Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук
(повна назва)

Кафедра Системотехніки
(повна назва)

# АТЕСТАЦІЙНА РОБОТА
# Пояснювальна записка

рівень вищої освіти другий (магістерський)

Дослідження методів обробки даних у високонавантажених системах
(тема)

Виконав: студент 2 курсу, групи СПРм-18-1
Ілюнін А.В.
(прізвище, ініціали)

Спеціальність 122 Комп'ютерні науки
(код і повна назва спеціальності)

Тип програми освітньо-професійна

Освітня програма Системне проектування
(повна назва освітньої програми)

Керівник доц. Хряпкін О.В.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри СТ _____ проф. Гребеннік І.В.
(підпис) (прізвище, ініціали)

2019 р.

*Атестаційна робота оформлена у відповідності до вимог діючих стандартів та методичних вказівок.*

*Матеріали атестаційної роботи не містять відомостей, що заборонені для опублікування у відкритих виданнях.*

*Попередній захист проведено.*

*Керівник атестаційної роботи* Хряпкін О.В.

Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
Кафедра системотехніки
Освітньо-кваліфікаційний рівень другий (магістерський)
Спеціальність 122 Комп'ютерні науки

Тип програми освітньо-професійна

Освітня програма Системне проектування

ЗАТВЕРДЖУЮ:
Зав. кафедри СТ _____
(підпис)
"____" _____ 2019 р.

# ЗАВДАННЯ
НА АТЕСТАЦІЙНУ РОБОТУ

студентові Ілюніну Андрію В'ячеславовичу
(прізвище, ім'я, по батькові)

1.Тема роботи (проекту) Дослідження методів обробки даних у високонавантажених системах
затверджена наказом по університету від "04" листопада 2019 р. № 1627 Ст
2. Термін здачі студентом закінченого роботи 12.12.2019
3. Вихідні дані до роботи *Перелік використовуваних програмних засобів: ОС Ubuntu 16.04, середовище розробки Intellij IDEA, локальний сервер, мобільний додаток під керуванням платформи Flutter*
4. Перелік питань, що потрібно опрацювати в роботі *4.1 Вступ. 4.2 Аналіз сучасного стану проблеми. 4.2.1 Аналіз та обґрунтування тенденцій інвестування в нові технології. 4.2.2 Визначення підходів та методів реалізації щодо реалізації високонавантаженої системи пошуку спеціалізованої інформації. 4.2.3 Постановка задачі. 4.3 Аналіз BigData та застосування суміжних інструментів. 4.3.1 Визначення BigData. 4.3.2 Генезіс BigData. 4.3.3 Характеристики BigData. 4.3.4 Інструменти та методи галузі BigData. 4.4 Аналіз методів паралельного програмування. 4.4.1 Визначення, роль та проблеми паралельних обчислень. 4.4.2 Аналіз моделі «1 потік – 1 запит». 4.4.3 Аналіз моделі пула потоків для обробки запитів. 4.4.4 Аналіз асинхронної моделі виконання. 4.4.5 Аналіз методу реактивного програмування. 4.5 Аналіз та вибір програмного середовища виконання. 4.5.1 Огляд та аналіз CLR. 4.5.2 Огляд та аналіз JVM. 4.5.3 Порівняння CLR та JVM та обґрунтування подальшого вибору. 4.5.4 Детальний огляд JVM та її компонентів з подальшим виявленням шляхів контекстної оптимізації. 4.6 Практичне порівняння методів та конфігурацій у контексті високонавантаженої системи для пошуку інформації. 4.6.1 Опис оточуючого середовища системи. 4.6.2 Підготовка, проведення та аналіз результатів роботи стандартної моделі прийому та обробки запитів. 4.6.3 Підготовка, проведення та аналіз результатів роботи реактивної моделі прийому на обробки запитів. 4.6.4 Порівняння отриманих результатів. 4.7 Висновки. 4.8 Перелік джерел посилань*

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, схем, плакатів) *5.1 Типова архітектура веб-сервісів (2 аркуші А4). 5.2 Асинхронна модель однопоточного прийому та делегації обробки запитів (1 аркуш А4). 5.3 Характеристики Big Data (3 аркуші А4). 5.4 Принцип роботи MapReduce (1 аркуш А4). 5.5 Архітектура Apache Hadoop (2 аркуші А4). 5.6 Огляд моделі «потік на запит» (4 аркуші А4) 5.7 Кореляція між кількістю потоків, часом на зміну активного потоку та даними, що зберігає кожен потік (1 аркуш А4). 5.8 Графічне відображення асинхронної моделі виконання (3 аркуші А4). 5.9 Порівняння Tomcat та Netty (3 аркуші А4). 5.10 Обґрунтування рентабельності реактивної моделі (3 аркуші А4). 5.11 Порівняння високорівневих архітектур CLR та JVM (1 аркуш А4). 5.12 Огляд JVM (2 аркуші А4). 5.13 Проведення експериментів зі синхронною моделлю (6 аркушів А4). 5.14 Проведення експериментів з реактивною моделлю (6 аркушів А4). 5.15 Порівняння результатів експериментів з реактивною та синхронною моделями (5 аркушів А4)*

6. Консультанти розділів роботи

| Найменування Розділу | Консультант (посада, прізвище, ім'я, по батькові) | Позначка консультанта про виконання розділу | |
|---|---|---|---|
| | | підпис | дата |
| Розділи спеціальної частини | доц. Хряпкін О.В. | | |
| | | | |

**КАЛЕНДАРНИЙ ПЛАН**

| Номер | Назва етапів роботи (проекту) | Термін виконання етапів роботи (проекту) | Примітка |
|---|---|---|---|
| 1 | *Отримання завдання атестаційної роботи* | 01.09.2019 | виконано |
| 2 | *Аналіз завдання та літератури з теми атестаційної роботи* | 03.09.2019 – 25.09.2019 | виконано |
| 3 | *Вибір засобів для розробки, технічних вимог до програми* | 25.09.2019 – 30.09.2019 | виконано |
| 4 | *Структурне проектування* | 30.09.2019 – 20.10.2019 | виконано |
| 5 | *Вибір середовища розробки програми* | *20.10.2019 – 25.10.2019* | виконано |
| 6 | *Розробка програми* | *25.10.2019 – 15.11.2019* | виконано |
| 7 | *Тестування програми* | *15.11.2018 – 17.11.2019* | виконано |
| 8 | *Оформлення пояснювальної записки* | *17.11.2019 – 28.11.2019* | виконано |
| 9 | *Оформлення графічної частини та презентаційних матеріалів* | *30.11.2019* | виконано |
| 10 | *Представлення атестаційної роботи ДЕК* | *16.12.2019* | виконано |

Дата видачі завдання _____

Студент _____
(підпис)

Керівник роботи _____ доц. Хряпкін О.В.
(підпис) (посада, прізвище, ім'я, по батькові)

# РЕФЕРАТ

Атестаційна робота містить: 151 сторінок, 39 рисунків, 15 джерел, 3 додатки.

Предмет дослідження – високонавантажені системи пошуку інформації.

Мета роботи – визначення найбільш ефективного методу обробки інформації у контексті високонавантаженої пошукової системи.

Методи дослідження – синхронний метод обробки запитів побудований на моделі “один потік – один запит”, що обмежений пулом потоків деякого фіксованого обсягу, а також метод реактивного програмування, що є також є обмеженим пулом потоків деякого значно меншого обсягу. Дослідження проводилося на IBM-сумісному ноутбуці (тактова частота процесору 2.5 ГЦ, обсяг оперативної пам’яті – 8 ГБ) за допомогою IDE Intellij IDEA та Spring Boot.

Результати дипломної роботи – дві реалізації веб-сервісу для пошуку інформації щодо авіарейсів: синхронна та реактивна, що мають єдиний інтерфейс.

Область застосування – реалізації веб-сервісів можуть бути застосовані для порівняння їхньої ефективності у різних середовищах виконання.

ВИСОКОНАВАНТАЖЕНІ СИСТЕМИ, РЕАКТИВНЕ ПРОГРАМУВАННЯ, АСИНХРОННА МОДЕЛЬ, JVM, GC, SPRING REACTOR, NETTY, TOMCAT

# ABSTRACT

Explanatory note: 151 p., 39 fig., 15 sources, 3 applications.

Object of research – search services under high load.

The aim of the thesis is to define the most effective data processing method in the context of search system under high load.

Research method – synchronous data processing model based on “thread per request” model which is being limited by a thread pool of fixed maximal size and method of reactive programming which is limited by a thread pool of a much smaller size. The study was conducted on an IBM-compatible laptop (CPU clock speed 2.5 HZ, RAM – 4 ГБ) using Intellij IDEA IDE and Spring Boot framework.

The results of the thesis – two implementations of a flights data search web-service – synchronous and reactive one, which have identical network interface.

Scope – web-service implementations can be used to compare their effectiveness in the context of special runtime environments.

HIGH LOAD, REACTIVE PROGRAMMING, ASYNCHRONOUS MODEL, JVM, GC, SPRING REACTOR, NETTY, TOMCAT

ЗМІСТ

ВСТУП……..…. 7

1 АНАЛІЗ СУЧАСНОГО СТАНУ ПРОБЛЕМИ ……. 9

1.1 Аналіз та обґрунтування тенденцій інвестування в нові технології... 9

1.2 Визначення підходів та методів щодо реалізації високонавантаженої системі пошуку спеціалізованої інформації…….. 13

1.3 Постановка задачі……. 16

2 АНАЛІЗ BIGDATA ТА ЗАСТОСУВАННЯ СУМІЖНИХ ІНСТРУМЕНТІВ 18

2.1 Визначення BigData…….……..….. 18

2.2 Генезіс BigData…….... …… 19

2.3 Характеристики BigData ……. 20

2.4 Інструменти та методи галузі BigData…… 28

3 АНАЛІЗ МЕТОДІВ ПАРАЛЕЛЬНОГО ПРОГРАМУВАННЯ……. 36

3.1 Визначення, роль та проблеми паралельних обчислень……... 36

3.2 Аналіз моделі «1 потік – 1 запит» …….. 38

3.3 Аналіз моделі пула потоків для обробки запитів…….. 41

3.4 Аналіз асинхронної моделі виконання …….. 45

3.5 Аналіз методу реактивного програмування…….. 48

4 АНАЛІЗ ТА ВИБІР ПРОГРАМНОГО СЕРЕДОВИЩА ВИКОНАННЯ…… 56

4.1 Огляд та аналіз CLR.....……. 56

4.2 Огляд та аналіз JVM……. 58

4.3 Порівняння CLR та JVM та обґрунтування подальшого вибору……. 59

4.4 Детальний огляд JVM та її компонентів з подальшим виявленням шляхів контекстної оптимізації…….. 62

5 ПРАКТИЧНЕ ПОРІВНЯННЯ МЕТОДІВ ТА КОНФІГУРАЦІЙ У КОНТЕКСТІ ВИСОКОНАВАНТАЖЕНОЇ СИСТЕМИ ДЛЯ ПОШУКУ ІНФОРМАЦІЇ…… 70

5.1 Опис оточуючого середовища системи…… 70
5.2 Підготовка, проведення та аналіз результатів роботи стандартної моделі прийому та обробки запитів…… 71
5.3 Підготовка, проведення та аналіз результатів роботи реактивної моделі прийому та обробки запитів…… 76
5.4 Порівняння отриманих результатів ……... 82
ВИСНОВКИ……. 86
ПЕРЕЛІК ПОСИЛАНЬ…… 88
ДОДАТОК А «ГРАФІЧНІ МАТЕРІАЛИ ДИПЛОМНОЇ РОБОТИ» ……. 90
ДОДАТОК Б «ТЕКСТ ПРОГРАМИ» …… 137
ДОДАТОК В «ВІДОМІСТЬ АТЕСТАЦІЙНОЇ РОБОТИ» …….. 148

# ВСТУП

З кожним роком обсяг інформації, що генерується, та веб-сервісів за допомогою яких вирішуються повсякденні задачі збільшуються. Разом з тим зростає й кількість веб-клієнтів, які є користувачами таких сервісів. Підтримка великої кількості клієнтських запитів, що опрацьовуються одночасно, становиться дедалі більш актуальною проблемою. Щодо об'ємів даних – до 2020 року обсяг даних, що зберігаються, збільшиться до 40-44 зеттабайт (1 ЗБ ~ 1 мільярд ГБ). До 2025 року – приблизно до 400 зеттабайт. Величезна частка цих даних необхідна для функціонування різних сфер людської діяльності і повинна бути доступною в глобальній мережі (через спеціальні механізми авторизації чи без них). Найбільш популярним прикладом цього є пошукові системи, що дозволяють здійснювати пошук релевантних веб-сторінок, документів і іншого роду ключових файлів по ряду ключових лексем: ця система є одним з найбільш явних прикладів високонавантажених систем з великим акцентом на опрацювання великої кількості даних. Найбільш популярні пошукові системи, такі як Google чи Yahoo!, постійно обробляють немислимі об'єми даних, при цьому час відгуку на запити в таких системах є критично малим.

Маючи це на увазі, важливо підкреслити що прогрес в розробці апаратного забезпечення в останні роки вже не підкоряється закону Мура – ми наблизилися до фізичної межі, де швидкість обчислень обмежена розміром атома і швидкістю світла. Традиційні підходи, конфігурації і інструменти втрачають рентабельність зі збільшенням обсягів оброблюваних даних а також зі збільшенням навантаження на саму систему, а після певного порогу вони стають зовсім невідповідними для вирішення поставлених завдань через накладених на них обмежень. Відповідно, ефективне отримання та обробка, а також управління структурованими і неструктурованими даними за допомогою сучасних підходів, технологій, інструментів, а також програмних компонентів – сфера, яка стає все більш важливою.

Метою даної роботи є дослідження методів, інструментів, технологій та програмних компонентів, що використовуються, при отриманні та обробці великих обсягів даних у контексті великої на сервер на прикладі пошукової системи, де головним фактором оптимізації є ліміт навантаження та час відгуку серверу.

У ході роботи були проаналізовані ключові для ефективності пошукової системи характеристики і вибрані, досліджені та проаналізовані методи та інструменти, що націлені на вирішення таких проблем з послідуючим аналізом їх застосування при реалізації високонавантаженої пошукової системи: ряд методів та інструментів категорії BigData, а також методи ефективної паралельного прийому та обробки запитів, а саме – реактивне програмування, що має своєю ціллю ефективну з точки зору паралельного довкілля обробку подій, що надходять, у вигляді потоку, коли під подіями можуть матися на увазі як клієнтські запити, так і виконання та паралелізм внутрішніх операцій в системи.

# 1 **АНАЛІЗ СУЧАСНОГО СТАНУ ПРОБЛЕМИ**

## 1.1 Аналіз та обґрунтування тенденцій інвестування в нові технології

Типовий веб-проект складається з трьох основних компонентів: клієнтського додатку (або просто “клієнту”), який зазвичай представлений веб-додатком для браузеру або додатком для смартфону, веб-серверу (одним або декількома з їх подальшою реплікацією), представленим центральною реалізацією бізнес-логіки системи за допомогою якогось фреймворку загального призначення (або специфічного призначення, наприклад e-commerce фреймворки та платформи, такі як SAP Hybris, Magento і т.п.) та сервером даних, зазвичай представленим за потребою реплікованим SQL СУБД сервером. Це є прикладом типової трирівневої архітектури програмного забезпечення (рис. 1.1).

Проекти, створені за такою архітектурою, представляють приблизно 92% відсотки від кількості усіх веб-проектів, тому що це є рентабельним рішенням в силу свого розвитку, підтримки зі сторони веб-співтовариства та open-source проектів, та популярності, де популярність є одним з ключових факторів завдяки тому, що полегшується задача пошуку необхідних для реалізації або підтримки проекту спеціалістів, а також навчання доступних в компанії інженерів з іншим профілем підготовки або спеціалізацією.

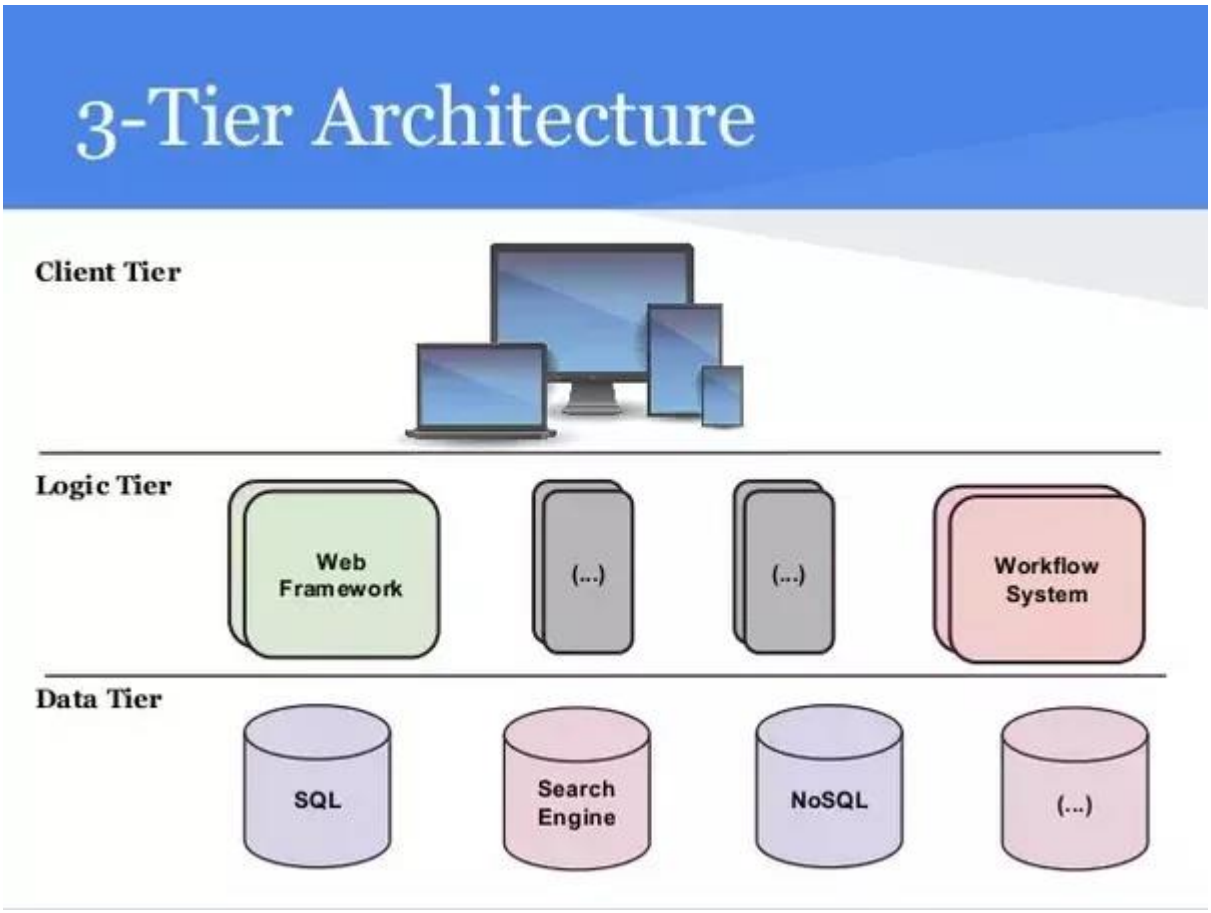


Рисунок 1.1 – Трирівнева архітектура ПО

При цьому системи дотримуються рішення, коли на одне з’єднання з сервером створюється один (або більше, залежить від певної задачі) потік виконання. Це значить, що при кількості паралельних з’єднань з сервером N, обсяг створених у системі потоків дорівнюватиме M, де M більше або рівне N. Графічна репрезентація цієї моделі наведена на рисунку 1.2.

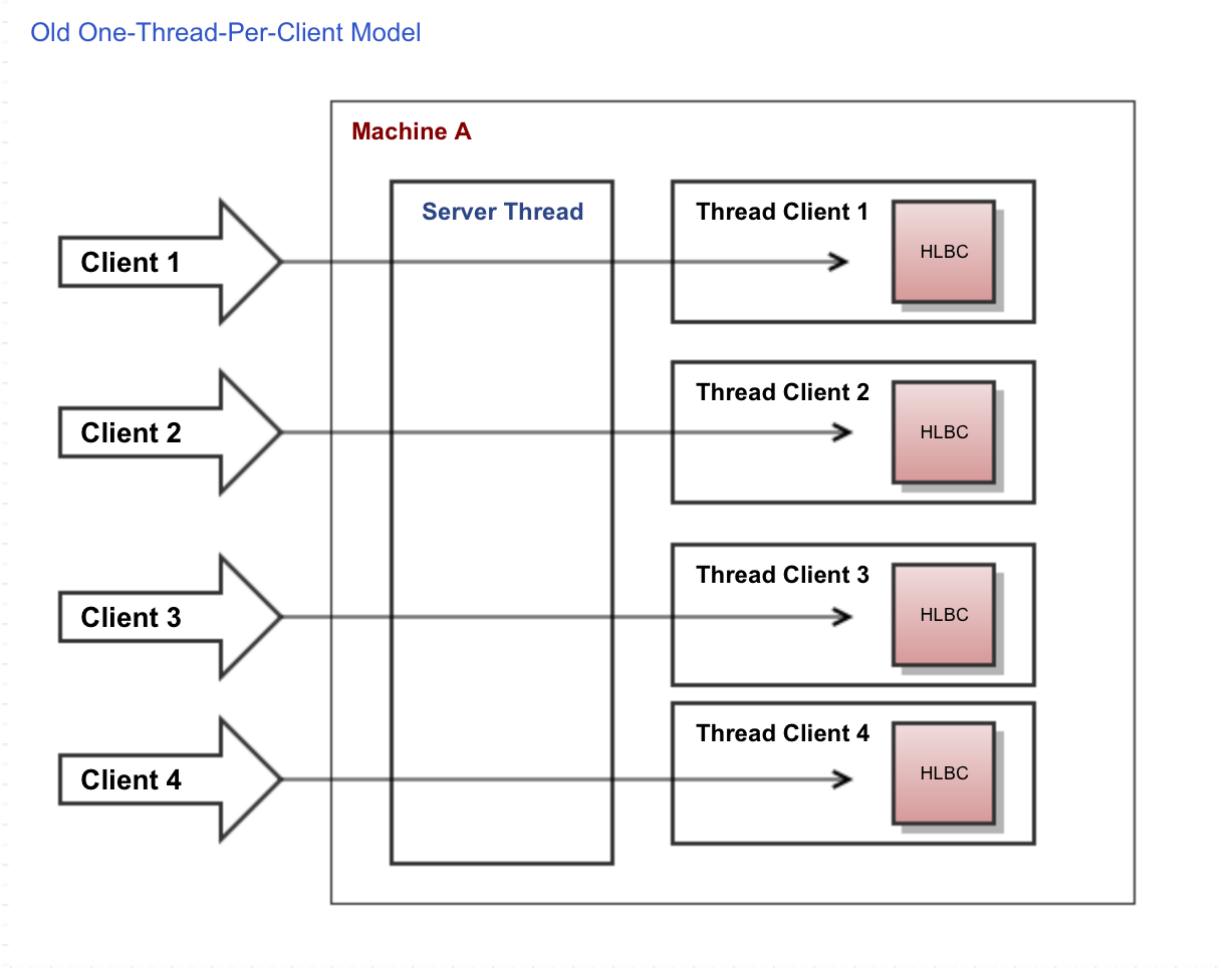


Рисунок 1.2 – Традиційна блокуюча модель “1 з’єднання – 1 потік”

Але проекти, пов’язані з важкими обчисленнями, або з аналізом та управлінням великими об’ємами даних, ставлять рентабельність вище описаного підходу до організації архітектури та багатопоточного довкілля під сумнів. Операційні системи мають обмежену кількість потоків які можуть бути створені, до того ж процесори мають ще більш обмежену кількість потоків які можуть виконувати роботу паралельно (звичайно до 24). При цьому інше вирішення проблеми, реплікація API-серверів та СУБД, збільшує ціну на підтримку системи згідно до алгебраїчної прогресії, але не прискорює обчислення чи обробку даних на еквівалентну величину. При цьому збільшуються ризики, пов’язані із некоректним розподілом обчислень між

інстансами, тому також потрібна конфігурація реверс-проксі, оркестратору сервісів, а також оптимізація та конфігурація самого реплікованого чи розподіленого серверу СУБД. Згодом витрати на підтримку та невеликий приріст ефективності з масштабуванням системи зменшують отриманий від системи корисний ефект.

Одним з рішень цієї проблеми, яке може бути більш рентабельним, а, відповідно, підходящим, є використання нових моделей паралельної обробки: асинхронна обробка запитів та даних, модель “реактор”, реактивна модель, модель базована на акторах, та інші. Усі ці рішення дозволяють значно оптимізувати систему за параметром роботи що є корисною для системи. При традиційному (блокуючому) підході кожний потік виконання має припинити своє виконання у декількох містах: наприклад, при створенні та очікуванні відповіді на запит до бази даних. В свою чергу, названі підходи дозволяють частково або навіть повністю ліквідувати паузи у роботі потоків виконання. Одним з таких прикладів є використання асинхронної моделі прийому та обробки запитів з використанням асинхронних бібліотек вводу-виводу, що дозволяє значно зменшити навантаження на сервері (рис. 1.3) [1].

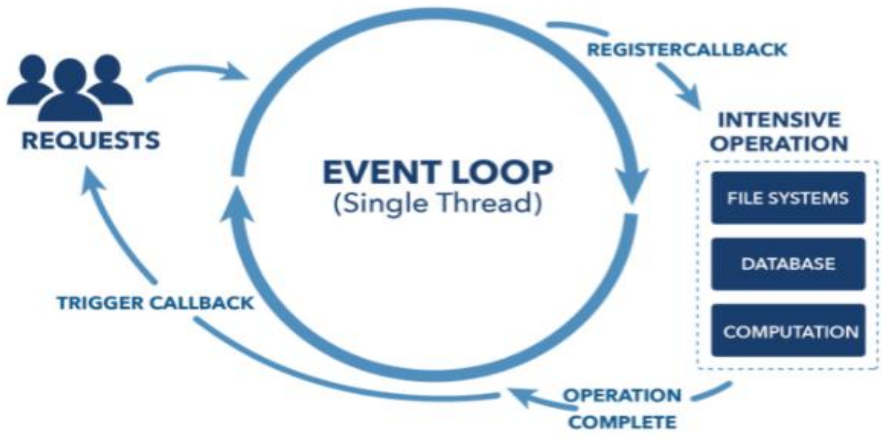


Рисунок 1.3 – Асинхронна модель однопоточного прийому та делегації обробки запитів (цикл обробки подій)

Іншою альтернативою в умовах, при яких обчислення є важкими (а, тобто, дуже довгими або інтенсивними з точки зору споживаної процесорної потужності), є створення кластерів для обчислень, що являють собою додатковий фізичний рівень (tier) у контексті вже описаної архітектурі. Тоді, рівень бізнес-логіки, раніше представлений одним фізичним серверним рівнем, який брав на себе як задачі обчислень, так і задачі, пов’язані з вводом і виводом, розділяється на різні фізичні рівні для серверів, які займаються задачами вводу-виводу та обчисленнями. Такий архітектурний зсув створив простір для створення великого ряду спеціалізованих інструментів, які націлені на підвищення ефективності систем, створених за такою архітектурою.

Тому причиною, з якої проекти з підвищеними вимогами до часу обчислень, розміру даних, що керуються, і т.п. націлені на використання методів, архітектурних підходів та інструментів інших категорій є рентабельність. Такі методології та підходи як реактивне програмування, кластерні обчислення, розподілені бази даних, реактивне програмування, NoSQL-довкілля, а також специфічні конфігурації програмних довкіль (наприклад, вибір певних реалізацій Garbage Collection чи самої JVM у JVM-довкіллях) є більш рентабельними при вирішенні таких задач, і в той же час вони є менш рентабельними (чи ефективними) при вирішенні задач більш типових категорій.

## 1.2 Визначення підходів та методів щодо реалізації високонавантаженої системі пошуку спеціалізованої інформації

Критичними аспектами в контексті задачі пошуку спеціалізованої інформації є:

- кількість потоків виконання що використовуються у ході обробки запитів;
- ефективність використання оперативної пам’яті;
- кількість запитів що є опрацьованими в єдиний момент часу;

– швидкість обробки даних (аналіз результатів на предмет відносної релевантності, фільтрація результатів, індексація, тощо).

Пошукова система являє собою складну систему, яка має доступ к величезній кількості файлів різних об’ємів, що будуть повністю чи частково використовуватися в якості відповідей на пошукові запити. Така система повинна мати можливість ефективно горизонтально масштабуватися, тобто становиться більш ефективною за допомогою встановлення допоміжних комп’ютерів, які будуть виступати у якості вузлів на які буде розподілятися навантаження. Важливо, також, відмітити, що обсяг дискового простору теж є однією з центральних проблем: база даних системи буде постійно поповнюватися новими архівами даних у швидкому темпі, і декількох заздалегідь підготовлених дисків швидко стане замало. Тому база даних системи також повинна мати підтримку горизонтального масштабування. Тобто, пошукова система має бути розподіленою системою.

Швидкість обробки запитів зумовлена декількома факторами, а саме:

– процесорною потужністю;
– кількістю ядер процесора;
– кількістю рівнів кешу процесора та їх обсягом;
– кількістю потоків що працюють водночас;
– обсягом оперативної пам’яті;
– частотою як процесору, так і чипів ОЗУ.

Тобто, швидкодія системи обумовлена як потужністю, так і завантаженістю системи в конкретний момент часу.

Кількість водночас опрацьованих запитів обумовлена тими ж факторами що и швидкодія: потужністю апаратного забезпечення та завантаженістю системи, але завантаженість системи включає в себе як зайняті потоки виконання, так і фізично допустиму кількість підключень. Зазвичай фізично допустима кількість підключень

на сервері є великою у порівнянні з іншими аспектами величиною, тому нею можна знехтувати.

Ефективність використання обчислювальних ресурсів обумовлена низькорівневою реалізацією мови програмування, програмного середовища, цільової платформою, конфігурацією платформи та програмного середовища, вибраними програмними алгоритмами та самою реалізацією програмного додатку, де останнє має набагато більшу вагу ніж інші: ряд найбільш популярних мов програмування та платформ доволі сильно ранжується при бенчмаркінгі в аспектах реалізації системи вводу-виводу та обчислень, але є багато прикладів високонавантажених систем які є ефективними при вирішенні поставлених задач, але при цьому використовують у середньому менш ефективні платформи та фреймворки. Більш важливим фактором є конфігурація платформи, ніж її вибір. Звісно, варто враховувати, що річ йде про платформи які надають приблизно однаковий набір можливостей, тобто середовище з одним потоком виконання (таке як JavaScript або Dart) яке масштабується за допомогою процесів не розглядається як альтернатива багатопоточному середовищу (такому як JVM (як платформа в цілому), Erlang VM).

Кількість водночас опрацьованих запитів, ефективність застосування апаратного забезпечення та швидкість обробки даних є як результатом горизонтального масштабування та вибору платформи, так і задіяних у процесі реалізації методів та підходів, а також конфігурації самої платформи. Представником підходу до ефективної утилізації апаратного забезпечення є реактивне програмування, що ставить перед собою мету ефективної обробки постійних джерел даних (у вигляді запитів включно). Прикладом конфігурації платформи може бути вибір JVM, мови програмування що буде компілюватися у Java байткод (тобто JVM-сумісної мови програмування), компілятора, серверного ПО, реалізації Garbage Collection, опцій Ahead Of Time та Just In Time компіляції і т.п.

Такі проблемні категорії як швидкість операцій вводу-виводу, швидкість обробки даних та керування дисковим простором є тими проблемами, які

вирішуються такою галуззю як BigData. Саме тому ця галузь повинна бути досліджена як потенційне джерело методів та інструментів для вирішення існуючих в даному контексті проблем.

## 1.3 Постановка задачі дослідження

Суб’єктом даного дослідження є пошукова система на стадії реалізації. Задачею дослідження є дослідження та аналіз методів, інструментів та програмних компонентів довкілля програмного засобу ціллю яких є вирішення ряду характерних для високонавантажених систем проблем, таких зменшення об’єму споживаної пам’яті, ефективність утилізації паралельного (конкурентного) довкілля системи, швидкість обробки даних та ефективність управління дисковими даними.

Вибір задачі дослідження обумовлений тим, що вищеописаний клас проблем стає дедалі все частішим при розробці веб-систем, але традиційні методи, що використовуються при реалізації таких систем, не являють собою ефективні рішення до вирішення цих проблем. Актуальність вирішення цієї проблеми стає все більш поширеною з кожним днем.

Для виконання задачі потрібно проаналізувати основні категорії методів та інструментів обробки даних у високонавантаженому середовищі, якими на даний момент є методи та інструменти галузі Big Data та передові методи та моделі щодо утилізації паралельного довкілля.

Після детального аналізу та порівняння категорій методів та інструментів обробки даних у високонавантажених середовищах потрібно вибрати найбільш підходящу з них та проаналізувати методи та інструменти що входять до вибраної категорії, після чого повинні бути вибрані ті методи, що будуть порівняні із стандартною моделлю роботи веб-серверів.

Після вибору цільових методів та інструментів має бути проведений ряд експериментів, що дадуть змогу проаналізувати відносну ефективність вибраних методів щодо стандартних. Експерименти при одних й тих же умовах щодо програмного середовища виконання та довкілля самої системи. Після проведення експериментів результати мають бути формалізовані, а самі результаті – порівняні один з іншим для отримання результатів щодо відносної ефективності цих методів.

# 2 **АНАЛІЗ BIGDATA ТА ЗАСТОСУВАННЯ СУМІЖНИХ ІНСТРУМЕНТІВ**

## 2.1 Визначення BigData

Назва галузі зазвичай асоціюється з отриманням (здобиччю), обробкою (аналізом) та управлінням даних, обсяг яких перевищує умовне, інтуїтивне "середньостатистичне" значення.

Відповідно до одного з джерел, термін `BigData` відноситься до наборів даних, розмір яких перевищує можливості типових баз даних (БД) по занесенню, зберігання, управління та аналізу інформації. При цьому варто врахувати, що світові репозиторії даних продовжують зростати з кожною хвилиною.

Проте, термін передбачає дещо крім обробки великих щодо середнього значення обсягів даних: що організації не просто створюють, генерують і реплікують величезні обсяги даних. Вони виробляють дані в форматі, що не відповідає традиційному формату обробки даних (наприклад, у форматі бази даних, яка перебуває під управлінням СУБД з сімейства SQL), наприклад у вигляді веб-журналів, відеозаписів, текстових документів, машинного коду, геопросторових даних в специфічному для домену форматі і т.п. В результаті цього корпорації можуть не мати відповідних для вирішення поставлених завдань інструментів (наприклад, установки зв'язків між частинами даних і аналізу для виведення корисного інформації). При цьому варто враховувати динамічну природу даних, особливо в високонавантажених системах: вони оновлюються все частіше; в результаті цього традиційні методи і інструменти для управління, обробки та аналізу даних в своєму розвитку не можуть дотримуватися такого темпу, що і є грунтом для створення технологій і інструментів, що відносяться до категорії BigData. Тому сам термін має на увазі роботу з даними

великих обсягів (порівнюючи з середньостатистичним максимально доступним об'ємом на носії, а також оперативної пам'яті) і різноманітного як формату, так і складу, яка, до того ж, має динамічну природу (часто оновлюється) і розташовується в ряду різних джерел з метою збільшення ефективності роботи, створення нових продуктів або поліпшення вже існуючих на підставі результатів обробки, а також для збільшення конкурентоспроможності.

## 2.2 Генезіс BigData

У мережі існує безліч джерел, робота з якими може звести нанівець можливість використання традиційних підходів і інструментів для обробки цих даних. Такими можуть виступати дані, які безперервно надходять з різного роду сенсорів, потоки даних з різних інтернет-сервісів (стрічка новин, повідомлення та події, що відбуваються в контексті соціальної мережі), потоки даних місцезнаходження GPS-модулів, централізовані репозиторії даних, що відносяться до певного домену (наприклад, регістр осіб, оформлених як ФОП) і т.п. Глобальний розмах і поширення подібних технологій і є основоположною для впровадження BigData практично в усі аспекти людського життя, основними з яких є, в порядку пріоритету: наукова діяльність, комерційна діяльність і державне управління.

З плином часу дані все більше віддаляються від своєї початкової статичної і застарілою природи, що мала на увазі втрату цінності після настання цільового події (наприклад, відвідування концерту, де в якості даних може виступати ціна). Вони стали сировиною для знаходження нових взаємозв'язків і створення нових припущень, які, в свою чергу, можуть бути економічно ефективними. Це означає те, що їх можна застосувати для вирішення суміжних або навіть зовсім не пов'язаних завдань, де вони будуть сировиною, над яким будуть проводитися аналітичні процеси іншого виду,

роду або категорії, які, в свою чергу, вже направлені на вирішення завдань з цільової сфери.

Найбільший внесок галузь внесла в розвиток так званого прогнозування, де ці прогнози будуються на основі потрібного (зазвичай в кілька разів перевершують традиційно прийняті) обсягу даних. Типовим прикладом може послужити прогноз курсу валют, прогноз товарів, які найчастіше супроводжують один одного при придбанні, або навіть прогноз потенційних злочинів на основі проаналізованих патернів поведінки. Все це, в свою чергу, є грунтом для такої галузі комп'ютерних наук як штучний інтелект. При створенні подібних прогнозів застосовуються технології і інструменти, засновані на застосуванні ряду математичних прийомів, що оперують на заданих габаритах даних. Основою для надійності й ефективності таких систем є все ті ж дані, надходження яких має забезпечуватися для калібрування алгоритмів, і, отже, створюються за їх допомогою прогнозів.

## 2.3 Характеристики BigData

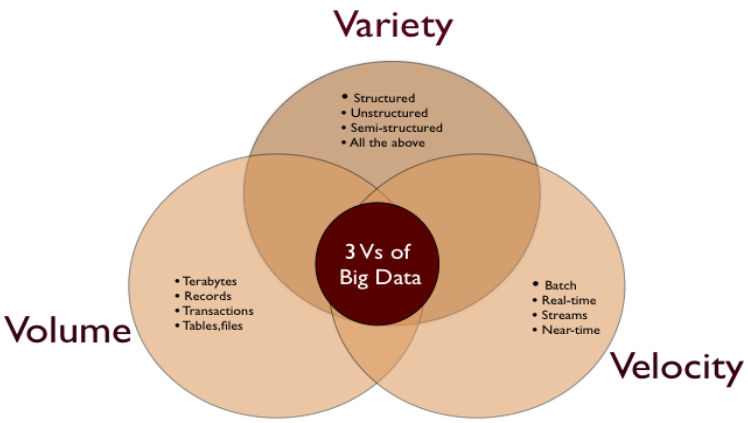


Рисунок 2.1 – BigData 3V

BigData описується в основному трьома (пізніше чотирма) відцентровими характеристиками: об'ємом, швидкістю і різноманіттям.

Першою характеристикою є обсяг. Він описує кількість і габарити генеруються і збережених даних. Ці значення визначають цінність і потенційне застосування, а також факт того чи може репозиторій (в концептуальному сенсі) розглядатися як BigData чи ні.

Обсяг є найбільш асоціюється з BigData характеристикою, тому що обсяг може бути великим. Мова тут йде про обсяг і \ або кількості даних які можуть приймати недосяжні навіть для спеціалізованих систем значення.

В даний час спостерігається експоненціальне зростання в сховищах даних, оскільки дані тепер все частіше є не просто текстовими. Дані зберігаються в форматі відео, музики та великовагових зображень в соціальних мережах. Часто терабайти і петабайт використовуються в системах зберігання для підприємств. У міру зростання баз даних додаток і архітектура, створені для підтримки цих даних, повинні часто піддаватися повторному аналізу і перегляду вже впроваджених рішень. Тому великовагові масиви даних дійсно є частиною BigData.

Наприклад, в контексті мережі Facebook повинні зберігатися фотографії, надіслані користувачами. При цьому важливо відзначити, що кількість користувачів Facebook перевищує чисельність населення Китаю. Кожен з цих користувачів може відправити необмежену безліч зображень. В цілому, Facebook зберігає порядку 250 мільярдів зображень.

В контексті BigData значення обсягу описує подібного роду габарити. У міру розвитку Галузі обсяги репозиторіїв даних в спеціалізованих системах будуть лише зростати.

Якщо акцентувати увагу на галузі Internet of Things, в якій прогрес в галузі BigData є одним з першочергових факторів для власного розвитку, то зручно

використати приклад зі зняття ряду показань, одержуваних за допомогою сенсорів або лічильників, підключених до єдиної мережі. Всі отримані за допомогою сенсорів дані будуть продовжувати складатися, поступово заповнюючи репозиторій новими даними.

Питання при цьому полягає в темпі надходження даних і їх фінальний обсяг. За оцінками Gartner, Cisco і Intel, число підключених пристроїв в середньому IoT середовищі буде варіюватися від 20 до 200.

Розглянемо проблему обсягу генеруємих одним пристроєм даних. У якості предметної області нехай виступає датчик температур. З рівнем деталізації в одну хвилину (один вимір в хвилину) обсяг становитиме 525 950 точок даних на рік. Якщо ці датчики будуть використовуватися для контролю за температурою в спеціалізованому спорудженні (наприклад, машинобудівний завод), то можуть утилізуватися близько тисячі датчиків – кількість залежить від об’єму самої споруди.

Об'єм як характеристика BigData має місце бути не тільки в спеціалізованих галузях за типом IoT. Як суб'єкт аналізу можна сконцентрувати увагу на додатках для смартфонів, а саме – додаток для нотаток. Все більша кількість виробників та постачальників додатків управляються даними в Cloud, тому користувачі можуть отримувати доступ до своїх нотаток на різних пристроях. Оскільки в багатьох Додатках використовується модель Freemium, де безкоштовна версія використовується в якості лідера втрат для преміум-версії, постачальники додатків на основі SaaS, як правило, мають багато даних для зберігання.

Наприклад, у додатку Todoist приблизно 10 мільйонів активних установок, згідно Google Play (що не має на увазі все установки в Інтернеті, як і iOS). Кожен користувач має списки заміток, при цьому всі дані повинні бути надійно збережені. Це, а також багато інших додатків все ще зберігають набагато більше даних, ніж практично будь-який додаток, створений і опублікований на ринку кілька років тому.

Займають велику частку в цій сфері також всі збори внутрішніх даних підприємств – від енергетики до охорони здоров'я та національної безпеки. Всі ці галузі генерують і збирають величезні обсяги даних [2].

База даних пошукової системи являє собою величезний репозиторій даних, тому можна вважати, що характеристика обсягу застосовна до системи, що розглядається.

В якості другої характеристики галузі виділяють швидкість генерації даних. Швидкість є мірою того наскільки швидко дані надходять в цільову систему.

Зростання даних і стрімкий розвиток соціальних мереж змінили те, як суспільство стало сприймати дані. Раніше можна було вважати, що дані вчорашнього дня є найбільш актуальними. Сьогодні люди відповідають в соціальних мережах, щоб повідомити їм про останні події. Іноді в соціальних мережах старі повідомлення (твіт, оновлення статусу і т. Д.) не цікавлять користувачів. Вони часто відкидають старі повідомлення і звертають увагу на останні оновлення. Рух даних тепер відбувається майже в реальному часі, а вікно оновлення скоротилося до частки секунд. Така швидкість оновлення даних є однією з відцентрових характеристик BigData.

Продовжуючи аналіз високого навантаження систем Facebook – його користувачі завантажують більш 900 мільйонів зображень в день в сумі, що дорівнює 328,5 млрд за рік. Це приблизно на третину більше ніж весь обсяг збережених зображень за 2017 рік (250 млрд).

Поширеним явищем також є отримання і аналіз інформації про резонанс викликаному маркетинговою кампанією бренду. Одним із прикладів реалізації є отримання потоку даних релевантних постів в Twitter, які пізніше є суб'єктом "аналізу настроїв".

Аналіз мережевих пакетів на предмет застосування відповідних заходів кібербезпеки також є живим прикладом "velocity". За глобальної мережі проходять величезні обсяги даних кожну секунду. Для корпоративної IT-команди частина цього

потоку повинна проходити через брандмауери в корпоративну мережу. Через зростання числа кібератак, кіберзлочинності і кібершпіонажу небажаний трафік може бути прихований в загальному потоці даних, що проходять через міжмережевий екран. Щоб уникнути витоків засекречених даних, цей потік даних повинен бути досліджений і проаналізований на предмет аномалій, патернів поведінки, які розцінюються як джерело потенційної загрози. Це стає все важче, оскільки все більше і більше даних захищаються за допомогою шифрування. У той же час, зловмисники мають можливість приховувати шкідливі пакети в контексті всього потоку зашифрованих пакетів.

Продовжуючи тему значення інтернету речей в BigData – з розвитком галузі IoT кількість підключених датчиків буде в світі буде рости. Вони є постійними джерелами інформації яку вони будуть передавати через мережу майже постійною швидкістю. У міру збільшення кількості датчиків збільшується і загальний потік даних.

База даних пошукової системи є суб’єктом постійного оновлення та поповнення новими файлами, тому за характеристикою швидкості оновлення даних пошукова система може вважатися відповідною для категоризації як BigData-пов’язана система [2].

Третьою ключовою характеристикою BigData є різноманітність даних.

Дані можуть зберігатися в багатьох форматах. Наприклад, у вигляді файлу бази даних, документа Excel або CSV, або, по суті, він може бути збережений як простий текстовий файл. Іноді дані навіть не в традиційному форматі, як ми припускаємо, вони можуть бути в формі відео, SMS, PDF або в спеціальному форматі, створеним спеціально для вирішення вузько направленого ряду завдань. Завданням організації в цьому плані є організація структури і підтримки інфраструктури для зберігання і обробки таких даних. Простіше коли дані в одному форматі, але в більшості випадків це не так. У реальному світі дані зберігаються в різних форматах, і це проблема, яка

долається за допомогою розвитку галузі BigData, акцентуючи особливу увагу на техніках, технологіях і інструментах.

Різноманітність визначає тип і характер даних. Це дає можливість людям, які займаються обробкою даних, ефективним чином отримувати і використовувати вихідну інформацію.

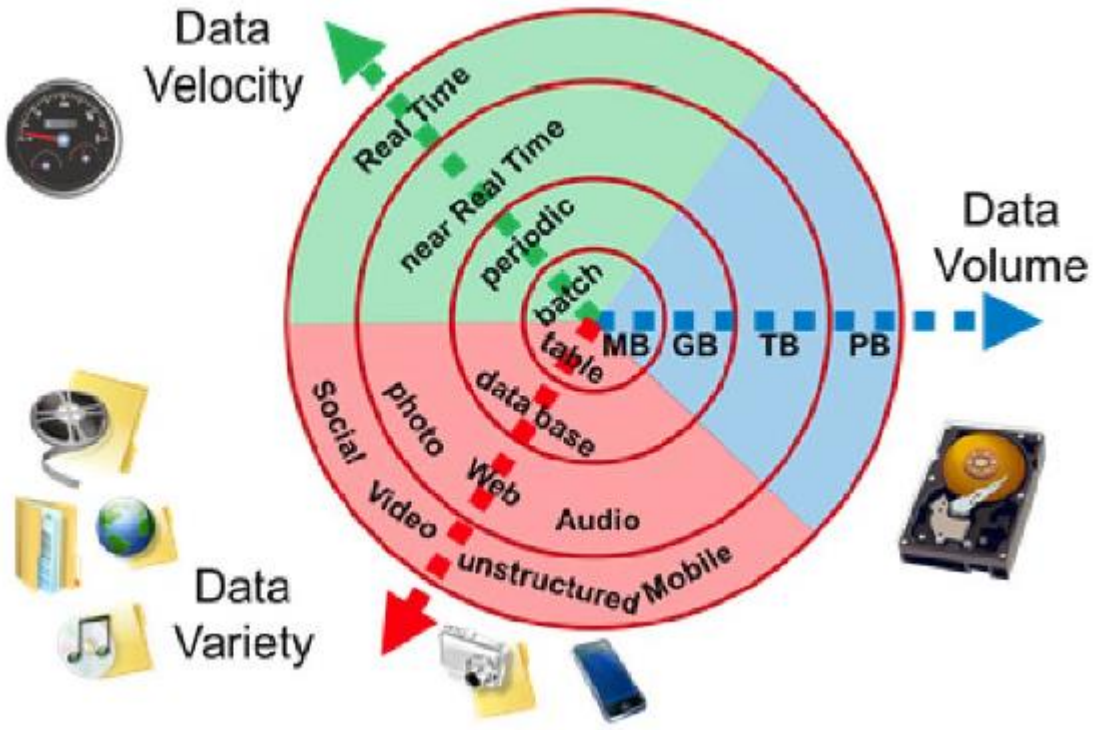


Рисунок 2.2 – Візуальне уявлення ряду характеристик BigData 3V

Згідно до різноманітності даних у контексті пошукової системи – це залежить від вимог до пошукової системи та її реалізації. Проте, більшість популярних пошукових систем має змогу виконувати пошук по ряду заданих лексем у різного роду файлах (PDF, онлайн-книги, CSV файли), а не тільки на веб-сторінках, тому пошукову

систему можна вважати придатним для BigData проекту кандидатом за цією характеристикою [2].

Також виділяють ще одну характеристику BigData яку відносять до однієї з ключових – достовірність (veracity).

Достовірність в BigData відноситься до спотворень, шуму і відхилень в даних. Чи є дані, які зберігаються і добуваються, значущими для аналізованої проблеми? Серед фахівців відома думка, що достовірність в аналізі даних є найбільшою проблемою, навіть якщо порівнювати її з 3V. При визначенні своєї стратегії щодо BigData необхідно, щоб команда і партнери працювали над тим, щоб забезпечити чистоту ваших даних і процеси, що запобігають накопиченню «брудних» даних в розроблюваних і підтримуваних системах.

Значимість достовірності висока через те, що дані відіграють вирішальну роль в ухваленні рішень і розробці стратегії в різних галузях, таких як роздрібна торгівля, охорона здоров'я, виробничі підрозділи, компанії-розробники програмного забезпечення і т.д. Однак ті ж дані можуть бути прийняті як інвалідні, якщо вони ненадійні або неточні. Невірні або неточні дані призводять до серйозних проблем, таких як спотворення інформації і помилкові / неправильні рішення.

Достовірність можна вважати ключовим елементом до релевантності даних, якщо вважати достовірність як ступінь самої релевантності. Але такого терміну як “достовірність” у чистому сенсі терміну у контексті пошукової системи нема, тому що пошукова система передбачає пошук релевантних файлів, де кінцевий користувач сам визначає чи наданий файл відповідає його запиту. Тому за характеристикою достовірності даних пошукову систему не можна вважати BigData-пов’язаною системою[3].

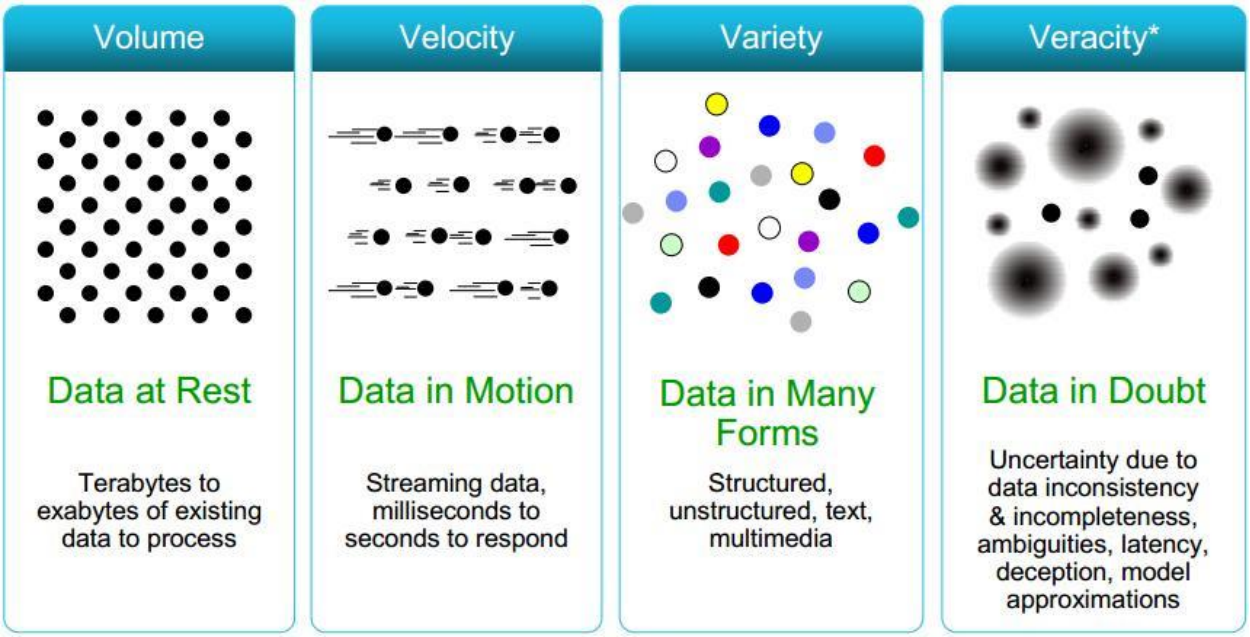


Рисунок 2.3 – Достовірність як частка 4V

Деякі галузі особливо залежні від даної характеристики, самої явною з яких є охорона здоров'я. Як приклад виступає пакет невірних дані, які свідчать про те, що конкретний діагноз буде відповідати певному набору симптомів у пацієнтів. Крім того, лікарі продовжать випускати лікування, засноване на цьому дослідженні, тільки для того, щоб потім зрозуміти, що воно не працює або може бути небезпечно для здоров'я пацієнтів. Неточні дані в області медицини або охорони здоров'я можуть виявитися привести до зворотного, тобто до негативного ефекту.

Неточні дані або обробка таких даних становлять загрозу у вигляді прийняття помилкових рішень у багатьох галузях. Це вказує на те, що достовірність даних вітальних для отримання результатів, які є основою для прийняття рішень.

Наступною характеристикою є мінливість (volatility) даних. Мінливість в BigData ставиться до того, як довго дійсні дані і як довго вони повинні зберігатися.

Присутність мінливості даних безпосередньо залежить від конкретного домену, який є втягнутим в галузь BigData. Найчастіше завдання, пов'язані з BigData увазі роботу з повною вибіркою даних, можливо зі спеціальною обробкою них в залежності від моменту часу в який вони були згенеровані. В інших доменах цей час може зовсім не враховуватися (наприклад, при дослідженні ефекту певних хвороб – якщо є зареєстрований випадок і повне формальне опис ходу хвороби, то факт того коли людина опинилася вражений не є фактором, що входять до уваги при аналізі). У свою чергу, в контексті, наприклад, зняття показань температури в цеху (з великою швидкістю оновлення даних), важливий своєчасний контроль температур і негайне реагування в разі якщо настане критична ситуація. Для цього важливі лише найбільш актуальні дані: якщо компанія не ставить перед собою цілей детального аналізу, а робочий процес всередині повністю налагоджений, то можна з упевненістю сказати, що у таких даних короткий термін значущості [4].

Мінливість даних є основою для релевантності результатів на кожен конкретний запит (такий як “сьогоднішні новини”), тому за характеристикою мінливості пошукову систему можна вважати BigData-пов’язаним проектом.

## 2.4 Інструменти та методи галузі BigData

Ефективна робота з BigData має на увазі дотримання такого ряду принципів:

- Горизонтальна масштабованість. Система повинна мати можливість ставати більш ефективної при додаванні до неї нових вузлів.
- Відмовостійкість. Принцип горизонтальної масштабованості має на увазі, що машин в кластері може бути багато. Це означає, що частина цих машин буде гарантовано виходити з ладу. Методи роботи з BigData повинні враховувати можливість таких збоїв і переживати їх без будь-яких значущих наслідків.

– Локальність даних. У великих розподілених системах дані розподілені на великій кількості вузлів. Якщо дані фізично знаходяться на одному сервері, а обробляються на іншому – витрати на передачу даних можуть перевищити витрати на саму обробку. Тому одним з найважливіших принципів проектування BigData-рішень є принцип локальності даних – дані повинні оброблятися на тому ж сайті, на якому вони зберігаються.

Сучасні засоби роботи з BigData так чи інакше йдуть цим трьом принципам. Для того, щоб їх дотримуватися – необхідні відповідні усталені методи, парадигми та інструменти. Однією з основних парадигм (підходів) BigData є MapReduce.

MapReduce – це модель розподіленої обробки даних, запропонована компанією Google для обробки великих обсягів даних на комп'ютерних кластерах.

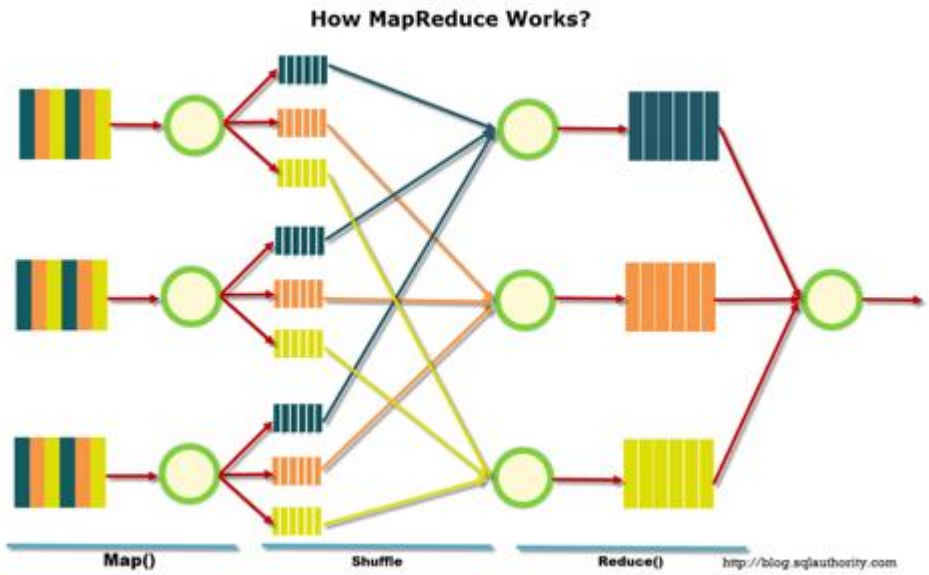


Рисунок 2.3 – Принцип роботи MapReduce

MapReduce передбачає, що дані організовані у вигляді деяких записів. Обробка даних відбувається в 3 стадії:

- стадія "Map". На цій стадії дані проходять попередню обробку за допомогою функції map (), яку визначає користувач (прикладної інженер, який розробляє систему). Робота цієї стадії полягає в попередній обробці і фільтрації даних. Робота дуже схожа на операцію map в функціональних мовах програмування; – призначена для користувача функція застосовується до кожної вхідного відрізку. Семантика самої функції map увазі трансформація вихідних даних в цільової формат, слідуючи заздалегідь визначеним алгоритмом. Функція map () застосована до однієї вхідного відрізку може результувати в набір пар ключ-значення кількістю від нуля до N, тобто в безліч. Дані, що зберігаються у вигляді ключів і значень визначаються користувачем. Дані з ідентичним ключем в майбутньому потраплять в один екземпляр функції reduce;
- стадія "Shuffle". У цій стадії результати функції map групуються за ознакою ідентичності ключа Надалі ці кошики прослужать входом для reduce;
- стадія "Reduce". Функція reduce задається користувачем і обчислює фінальний результат для кожної групи. Безліч всіх значень, які були повернуті функцією reduce (), є фінальним результатом MapReduce-завдання.

Одним з основоположних інструментів для роботи з BigData є Apache Hadoop. Спочатку Hadoop був, в першу чергу, інструментом для зберігання даних і запуску MapReduce-задач, зараз же Hadoop є великий стек технологій, так чи інакше пов'язаних з обробкою великих даних (не тільки за допомогою MapReduce).

Hadoop є проектом фонду Apache Software Foundation, представленим у вигляді вільно поширюваним набором утиліт, бібліотек і фреймворком для розробки і виконання розподілених програм, що працюють на кластерах з сотень і тисяч вузлів. Використовується для реалізації пошукових і тематичних механізмів багатьох високонавантажених веб-сайтів, в тому числі, для Yahoo! і Facebook. Розроблено на Java в рамках обчислювальної парадигми MapReduce [5].

Однією з основних цілей Hadoop спочатку було забезпечення горизонтальної масштабованості кластера за допомогою додавання недорогих вузлів, без вдавання до потужних серверів і дорогим мереж зберігання даних. Функціонуючі кластери розміром в тисячі вузлів підтверджують здійсненність і економічну ефективність таких систем, так, станом на 2011 рік відомо про великих кластерах Hadoop в Yahoo (більше 4 тис. Вузлів з сумарною місткістю зберігання 15 петабайт), Facebook (близько 2 тис. Вузлів на 21 петабайт) і EBay (700 вузлів на 16 петабайт).

Основними компонентами Hadoop є HDFS та YARN.

Hadoop Distributed File System (HDFS) – файлова система, призначена для зберігання файлів великих розмірів, по блоках розподілених між вузлами обчислювального кластера. Всі блоки в HDFS (крім останнього блоку файлу) мають однаковий розмір, і кожен блок може бути розміщений на декількох вузлах, розмір блоку і коефіцієнт реплікації (кількість вузлів, на яких повинен бути розміщений кожен блок) визначаються в налаштуваннях на рівні файлу. Завдяки реплікації забезпечується стійкість розподіленої системи до відмов окремих вузлів.

Hadoop YARN (Yet Another Resource Negotiator) – модуль, що з'явився з версією 2.0 (2013), що відповідає за управління ресурсами кластерів і планування завдань. Якщо в попередніх випусках ця функція була інтегрована в модуль MapReduce, де була реалізована єдиним компонентом (Job Tracker), то в YARN функціонує логічно самостійний демон – планувальник ресурсів (Resource Manager), абстрагується все обчислювальні ресурси кластера і керуючий їх наданням додатків розподіленої обробки. Працювати під керуванням YARN можуть як MapReduce-програми, так і будь-які інші розподілені додатки, що підтримують відповідні програмні інтерфейси; YARN забезпечує можливість паралельного виконання декількох різних завдань в рамках кластера і їх ізоляцію (за принципами мультіарендності).

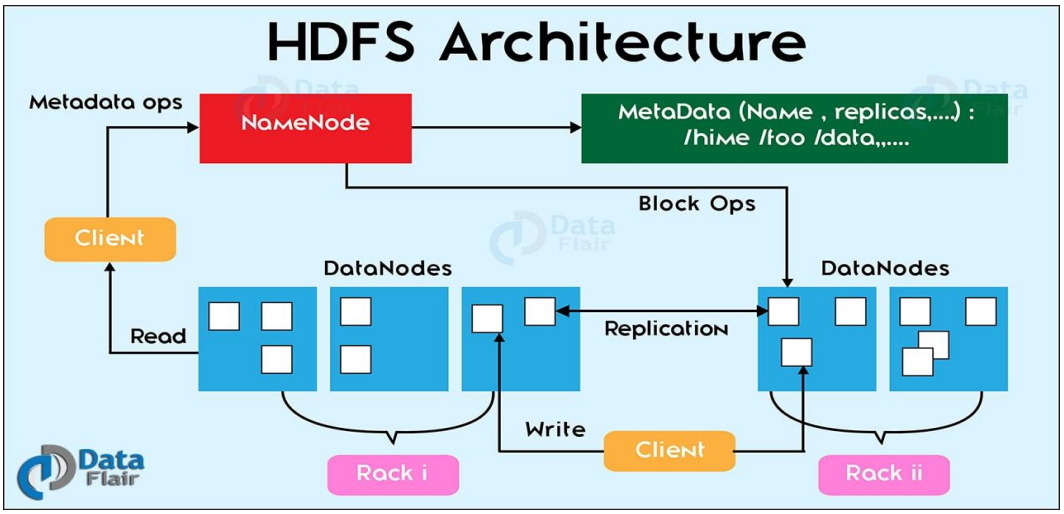


Рисунок 2.4 – Архітектура розподіленої файлової системи Hadoop

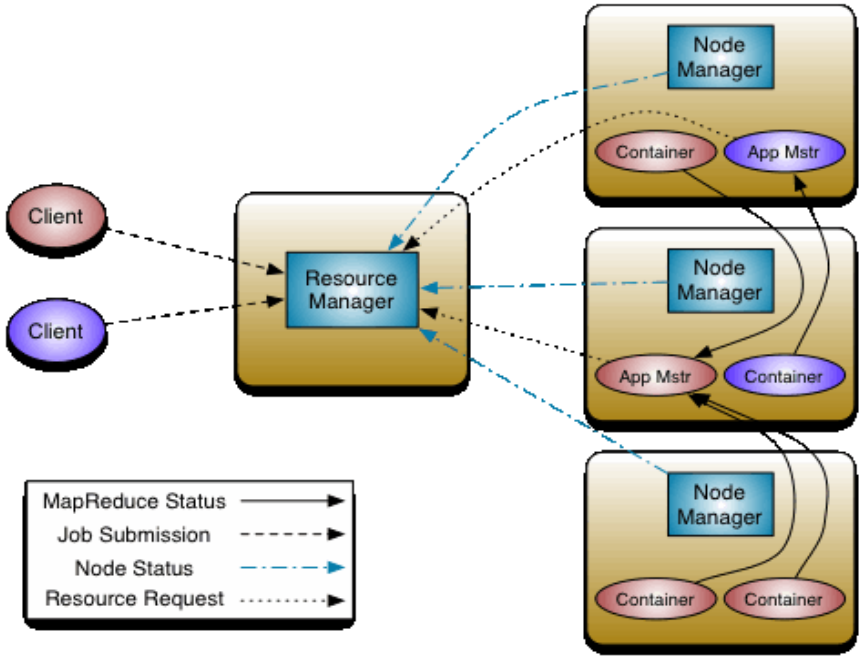


Рис. 2.5 – Архітектура Hadoop YARN

Hadoop ефективно вирішує завдання управління обчислювальними кластерами і горизонтального масштабування системи, але через те що його реалізація MapReduce має на увазі роботу з дисковим сховищем, що виконуються MapReduce операції вимагають значних витрат часу на виконання. Завдання ефективної обробки даних в свою чергу краще вирішує інструмент під назвою Spark.

Apache Spark – фреймворк з відкритим вихідним кодом для реалізації розподіленої обробки неструктурованих і слабо структурованих даних, що входить в екосистему проектів Hadoop. На відміну від класичного обробника з ядра Hadoop, Spark використовує спеціалізовані примітиви для рекурентної обробки в оперативній пам'яті, завдяки чому дозволяє отримувати значний виграш в швидкості роботи для деяких класів задач.

Apache Spark може працювати поверх Hadoop-контрольованого середовища і використовуватися як інструмент обробки і аналізу даних.

Вкрай ефективним і популярним інструментом роботи з керуванням неструктурованих даних є Apache Cassandra. Apache Cassandra є розподіленою системою управління базами даних, що відноситься до класу NoSQL-систем і розрахованої на створення високо масштабованих і надійних сховищ величезних масивів даних, представлених у вигляді хешу. Розподілена hash-система забезпечує практично лінійну масштабованість при збільшенні обсягу даних. Використовує модель зберігання даних на базі сімейства стовпців, чим відрізняється від систем, подібних MemcacheDB, які зберігають дані тільки в зв'язці «ключ – значення», можливістю організувати зберігання хеш з декількома рівнями укладення. Відноситься до категорії стійких до відмов СУБД: поміщені в базу дані автоматично реплікуються на кілька вузлів розподіленої мережі або навіть рівномірно розподіляються в декількох дата-центрах, при збою вузла його функції на льоту підхоплюються іншими вузлами, додавання нових вузлів в кластер і оновлення версії

Cassandra проводиться на льоту, без додаткового ручного втручання і переконфігурації інших вузлів.

Вищевказаний ряд інструментів є набором найбільш популярних і часто використовуваних рішень для реалізацій проектів пов'язаних зі зберіганням, управлінням і обробкою масивних наборів структурованих і неструктурованих даних. Всі вони так чи інакше націлені на підтримку горизонтальній масштабованості, тобто на збільшення ефективності роботи зі збільшенням кількості доступних в системі вузлів в прямо пропорційному відношенні. Ряд цих інструментів може бути використаний для ефективного вирішення завдань при реалізації пошукової системи, де розподілена файлова система Hadoop може бути використана для зберігання файлів різної структури і формату, Apache Spark може бути використаний для вирішення завдань аналізу даних на предмет зважування релевантності при подальшому прийнятті системою рішення повернення того чи іншого файлу або ряду даних в якості відповіді користувачеві пошукової системи, а Apache Cassandra може використовуватися в якості сховища даних, що мають форма т, що відповідає формату збережених в базі даних записів, наприклад при зберіганні метаданих пошуку для подальшого визначення більш релевантних для заданих запитів даних.

Маючи усе це на увазі, важливо відмітити, що реалізація пошукової системи за допомогою спеціалізованих BigData інструментів тягне за собою повну зміну архітектури проекту, добавляючи ще один, кластерний рівень, на якому буде проходити обробка даних. При цьому, це рішення не гарантує вирішення поставлених проблем з високою завантаженістю системи (тобто опрацювання великої кількості паралельних запитів водночас), а лише адресують проблеми управління і обробки великих об’ємів даних якими система вже може оперувати. Якщо кінцевий результат покаже, що ефективність системи за цільовими характеристиками не виросла на очікувану величину, то той рівень апаратних та витрат на розробку, що використовувався для вирішення задачі, не виправдає себе. У зв’язку з цим потрібно

проаналізувати альтернативні рішення щодо проектування і реалізації системи, що будуть дозволяти абстрагуватися від інструментів на рівні програмного коду, а також витрати на які становитимуть менше значення ніж витрати на реалізацію BigData рішення.

# 3 **АНАЛІЗ МЕТОДІВ ПАРАЛЕЛЬНОГО ПРОГРАМУВАННЯ**

## 3.1 Визначення, роль та проблеми паралельних обчислень

Паралельне обчислення – це тип обчислювальної архітектури, в якому кілька процесорів одночасно виконують обчислення. Паралельні обчислення допомагають виконувати інтенсивні обчислення, поділяючи навантаження на більш ніж один процесор, які мають змогу виконувати обчислення одночасно [6].

Розробка паралельного програмного забезпечення традиційно вважається великою витратою часу та зусиль. Це можна значною мірою віднести до властивої складності конкретизації та координації паралельних завдань, відсутності портативних алгоритмів, стандартизованих середовищ та наборів інструментів для розробки програмного забезпечення. У контексті високої швидкості розвитку мікропроцесорів сумнів необхідності докладання великих зусиль для використання паралелізму як засобу прискорення роботи додатку збільшується. Зрештою, якщо для розробки паралельної програми потрібні два роки, то за цей час основна апаратна та / або програмна платформа стає застарілою, і зусилля з розвитку явно марно витрачаються.

Однак є деякі безпомилкові тенденції в дизайні апаратних засобів, які вказують на те що одно процесорні (або неявно паралельні) архітектури можуть не мати можливості підтримувати швидкість реалізованих приростів продуктивності в майбутньому. Це є наслідком відсутності неявного паралелізму, а також інших вузьких місць, таких як шлях даних та пам'ять. В той самий час, стандартизовані апаратні інтерфейси скоротили час обороту від розвитку мікропроцесора до паралельної машини на основі мікропроцесора. Крім того, було досягнуто значного

прогресу в стандартизації середовищ програмування для забезпечення їхнього більш тривалого життєвого циклу [7].

Загальна швидкість обчислення визначається не тільки швидкістю процесора, але і здатністю системи пам'яті подати до нього дані. Хоча частота тактових частот процесорів високого класу зросла приблизно на 40% на рік за останнє десятиліття, доступ до DRAM покращився лише приблизно 10% цей інтервал. У поєднанні зі збільшенням інструкцій, що виконуються за один цикл, цей розрив між швидкістю процесора та пам'яттю представляє надзвичайно вузьке місце з точки зору оптимізації швидкості виконання програми. Це зростаюча невідповідність між швидкість процесора та затримкою DRAM, як правило, поєднуються з ієрархією набагато швидших пристроїв пам'яті, званих кешами, які покладаються на локальність посилання на дані для забезпечення більш високої продуктивності системи пам'яті. Крім затримки, ефективна пропускна здатність мережі між

DRAM та процесором створюють інші проблеми для стійких обчислень.

Загальна продуктивність системи пам'яті визначається часткою загальних запитів на пам'ять, яку можна задовольнити за допомогою кешу. Паралельні платформи, як правило, покращують продуктивність системи пам'яті, оскільки вони забезпечують більші сукупні кеші, та більшу пропускну здатність сукупної пам'яті до системи пам'яті. Крім того, принципи, які лежать в основі паралельних алгоритмів, а саме локальність даних посилання, також надають кешовані послідовні алгоритми. Цей аргумент можна поширити на диски, де паралельні платформи можуть використовуються для досягнення високої сумарної пропускної здатності до вторинного сховища. Тут паралельні алгоритми дають уявлення про розвиток поза профільних обчислень. Дійсно, одні з найбільш швидко зростаючих областей додатків паралельних обчислень на серверах даних (сервери баз даних, веб-сервери) покладаються не стільки на високі сукупні показники обчислень, скільки на здатність викачувати дані швидшими темпами [7].

По мірі розвитку мережевої інфраструктури почало формуватися бачення використання Інтернету як одного великого неоднорідного паралельного / розподіленого обчислювального середовища. Багато додатків природно піддаються таким парадигмам обчислень. Деякі з найбільш вражаючих програм масово паралельних обчислень були створені в контексті широкомасштабних розподілених платформ. Проект SETI використовує потужність великої кількості домашніх комп'ютерів для аналізу електромагнітних сигналів з космосу. Інші подібні зусилля намагаються розділити надзвичайно великі цілі числа та вирішити великі проблеми дискретної оптимізації [7].

У багатьох додатках є обмеження щодо розташування даних та / або ресурсів у мережі Інтернет. Прикладом такої програми є видобуток великих комерційних наборів даних, що поширюються на відносно низької пропускної здатності мережі. У таких програмах, навіть якщо обчислювальна потужність доступна для виконання необхідної задачі, не вдаючись до паралельних обчислень, збирати дані в центральному місці неможливо. У цих випадках мотивація до використання паралелізму походить не лише від потреби в обчислювальних ресурсах, а й від нездійсненності або небажаності альтернативних (централізованих) підходів [7].

## 3.2 Аналіз моделі “1 потік – 1 запит”

У 90-х роках (і по сьогоднішній день) модель багатопоточності, якою користувалися сервери, використовувала багато потоків, життєвий цикл яких був надзвичайно короткочасним. Потік створюється, отримує вхідний запит, опрацьовує його, повертає відповідь, а потім звільняється. Тобто, в архітектурі «потоку на запит» система виділяє новий потік для обробки кожного вхідного запиту, і коли цей запит завершений, він звільняє системний потік. Це, як правило, означає, що на сервері багато потоків виконання (з їх швидким створенням та звільненням) (рис. 3.1).

Під час роботи з системами з низьким показником QPS це не є величезною проблемою, оскільки накладні витрати на створення та звільнення потоків не настільки значні – особливо з базами даних, на взаємодію з якими витрачається набагато більше часу.

Ця модель є типовою для більшості веб-проектів, її перевагами є відносна простота, відносна відмовостійкість (обумовлена тим, що всі можливі помилки у процесі виконання можуть бути обумовлені лише несправністю самої програми або програмного середовища, тому що, завдяки характерній ізольованості між потоками, ймовірність виникнення помилки, пов’язаної з конкурентним виконанням дуже і дуже мала), легкість та прозорість горизонтальної масштабуємості (зазвичай при використанні якогось реверс-проксі, такого як nginx, для розподілу навантаження на вузли) [9].

Проте ця модель має і свої недоліки. Найбільш показниковим з недоліків є навантаження на пам’ять, що визване кількістю водночас “живих” потоків виконання. Потоки, наприклад, у JVM середовищі є досить дорогими, зазвичай використовують сегмент пам’яті розміром в 1 мегабайт кожен, незалежно від того, чи вони активні, чи простоюють. Якщо надавати на кожен запит свій власний потік, то він, як правило, буде заблокованим у ряду місць в програмі, не роблячи ніякої корисної роботи, а лише використовуючи цінні та скінченні системні ресурси. Зрештою, фреймворк повинен або перестати приймати нові запити (тому що він більше не може створити потоки) або почати відключати старі з'єднання.

Масштабованість у даному контексті представляє собою сумлінне вирішення проблеми, тому що хоч воно й просте та прозоре, проте не дуже ефективне порівнюючи з горизонтальним масштабуванням системи або реплікацією додатку при використанні більш сучасних методів роботи з багатопоточним середовищем виконання.

Не менш серйозною проблемою в даному контексті є великі накладні витрати на створення та звільнення потоків виконання. Це пов’язано з тим, що відносна швидкодія сучасних систем управління базами даних та виконуваних інструкцій високорівневих мов програмування стає все більш та більш оптимальною (тобто усе становиться швидше). Проте самі потоки були та залишаються приголомшливо важкими, особливо якщо мати на увазі роль операційної системи при такої наданні високорівневої абстракції для роботи з ними та підтримки безпечної та передбачуваної поведінки при їх роботі (виконанні наданих інструкцій).

Тому можна постановити, що серверна архітектура “потік на з’єднання” підходить типу більшою мірою до невеликих веб-додатків, але не є достатньо оптимальним варіантом при розробці високонавантаженої системи зі збільшеними вимогами до системи вводу-виводу або до алгоритмічних обчислень.

Рисунок 3.1 – Архітектура типу “1 потік – 1 запит”

## 3.3 Аналіз моделі пула потоків для обробки запитів

Розвиток баз даних та систем управління базами даних, ефективності та швидкодії коду який є продуктом популярних високорівневих мов програмування, апаратного забезпечення (як швидкодії процесору так і кількості ядер, що відображає кількість потоків що можуть виконувати інструкції в єдиний момент часу) та інших важливих компонентів, при цьому маючи на увазі що потоки не стали легшими для операційних систем, призвів до появлення наступного етапу у розвитку серверних архітектур, а саме до використання пулів потоків для обробки клієнтських запитів (рис. 3.2).

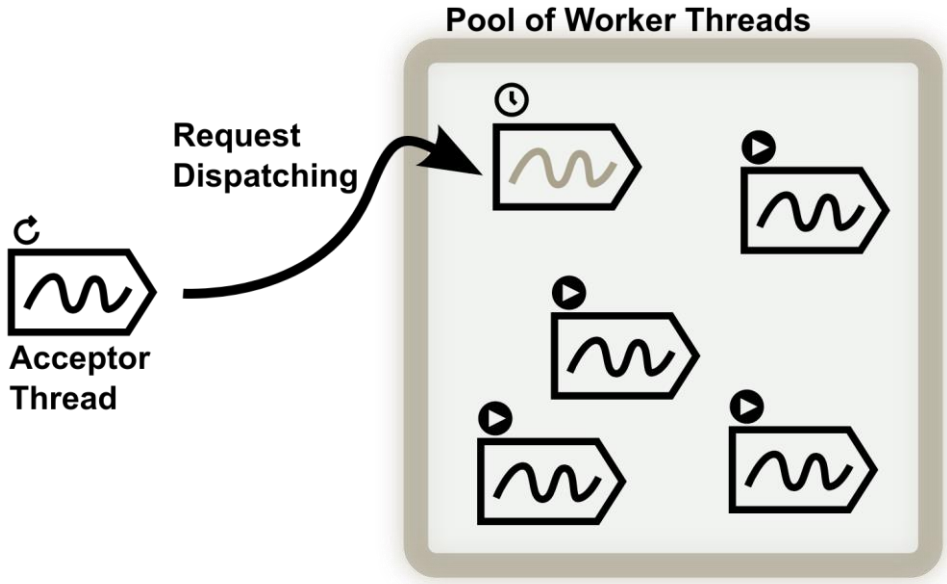


Рисунок 3.2 – Модель обробки клієнтських запитів, побудована на основі пулу потоків

Цей метод передбачає, що замість того, щоб створювати та звільняти потоки виконання, додаток на момент свого розгортання буде створювати один або декілька пулів потоків, які можуть простоювати, але при отриманні декількох одночасних запитів вони будуть розподіляти всю доступну роботу між потоками у цьому пулі [9]. Модель пула потоків представлена на рисунку 3.3. Модель реалізації пула потоків на мові програмування Java представлена на рисунку 3.4.

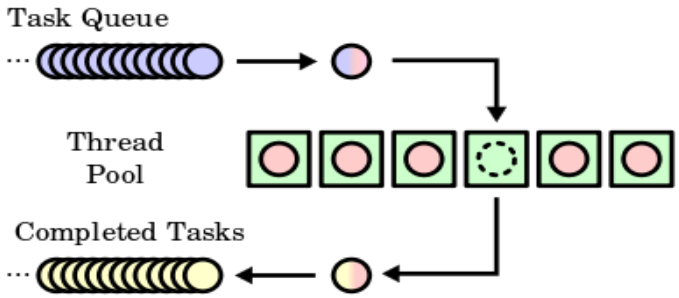


Рисунок 3.3 – Модель пулу потоків

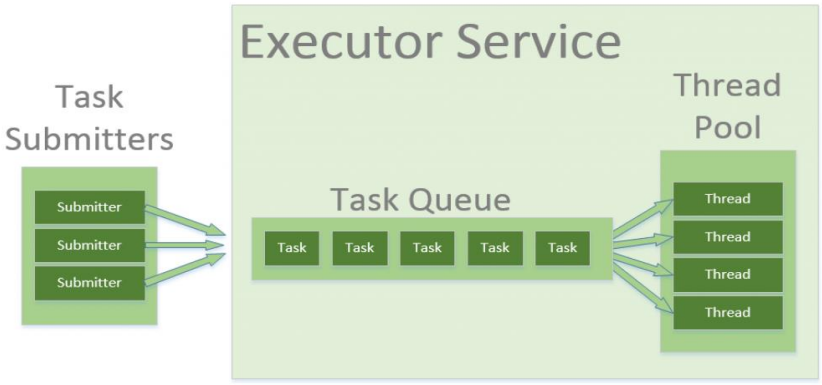


Рисунок 3.4 – Модель реалізації пула потоків в Java

Найбільшою перевагою цього рішення, особливо при порівнянні із попереднім рішенням, є мінімум накладних витрат на створення потоків та їх звільнення. Це дає величезну перевагу з точки зору оптимізації системи як у контексті швидкодії, так і у контексті витрат на пам’ять. Також важливо підкреслити, що проектування, розробка та підтримка системи з пулом потоків не є набагато складнішою задачею ніж проектування системи що підкоряється моделі “один потік на запит”.

Проте кількість водночас отриманих запитів, що мають бути опрацьовані якомога швидше, може набагато перевищувати кількість доступних у пулах потоків, або, навпаки, мати набагато меншу кількість. Цю проблему можна вирішити завдяки динамічній конфігурації пулів потоків: вони можуть мати деяку статичну кількість потоків, що мають бути активними незалежно від навантаження на пул, та наряду з тим динамічно розширюватися якщо в черзі завдань є завдання що не можуть бути оброблені в даний момент часу та звільнятися після того як робота була виконана.

Тому одним з недоліків такої архітектури є її дуже важка адаптивність та конфігурація задля того щоб система споживала найменшу кількість ресурсів для обробки потрібної кількості запитів (тобто була найбільш ефективною з точки зору як споживаних ресурсів, так і швидкодії). Також важливо підкреслити, що цей підхід не вирішує проблем з блокуючим доступом до системи вводу-виводу, що є найбільш “вузьким місцем” у роботі таких систем.

Важливо також, що після того, як у системі з’явилося забагато робіт для виконання та вичерпано пул непрацюючих потоків, нові роботи (наприклад, вхідні веб-з'єднання) повинні чекати перш ніж вони будуть оброблені. У крайньому випадку нові з'єднання можуть просто вичерпатися, а клієнт взагалі може не отримати з’єднання з сервером. Іншим прикладом є програми GUI, де потік інтерфейсу може потенційно замикатися і значно знижує чутливість програми.

Хоча створення масштабованого пулу потоків може здатися відносно прийнятним варіантом, з часом з’явиться проблема виснаження ресурсів. Якщо потік

блокується в очікуванні на якусь подію яка не настає довгий час, то це стає прикладом безглуздого вилучення ресурсів операційної системи [9]. Хоча це зручний спосіб проектування системи, витрачені цикли процесора та час потоку ОС ніколи не є ефективним способом досягнення масштабованості. Також варто згадати про вартість потоків ОС у контексті клієнтських систем: кореляція між швидкістю активації та дезактивації потоку в залежності від даних що той зберігає та кількості активних потоків у системі представлено на рисунку 3.5.

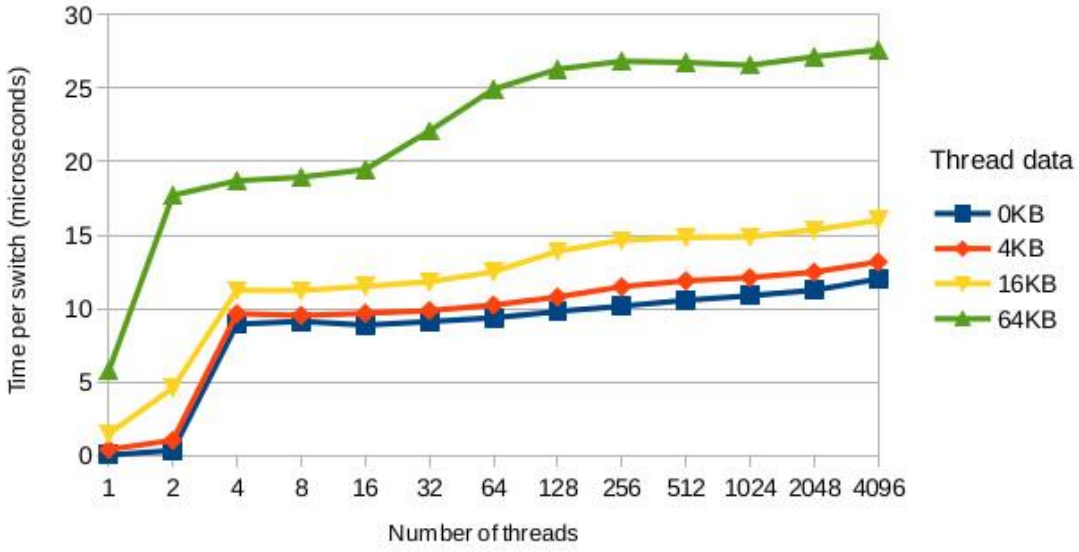


Рисунок 3.5 – Демонстрація кореляції між кількістю потоків у системі, часом на зміну активного потоку та даними, що зберігає кожен потік

Застосування пулу потоків є гарною альтернативою типовій “один потік на з’єднання” моделі, але вона все ж ділить з нею багато недоліків які є критичними з точки зору роботи системи.

## 3.4 Аналіз асинхронної моделі виконання

Переважна більшість кодової бази у світі є синхронною. Це означає що виконання програмного коду (може з циклами, розгалуженнями, зупинками, і відновленнями) з урахуванням тих же параметрів виконується у детермінованому порядку. Синхронний код в теорії є набагато більш простим, оскільки можна слідувати прямій послідовності виконання самого програмного коду. Проблема полягає в тому, що програма, яка виконується синхронно, може давати результати занадто повільно або можливо виконувати тривалі операції, які залишають програму невідповідною для подальшого введення (наприклад, вона сервер може просто перестати приймати запити, тому що всі потоки зайняті). Тому системи, що ставлять перед собою метою виконувати декілька задач уникаючи блокування, зазвичай використовують асинхронну модель програмування [8].

Коли блокування потоків є проблемою для поки що найбільш відповідного рішення з пулом потоків, асинхронна модель виконання може вирішити цю проблему у багатьох випадках. У багатьох додатках, які використовують пули потоків, потоки насправді не роблять багато, але чекають чогось іншого – відповіді від бази даних, іншого веб-сервісу або певного вводу користувача (рис. 3.6). Якщо замість потоку просто очікуючи на відповідь можна було налаштували якусь систему, щоб викликати якийсь код, коли він доступний, можна було б повернути потік до пулу, щоб виконати якусь іншу корисну роботу (рис. 3.7). Таким чином, додаток мав би можливість виконувати більше робіт ніж кількість потоків які є доступними у системі.

Асинхронність у комп'ютерному програмуванні характеризується виникненням подій, незалежних від основного потоку програми та способів роботи з такими подіями [9].

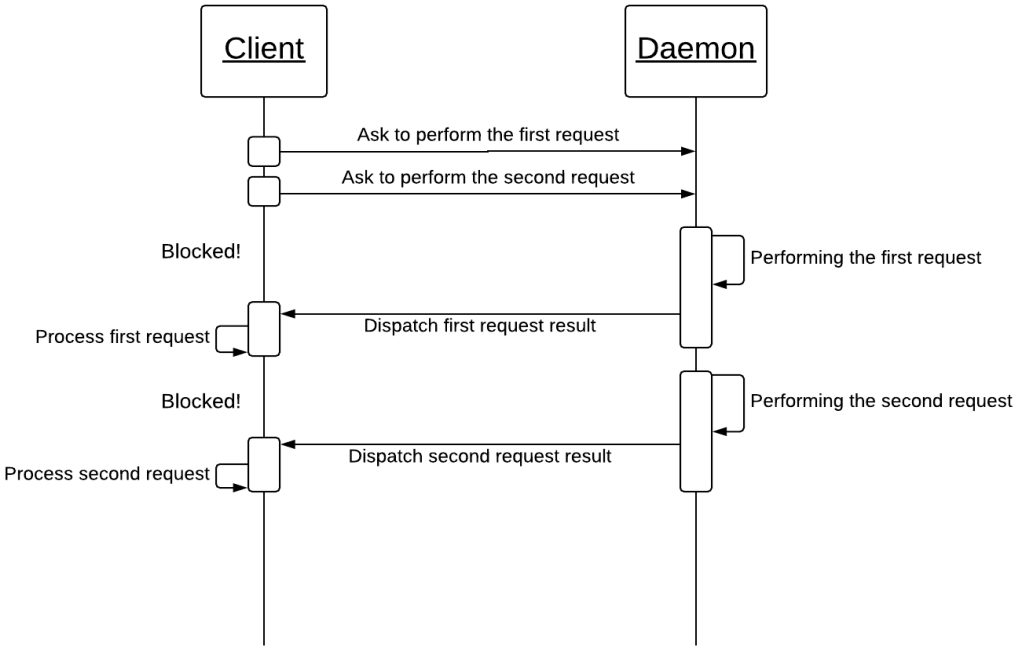


Рисунок 3.6 – Графічне відображення синхронної моделі виконання

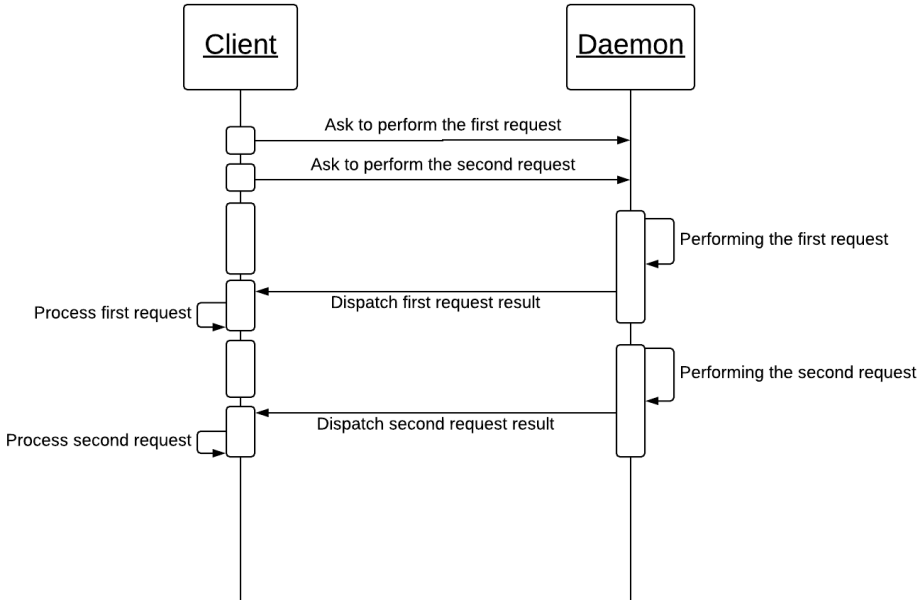


Рисунок 3.7 – Графічне відображення асинхронної моделі виконання

Найбільш характерною реалізацією асинхронної моделі навіть у мовах програмування з явною синхронною моделлю є цикл обробки подій (або просто “цикл подій”).

Цикл подій – це програмна конструкція (об’єкт), що чекає і розсилає події або повідомлення в програмі. Цикл подій працює, надсилаючи запит до якогось внутрішнього або зовнішнього "постачальника подій" (який, як правило, блокує запит, поки подія не надійшла), а потім викликає відповідного обробника подій ("розсилає подію"). Коли цикл подій утворює структуру центрального потоку управління програми, як це часто робиться, його можна назвати основним циклом або циклом основної події. Цей заголовок є відповідним, оскільки такий цикл подій знаходиться на найвищому рівні контролю в межах програми [1]. Реалізація циклу подій представлена на рисунку 3.8.

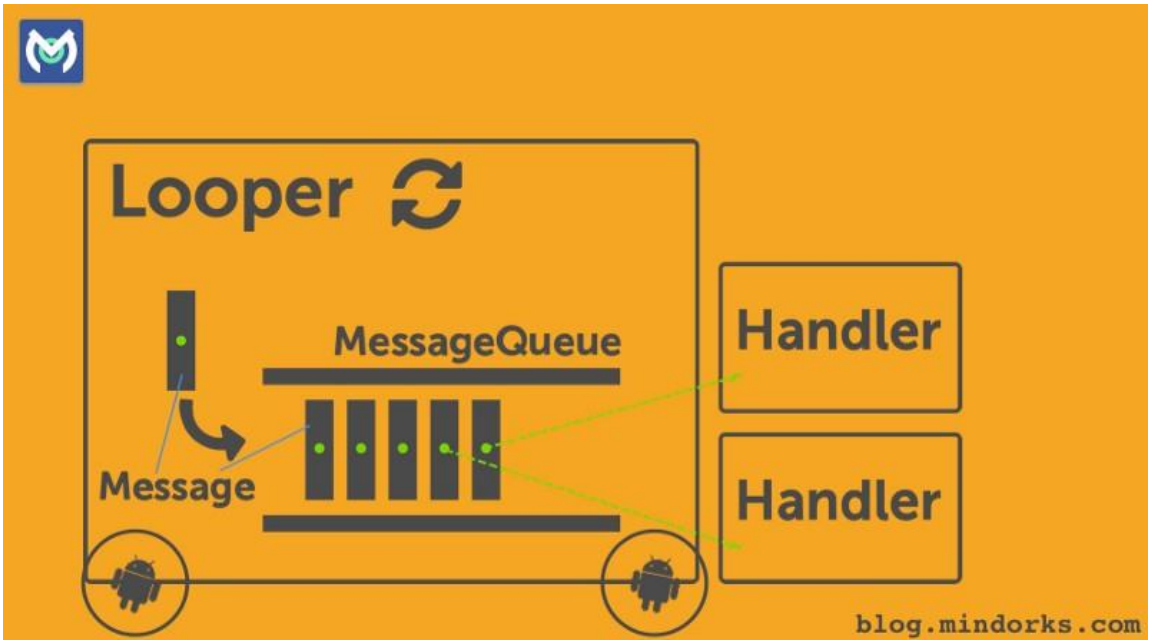


Рисунок 3.8 – Реалізація циклу подій у платформі Android

Висновком є той факт, що асинхронна модель виконання є ефективним доповненням до будь-якої моделі базованої на потоках виконання. Її основною проблемою є складність розробки та підтримки програм, що базовані на асинхронній моделі.

## 3.5 Аналіз методу реактивного програмування

Реактивне програмування є відповідним інструментом для вирішення прикладних в таких сценаріях:

- реагування та обробка будь-яких подій, пов'язаних із затримкою вводу-виводу, з диска або мережи, враховуючи, що ввід-вивід є асинхронним (запит робиться, час проходить, відповідь може бути або не бути отриманою, на що потім спрацьовують зареєстровані обробники);
- поводження з подіями або даними, поданими на додаток від генератора, який він не може контролювати (системні події з сервера, події користувача) [10].

Реактивне програмування також є дуже ефективною програмною моделлю при великих навантаженнях на систему вводу-виводу.

В якості прикладу порівняння традиційної та реактивної реалізацій компоненту що потрапляє під високу завантаженість взято такі реалізації контейнера сервлетів для JavaEE як Tomcat (традиційний) та Netty (реактивний). У якості джерела бенчмарків виступає документ “RxNetty vs Tomcat Performance Results” [11].

На рисунку 3.9 представлені результати споживання ресурсів процесору на обробку одного запиту. Результат показав, що Netty використовує на 40 відсотків менше ресурсів ніш Tomcat. Це обумовлено тим, що Netty використовує меншу кількість потоків виконання, а тому не навантажує процесор та процесорні кеши при

збереженні та відновленні стану потоків виконання при зміні контексту (context switch).

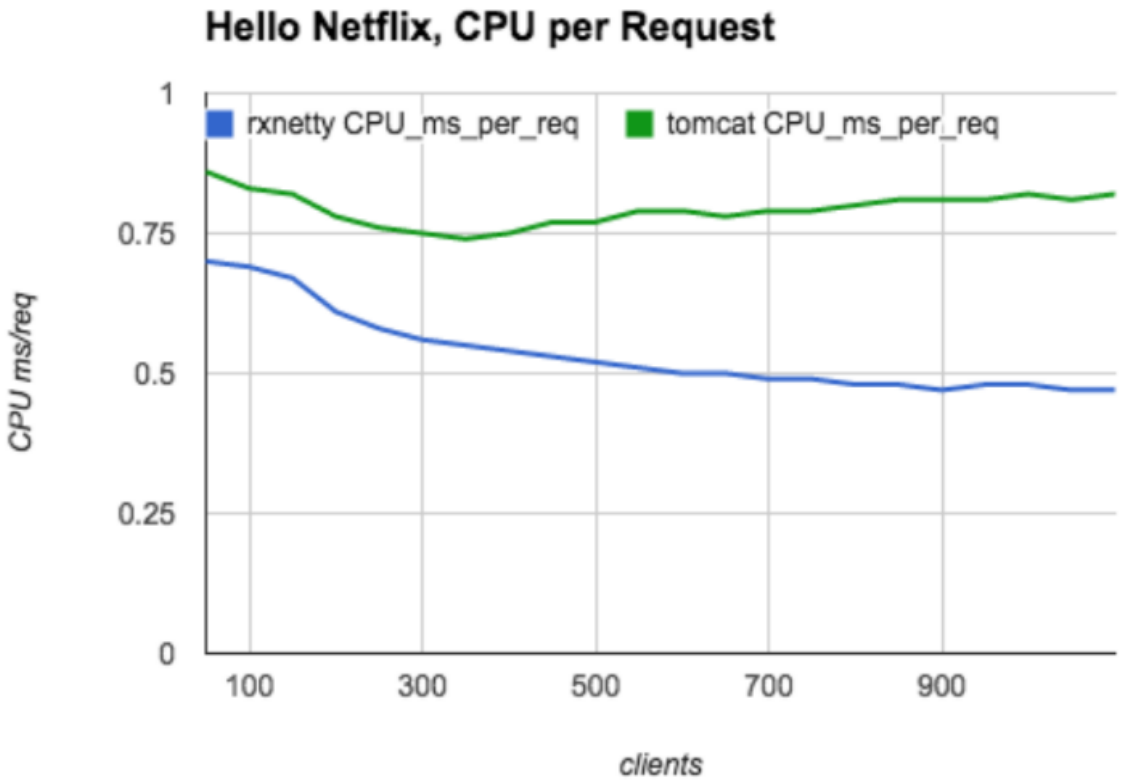


Рисунок 3.9 – Порівняння ресурсів CPU для обробки запиту Tomcat (зелений) та Netty (синій), що використовуються

Стосовно темпу прийому запитів, Netty показав на 46% кращий результат ніж Tomcat (рис 3.10). Це обумовлено теж здебільшого навантаженням процесору.

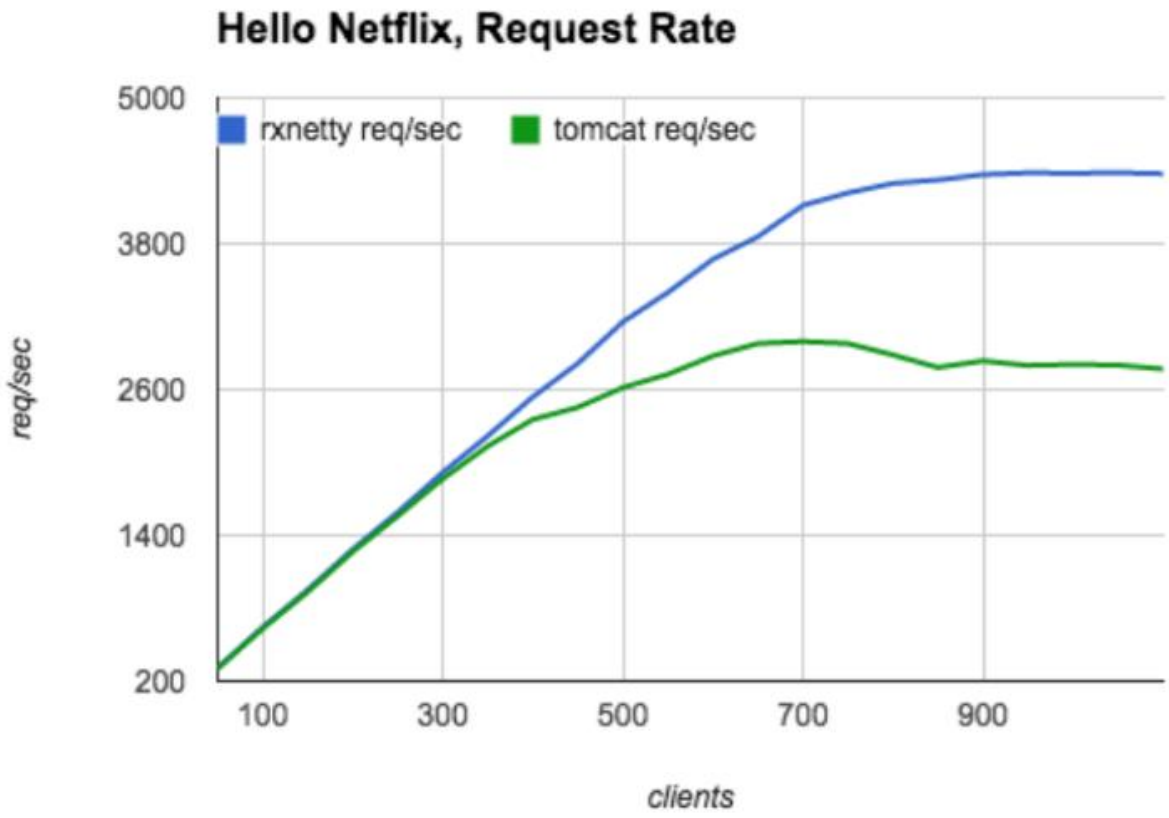


Рисунок 3.10 – Порівняння темпу прийому запитів в Tomcat та Netty

Важливою характеристикою також є максимальна затримка обробки запиту. Тут Netty (650 мс) дав приблизно у три рази кращий результат ніж Tomcat (2200 мс) (рис. 3.11).

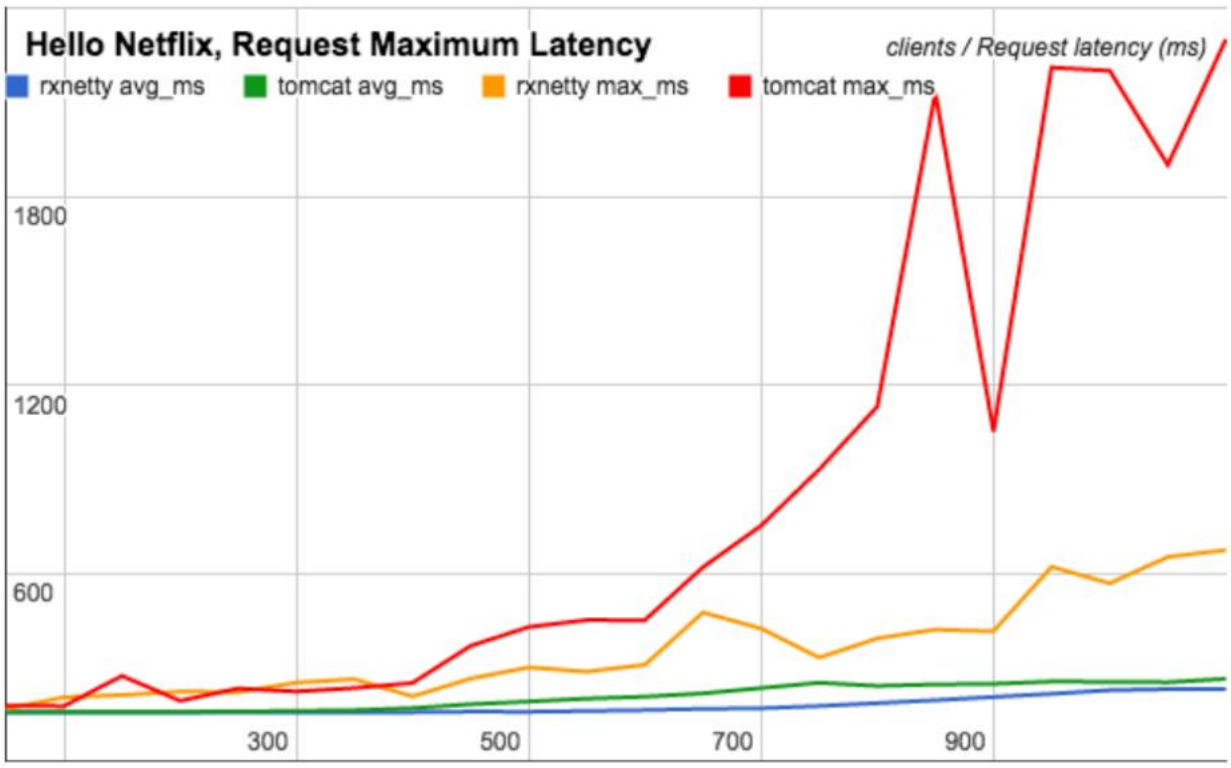


Рисунок 3.11 – Порівняльна характеристика максимальної затримки обробки запитів у Tomcat та Netty

Ці бенчмарки показують що реактивне програмування є набагато більш ефективним методом при вирішенні задач у контексті високої завантаженості на систему вводу-виводу.

Реактивне програмування — це парадигма програмування, побудована на потоках даних і розповсюдженні змін. Це означає, що у мовах програмування має бути можливість легко виразити статичні чи динамічні потоки даних, а реалізована модель виконання буде автоматично розсилати зміни через потік даних.

Наприклад, в імперативному програмуванні вираз “a = b + c” означає, що “a” отримує результат виконання “b + c” безпосередньо під час обчислення виразу, і потім, якщо значення “b” і “c” зміняться, це не впливатиме на вже обчислене значення “a”.

Проте, в реактивному програмуванні значення “a” буде автоматично оновлено за новими значеннями, що є протилежністю функціонального програмування.

Сучасна програма електронної таблиці є прикладом реактивного програмування. Комірки електронної таблиці можуть мати буквальні значення, або формули типу «=B1+C1», що обчислюються за значеннями інших комірок. Коли б не змінилося значення іншої комірки, значення формули оновлюється автоматично.

Інший приклад — це мова опису апаратури типу Verilog. У цьому випадку, реактивне програмування дозволяє моделювати зміни по ходу їх розповсюдження по електричному ланцюгу.

Реактивне програмування спочатку пропонувалося як засіб простого створення інтерфейсів користувача, анімацій у системах реального часу, але стало загальною парадигмою програмування.

Розглянемо тенденцію розвитку типового веб-проекту, що реалізує типову блокуючу модель (рис. 3.12). Приблизно 40% проектних ресурсів надходить на компенсацію розробки програмного забезпечення, 50% надходить на придбання та підтримку апаратного забезпечення, та останні 10 – на підтримку програмного забезпечення.

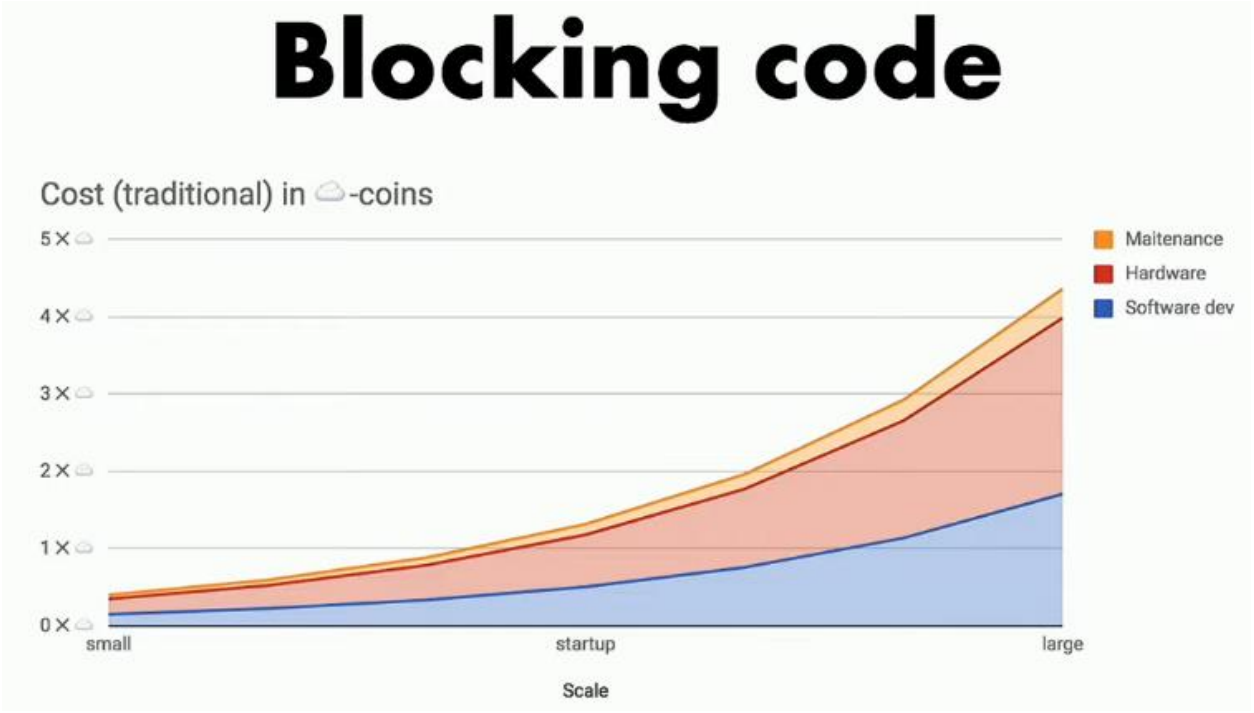


Рисунок 3.12 – Відносна характеристика коштів, що витрачаються, на реалізацію та підтримку проекту з блокуючою моделлю

Для проекту, реалізованого за реактивним підходом, розподіл коштів та їх варіювання в залежності від розміру самого проекту дещо інакші (рис 3.13). Витрати на проект невеликого масштабу є набагато більшими з точки зору компенсації розробки програмного забезпечення; те ж саме стосується також проектів середніх розмірів. У випадку великих проектів, навпаки, остаточна сума становить майже у 2 рази меншу величину ніж для першого, блокуючого варіанту. Реактивна модель набагато складніша для розуміння та підтримки, але у випадку великих високонавантаженого контексту (як пошукова системи) витрати на апаратну частину значно менші, що дозволяє зменшити кінцеву суму реалізації проекту майже у 2 рази.

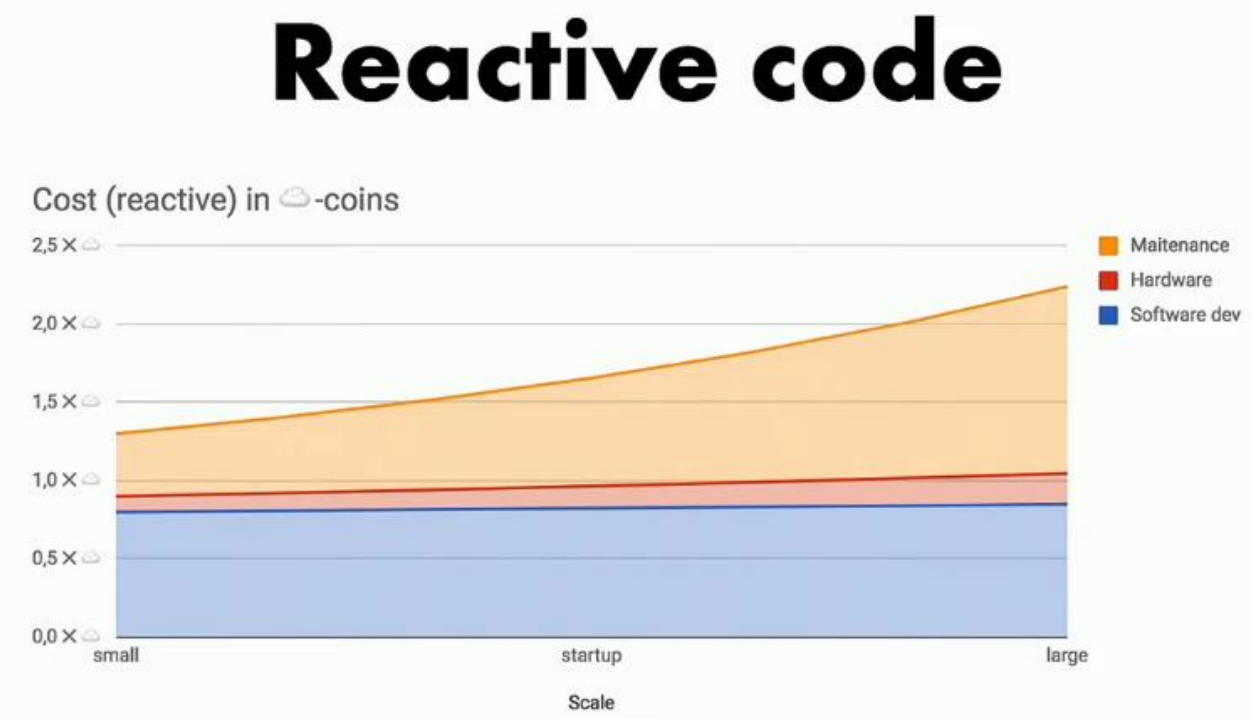


Рисунок 3.13 – Відносна характеристика коштів, що витрачаються, на реалізацію та підтримку проекту з реактивною моделлю

Тому важливо з’ясувати розмір проекту, який розробляється, щоб визначити, чи потрібно застосовувати метод реактивного програмування для вирішення проектних задач (рис. 3.14).

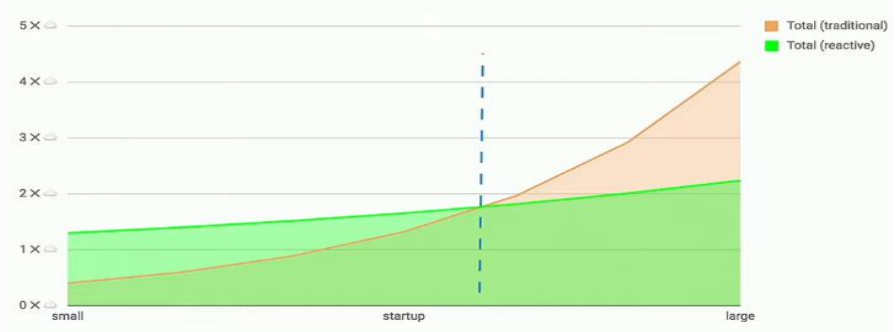


Рисунок 3.14 – Порівняльна характеристика бюджету проекту в залежності від розміру та вибраного методу

Реактивне програмування можна вважати реалізацією асинхронного програмування з тим винятком що програми, що реалізовані за допомогою реактивної моделі, набагато простіше розробляти, масштабувати, підтримувати та проектувати. Усі операції та роботи можна описувати у декларативному стилі, так само як і політики щодо розпаралелювання робіт. Тобто можна вважати, що реактивна модель повністю нейтралізує проблеми чистої асинхронної моделі, при тому повністю підтримуючи та реалізуючи її виняткові (у позитивному сенсі) характеристики. Варто зауважити, що реалізація реактивної моделі для Java (RxJava2) утилізує пули потоків при роботі за асинхронними подіями та їх обробкою.

Пошукова система є прикладом високонавантаженого та об’ємного проекту, тому можна встановити, що застосування складних методів паралельного та конкурентного програмування є вдалим рішенням для реалізації проекту.

# 4 АНАЛІЗ ТА ВИБІР ПРОГРАМНОГО СЕРЕДОВИЩА ВИКОНАННЯ

Програмна середовище виконання є одним з найважливіших факторів при проектуванні та реалізації додатку. У контексті розробки високонавантажених систем найбільш важливими факторами є:

- ефективна підтримка паралелізму на рівні потоків виконання та \ або асинхронної моделі програмування;
- навантаження на пам’ять;
- швидкість роботи.

Найбільш популярними середовищами виконання із відносно прийнятними навантаженням на пам’ять та швидкість роботи, але із визначним з точки зору ефективності рівнем підтримки паралельного програмування є CLR (Common Language Runtime) та JVM (Java Virtual Machine).

## 4.1 Огляд та аналіз CLR

.NET Framework надає програмне середовище виконання, яке називається CLR (Common Language Runtime), яке виконує програмний код і надає інструменти, що полегшують процес розробки.

Компілятори та інструменти розкривають функціональні можливості CLR та дозволяють писати код корисний для цього керованого середовища виконання. Код, який ви розробляєте за допомогою компілятора мови, який орієнтується на конкретну програмне середовище, називається керованим кодом; йому вигідні такі функції, як інтеграція між мовами, обробка виключень між мовами, покращена безпека, підтримка версій та розгортання, спрощена модель взаємодії компонентів та послуги налагодження та профілювання.

Для надання можливості середовищу виконання надавати корисні можливості керованому коду, компілятори мови повинні надавати метадані, що описують типи, члени та посилання у вашому коді. Метадані зберігаються разом з кодом; кожен портативний виконуваний файл (PE) CLR, що завантажується, містить метадані. CLR використовує метадані для пошуку та завантаження класів, викладення екземплярів у пам'яті, вирішення викликів методів, генерування власного коду, забезпечення безпеки та встановлення меж контексту виконання.

CLR автоматично обробляє макет об'єкта та керує посиланнями на об'єкти, випускаючи їх, коли вони більше не використовуються (garbage collection). Об'єкти, життя яких управляється таким чином, називаються керованими даними. Збір сміття усуває витоки пам’яті (memory leaks), а також деякі інші поширені помилки програмування.

CLR дозволяє легко розробляти компоненти та програми, об’єкти яких взаємодіють між сумісними мовами. Ця мовна інтеграція можлива тому, що компілятори мов та інструменти, націлені на середовище виконання, використовують загальну систему типів, визначену для цього середовища, і вони дотримуються правил виконання для визначення нових типів, а також для створення, використання, зберігання та прив'язки до типів.

Як частина їх метаданих, усі керовані компоненти несуть інформацію про компоненти та ресурси, проти яких вони були побудовані (залежності). У процесі виконання ця інформація використовується для того, щоб компонент або додаток мали вказані версії всього необхідного, що робить код менш ймовірним пошкодити виконання через деяку невдалу (незадоволену) залежність. Інформація про реєстрацію та дані про стан більше не зберігаються в реєстрі, де їх складно встановити та підтримувати. Натомість інформація про визначені вами типи (та їх залежності) зберігається з кодом як метадані, що робить завдання реплікації та видалення компонентів набагато менш складними.

CLR забезпечує наступні переваги:

- поліпшення продуктивності;
- можливість легко використовувати компоненти, розроблені іншими мовами;
- типи розширень, що надаються бібліотекою класів;
- мовні функції, такі як успадкування, інтерфейси та перевантаження для об'єктно-орієнтованого програмування;
- підтримка явних безкоштовних потоків, що дозволяє створювати багатопоточні, масштабовані програми;
- підтримка структурованої обробки виключень;
- підтримка спеціальних атрибутів;
- збір сміття (GC).

## 4.2 Огляд та аналіз JVM

Віртуальна машина Java є наріжним каменем мови програмування Java. Вона є компонентом технології Java, яка відповідає за кросплатформенне виконання Java (або іншої JVM-сумісної мови) коду, невеликий розмір її скомпільованого коду та можливість Java захищати користувачів від шкідливих програм. Віртуальна машина Java – це абстрактна обчислювальна машина. Як і справжня обчислювальна машина, вона має набір інструкцій і використовує різні області пам'яті. Досить поширеною є реалізація мови програмування за допомогою віртуальної машини [12].

JVM спочатку була тісно пов’язана з мовою програмування Java, де Java грала роль високорівневої мови програмування яка за допомогою своїх конструкцій надавала розробнику змогу використовувати ряд можливостей які підтримує сама JVM.

JVM є специфікацією абстрактної машини, що має безліч реалізацій, тому спочатку важливо сконцентруватися на інтерфейсі машини.

JVM гарантує інтерфейс та надає можливості, дуже схожі з CLR, найважливішими з яких для проблеми на даний момент є ефективна підтримка паралелізму (аж до низькорівневих конструкцій які є часто необхідними у багатопоточному середовищі що дозволяють напряму використовувати методи інтерфейсу процесорів, такі, наприклад, як compareAndSet).

Важливим фактором з точки зору JVM є те що JVM як середовище виконання легко конфігурується, тому якщо якась реалізація віртуальної машини може не підходити для реалізації задачі, завжди можна знайти іншу, яка надає потрібні можливості чи має більш оптимізовані характеристики де оптимізація має місце у контексті якоїсь визначеної характеристики (наприклад, час виконання або навіть швидкість зміни контексту активного потоку).

## 4.3 Порівняння CLR та JVM та обґрунтування подальшого вибору

CLR та JVM – це віртуальні машини, і вони обидва виконують незалежні від платформи операції, а саме – виконують компільований вихідний код. По суті, ці віртуальні машини виступають як посередницький крок між вихідним кодом розробника та машинним кодом системи, що забезпечує більшу зручність використання різними типами процесорів. На зображенні нижче ви можете побачити, як функції високого рівня у двох практично однакові (рис. 4.1).

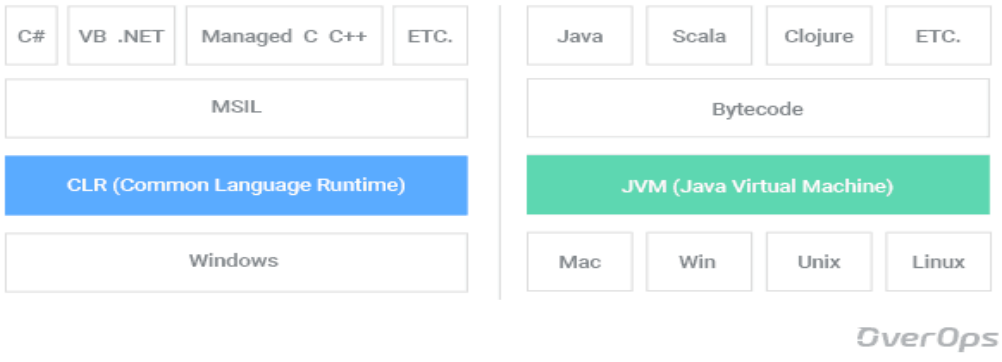

Рисунок 4.1 – Порівняння високорівневих архітектур CLR та JVM

Крім основної функціональності, обидва також включають методи збору сміття, безпеку на рівні виконання та поводження з винятками. І остання подібність, яку слід зазначити, обидва використовують операції на основі стека, найпоширеніший підхід до зберігання та отримання операндів та їх результатів.

Але, звичайно, для кожної подібності, яку поділяють ці віртуальні машини, можна знайти різницю у впровадженні.

Одна з потенційно фундаментальних різниць між CLR та JVM (хоча це, ймовірно, вплинуло на спосіб їх розвитку), полягає в тому, що JVM була створена для роботи спеціально з Java, тоді як CLR був розроблений так, щоб бути нейтральним до мови. І навпаки, CLR спочатку був розроблений лише для роботи в ОС Windows, тоді як JVM завжди був нейтральним до ОС. Зараз же усе змінилося: існує CoreCLR, який працює на Linux та Mac, і багато інших мов було розроблено для роботи з JVM (Groovy, Scala, Kotlin, JRuby, Clojure, та ін.). Це призводить до того, що здебільшого відмінності між CLR та JVM є також ознаками відмінностей між мовами, які їх використовують. Можна сказати, що деякі найзначніші відмінності між мовами (для аргументів, скажімо, C # та Java) реально реалізовані на рівні VM. Як висновок – спочатку обидві віртуальні машини мали свої серйозні обмеження стосовно мов або операційних систем що підтримуються, але згодом вони були усунені і з точки зору цих аспектів обидва рішення можна вважати рівноважними.

Інша велика різниця, яку є явною на рівні VM, полягає в тому, що хоча обидва використовують компіляцію JIT (Just-in-Time), компілятор не закликається запускатись одночасно. CLR компілює весь код MSIL в машинний код, коли він викликається під час виконання. JVM використовує спеціалізований механізм продуктивності під назвою HotSpot, щоб зробити JIT компіляцію Java-байт-коду в машиночитаний код. Він відрізняється тим, що він компілює та оптимізує «гарячі точки» в коді, які використовуються найчастіше. Кожна з цих стратегій компіляції має

свої компроміси щодо ефективності. Оскільки CLR одноразово збирає весь машинний код, коли він викликається під час виконання, час виконання може покращитися в деяких умовах. Але, з іншого боку, якщо невелика частина коду буде потрібна на високій частоті, компілятор HotSpot Java може з часом підвищити ефективність за допомогою додаткової оптимізації. Варто відзначити, що деякі інші реалізації JVM підтримують AOT (компіляцію заздалегідь), якщо це є більш прийнятно у визначених умовах. Як висновок – JVM надає більше можливостей щодо оптимізацій ніж CLR.

Дослідження, проведене 2003 р. Джеремі Зінгер в Кембриджському університеті, розглядало відмінності в роботі між CLR та JVM і виявило, що, принаймні, на той час, їх продуктивність була практично однаковою. Дослідження виявило, що компілятор Java виробляє менший байт-код, що може впливати на продуктивність VM. Однак, коли справа доходила до запущеного коду, час виконання вийшов майже 50/50, при цьому CLR та JVM виконувались швидше приблизно в половину часу. Виходячи з цього дослідження, якщо мова йде про жорстоке протистояння між Java та користувачами .NET, схоже, що самі віртуальні машини відіграють більшу роль у розмежуванні можливостей, ніж вони безпосередньо впливають на ефективність додатків. Це також має сенс під час перегляду використання тієї чи іншої віртуальної машини. У корпоративних компаніях розробники працюють не з рідними мовами на основі CLR та JVM. Це може статися особливо у великих компаніях з багатьма різними відділами, які працюють над різними проектами та додатками.

Основні подібності між CLR та JVM включають:

- обидва віртуальних машини (віртуальних машин);
- в обох передбачено збирання сміття;
- обидва використовують операції на основі стека;
- обидва включають безпеку на рівні виконання;
- обидва мають методи обробки винятків;

- обидві є агностичними щодо хостової ОС;
- обидві є агностичними щодо мови програмування.

Основні відмінності між CLR та JVM включають:

- CLR використовує компілятор JIT, JVM використовує спеціалізований компілятор JIT під назвою Java HotSpot; інші реалізації JVM включають багато інших можливостей щодо компіляції, аж до AOT;
- CLR включає в себе інструкції щодо корутин, замикань та декларації / маніпулювання вказівниками, коли JVM – ні;
- JVM є сумісною з більш надійними засобами вирішення помилок та засобами моніторингу [13].

З цього можна постановити, що хоча відмінності між цими віртуальними машинами є мінімальними, JVM, завдяки великій кількості існуючих реалізацій, має більший розмах щодо можливих оптимізацій.

## 4.4 Детальний огляд JVM та її компонентів з подальшим виявленням шляхів контекстної оптимізації

Віртуальна машина Java – це незалежна від платформи абстрактна машина, яка забезпечує середовище виконання, в якому виконується байт-код Java. Це частина середовища виконання Java, яка перетворює байт-код Java в машиночитану мову. Основний метод, який ми маємо в програмі Java, насправді викликається самою віртуальною машиною Java (рис. 4.2).

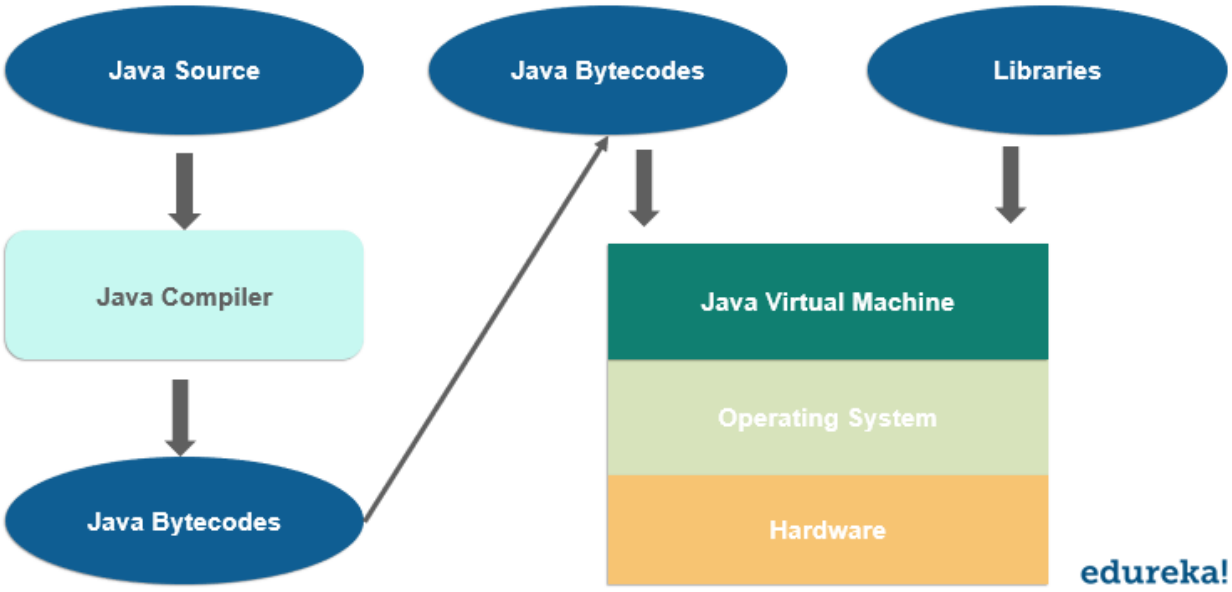


Рисунок 4.2 – Компіляція та інтерпретація коду в JVM

Віртуальна машина Java містить 3 основних компонента що впливають на процес виконання: підсистема завантажувача класів (ClassLoader), область даних процесу виконання (runtime data area) та движок виконання (рис. 4.3) [14].

Функціонал динамічного завантаження класів в Java керується підсистемою ClassLoader. Він завантажує, посилається. і ініціалізує файл класу, коли він вперше посилається на клас під час виконання, а не час компіляції.

На стадії завантаження класи будуть завантажені цим компонентом. BootStrap ClassLoader, Extension ClassLoader та Application ClassLoader – це три ClassLoader-а, які використовуються у цьому процесі. BootStrap ClassLoader відповідає за завантаження класів з системної директорії класів для завантажувача – нічого крім rt.jar. Найвищий пріоритет буде надано цьому навантажувачу. Extension ClassLoader відповідає за завантаження класів, які знаходяться всередині папки ext (jre \ lib). Application ClassLoader відповідає за завантаження classpath рівня програми. Вищенаведені ClassLoaders дотримуватимуться алгоритму ієрархії делегування під час завантаження файлів класу.

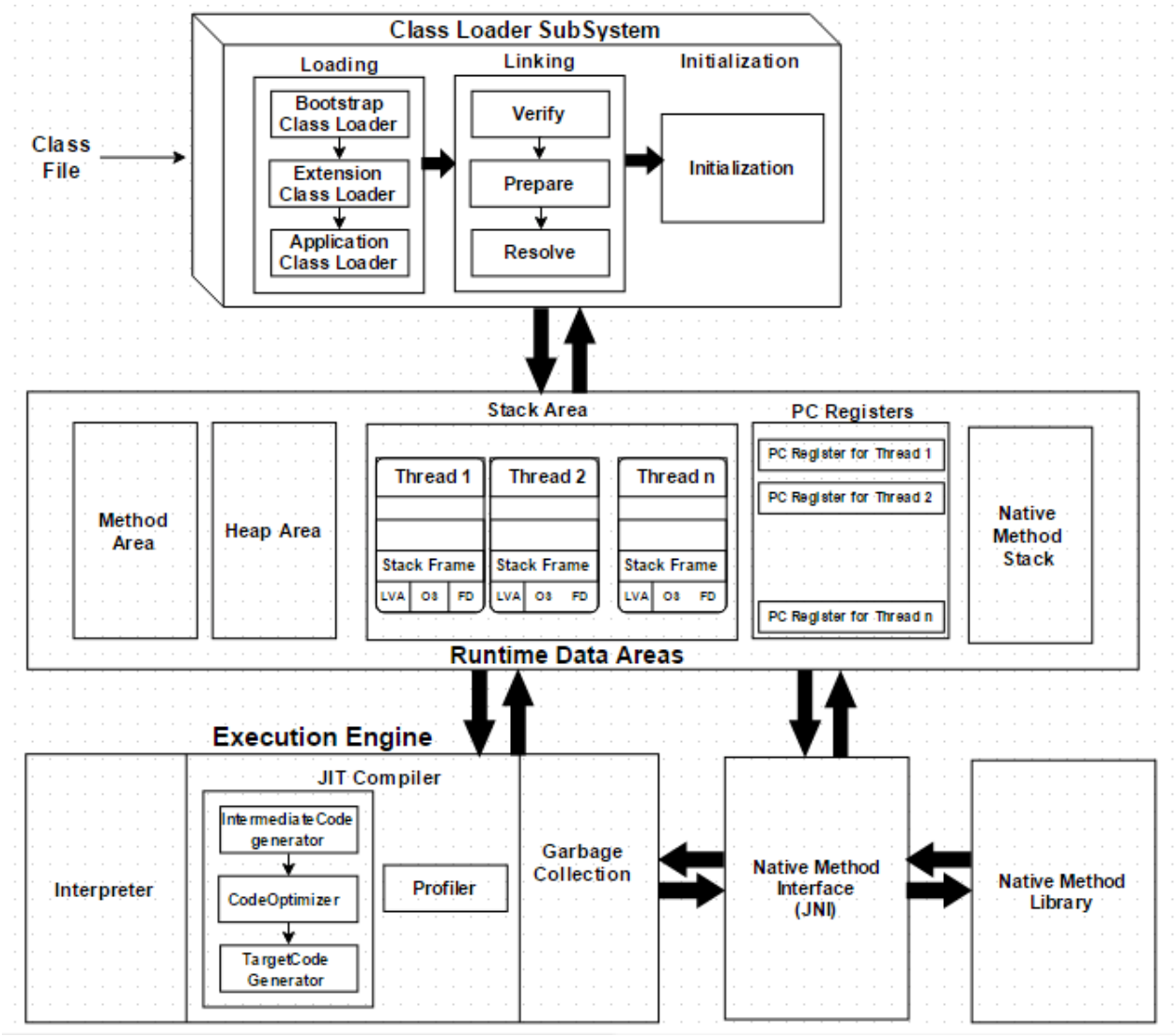


Рисунок 4.3 – Архітектура JVM

Під час зв’язування характерні 3 етапи: перевірка, підготовка та резолюція. Під час першої стадії перевірявач байт-коду перевірить правильність генерованого байтового коду, якщо перевірка не вдасться, то буде викинута помилка перевірки (VerifyError). Під час підготовки для всіх статичних змінних пам'ять буде розподілена і присвоєна значенням за замовчуванням. Під час резолюції усі символьні посилання на пам'ять замінюються оригінальними посиланнями з області методу.

Заключною фазою завантаження класів є ініціалізація. На цьому етапі всі статичні змінні будуть присвоєні вихідним значенням, і статичний блок кожного завантаженого класу буде виконаний.

Зона даних під час виконання розділена на п'ять основних компонентів: область методу, область купи, область стеку (підрозділяється на 3 підобласті), регістри комп’ютеру та стеки нативних методів.

В області методів зберігатимуться всі дані на рівні класу, включаючи статичні змінні. На JVM існує лише одна область методів, і це спільний ресурс.

В зоні купи зберігатимуться всі об’єкти та відповідні змінні екземпляри та масиви. Тільки один екземпляр купи ініціалізується у JVM. Оскільки області Method і Heap діляться пам’яттю на кілька потоків, збережені дані не є безпечними для потоків.

Область стека – для кожного потоку буде створено окремий стек виконання. Для кожного виклику методу буде зроблено один запис у пам'яті стека, який називається Stack Frame. Усі локальні змінні будуть створені в пам'яті стека. Область стека є безпечною для потоків, оскільки вона не є спільним ресурсом. Рамка стека поділена на три підрозділи: локальний змінний масив (пов'язаний з методом, скільки локальних змінних задіяно і відповідні значення будуть збережені тут), стек операндів (якщо для виконання потрібна якась проміжна операція, стек операнда виконує функцію робочої області виконання для виконання операції), та дані кадру (тут зберігаються всі символи, що відповідають методу; у випадку будь-якого винятку інформація про блок вловлювання буде зберігатися в даних кадру).

Регістри – кожен потік матиме окремі регістри, щоб містити адресу поточної інструкції виконання після виконання інструкції, регістр ПК буде оновлений наступною інструкцією.

Native Method stacks – Native Method Stack вміщує інформацію про власний метод. Для кожного потоку буде створений окремий стек методів.

Байтовий код, який присвоєний області даних під час виконання, буде виконуватися движком виконання. Движок виконання зчитує байт-код і виконує його по частинах.

Інтерпретатор інтерпретує байт-код швидше, але виконується повільно. Недоліком є те, що коли один метод викликається багаторазово, кожен раз потрібно нове “тлумачення”.

Компілятор JIT нейтралізує недоліки перекладача. Виконання движку використовує допомогу інтерпретатора при перетворенні байтового коду, але коли він знаходить повторний код, він використовує компілятор JIT, який збирає весь байт-код і змінює його на нативний код. Цей нативний код буде використовуватися безпосередньо для повторних викликів методів, які покращують продуктивність системи.

Генератор проміжного коду виробляє проміжний код.

Оптимізатор коду – відповідає за оптимізацію проміжного коду, згенерованого вище.

Генератор цільового коду – відповідальний за генерацію машинного коду або власного коду

Профілер – спеціальний компонент, який відповідає за пошук гарячих точок, тобто метод називається кілька разів чи ні.

Сміттєзбірник: збирає та видаляє непов'язані об’єкти. Збір сміття може бути запущений за допомогою виклику System.gc (), але виконання не гарантується. Збір сміття CBM збирає об’єкти, які створюються.

Компонентами JVM, що легко змінити, є сама реалізація JVM, що реалізує усі компоненти окрім компоненту керування пам’яті, та сам збірник мусору (Garbage Collector), що є достатньо сильно пов’язаним з реалізацією віртуальної машини (тому що алокація об’єктів є специфічним процесом відносно реалізації віртуальної машини), але майже всі активні віртуальні машини можуть підтримувати декілька реалізацій збірника мусору.

Сміттєзбірники в JVM відрізняються один від одного за декількома факторами та потребують більш детального аналізу.

Сміттєзбірник працює в два етапи, відомі як “позначення” та “вбирання”. Фаза позначення є тою фазою, де збирач сміття визначає, які фрагменти пам'яті використовуються, а які – ні. На фазі вбирання сміттєзбірник видаляє об'єкти, ідентифіковані під час фази «позначення»[15].

Переваги:

- Немає вручну розподілу пам’яті / обробки місця розташування, оскільки невикористаний простір пам'яті автоматично обробляється GC.
- Немає накладних витрат на обробку вказівника.
- Автоматичне управління витоком пам’яті (GC самостійно не може гарантувати повноцінне рішення протікання пам’яті, однак, вона дбає про хорошу її частину).

Недоліки:

- Оскільки JVM повинна відслідковувати створення / видалення посилань на об'єкт, то для цього потрібна додаткова потужність процесора, що має місце бути крім оригінальної програми. Це може вплинути на виконання запитів, для яких потрібно багато пам'яті
- Програмісти не можуть контролювати планування часу процесора, присвяченого звільненню об'єктів, які більше не потрібні
- Використання деяких реалізацій GC може призвести до непередбачуваної зупинки програми
- Автоматизоване управління пам’яттю не буде настільки ефективним, як правильне ручне розподіл пам’яті / розсилка

JVM має чотири типи реалізації GC:

- Серійний
- Паралельний

- CMS
- G1

Серійний сміттєзбірник є найпростішою реалізацією, оскільки вона в основному працює з одним потоком виконання. В результаті ця реалізація GC заморожує всі потоки додатків під час її запуску. Отже, не дуже добре використовувати його у багатопотокових програмах, таких як серверне середовище. Це сміттєзбірник для більшості застосунків, які не мають невеликих вимог до паузи та працюють на машинах клієнтського стилю.

Паралельний сміттєзбірник йде за замовчуванням JVM. На відміну від послідовного збирача сміття, він використовує кілька потоків для управління простором купи. Але він також заморожує інші потоки додатків під час виконання GC. Розробник може контролювати максимальну кількість потоків збору сміття та призупинити час, пропускну здатність та слід (розмір купи).

Реалізація CMS використовує кілька потоків для збирання сміття. Він призначений для додатків, які віддають перевагу більш коротким паузам збору сміття, і які можуть дозволити ділитися ресурсами процесора зі збирачем сміття під час роботи програми. Простіше кажучи, програми, що використовують цей тип GC, в середньому реагують повільніше, але не перестають реагувати на збирання сміття. Швидкий момент, який слід зазначити тут, є те, що оскільки цей GC є одночасним, виклик явного збору сміття, наприклад використання System.gc(), коли паралельний процес працює, призведе до відмови / переривання паралельного режиму. Якщо більше 98% від загального часу витрачається на збирання сміття CMS, а менше 2% відбирають купу, то колектор CMS викидає OutOfMemoryError.

G1 (Garbage First) призначений для додатків, що працюють на багатопроцесорних машинах з великим простором пам'яті. Він доступний з JDK7 Update 4 та з наступних версій. Колектор G1 замінив колектор CMS, оскільки він більш ефективний. На відміну від інших колекторів, колектор G1 розділяє купу на безліч областей купи однакового розміру, кожен з яких є суміжним діапазоном

віртуальної пам'яті. Виконуючи збирання сміття, G1 показує одночасну глобальну фазу маркування (тобто фазу 1, відому як маркування) для визначення життєздатності об'єктів у всій купі. Після завершення фази позначення G1 знає, які регіони в основному порожні. Спочатку він збирається в цих районах, що зазвичай дає значну кількість вільного простору (тобто фаза 2, відома як підмітання). Саме тому цей спосіб збору сміття називають garbage-first.

# 5 ПРАКТИЧНЕ ПОРІВНЯННЯ МЕТОДІВ ТА КОНФІГУРАЦІЙ У КОНТЕКСТІ ВИСОКОНАВАНТАЖЕНОЇ СИСТЕМИ ДЛЯ ПОШУКУ ІНФОРМАЦІЇ

## 5.1 Опис оточуючого середовища системи

У якості цільового середовища виконання була використана HotSpot JVM. Версія JDK: 1.8.0_222. У якості основної мови програмування був використаний Kotlin, що є JVM-сумісною мовою програмування. Також був використаний Spring Framework п’ятої версії для реалізації як моделі “потік на запит”, так і реактивної; ці реалізації представлені двома незалежними додатками. У ролі контейнера сервлетів для моделі “потік на запит” виступає Tomcat версії 9.0.27, а для реактивної моделі – Netty Web Server; для обох контейнерів була використана конфігурація Spring Boot за замовчуванням.

Сервер розгорнутий глобально та є доступним через мережу інтернет, тобто був налаштований віртуальний сервер через програмне забезпечення роутеру з подальшою переадресацією на локально розгорнутий сервер. Клієнтами серверів виступали декілька мобільних пристроїв що робили вказану за допомогою GUI розробленого додатку кількість запитів на розгорнутий сервер.

Для профілювання роботи веб-сервісу був використаний профайлер VisualVM, що є частиною JDK.

У якості інформаційної частини програмного забезпечення була використана СУБД MySQL, база даних якої була наповнена з датасету щодо пошукових запитів з ключовими словами відносно квитків на авіарейси та перельотів за допомогою власноруч створеного конвертера з файлів CSV формату. Усього у базі даних розташовується 96000 кортежів.

## 5.2 Підготовка, проведення та аналіз результатів роботи стандартної моделі прийому та обробки запитів

У контексті наступного ряду експериментів була використана модель прийому та обробки запитів “потік на запит” що реалізується фреймворком Spring MVC, а у якості контейнеру сервлетів виступає Tomcat.

На рисунку 5.1 показаний стан системи на момент запуску веб-сервісу з виконанням одного запиту для того щоб фреймворк мав змогу проініціалізувати усі компоненти що будуть використовуватися у ході експерименту. Н рисунку 5.2 показаний стан потоків системи. Для першого експерименту був використаний паралельний збірник сміття за допомогою додаткового флагу “-XX:+UseParallelGC”.

На рисунку 5.2 можна побачити, що в Tomcat було ініціалізовано пул потоків первинним об’ємом в 10 потоків виконання, що очікують вхідних запитів.

Варто відмітити, що для Tomcat використовується неблокуючий адаптер вводу-виводу (про це свідчать назви потоків, що містять префікс “nio” – non-blocking input-output).

У ході експерименту з двох смартфонів було відправлено по одній тисячі запитів на розгорнутий веб-сервіс.

На рисунку 5.3 показано стан системи після виконання двох тисяч запитів, по одній тисячі на кожен смартфон. На рисунку 5.4 показаний стан потоків.

Щодо споживання пам’яті, то для віртуальної машини було зарезервовано приблизно 820 мегабайт пам’яті, коли максимально використаний простір склав 526 мегабайт. Паралельний збірник сміття показав достатньо високу швидкість виконання, що обумовлюється кількістю потоків, що використовуються. Можна встановити, що збір об’єктів, що не використовуються, трапляється у момент, коли куча заповнює від 60 відсотків місця що виділяється для віртуальної машини.

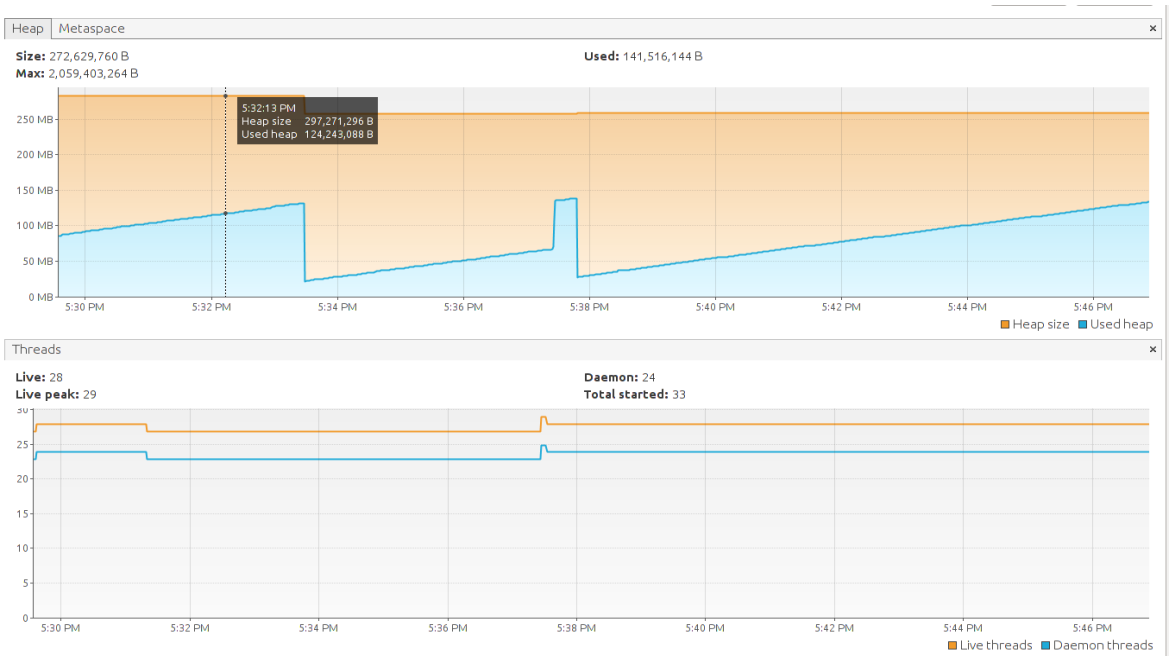


Рисунок 5.1 – Стан веб-серверу з пулом потоків після ініціалізації зі збірником сміття Parallel

Рисунок 5.2 – Стан потоків виконання веб-сервісу з пулом потоків зі збірником сміття Parallel

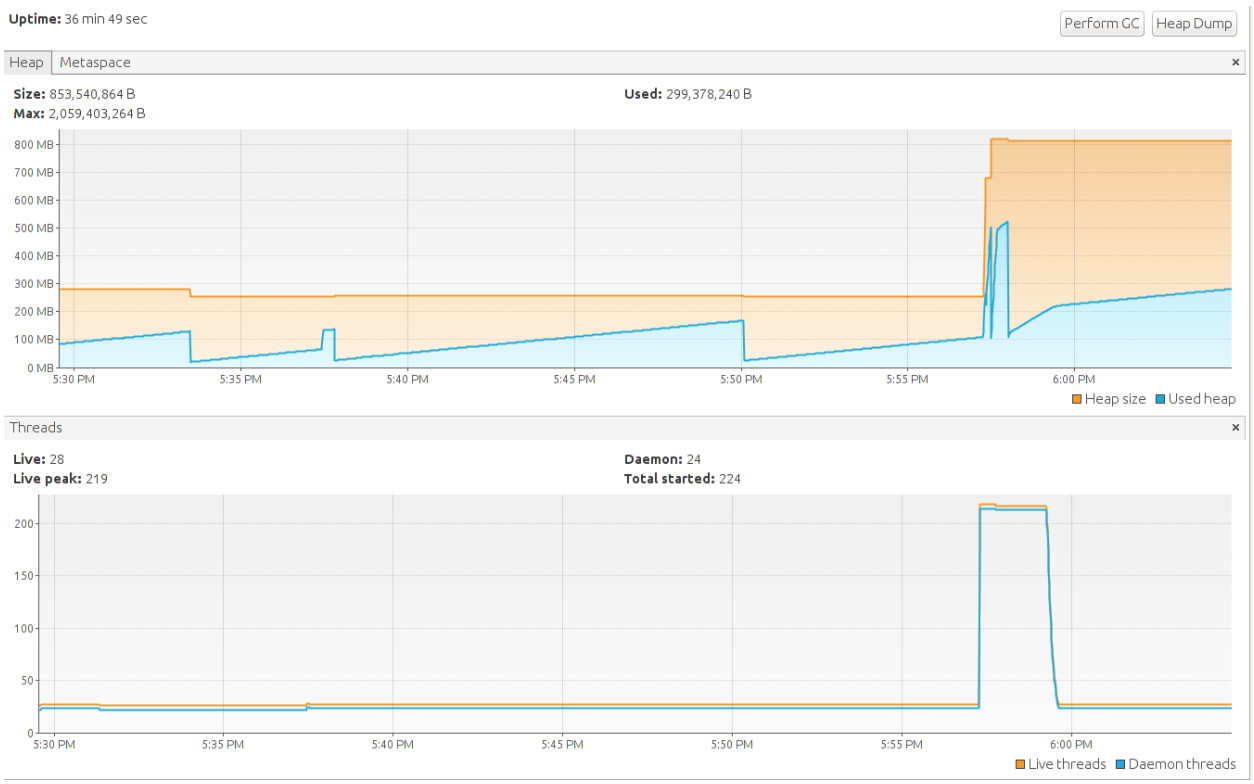


Рисунок 5.3 – Стан системи з пулом потоків після виконання двох тисяч одночасних запитів з паралельним збірником сміття

Щодо стану потоків виконання, можна відзначити що реалізація контейнеру сервлетів Tomcat є дуже ефективною, тому що замість блокування потоки знаходяться у стані “parked”, що значить, що потік, замість того щоб бути активним, але заблокованим, стає неактивним, та потоки, що знаходяться у цьому стані використовують набагато менше системних ресурсів. Варто відзначити, що максимальний обсяг пулу потоків за конфігурацією за замовчуванням становить 200, тому максимальна кількість потоків у системі становить 220, де 20 потоків у системі є системними. Якщо у систему не надходило достатньо запитів щоб максимальна кількість потоків працювала водночас, контейнер сервлетів починає повинно

звільняти потоки виконання: саме тому кількість активних потоків швидко впала до мінімальної відзначки в десять після виконання поставлених запитів.

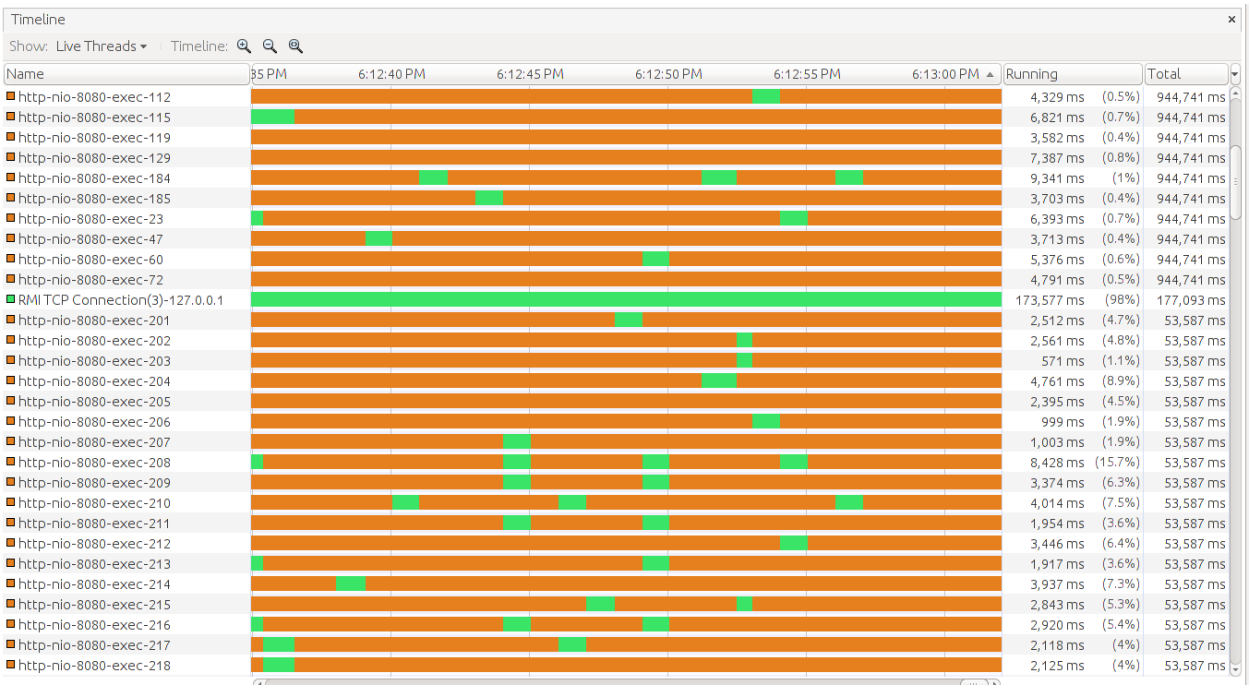


Рисунок 5.4 – Стан потоків системи з пулом потоків на момент виконання двох тисяч одночасних запитів з паралельним збірником сміття

Наступний варіант реалізації включає альтернативну реалізацію збірника сміття – G1. На рисунку 5.5 показано стан віртуальної машини після ініціалізації веб-сервісу.

Зі стану пам’яті можна встановити, що збірник мусору G1 чистить пам’ять набагато частіше ніж Parallel, а також у відносно стабільному темпі. Якщо збірник мусору Parallel збирав “мертві” об’єкти набагато менш детерміновано, тобто не у стабільному ритмі та із змінним порогом використовуємої віртуальною машиною пам’яті, то G1 робить це детерміновано щодо порогу використовуємого обсягу пам’яті та у стабільному ритмі – приблизно кожні 6 хвилин.

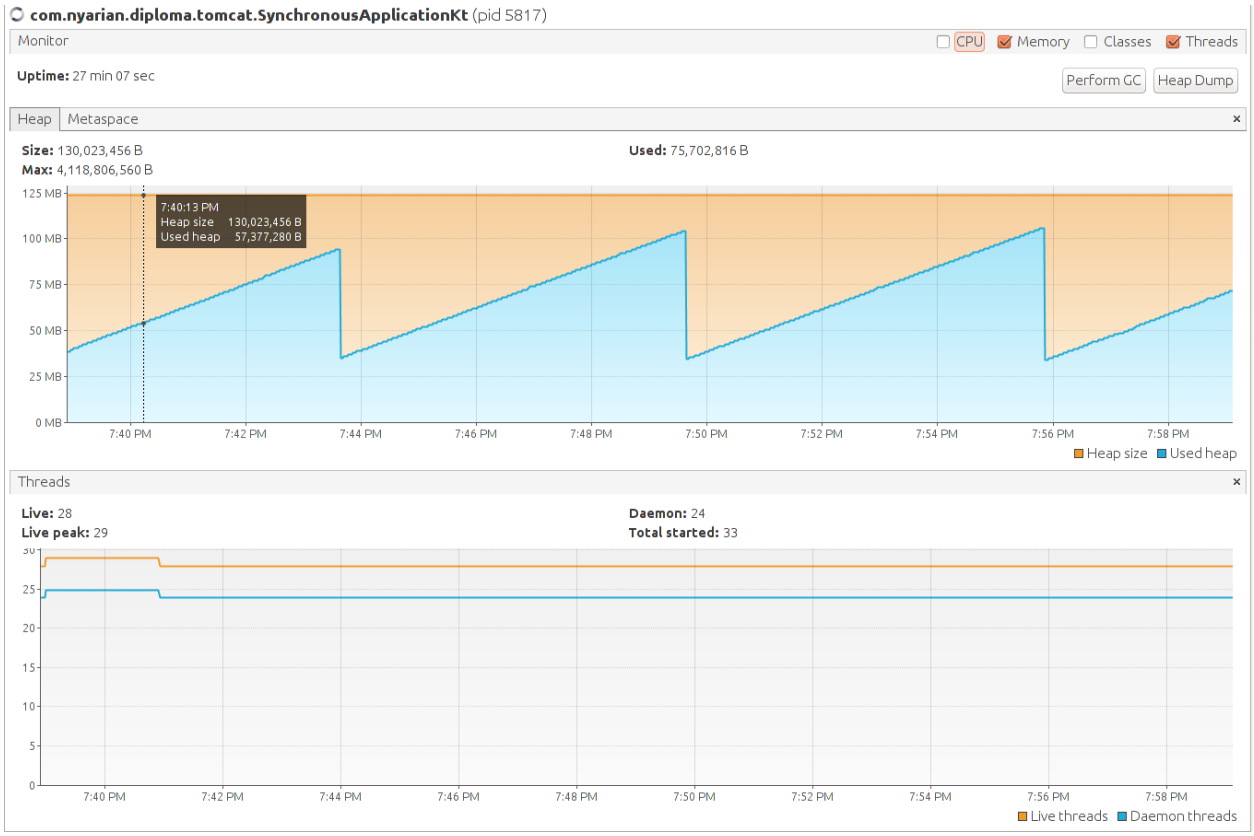


Рисунок 5.5 – Стан веб-серверу з пулом потоків після ініціалізації зі збірником сміття G1

На рисунку 5.6 зображений загальний стан використання пам’яті та потоків виконання після двох тисяч запитів. Обсяг пам’яті, що був виділений для віртуальної машини, становить 854 мегабайти; максимально використаний обсяг пам’яті становить 604 мегабайти. З цього можна постановити, що щодо споживання пам’яті збірник сміття Parallel дав кращий (менший) результат ніж G1.

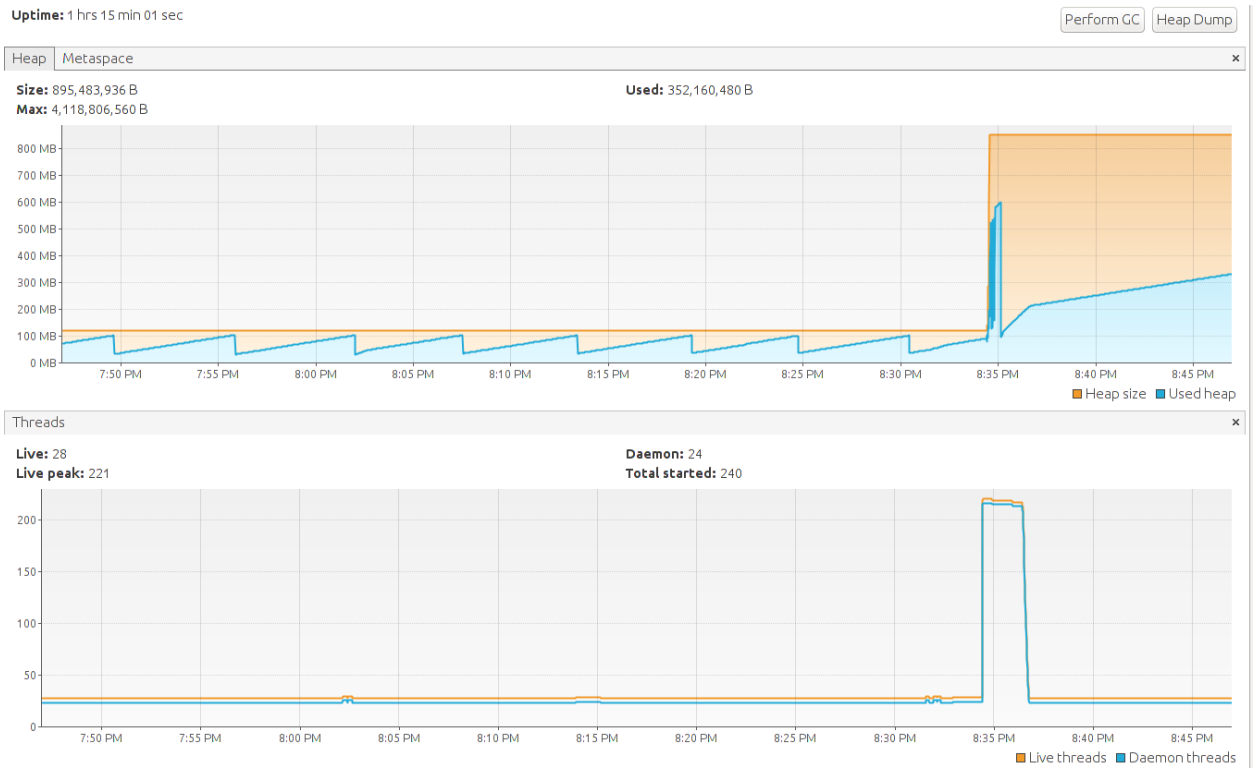


Рис. 5.6 – Загальний стан веб-сервісу з пулом потоків після виконання двох тисяч запитів з G1

## 5.3 Підготовка, проведення та аналіз результатів роботи реактивної моделі прийому та обробки запитів

У контексті наступного ряду експериментів була використана реактивна модель прийому та обробки запитів що реалізується фреймворком Spring Reactor, а у якості контейнеру сервлетів виступає Netty.

На рисунку 5.7 показаний стан системи на момент запуску веб-сервісу з виконанням одного запиту для того щоб фреймворк мав змогу проініціалізувати усі компоненти що будуть використовуватися у ході експерименту. На рисунку 5.8 показаний стан потоків системи. Для першого експерименту був використаний паралельний збірник сміття за допомогою додаткового флагу “-XX:+UseParallelGC”.

На рисунку 5.8 можна побачити, що у сервісі було ініціалізовано сім потоків виконання для періодичного опитування сокетів щодо нових запитів. На відміну від попередньої реалізації, а саме – від пулу потоків, що знаходились у стані “parked” доки нові запити не надійшли, потоки у даній реалізації є активними та здійснюють постійне опитування сокетів. Це створює деякі накладні витрати на підтримку веб-сервісу в пасивному стані, тобто у стані, коли він не отримує ніяких запитів. Проте, загальна кількість створених потоків менша ніж в попередній реалізації аж на дев’ять, що є достатньо великою кількістю. Підсумком цього є те, що в загальному випадку для систем з малим навантаженням рішення с пулом потоків та синхроною реалізацією є більш ефективним з точки зору споживаних ресурсів, тому що для підтримки неактивних потоків потрібно дуже мало системних ресурсів.

Для того, щоб мати змогу порівняти результати з реалізацією моделі “потік на з’єднання” потрібно створити середовище, еквівалентну тому середовищу що оточувало попередні експерименти. Тобто, потрібно здійснити дві тисячі паралельних запитів на сервер. Стан системи після обробки запитів наведений на рисунку 5.9, стан потоків виконання – на рисунку 5.10.

З описання стану системи можна постановити, що розмір купи, що зарезервований, на двадцять відсотків менший ніж у попередній реалізації. Обсяг купи, що був зайнятий під час обробки запитів, менший на 49%. Кількість потоків виконання становила 21, на відміну від попередньої реалізації, що становила 220, тобто різниця в 90%. Варто також відмітити, що сміття було зібрано два рази, на відміну від трьох разів у попередньому випадку.

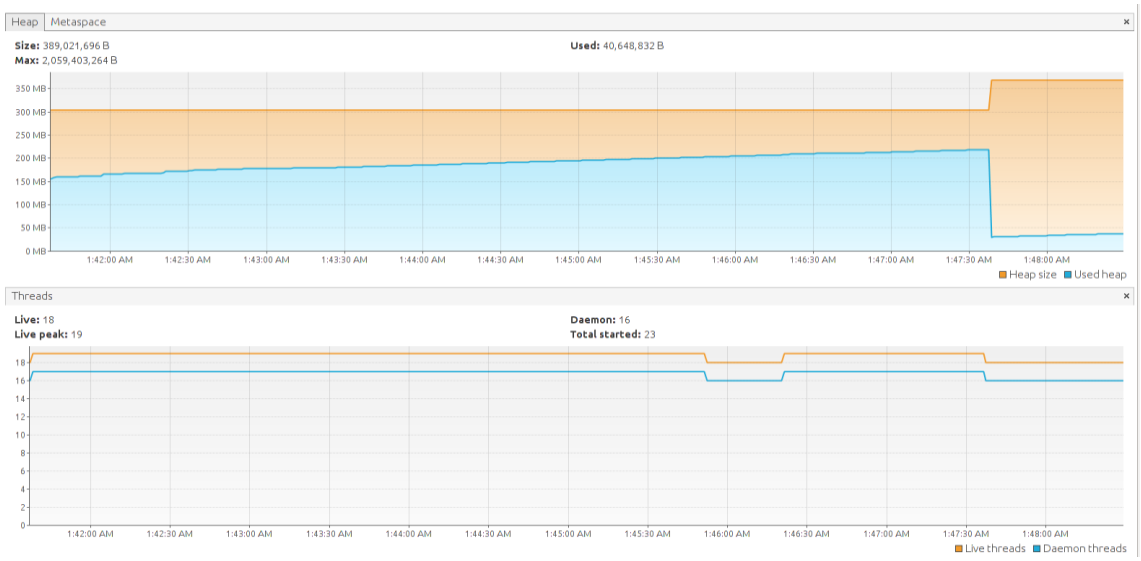


Рисунок 5.7 – Стан реактивного веб-серверу після ініціалізації зі збірником сміття Parallel

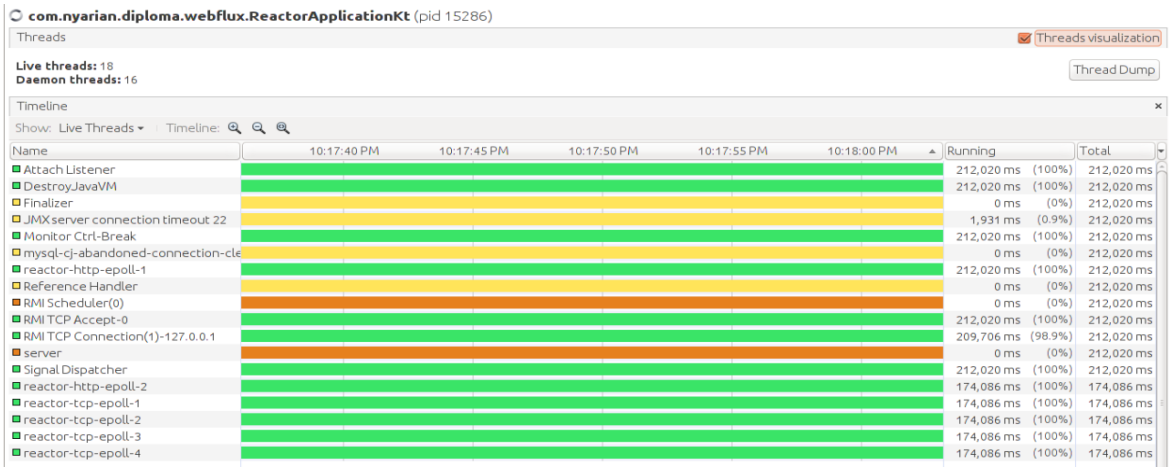


Рисунок 5.8 – Стан потоків виконання реактивного веб-сервісу зі збірником сміття Parallel

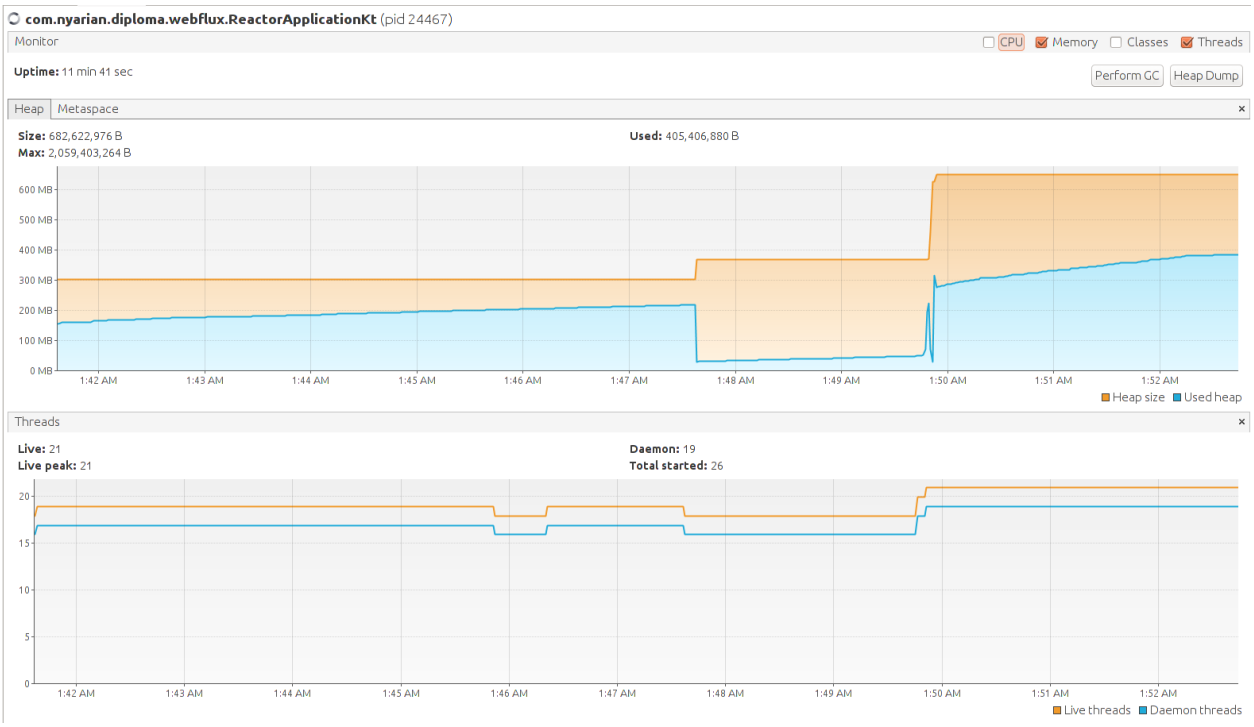


Рисунок 5.9 – Загальний стан системи реактивного веб-сервісу зі збірником сміття Parallel після виконання 2000 запитів

Рисунок 5.10 – Стан потоків виконання реактивного веб-сервісу зі збірником сміття Parallel на момент виконання 2000 запитів

Щодо потоків виконання, то можна постановити, що поточна реалізація створила усього три допоміжні потоки (три нижніх на рисунку 5.10), коли попередня реалізація створила 190 допоміжних потоків, і могла створювати стільки, скільки надійшло запитів, якщо пул потоків був необмеженим.

Для другого експерименту використаємо збірник сміття G1. Загальний стан системи після ініціалізації представлений на рисунку 5.11. Можна постановити, що поточний зарезервований обсяг купи складає 40% від зарезервованого обсягу купи з Parallel збірником сміття. Максимальний обсяг зайнятого місця в купі становив 94 мегабайти проти 220 мегабайт у випадку з Parallel GC, тобто складає 43% від обсягу купи у попередній реалізації.

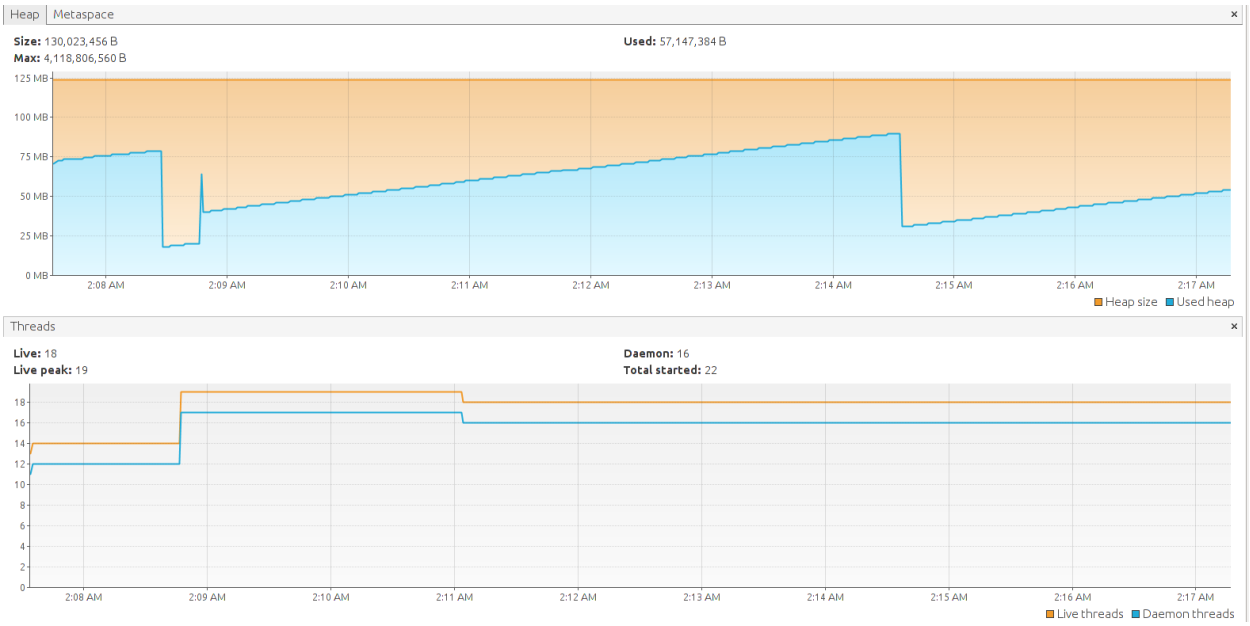


Рисунок 5.11 – Стан реактивного веб-серверу після ініціалізації зі збірником сміття G1

Результат проведення експерименту для поточної реалізації представлений на рисунку 5.12. Зарезервований обсяг купи становить 828 мільйонів байт, на відміну від 682 мільйонів у попередній реалізації. Тобто, результат гірший на 18%. Максимально

зайнятий простір становить 525 мільйонів байт, коли для попередньої реалізації він становив 320 мільйонів. Тобто, результат погіршився на 39 відсотків. Проте збір сміття був виконаний лише один раз на відміну від попередньої реалізаціями з двома зборами сміття. З цього можна постановити, що хоча ця реалізація потребує більше пам’яті, вона затримує систему на 50% менше фазою “stop the world”. Обидві реалізації мають свої переваги та недоліки, та проте у разі високонавантаженої системи стратегія G1 більш вигідна: у реальних умовах навантаження рідко різко падає, а резервація пам’яті для купи є достатньо складним та важким процесом як для процесу, так і для самої операційної системи; також, менша кількість проходів збірки сміття є в загальному випадку більш привабливою характеристикою ніж такий відхил в навантаженні на пам’ять.

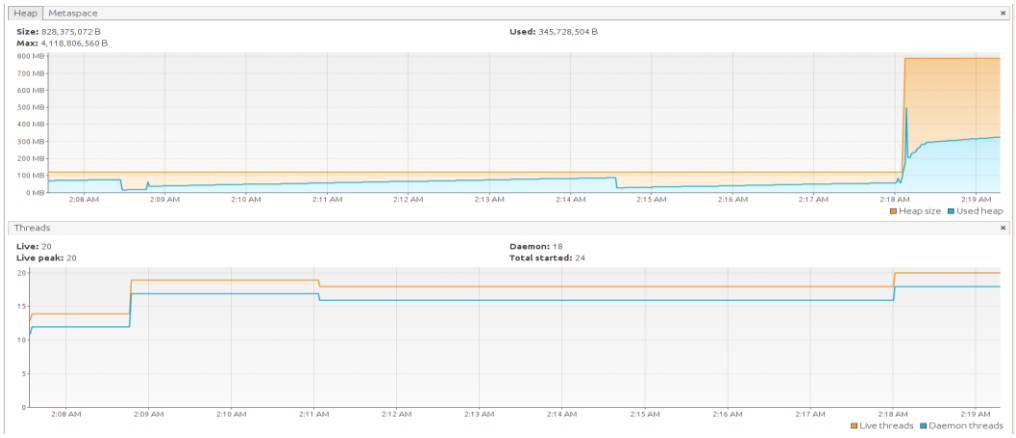


Рисунок 5.12 – Стан реактивного веб-серверу після двох тисяч запитів зі збірником сміття G1

## 5.4 Порівняння отриманих результатів

У ході експериментів було визначено, що для моделі “потік на запит”, що обмежена обсягом пулу потоків, що обробляють запити, збірник сміття Parallel є більш ефективним рішенням. Щодо реактивної реалізації, G1 був визначений як більш вигідне для високонавантаженої системи рішення.

Щодо зарезервованого обсягу купи, реактивна система є кращою на 3% – 852 мільйони байт проти 828 мільйонів (рис 5.13).

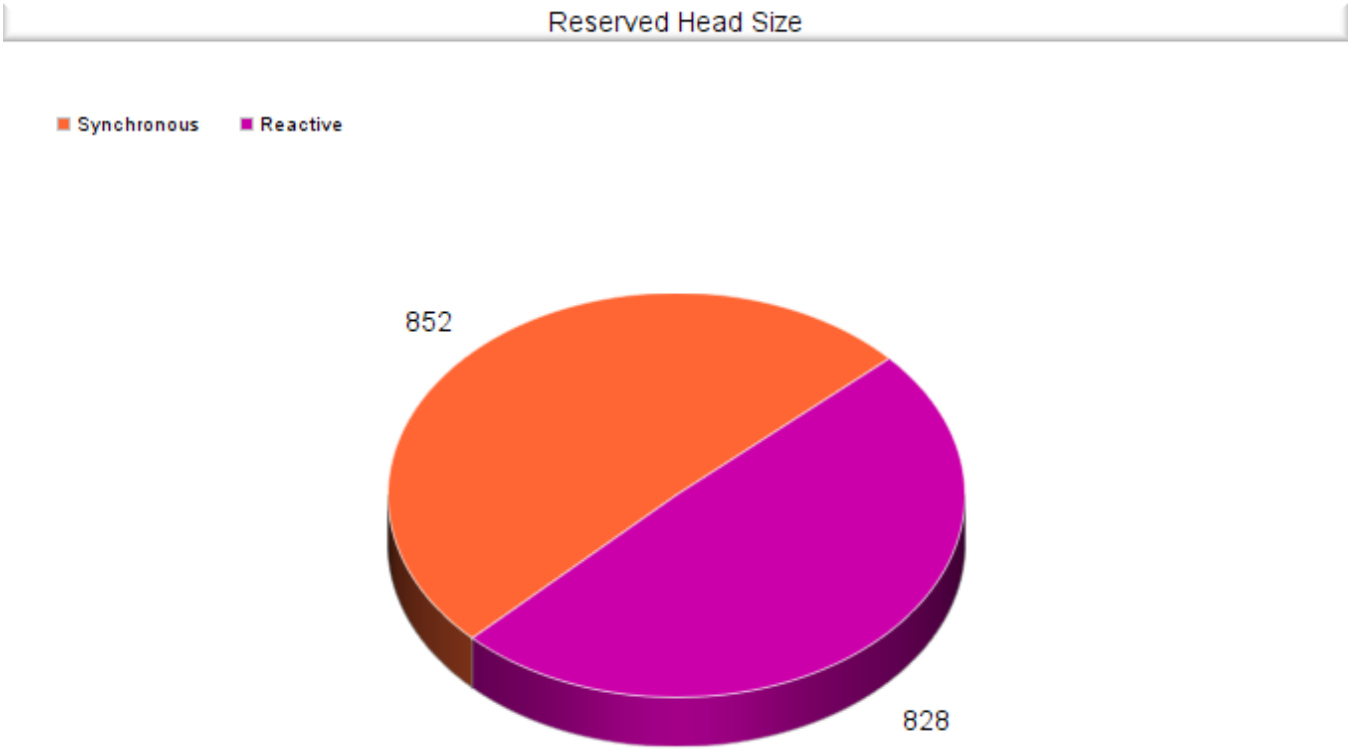


Рисунок 5.13 – Порівняння зарезервованого обсягу купи

Щодо використаного обсягу купи, відхилення у роботі систем менше ніж 0.005% (рис. 5.14).

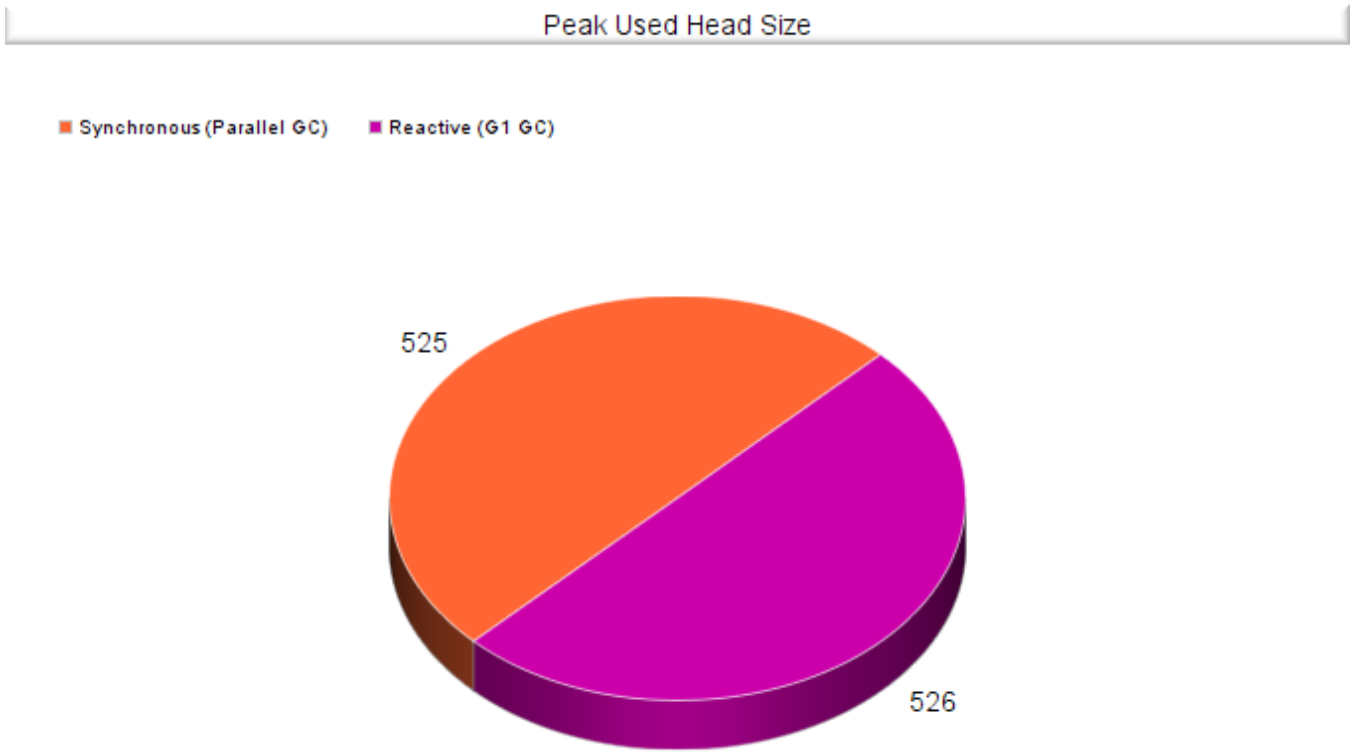


Рисунок 5.14 – Порівняння максимально використаного обсягу купи

Щодо кількості махів збору сміття, то синхронна система виконала 2 махи проти одного маху в контексті реактивної системи (рис. 5.15); з цього можна постановити, що навантаження на пам’ять у разі реактивної системи набагато менше, ніж у разі синхронної, тому що купа була заповнена до порогового значення двічі для першого середовища виконання. Це означає, що щодо навантаження на пам’ять реактивна система має набагато більш гарний результат.

Щодо стану потоків виконання, реактивна система є кращою ніж синхронна на 90% – 21 потік проти 220, де 220 є встановленим пороговим значенням (рис. 5.16). Швидкість, за яку система надала результат клієнту, є трохи вищою у разі реактивної системи, що свідчить про неефективне керування потоками в разі синхронної системи щодо реактивної.

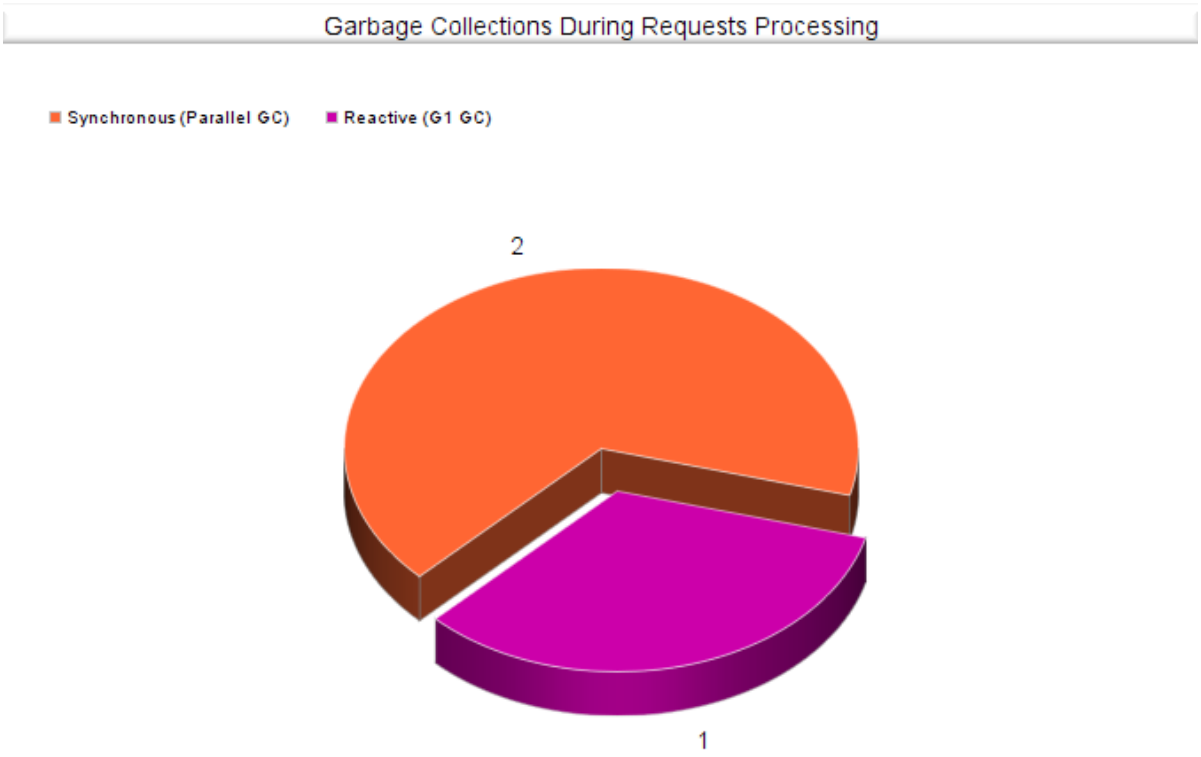


Рисунок 5.15 – Порівняння кількості збірок мусору при обробці запитів

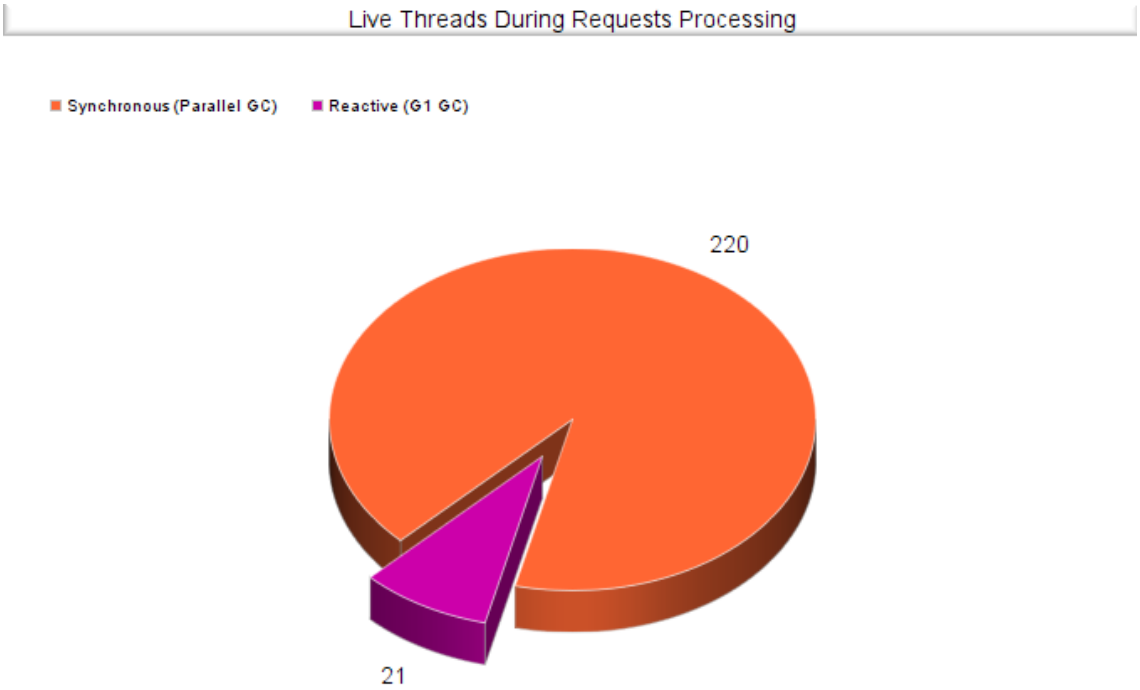

Рисунок 5.16 – Порівняння кількості активних потоків у системі

Щодо часу виконання, то синхронна реалізація надала відповіді по двом тисячам запитів за 20 секунд та 777 мілісекунд. У контексті реактивної реалізації дві тисячі запитів були опрацьовані за 23 секунди та 284 мілісекунди. Порівняння наведене на рисунку 5.17.

З цього можна постановити, що реактивна система є набагато більш ефективною ніж синхронна її реалізація у контексті даного середовища та навантаження на систему. Також варто відзначити, що для реактивної системи реалізація збірника сміття за замовчуванням виявилася більш ефективною ніж попередня реалізація. Проте якщо першочерговим фактором системи є швидкість, то синхронна модель надала набагато кращий результат, тому з точки зору швидкості виконання синхронна модель є більш ефективною ніж реактивна.

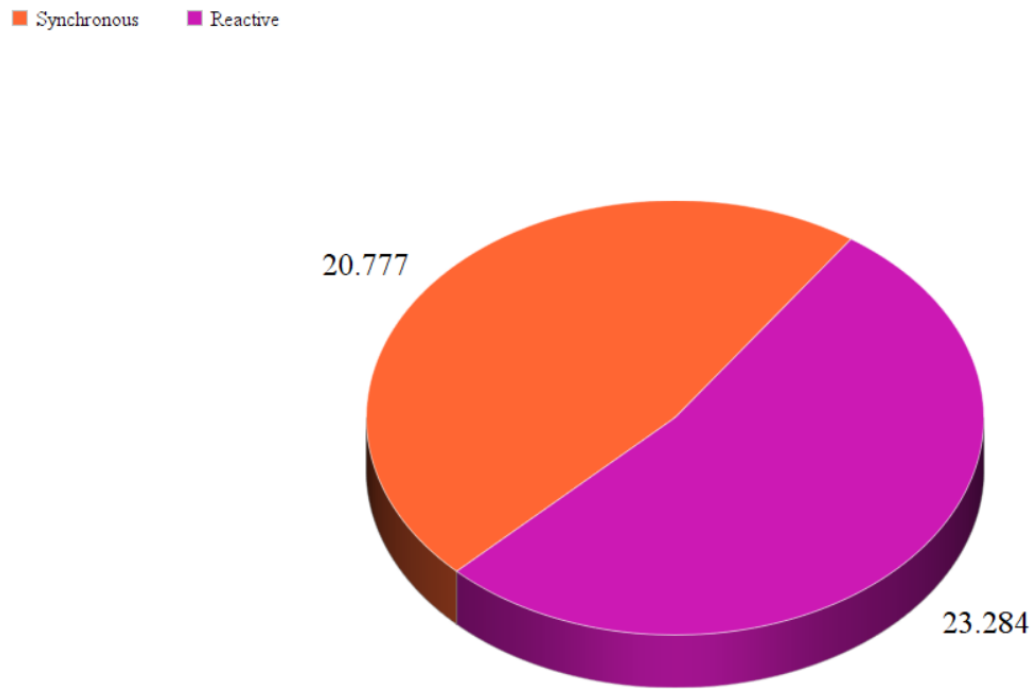


Рисунок 5.17 – Порівняння часу обробки двох тисяч запитів

# ВИСНОВКИ

У якості основної проблеми даної атестаційної роботи була освітлена проблема обробки даних у контексті високонавантажених систем, а саме – високонавантаженої системи пошуку інформації. Була підкреслена її актуальність та виділені шляхи рішення.

Основними підходами щодо вирішення цієї проблеми були виділені передові методи утилізації середовища виконання що підтримує паралелізм, а саме – метод асинхронного програмування та його ідеологічний нащадок – метод реактивного програмування, а також методи та інструменти що входять до галузі Big Data. Вибраним методом вирішення проблеми був вибраний метод реактивного програмування, що має на увазі ефективну з точки зору утилізації системних ресурсів та потоків виконання обробку потенційно нескінченного потоку подій (у контексті розробленої системи – запитів), це обумовлено більшою гнучкістю порівнюючи із методами та інструментами категорії Big Data.

Поставлені у практичній частині роботи експерименти мали на увазі порівняння ефективності стандартного підходу щодо проектування та реалізації веб-сервісів, а саме моделі “потік на запит” з обмеженим пулом потоків що використовуються для обробки цих запитів з ефективністю системи, що була реалізована за допомогою реактивного програмування з подальшою оцінкою ефективності альтернативних конфігурацій середовищ виконання HotSpot JVM, а саме – альтернативних реалізаціях компоненту збірника сміття. У ході проведення експериментів було доказано, що реактивна модель у контексті описаної оточуючого середовища системи є набагато більш ефективним підходом щодо проектування та реалізації потенційно високонавантаженого веб-сервісу, особливо з точки зору використання та резерву потоків виконання та циклів збору сміття. Проте з точки зору виключно швидкості обробки запитів синхронна модель мала на

15% кращий результат, тому для систем де швидкість є першочерговим аспектом системи та навантаження співпадає із середовищем що було активним у процесі проведення експериментів в ході виконання даної атестаційної роботи синхронна модель обробки запитів і даних можна вважати більш ефективним рішенням.

На підставі виконаних у атестаційній роботі задач можна постановити, що атестаційна робота відповідає постановці завдання та була виконана у повному об’ємі.

# ПЕРЕЛІК ПОСИЛАНЬ


1. Event Loop. [Електронний ресурс]. 16.10.2019. — Режим доступу: https://en.wikipedia.org/wiki/Event_loop
2. Volume, velocity, and variety: Understanding the three V's of big data. [Електронний ресурс]. 21.03.2018. — Режим доступу: https://www.zdnet.com/article/volume-velocity-and-variety-understanding-the-three-vs-of-big-data/
3. Data Veracity. [Електронний ресурс]. 28.11.2012. — Режим доступу: https://www.datasciencecentral.com/profiles/blogs/data-veracity
4. How to Ensure the Validity, Veracity, and Volatility of BigData. [Електронний ресурс].11.06.2017. — Режим доступу: https://www.dummies.com/programming/big-data/engineering/how-to-ensure-the-validity-veracity-and-volatility-of-big-data/
5. Big Data. Часть 2: Hadoop. [Електронний ресурс]. 05.10.2015. — Режим доступу: https://habr.com/ru/company/dca/blog/268277/
6. Паралельні обчислення. [Електронний ресурс]. 03.10.2019 . — Режим доступу: https://www.techopedia.com/definition/8777/parallel-computing
7. Ananth Grama, George Karypis, Vipin Kumar, Anshul Gupta. Introduction to Parallel Computing, Second Edition. – “Pearson”, 2003. – C.13.
8. Richard Blewett, Andrew Clymer. Pro Asynchronous Programming with .NET, First Edition. – “Apress”, 2013. – 6 стр.
9. Andrii Iliunin. Coroutines In Kotlin, Part 1. [Електронний ресурс]. 03.12.2018 . — Режим доступу: https://docs.google.com/presentation/d/1mqUF3ioTNdthQ_U4yEJ3ieGGw6d9kk--_OMxa0upuK8/edit?usp=sharing
10. Tomasz Nurkiewicz & Ben Christensen. Reactive Programming with RxJava: Creating Asynchronous, Event-Based Applications, First Edition. – “O’REILLY Media”, 2017. – C. 3.

11. RxNetty vs Tomcat Performance Results. [Електронний ресурс]. 09.09.2015. — Режим доступу: https://github.com/Netflix-Skunkworks/WSPerfLab/blob/master/test-results/RxNetty_vs_Tomcat_April2015.pdf
12. Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley. The Java Virtual Machine Specification, Java SE 7 Edition, [Електронний ресурс]. 28.02.2013. — Режим доступу: https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-1.html#jvms-1.2
13. CLR vs JVM: How the Battle Between C# and Java Extends to the VM-Level . [Електронний ресурс]. 24.01.2019. — Режим доступу: https://blog.overops.com/clr-vs-jvm-how-the-battle-between-net-and-java-extends-to-the-vm-level/
14. The JVM Architecture Explained. [Електронний ресурс]. 02.09.2016. — Режим доступу: https://dzone.com/articles/jvm-architecture-explained
15. JVM Garbage Collectors. [Електронний ресурс]. 12.12.2018. — Режим доступу: https://www.baeldung.com/jvm-garbage-collectors/