КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

ІМЕНІ ТАРАСА ШЕВЧЕНКА

ФАКУЛЬТЕТ РАДІОФІЗИКИ, ЕЛЕКТРОНІКИ ТА КОМП'ЮТЕРНИХ СИСТЕМ

Кафедра радіотехніки та радіоелектронних систем

До захисту допущено: «На правах рукопису»
Завідувач кафедри _____ Ігор АНІСІМОВ
« __ » червня 2023 р.

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

на тему:

«РЕАЛІЗАЦІЯ БАГАТОПОТОКОВОЇ КЛІЄНТ-СЕРВЕРНОЇ ПРОГРАМИ ЧАТУ ЗА ДОПОМОГОЮ JAVA»

Виконав:
студентка 4-го курсу
денної форми навчання
спеціальності 172 - Телекомунікації та радіотехніка
ОП «Інформаційна безпека телекомунікаційних систем і мереж»
Кохтаєва Аліна Еівазівна _____

Науковий керівник:
к.ф.-м.н., доц. Кононов Михайло Володимирович _____

Рецензент:
к.ф.-м.н., доц. Судаков Олександр Олександрович _____

Засвідчую, що у цій бакалаврській роботі
немає запозичень з праць інших авторів без
відповідних посилань
Студент _____

Робота допущена до захисту в ЕК рішенням кафедри радіотехніки та радіоелектронних систем від «22» червня 2023 р., протокол № 21.

Завідувач кафедри радіотехніки та радіоелектронних систем,
доктор фіз.-мат. наук, професор
Анісімов Ігор Олексійович _____

Київ – 2023

ЗМІСТ

ВСТУП

Актуальність теми: попит на спілкування в режимі реального часу між користувачами або користувачем та сервером та потреба в спеціалізованих програмних засобах для реалізації чату пов'язана з тим, що виділений застосунок для спеціальних потреб підприємців є більш ефективним та безпечним за ті, що створені для широкого застосування. Таким чином, існує необхідність розробки спеціалізованих застосунків для чату, особливо тих, які використовують багатопотоковість, що забезпечує повноту використання ресурсів комп'ютера. Для реалізації таких програмних засобів є доречним використати клієнт-серверну архітектуру. З метою забезпечення гнучкості та кросплатформенного використання є зручним за основу проекту використати Java.

Метою дослідження є створення багатопотокового (створення потоків та їх синхронізація) чату на основі сокетів; реалізація клієнтської сторони та створення сервера обробника подій.

Об'єкт дослідження: ефективність використання комп'ютера.

Предмет дослідження: застосування сокетів для програми багатопотокового чату.

Задачі дослідження:

1. Створення клієнта на мові програмування Java.
2. Створення сервера, що стабільно функціонує, обмінюється повідомленнями з програмою-клієнтом і обробляє описані в них події.
3. Створення канальної системи спілкування між програмою-клієнтом та програмою-сервером на основі використання сокетів.

РОЗДІЛ 1. ТЕОРЕТИЧНІ ВІДОМОСТІ

У даному розділі будуть розглянуті основні поняття, необхідні для реалізації проекту, а також наведені технології, які використовувалися для його імплементації.

1.1 Багатопотоковість як спосіб збільшення ефективності програмних продуктів

Будь-яка програма створює процес, для якого операційна система виділяє потрібні ресурси, зокрема адресний простір. Виконання процесу починається з виконання основного потоку – програмного коду. Такий потік ініціалізує, виконує та завершує роботу процесора. Паралельно з ним можуть бути запущені додаткові потоки, але вони діють в межах одного й того ж адресного простору, тому що для них додаткові ресурси не будуть виділятися. З цього моменту створення декількох потоків програма стає багатопотоковою.

Багатопотоковість відноситься до техніки паралелізму (концепції виконання кількох завдань або операцій одночасно з метою покращення продуктивності та більш ефективного використання доступних системних ресурсів), яка дозволяє одночасно виконувати декілька потоків виконання в межах одного процесу. Це дозволяє програмі виконувати кілька завдань одночасно, використовуючи потужність сучасних процесорів із кількома ядрами або паралельно виконуючи завдання на одному ядрі за допомогою методів розподілу часу.

Потоки — це легкі одиниці виконання в процесі, які можуть незалежно виконувати завдання. Потік міститься всередині процесу, а різні потоки

одного процесу розділяють деякі ресурси, в той час як різні процеси ресурси не розділяють. Кожен потік має власний програмний лічильник, стек і локальні змінні, але вони спільно використовують той самий простір пам’яті, файлові дескриптори та інші ресурси в рамках процесу. Багатопотоковість забезпечує модель програмування, яка дозволяє розробникам писати код, в якому декілька потоків працюють незалежно та потенційно взаємодіють або синхронізуються один з одним.

При належному використанні потоки можуть зменшити витрати на розробку та обслуговування та підвищити продуктивність складних програм. Потоки полегшують моделювання роботи та взаємодії користувачів, перетворюючи асинхронні робочі процеси на здебільшого послідовні. Вони також можуть перетворити заплутаний код у прямолінійний код, який легше писати, читати та підтримувати. [2]

В однопотоковій програмі в будь-який момент часу виконується лише одна послідовність інструкцій. Однак у багатопотоковій програмі може існувати кілька послідовностей інструкцій, що виконуються одночасно, що дозволяє різним частинам програми виконуватися незалежно.

Головною перевагою багатопотоковості є можливість паралельно виконувати незалежні завдання, тим самим скорочуючи загальний час виконання. Розділивши велике завдання на менші підзадачі, які можна виконувати одночасно, можна досягти значного підвищення швидкості. Наприклад, графічна програма може використовувати один потік для візуалізації зображень і інший потік для обробки введених користувачем даних, що забезпечує більш чутливу та ефективну роботу користувача. Вона дозволяє програмному забезпеченню краще використовувати доступні системні ресурси, такі як ядра процесору і пам'ять. Розподіляючи завдання між кількома потоками, багатопотокова програма може більш ефективно використовувати всі доступні ресурси, що призводить до більш швидкого та ефективного виконання.

Багатопотоковість може допомогти покращити реакцію програмного забезпечення, дозволяючи певним завданням виконуватися у фоновому режимі, поки користувач взаємодіє з програмою. Візьмемо за приклад текстовий редактор, за допомогою якого користувачі можуть відкривати великі файли. Коли користувач відкриває особливо великий файл, програмі потрібно завантажити вміст файлу в пам’ять, що може зайняти значну кількість часу. Без багатопотоковості програма не реагувала б під час процесу завантаження, оскільки основний потік був би зайнятий читанням і обробкою файлу, не здатний відповідати на дії користувача. А за допомогою багатопотоковості створюється можливість розділити ці дві задачі. Також вона може спростити програмування, дозволяючи розробникам розбивати складні завдання на менші, більш керовані частини, які можна виконувати паралельно. Це може полегшити написання програмного забезпечення, що масштабується – тобто описує здатність покращити пропускну здатність або ємність при додаванні додаткових обчислювальних ресурсів (таких як додаткові процесори, пам’ять, сховище або пропускна здатність вводу-виводу) – та підтримується. [5] Окрім того, багатопотоковість забезпечує більшу гнучкість у проектуванні програмних систем. Використовуючи кілька потоків, розробники можуть створювати більш складні програми, які можуть виконувати ширший спектр завдань і робочих навантажень. [2]

В основі багатопотоковості в програмах чату лежить створення та планування потоків. Коли користувачі підключаються до програми, базова система розподіляє ресурси та призначає виділений потік для обробки взаємодії кожного користувача. Цей розподіл застосовує функціональність операційної системи, використовуючи складні алгоритми, такі як циклічне планування або планування на основі пріоритетів, щоб забезпечити справедливий доступ до системних ресурсів між конкуруючими потоками.

Досягнення безперебійного обміну повідомленнями між користувачами в багатопотоковому середовищі потребує складних методів міжпотокового зв’язку. Для полегшення передачі повідомлень між потоками

використовуються потокобезпечні структури даних, такі як черги або буфери. Оскільки один потік отримує й обробляє повідомлення, він ставить його в чергу для передачі, тоді як інший потік, призначений для трансляції, знімає повідомлення з черги та розповсюджує його відповідним одержувачам. Ці канали зв’язку, часто синхронізовані за допомогою механізмів блокування, забезпечують безперебійну передачу повідомлень, мінімізуючи затримки та забезпечуючи взаємодію між користувачами в реальному часі. [4]

1.2 Особливості клієнт-серверної архітектури на боці сервера

Архітектура клієнт-сервера — це обчислювальна модель, яка складається з серверу та клієнту. Сервер – це програма, яка надає клієнтам можливість користуватися своїми ресурсами, а клієнт – програма, що використовує мережеві ресурси, а не власні. [1] У цій архітектурі сервер відповідає за керування даними та ресурсами, а клієнти відповідають за надсилання запитів на ці ресурси, тобто клієнти надсилають запити серверу, а сервер відповідає на цей запит. Ця модель широко використовується в корпоративних обчисленнях, де клієнти та сервери можуть бути розташовані в різних місцях і спілкуватися через мережу.

Зв'язок між клієнтами і серверами може бути як синхронним, так і асинхронним. Синхронне спілкування передбачає очікування клієнтом відповіді сервера перед тим, як продовжити його виконання. У цьому типі зв’язку клієнт надсилає запит на сервер і чекає на відповідь. Сервер обробляє запит і надсилає відповідь клієнту. Потім клієнт використовує відповідь, щоб продовжити виконання алгоритму зв’язку з сервером. З іншого боку, асинхронний зв’язок дозволяє клієнту продовжувати виконання, чекаючи на відповідь сервера.

В клієнт-серверній архітектурі існують різні види “спілкування” клієнта та сервера, а саме:

- 2-рівнева архітектура: у цій архітектурі клієнт спілкується безпосередньо з сервером. Клієнт відповідає за логіку презентації, тоді як сервер відповідає за бізнес-логіку та зберігання даних. Ця архітектура проста у реалізації, але може бути менш масштабованою та менш безпечною;
- Багаторівнева архітектура: у цій архітектурі система поділена на кілька рівнів, кожен з яких відповідає за певний набір завдань. Ця архітектура є гнучкою, але також є найскладнішою для реалізації; [6]
- 3-рівнева архітектура: у цій архітектурі клієнт зв’язується з сервером додатків, який, у свою чергу, спілкується з сервером бази даних. Логіка представлення, бізнес-логіка та зберігання даних розділені на власні рівні. Ця архітектура більш масштабована та безпечна, ніж 2-рівнева архітектура, але є більш складною для реалізації;
- P2P (peer to peer) архітектура: у цій архітектурі всі вузли в мережі діють як клієнти, так і сервери, спілкуючись один з одним напряму. Ця архітектура є високодецентралізованою, але також може бути менш безпечною для певних задач та складнішою в управлінні;
- Хмарна архітектура: у цій архітектурі сервер розміщено в хмарному середовищі, наприклад Amazon Web Services або Microsoft Azure. Ця архітектура має високу масштабованість і гнучкість, і може бути менш дорогою, ніж традиційна клієнт-серверна архітектура, але також може бути складнішою в управлінні. [7]

В архітектурі клієнт-сервер багатопотоковість відіграє вирішальну роль у забезпеченні ефективної й одночасної обробки сервером кількох запитів

клієнтів. Програма отримує запити від кількох клієнтських програм і надає відповіді на основі цих запитів.

Одним з варіантів реалізації сервера є модель «потік на клієнта». В цьому випадку на сервері виділяється (відкривається, або виділяється з пула) окремий потік для кожного клієнта, який до нього підключається, призначений для обробки цього конкретного клієнта. Кожен з цих потоків прослуховує вхідні повідомлення від клієнта, і коли повідомлення отримано, він транслює його всім іншим підключеним клієнтам.

Кожен з потоків може бути реалізований як цикл, в якому прослуховуються запити клієнта, зчитуються дані від клієнта, обробляються запити, генеруються відповіді, які надсилаються назад клієнту. Цей цикл продовжується, доки клієнтське підключення не буде розірвано. Клієнтська програма також зазвичай використовує багатопотоковість, коли один потік прослуховує вхідні повідомлення від сервера, а інший потік надсилає повідомлення на сервер. Це дозволяє клієнту отримувати та надсилати повідомлення одночасно без блокування.

Багатопотоковість використовується сервером для обслуговування декількох клієнтів одночасно, без блокування, тобто не чекаючи завершення кожного завдання, перш ніж переходити до наступного. Крім того цей паралелізм дозволяє серверу повноцінно використовувати доступні системні ресурси, такі як ядра центрального процесору (ЦП), і забезпечує кращу масштабованість. Зазначимо, що заради масштабованості на відміну від забезпечення максимальної швидкодії суто розрахункової задачі розділенням на потоки в даному випадку кількість потоків не має бути обмежена кількістю ядер.

Багатопотоковість в архітектурі клієнт-сервер також передбачає управління життєвим циклом клієнтських потоків. Це включає належне створення потоку, завершення та очищення ресурсів. Пули потоків можуть реалізовувати такі стратегії, як динамічне створення потоків або повторне використання потоків для оптимізації використання ресурсів. [8]

1.3 Стек протоколів TCP/IP

Зв’язок між клієнтом та сервером може здійснюватися за допомогою різних протоколів, наприклад TCP, HTTP, UDP.

Зауважимо, що TCP/IP використовується як базовий стек для мережевого зв’язку між клієнтом і сервером за допомогою сокетів, які є кінцевими точками для зв’язку між двома процесами.

TCP/IP — це складний і фундаментальний стек протоколів, який служить основою сучасної мережі, сприяючи надійному та ефективному зв’язку між взаємопов’язаними комп’ютерними мережами. TCP/IP втілює комплексний набір правил, протоколів і процедур, які забезпечують плавний і стандартизований обмін пакетами даних між пристроями. Він складається з кількох протоколів, які працюють на різних рівнях мережевого стеку та працюють разом, щоб забезпечити надійний і ефективний зв’язок між пристроями.

Стек протоколів TCP/IP включає в себе такі протоколи:

- інтернет-протокол (IP): відповідає за надання схеми адресації, яка однозначно ідентифікує кожен пристрій у мережі. Він також забезпечує функцію маршрутизації, яка дозволяє пересилати пакети з одного пристрою на інший через різні мережі;
- протокол керування передачею (TCP): відповідає за забезпечення надійної, упорядкованої та перевіреної на помилки доставки даних між програмами, що працюють на різних пристроях. Він встановлює віртуальне з’єднання між відправником і одержувачем і забезпечує доставку пакетів у правильному порядку та без помилок;
- протокол дейтаграм користувача (UDP): простіший протокол без з’єднання, який використовується для програм, які вимагають

низької затримки та не потребують гарантій надійності, які надає TCP;

- протокол керуючих повідомлень Інтернету (ICMP): ICMP використовується для повідомлення про помилки та іншої інформації, пов’язаної з мережею, між пристроями;
- протокол розпізнавання адрес (ARP): ARP використовується для зіставлення IP-адреси пристрою з його фізичною MAC-адресою в мережі;
- система доменних імен (DNS): DNS — це протокол, який перетворює зрозумілі людині доменні імена (наприклад, www.example.com) в IP-адреси, зрозумілі комп’ютерам. [11]

TCP (Transmission Control Protocol) є протоколом транспортного рівня, який забезпечує надійну та впорядковану доставку даних між клієнтом та сервером. Він обробляє сегментацію, керування потоком, встановлення з'єднання та інші аспекти передачі даних. UDP (User Datagram Protocol) також є протоколом транспортного рівня, але на відміну від TCP він не забезпечує надійну доставку даних або встановлення з'єднання. UDP є більш легковаговимі швидшим, тому він часто використовується для передачі потокових даних, відео та аудіо, а також у ситуаціях, де невелика втрата даних не критична. [11] У програмі багатопотокового чату надійність і порядок доставки даних є важливими для підтримки цілісності сеансу чату. Кожне повідомлення, надіслане клієнтом, має бути повністю отримано сервером, а потім переслано всім іншим клієнтам у чаті. TCP допомагає гарантувати, що повідомлення не буде втрачено чи пошкоджено, і що вони доставлені в тому порядку, в якому вони були надіслані. Отже, порівняно з UDP, TCP є більш конвенціональним для використанні у програмах багатопотокових чатів, так як він забезпечує цілісність інформації та гарантію успішного надсилання пакетів, що є більш пріоритетним за швидкістю надсилання (як, наприклад, на стримінгових сервісах). Він забезпечує доставку даних одержувачу в тому ж порядку, в якому вони були

надіслані, і без помилок. TCP забезпечує контроль потоку, контроль перевантаження та механізми виявлення помилок, які забезпечують ефективну та надійну передачу даних у мережі.

1.4 Сокети та їх реалізація в Java

Сокети — це рівень абстракції (концептуальна межа або інтерфейс, який відділяє деталі реалізації конкретного компонента або системи від її користувачів або клієнтів), який забезпечує міжпроцесний зв'язок (IPC – inter-process communication) і мережевий зв'язок між різними об'єктами в розподіленому обчислювальному середовищі. Вони служать кінцевими точками для двонаправленого потоку даних, дозволяючи програмам обмінюватися інформацією через мережу, незалежно від задіяного апаратного забезпечення, операційних систем або протоколів. Він функціонує як рівень абстракції програмного забезпечення, який інкапсулює тонкощі мережевого стеку та забезпечує програмований інтерфейс для додатків, які використовують складні шаблони зв'язку. Коли сокет створюється, він представляє кінцеву точку, через яку дані можуть передаватися або отримуватись у мережевому середовищі. Він містить різні компоненти, зокрема IP-адресу, тип протоколу, номер порту, параметри сокета та інформацію про стан, які разом визначають його поведінку та характеристики. У контексті сокетів IP-адреса вказує мережеве розташування конкретної машини. [11]

Існує два основних види сокетів:

- потокові сокети: цей тип сокетів забезпечує надійне потокове з'єднання між двома процесами, подібне до телефонного дзвінка. Він забезпечує надійний, послідовний і недуплікований потік даних без меж записів;

- сокети дейтаграм: цей тип сокетів забезпечує ненадійний канал зв’язку без з’єднання між двома процесами, схожий на поштову пошту. Він забезпечує пакетно-орієнтоване транспортування без з’єднання, яке не гарантовано прибуде до місця призначення або в тому ж порядку, у якому було надіслано. [12]

Зв'язок через сокет – це процес обміну даними між клієнтом і сервером за допомогою сокетів. Сокети — це кінцеві точки двостороннього зв’язку між двома програмами, запущеними в мережі. Клієнтська та серверна програми створюють сокети та використовують їх для зв’язку одна з одною. Зв'язок через сокет включає кілька етапів:

- створення сокетів: як клієнтська, так і серверна програми створюють сокети для встановлення з’єднання. Клієнтський сокет використовується для підключення до серверного сокета;
- прив’язка сокетів: серверна програма прив’язує сокет до певного номера порту на головній машині;
- прослуховування з’єднань: серверна програма прослуховує вхідні з’єднання на прив’язаному сокеті;
- прийняття підключень: коли клієнт підключається до сокета сервера, серверна програма приймає з’єднання та створює новий сокет для зв’язку;
- надсилання та отримання даних: клієнтські та серверні програми можуть надсилати та отримувати дані через сокети за допомогою операцій читання та запису;
- закриття сокетів: коли зв’язок завершено, сокети закриваються, щоб звільнити ресурси. [18]

У TCP/IP зв’язок через сокет передбачає встановлення з’єднання між двома кінцевими точками через мережу. Зв'язок заснований на моделі клієнт-сервер, де серверний сокет прослуховує вхідні з'єднання, а клієнтський сокет ініціює запит на з'єднання.

Щоб встановити з’єднання через сокет, клієнтський сокет надсилає запит на з’єднання сокету сервера, який приймає з’єднання та створює новий сокет для зв’язку з клієнтом. Після встановлення з’єднання дані можуть передаватися між двома сокетами за допомогою методів send() і receive().

Зв’язок через сокет у TCP/IP можна використовувати для різноманітних цілей, зокрема для надсилання та отримання повідомлень, передачі файлів і потокової передачі аудіо- та відеоданих у реальному часі. Це фундаментальний будівельний блок для багатьох мережевих програм і широко використовується у веб-програмах, клієнтах електронної пошти та програмах обміну миттєвими повідомленнями.

РОЗДІЛ 2. ПОРІВНЯННЯ ВИКОРИСТАННЯ JAVA З ІНШИМИ ПЛАТФОРМАМИ РЕАЛІЗАЦІЇ ПРОГРАМНИХ ЗАСОБІВ

2.1 Загальний огляд мов програмування, які підходять для реалізації багатопотоковості

Наразі створюються нові мови програмування та вдосконалюються старі для того, щоб покращити досвід користування потоками та паралелізмом. Виявлення мов програмування, які мають найкращу ефективність під час виконання багатопотокових процесів, є важливим.

Оскільки програма, яка розглядається в даному проекті, заснована на багатопотоковості, ми будемо розглядати наступні мови: Java, Python, Go, а також C++ та C#.

2.1.1 Особливості реалізації багатопотоковості в Java

Java: це об’єктно-орієнтована мова програмування з вбудованою підтримкою паралелізму. Вона включає в себе багатий набір функцій, що робить її придатною для розробки широкого спектру програм, від програмного забезпечення для настільних комп’ютерів до веб-розробки, мобільних програм, корпоративних систем тощо. Крім того, Java підтримує автоматичне керування пам’яттю завдяки очищення пам’яті, що полегшує роботу з багатопотоковими програмами, оскільки опціональне використання обмежених процесом ресурсів для потоку є важливим.

Але треба відмітити, що багатопотоковість у Java також створює проблеми, такі як синхронізація потоків, умови змагання та взаємоблокування. Розробнику необхідно використовувати належні

механізми синхронізації, такі як блокування, семафори та монітори, щоб забезпечити безпеку потоків і запобігти проблемам одночасного доступу.

Можливості багатопотоковості Java в основному побудовані на Java Thread API, який дозволяє розробникам створювати потоки та керувати ними. Клас Thread в Java надає методи для створення, запуску, призупинення, відновлення та зупинки потоків. Розширюючи клас Thread або реалізовуючи інтерфейс Runnable, розробники можуть визначати код, який виконуватиметься одночасно в окремих потоках.

Реалізація багатопотоковості в Java включає в себе застосування:

- механізмів блокування та синхронізації;
- транзакційної пам'яті – концепції, яка забезпечує абстракцію вищого рівня для керування спільною пам'яттю в паралельних системах. Вона спрямована на спрощення реалізації паралельного програмування. [9]

2.1.2 Багатопотокова модель в Python

Python — це універсальна мова програмування високого рівня, відома своєю простотою, читабельністю та великою стандартною бібліотекою. Python забезпечує вбудовану підтримку багатопотоковості через свій модуль потоків. Він забезпечує комфортний інтерфейс для створення потоків і керування ними.

Важливо зазначити, що багатопотокова модель Python має певні обмеження через глобальне блокування інтерпретатора (Global Interpreter Lock – GIL), яке впливає на продуктивність багатопотокових програм, пов'язаних з ЦП. GIL гарантує, що лише один потік одночасно виконує байт-код Python, ефективно серіалізуючи виконання потоків. Це означає, що пов'язані з ЦП завдання не отримують повної вигоди від багатопотоковості в Python, оскільки лише один потік може виконувати інструкції Python у будь-

який момент часу. Як наслідок, багатопотокові програми, пов’язані з процесором, можуть не досягти значного покращення продуктивності порівняно з однопотоковими аналогами. Через GIL багатопотоковість Python не підходить для пов’язаних із ЦП завдань, які потребують справжнього паралелізму між кількома ядрами ЦП. Щоб досягти паралелізму для завдань, пов’язаних із процесором, розробники часто покладаються на багатопроцесорність, яка включає кілька процесів Python замість потоків. [18]

2.1.3 Реалізація багатопотоковості в C++ та C#

C# — це сучасна об’єктно-орієнтована мова програмування, розроблена Microsoft як частина .NET framework. Він створений як універсальна мова, придатна для широкого спектру програм, включаючи веб-розробку, програмне забезпечення для настільних ПК, розробку ігор тощо. C# поєднує потужність C++ із простотою мов високого рівня, надаючи багатий набір функцій для ефективного та виразного програмування.

C# забезпечує високорівневі абстракції для багатопотоковості, полегшуючи написання одночасного та паралельного коду. Паралельна бібліотека завдань (TPL) у C# пропонує такі конструкції, як завдання, паралельні цикли та паралельний LINQ (PLINQ), які абстрагуються від багатьох складнощів багатопотоковості. C# пропонує ряд примітивів синхронізації, таких як блокування, монітори та одночасні колекції, щоб забезпечити безпеку потоків і координувати доступ до спільних ресурсів.

Незважаючи на те, що C# надає абстракції для багатопотоковості, ці абстракції можуть призвести до певних витрат на продуктивність порівняно з мовами нижчого рівня, такими як C++.

C++ — це потужна та універсальна мова програмування, яка поєднує парадигми процедурного та об’єктно-орієнтованого програмування. Це

розширення мови програмування C з додатковими функціями та можливостями. C++ забезпечує повну підтримку низькорівневого програмування, а також пропонує високорівневі абстракції, що робить його придатним для широкого діапазону програм, включаючи системне програмування, розробку ігор, вбудовані системи тощо.

У багатопотоковості C++ забезпечує прямий контроль над керуванням пам'яттю та примітивами синхронізації потоків, дозволяючи розробникам точно налаштовувати багатопотокові програми для продуктивності та ефективності. Вона пропонує такі функції, як м'ютекси (класи, які створені для сигналізування у тому випадку, коли критичні розділи коду потребують ексклюзивного доступу, запобігаючи одночасному виконанню інших потоків із таким самим захистом і доступу до тих самих місць пам'яті), змінні умови та атомарні операції (призначені для надання доступу до спільних даних без додаткового захисту) для керування синхронізацією та забезпечення безпеки потоків. C++ дозволяє точно контролювати системні ресурси, що робить його придатним для розробки високопродуктивних багатопотокових програм. Він забезпечує ефективне керування пам'яттю та низькорівневий доступ, дозволяючи розробникам оптимізувати код для максимальної пропускної здатності та мінімальних втрат.

Однак багатопотокове програмування на C++ може бути складним і схильним до помилок. Вирішення проблем паралелізму, таких як умови змагань, взаємоблокування та невідповідності даних, вимагає ретельного проектування та реалізації. Низькорівневий характер C++ дає розробникам більший контроль, але також підвищує ризик появи незначних помилок і проблем із синхронізацією. Ця мова не має вбудованих абстракцій високого рівня для керування паралелізмом, таких як потокобезпечні колекції або конструкції вищого рівня для паралельного програмування. Розробники часто покладаються на зовнішні бібліотеки або спеціальні реалізації для таких абстракцій, що може вимагати додаткових зусиль і досвіду. Окрім того, код C++, написаний для багатопотоковості, може бути менш переносимим на

різні платформи та операційні системи. Примітиви синхронізації та керування потоками можуть відрізнятися між системами, вимагаючи специфічного для платформи коду або ретельної абстракції для забезпечення сумісності. [19]

Важливо відзначити, що C++ і C# можуть досягти ефективної та надійної багатопотоковості, але вони відрізняються підходами та рівнем контролю, який вони забезпечують. C++ перевершує низькорівневе керування та оптимізацію продуктивності, тоді як C# пропонує абстракції та спрощення більш високого рівня, що полегшує написання одночасного коду зі зниженим ризиком помилок. Вибір між C++ і C# для багатопотокового програмування залежить від таких факторів, як вимоги програми, міркування продуктивності та переваги розробника.

2.1.4 Реалізація багатопотоковості в Go

Розроблена Google Go поєднує в собі простоту програмування мов з динамічною типізацією та продуктивність і безпеку мов зі статичною типізацією.

У Go багатопотоковість і паралелізм є фундаментальними особливостями мови, які підтримуються за допомогою програм і каналів. Підхід Go до багатопотоковості відрізняється від традиційних потокових моделей у таких мовах, як Java або C++. Замість явного керування потоками, Go представляє goroutines як більш легку та ефективну альтернативу. Goroutines керуються середовищем виконання Go, яке динамічно розміщує їх у кількох потоках операційної системи, дозволяючи паралельне виконання коду. Go містить потужний механізм зв’язку під назвою «канали» для обміну даними та синхронізації підпрограмм. Канали забезпечують безпечну та ефективну передачу даних між підпрограмами та сприяють координації та синхронізації. Канали спрощують розробку та реалізацію одночасного коду,

усуваючи багато питань синхронізації низького рівня. [21] Go надає вбудовані примітиви синхронізації, як-от м’ютекси та змінні умови, для більш точного контролю над спільними ресурсами. Ці примітиви дозволяють розробникам захищати важливі розділи коду та явно керувати синхронізацією, коли це необхідно.

Незважаючи на те, що goroutines легкі порівняно з традиційними потоками, вони все одно потребують деяких системних ресурсів. Створення та керування великою кількістю підпрограм може призвести до збільшення використання пам’яті та накладних витрат. Коли декільком підпрограмам потрібно отримати доступ до спільних ресурсів, для забезпечення цілісності даних і запобігання конкуренції необхідно використовувати механізми синхронізації, такі як блокування або канали. Однак використання цих механізмів може створити додаткові накладні витрати та потенційно вплинути на продуктивність. [20] Підсумовуючи, можна сказати, що Go має такі обмеження, як відсутність загальних програм, обмежена підтримка розподілених обчислень і вплив збирання сміття на програми реального часу.

2.2 Порівняння швидкодії мов

У паралельному програмуванні ефективність визначається як обсяг роботи, який необхідно виконати, тоді як «продуктивність» вказує на те, наскільки швидко алгоритм може завершити певну роботу. Швидше впровадження не обов’язково є ефективним, де ефективність вказує на повне використання доступних апаратних ресурсів. У програмі, що виконується в паралельній системі, можливо таке, що деякі інструкції потрібно виконувати послідовно. Це послідовне виконання має обмежуючий фактор для прискорення програми, тому навіть додавання більшої кількості процесорів може не призвести до пришвидшення програми. Наприклад, якщо програмі потрібно 20 хвилин, щоб завершити використання послідовного коду з одним

потоком, і якщо 5-хвилинну частину коду неможливо зробити паралельною, решту 15 хвилин обробки можна записати як паралельний код. У таких ситуаціях, незалежно від того, скільки потоків присвячено розпаралеленому виконанню цієї програми, мінімальний час виконання не може бути менше 15 хвилин.

Одним із найважливіших критеріїв паралельних обчислень є визначення того, наскільки швидше працює паралельний алгоритм порівняно з найкращим послідовним. Цей захід відомий як «прискорення». Іншими словами, прискорення — це збільшення швидкості, досягнуте паралельним виконанням порівняно з послідовним виконанням. Будь-яка програма, яка забезпечує більшу швидкість, не обов'язково ефективна.

Існують наступні критерії для порівняння швидкодії названих у попередньому розділі мов:

- Множення матриць: це важлива, але проста математична операція в лінійній алгебрі. В інформатиці множення матриць представляє великий інтерес для багатьох прикладних областей. Класичний алгоритм множення матриць (CMMA) можна легко реалізувати будь-якою мовою програмування. CMMA працює в O(N^3), де N є розмірністю матриці;
- Quicksort: це алгоритм сортування, який працює наступним чином: спочатку масив розбивається на два підмасиви, нижчі елементи та вищі елементи відповідно. Потім він рекурсивно сортує ці підмасиви. Він має середню складність сортування O(n log(n)), але за рідкісних обставин знижується до O(n^2);
- Гра «Життя» Конвея: Варіант клітинного автомату, який запропонував Джон Хортон Конвей, математик із коледжу Гонвіля та Каюса Кембриджського університету, у 60-х роках. Правила переходу: Правило 1, Виживання: якщо жива клітина має двох або трьох живих сусідів, вона виживає. Правило 2, Смерть: якщо жива клітина має менше двох або більше трьох живих сусідів, вона гине. Правило 3,

Народження: якщо мертва клітина має рівно трьох живих сусідів, вона народжується.

Базуючись на критеріях, названих вище, можемо зробити наступні висновки:

- За критерієм множення матриць:

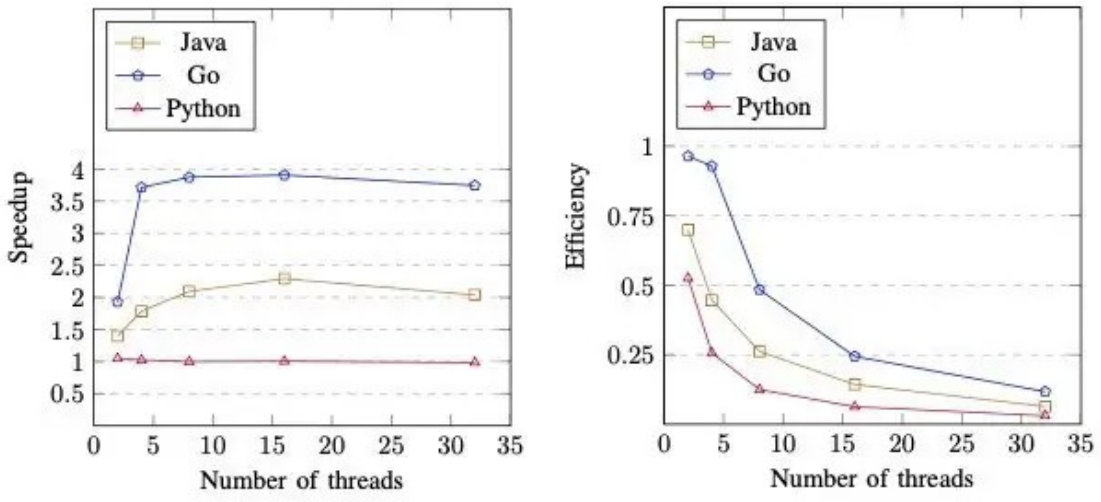

Рис. 2.1. Порівняння мов за критерієм множення матриць [15]

На рис. 2.1. графік зліва зображує ефективність потоків у швидкодійності, а графік справа демонструє їх продуктивність в ефективності. Як ми бачимо, зі збільшенням потоків ефективність знижується, але швидкодія залишається на тому ж рівні. Java є найкращою в послідовному виконанні, тоді як Go виконала на 45% більше часу в послідовному виконанні. Найгірша продуктивність у Python з різницею у 30%;

- За критерієм Quicksort:

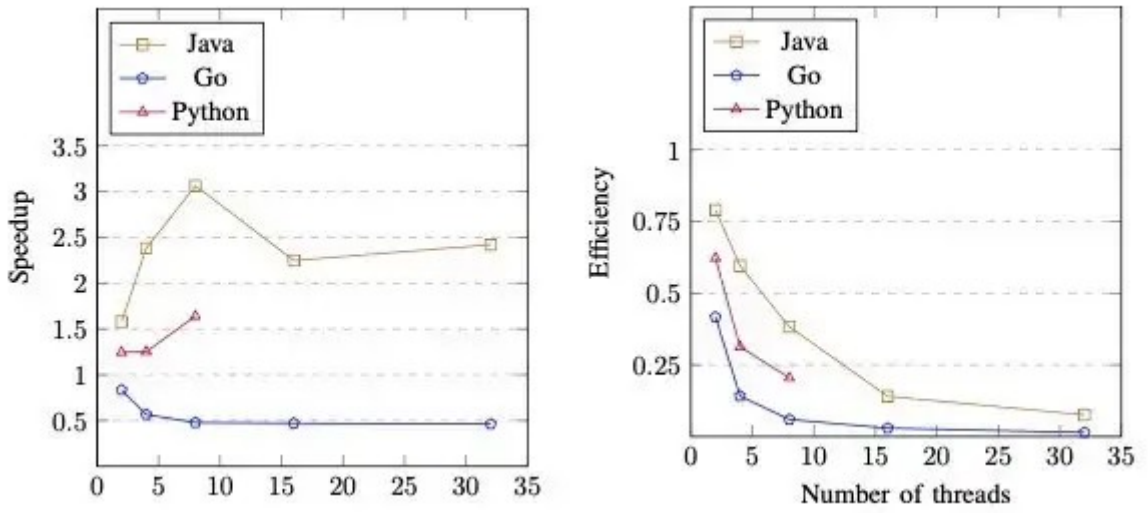

Рис. 2.2. Порівняння мов за критерієм Quicksort [15]

За цим показником Java має найкращі багатопотокові результати, хоча ми бачимо, що її ефективність починає спадати на більш ніж 15 потоках. Загальні показники реалізації Python є дуже поганими. Go має цікаві результати, оскільки багатопотокові запуски змінюються від поганого до найгіршого (рис. 2.2.).

- За критерієм гри «Життя» Конвея:

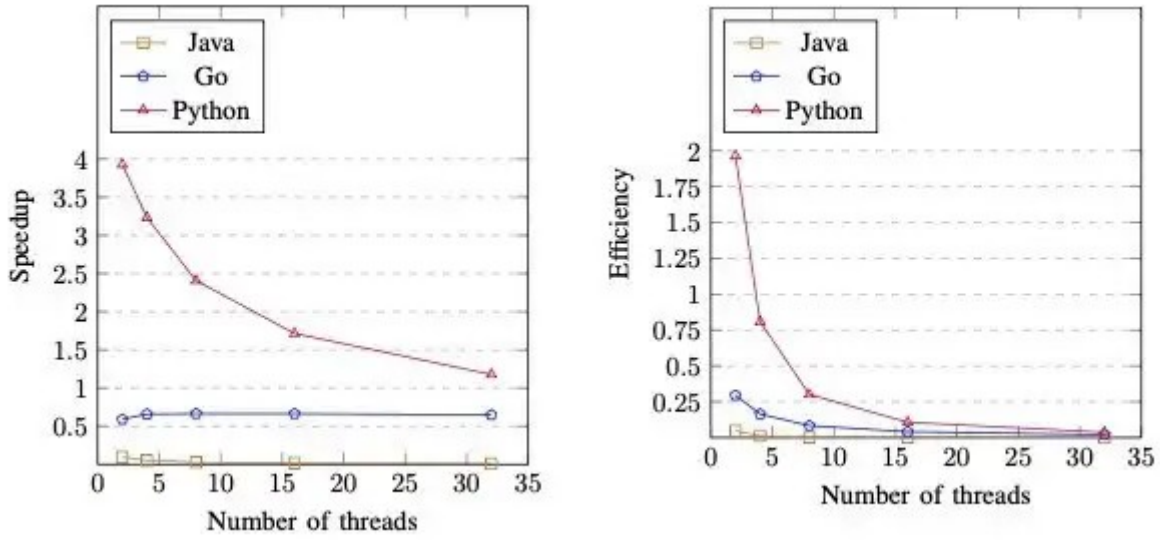

Рис. 2.3. Порівняння мов за критерієм гри «Життя» Конвея [15]

Python показує найкращі результати. Ефективність Java дуже близька до нуля, і існує незначна різниця між Go та Java (рис. 2.3.).

Стандартні бібліотеки Python для обробки паралелізму не такі хороші, як бібліотеки, реалізовані в інших мовах програмування, як Java та Go. Важко сказати, що краще між Java і Go, оскільки Java перевершує Go у тесті множення матриць, але Go перевершує Java в експерименті швидкого сортування. [15] Але задачі реалізаціїї багатопотокового чату серед цих трьох мов найбільш підходящою можна вважати Java.

Щодо C# та C++, для них існують два поширених типи багатопотокових проблем, які може бути важко виявити. Умови гонки виникають, коли поведінка програми залежить від послідовності або часу неконтрольованих подій – цей тип помилки може призвести до збоїв або пошкодження даних в пам’яті. Другою проблемою є взаємоблокування. Воно виникає, коли кілька потоків заблоковано під час конкуренції за ресурси.

Один потік застряг в очікуванні другого потоку, який застряг в очікуванні першого. Цей тип помилки може спричинити зависання програм.

Як висновок, ми можемо сказати, що найбільш оптимальною мовою для реалізації багатопотоковості багатокористувацького чату на основі сокетів є мова програмування Java, оскільки ця мова забезпечує вбудовану підтримку на рівні мови для багатопотоковості через клас java.lang.Thread і пов’язані API. Це полегшує створення та керування потоками в програмах Java.

РОЗДІЛ 3. РОЗРОБКА АРХІТЕКТУРИ ПРОЕКТУ ТА ЙОГО РЕАЛІЗАЦІЯ

3.1. Структура програми та блок-схеми класу сервера та клієнта

В розділі представлені: загальна блок-схема, які описує структуру та логіку програми; опис методів (циклів, класів, засобів), якими програма буде реалізована. Наведені пояснення, чому саме такий метод був обраний.

В даній програмі реалізовані класи серверу та клієнту, які “спілкуються” один з одним. Для прикладу був реалізований алгоритм зчитування сервером кількості надісланих від клієнту слів та надсилання відповіді у вигляді фрази. (Одночасно в чаті може знаходитися велика кількість клієнтів, кожен з яких може взаємодіяти з сервером). Після закінчення роботи з сервером клієнт виходить з чату, про що сервер також повідомляє йому.

Структура програми: У відповідь на кожного клієнта сервер створює новий потік і дозволяє всі читання та запис цього клієнта через цей потік.

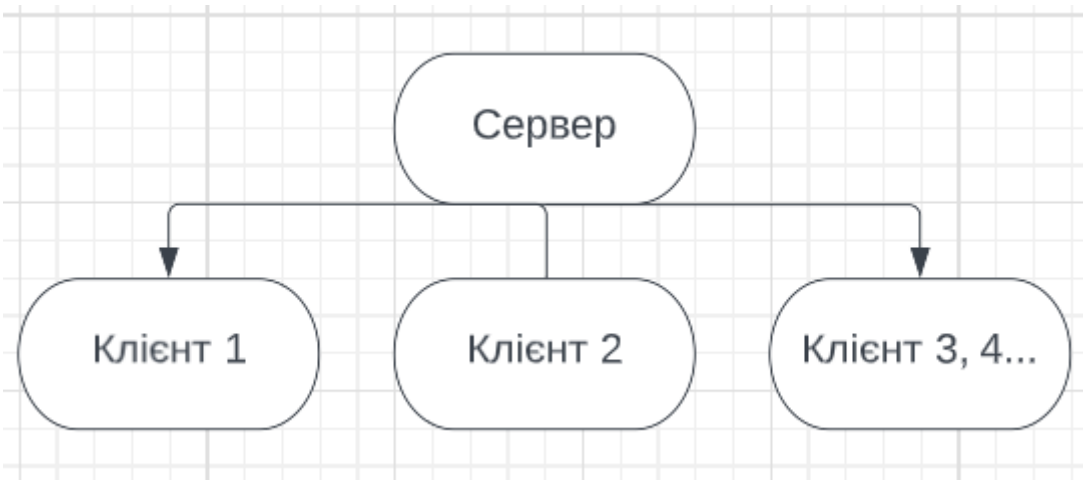

Рис. 3.1. Загальна структура програми та реалізація багатопотоковості

На рис. 3.1. зображений принцип роботи багатопотоковості: сервер одночасно працює з великою кількістю клієнтів.

Програмне рішення надано з сокетами та потоками введення-виведення, доступними в Java. Сокети - це складні структури даних. У цьому рішенні також використовуються вхідні та вихідні потоки, що створює можливість пересилання та отримання даних у двосторонніх з'єднаннях сокетів. Інтернет-адреса, яка є номером порту, необхідна для встановлення з'єднання з Інтернетом за допомогою сокетів.

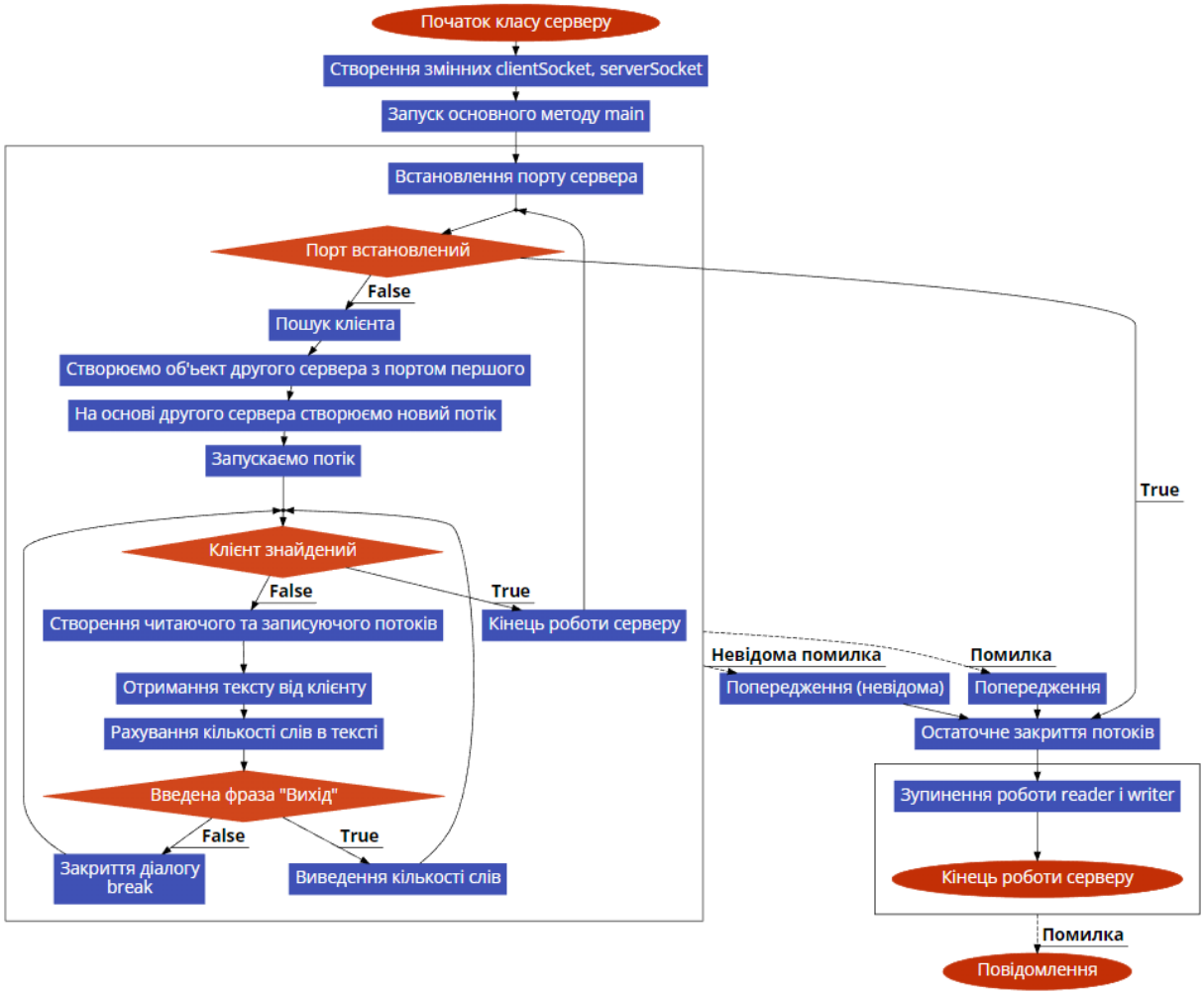

Рис. 3.2. Блок-схема структури сервера у файлі Server.java

На рис. 3.2 наведена блок-схема структури класу серверу. Сервер займає порт, після чого створює сокет для зв'язку з клієнтом і переводить цей сокет у режим очікування підключення. Після цього сервер створює два

потоки введення та виведення для спілкування з клієнтом: читає та записує. Він зчитує надісланий клієнтом текст, перевіряє, чи клієнт написав фразу "Вихід". Якщо прочитав, він зупиняє сокет і чекає підключення знову., Якщо ні, тоді відправляє йому відповідь (фразу "Ви написали n слів"). У разі виникнення ще одного підключення сервер створює та запускає ще один потік класса Thread і передає у його конструктор об’єкт іншого класа

SecondServer, який реалізує інтерфейс Runnable.

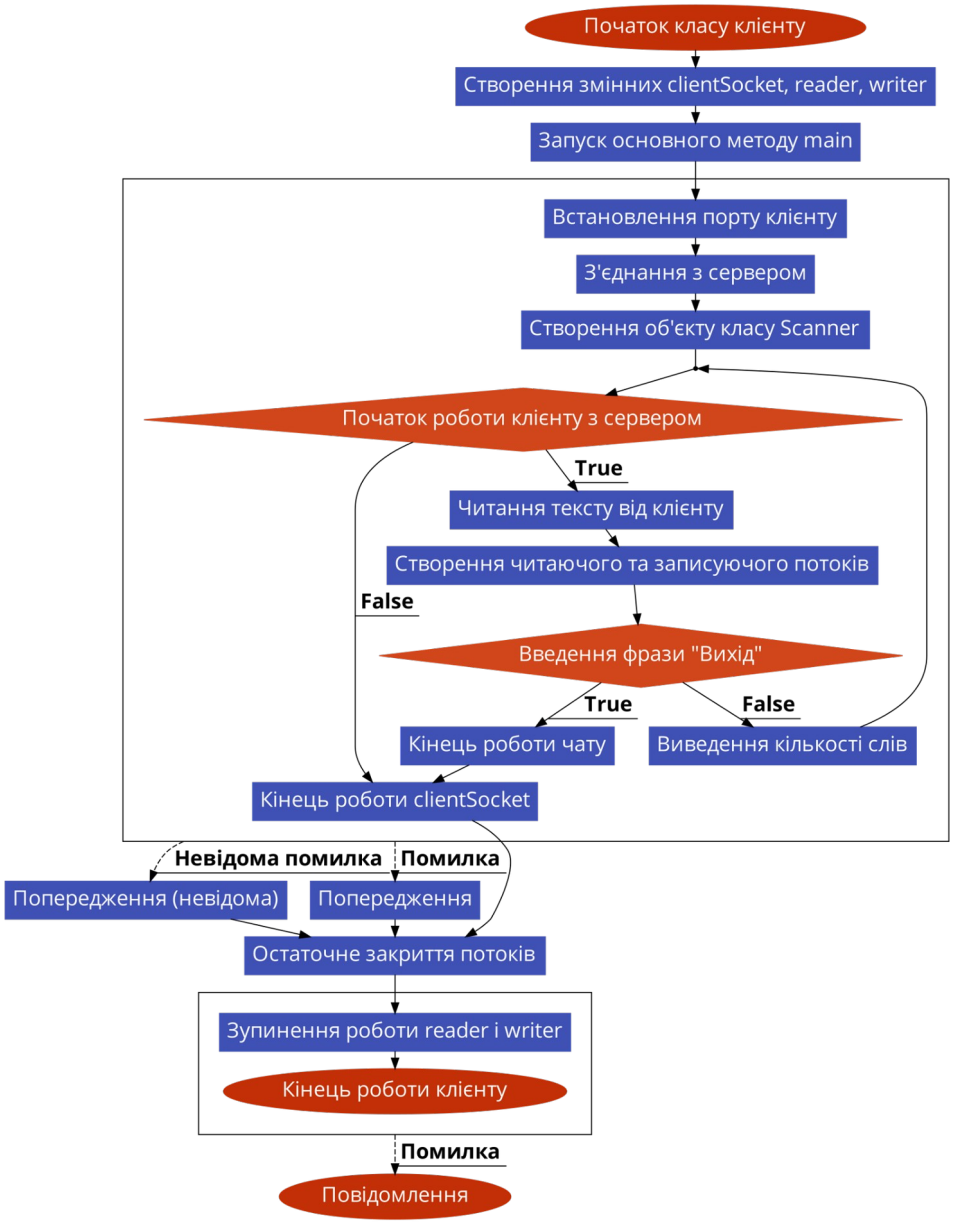

Рис 3.3. Блок-схема структури типового файлу клієнта ClientManagement.java

Клієнти реалізовані як типові файли, інструкція для виконання яких міститься в файлі ClientManagement.java. З боку клієнта (рис. 3.3) після ініціації клієнтського процесу відбувається створення потоку між сокетом клієнта та сокетом сервера передачі даних (у разі тексту). Клієнт надсилає повідомлення серверу, отримує відповідь у вигляді повідомлення про кількість слів у повідомленні, цей цикл може повторюватися нескінченну кількість разів. Закінчити взаємодію з сервером можливо введенням фрази "Вихід", після чого відбувається закриття потоків та закінчення сесії.

3.2. Реалізація сокетів

Об'єкти Sockets використовуються для створення з'єднання між клієнтом і сервером. TCP-сокети — це сокети, орієнтовані на з’єднання, і нашій програмі чату також потрібна парадигма зв’язку, орієнтована на з’єднання, тому ми використовуємо об’єкти Socket і Server Socket для зв’язку. Процедура встановлення з’єднання складається з наступних кроків:

- Створення мережевого підключення;
- Відкриття сокетів;
- Створення вхідних/вихідних потоків;
- Закриття сокетів.

```
clientSocket = new Socket("localhost",8000);
System.out.println(name+", почніть писати...");

reader = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
writer = new BufferedWriter(new OutputStreamWriter(clientSocket.getOutputStream()));
```

Рис. 3.4. Створення клієнтського сокета у файлі ClientManagement.java

Процес створення сокета та клієнта є ідентичним; він наведений на рис. 3.4 на прикладі клієнтського сокета. Серверний сокет призначений лише для прослуховування запитів від сервера на виділеному номері порту. Сервер продовжить прослуховувати запити від клієнта на цьому номері порту. Після

прийняття запиту сервер створює інший сокет для встановлення повного з’єднання.

```
private static void stopClient(){
    try {
        reader.close();
        writer.close();
        clientSocket.close();
    } catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}
```

Рис. 3.6. Закриття сокетів та потоків для клієнта та сервера

Як зображено на рис. 3.6., для припинення роботи клієнта та сервера є необхідність в закритті потоків читання та записування, а також сокету.

РОЗДІЛ 4. ТЕСТУВАННЯ ПРОГРАМИ ТА ПРЕДСТАВЛЕННЯ ЇЇ ГРАФІЧНОГО ІНТЕРФЕЙСУ

4.1. Визначення вимог для тестування програмного продукту

В програмному забезпеченні повинні бути реалізовані наступні функції:

- запуск сервера та клієнта;
- стабільне з’єднання між клієнтом та сервером;
- базовий функціонал спілкування між клієнтами через сервер;
- реалізований графічний інтерфейс для клієнта-користувача.

Програмне забезпечення “Багатопотоковий чат” було протестовано на тестовій платформі ноутбука MSI GV62 8RD з наступними характеристиками:

- накопичувачі даних: NVMe Samsung SSD 970; WDC WD10SPZX-17Z10T0;
- процесор: Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz;
- відеокарта: NVIDIA GeForce GTX 1050 Ti.

4.2. Тестування програми

Для перевірки першого пункту технічного завдання необхідно запустити файли сервера та клієнта. Для ініціалізації роботи програми адміністратор запускає файл серверу Server.java, а клієнт під’єднується через ClientManagement.java.

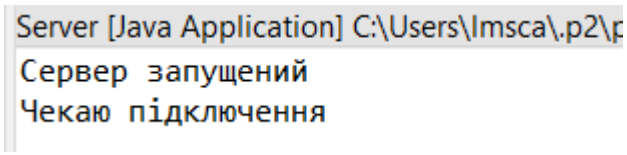

Рис. 4.1. Запуск сервера

В консолі адміністратор повинен побачити фразу “сервер запущений”, як наведено на рис. 4.1. Це свідчить про те, що сервер працює стабільно та очікує на під’єднання клієнтів.

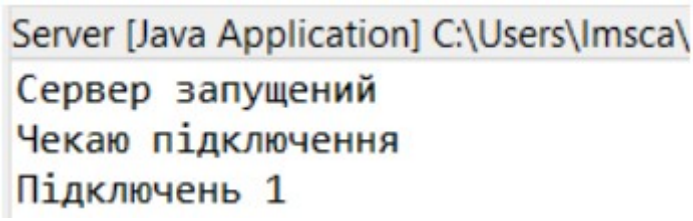

Рис. 4.2. Успішно приєднаний до сервера перший клієнт

Коли до серверу підключається новий клієнт, адміністратор бачить повідомлення на консолі (рис. 4.2). З’єднання успішно встановлено, а користувач може почати спілкуватися, коли приєднуються інші користувачі.

Для перевірки реалізації багатопотоковості в програмі необхідно підключення як мінімум двох клієнтів.

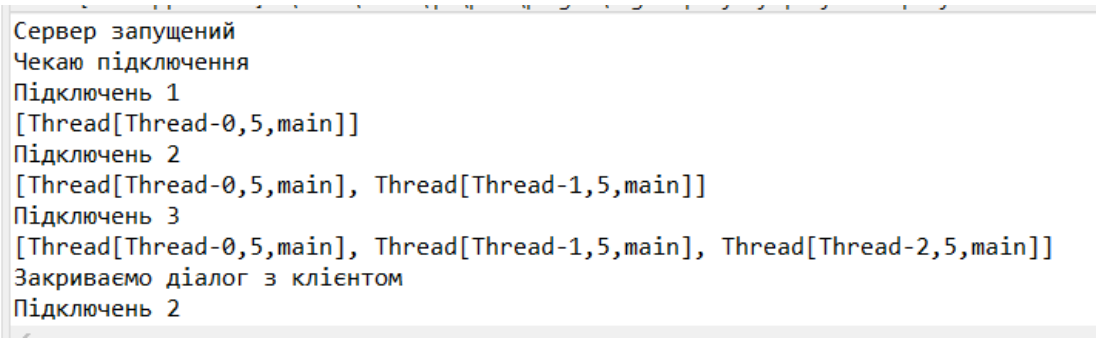

Рис. 4.3. Підключення декількох клієнтів до серверу

На рис. 4.3 видно, що до серверу приєдналися 3 клієнта – тобто він здатний обслуговувати декількох клієнтів одночасно без переривання.

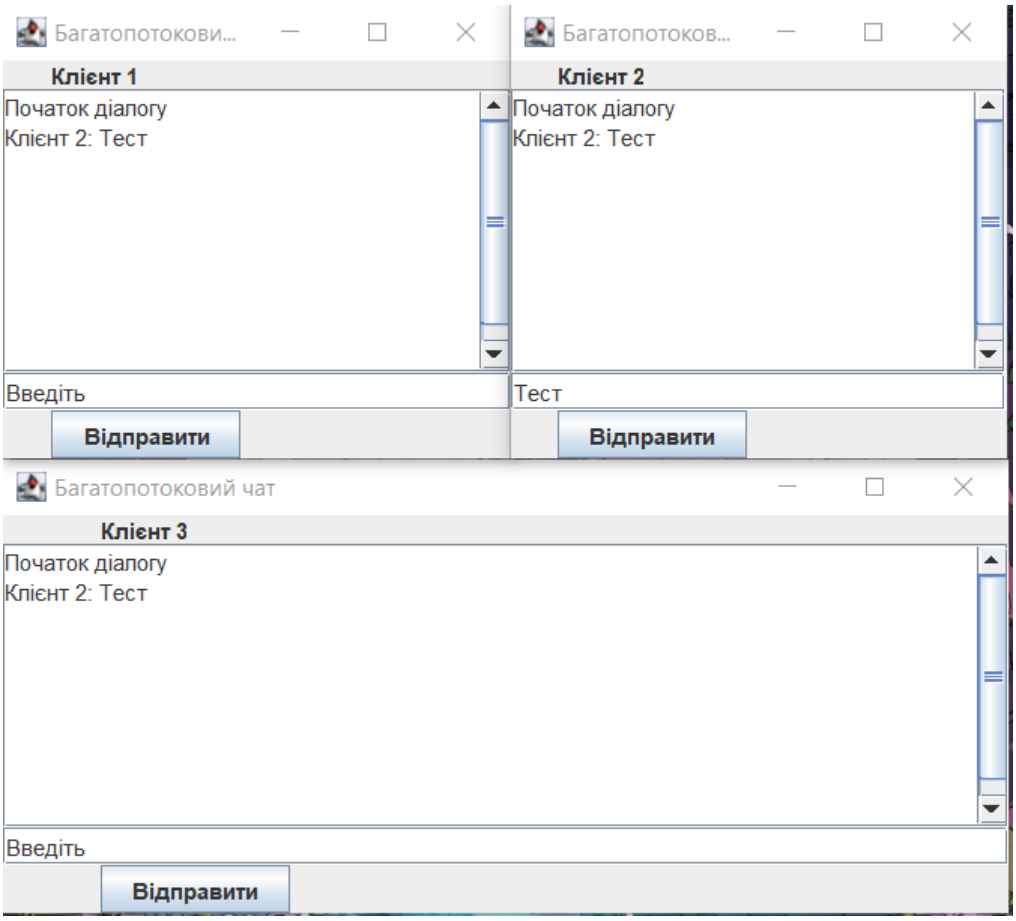

Рис. 4.4. Надсилання повідомлення одного з клієнтів іншим через сервер

Для перевірки другого пункту один з клієнтів надсилає будь-яке повідомлення через текстове поле. Сервер повинен обробити це повідомлення та надіслати його усім іншим користувачам – це продемонстровано на рис. 4.4.

Клієнти можуть вільно спілкуватися між собою через графічний інтерфейс:

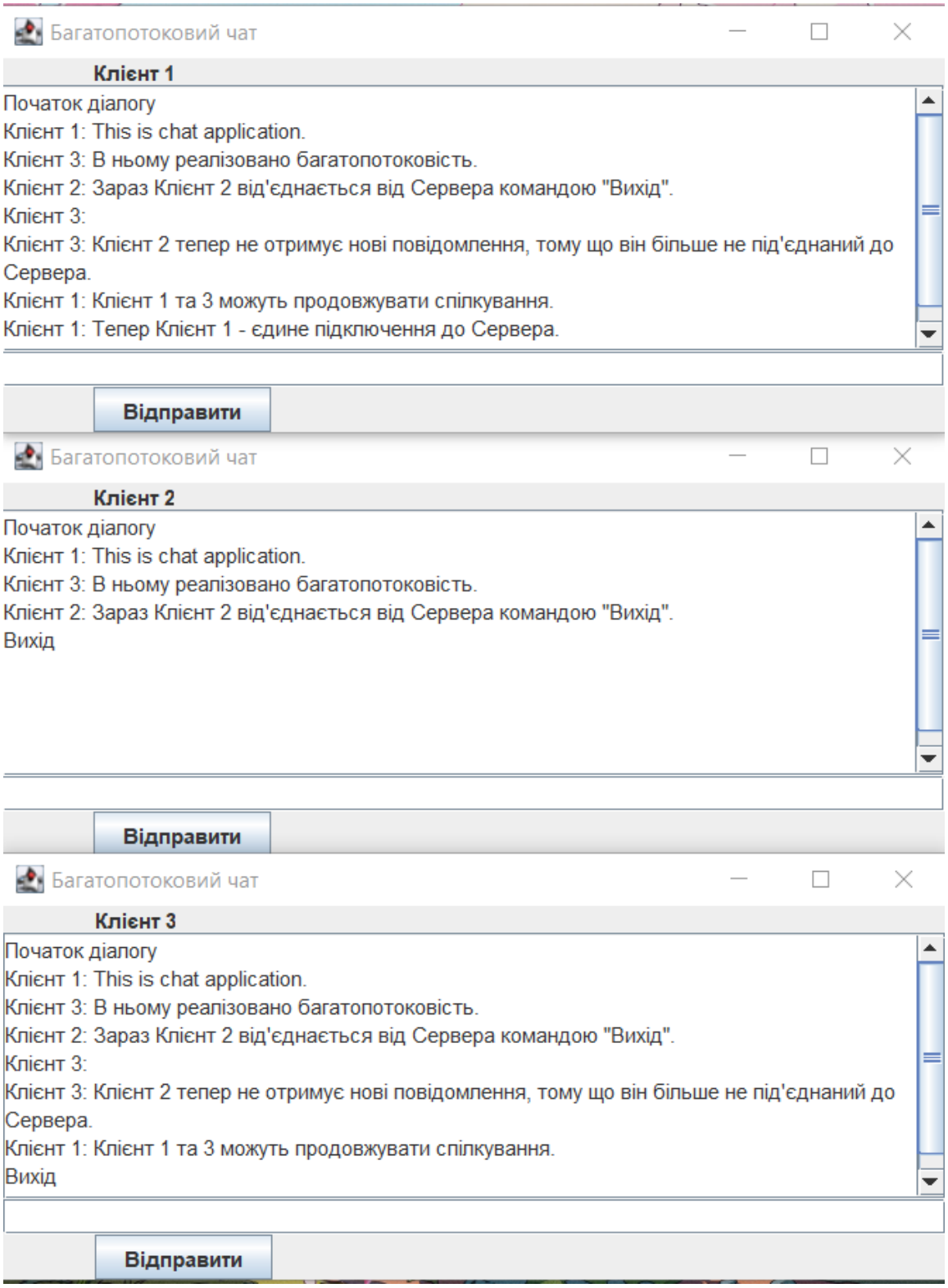

Рис. 4.5. Одночасне спілкування клієнтів

Окрім можливості спілкування користувачів, на рис 4.5. зображений базовий графічний для більшого QOE (quality of experience).

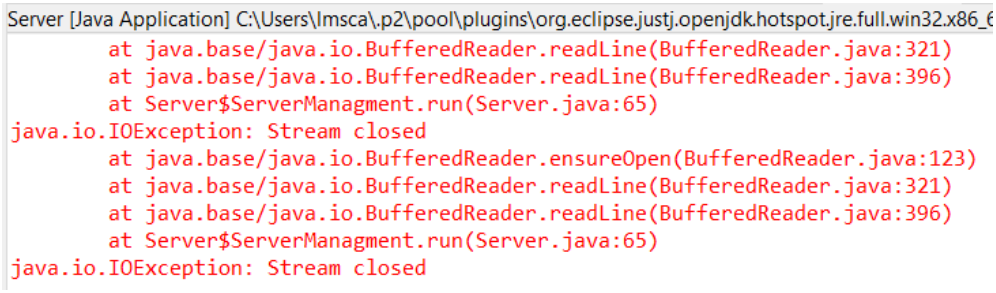

Рис. 4.6. Помилка при неправильному виході з програми

Під час тестування було виявлено помилку (рис. 4.6). Вийти з чату можна за допомогою фрази “Вихід”, але якщо користувач цього не зробив і закрив інтерфейс користувача через хрестик, виникає помилка, пов’язана з закриттям потоків.

Таблиця 4.1.

Результати тестування

№	Функція, зауваження або помилка	Відповідність технічному завданню (ТАК/НІ)
1	Запуск сервера та клієнта	ТАК
2	Стабільне з’єднання між клієнтом та сервером	ТАК
3	Базовий функціонал спілкування між клієнтами через сервер	ТАК
4	Реалізований графічний інтерфейс для клієнта-користувача	ТАК
5	Зауваження – повідомлення не відправляється кнопкою “Enter” на клавіатурі	-
6	Зауваження – текст вже надісланого повідомлення залишається в полі для набору тексту	-
7	Помилка – у разі закриття вікна клієнту хрестиком потоки не закриваються, що визиває навантаження центрального процесору	-

В табл. 4.1. наведені результати тестування програми. Як висновок, програма реалізована згідно з завданням, але її функціонал є не зовсім стабільним.

ВИСНОВКИ

В результаті виконання даної дипломної роботи була успішно реалізована багатопотокова клієнт-серверна програма чату з використанням мови програмування Java. Розроблений чат дозволяє користувачам взаємодіяти в режимі реального часу, обмінюючись повідомленнями через мережу.

Під час розробки програми було проведено дослідження та вивчено основні принципи мережевого програмування, багатопотоковості та протоколу TCP/IP. Було використано інструменти та бібліотеки мови Java, такі як Socket та ServerSocket, для забезпечення з'єднання та передачі даних між клієнтом та сервером.

У процесі розробки були враховані основні вимоги до чат-програм, зокрема забезпечення надійності передачі повідомлень, підтримки одночасного підключення декількох користувачів, реалізації простого та зручного інтерфейсу користувача. Було виконано тестування програми.

Отримані результати підтвердили ефективність та правильність реалізації програми чату. Програма працює стабільно, забезпечуючи швидку передачу повідомлень та задовольняючи основні потреби користувачів. Розроблений чат може бути використаний як основа для подальшого розширення та вдосконалення, наприклад, додавання нових функцій.

ДОДАТОК А. КЛАС СЕРВЕРУ

ДОДАТОК Б. КЛАС КЛІЄНТУ

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Інформаційні мережі: навчальний посібник / Ю. В. Коваль, А. Б. Ставровський. – Київ, 2021. – 84 с.
2. Brian Goetz. Java Concurrency in Practice / Brian Goetz, Tim Peierls., 2010. – 432 с.
3. Нікітенкова С. П. Багатопоточне програмування мовою JAVA: навчально-методичний посібник / С. П. Нікітенкова., 2015. – 90 с.
4. Scott Oaks. Java Threads: Understanding and Mastering Concurrent Programming / Scott Oaks, Henry Wong., 2004. – 362 с. – (3).
5. Каршенбойм Й. Стислий курс HDL. Додатки до розділу автоматів станів. Багатоядерність, багатозадачність, багатопоточність / Йосип Каршенбойм. // Компоненти та технології. – 2009. – №4.
6. Alex Berson. Client/server architecture / Alex Berson., 1996. – (2).
7. John Terra. What is Client-Server Architecture? Everything You Should Know [Електронний ресурс] / John Terra – Режим доступу до ресурсу: https://www.simplilearn.com/what-is-client-server-architecture-article#:~:text=The%20three-tier%20client-server,tier%20comprising%20the%20database%20server
8. Benjamin Erb. Server Architectures [Електронний ресурс] / Benjamin Erb. – 2018. – Режим доступу до ресурсу: https://berb.github.io/diploma-thesis/original/042_serverarch.html.
9. Опівалов С. А. Методи роботи з потоками в мові Java / С. А. Опівалов. // Міжнародний журнал гуманітарних та природничих наук. – 2023. – №4. – С. 79.
10. Elliotte Rusty Harold. Java Network Programming / Elliotte Rusty Harold., 2013. – 506 с.
11. Kevin Fall. TCP/IP Illustrated: The Protocols, Volume 1 (Addison-Wesley Professional Computing Series) / Kevin Fall, W. Stevens., 2011. – 1056 с.

12. Overview of Sockets [Електронний ресурс] – Режим доступу до ресурсу: https://docs.oracle.com/cd/E19683-01/816-5042/6mb7bck6b/index.html.
13. Основи програмування TCP-сокетів на Java [Електронний ресурс]. – 2020. – Режим доступу до ресурсу: https://medium.com/nuances-of-programming/main-tcp-java-594b6519d494.
14. Brian "Beej Jorgensen" Hall. Beej's Guide to Network Programming: Using Internet Sockets / Brian "Beej Jorgensen" Hall., 2019. – 180 с.
15. Jose Pablo. Java, Go, and Python: a multi-thread performance comparison [Електронний ресурс] / Jose Pablo. – 2022. – Режим доступу до ресурсу: https://levelup.gitconnected.com/java-go-and-python-a-multi-thread-performance-comparison-28e942cb73e6.
16. В'язовик Н.А. Програмування Java: навчальний посібник / В'язовик Н.А.., 2016. – 603 с.
17. Java Networking [Електронний ресурс] – Режим доступу до ресурсу: https://www.javatpoint.com/java-networking.
18. Jasleen Kaur Sondhi. Multithreading in Python [Електронний ресурс] / Jasleen Kaur Sondhi – Режим доступу до ресурсу: https://www.scaler.com/topics/multithreading-in-python/.
19. Multithreading in C++ [Електронний ресурс] – Режим доступу до ресурсу: https://www.tutorialride.com/cpp/multithreading-in-c.htm.
20. Manoj Debnath. Intro to Concurrency in Go [Електронний ресурс] / Manoj Debnath. – 2021. – Режим доступу до ресурсу: https://www.developer.com/languages/intro-to-concurrency-in-go/.
21. What are the benefits and drawbacks of using coroutines and channels for thread synchronization in Go? [Електронний ресурс] – Режим доступу до ресурсу: https://www.linkedin.com/advice/0/what-benefits-drawbacks-using-coroutines-channels.

22. Lesson: All About Sockets [Електронний ресурс] – Режим доступу до ресурсу: https://docs.oracle.com/javase/tutorial/networking/sockets/index.html.
23. Куліков С.С. Тестування програмного забезпечення. Базовий курс: практичний додаток — Мінськ: видавництво “Чотири четвірки”, 2015 — 294 с.