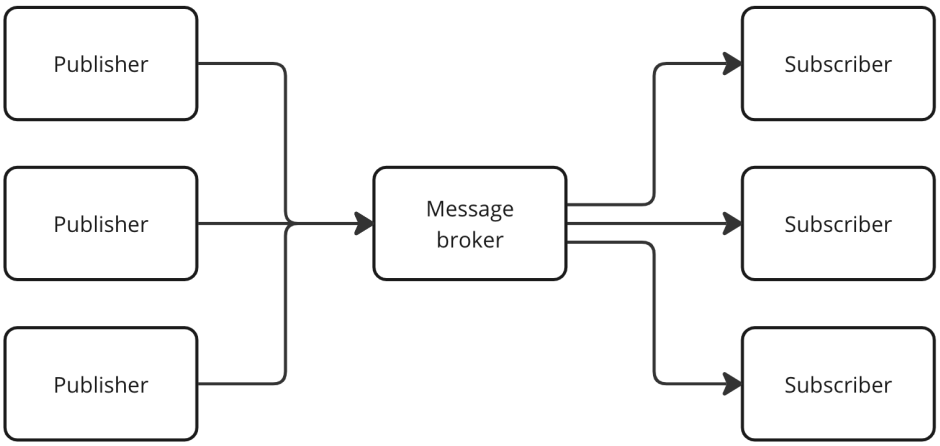

Рисунок 1.1 – Принципова діаграма взірця Pub-Sub

Цілком природньо може виникнути зауваження, що взірець Pub-Sub можна розглядати як масштабовану версію класичного взірця проектування «Спостерігач» (англ. Observer) [6]. І дійсно, за аналогією зі «Спостерігачем», підписники у Pub-Sub є спостерігачами, теми повідомень – спостереженими, а відправники і отримувачі повідомлень є логічно розділеними одні від одних.

Оскільки в розподіленій системі обробкою повідомлення буде займатись окремий вузол кластеру, який є підписником відповідної теми, взірець Pub-Sub позбувається одного з важливих недоліків взірця «Спостерігач» – обмеження на складність коду обробника. У класичній однопотоковій імплементації взірця «Спостерігач», код обробки повідомлень кожного зі спостерігачів зазвичай послідовно виконується в тому ж потоці, що і відправник повідомлення. Це вимагає строгого контролю зі сторони розробника за кількістю спостерігачів та складністю коду обробки повідомлення кожного з них. Іншим вирішенням цієї проблеми може бути побудова окремої багатопотокової реалізації цього взірця і

коректне її застосування. За своєю природою, у взірці Pub-Sub немає цього недоліку – брокеру достатньо надіслати копії повідомлення усім підписникам, що є відносно низькозатратною операцією, і він не очікує виконання обробки цих повідомлень і не блокує подальше виконання коду їх відправника.

Деякі реалізації взірця Pub-Sub також дозволяють налаштувати розподіл повідомлень порівну між підписниками, замість розсилки копій усіх повідомлень усім підписникам. Це дозволяє також використовувати черги повідомлень і для розподілу задач між вузлами кластеру. Проте, у своїй базовій версії, взірець Pub-Sub має обмеження, важливе саме для розподілених систем – відсутність можливості отримати результат обробки повідомлення, що дуже часто є необхідним для об’єднання результатів обчислень або повернення користувачам результатів їх запитів. Для цих цілей на основі взірця Pub-Sub було створено його розширену версію – взірець Return Address (англ. «зворотня адреса») [7]

1.3.3. Взірець Return Address

Цей взірець дозволяє використати звичні черги повідомлень, які не надають інших можливостей крім як односторонньої відправки повідомлень, для повноцінної двосторонньої комунікації між вузлами кластеру. Ідея взірцю є достатньо простою і елегантною – оскільки кожна черга повідомлень дозволяє агрегацію повідомлень у велику кількість віртуальних черг (тем), ми можемо створити окрему тему для кожного вузла, який хоче отримувати відповіді. Тоді, включивши у кожне повідомлення зворотну адресу у вигляді теми, за якою треба надіслати відповідь, будь-який вузол, що отримав повідомлення, може надіслати відповідь, скориставшись тією ж самою чергою повідомлень. Через поширеність та універсальність цього взірця він був включений у велику кількість реалізацій черг повідомлень.

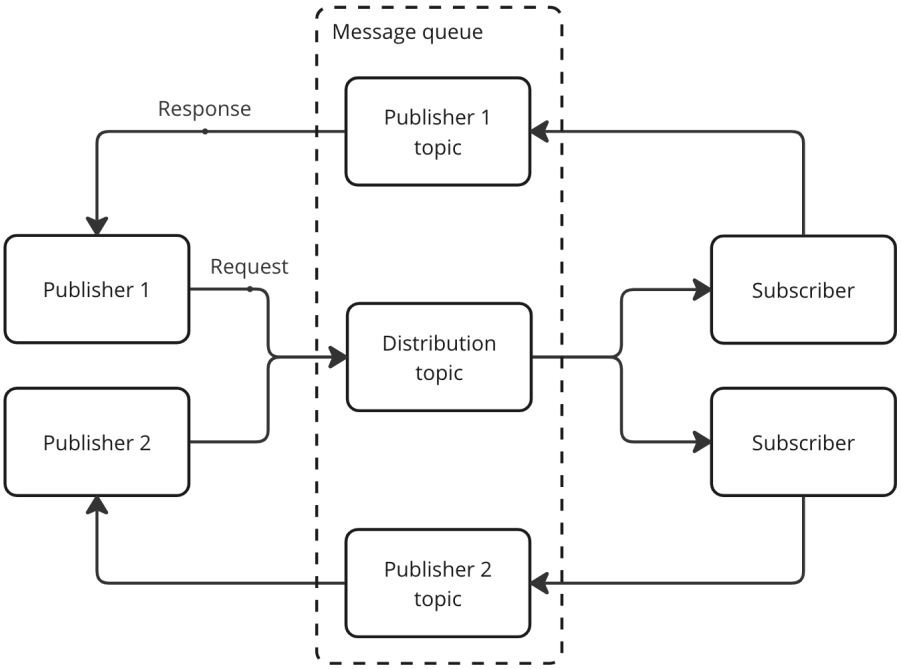

Рисунок 1.2 – Принципова діаграма взірця Return Address

Цей взірець дозволяє побудувати всередині кластеру надійну систему обміну повідомленнями між вузлами. За рахунок механізмів розподілу та перевідправки повідомлень усередині черги, така комунікація є надійнішою ніж комунікація напряму, а розділення відправника і отримувача дозволяє відправнику повідомлення не зважати на внутрішню структуру кластеру та її зміни, адже вся пряма комунікація відбувається лише із чергою повідомлень.

1.4 Механізми комунікації, досліджені в цій роботі

У цій роботі було приділено увагу централізованим механізмам комунікації у кластері та впливу багатопотокових моделей класичних взірців проектування на таку комунікацію. В якості модельної задачі було розроблено власну реалізацію черги повідомлень, яка імплементує взірець Return Address. Цю реалізацію було досліджено та протестовано у порівнянні зі стандартними популярними рішеннями.

РОЗДІЛ 2: БАГАТОПОТОКОВІ МОДЕЛІ ВЗІРЦІВ ПРОЕКТУВАННЯ У МАСШТАБОВАНОМУ ПРОГРАМНОМУ ЗАБЕЗПЕЧЕННІ

2.1 Проблема ефективної комунікації

Використання посередника для комунікації між вузлами кластеру дозволяє розробникам перенести на цього посередника декілька важливих проблем і зфокусуватись на розробці безпосередньо програмного забезпечення. Черга повідомлень у якості посередника може покращити стійкість системи до відмов, зробити її більш гнучкою та відкритою до змін. Проте це, у свою чергу, висуває високі вимоги до якості імплементації цього посередника. Адже, якщо черга повідомлень буде неефективно або ненадійно виконувати свої функції, це ставить під загрозу працездатність усієї системи та нівелює усі переваги її використання.

Для ефективної та надійної роботи, черга повідомлень має максимально ефективно утилізувати системні ресурси, такі як мережеві або доменні сокети, файлову систему та інші. Класичні імплементації черг працюють за ідіомою store-and-forward (англ. зберегти та перевідправити), відомою із телекомунікаційного середовища. Оскільки черга має отримати, зберегти, обробити, а після цього відправити повідомлення, оптимізація її роботи може включати оптимізації роботи із системними викликами, виділення пам’яті, використовуваних структур даних та інших аспектів імплементації.

Правильний вибір механізму доступу до файлової системи та мережі та відповідних системних викликів може мати істотний вплив на швидкодію черги повідомлень та доцільність її використання. У цій роботі розглянуто новітні системні виклики ОС Linux для доступу до файлової системи та їх застосування у багатопотоковій моделі взірця проектування «Проактор».

2.2 Обмеження класичних механізмів асинхронного введення-виведення

У минулорічній роботі, на прикладі модельної задачі «Веб-сервер», було продемонстровано переваги асинхронного доступу до файлової системи над синхронним для програмного забезпечення, що значною мірою залежить на доступ до файлової системи або мережі. Проте остання імплементація цієї задачі все ще не змогла наздогнати еталонну, висвітлюючи наявність потенційних оптимізацій.

Минулого року, в якості механізму асинхронного доступу до файлової системи, було використано перевірену часом бібліотеку ОС Linux libaio. На відміну від крос-платформених POSIX-сумісних імплементацій, вона надавала ширший доступ до механізмів очікування на настання асинхронних подій, і дозволяла отримувати результати виконання операцій над файловою системою із мінімальною кількістю очікувань. Проте ця система також мала кілька недоліків. Наприклад, вона вимагала явної передачі вказівників на всі запити, відповідь на які ми очікуємо, що вимагає додаткових виділень пам’яті та зберігання усіх запитів у оперативній пам’яті. Також ця бібліотека надає асинхронний доступ лише для операцій читання та запису, унеможливлюючи лише її використання для побудови програм, що працюють із мережею, адже для роботи із підключеннями клієнтів, як мінімум, необхідно виконувати ще й системний виклик accept(), який синхронно очікує на нове з’єднання від клієнта і є блокуючим.

Необхідність суміщати блокуючі та неблокуючі операції у програмі відчутно підвищує її складність та зменшує переваги, що надаються асинхронними операціями. Так, для того, аби очікування нових клієнтів не блокувало виконання програми та обробку запитів інших клієнтів, операція accept() має виконуватись у виділеному потоці, накладаючи додаткові витрати на перемикання контекстів. А необхідність зберігати всі запити, які можуть надходити із декількох потоків, для їхнього очікування, вимагає використання

потокобезпечних реалізацій черги та інших структур даних, які часто є менш ефективними ніж їх стандартні аналоги.

Також у 2018 році, після відкриття вразливостей у механізмах спекулятивного виконання більшості наявних центральних процесорів Spectre та Meltdown, гостро постало питання зменшення залежності програм користувацького середовища (особливо, тих, що інтенсивно працюють із механізмами введення-виведення) на системні виклики ядра, на які цілковито покладалась бібліотека libaio. Механізм KPTI (англ. Kernel Page Table Isolation, ізоляція таблиць сторінок в ядрі), який забезпечує захист від цих вразливостей, значною мірою погіршив швидкодію системних викликів операційної системи, сповільнивши їх до 30% [8].

З огляду на вищеперераховані обмеження, починаючи із 2019-го року, розробником Дженсом Аксбо з компанії Meta ведеться розробка абсолютно нового механізму виконання асинхронних операцій введення-виведення – io_uring.

2.3 Підсистема io_uring операційної системи Linux

Io_uring – новітній механізм введення-виведення в операційній системі Linux, фіналізація стандарту якого завершилась лише в середині 2022-го року. Під час його проектування за мету було поставлене вирішення наступних проблем класичних варіантів асинхронного введення-виведення:

- Поточні механізми підтримують лише вкрай обмежений набір операцій (читання та запис) і не можуть використовуватись самостійно для переважної більшості програм
- Поточні механізми є не дуже ефективними через необхідність виконання великої кількості дороговартісних з точки зору швидкодії системних викликів
- Поточні механізми є незручними для використання розробниками через великі об’єми типового коду, необхідного для їх підтримки

Для вирішення цих проблем було запропоновано повністю новий спосіб комунікації між програмами користувацького середовища та ядром без застосування системних викликів.

2.3.1. Архітектура механізму io_uring

Перш за все перед розробником механізму io_uring постало питання зменшення кількості системних викликів, необхідних для виконання операцій асинхронного введення-виведення. Для цього було вирішено застосовувати спільні ділянки пам’яті, які поділятимуть програми користувацького середовища і ядро ОС. За рахунок використання цих ділянок пам’яті для збереження потокобезпечних черг, що зберігають запити і відповіді на них, надається можливість робити запити на операції введення-виведення взагалі без використання системних викликів (Рисунок 2.1).

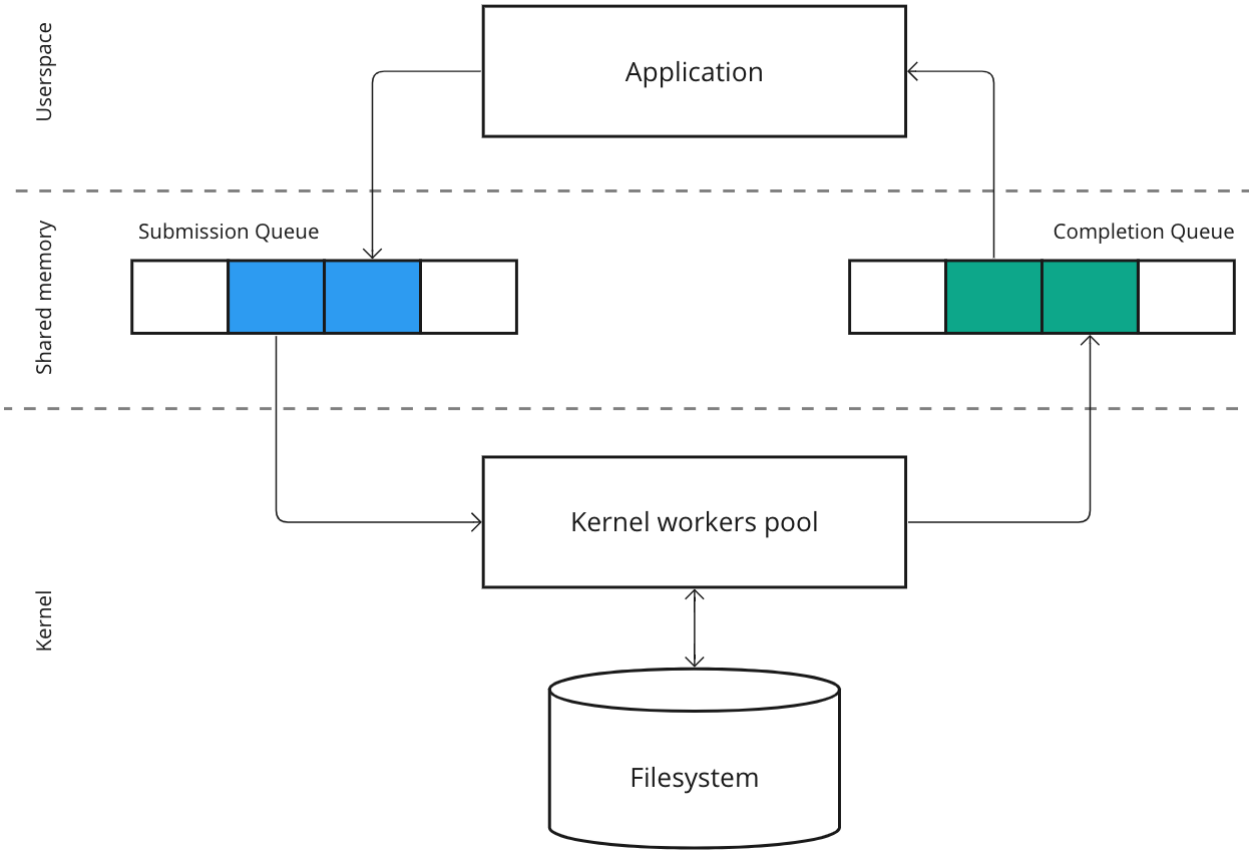

Рисунок 2.1 – Схема взаємодії програми користувацького середовища і ядра ОС через кільцеві черги

У якості структури даних для обміну запитами використовується двостороння кільцева черга. У спільній ділянці пам’яті створюються дві черги, одна для запитів (англ. submission queue), інша – для результатів (англ. completion queue). Оскільки програма має додавати елементи лише у чергу запитів, а ядро – лише у чергу результатів, це теж дозволяє спростити багатопотокову синхронізацію між ядром та користувацьким середовищем. Можна помітити, що такий спосіб взаємодії дещо обмежує очікування результатів асинхронних операцій в порівнянні із класичним механізмом libaio. Так, оскільки із черги результатів за один раз можна дістати лише перший елемент, у розробників більше немає можливості стати в очікування за виконанням конкретного запиту – вони мають обробляти усі результати так швидко, як вони надходять. Це обмеження, хоча і може бути помітним, зазвичай, не є суттєвим, адже, зрештою, всі запити, які програма надсилає до системи введення-виведення мають бути оброблені. Перехід до нової моделі очікування на результати операцій може потребувати незначної адаптації зі сторони розробника, але така зміна дозволяє відчутно спростити як внутрішні механізми очікування, так і саму програму, адже такий механізм синхронізації дозволяє не зберігати відправлені запити на стороні програми.

2.3.1.1 Виконання операцій асинхронного введення-виведення на боці ядра ОС

Після того, як запит на асинхронну операцію було передано ядру ОС, він має бути виконаний операційною системою. Для цього на стороні ядра імплементовано взірець «пул потоків», які і виконують операції введення-виведення. Для різних видів операцій існує два ліміти на кількість одночасних потоків. Для операцій, результат виконання яких (позитивний або негативний) точно існує, таких, як читання з диску, кількість робочих потоків обмежена кількістю наявних ядер процесора. Для операцій, результат яких очікується, але не гарантується, таких, як операції із мережевими сокетами, кількість робочих потоків обмежується лише лімітами операційної системи. Для уникнення

неконтрольованого створення потоків на операції, які можуть не виконатись, робочі потоки, за можливості, використовують операції неблокуючого введення-виведення poll() та select(). Кількість робочих потоків автоматично змінюється в залежності від кількості наявних запитів, і ця поведінка може налаштовуватись користувачем в залежності від потреб конкретної програми.

2.4 Застосування liburing для асинхронного введення-виведення

Оскільки io_uring – це підсистема, яка все ще знаходиться в активній розробці, але вже входить у комплект поставки ОС Linux, її розробка обмежена жорстким правилом “never break userspace” (англ. «ніколи не ламати середовище користувача»), встановленим головним мейнтейнером Лінусом Торвальдсом [9]. Це правило забороняє робити будь-які ламаючі зміни до інтерфейсу ядра у користувацькому середовищі після того, як оновлення інтерфейсу були доєднані до головної гілки репозиторію Linux. Це правило значно ускладнює роботу із API у розробці, адже якщо старий функціонал буде гарантовано працювати, як очікується, додавання нового функціоналу, особливо до вже існуючих системних викликів, часто вимагає або створення нових системних викликів, або передачі додаткових параметрів виклику у допоміжній структурі даних. Через це, для спрощення використання певних API ядра, створюються бібліотеки-обгортки над системними викликами операційної системи.

Для io_uring такою бібліотекою є liburing – вона є зручним і рекомендованим способом взаємодії з підсистемою io_uring. Вона надає зручний та уніфікований набір функцій для всіх видів операцій введення-виведення та очікування на результати цих операцій. Робота із цією бібліотекою на прикладі простого читання файлу відбувається наступним чином:

1. За допомогою функції io_uring_queue_init() ініціалізуються спільна ділянка пам’яті та черги запитів і результатів. Розробник має можливість вказати необхідний розмір обох черг в залежності від потреб застосунку;

2. За допомогою функції io_uring_get_sqe() із черги запитів дістається вказівник вільний запис, в який можна помістити запит на асинхронну операцію введення-виведення;
3. За необхідності, за допомогою функції io_uring_sqe_set_data() до запиту можна прикріпити довільний вказівник. Після виконання запиту цей вказівник буде повернений разом із результатом операції. Цей вказівник можна використати для збереження додаткових даних про операцію (наприклад, тип виконаної операції або дескриптор файлу, над яким ця операція виконується);
4. За допомогою функції io_uring_prep_read(), передавши в якості аргументів вказівник на запис у черзі та стандартні аргументи системного виклику read(), готується запит на читання із файлової системи;
5. Усі запити, додані в чергу, відправляються на виконання викликом функції io_uring_submit();
6. Коли програма має отримати результат виконання цієї операції, можна викликати одну із функцій очікування, наприклад, io_uring_wait_cqe(). Ця функція є блокуючою і повертає результат виконання найшвидшого запиту з черги.

Як можна побачити, створення та обробка запиту на виконання асинхронної операції введення-виведення з допомогою liburing є значно більш багатослівною ніж синхронний системний виклик read(). Проте, у порівнянні зі старим API libaio, liburing є значно простішим у розумінні та імплементації та є більш універсальним.

2.5 Адаптація моделі взірця «Проактор» до роботи із io_uring

Оскільки бібліотека liburing, як і libaio, надає можливість справжнього асинхронного виконання операцій введення виведення (на відміну від механізму poll() / select(), який є блокуючим, але мінімізує кількість виконань), вона може бути застосована для вдосконалення імплементації взірця «Проактор» (цитата проактор). У такому разі у ролі виконавця асинхронних подій виступатиме підсистема io_uring, а проактивний ініціатор, обробник та диспетчер виконання знаходитимуться на стороні програми користувацького середовища. За рахунок об'єднання виконавця асинхронних подій та проактивного ініціатора із диспетчером виконання через спільну ділянку пам'яті, реалізація проактора на основі io_uring може стати кращою з точки зори швидкодії, аніж попередня на основі libaio.

Для реалізації взірця «Проактор» на основі io_uring, можна скористатись і бібліотекою liburing напряму, але таке рішення буде достатньо багатослівним і важким для розуміння. Для роботи із проактором, розробники потребують зрозумілих механізмів ініціації і обробки асинхронних подій, і тому рішення, яке дозволило б інженером скористатись перевагами механізму io_uring, але приховало б деталі імплементації, було б значно легшим для розуміння та використання. Для досягнення цих цілей та полегшення розробки власної реалізації «Проактору» та модельних задач на його основі, було вирішено побудувати об'єктно-орієнтовану абстракіцю над io_uring – бібліотеку IOUring.

2.5.1. Побудова об'єктно-орієнтованої абстракції над механізмом io_uring

Через новизну механізму io_uring, сталі інструменти розробки на його основі практично відсутні. Отже, бібліотеки, які могли б спростити роботу із ним, є вкрай необхідними для популяризації цього механізму та спрощення переведення програм, що працюють із класичними механізмами асинхронного введення-виведення, на новіший та більш ефективний io_uring.

Перед розробкою об'єктно-орієнтованої абстракції над бібліотекою liburing, до такої абстракції були висунуті наступні вимоги:

- Усунення необхідності ручного використання системних викликів та функцій бібліотеки liburing – майбутня абстракція над liburing має бути самодостатнім механізмом роботи із підсистемою io_uring;
- Мінімізація можливих помилок використання, пов'язаних із ініціалізацією та знищенням ресурсів io_uring, та буферів введення-виведення;
- Універсальність майбутньої абстракції – вона має бути придатною як для імплементації взірця «Проактор», так і для використання у якості простого механізму асинхронного введення-виведення;

Відповідно до цих вимог було спроектовано два інтерфейси для роботи із io_uring: IIOUringRequestHandler та IIOUringEventHandler для надсилання запитів та обробки результатів асинхронних запитів, відповідно. На основі цих інтерфейсів було побудовано основний клас бібліотеки – IOUring.

```
class IIOUringRequestHandler
{
public:
  virtual ~IIOUringRequestHandler();

  void requestRead(const int32_t fileDescriptor, const uint32_t length,
      const uint64_t offset, void * const tag = nullptr);
  void requestWrite(const int32_t fileDescriptor, const uint8_t * const
      buffer, const uint32_t length, const uint64_t offset, void * const
      tag = nullptr);
  void requestAccept(const int32_t fileDescriptor, sockaddr * addr,
      socklen_t * addrlen, void * const tag = nullptr);
  void requestClose(const int32_t fileDescriptor, void * const tag =
      nullptr);
  void requestConnect(const int32_t fileDescriptor, const sockaddr *
      addr, socklen_t addrlen, void * const tag = nullptr);
};
```

Приклад 2.1 – Публічна частина інтерфейсу IIOUringRequestHandler

Інтерфейс IIOUringRequestHandler визначає клас, який застосовується для відправки запитів на асинхронні події до io_uring. Для кожної операції (читання,

запис, очікування на клієнта, закриття файлу, під'єднання до сокету) визначений відповідний метод, список аргументів якого співпадає зі списком аргументів відповідного методу бібліотеки liburing для мінімізації кількості адаптуючого коду всередині класу IOUring та спрощення роботи із бібліотекою для розробників, які вже мали досвід роботи із liburing.

Другий інтерфейс бібліотеки – IIOUringEventHandler призначений для імплементації користувачами бібліотеки. Він містить методи для обробки результатів асинхронних операцій кожного з видів. Екземпляр цього інтерфейсу передається у конструктор класу IOUring, і методи цього екземпляру будуть виконуватись кожного разу, коли io_uring надає результат виконання однієї із поставлених в чергу операцій.

```
class IIOUringEventHandler
{
public:
  virtual ~IIOUringEventHandler();

  void init(IIOUringRequestHandler* ring);
  void onRead(IIOUringRequestHandler* ring, int32_t fileDescriptor,
    uint8_t * buffer, int32_t length, int64_t offset, void * tag);
  void onWrite(IIOUringRequestHandler* ring, int32_t fileDescriptor,
    int32_t length, int64_t offset, void * tag);
  void onAccept(IIOUringRequestHandler* ring, int32_t fileDescriptor,
    int32_t newFileDescriptor, void * tag);
  void onConnect(IIOUringRequestHandler* ring, int32_t fileDescriptor,
    int32_t result, void * tag);
};
```

Приклад 2.2 – Публічна частина інтерфейсу IIOUringEventHandler

В якості аргументу до кожного з методів обробки передається результат відповідної операції та вказівник на екземпляр інтерфейсу IIOUringRequestHandler. Це дозволяє під час обробки будь-якої з операцій додати в чергу io_uring нові запити, тим самим використовуючи io_uring у якості циклу подій. Можливість надсилати нові запити до системи введення-виведення у відповідь на результати попередніх запитів та збереження внутрішнього стану у

реалізації інтерфейсу IIOUringEventHandler дозволяє створити таку імплементацію цього інтерфейсу, яка водночас стане реалізацією взірця проектування «Проактор». Клас IOUring, в свою чергу, реалізовує інтерфейс IIOUringRequestHandler та взаємодіє із бібліотекою liburing для відправки та обробки запитів на асинхронні операції введення-виведення. Для спрощення розробки, відлагодження та використання бібліотеки, усі її класи проектувались із застосуванням ідіоми NVI (англ. Non-Virtual Interface, невіртуальний інтерфейс).

2.5.1.1 Ініціалізація та знищення структур io_uring

Для роботи із підсистемою io_uring, необхідно ініціалізувати спільну ділянку пам’яті та відповідні структури даних, для чого бібліотека liburing надає системний виклик io_uring_queue_init(). Так само для коректного завершення роботи із підсистемою, необхідно знищити виділені ресурси за допомогою відповідної функції io_uring_queue_exit(). Перша функція приймає вказівник на об’єкт структури io_uring, який використовується у практично всіх подальших функціях бібліотеки liburing.

Щоб спростити ініціалізацію та знищення об’єктно-орієнтованої абстракції над io_uring, було вирішено застосувати в ній ідіому RIAA (англ. Resource Acquisition is Initialization). За цією ідіомою, у конструкторі класу IOUring викликається функція io_uring_queue_init(), а в деструкторі – io_uring_queue_exit(). За рахунок цього, розробникам не потрібно виконувати ручної ініціалізації, а знищення буде гарантовано виконане після знищення відповідного об’єкту. Також, після ініціалізації, об’єкт класу IOUring зберігає у приватному полі об’єкт структури io_uring, який потім додається до всіх викликів функцій бібліотеки liburing, знімаючи ще один обов’язок із розробників, що користуються цією підсистемою.

2.5.1.2 Надсилання запитів на асинхронні операції введення-виведення

Для відправки запитів до підсистеми liburing, клас IOUring надає відповідні методи request*(). Список аргументів кожного із цих методів значною мірою повторює список аргументів відповідних функцій бібліотеки liburing. Реалізація кожного із цих методів використовує можливість додавання довільного вказівника data до кожного запиту до підсистеми io_uring. До кожного запиту додається в якості вказівника data додається вказівник на об’єкт внутрішньої структури IOUring::UserData. Об’єкт цієї структури містить тип виконуваної операції для подальшого визначення її обробника, дескриптор файлу, над яким виконується ця операція, та адреса буфера та зміщення для операцій, що оперують із буферами введення-виведення.

Також, за аналогією із бібліотекою liburing, усі методи класу IOUring та, відповідно, структура UserData надають можливість користувачу бібліотеки додати до запиту до файлової системи довільний вказівник tag. Цей вказівник можна застосувати для прикріплення до запиту певного об’єкту, який потрібен для обробки результату цього запиту, або у якості восьмибайтової (на 64-бітних системах) змінної для збереження, наприклад, бітового поля.

```
struct UserData {
  RequestType requestType;
  int32_t fileDescriptor;
  uint8_t * buffer;
  uint64_t offset;
  void * tag;
};
```

Приклад 2.3 – Заголовок структури IOUring::UserData

Для операцій читання, метод requestRead створює у пам’яті буфер відповідного розміру, який буде видалений після обробки результату відповідного запиту.

Після додавання вказівника data до запиту, запит одразу відправляється на виконання за допомогою функції io_uring_submit().

2.5.1.3 Отримання та обробка результатів виконання асинхронних операцій

Після того, як запити на операції введення-виведення були відправлені на обробку, користувач бібліотеки в необхідний момент часу може стати в очікування на результати. Для цього, клас IOUring має метод waitAndProcessOne(). Цей метод є блокуючим і очікує на отримання результату виконання будь-якої із відправлених операцій від підсистеми io_uring за допомогою функції io_uring_wait_cqe(). Після отримання об’єкту структури даних io_uring_cqe (англ. CQE – Completion Queue Entry, елемент черги виконання), він передається на опрацювання приватному методу processCompletionEvent(). Цей метод залежно від типу виконаної операції, отриманої із відповідного об’єкту структури UserData, викликає відповідний метод наданої в конструкторі класу IOUring імплементації інтерфейсу IIOUringEventHandler. Після того як код відповідного обробника було виконано, видаляються буфери пам’яті, виділені класом IOUring та відповідний об’єкт структури UserData.

2.5.1.4 Використання декількох обробників подій із одним екземпляром IOUring

Для програм, які виконують декілька функцій введення-виведення одночасно (наприклад, файловий сервер, що обслуговує клієнтів одночасно за декількома протоколами), може бути недоцільним створення одночасно декількох екземплярів IOUring. Робота із декількома чергами запитів ускладнює очікування на результати роботи кожної з них та може вимагати створення додаткових потоків та ускладнення роботи між ними. Для спрощення роботи із бібліотекою до неї було додано класи IOUringRequestHandlerTagProxy та TagSplitterProxyIOUringEventHandler (Рисунок 2.2).

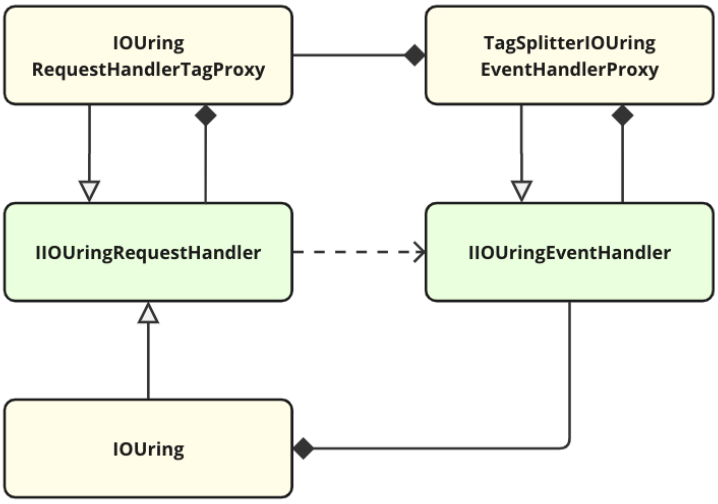

Рисунок 2.2 – Діаграма класів бібліотеки IOUring

Ці класи імплементують взірець «Проксі» для інтерфейсів IIOUringRequestHandler та IIOUringEventHandler відповідно. Використовуючи додатковий атрибут tag, котрий приймають методи класу IOUring, ці класи приймають запити від декількох обробників подій та конвертують усі надіслані ними теги в унікальні, зберігаючи відображення між отриманими та надісланими тегами. Потім це відображення використовується у зворотному напрямку, передаючи обробникам відповіді саме на ті запити, які надсилались відповідними обробниками.

Такий підхід дозволяє використовувати одразу декілька обробників подій із одним екземпляром класу IOUring без жодної модифікації самих обробників. Таким чином спрощується робота із асинхронними подіями зі сторони користувачів бібліотеки, адже їм необхідно очікувати на події лише з одного джерела, що особливо важливо у випадку очікування результатів запитів, які можуть не виконатись (наприклад, під’єднання нового клієнта чи отримання даних з мережі).

2.5.2. Перспектива застосування std::future для отримання результатів асинхронних операцій

Хоча реалізований інтерфейс роботи із бібліотекою і є достатньо зручним для використання та універсальним, потенційне використання стандартних методів взаємодії із асинхронними подіями зменшило б поріх входження до застосування цієї бібліотеки, спростило роботу із нею та покращило б її інтеграцію в існуючі програми. Можливість надсилання запиту на асинхронну операцію та отримання відповіді на цей запит натякає на потенційне застосування механізмів std::promise та std::future зі стандартної бібліотеки мови C++ для опису стандартного та універсального способу виконання асинхронних операцій введення-виведення.

На жаль, хоча такий варіант і вбачається найбільш очевидним для реалізації подібної бібліотеки, поточний стан класів std::promise та std::future не дозволяє їх повноцінне використання. Справа в тому, що з моменту написання минулої роботи актуальна імплементація цих класів все ще не дозволяє очікування на будь-який Future із певної їх множини, а лише на усю певну множину або на один Future. Ця поведінка є дуже важливою для програм, що працюють, наприклад, із мережевими з’єднаннями. Оскільки певні з подій, на які відбувається очікування, можуть ніколи не відбутися, вимога очікування на всі події унеможливлює їх ефективне застосування, змушуючи розробників або створювати окремі потоки на кожну із подій (що повністю нівелює доцільність взагалі працювати із асинхронним введенням-виведенням), або виконувати неефективне очікування на події із таймером. Проблема очікування із таймером полягає в тому, що навіть якщо якісь із очікуваних подій виконались під час очікування на іншу довготривалу подію, програма зможе обробити їх результат лише по завершенню таймеру. Виставлення таймеру на довші проміжки збільшує загальний час неефективного простою програми, а зменшення проміжку збільшує обсяг нерезультативних обчислень, забираючи ресурси центрального процесору від корисних обчислень. Минулорічна робота продемонструвала неможливість

заміни повноцінних механізмів часткового очікування іншими доступними механізмами синхронізації без значної втрати швидкодії програми.

Комітет стандартів C++ не може узгодити стандартний спосіб роботи із класами std::promise та std::future. Пропозицію до стандарту, що розглядалася минулого року (P0443R12), було відхилено, а замість неї на розгляді зараз перебуває нова зі схожими ідеями – P2300R6 [10]. Доля цієї пропозиції перебуває під питанням, і навіть якщо її приймуть до стандарту найближчим часом, доведеться чекати ще деякий час, поки вона буде доступною до використання у стабільних версіях популярних компіляторів мови C++. Альтернативою могла би стати реалізація бібліотеки на основі сторонніх реалізацій Future (наприклад, із бібліотеки Boost), але це накладає додаткові залежності на користувачів такої бібліотеки і ускладнює її використання в інших програмах.

РОЗДІЛ 3: РЕАЛІЗАЦІЯ МАСШТАБОВАНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ НА ОСНОВІ ОТРИМАНИХ МОДЕЛЕЙ

3.1 Порівняння моделей взірця «Проактор» на основі libaio та IOUring

У минулорічній роботі нами було продемонстровано модель взірця «Проактор» на основі класичної бібліотеки для асинхронного введення-виведення libaio. І хоча вона показала себе непогано у порівнянні з попередньою реалізацією на основі емуляції асинхронної роботи із файловою системою, під час її використання у модельній задачі «HTTP-сервер», швидкодія цієї моделі виявилась недостатньою для досягнення результатів, продемонстрованих еталонною реалізацією HTTP-сервера – програмного забезпечення nginx. Для визначення базового рівня швидкодії моделі взірця «Проактор» на основі механізму io_uring та доцільності її застосування, було вирішено перенести задачу «HTTP-сервер» на новий механізм виконання операцій асинхронного введення-виведення.

3.1.1. Реалізація HTTP-серверу на основі IOUring

Поточна реалізація класу IOUring надає доступ до усіх операцій, необхідних для реалізації HTTP-серверу: закриття файлу, очікування на під’єднання клієнта, читання і запис у файли та сокети. Єдині дві файлові операції, які мають виконуватись синхронно, бо не мають реалізації у бібліотеці liburing – це відкриття файлу (системний виклик open()) та отримання його розміру (системний виклик fstat()).

Через значну зміну механізмів введення виведення було прийняте рішення розробляти HTTP-сервер «з нуля». Основну логіку серверу було поміщено у клас HttpServerIOUringEventHandler. Він є нащадком класу IIOUringEventHandler і реалізує взірець «Проактор» на основі бібліотеки IOUring. Черги запитів та відповідей механізму io_uring використовуються у ролі черги подій, на основі якої побудований замкнений цикл серверу. Відсутність необхідності реалізації та використання власної потокобезпечної черги спрощує структуру та може

покращити швидкодію програми. Механізм io_uring дозволяє асинхронне виконання практично всіх операцій, необхідних для роботи HTTP-серверу, мінімізуючи час неефективного очікування головного потоку (Рисунок 3.1).

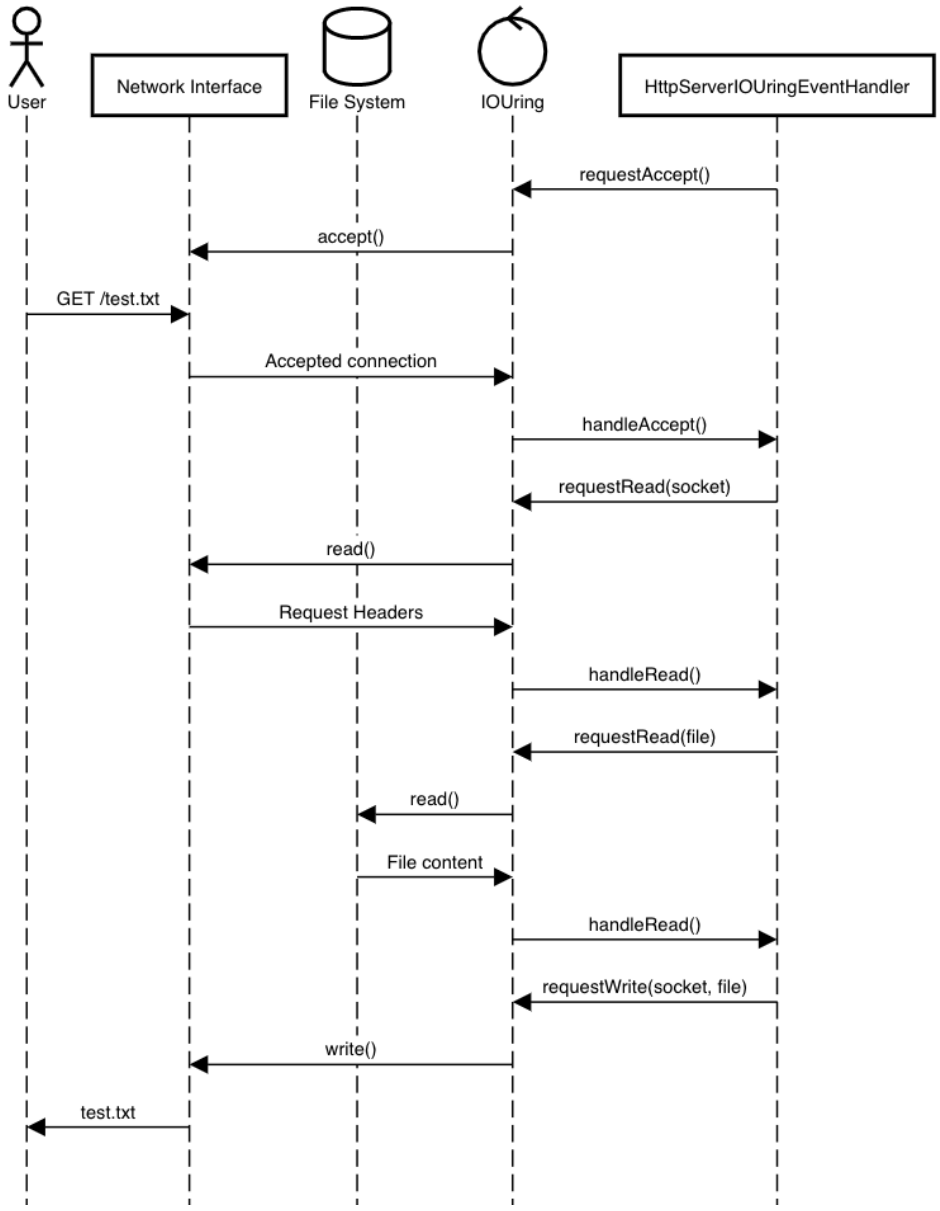

Рисунок 3.1 – Діаграма виконання HTTP-запиту з використанням IOUring

Для порівняння швидкодії старого та нового підходів до побудови HTTP-серверу, на основі IOUring було реалізовано ідентичну до минулорічної частину протоколу HTTP – сервер підтримує лише GET-запити на отримання файлів. Реалізація інших методів протоколу HTTP потребує додаткового коду, але не має жодним чином впливати на швидкодію GET-запитів, адже логіка для різних

HTTP-методів була б виокремлена в окремі компоненти коду та не виконувалась би для GET-запитів. Для обробки заголовків запитів, що надходять від користувачів, було розроблено клас HeadersBuilder. Оскільки довжина отримуваних від клієнта заголовків є ненормованою та різниться від запиту до запиту, у екземпляр класу HeaderBuilder можна передавати необроблені двійкові буфери, отримані від клієнта. На основі змісту цих буферів, HeadersBuilder автоматично визначає, чи всі частини заголовків були отримані від клієнта, і після отримання останньої частини, створює екземпляр структури Headers, яка включає в себе усі отримані від клієнту заголовки запиту. Після перевірки коректності роботи нової реалізації HTTP-серверу, була виконана перевірка її швидкодії в порівнянні з минулорічною та еталонною реалізаціями.

3.1.2. Вимірювання швидкодії HTTP-серверу

В якості еталонної реалізації було обрано веб-сервер nginx. Використовуючи всередині взірець «Реактор» та асинхронні операції введення-виведення, nginx є високоефективною та перевіреною часом імплементацією HTTP-серверу. Із долею використання у 22% [11] серед усіх веб-серверів, nginx на момент написання роботи є найпопулярнішим веб-сервером у світі. В якості конкретної версії було обрано 1.22.1 – версію, що входить до комплекту поставки останньої версії операційної системи Manjaro Linux, яка і використовувалась для розробки та тестування.

Для порівняння швидкодії реалізацій HTTP-серверів, було використано комп’ютер наступної конфігурації:

- Операційна система: Manjaro Linux, ядро версії 6.1.26
- Процесор: Intel Core i5-8500, 6 фізичних ядер, частота 4.1 ГГц
- Оперативна пам’ять: 16 ГБ DDR4, 2400 МГц
- Сховище: 512 ГБ nVME SSD, протокол PCIe 3.0 x4

Операційна система Manjaro Linux заснована на дистрибутиві Arch Linux, який пропонує підхід до оновлень під назвою rolling release (англ. їдучий реліз). Ідея цього підходу полягає у постійному оновлені всіх компонентів системи,

включаючи ядро ОС, без прив’язки до якихось сталих версій ядра або інших системних пакетів. Цей підхід, хоча і дещо підвищує ризик несумісності між певними компонентами операційної системи, дозволяє використовувати останні напрацювання операційної системи. Так, розробка практичної частини роботи виконувалась на ядрі Linux версії 6.1, у той час як у репозиторії Debian ця версія ядра стане доступною лише із релізом 10 червня. Під час роботи із компонентами операційної системи, які знаходяться в активній розробці, таких як io_uring, можливість використання останньої доступної їх версії дозволяє мінімізувати ризик натрапляння на помилку в їх реалізації або на застарілу неефективну їх версію.

У налаштуваннях веб-серверу nginx було відключено кешування, оскільки воно ускладнює тестування швидкодії базових механізмів введення-виведення. За увімекненого кешування, відповіді на запити можуть діставатись зі значно швидшої оперативної пам’яті, спричиняючи неінформативні результати тестування швидкодії введення-виведення.

Для тестування швидкодії HTTP-серверу у високонавантажених сценаріях, було протестовано швидкість усіх трьох імплементацій веб-серверів на передачі файлів розміром 4, 25 та 100 КБ в залежності від кількості одночасно підключених клієнтів. В якості метрики швидкодії було обрано кількість оброблюваних запитів на секунду – ця метрика дозволяє оцінити сукупну швидкодію механізмів мережевої комунікації, роботи із файловою системою та обробки запитів користувачів. Для тестування швидкодії було використано програму ApacheBench. З її допомогою можна налаштовувати бажану кількість одночасно підключених користувачів та кількість запитів, які необхідно використати. По завершенню, ApacheBench розраховує середню кількість запитів на одиницю часу та підраховує кількість успішно та неуспішно виконаних запитів, що дозволяє оцінити швидкодію та надійність тестованої реалізації HTTP-серверу. Для кожної кількості одночасних підключень для кожної тестованої реалізації веб-серверу, здійснювалось три тести швидкодії по

десять тисяч запитів кожен. Середнє арифметичне з трьох виконань використовувалось для порівняння швидкодії веб-серверів.

Минулорічна реалізація веб-серверу продемонструвала непогану швидкодію та значно перевершила позаминулорічну, але не змогла досягнути результатів еталонної:

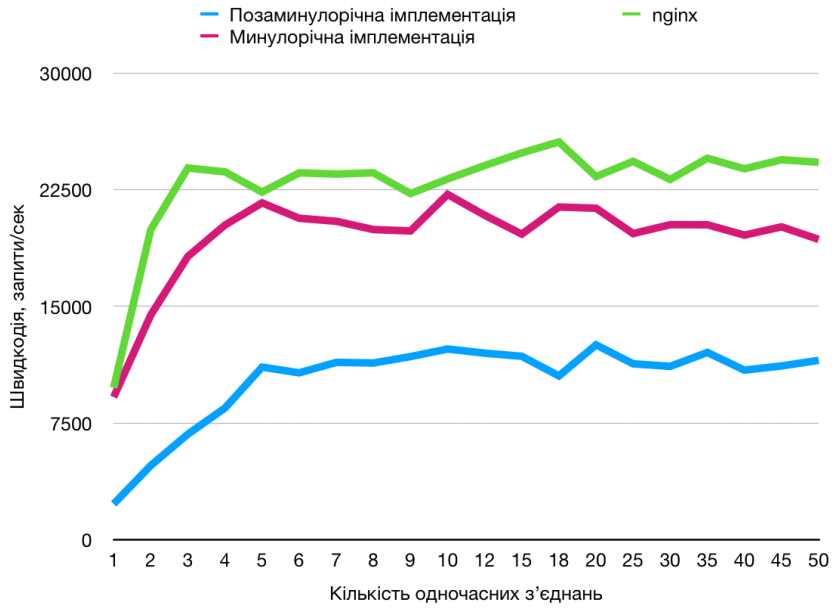

Рисунок 3.2 – Швидкодія минулорічної та позаминулорічної реалізацій веб-серверу на файлах розміру 4 КБ (вимірювання з минулорічної роботи).

Цьогорічна реалізація демонструє ще кращі результати і нарешті перевершує еталонну реалізацію nginx, що гарно видно на великій кількості одночасних підключень (Рисунок 3.3). На графіку видно, що починаючи із восьми одночасно підключених клієнтів, власна реалізація починає стабільно переганяти еталонну у кількості одночасно оброблених запитів, і зі збільшенням кількості клієнтів розрив продовжує збільшуватись.

Та сама поведінка повторюється і на тестах із файлами розміром у 25 (Рисунок 3.4) та 100 (Рисунок 3.5) КБ. Ідентично до минулорічних вимірювань, минулорічна імплементація демонструє покращення своєї швидкодії відносно еталонної зі збільшенням розміру файлів, але не може досягти її. Також, під час цьогорічного тестування, було помічено, що на великих кількостях одночасних підключень (починаючи із тридцяти), через особливості застосування

компонентів фреймворку Qt, минулорічна реалізація не завжди може обробити усі запити, і певний відсоток запитів закінчується помилкою. Цьогорічна реалізація позбавлена цього недоліку і успішно обробила усі надіслані запити протягом виконання тестування.

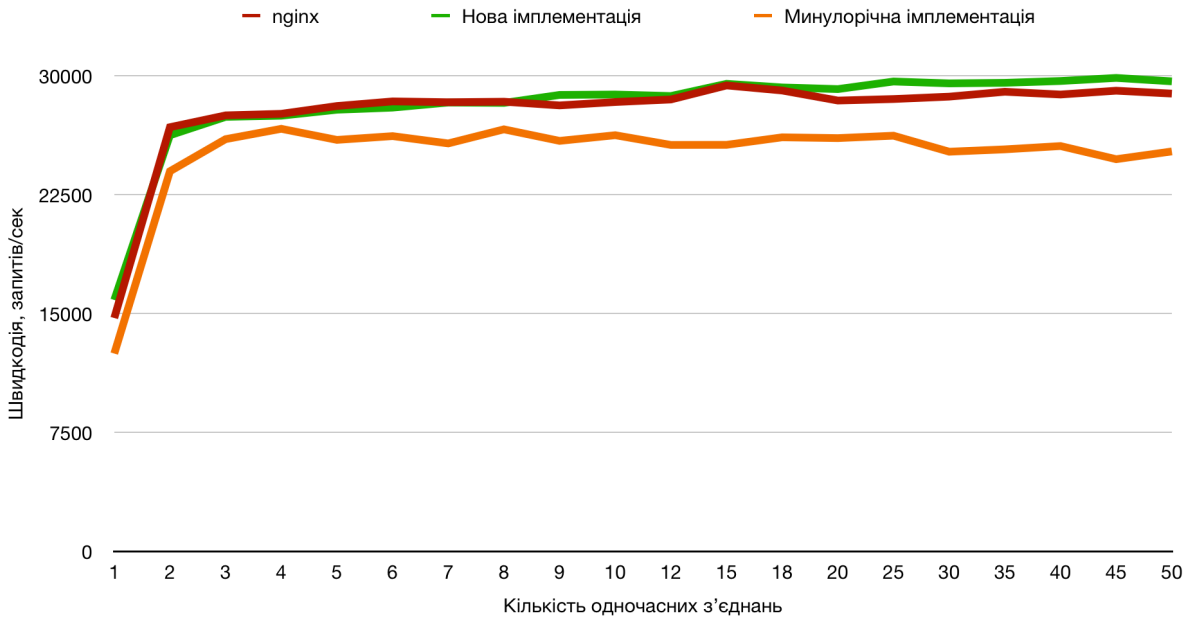

Рисунок 3.3 – Швидкодія протестованих реалізацій веб-серверів на файлах розміром у 4 КБ

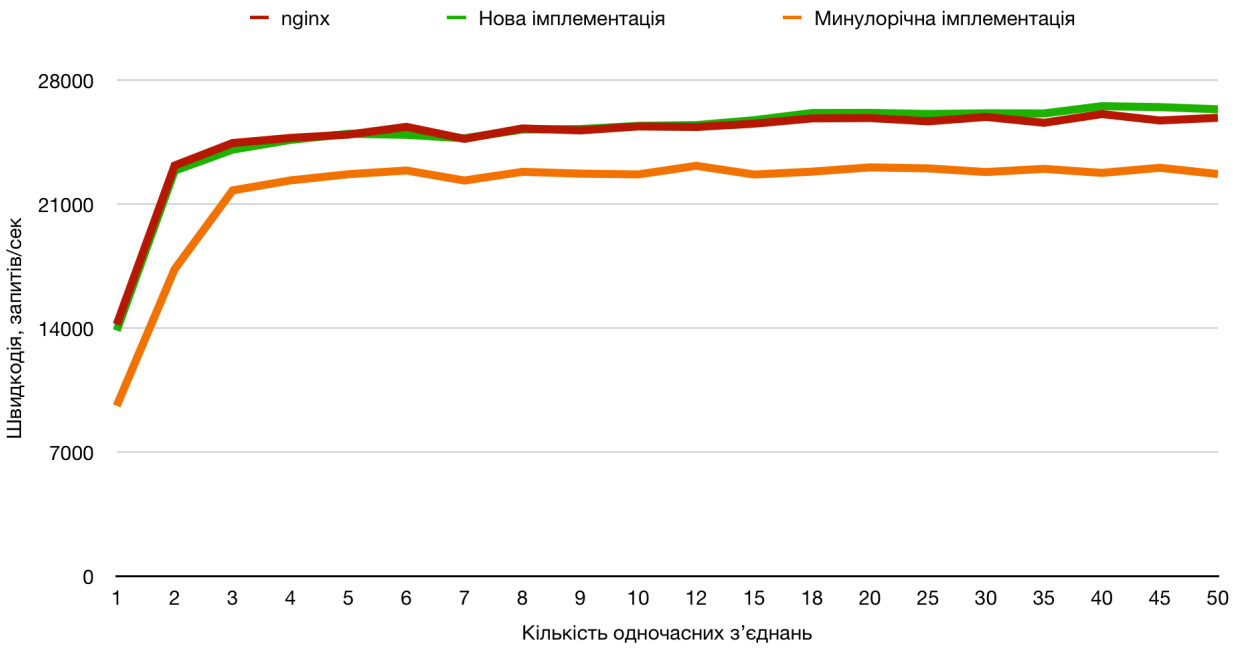

Рисунок 3.4 – Швидкодія веб-серверів на файлах розміром у 25 КБ

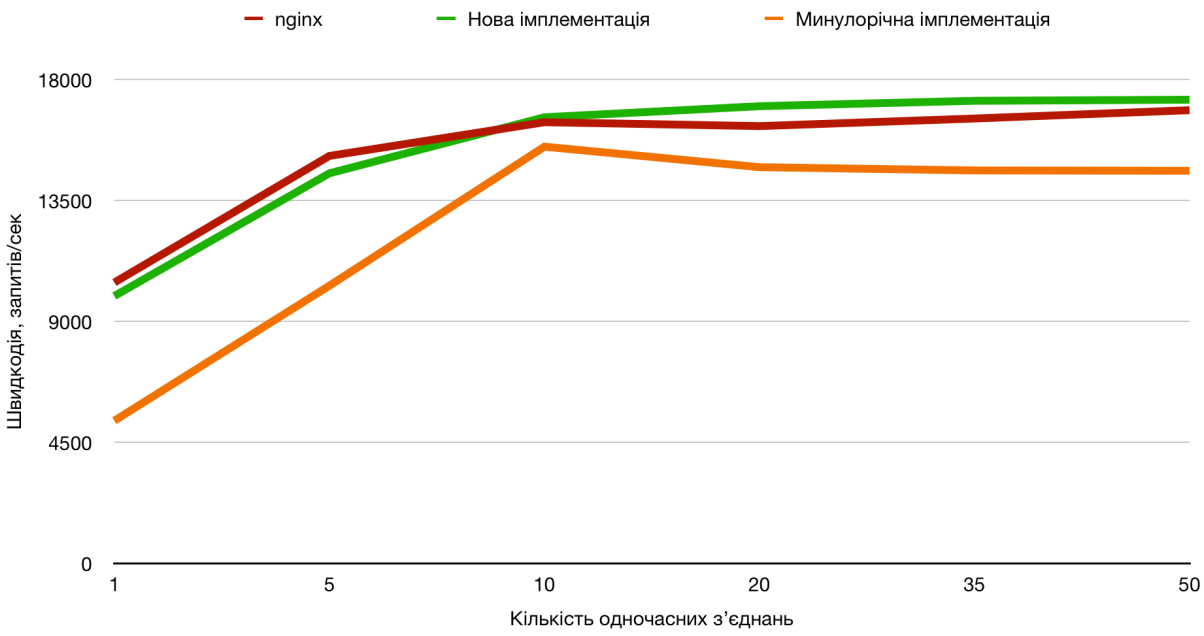

Рисунок 3.5 – Швидкодія веб-серверів на файлах розміром у 100 КБ

Отримані результати тестування демонструють перевагу нового механізму введення-виведення io_uring над класичним libaio та підтверджують доцільність застосування взірця «Проактор» для високонавантаженого та масштабованого програмного забезпечення.

3.1.3. Можливі подальші оптимізації

Враховуючи відносну простоту нової реалізації веб-серверу в порівнянні з nginx, можна стверджувати, що реалізовані в ній базові механізми введення-виведення працюють із хорошим рівнем швидкодії, а отже для покращення результатів цієї реалізації, першочергову оптимізацію краще здійснювати в інших компонентах веб-серверу. Серед потенційних оптимізацій, які теоретично можуть ще покращити швидкодію веб-серверу, можна виділити:

- Кешування – у цілях тестування, кешування серверу nginx було вимкнене. Проте для реального застосування HTTP-серверу кешування часто затребуваних файлів може грати неймовірно важливу роль

- Зміна порядку виконання деяких асинхронних операцій – надсилання запитів на наступні асинхронні операції якомога раніше дозволить пришвидшити їх виконання і мінімізувати час очікування потоку
- Зміна розміру буферу читання в залежності від файлу – оскільки сервер уже знає розмір файлу, який йому необхідно відправити, адаптація виділення пам’яті під цей розмір, дозволить мінімізувати кількість зайвих або неефективних операцій введення-виведення

Варто зазначити, що поточна реалізація HTTP-серверу бракує практично всіх стандартних на сьогоднішній день функцій. Крім ефективного надання файлів, сучасний сервер має підтримувати як мінімум захищені протоколи HTTPS та HTTP/2.0, а також надавати можливість гнучкої конфігурації в залежності від конкретних запитів користувачів. Тому до практичного застосування цього веб-серверу, він потребує значного доопрацювання. В цілому ж, той факт, що реалізація HTTP-серверу, створена в рамках дипломної роботи, може наблизитись та навіть перевершити за швидкодією стандартну для індустрії імплементацію, є чудовим підтвердженням ефективності механізму io_uring та того, що він є гідною заміною libaio.

3.2 Застосування моделі взірця «Проактор» на основі IOUring у масштабованому програмному забезпеченні

Другою задачею, для якої було вирішено застосувати нову модель взірця «Проактор» стала реалізація черги повідомлень для комунікації всередині кластеру. Ефективна робота із мережевим введенням-виведенням може відчутно зменшити затримки комунікації всередині кластеру і покращити загальну швидкодії усієї системи. А запровадження покращень у стандартний взірець черги повідомлень дозволяє за потреби швидко та з мінімальними зусиллями перевести існуючі екземпляри масштабованого програмного забезпечення на більш ефективну реалізацію черги. У цій роботі було розроблено базову

реалізацію брокера повідомлень, який рівномірно розподіляє повідомлення між вузлами-робітника та реалізовує всередині взірець «Request-Reply».

3.2.1. Реалізація черги повідомлень

Для демонстрації роботи масштабованої системи, крім брокеру повідомлень, були розроблені приклади імплементацій вузла-клієнта та вузла-робітника. Для цих цілей були створені класи TcpMessageQueue-BrokerIOUringEventHandler, TcpMessageQueueClientIOUringEventHandler та TcpMessageQueueWorkerIOUringEventHandler, усі з яких реалізують інтерфейс IIOUringEventHandler. Конструктори класів клієнта та робітника приймають об’єкти класів, що реалізують інтерфейси IMessageQueueClient та IMessageQueueWorker, які можуть бути імплементовані користувачем черги повідомлень та містять логіку обробки повідомлень та відповідей на них. А класи-обробники подій IOUring містять низькорівневу логіку передачі повідомлень мережею на основі бібліотеки IOUring.

3.2.1.1 Протокол обміну повідомленнями

Для мінімізації додаткових витрат на комунікацію для власної реалізації черги повідомлень було розроблено простий протокол обміну повідомленнями. У якості транспортного протоколу в межах цієї роботи було вирішено обрати протокол TCP – він дозволяє перекласти проблеми ненадійного мережевого зв’язку, встановлення та підтримки з’єднань на операційну систему. В подальшому, для покращення швидкодії черги повідомлень, доцільним є її переведення на протокол обміну повідомлень на основі протоколу транспортного рівня UDP, який надає значно ширші можливості контролю за передачею даних.

Оскільки протокол TCP оперує потоками даних, а не окремими датаграмами, протокол обміну повідомлень має надавати можливість виокремлювати окремі повідомлення із двійкового потоку даних. Для цього кожне повідомлення містить довжину корисної інформації. Задля зменшення обсягу додаткової інформації, що передається мережею, було розроблено

простий механізм стиснення інформації про розмір повідомлення. Так, якщо повідомлення займає 253 байти або менше, то розмір повідомлення передається одним байтом. Якщо повідомлення займає від 254 до 65536 байт, перший байт розміру встановлюється як 254, а наступні два байти містять довжину повідомлення. Якщо повідомлення більше за 65535 байт, перший байт встановлюється як 255, і наступні вісім байтів містять розмір повідомлення.

Також, крім розміру повідомлення, до кожного повідомлення додається один сервісний байт. Він може містити інформацію про тип повідомлення, або помилкову ситуацію, що сталася на стороні вузла-клієнта або робітника.

Для перевикористання реалізацій цього протоколу комунікації було розроблено класи TcpMessageStreamFrameFormatter та TcpMessage-StreamFrameParser, які містять усю логіку обгортки двійкових повідомлень додатковою інформацією та перетворення двійкового потоку у набір окремих повідомлень. Ці класи реалізують інтерфейси IFrameFormatter та IFrameParser відповідно, що дозволяє в майбутньому замінити протокол комунікації на інший без необхідності зміни решти програми. Клас IFrameParser за аналогією із класом HeadersBuilder, який використовувався для отримання заголовків HTTP-запиту, може отримувати двійкові буфери довільного розміру, зміст яких буде перетворений на окремі повідомлення.

```
class IFrameParser
{
public:
  virtual ~IFrameParser();

  void pushFrame(const std::vector<uint8_t> frame);
  std::pair<uint8_t, std::vector<uint8_t>> popMessage();
  bool hasMessage();
};
```

Приклад 3.1 – Публічна частина інтерфейсу IFrameParser

3.2.1.2 Механізм комунікації між вузлами

Використання такої черги повідомлень передбачає поділ вузлів кластерів на 3 типи: клієнти, робітники та брокер. Клієнти надсилають на виконання запити, які розподіляються брокером між вузлами-робітниками. Вузли-робітники, в свою чергу, отримують запит, оброблюють його, та відправляють відповідь назад брокеру, який надсилає її відповідному клієнту (Рисунок 3.6).

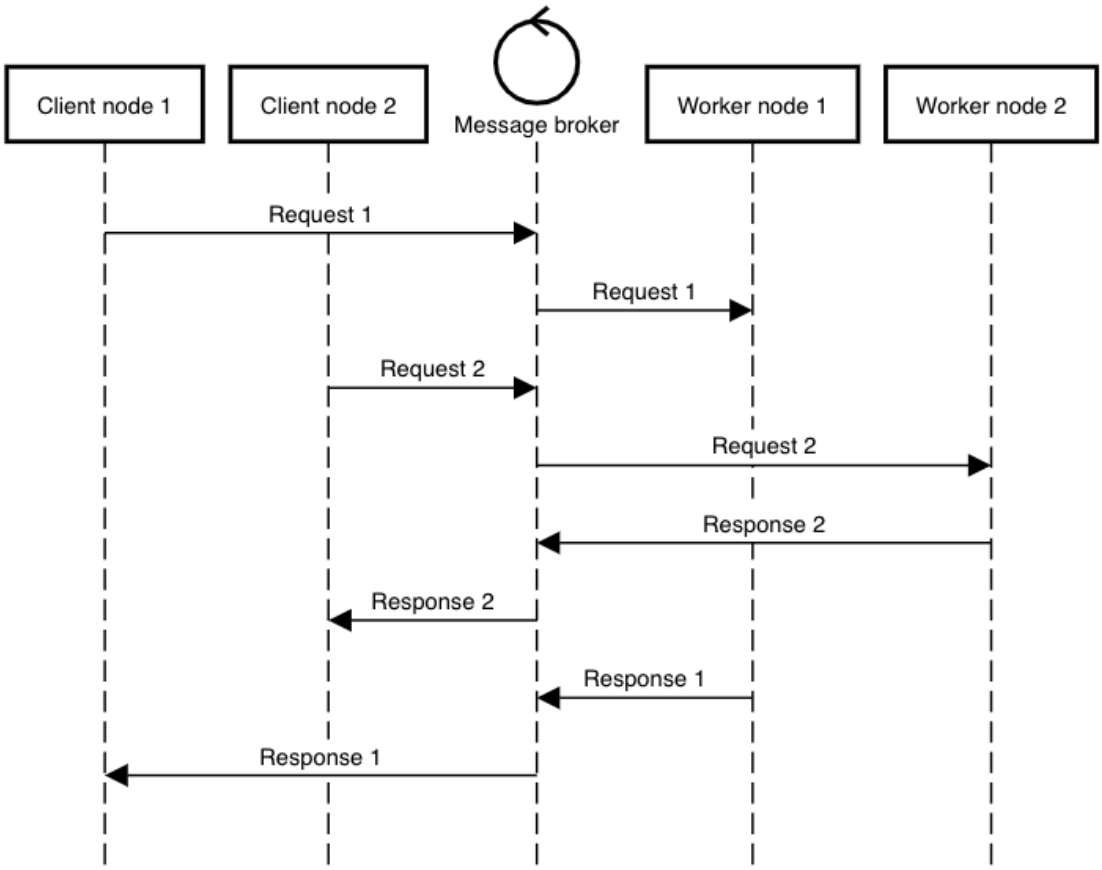

Рисунок 3.6 – Діаграма виконання двох запитів через чергу повідомлень

Кожен вузол взаємодіє із власним екзкемпляром класу IOUring, через який за допомогою класів-обробників подій відбувається мережева комунікація із брокером. А за допомогою класу TagSplitterProxyIOUringEventHandler, якщо якийсь із вузлів потребує додаткової роботи із файловою системою чи мережею, він може підключити додаткові обробники подій до вже існуючого екземпляру IOUring для спрощення керування операціями асинхронного введення-виведення.

Для початку роботи із брокером, і вузол-клієнт, і вузол-робітник мають спершу надіслати брокеру службове повідомлення. Для клієнта це порожнє повідомлення із сервісним байтом зі значенням 2, для робітника – зі значенням 4. Після відправки службового повідомлення, клієнт може надсилати брокеру звичайні повідомлення, які будуть розподілені між робітниками. Якщо до брокеру під'єднаний вільний робітник, який не виконує ніякий запит, то повідомлення одразу перенаправляється йому. Якщо ж вільних робітників немає, то повідомлення додається у внутрішню чергу, і буде передане, як тільки один із робітників звільниться, або приєднається новий робітник.

Така схема комунікації є дуже поширеною у масштабованому програмному забезпеченні, адже дозволяє в залежності від навантаження системи змінювати кількість вузлів-робітників або клієнтів і адаптуватись до зовнішніх умов. А той факт, що вся комунікація відбувається централізовано, означає що окремі вузли не потребують знань про зміну структури кластеру для їх ефективної та коректної роботи. Для визначення, чи може застосування покращених моделей взірця «Проактор» теоретично покращити швидкодію комунікації у розподілених системах, наступним кроком стало вимірювання швидкодії отриманої реалізації у порівнянні зі стандартними рішеннями.

3.2.2. Вимірювання швидкодії черги повідомлень

Для вимірювання швидкодії власної реалізації черги повідомлень, необхідно було розробити просту штучну задачу, яка б цілковито залежала від швидкодії комунікації між вузлами. Це дозволить уникнути впливу інших аспектів виконання програми та найкраще продемонструвати ступінь впливу швидкодії комунікації на систему.

В якості такої задачі було вибрано програму «Echo». Клієнт може надіслати на обробку брокеру будь-яке повідомлення, а робітник після його отримання просто повертає його у якості відповіді. Така задача, хоча і є достатньо примітивною, дозволяє явно виміряти швидкість передачі повідомлень мережею,

та дозволяє продемонструвати роботу системи у різних сценаріях (наприклад, із повідомленнями різної довжини).

Для вимірювання швидкодії системи, клієнт відправляє робітнику через брокера десять тисяч повідомлень фіксованої довжини. Потім клієнт заміряє час отримання відповіді на кожне з них та виводить отримані результати по завершенню передачі даних. Середнє значення часу обробки одного повідомлення від надсилання до отримання відповіді використовується у якості метрики швидкодії. Для перевірки швидкодії систем у різних сценаріях використання, була виміряна швидкодія передачі повідомлень розміром від 4 байт до одного мегабайту. Більші повідомлення, хоча і можуть бути передані цими системами, дещо виходять за стандартні сценарії використання черг повідомлень. Наприклад, черга повідомлень Amazon SQS, достатньо популярна серед розробників, навіть не дозволяє надсилати повідомлення розміром більші зв 256 кілобайт. В якості готових реалізацій черги повідомлень, із якими буде здійснюватись порівняння, було вибрано RabbitMQ та ZeroMQ.

3.2.2.1 Брокер повідомлень RabbitMQ

RabbitMQ – популярний брокер повідомлень із відкритим вихідним кодом. За допомогою системи плагінів він імплементує декілька відкритих протоколів обміну повідомленнями, що може полегшити його впровадження у існуючі розподілені системи. А основним протоколом, із яким працює RabbitMQ, є AMQP (англ. Advanced Message Queuing Protocol, досконалий протокол чергування повідомлень). Це відкритий протокол обміну повідомленнями, що був розроблений у 2003 році JPMorgan Chase та робочою групою OASIS. В якості протоколу транспортного рівня AMQP використовує протокол TCP.

RabbitMQ є масштабованим брокером повідомлень. та може працювати одночасно на декількох вузлах задля підвищення надійності та швидкодії його роботи. А можливості протоколу AMQP та самого брокеру дозволяють гнучко налаштовувати правила та механізми доставки повідомлень. Для роботи із

RabbitMQ існують бібліотеки для всіх поширених мов програмування, а сам брокер може працювати на вузлах під керуванням Linux, Windows та macOS.

3.2.2.2 Бібліотека обміну повідомленнями ZeroMQ

ZeroMQ – популярна бібліотека для комунікації між вузлами та між процесами всередині одного вузла. Протокол передачі повідомлень ZeroMQ підтримує використання різних протоколів транспортного рівня, таких як TCP, UDP та доменні сокети UNIX. На відміну від RabbitMQ, ZeroMQ не надає готову реалізацію брокера, але надає механізми передачі та трансляції повідомлень між вузлами кластеру. Це дозволяє як реалізувати власний брокер повідомлень, так і комунікувати між вузлами напряму. Розробники стверджують, що протокол передачі повідомлень ZeroMQ оптимізований для передачі малих повідомлень, але може застосовуватись і для повідомлень великого розміру.

Для порівнянні швидкодії ZeroMQ, у якості брокера повідомлень було використано один із офіційних прикладів застосування бібліотеки, наданих розробників. Механізм роботи цього брокеру є достатньо близьким до механізму роботи власної імплементації, адаптований до специфіки бібліотеки ZeroMQ. У якості протоколу транспортного рівня, для створення рівних умов із іншими системами, було обрано протокол TCP.

3.2.2.3 Результати виконання

За результатами виконання тестування (Рисунок 3.7), власна реалізація черги повідомлень на основі IOUring демонструє впевнену перевагу над іншими реалізаціями черг для повідомлень розміром до 16 КБ. У проміжку 32 КБ - 128 КБ переважає ZeroMQ, а для більших повідомлень переважати починає RabbitMQ. В цілому, власна реалізація демонструє значну перевагу над іншими для малих повідомлень (30 мікросекунд проти 99 для ZeroMQ), показуючи доцільність застосування механізму io_uring та моделі взірця «Проактор» на його основі для комунікації у масштабованому програмному забезпеченні.

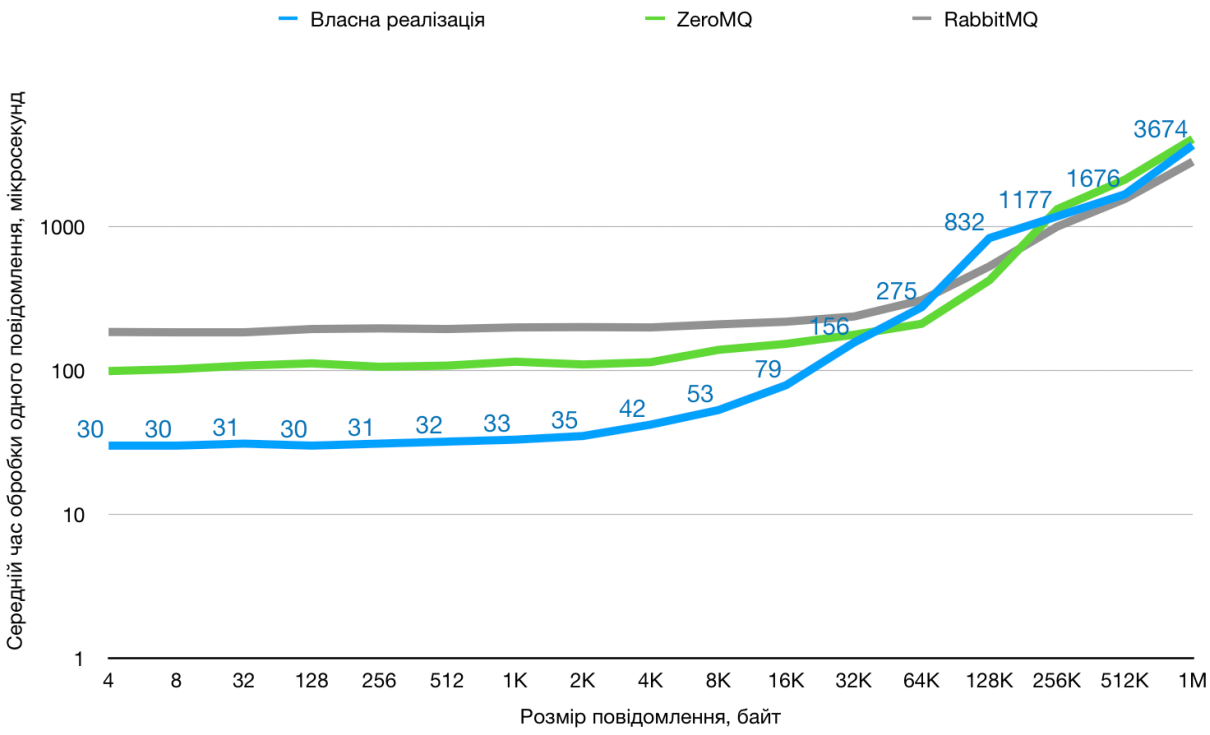

Рисунок 3.7 – Графік часу обробки повідомлень у залежності від їхнього розміру, логарифмічна шкала. Синіми цифрами на графіку позначені результати власної реалізації

3.2.3. Можливі подальші оптимізації

Хоча отримані результати є дуже багатообіцяючими, власна реалізація черги повідомлень є ще далекою від її повноцінного застосування у реальних високонавантажених системах. Цій реалізації бракує механізмів безпеки, таких як шифрування та автентифікація, та повноцінної обробки помилкових ситуацій та механізмів відновлення після відмов.

З точки зору швидкодії, можна помітити, що починаючи із 16 КБ, власна реалізація сильно втрачає у швидкодії, і ця втрата є більш вираженою, ніж в інших реалізацій. Це може бути пов'язано із особливостями виділення пам’яті для повідомлень у власній реалізації та потребувати оптимізації роботи із пам’яттю задля досягнення кращої та більш передбачуваної швидкодії.

ВИСНОВКИ

У роботі було розглянуто механізми комунікації у масштабованому програмному забезпеченні та пов’язані із ними взірці проектування масштабованого програмного забезпечення. Потім було розглянуто сучасний механізм введення-виведення – io_uring та багатопотокову модель взірця «Проактор» на його основі. Для реалізації цієї моделі було розроблено об’єктно-орієнтовану бібліотеку-обгортку над цим механізмом введення-виведення – IOUring. На основі цієї бібліотеки було реалізовано модельну задачу минулого року – HTTP-сервер та механізм для комунікації між компонентами масштабованого програмного забезпечення – чергу повідомлень. Було досліджено та виміряно вплив нової моделі взірця «Проактор» на швидкодію цих імплементацій та було здійснене порівняння їх швидкодії із еталонними.

Для HTTP-серверу, нова реалізація на основі IOUring продемонструвала результати, що суттєво переважають минулорічну реалізацію, та нарешті змогла наздогнати та обійти еталонну реалізацію – nginx за швидкістю обробки запитів користувачів. Тестування передбачало вимірювання швидкодії веб-серверу із різними розмірами файлів та різною кількістю одночасно під’єднаних клієнтів, що дозволяє оцінити роботу протестованих реалізацій у різних сценаріях роботи. Враховуючи мінімальний обсяг оптимізацій, включених у власну реалізацію, цей результат може бути покращений ще далі.

Реалізація черги повідомлень теж продемонструвала гарні результати – для повідомлень розміром до 16 КБ, власна реалізація черги впевнено переважає над двома еталонними реалізаціями, із якими проводилось порівняння – ZeroMQ та RabbitMQ. Починаючи із 32 КБ, розрив між імплементаціями зменшується, але власна реалізація черги все ще демонструє гідний результат, поступаючись RabbitMQ, але переганяючи ZeroMQ. Тестування проводилось із одним під’єднаним клієнтом на великій кількості повідомлень та не є вичерпним. Для отримання повної картини поведінки власної реалізації черги у порівнянні з еталонними, доцільним є тестування усіх реалізацій на повноцінному кластері з

декількох машин, адже тестування на одній машині із декількома клієнтами може бути нерепрезентативним через те, що той самий комп’ютер виконує функції як клієнта, так і обробника та брокера повідомлень, що може суттєво вплинути на ефективність виконання цих функцій та зробити результати цього тестування не відповідаючими дійсності.

Результати цієї роботи можуть бути використані для створення повноцінних реалізацій HTTP-серверу та черги повідомлень, які можуть бути застосовані у реальних високонавантажених системах. Розроблена бібліотека IOUring надає просту та швидку основу для розробки програм, що інтенсивно взаємодіють із підсистемою введення-виведення, а моделі взірця «Проактор» на її основі працюють краще, аніж на основі інших доступних механізмів асинхронного введення-виведення.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Proactor – An Object Behavioral Pattern for Demultiplexing and Dispatching Handlers for Asynchronous Events – Irfan Pyarali, Tim Harrison, Douglas C. Schmidt – St. Louis, MO: Washington University, Department of Computer Science, 1997
2. Moore's law at 50: Are we planning for retirement? / Greg Yeric – Austin, TX: ARM Research, 2015
3. AMD EPYC 9004 Series Server Processors / Santa Clara, CA: AMD, 2022
4. Designing Data-Intensive Applications / Martin Kleppmann – Sebastopol, CA: O'Reilly Media, 2017
5. Locality Awareness in a Peer-to-Peer Publish/Subscribe Network / Fatemeh Rahimian, Thinh Le Nguyen Huu, Sarunas Girdzijauskas – Kista, Sweden: Swedish Institute of Computer Science, 2012
6. Design Patterns: Elements of Reusable Object-Oriented Software / Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides – Boston, Addison-Wesley: 1994
7. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions / Gregor Hohpe, Bobby Woolf – Boston, Addison-Wesley: 2003
8. KPTI: a Mitigation Method against Meltdown / Lars Müller – Wiesbaden, Germany, WAMOS'18: 2018
9. Gcc and kernel stability / Linus Torvalds – fa.linux.kernel newsgroup, 2005
10. Std::execution | P2300R6 / Michał Dominiak та інші: SG1 – Concurrency and Parallelism, 2023
11. April 2023 Web Server Survey / Netcraft, 2023: https://news.netcraft.com/archives/category/web-server-survey/

ДОДАТОК А

(довідниковий)

UML-діаграма класів практичної частини кваліфікаційної роботи