Хмельницький національний університет
Факультет інформаційних технологій
Кафедра комп'ютерної інженерії та інформаційних систем

КВАЛІФІКАЦІЙНА РОБОТА

бакалавр
Освітній рівень

Проміжне програмне забезпечення мультикомп'ютерних систем з топологією «Решітка»
Назва теми

КвРКІ 200109.20.01.07 ПЗ
Шифр

Галузь знань *12 «Інформаційні технології»*
Шифр, назва

Спеціальність *123 «Комп'ютерна інженерія»*
Шифр, назва

Освітня програма *«Комп'ютерна інженерія та програмування»*
Назва

Виконав: студент IV курсу, група КІ2-20-1 ______ Ю. В. Ільчишина
Підпис Ініціали, прізвище

Керівник ______ К. М. Березька
Підпис, дата Ініціали, прізвище

Нормоконтролер ______ І. О. Засорнова
Підпис, дата Ініціали, прізвище

До захисту допускаю:
Зав. кафедри комп'ютерної інженерії та інформаційних систем ______ Т. О. Говорущенко
Підпис Ініціали, прізвище

«12» червня 2024 р.

Хмельницький 2024

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет Інформаційних технологій

Кафедра Комп'ютерної інженерії та інформаційних систем

Освітній рівень бакалавр

Галузь знань 12 Інформаційні технології

Спеціальність 123 Комп'ютерна інженерія

Освітня програма «Комп'ютерна інженерія та програмування»

ЗАТВЕРДЖУЮ

Зав. кафедри Т.О.Говорущенко

"10" 01 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ БАКАЛАВРА

Ільчишиній Юлії Віталіївні

Прізвище, ім'я, по батькові студента

1. Тема проекту (роботи) Проміжне програмне забезпечення мультикомп'ютерних систем з топологією «Решітка»

Керівник проекту (роботи) Березька Катерина Миколаївна, к.т.н., доцент.

Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 15.02.2024 р. № 8

2. Строк подання студентом проекту (роботи) на кафедру 01.06.2024 р.

3. Вихідні дані до проекту (роботи) Завдання на кваліфікаційну роботу

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)

Теоретичне дослідження проміжного програмного забезпечення мультикомп'ютерних систем з топологією «Решітка»

Архітектура та функціональність мультикомп'ютерної системи

Практичне дослідження проміжного програмного забезпечення мультикомп'ютерних систем з топологією «Решітка

5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень)

Алгоритм роботи першого експерименту проміжного програмного забезпечення мультикомп'ютерної системи

Алгоритм роботи четвертого експерименту проміжного програмного забезпечення мультикомп'ютерної системи

Результати роботи

6. Консультанти розділів дипломного проекту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийн
Нормоконтроль	Засорнова І.О., доцент кафедри КІС		
Антиплагіат	Нічепорук А.О., доцент кафедри КІС		

7. Дата видачі завдання « 10 » 01 2024 р.

КАЛЕНДАРНИЙ ПЛАН

№з/п	Назва етапів (розділів) дипломного проекту (роботи)	Термін виконання етапів проекту (роботи)	Приміт
1	Вибір напряму дослідження та узгодження тематики кваліфікаційної роботи з керівником	10.01.2024	виконан
2	Ознайомлення з предметною областю; формулювання мети та задач дослідження;	10.02.2024	виконан
3	Робота над розділом 1 – дослідження основних елентів та постановка задачі	21.02.2024	виконан
4	Робота над розділом 2 – дослідження предметної області та вибір компонентів для реалізації мультикомп'ютерної системи з топологією «Решітка»	19.03.2024	виконан
5	Робота над розділом 3 – реалізація мультикомп'ютерної системи з топологією «Решітка» та проведення дослідів роботи	15.05.2024	виконан
6	Оформлення пояснювальної записки згідно вимог	28.05.2024	виконан
7	Попередній захист ВКР	30.05.2024	виконан
8	Захист ВКР на засіданні ЕК	Червень 2024 року	

Студент ______ Підпис ______ Ю. В. Ільчишина ______ Ініціали, прізвище

Керівник роботи ______ Підпис ______ К. М. Березька ______ Ініціали, прізвище

№ рядка	формат	Позначення	Найменування	Кіл. листів	№ екз	Примітка
			Текстові документи			
1		КвРКІ 200109.20.01.07 ПЗ	Пояснювальна записка	66		
			Графічні матеріали			
2		КвРКІ 200109.20.01.07 Е8	Алгоритм роботи першого експерименту проміжного програмного забезпечення мультикомп'ютерної системи	1		
3		КвРКІ 200109.20.01.07 Е8	Алгоритм роботи четвертого експерименту проміжного програмного забезпечення мультикомп'ютерної системи	1		
4		КвРКІ 200109.20.01.07 Е8	Результати роботи	1		

Зм	Арк	№ докум	Підпис	Дата	КвРКІ 200109.20.01.07 ВП		
Розробив		Ільчишина			Проміжне програмне забезпечення мультикомп'ютерних систем з топологією «Решітка»	Літера	Аркуш	Аркушів
Перевір.		Березька		10.06		У	1	1
Н. контр.		Засорнова				ХНУ, КІ2-20-1		
Затв.		Говорущенко		10.06				

АНОТАЦІЯ

Тема кваліфікаційної роботи: «Проміжне програмне забезпечення мультикомп'ютерних систем з топологією «Решітка»».

Автор роботи: Ільчишина Юлія Віталіївна.

Керівник роботи: Березька Катерина Миколаївна.

Пояснювальна записка: 66 с., 16 рис., 3 дод., 47 джерел.

Графічна частина: 3 креслення.

Метою кваліфікаційно роботи є створення проміжного програмного забезпечення мультикомп'ютерної системи з топологією «Решітка». Важливою умовою для роботи з даною мультикомп'ютерною системою, можливість маштабованості системи, навіть для великих розмірів та проведення досліжень для оптимізації роботи та покращення створеного проміжного програмного забезпечення мультикомп'ютерної системи.

Основним завданням даної кваліфікаційнох роботи є ознайомитися та провести аналіз інформації щодо мультикомп'ютерних систем, описати принципи функціонування, переваги та недоліки різних топологій та потрібної топології «Решітка». Створити проміжне програмне забезпечення відповідно до даної теми та провести потрібні дослідження та експерименти, описати ефективність зроблених експериментів.

________________ 30.05.2024

Підпис студента Дата

ЗМІСТ

ВСТУП 4
1 ТЕОРЕТИЧНЕ ДОСЛІДЖЕННЯ ПРОМІЖНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ МУЛЬТИКОМП'ЮТЕРНИХ СИСТЕМ З ТОПОЛОГІЄЮ «РЕШІТКА»...5
1.1 Мультикомп'ютерна система. Відмінності мультиком'ютерної системи та однокомп'ютерної системи...5
1.2 Вузли в мультикомп'ютерних системах..8
1.3 Вузли в мультикомп'ютерних системах..10
1.4 Висновки 15
2 АРХІТЕКТУРА ТА ФУНКЦІОНАЛЬНІСТЬ МУЛЬТИКОМП'ЮТЕРНОЇ СИСТЕМИ...16
2.1 Топології мультикомп'ютерної системи. Топологія решітка 16
2.2 Механізми та алгоритми, підтримка цілісності та забезпечення функціоналу мультикомп'ютерної системи...25
2.3 Сокети. Зв'язки для пердачі повідомлень в мультикомп'ютерні системі з топології решітка...38
2.4 Алгоритми виявлення та виправлення помилок..................................39
2.5 Висновки 43
3 ПРАКТИЧНЕ ДОСЛІДЖЕННЯ ПРОМІЖНОГО ПРОГРПМНОГО ЗАБЕЗПЕЧЕННЯ МУТИЛЬТИКОМП'ЮТЕРНИХ СИСТЕМ З ТОПОЛОГІЄЮ «РЕШІТКА»...44
3.1 Структура програми та доцільність використання мови програмування С++. Бібліотеки для програмного забезпечення.................44
3.2 Логіка роботи проміжного програмного забезпечння мультикомп'ютерної системи з топологією «Решітка» 53
3.6. Результати роботи проміжного програмного забезпечння мультикомп'ютерної системи з топологією «Решітка» та експеременти ...56
3.6. Висновки 65

Зм.	Арк.	№докум.	Підпис	Дата	КвРКІ 200109.20.01.07 ПЗ			
Виконав		Ільчишина Ю.В.			Проміжне програмне забезпечення мультикомп'ютерних систем з топологією «Решітка»	Літера	Аркуш	Аркушів
Перевір.		Березька		10.06		у	2	66
Н.контр.		Засорнова І.О.				ХНУ КІ2-20-1		
Затвер.		Говорущенко Т.О.		10.06				

ВИСНОВКИ ... 68
ДОДАТОК А ... 75
ДОДАТОК Б ... 76
ДОДАТОК В ... 77
ДОДАТОК Г ... 77
ДОДАТОК Д ... 83
ДОДАТОК Е ... 90

ВСТУП

На сьогодні, поняття комп'ютерних систем, а саме мультикомп'ютерних систем, досить сильно турбує багатьох користувачі мережі Інтернет тому, що дані системи являються потужними обчислювальними ресурсами для вирішення складних завдань. Хоча дані системи можуть взаємодіяти з великими кількостями комп'ютерів, вони всі працюють як єдиний, налагоджений, обчислюваний механізм.

У мультикомп'ютерних системах існує багато варіантів з'єднання комп'ютерів між собою, цю частину прийнято називати топологією, але саме топологія «Решітка» є найоптимальнішим варіантом для з'єдання вузлів між собою. Саме у цій топології вузол має можливість з'єднання з чотирма сусідніми вузлами, що забезпечує належну стійкість систем до збоїв та відмов. Завдяки цьому мультикомп'ютерна система може належно виконувати свою роботу, навіть якщо один вузол відмов або сталася помилка.

Метою кваліфікаційно роботи є створення проміжного програмного забезпечення мультикомп'ютерної системи з топологією «Решітка». Важливою умовою для роботи з даною мультикомп'ютерною системою, можливість маштабованості системи, навіть для великих розмірів та проведення досліжень для оптимізації роботи та покращення створеного проміжного програмного забезпечення мультикомп'ютерної системи.

Основним завданням даної кваліфікаційнох роботи є ознайомитися та провести аналіз інформації щодо мультикомп'ютерних систем, описати принципи функціонування, переваги та недоліки різних топологій та потрібної топології «Решітка». Створити проміжне програмне забезпечення відповідно до даної теми та провести потрібні дослідження та експерименти, описати ефективність зроблених експериментів.

1 ТЕОРЕТИЧНЕ ДОСЛІДЖЕННЯ ПРОМІЖНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ МУЛЬТИКОМП'ЮТЕРНИХ СИСТЕМ З ТОПОЛОГІЄЮ «РЕШІТКА»

1.1 Мультикомп'ютерна система. Відмінності мультиком'ютерної системи та однокомп'ютерної системи

Мультикомп'ютерна система [1] є типом обчислювальної системи, що складається з декількох окремих комп'ютерів або вузлів, що з'єднані між собою з метою спільної роботи над досягненням певної мети. У відміну від традиційних однокомп'ютерних систем, мультикомп'ютерні системи розподіляють обчислювальні завдання між кількома вузлами, забезпечуючи паралельну обробку і підвищуючи продуктивність для певних типів додатків. Мультикомп'ютерну систему також називають як паралельну комп'ютерну систему.

У мультикомп'ютерних системах, кожен вузол зазвичай функціонує як відокремлений елемент, оскільки він оснащений власним процесором, пам'яттю та сховищем даних, що дає змогу йому виконувати завдання незалежно від інших. Ці вузли, або, іншими словами, комп'ютери, взаємодіють між собою через мережу, що дозволяє їм обмінюватися даними та координувати свою роботу. Залежно від потреб системи, а також від архітектури та дизайну системи, зв'язок між вузлами може бути здійснений за допомогою різних методів, таких як передача повідомлень або спільне використання пам'яті.

Мультикомп'ютерні системи часто використовуються в середовищах високопродуктивних обчислень (HPC) [2-5], де потрібна величезна обчислювальна потужність для таких завдань, як наукове моделювання, аналіз даних і складні симуляції. Розподіляючи робоче навантаження між кількома вузлами, багатокомп'ютерні системи можуть досягти значного прискорення і масштабованості порівняно з однокомп'ютерними системами.

Тепер детальніше розглянемо відмінності між однокомп'ютерною системою. Отже, в однокомп'ютерні системі використовується лише один

центральний процесор, який виконує інструкції, керує пам’яттю та обробляє операції вводу та виводу. Кожна людина використовує подібну систему в своєму житті, прикладом однокомп’ютерної системи є персональний комп’ютер, ноутбук, смартфони та сервери, які працюють незалежно.

Робота системи лише на одному процесорі є причоною того, що завдання в однокомп’ютерні системі виконуються лише в послідовному порядку, якщо немає підтримки паралельної обробки за допомогою таких технологій, як багатозадачність або багатопоточність. А у випадку виконання завдань у мультикомп’ютерні системі з початку підтримують паралелізм, оскільки системи з декількома комп’ютерами складаються з декількох процесорів, які можуть виконувати завдання одночасно.

До відмінностей між цими системами відносять і маштабованість. Однокомп'ютерні системи мають обмежену масштабованість. Хоча можна підвищити продуктивність шляхом модернізації, наприклад, додавання потужніших процесорів або збільшення пам'яті, але існує практична межа того, наскільки можна підвищити продуктивність без повної заміни системи.

Мультикомп'ютерні системи пропонують кращу масштабованість за рахунок додавання більшої кількості комп'ютерів або процесорів до системи, отже продуктивність може бути збільшена лінійно (за умови належної програмної підтримки та ефективного зв'язку між вузлами).

Говорячи про відмінності варто згадати також і відмовостійкість, у англійські мові для назви цього терміну використовують словосполучення слів «Fault Tolerance». Системам з одним комп'ютером може не вистачати відмовостійкості, оскільки збій в одному процесорі може вивести з ладу всю систему.

Мультикомп'ютерні системи можуть бути спроектовані з механізмами резервування та відмовостійкості. Навіть якщо один вузол виходить з ладу, система може продовжувати надійно працювати з рештою вузлів, зменшуючи

ризик повної відмови системи. Мультикомп’ютерна система є доволі стійкою до збоїв в окремих вузлах.

Процес використання ресурсів також відноситься до різниці цих двох систем. Системи з одним комп'ютером зазвичай мають обмежені можливості спільного використання ресурсів. Такі ресурси, як пам'ять, сховище та обчислювальна потужність, призначені для одного користувача або програми.

Мультикомп'ютерні системи можуть ділити ресурси між кількома вузлами, що дозволяє ефективніше використовувати ресурси та краще підтримувати багатозадачність і паралельну обробку, унаслідок чого підвищується результат.

Для системи важлива також комунікація. У випадку однокомп’ютерних систем витрати на зв’язок є мінімальними за рахунок роботи лише одного процесору.

Мультикомп'ютерні системи можуть мати вищі комунікаційні витрати через необхідність міжвузлового зв'язку, особливо в сценаріях розподілених обчислень. Ефективні комунікаційні протоколи та мережеві архітектури мають вирішальне значення для мінімізації цих витрат. Зв'язок між вузлами має вирішальне значення для координації та обміну даними в багатокомп'ютерній системі.

Вибір між однокомп'ютерною та багатокомп'ютерною системою залежить від таких факторів, як вимоги до продуктивності, масштабованості, відмовостійкості та бюджетних обмежень.

Для належного та ефективного працювання, забезпечення надійності та маштабованості мультикомп’ютерних систем потрібно правильно підібрати компоненти для даної системи та мати деякі ключові характеристики.

Вибір апаратних компонентів може змінюватися в залежності від конкретних вимог та задач мультикомп’ютерної системи. Для прикладу можна розглянути кластер високопродуктивних обчислень (HPC) можуть мати потужні процесори та великі обсяги оперативної пам'яті для виконання інтенсивних обчислювальних завдань, тоді як для вузлів розподіленої системи зберігання даних пріоритетними можуть бути ємність та надійність зберігання.

Також в залежності від структури системи, на вузлах можуть працювати однакові або різні операційні системи в додаток до яких на різних вузлах можуть працювати різні програмні компоненти і додатки, адаптовані до конкретних завдань, які вони виконують в системі. Це може бути проміжне програмне забезпечення для розподілених обчислень, бази даних, веб-сервери або спеціальне програмне забезпечення для спеціалізованих додатків.

Однією з основних переваг мультикомп'ютерних систем є їх масштабованість. Вузли можуть бути легко додані або видалені з системи, щоб пристосуватися до зміни робочого навантаження або вимог до ресурсів. Маштабність про яку йдеться мова може досягнутися за доромогою таких методів, як горизонтальне масштабування (додавання нових вузлів) або вертикальне масштабування (модернізація існуючих вузлів за допомогою більш потужного обладнання). Масштабованість гарантує, що система зможе ефективно справлятися зі зростаючими робочими навантаженнями.

Мультикомп’ютерні системи часто використовуються для надійного та безперервного процесу в разі випадку відмови вузлів або збоїв у мережі. Такі методи, як резервування, реплікація та балансування навантаження, можуть бути використані для пом'якшення впливу збоїв і забезпечення безперебійної доступності сервісів. Протоколи розподіленого консенсусу, такі як Paxos або Raft, можуть використовуватися для досягнення згоди між вузлами за наявності збоїв або розділів мережі.

Загалом, мультикомп'ютерні системи забезпечують потужну і гнучку платформу для вирішення обчислювально інтенсивних завдань, використовуючи колективну обчислювальну потужність декількох з'єднаних між собою комп'ютерів.

1.2 Вузли в мультикомп’ютерних системах

В мультикомп’ютерних системах вузли — це найголовніший елемент в данній системі. Вузли — це окремі обчислюванні одиниці, які з'єднані між собою, щоб сформувати більшу розподілену обчислювальну інфраструктуру. Вузлом може бути будь-який обчислювальний пристрій, наприклад, комп'ютер, сервер або процесор, в мультикомп'ютерній системі. Кожен вузол, як правило, має власні обчислювальні можливості, пам'ять та підключення.

У таких великих проєктів як мутикомп’ютерні системи існують різні типи вузлів, як-то обчислювальні вузли (англійською мовою «Compute nodes»), вузли пам’яті (англійською мовою «Storage nodes»), комунікаційні вузли (англійською мовою «Communication nodes»), вузли-шлюзи (англійською мовою «Gateway nodes» [6-8]).

Обчислювальні вузли відповідальні за виконання обчислювальних завдань і обробку даних. Зазвичай вузли подібного типу містять процесори, пам'ять (оперативну пам'ять — RAM) та інші компоненти, необхідні для обчислень. Обчислювальні вузли виконують такі завдання, як обробка даних, аналіз, імітація, моделювання та інші обчислювальні операції. У кластерах високопродуктивних обчислень (HPC), суперкомп'ютерах і середовищах грід-обчислень обчислювальні вузли є основними робочими елементами, що відповідають за виконання складних обчислень.

Вузли пам’яті призначені для зберігання, пошуку, реплікації, резервного копіювання та керування даними в мультикомп'ютерній системі. Вони можуть складатися з жорстких дисків (HDD [9-11]), твердотільних накопичувачів (SSD [12-15]) або інших пристроїв зберігання даних. Вузли пам’яті даних часто надають доступ до спільних ресурсів зберігання, таких як розподілені файлові системи або мережеві сховища (NAS).

Комунікаційні вузли забезпечують зв'язок і передачу даних між іншими вузлами системи. Комунікаційні вузли відіграють вирішальну роль у підтримці

зв'язності мережі, забезпеченні надійного обміну даними та підтримці ефективної міжвузлової комунікації. У деяких архітектурах для виконання мережевих функцій використовуються спеціалізовані комунікаційні вузли, такі як комутатори або маршрутизатори.

Вузли-шлюзи служать точками входу і виходу для зв'язку між мультикомп'ютерною системою і зовнішніми мережами або системами. Вони забезпечують зв'язок між мультикомп'ютерною системою та зовнішніми мережами, такими як Інтернет або іншими обчислювальними кластерами. Вузли-шлюзу часто виконують такі завдання, як трансляція протоколів, брандмауер, трансляція мережевих адрес (NAT [16-17]) і маршрутизація між різними мережевими доменами. Ці вузли дозволяють мультикомп'ютерній системі взаємодіяти із зовнішніми ресурсами, отримувати доступ до віддалених джерел даних або надавати послуги зовнішнім клієнтам.

Кожен тип вузлів виконує обчислення або завдання, які призначені їм системою. Також для перевірки роботи або результатів роботи вузлів існують спеціальні інструменти моніторингу та програмне забезпечення, які допомагають адміністраторам відстежувати стан вузлів або показники продуктивності.

Вузли відіграють вирішальну роль у функціонуванні та продуктивності багатокомп'ютерних систем, а ефективне управління ними має важливе значення для ефективного та надійного досягнення бажаних обчислювальних цілей.

1.3 Вузли в мультикомп'ютерних системах

Загалом, розвиток мультікомп'ютерних систем був поступовим процесом, який тривав десятиліттями і був відзначений постійними інноваціями і вдосконаленням у галузі мереж, паралельних обчислень та технологій розподілених систем. Сьогодні мультікомп'ютерні системи є загальнопоширеними, підтримуючи різноманітність застосувань та сервісів, які

використовують розподілені обчислення, паралельну обробку та колаборативні архітектури обчислень.

Мультикомп'ютерні системи розвивалися з часом на фоні досягнень у галузі обчислювальних технологій. Концепція з'єднання кількох комп'ютерів для спільної роботи має свої коріння ще з початкових днів історії обчислювальної техніки. Однак формальний розвиток і широке впровадження мультікомп'ютерних систем можна прослідкувати за кількома ключовими етапами.

Перший етап розвитку мультикомп’ютерних систем відбувся в 1960-1970-тих роках. У цей момент починають з’являтися ранні форми розподілених обчислювальних систем, часто називаються "комп'ютерними мережами". Наприклад, ARPANET [18-20] (Мережа агентства передових досліджень проектів), яку розроблено наприкінці 1960-х років Міністерством оборони США, і ранні локальні мережі, такі як Ethernet [21-23].

Другий етап йшов послідовно в 1980 роки. В 1980-х роках відбулися значні досягнення в галузі технологій мереж і почалося формування більш складних архітектур розподілених обчислювальних систем. Наукові установи та університети почали експериментувати з розподіленими обчислювальними системами для паралельної обробки [24-25] та наукових обчислень.

Під час третього етапу-десятиліття в 1990 роках спостерігався розквіт мультикомп'ютерних систем у різних галузях, завдяки досягненням у сфері апаратного забезпечення, мережі та програмних технологій. Високопродуктивні обчислювальні кластери [26-30], платформи паралельних обчислень та розподілені обчислювальні фреймворки стали популярними для розв'язання обчислювально-інтенсивних завдань та великих обчислень.

Чертвертий етап — 2000-ті роки і до наших днів. У XXI столітті продовжується зростання та диверсифікація мультикомп'ютерних систем у різних галузях, таких як фінанси, охорона здоров'я, виробництво та розваги. Платформи хмарних обчислень з'явилися як домінуюча парадигма, надаючи масштабований,

доступний за запитом доступ до мультикомп'ютерних систем та розподілених обчислювальних ресурсів. Продовжується розвиток мультикомп'ютерних систем з появою хмарних обчислень, аналізу великих даних та Інтернету речей (IoT). Платформи хмарних обчислень, такі як Amazon Web Services (AWS [31-33]), Microsoft Azure [33-35] та Google Cloud Platform [36-37] (GCP), використовують мультикомп'ютерні архітектури для надання масштабованих, доступних за запитом обчислювальних ресурсів через Інтернет. Технології великих даних, включаючи Apache Hadoop [38] та Apache Spark [39], використовують потужність мультикомп'ютерних систем для обробки та аналізу обширних наборів даних, розподілених по кількох вузлах. Обчислення на краю з'явилося як парадигма для розгортання.

Сьогодні, мультикомп'ютерні системи широко використовуються різними організаціями, від малих підприємств до великих підприємств, наукових установ, урядових агентств та постачальників хмарних послуг. Частота їх використання залежить від кількох факторів, включаючи характер робочого навантаження, вимоги до продуктивності, обмеження бюджету та технологічні досягнення. Далі наведено декілька прикладів, де досить часто використовують мультикомп’ютерні системи. Наприклад:

1. Високопродуктивні обчислення (HPC). Галузі, такі як наукові дослідження, інженерія, фінанси, розвідка нафти та газу, широко використовують мультикомп'ютерні системи для складних симуляцій, моделювання, аналізу даних та обчислювальних досліджень.

HPC-кластери, суперкомп'ютери та інфраструктури решітчастих обчислень є поширеними випадками використання мультикомп'ютерних систем в цих галузях.

У наукових дослідженнях мультикомп'ютерні системи широко використовуються у наукових дослідженнях для завдань, таких як обчислення руху рідини, моделювання клімату, молекулярна динаміка та ядерні симуляції. Ці

системи дозволяють дослідникам обробляти величезні набори даних та запускати складні симуляції, які потребують великої обчислювальної потужності.

У галузі інженерії різні галузі інженерії, такі як авіаційна, автомобільна та цивільна, використовують мультикомп'ютерні системи для аналізу скінченних елементів, обчислювальної структурної механіки та інших інженерних симуляцій. Кластери високопродуктивних обчислень дозволяють інженерам оптимізувати конструкції, аналізувати продуктивність та моделювати сценарії реального світу.

Сфери пов'язані з фінансами, а саме у фінансовому секторі мультикомп'ютерні системи використовуються для аналізу ризиків, алгоритмічної торгівлі, оптимізації портфеля та торгівлі з високою частотою. Ці системи дозволяють фінансовим установам обробляти величезні обсяги ринкових даних, проводити складні обчислення та приймати швидкі торгові рішення.

2. Центри обробки даних та хмарні обчислення. Постачальники хмарних послуг використовують мультикомп'ютерні системи для створення масштабованої та надійної інфраструктури для надання хмарних послуг, включаючи обчислювальні ресурси, зберігання та мережу.

Центри обробки даних використовують мультикомп'ютерні системи для підтримки віртуалізації, контейнеризації та розподіленого обчислення, що відповідає зростаючому попиту на хмарні додатки та послуги.

Організації використовують мультикомп'ютерні системи, надані постачальниками хмарних послуг, для хостингу віртуальних машин, зберігання та мережевих ресурсів. Послуги IaaS дозволяють підприємствам масштабувати свою інфраструктуру динамічно, платити лише за використані ресурси та звільнити від управління апаратним та програмним забезпеченням.

Платформи PaaS використовують мультикомп'ютерні архітектури для надання середовищ розробки та розгортання для створення та запуску додатків. Розробники можуть зосередитися на написанні коду без переймання про підтримку інфраструктури, тоді як постачальники PaaS відповідають за надання ресурсів, масштабування та обслуговування.

Додатки SaaS розміщені на мультикомп'ютерних системах, що дозволяє користувачам отримувати доступ до програмних додатків через Інтернет без потреби у встановленні або обслуговуванні. Архітектури багатоквартирних додатків дозволяють постачальникам SaaS обслуговувати кілька клієнтів зі спільної інфраструктури, досягаючи економічної ефективності та ефективності витрат.

3. Аналітика великих даних та машинне навчання. Організації, які обробляють великі обсяги даних для аналізу, машинного навчання та штучного інтелекту, часто використовують мультикомп'ютерні системи для ефективної обробки великих обсягів даних. Розподілені системи обчислень, такі як Apache Hadoop, Apache Spark та TensorFlow [40-41], використовують мультикомп'ютерні системи для паралельної обробки даних та завдань навчання на кількох вузлах.

Мультикомп'ютерні системи використовуються у середовищах зберігання даних для зберігання, обробки та аналізу великих обсягів структурованих та неструктурованих даних. Сховища даних дозволяють організаціям отримувати інсайти з історичних та реальних даних, підтримувати прийняття рішень та приводити до ініціатив бізнес-інтелекту.

Мультикомп'ютерні системи забезпечують потужність обчислень для навчання складних моделей машинного навчання та алгоритмів штучного інтелекту, обробки великих обсягів даних та виконання завдань виведення. Розподілені фреймворки машинного навчання та бібліотеки глибокого навчання використовують мультікомп'ютерні архітектури для прискорення навчання та виведення моделей.

4. IT-інфраструктура підприємства. Мультикомп'ютерні системи часто використовуються в якості частини IT-інфраструктури підприємства для підтримки ключових бізнес-застосунків, баз даних, веб-серверів та систем управління підприємством (ERP [42-43]). Мультикомп'ютерні системи дозволяють підприємствам досягти високої доступності, масштабованості та

надійності їх IT-послуг, забезпечуючи різноманітність робочих навантажень та вимог користувачів.

Великі підприємства розгортають мультикомп'ютерні системи у своїх центрах обробки даних для підтримки критичних застосунків, баз даних та сервісів. Ці системи забезпечують високу доступність, стійкість до відмов та масштабованість для властивостей обчислення сучасних підприємств.

Компанії з веб-хостингу використовують мультикомп'ютерні системи для надання послуг спільного хостингу, віртуальних приватних серверів (VPS [44]), присвячених серверів та хостингу в хмарі для клієнтів. Архітектури мультікомп'ютерних систем дозволяють ефективне використання ресурсів, ізоляцію та налаштування середовищ хостингу.

Крім вище написаного мультикомп'ютерні системи, розгорнуті на межі мережі, підтримують застосунки обчислення на межі в області Інтернету речей. Ці системи обробляють та аналізують дані сенсорів, керують виконавчими пристроями та приймають рішення в реальному часі ближче до джерела генерації даних. Архітектури обчислень на межі підвищують ефективність, зменшують затримки та зберігають пропускну спроможність у розгорнутому Інтернеті речей.

Мультікомп'ютерні системи досить часто ще використовують у блокчейнах мереж як майнингові ріги, використовуються для підтвердження транзакцій та захисту блокчейн мереж. Ці системи виконують криптографічні обчислення для досягнення консенсусу та збереження цілісності розподіленого реєстру. Майнери криптовалют вносять обчислювальну потужність у мережу в обмін на винагороди, такі як нові монети та комісійні збори.

Ці всі приклади відображають різноманітні застосування та використання мультікомп'ютерних систем у різних галузях та областях, демонструючи їх важливість у підтримці критичних операцій бізнесу, забезпеченні інновацій та сприянні технологічному прогресу.

1.4 Висновки

У першому розділі було ретельно розглянути саме поняття мультикомп'ютерної системи, також зроблено порівняння однокомп'ютерної системи та мультикомп'ютерної системи, описано типи вузлів, розглянутий розвиток мультикомп'ютерних систем від початку створення до наших днів, а також наведено декілька прикладів, де досить часто використовують мультикомп'ютерні системи та потенціал застосування мультикомп'ютерних мереж.

2 АРХІТЕКТУРА ТА ФУНКЦІОНАЛЬНІСТЬ МУЛЬТИКОМП'ЮТЕРНОЇ СИСТЕМИ

2.1 Топології мультикомп'ютерної системи. Топологія решітка

У контексті мультикомп'ютерної системи топологія є ключевим поняттям. Адже тут топологія відіграє ключову роль, описуючи фізичну або логічну структуру розташування обчислювальних вузлів і їх взаємозв'язки. Вона визначає, як саме ці вузли пов'язані один з одним, а також маршрути, через які протікають дані та комунікація всередині системи. Важливим аспектом топології є визначення ефективності, продуктивності, масштабованості та стійкості до помилок мультікомп'ютерних систем. Правильність підбору топології відіграє важливу роль в ефективності роботи та використання мультикомп'ютерної системи.

Конкретна топологія мережі визначає структуру мережі впливає на такі аспекти, як затримка комунікації, пропускна здатність, надійність та використання ресурсів. Її можна графічно зображати, або іншими словами — візуалізувати, у вигляді графічних схем, які демонструють зв'язки між вузлами для комунікації і шляхи передачі інформації, даних. Серед найпоширеніших топологій у мультікомп'ютерних системах можна відзначити топологію на основі шин, кільця, сітки, гіперкуба та дерева, кожна з яких має свої особливості та відповідність для різноманітних застосувань, вимог і потреб.

У сучасних мультікомп'ютерних системах, топологія описує фізичне або логічне розташування вузлів обчислювальної мережі та їхніх взаємозв'язків. Це важливий аспект, що визначає, як саме вузли з'єднані між собою і якими маршрутами проходять дані та комунікації всередині системи. Кожна конкретна топологія мережі має вплив на такі показники, як затримка передачі даних, пропускна здатність, надійність та використання ресурсів. Отже, топологія визначає не лише структуру мережі, але й її ефективність, продуктивність, масштабованість та стійкість до помилок.

Загалом, топологія визначає фундаментальну структуру вся робота мультікомп'ютерної системи, формуючи їх архітектуру та функціональність відповідно до вимог різних обчислювальних завдань і умов середовища.

Повертаючись до візуалізації (зображення графічно) роботи топологій в мультикомп’ютерній системі кожну топологію можна подати у вигляді простого графу, якмй допоможе краще розуміти принцип дії. У цих графах вершини відповідають процесорним вузлам, а ребра слугують каналами для зв’язку.

Існують деякі загальні класифікації топологій мереж залежно від їхньої природи. З одного боку, топології можна поділити на фіксовані та реконфігуровані. Перші є статичними і не змінюють свою конфігурацію з часом, тоді як реконфігуровані дозволяють динамічно змінювати структуру мережі. З іншого боку, топології можна розділити на регулярні та нерегулярні. Регулярні топології мають однорідну або подібну структуру між вузлами, тоді як нерегулярні можуть мати різноманітні конфігурації без чітких шаблонів.

Наприклад, у фіксованих топологіях, таких як "зірка" або "кільце", зв'язки між вузлами залишаються незмінними протягом тривалого періоду. У той же час, у реконфігурованих топологіях, які можуть бути представлені, наприклад, мережею "меш", адміністратор може змінювати з'єднання між вузлами в залежності від потреб мережі. Регулярні топології, такі як "Сітка", відображають однорідну структуру з'єднання, тоді як нерегулярні топології, наприклад, "Матриця", можуть мати різні комбінації з'єднань між вузлами, що робить їх більш гнучкими, але складнішими для управління.

Далі наведено зображенння широко використовуваних топологій які згадувалися у тексті вище цього розділу. На рисунку 2.1 зображено графічно зображений принцип роботи топології «Лінійка».

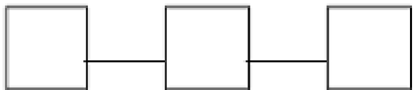


Рисунок 2.1 — Графічно зображений принцип роботи топології «Лінійка»

У топології «Лінійка», також відомій як топологія лінійна, обчислювальні вузли розташовані послідовно в одному ряду або шляху, причому кожен вузол безпосередньо підключений до своїх сусідніх сусідів. Ця топологія нагадує пряму лінію, з передачею даних від одного кінця до іншого. Саме від такого типу з’єднання пішла назва для даної топології.

У топології «Лінійка» кожен обчислювальний вузол підключений точно до двох інших вузлів, крім кінцевих вузлів, які підключені тільки до одного сусіднього вузла. Це створює лінійний ланцюг вузлів, з передачею даних від одного вузла до іншого вздовж лінії. Коли вузол потребує спілкування з іншим вузлом у лінії, він відправляє пакети даних вздовж лінії до призначеного вузла. Передача даних відбувається послідовно від одного вузла до наступного, слідуючи лінійному шляху, визначеному топологією.

Вузли у топології «Лінійка» виступають як отримувачі та передавачі даних. Кожен вузол отримує дані від попереднього вузла у лінії, обробляє їх у разі необхідності, а потім пересилає їх наступному вузлу в лінії, поки вони не дійдуть до призначеного вузла. У топології «Лінійка» затримка комунікації збільшується лінійно з відстанню між вузлами. Чим далі розташовані два вузли, тим більше часу потрібно для передачі даних між ними. Це може призвести до вищої затримки для вузлів, що розташовані на кінцях лінії, порівняно з вузлами у середині.

Топологія «Лінійка» пропонує обмежену стійкість до відмов, оскільки відмова одного вузла може порушити комунікацію по всій лінії. Однак деякі лінійні топології можуть включати зайвість або альтернативні шляхи для пом'якшення впливу відмови вузла. Топологію «Лінійка» можна розширити,

додавши більше вузлів до лінії, тим самим збільшивши її довжину. Однак, зі зростанням кількості вузлів може збільшуватися затримка комунікації, а управління довшими лініями може стати складнішим.

На рисунку 2.2 представлено графічно зображений принцип роботи топології «Кільце».

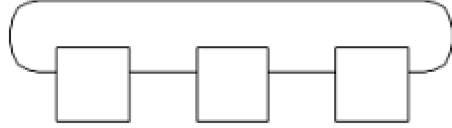


Рисунок 2.2 — Графічно зображений принцип роботи топології «Кільце»

У топології «Кільце», обчислювальні вузли розташовані у круговому або кільцеподібному порядку, причому кожен вузол підключений до точно двох сусідніх вузлів, створюючи коловий ланцюг або кільце вузлів та утворюючи закриту мережу. Передача даних відбувається послідовно навколо кільця, при цьому кожен вузол має власний шлях для спілкування зі своїми сусідніми сусідами.

Коли вузол потребує спілкування з іншим вузлом у кільці, він відправляє пакети даних у певному напрямку, будь то за годинниковою або проти годинникової стрілки, навколо кільця. Кожен вузол отримує пакет даних, обробляє його у разі необхідності, а потім пересилає наступному вузлу в кільці, поки він не дістанеться призначеного вузла. Передача даних у топології кільця є постійною, з пакетами даних, що обертаються навколо кільця безстроково, поки вони не дійдуть свого призначення. В кільці немає фіксованої початкової або кінцевої точки, оскільки дані можуть рухатися в обох напрямках навколо петлі.

На відміну від інших топологій, де дані можуть пройти кілька шляхів до свого призначення, в топології кільця є тільки один шлях передачі по круговому ланцюгу. Це спрощує маршрутизацію даних, але також може призвести до заторів або затримок у разі виникнення проблем або відмов у мережі. Затримка

комунікації у топології кільця залежить від кількості вузлів у кільці та швидкості передачі даних в мережі. Довші кільці з більшою кількістю вузлів можуть виказувати вищу затримку через збільшення часу реакції навколишніх вузлів на передачу даних. У той же час, швидкість передачі даних також може впливати на продуктивність мережі, забезпечуючи швидке переміщення даних навколо кільця та зменшуючи загальну затримку.

Топологія «Кільце» надає певний рівень стійкості до відмов, оскільки дані можуть продовжувати обертатися навколо кільця, навіть якщо один вузол виходить з ладу. Однак якщо вузол або з'єднання в кільці вийде з ладу, це може нарушити комунікацію для всього сегмента мережі поза точкою відмови, поки мережа не переконфігурується або не відновиться.

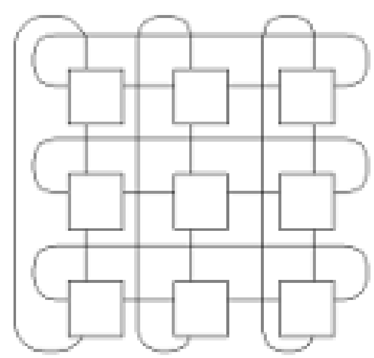


Рисунок 2.3 — Графічно зображений принцип роботи топології «Тор 3x3»

Топологія «Тор 3x3» — це топологія мережі, що утворює тривимірну сітку з зв'язками з перемотуванням, нагадуючи форму тора або кільця. На рисунку 2.3 представлено графічно зображений принцип роботи топології «Тор 3x3».

У «Тор 3x3» обчислювальні вузли розташовані у тривимірній сітці з трьома рядами, трьома колонками та трьома шарами. Кожен вузол підключений до шести сусідніх вузлів: двох вузлів у тому ж ряду, двох вузлів у тій же колонці та двох вузлів у тому ж шарі. Додатково, існують зв'язки перемотування між протилежними сторонами сітки, що дозволяють даним подорожувати по тору безперешкодно. Коли вузол потребує спілкування з іншим вузлом у торі, він

відправляє пакети даних до призначеного вузла за прямим шляхом по сітці. Дані можуть подорожувати горизонтально, вертикально або по діагоналі в межах того ж шару сітки. Якщо призначений вузол не знаходиться безпосередньо поруч у тому ж шарі, дані можуть обгортатися навколо тора, проходячи через зв'язки перемотування, щоб ефективно досягти свого призначення.

Зв'язки перемотування у даній топології дозволяють даним безперервно переходити з одного боку сітки на протилежний, створюючи неперервну та взаємопов'язану мережу. Ця функція перемотування усуває потребу у складних маршрутних алгоритмах та забезпечує, що дані завжди знайдуть шлях до свого призначення, навіть якщо для цього потрібно перетнути кілька шарів або обертатися навколо тора.

Топологія «Тор 3х3» є стійкою до відмов, надаючи кілька резервних шляхів для передачі даних. Якщо вузол або зв'язок виходить з ладу, дані можуть перенаправлятися через альтернативні шляхи, використовуючи зв'язки перемотування для збереження зв'язності всередині тора. Ця резервність допомагає зменшити вплив відмов і забезпечує неперервну роботу мережі. Цю топологію можна масштабувати, додавши більше рядів, колонок та шарів до сітки, розширюючи її розмір і потужність. Кожен додатковий вузол збільшує обчислювальну потужність та пропускну спроможність мережі, зберігаючи при цьому ефективність і стійкість до відмов топології тора.

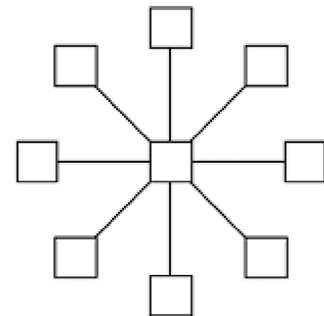


Рисунок 2.4 — Графічно зображений принцип роботи топології «Зірка»

У топології "Зірка" [46] обчислювальні пристрої з'єднані з центральним комутатором або концентратором. Це нагадує структуру зірки з центральним вузлом (комутатором або концентратором), з якого виходять промені, що представляють підключені пристрої. На рисунку 2.4 можна побачити графічно зображений принцип роботи топології «Тор 3х3».

У топології "Зірка" є один центральний вузол — комутатор або концентратор, до якого підключені всі обчислювальні пристрої у мережі. Кожен обчислювальний пристрій (комп'ютер, принтер, тощо) підключений безпосередньо до центрального вузла. Кожен пристрій має свій власний кабель, який йде від нього до комутатора або концентратора. Коли відбувається комунікація між двома пристроями, вони взаємодіють через центральний вузол, що може збільшити час передачі даних.

Отже, всі дані між пристроями в мережі проходять через центральний вузол. Коли один пристрій надсилає дані до іншого, вони спочатку пересилаються до центрального вузла, який потім пересилає їх до призначеного пристрою. Це також означає, що пристрої можуть комунікувати паралельно, а не послідовно.

Топологія "Зірка" забезпечує високий рівень стійкості, оскільки відмова одного пристрою не впливає на роботу інших. Вона також проста у налаштуванні та управлінні. Однак пропускна здатність мережі може бути обмеженою пропускною здатністю центрального вузла. Багато пристроїв, що намагаються передавати дані одночасно, можуть призвести до конфліктів та затримок.

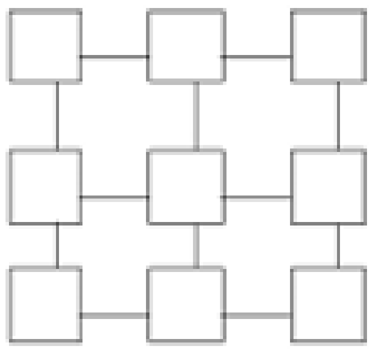


Рисунок 2.5 — Графічно зображений принцип роботи топології «Решітка»

І останнім прикладом розглянемо потрібну та обрану для виконання кваліфікаційної роботи здобувача вищої освіти топологію — топологія «Решітка» [47]. На рисунку 2.5 представлено графічно зображений принцип роботи топології «Решітка».

У топології «Решітка» обчислювальні вузли розташовані у структурованому сітчастому вигляді, схожому на решітку або мережу. Кожен вузол з'єднаний з сусідніми вузлами, утворюючи регулярну сітку.

Обчислювальні вузли розташовані у сітчастій структурі, схожій на решітку або мережу. Кожен вузол з'єднаний зі своїми сусідами у регулярному та структурованому порядку. Розташування може бути двовимірним (2D) або тривимірним (3D) в залежності від конкретної топології.

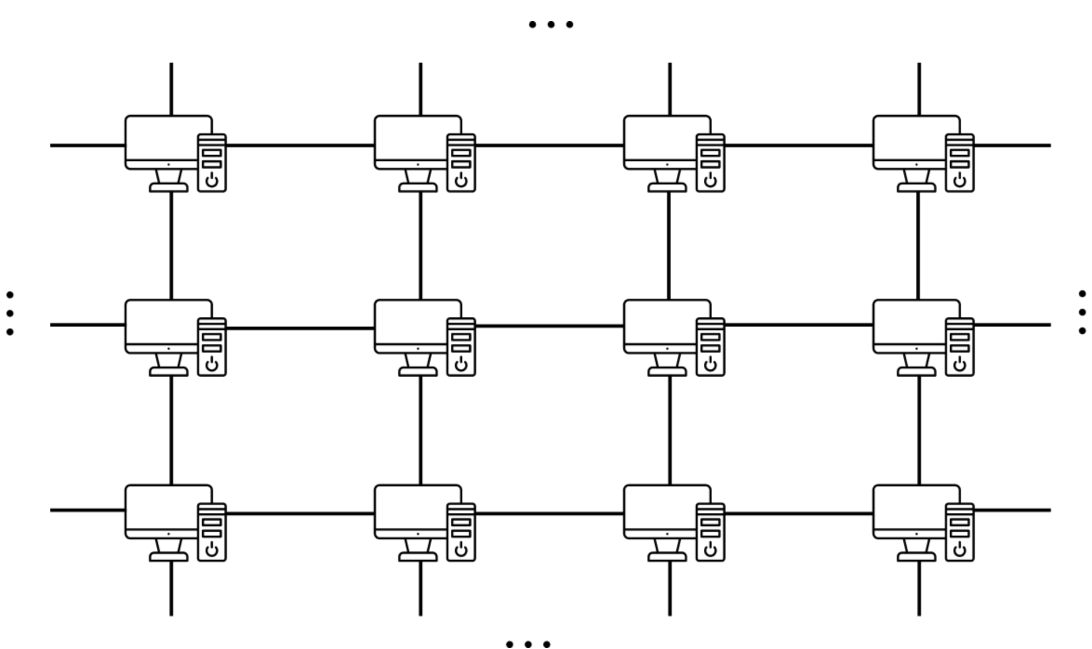


Рисунок 2.6 — Схема підключень вузлів мультикомп'ютерної системи «Решітка»

Кожен вузол безпосередньо з'єднаний зі своїми сусідами в решітці. На рисунку 2.6 можна наочно побачити схему підключень вузлів мультикомп'ютерної системи «Решітка» У двовимірній решітці кожен вузол зазвичай має зв'язки з його чотирма найближчими сусідами (вгорі, внизу, зліва та

справа). У тривимірній решітці кожен вузол зазвичай з'єднаний з його шістьма найближчими сусідами (вгорі, внизу, зліва, справа, зверху та знизу). Це структуроване з'єднання формує регулярну сіткову мережу.

Коли вузол потребує спілкування з іншим вузлом у решітці, він відправляє пакети даних безпосередньо до цільового вузла через підключені шляхи. Дані можуть переміщатися вздовж структури решітки, перескакуючи з одного вузла на його сусідів, доки не досягнуть свого призначення. Цей прямий шлях сприяє ефективній передачі даних у мережі решітки.

Маршрутизація в топології «Решітка» зазвичай є структурованою та детермінованою. Кожен вузол слідує заздалегідь визначеному алгоритму маршрутизації, щоб визначити шлях для передачі даних на основі розташування цільового вузла в решітці. Це структурована маршрутизація спрощує процес маршрутизації та забезпечує надійну доставку даних.

Топологія «Решітка» властива вбудована масштабованість. Зі зростанням кількості вузлів решітка може бути розширена шляхом додавання нових рядів, стовпців або шарів для розміщення додаткових вузлів. Ця масштабованість робить топологію «Решітка» підходящою як для малих, так і для великих розподілених систем.

Топологія «Решітка» має вбудовану стійкість до відмов. Якщо вузол або з'єднання вийде з ладу, дані можуть бути перенаправлені через альтернативні шляхи вздовж структури решітки, щоб дістатися до свого призначення. Запасні шляхи у топології решітка допомагають пом'якшити наслідки відмов вузлів і забезпечують неперервну роботу мережі.

Топологія «Решітка» застосовується в різних розподілених системах обчислення, включаючи паралельне обчислення, мережі датчиків і грід-системи. Її регулярнарна та структурована природа сприяє ефективній обмін даними та координації обчислень між вузлами. До цього, топологія решітка може бути використана для широкого спектру застосувань, включаючи наукові дослідження, обробку великих обсягів даних, суперкомп'ютерні системи, ігрові сервери та інші

області, де потрібна велика обчислювальна потужність та надійна комунікація між вузлами. В цілому, топологія «Решітка» є ефективним і надійним вибором для побудови розподілених систем, які вимагають структурованої та масштабованої мережевої архітектури.

2.2 Механізми та алгоритми, підтримка цілісності та забезпечення функціоналу мультикомп’ютерної системи

Сьогодні у сучасному взаємопов'язаному світі багатокомп'ютерні системи є основою складних застосунків у різних галузях, таких як обробка даних, розподілені обчислення та хмарні сервіси. Забезпечення цілісності та надійності цих систем є критично важливим для їх успішної роботи. Особливо важливими є механізми та алгоритми, які допомагають підтримувати цілісність та надійність багатокомп'ютерної системи.

Багатокомп'ютерні системи включають кілька комп'ютерів або вузлів, які взаємодіють як частина більшої мережі. Вони можуть варіюватися від малих кластерів до великих систем, таких як центри обробки даних та хмарні інфраструктури. Ці системи забезпечують обчислювальну потужність і сховище, необхідні для роботи зі складними застосунками та обробки великих масивів даних.

Отже, основними механізмами є:

1. Стійкість до збоїв.
2. Балансування навантаження.
3. Алгоритми досягнення згоди.
4. Захист інформації.
5. Відновлення після збоїв.

Стійкість до збоїв — це здатність багатокомп'ютерної системи продовжувати працювати, незважаючи на відмову одного чи кількох компонентів. Для забезпечення стійкості часто використовуються такі методи, як реплікація,

резервування та перемикання. При реплікації дані та послуги дублюються на декілька вузлів. Якщо один вузол відмовляє, інший вузол може швидко взяти на себе його роботу. Резервуванням є надлишкові компоненти та системи забезпечують резервування у разі відмови певних елементів. А автоматичні механізми перемикання виявляють відмови і перенаправляють навантаження на працездатні вузли.

Балансування навантаження забезпечує ефективне використання ресурсів багатокомп'ютерної системи. Воно сприяє рівномірному розподілу навантаження між вузлами.

У мультикомп'ютерних системах з топологією решітки, забезпечення цілісності даних та виявлення і виправлення помилок є важливим завданням, оскільки ці системи характеризуються високим рівнем взаємозв'язку та розподіленими обчисленнями. Розглянемо основні методи забезпечення цілісності даних, а також алгоритми виявлення та виправлення помилок у такій системі.

Забезпечення цілісності даних у мультикомп'ютерній системі з топологією решітка є важливим для забезпечення надійності та безперервності роботи системи. Методи забезпечення цілісності даних у такій системі:

1. Розміщення дублбованих копій даних.
2. Контроль цілісності даних.
3. Синхронізація даних.

Розміщення дубльованих копій даних є основним методом забезпечення цілісності даних у розподілених системах, включаючи ті, що мають топологію решітки. Це передбачає зберігання копій даних в різних вузлах системи для підвищення надійності та відмовостійкості. У випадку пошкодження або втрати даних в одному вузлі, інший вузол може надати резервну копію.

Підтримка різних версій даних в різних вузлах може забезпечити безперервність роботи навіть у випадку пошкодження окремого вузла або втрати оригінальних даних в одному з вузлів.

Крім того розподіленість даних у різних вузлах системи таким чином, щоб забезпечити баланс навантаження та оптимізувати час доступу. Стратегія збереження резервних копій даних для швидкого відновлення системи слугує чудовим способом для збереження та відновлення всієї системи при критичних ситуаціях.

Контроль цілісності даних допомагає виявити пошкодження даних під час їх передачі або зберігання.

Використовуються контрольні суми або хеш-функції для перевірки цілісності даних. Дані перед відправленням або зберіганням хешуються, і контрольний хеш зберігається або передається разом з даними. При отриманні або зчитуванні даних їх хеш знову обчислюється і порівнюється з очікуваним.

Деякі системи можуть додавати контрольні блоки до даних, які містять додаткову інформацію про дані, наприклад, довжину або контрольні суми, що допомагає перевіряти цілісність.

Цілісність даних перевіряється під час передачі даних по мережі. Якщо дані були пошкоджені під час передачі, це може бути виявлено на основі контрольних сум.

Синхронізація даних забезпечує узгодженість і актуальність даних між різними вузлами системи. Взагалі, синхронізація даних — це процес усунення різниць між двома наборами даних. Вважається, що раніше ці копії були однакові, але потім одна або обидві копії могли зазнати незалежних змін. Отже, для синхронізації даних застосовуються протоколи для узгодження станів різних вузлів системи. Вони визначають правила обміну даними, щоб забезпечити узгодженість.

Часові мітки застосовуються в багатьох сферах, таких як комп'ютерні системи, бази даних, мережеві протоколи та програмування, щоб контролювати порядок подій, синхронізувати процеси або визначати часові рамки для операцій.

Часова мітка (таймстемп) — це значення, яке визначає конкретний момент часу. Як правило, вона виражається числом, яке представляє дату та час у

певному форматі, наприклад, у вигляді кількості секунд або мілісекунд, що минули з початку певної епохи (наприклад, з 1 січня 1970 року за UTC, відомого як Unix-час).

Часові мітки використовуються для відстеження актуальності даних. Вони допомагають визначити, які дані є найсвіжішими, і забезпечують синхронізацію даних між вузлами.

Механізм розподілених транзакцій — це технологія, яка дозволяє координувати та керувати виконанням транзакцій, що охоплюють кілька різних вузлів у розподіленій системі, такій як база даних або інша інформаційна система, розміщена на кількох серверах. Впровадження механізмів розподілених транзакцій дозволяє забезпечити цілісність і узгодженість даних під час їх зміни в різних вузлах. Вузли системи можуть обмінюватися даними для узгодження своїх станів. Наприклад, вузли можуть обмінюватися інформацією про оновлення або зміни даних. Система може моніторити стан даних і виправляти розбіжності, наприклад, шляхом обміну даними між вузлами або застосування інших методів синхронізації.

У механізмі розподілених транзакцій є основні аспекти, такі як: цілісність даних, протоколи узгодження, розподілена транзакція та компенсація.

Цілісність даних — це забезпечення, що дані залишаються узгодженими та цілісними в усіх частинах розподіленої системи, навіть у разі збоїв або помилок.

Протоколи узгодження існують для забезпечення узгодженості та атомарності транзакцій у розподіленій системі використовуються протоколи, такі як двофазне або трьохфазне узгодження. Двофазне і трьохфазне узгодження — це протоколи, які використовуються для забезпечення атомарності та узгодженості розподілених транзакцій у системах з декількома учасниками (вузлами або процесами). Ці протоколи координують рішення про підтвердження чи відхилення транзакції між усіма учасниками.

У двохфазному узгодженні існує два етапа при виконані процесу: фази підготовки та завершення. Фаза підготовки характеризується тим, що

координатор транзакції відправляє запит усім учасникам (вузлам) транзакції, питаючи, чи готові вони завершити транзакцію та кожен учасник відповідає у свою чергу координатору, чи готовий він завершити транзакцію. Якщо так, то учасник робить необхідні підготовчі дії (наприклад, блокує ресурси). При фазі завершення якщо всі учасники відповіли "готовий", координатор відправляє всім учасникам повідомлення "завершити транзакцію", кожен учасник завершує транзакцію (наприклад, фіксує зміни даних). Але якщо хоча б один учасник відповів "не готовий" або сталася помилка, координатор відправляє всім учасникам команду "відкатити" транзакцію.

Трьохфазне узгодження відрізняється тим, що розширює та доповнює двофазне узгодження, додаючи проміжну фазу для підвищення надійності та безпеки. Тут відбуваються всі етапи як і при двохфазному узгодженні лише додається до фази підготовки та фази завершення ще фаза готовності. У двох відомих, повторювальних уже фазах все так само, лише фаза готовності слугує для того, щоб усі учасники повідомили своєю відповідю «готовий» до роботи, координатор посилає всім учасникам повідомлення «готовність» та учасники чекають подальших інструкцій від координатора. І у фазі завершення тепер якщо хоч один учасник відповів "не готовий" або сталася помилка, координатор може вирішити не завершувати транзакцію.

Трьохфазне узгодження дає додатковий рівень захисту, оскільки у разі втрати зв'язку між учасниками вони можуть надійно визначити, чи завершити транзакцію, чи відкотити її. Однак цей протокол є складнішим і має вищі витрати на зв'язок у порівнянні з двофазним узгодженням.

Розподілена транзакція — це транзакція, що охоплює кілька різних вузлів або систем в розподіленому середовищі, таких як бази даних або інші інформаційні системи, розміщені на різних серверах або в різних географічних локаціях. Вона виконується в кілька кроків, координуючи роботу учасників транзакції так, щоб вони діяли як єдиний логічний процес, забезпечуючи цілісність і узгодженість даних.

Розподілені транзакції дозволяють проводити складні операції в розподілених системах, але їх належне виконання вимагає ретельного планування, вибору правильних протоколів узгодження і надійних механізмів компенсації у разі збоїв.

Компенсація в контексті розподілених транзакцій — це механізм, який дозволяє відкотити (або компенсувати) дії, виконані в рамках транзакції, якщо частина або вся транзакція не може бути завершена успішно. Це важлива складова забезпечення узгодженості та цілісності даних у розподілених системах, особливо коли операції зачіпають кілька вузлів або сервісів.

При роботі механіхму компенсації кожен учасник транзакції записує дії, які він виконує в процесі транзакції, щоб можна було їх відкотити в разі необхідності. Якщо транзакція не може бути завершена повністю, або відбувається збій, система викликає компенсаційні дії для кожного учасника. Компенсаційні дії можуть включати відновлення початкового стану ресурсів, скасування внесених змін, або інші дії, необхідні для повернення системи до стану перед транзакцією.

Координатор транзакції (якщо такий є) або самі учасники мають координувати процес компенсації, щоб гарантувати, що всі зміни, які були зроблені в рамках транзакції, будуть успішно відкотені. Часто це включає використання протоколів узгодження, таких як двофазне або трьохфазне узгодження, щоб переконатися, що всі учасники завершують компенсацію узгодженим чином.

Компенсаційні дії повинні бути надійними та добре протестованими, оскільки вони є останньою лінією захисту від збоїв у транзакціях. Вони також повинні бути ідемпотентними, тобто їх виконання має приводити до одного і того ж результату, незалежно від того, скільки разів вони виконуються.

В звичайному житті ми люди стикаються з компенсацією у банківські справі, наприклад, при переказі коштів, якщо транзакція зазнає невдачі, потрібно відкотити всі зміни, включаючи повернення коштів до вихідного балансу або

наприклад, при онлайн оформленні замовлення, якщо виникає проблема, може знадобитися скасувати замовлення та відмовитися від бронювання.

Компенсація є критичним механізмом у розподілених системах, де забезпечення цілісності та узгодженості даних вимагає обробки складних транзакцій між декількома вузлами або сервісами.

Одним з відомих методів для забезпечення цілісності даних і узгодженості є обмін даними між вузлами та постайний моніторинг і виправлення розбіжностей.

Обмін даними між вузлами у мультикомп'ютерній системі є ключовим аспектом забезпечення цілісності даних та узгодженості між різними частинами системи. У системах з топологією решітка вузли зазвичай мають прямі з'єднання з кількома сусідами, що дозволяє встановлювати ефективний обмін даними. Існують принципи обміну: прямі з’єднання, маршрутизація, протоколи обміну даними, буферизація, контроль доступу до мережі, виявлення та випралення помилок, розподілені протоколи консесусу, відстеження стану вузлів, точки відновлення.

У мультикомп'ютерній системі з топологією решітка прямі з'єднання є основним способом взаємодії між сусідніми вузлами. Така топологія являє собою мережу, в якій вузли розміщені у вигляді решітки (часто в лінійній, квадратній або тривимірній решітці), а кожен вузол з'єднаний з кількома сусідами.

Кожен вузол з'єднаний з безпосередньо сусідніми вузлами, що дозволяє передавати дані на невеликі відстані. Топологія решітка може бути одновимірною (лінія), двовимірною (сітка), або тривимірною (решітка), і кожен вузол з'єднаний з сусідами по всіх вимірах. В більшості випадків кожен вузол має симетричну кількість з'єднань з сусідами (наприклад, в квадратній решітці кожен внутрішній вузол з'єднаний з чотирма сусідами). Прямі з'єднання дозволяють використовувати просту і ефективну маршрутизацію даних від одного вузла до іншого. Дані можуть передаватися від вузла до вузла по найкоротшому шляху.

Прямі з'єднання дозволяють використовувати просту і ефективну маршрутизацію даних від одного вузла до іншого. Дані можуть передаватися від

вузла до вузла по найкоротшому шляху. Прямі з'єднання забезпечують високу пропускну здатність для обміну даними між вузлами, що дозволяє ефективно виконувати паралельні обчислення. Завдяки тому, що кожен вузол з'єднаний з кількома сусідами, у разі виходу з ладу одного вузла, дані можуть бути перенаправлені через інші шляхи. Прямі з'єднання дозволяють рівномірно розподілити навантаження між вузлами, що сприяє оптимізації обчислювальних ресурсів.

Маршрутизація в мультикомп'ютерній системі з топологією решітка є процесом вибору оптимального шляху для передачі даних від одного вузла до іншого. У топології решітка вузли розміщені у вигляді лінійної, квадратної або тривимірної решітки, і кожен вузол з'єднаний з кількома сусідами. Маршрутизація має вирішальне значення для забезпечення ефективного та надійного обміну даними.

Маршрутизація має бути стійкою до виходу з ладу вузлів. У випадку збою вузла, маршрутизатори можуть обрати альтернативні маршрути. Алгоритми маршрутизації повинні уникати циклічних маршрутів (петлі), щоб уникнути заплутування передачі даних. Алгоритми маршрутизації можуть намагатися балансувати навантаження між різними шляхами, щоб уникнути перевантаження окремих вузлів або з'єднань. У системі з топологією решітка можна використовувати координати вузлів (наприклад, x, y для двовимірної решітки) для вибору напрямків маршрутизації. Наприклад, вузол може обирати рух за координатою x або y для наближення до цільового вузла.

Існують певні алгоритми маршрутизації в топології решітка. Такі як маршрутизація за координатами, маршрутизація XY (Картезіанська), маршрутизація Wormhole routing, маршрутизація Hot-Potato routing.

При маршрутизації за координатами двовимірній або тривимірній решітці можна використовувати координати вузлів для вибору маршруту. Наприклад, у двовимірній решітці вузол може рухатися вздовж координат x та y, щоб наблизитися до цільового вузла. За маршрутизацією XY (Картезіанська) алгоритм,

який використовує спочатку рух по осі x, а потім по осі y для наближення до цільового вузла. При використані маршрутизації Wormhole routing данні передаються у вигляді потоків-черв'яків, які проходять через мережу використовуючи мімум часу. Метод маршрутизації Hot-Potato routing, полягає в швидкому пересиланні пакетів до наступного вузла, щоб уникнути затримок у чергах.

Маршрутизація в топології решітка є ключовим елементом для забезпечення ефективного та надійного обміну даними між вузлами. Вибір відповідного алгоритму маршрутизації залежить від вимог системи, зокрема пропускної здатності, надійності та затримки.

Протоколи обміну даними визначають правила та механізми, згідно з якими вузли в мультикомп'ютерній системі взаємодіють один з одним для передачі та прийому даних. У системах з топологією решітка протоколи обміну даними мають враховувати структуру мережі та забезпечувати надійність, швидкість та узгодженість передачі даних. Нижче наведено більш детальний опис протоколів обміну даними в такій системі. Вибір протоколу обміну даними залежить від вимог конкретної системи, включаючи швидкість, надійність, аутентифікацію та безпеку.

Буферизація — це процес тимчасового зберігання даних у спеціальному проміжному буфері під час їх передачі між різними частинами системи, наприклад, між вузлами у мультикомп'ютерній системі з топологією решітка. Буферизація виконує кілька важливих функцій, які покращують ефективність та надійність обміну даними.

В системах, де швидкості передачі даних між вузлами різняться, буфери дозволяють вирівняти різницю у швидкості відправки та прийому даних. Буферизація може зменшити затримки у передачі даних, особливо під час тимчасових навантажень або при нестачі ресурсу для обробки даних. Буферизація забезпечує механізм контролю потоку, щоб уникнути перевантаження приймаючого вузла, який може бути більш повільним, ніж відправник.

Буферизація дозволяє даним створювати чергу очікування на передачу або обробку, що особливо корисно в умовах високого навантаження. Буферизація забезпечує можливість асинхронної передачі, коли відправник може продовжувати свою роботу, поки приймач обробляє дані, що вже надійшли.

Бувають різні типи будеризації: вхідні буфери (зберігають дані, що надходять у вузол, поки вони не будуть оброблені або передані далі), вихідні буфери (зберігають дані, що відправляються з вузла, поки вони не будуть передані іншому вузлу) та перехідні буфери (використовуються для тимчасового зберігання даних між вузлами або у вузлі під час їх обробки).

Отже, буферизація допомагає зменшити затримки передачі та забезпечити більш плавний потік даних, що підвищує загальну продуктивність системи. Буфери допомагають відстежувати та контролювати дані, що знижує ризик втрати або пошкодження даних. Буферизація дозволяє більш ефективно використовувати ресурси мережі та обчислювальні потужності вузлів.

Загалом, буферизація є важливим механізмом для покращення ефективності, надійності та узгодженості обміну даними у мультикомп'ютерній системі з топологією решітка.

Контроль доступу до мережі та виявлення і виправлення помилок є ключовими елементами для забезпечення ефективного та надійного обміну даними у мультикомп'ютерних системах з топологією решітка.

Контроль доступу до мережі — це процес регулювання доступу вузлів до мережевих ресурсів, таких як смуга пропускання або канали передачі. Він спрямований на запобігання конфліктам і забезпечення ефективного розподілу мережевих ресурсів. Методи контролю доступу часто включають виявлення конфліктів (колізій) та механізми їх вирішення, наприклад, випадкові затримки перед повторним надсиланням даних.

Виявлення та виправлення помилок є критично важливими для забезпечення цілісності та надійності передачі даних у мережі.

У випадку виявлення помилки або втрати даних можуть застосовуватися механізми повторного відправлення даних, використання дубльованих копій або відновлення з попередніх точок.

Методи виявлення помилок: контрольна сума — checksum, CRC — Cyclic Redundancy Check, парність. Контрольна сума — обчислюється контрольна сума для переданих даних, яка перевіряється отримувачем для виявлення помилок. Cyclic Redundancy Check — циклічний надлишковий код дозволяє виявляти помилки у переданих даних, обчислюючи контрольний код і порівнюючи його з очікуваним. Парність — використовуються біти парності для перевірки цілісності даних (парність може бути парною або непарною).

Деякі системи використовують шифрування та інші методи безпеки для захисту даних під час передачі і запобігання несанкціонованим змінам.

Виявлення та виправлення помилок у поєднанні з ефективним контролем доступу до мережі допомагають забезпечити надійний та ефективний обмін даними між вузлами в мультикомп'ютерній системі з топологією решітка.

Розподілені протоколи консенсусу є основними механізмами забезпечення узгодженості, надійності та безпеки у мультикомп'ютерних системах, зокрема тих, які використовують топологію решітка. У таких системах вузли можуть працювати незалежно один від одного, але при цьому повинні узгоджувати стан системи для досягнення спільної мети. Протоколи консенсусу допомагають вузлам узгоджувати свої дії та стан для досягнення єдиного рішення або стану.

Протоколи консенсусу спрямовані на те, щоб всі вузли системи узгодилися щодо певної пропозиції або стану, наприклад, результату обчислень або оновлення даних. Багато протоколів консенсусу призначені для роботи в умовах ненадійних мереж, збоїв вузлів, мережевих розривів, і повинні забезпечувати узгодженість навіть в таких умовах. В більшості протоколів консенсусу рішення приймається за принципом більшості, що забезпечує високу ймовірність узгодження у випадку збоїв.

У даних протоколах вузли можуть мати різні ролі, наприклад, лідер є основним вузлом, який пропонує рішення або оновлення, і координує їхнє поширення серед інших вузлів, та учасники, які голосують за нього. Він відповідає за збір пропозицій та прийняття рішень щодо них, а також контролює процес вибору консенсусу та надає підсумкові рішення всім учасникам.

Учасники є вузлами, які приймають пропозиції від лідера та голосують за або проти них, а також відповідають за підтримку консенсусу в системі, відправляючи свої голоси лідеру. Учасники виконують рішення, прийняті лідером, якщо вони відповідають вимогам протоколу консенсусу.

Спостерігачі можуть бути вузлами, які не беруть активної участі в голосуванні, але спостерігають за ходом консенсусу. Вони можуть бути залучені для відстеження процесу консенсусу або для діагностики та моніторингу системи.

Деякі системи використовують резервних лідерів, які беруть на себе роль лідера, якщо первинний лідер виходить з ладу або стає недоступним. Це допомагає забезпечити безперервність роботи системи та стабільність процесу консенсусу. Лідери та резервні лідери допомагають забезпечити безперервність роботи системи навіть у разі збоїв. Виділення лідера та резервних лідерів допомагає системам бути більш стійкими до збоїв вузлів.

Ролі вузлів дозволяють розподілити обчислювальні завдання та управління процесом консенсусу між вузлами, оптимізуючи продуктивність.

У мультикомп'ютерних системах з топологією решітка та розподіленими протоколами консенсусу відстеження стану вузлів та використання точок відновлення є ключовими механізмами забезпечення надійності та безперервності роботи системи. Розглянемо кожен з цих аспектів більш детально.

Відстеження стану вузлів в мультикомп'ютерних системах передбачає моніторинг активності, доступності та продуктивності окремих вузлів системи. Це дозволяє забезпечити безперервність обміну даними та стабільність роботи системи.

Вузли можуть відправляти та отримувати сигнали живості або "heartbeats" для підтвердження своєї активності та доступності. Також вони можуть відстежувати власну продуктивність та передавати інформацію про ресурси, наприклад, про завантаження CPU, використання пам'яті та інших ресурсів.

Моніторинг стану вузлів дозволяє виявляти збої та мережеві розриви у вузлах. У випадку збою, система може вжити заходів для вирішення проблеми, наприклад, вибрати новий лідер. Відстеження стану вузлів допомагає оптимізувати маршрутизацію та балансувати навантаження між вузлами, щоб забезпечити ефективність системи.

Точки відновлення (або контрольні точки) є моментами у часі, коли стан системи, включаючи дані та метадані, зберігається для подальшого відновлення у разі збоїв або катастрофічних подій.

Точки відновлення передбачають збереження стану системи у вигляді контрольних точок. Це може включати дані, журнали транзакцій, стан вузлів та інші метадані. Дані точки можуть створюватися періодично або у відповідь на важливі події, такі як завершення транзакцій або змін у системі.

У разі збою або катастрофи, система може відновитися до попередньої точки відновлення, що дозволяє мінімізувати втрату даних та зменшити час простою. Точки відновлення забезпечують узгодженість системи у разі відновлення, допомагаючи зберегти цілісність даних та послідовність операцій, а також вони можуть поєднуватися з механізмами резервного копіювання для створення більш надійної стратегії відновлення.

Відстеження стану вузлів та використання точок відновлення є важливими частинами багатьох сучасних розподілених систем, оскільки вони допомагають забезпечити стабільність, узгодженість і надійність навіть у випадках збоїв або катастрофічних подій. Загалом, обмін даними між вузлами у мультикомп'ютерній системі з топологією решітка є складним, але важливим завданням для забезпечення цілісності, узгодженості та надійності системи.

Протоколи узгодження, часові мітки, розподілені транзакції, обмін даними між вузлами — ці всі методи спрямовані на забезпечення цілісності даних і узгодженості між різними вузлами мультикомп'ютерної системи з топологією решітка.

2.3 Сокети. Зв’язки для пердачі повідомлень в мультикомп’ютерні системі з топології решітка

Мультикомп'ютерні системи з решітчастою топологією є важливими в галузі обчислювальної техніки, особливо в контексті високопродуктивних обчислень та розподілених систем. Одним з ключових аспектів таких систем є ефективна комунікація між вузлами. Сокети грають вирішальну роль у забезпеченні цієї комунікації, дозволяючи обмінюватися повідомленнями між вузлами в мережі. Цей розділ детально розгляне використання сокетів для комунікації у мультикомп'ютерних системах з решітчастою топологією.

Сокет – це програмна абстракція, що представляє кінцеву точку для обміну даними між комп'ютерами в мережі. Існує два основних типи сокетів: TCP-сокети (Stream Sockets) та UDP-сокети (Datagram Sockets).

Stream Sockets забезпечують надійний, орієнтований на з'єднання спосіб передачі даних. Вони гарантують доставку даних у правильному порядку і без втрат.

Datagram Sockets [45] забезпечують ненадійний, орієнтований на повідомлення спосіб передачі даних. Вони швидші, але не гарантують доставку даних у правильному порядку або без втрат.

Кожен вузол у системі створює сокет для прослуховування вхідних з'єднань та окремі сокети для встановлення вихідних з'єднань з іншими вузлами. Це досягається за допомогою стандартних бібліотек, доступних у більшості мов програмування.

Для реальної системи з решітчастою топологією можуть знадобитися додаткові аспекти, такі як обробка помилок, оптимізація продуктивності та

безпека даних. Важливо враховувати масштабованість системи та можливість інтеграції з іншими мережевими технологіями. Тому, реалізація комунікації з використанням сокетів є фундаментальною частиною розробки мультикомп'ютерних систем з решітчастою топологією, але потребує ретельного планування та тестування.

Сокети є критично важливими для ефективної комунікації у мультикомп'ютерних системах з решітчастою топологією. Вони забезпечують надійний та гнучкий спосіб обміну повідомленнями між вузлами, дозволяючи реалізовувати складні обчислювальні задачі і підтримувати високу продуктивність системи. Важливими аспектами є ініціалізація та прив'язка сокетів, встановлення з'єднань, алгоритми маршрутизації, синхронізація та управління трафіком. Використання сокетів дозволяє ефективно організувати обмін даними та забезпечити надійну роботу мультикомп'ютерної системи.

2.4 Алгоритми виявлення та виправлення помилок

Алгоритми виявлення та виправлення помилок мають критичне значення в сучасних обчислювальних системах, включаючи мультикомп'ютерні системи з топологією решітка. Їхня важливість визначається тим, що вони допомагають забезпечити цілісність, надійність та узгодженість даних у системі, що є основою для її ефективної роботи.

Алгоритми виявлення та виправлення помилок гарантують, що дані, які передаються або зберігаються в системі, залишаються цілісними та не зазнають змін через помилки або збої. Використання таких алгоритмів підвищує надійність системи, оскільки вони дозволяють виявляти та виправляти помилки, знижуючи ризик збоїв і втрат даних. Завдяки механізмам виявлення та виправлення помилок система може запобігти корупції даних, забезпечуючи їхню точність і узгодженість.

У розподілених системах алгоритми виявлення та виправлення помилок є особливо важливими, оскільки вони забезпечують узгодженість даних між різними вузлами та точність передачі даних через мережу.

Ці алгоритми допомагають забезпечити доступність системи, оскільки вони виявляють і виправляють помилки в обмінних операціях, тим самим підвищуючи відмовостійкість. Алгоритми виявлення та виправлення помилок можуть допомогти виявити підозрілі зміни або атаки на дані, сприяючи безпеці системи.

У системах з високопродуктивними обчисленнями алгоритми виявлення та виправлення помилок можуть забезпечити безперебійну роботу навіть за наявності високого рівня шумів або помилок. Завдяки таким алгоритмам обмін даними може бути оптимізований, що забезпечує ефективність і продуктивність системи.

Алгоритми виявлення та виправлення помилок є основоположними для сучасних обчислювальних систем, особливо в мультикомп'ютерних системах з розподіленою архітектурою. Вони допомагають підтримувати надійність, узгодженість та доступність даних, що є критично важливим для стабільної та ефективної роботи систем.

Перевірка парності — це метод виявлення помилок, який використовується для перевірки цілісності даних. Він заснований на принципі підрахунку кількості одиничних бітів (бітів зі значенням "1") у даних та додаванні додаткового біту, відомого як біт парності, щоб зробити загальну кількість одиничних бітів парною або непарною, залежно від використовуваного типу парності.

Чекові біти є одним із найпростіших методів виявлення помилок. Він передбачає додавання бітів парності до даних. Біт парності може бути або парним (парність за умовчанням), або непарним, залежно від вимог системи. Парність перевіряється під час передачі та прийому даних. Якщо кількість одиничних біт у даних не відповідає очікуваній парності, це може вказувати на можливу помилку.

Однак цей метод може виявити тільки непарну кількість помилок (наприклад, однобітову помилку) і не може виправити помилки або виявити парні помилки.

Це простий, але ефективний метод виявлення помилок у даних, особливо в середовищах з низьким рівнем шумів або помилок. Однак, у складних і шумних середовищах інші методи виявлення та виправлення помилок можуть бути більш ефективними.

Коди Хеммінга — це коди корекції помилок, які використовують контрольні біти для виявлення та виправлення однобітових помилок у даних. Вони названі на честь Річарда Хеммінга, який розробив цей метод у 1950-х роках. Коди Хеммінга є одними з найвідоміших та найвикористовуваніших алгоритмів для забезпечення цілісності даних у різних сферах, включаючи комп'ютерні мережі та пам'ять комп'ютерів.

Коди Хеммінга передбачають додавання контрольних бітів до даних для забезпечення можливості виявлення та виправлення однобітових помилок. Контрольні біти — це додаткові біти, які додаються до даних у певному кодуванні для виявлення та виправлення помилок. Контрольні біти розміщуються у визначених позиціях у даних і обчислюються за допомогою спеціальних схем, заснованих на бітових операціях.

Контрольні біти додаються до блоку даних під час кодування, щоб забезпечити можливість виявлення та виправлення помилок. Контрольні біти містять інформацію про стан даних, яку можна використовувати для виявлення або виправлення помилок. Контрольні біти розміщуються у певних позиціях у блоках даних згідно з алгоритмом кодування. Це дозволяє здійснювати ефективну перевірку даних.

Під час передачі або зберігання даних контрольні біти обчислюються та перевіряються, щоб виявити будь-які помилки.

За наявності помилки контрольні біти допомагають виявити її та іноді навіть виправити. Контрольні біти є важливою складовою у забезпеченні

цілісності даних та уникненні втрат через помилки під час їх передачі або зберігання. Вони дозволяють системам виявляти та виправляти помилки, що робить їх особливо корисними в критичних застосуваннях, де надійність даних має велике значення.

Контрольні біти відіграють важливу роль у мультикомп'ютерних системах, забезпечуючи цілісність, надійність та точність даних, що передаються між різними вузлами або процесорами. У мультикомп'ютерних системах дані часто передаються через комунікаційні мережі, і важливо виявляти та виправляти помилки, які можуть виникати під час передачі. Ось детальний огляд ролі контрольних бітів у таких системах.

У мультикомп'ютерних системах дуже важливо підтримувати цілісність даних між різними вузлами. Контрольні біти використовуються для гарантування, що дані, отримані вузлом, є такими ж, як і дані, передані відправником.

Це більш надійний метод, де до даних додається серія контрольних бітів або контрольна сума. Вузол, що приймає, обчислює контрольну суму заново і порівнює її з переданою, щоб виявити помилки.

Контрольні біти є невід'ємною частиною комунікаційних протоколів, що використовуються в мультикомп'ютерних системах, таких як Ethernet, TCP/IP та інші. Ці протоколи визначають, як контрольні біти використовуються для управління виявленням помилок, керуванням потоком та кадруванням даних.

Хоча це не основна їхня роль, контрольні біти можуть також сприяти безпеці даних, допомагаючи виявляти несанкціоновані зміни та забезпечуючи, що дані не були змінені під час передачі. У поєднанні з контрольними бітами хеш-функції можуть забезпечити спосіб перевірки цілісності та автентичності даних, додаючи рівень безпеки під час передачі даних.

Вони використовуються в різних алгоритмах, таких як коди корекції помилок (наприклад, коди Хеммінга, коди Ріда-Соломона), методи перевірки цілісності (наприклад, перевірка парності), а також у мережевих протоколах для перевірки правильності передачі даних через мережу.

Коди Хеммінга використовують формулу, яка зв'язує кількість контрольних бітів r, кількість інформаційних бітів k, та загальну кількість бітів n, формула 2.1.

$$N=k+r \text{ і } 2^r \geq k+r+1 \quad (2.1).$$

Ця формула допомагає визначити необхідну кількість контрольних бітів для даного обсягу даних.

Контрольні біти розміщуються у позиціях, які є степенями двійки (тобто, 1, 2, 4, 8, 16, ...). Контрольні біти обчислюються шляхом аналізу бітових позицій даних. Кожен контрольний біт перевіряє певну комбінацію бітових позицій. Під час виявлення помилки контрольні біти дозволяють визначити позицію біта, який зазнав помилки, шляхом обчислення синдрому (поєднання контрольних бітів). Після виявлення помилки код Хеммінга дозволяє виправити однобітову помилку, перевернувши значення біта в зазначеній позиції.

Коди Хеммінга широко використовуються в комп'ютерній пам'яті для забезпечення цілісності даних, у мережевих протоколах та інших застосуваннях. Вони є популярним вибором для виявлення та виправлення помилок у системах з обмеженим обсягом даних та відносно низьким рівнем шумів.

Коди Хеммінга є ефективним методом забезпечення цілісності даних та їхньої надійності у різних обчислювальних системах. Вони прості в реалізації та добре підходять для середовищ з невеликим обсягом помилок.

2.5 Висновки

У другому розділі детально розібрано понятття топології, писано роль топології у мультикомп’ютерні системі, описані частовикористовувані топології, та детально розібране питання топології «Решітка». Також для роботи були розібрані механізми та алгоритми, підтримка цілісності та забезпечення функціоналу мультикомп’ютерної системи. Важливим етапом в даному розділі є

описання поняття сокетів для з'єднання між вузлами та описані алгоритми виявлення та виправлення помилок.

3 ПРАКТИЧНЕ ДОСЛІДЖЕННЯ ПРОМІЖНОГО ПРОГРПМНОГО ЗАБЕЗПЕЧЕННЯ МУТИЛЬТИКОМП'ЮТЕРНИХ СИСТЕМ З ТОПОЛОГІЄЮ «РЕШІТКА»

3.1 Структура програми та доцільність використання мови програмування C++. Бібліотеки для програмного забезпечення

Ця програма, написана мовою програмування C++, призначена для моделювання мережі вузлів, організованих у вигляді решітки. Вона демонструє механізм розповсюдження повідомлень по сітці, використовуючи алгоритм пошуку в ширину (BFS) для ефективної та надійної доставки повідомлень. Застосування BFS забезпечує безпечне переміщення даних без ризику виникнення помилок переповнення стека, що можуть виникнути при використанні глибокої рекурсії. Це особливо важливо для підтримання стабільності та надійності системи в умовах великих мережевих структур.

Програма починає свою роботу з ініціалізації сітки вузлів, де кожен вузол отримує інформацію про своїх сусідів. Така структура дозволяє створити взаємозв'язану мережу, де кожен вузол може обмінюватися повідомленнями з іншими через прямі зв'язки. Користувачеві надається можливість вказати початковий вузол-відправник та кінцевий вузол-отримувач, після чого програма розпочинає процес передачі повідомлення. Алгоритм пошуку в ширину забезпечує оптимальний шлях для передачі повідомлення, обходячи вузли послідовно, починаючи з вузла-відправника.

У процесі передачі повідомлення програма перевіряє кожен вузол на наявність зв'язку з сусідніми вузлами, уникаючи циклів і забезпечуючи, що повідомлення рухається у напрямку до кінцевого отримувача. Якщо повідомлення успішно досягає вузла-отримувача, програма виводить повідомлення про успішну доставку. У разі виникнення проблем із зв'язком або якщо шлях до отримувача заблокований, програма інформує користувача про невдалу спробу доставки.

Такий підхід дозволяє забезпечити високу надійність та передбачуваність роботи мережі.

Крім основної функції передачі повідомлень, програма також може бути розширена для включення додаткових можливостей, таких як маршрутизація на основі пріоритетів або додавання нових вузлів у реальному часі. Це робить її гнучким інструментом для моделювання різноманітних сценаріїв мережевого спілкування та тестування алгоритмів передачі даних. У майбутньому можливі також оновлення, які включатимуть підтримку різних типів повідомлень та адаптацію до змін у топології мережі.

Для виконання було обрано мову програмування саме C++, адже C++ є потужною та універсальною мовою програмування, яка забезпечує високу продуктивність і гнучкість. Незважаючи на деяку складність, вона залишається популярною серед розробників і використовується в різних галузях, від системного програмування до ігрової індустрії. Завдяки своїй продуктивності, C++ є популярним вибором для наукових і чисельних обчислень, де необхідна висока швидкість і точність.

Вона була розроблена в середині 1980-х років як розширення мови C, додавши об'єктно-орієнтовані можливості. C++ використовується для створення різноманітного програмного забезпечення. Спочатку вона називалася "C with Classes" (C з класами) і була розроблена для додавання об'єктно-орієнтованого програмування до C. Пізніше мову перейменували на C++ як символ інкременту, що відображає її розширені можливості.

Мультикомп'ютерні системи, особливо з топологією решітка, використовуються в різних галузях, включаючи обчислювальні кластери, розподілені системи та паралельні обчислення. Вибір мови програмування для розробки таких систем має велике значення, оскільки впливає на продуктивність, масштабованість та надійність. C++ є однією з найбільш доцільних мов для створення мультикомп'ютерних систем з топологією решітка.

C++ забезпечує високу продуктивність завдяки своєму низькорівневому доступу до апаратних ресурсів та ефективному керуванню пам'яттю. У мультикомп'ютерних системах, де обчислювальні ресурси розподілені між багатьма вузлами, швидкість і ефективність виконання коду є критично важливими. C++ дозволяє оптимізувати алгоритми та реалізувати високопродуктивні рішення, що зменшують затримки та підвищують пропускну здатність системи.

Мультикомп'ютерні системи вимагають точного контролю над комунікацією між вузлами, управління потоками та ресурсами. C++ надає розробникам можливість точно контролювати всі аспекти роботи програми, включаючи розподіл пам'яті, синхронізацію потоків та управління ресурсами. Це дозволяє створювати оптимізовані рішення, адаптовані до конкретних вимог системи.

C++ підтримує об'єктно-орієнтоване програмування (ООП), що дозволяє моделювати складні системи за допомогою зрозумілих і структурованих класів. У мультикомп'ютерних системах, де кожен вузол може виконувати окремі завдання або функції, ООП дозволяє створювати модульний і повторно використовуваний код. Це спрощує розробку, тестування та підтримку великих систем.

C++ має велику кількість бібліотек та інструментів, які можуть бути використані для розробки мультикомп'ютерних систем. Наприклад, бібліотеки для паралельних обчислень (таких як MPI, OpenMP) забезпечують високорівневі інтерфейси для комунікації та синхронізації між вузлами. Використання таких бібліотек значно спрощує розробку розподілених систем і підвищує їх ефективність.

Мультикомп’ютерні системи повинні мати можливість легкого масштабування для обробки зростаючих обсягів даних або додавання нових вузлів. Завдяки своїй гнучкості та потужності C++ дозволяє розробляти масштабовані рішення, здатні адаптуватися до змін у структурі системи. Це

особливо важливо для систем із решітчастою топологією, де додавання нових вузлів або розширення мережі має відбуватися без серйозних змін коду.

Тому має сенс використовувати C++ для створення багатокомп'ютерних систем із решітчастою топологією через його високу продуктивність, гнучкість, контроль, підтримку ООП, потужні бібліотеки та інструменти та масштабованість. Ці переваги роблять C++ ідеальною мовою для розробки складних розподілених систем, які потребують ефективного управління ресурсами та високої продуктивності.

У сучасному світі використання мереж стає все більш поширеним, особливо в контексті розподілених систем та паралельних обчислень. У цьому контексті створення ефективної системи комунікації між вузлами мережі та забезпечення надійної доставки повідомлень стає ключовою задачею для розробників програмного забезпечення.

Сучасний розвиток інформаційних технологій супроводжується необхідністю використання мереж для передачі даних та комунікації між різними вузлами. Одним з підходів до створення таких мереж є використання решітки вузлів, де кожен вузол має зв'язки з безпосередніми сусідами. У цьому контексті виникає потреба у розробці ефективної системи доставки повідомлень, що забезпечує надійне спілкування між вузлами мережі.

Створення мережі на основі решітки та системи доставки повідомлень в мові програмування C++ є актуальною задачею. Для цього можна використати різні структури даних та алгоритми, які дозволяють забезпечити ефективну комунікацію між вузлами.

Структура програми, розробленої для цієї задачі, включає в себе оголошення класу "Вузол", який представляє окремий вузол у мережі. Кожен вузол має можливість підключатися до сусідніх вузлів та відправляти повідомлення. Для забезпечення ефективної доставки повідомлень може використовуватися алгоритм пошуку в ширину (BFS), який дозволяє швидко знаходити шлях між вузлами.

Функціональні аспекти програми включають в себе процес підключення вузлів до мережі, надсилання повідомлень та обробку результатів доставки. Кожен вузол може підключатися до інших вузлів та передавати їм повідомлення, що дозволяє створювати мережі з різною топологією та складністю.

Важливість даної теми полягає у можливості створення ефективних та надійних мереж для передачі даних та комунікації між вузлами. Розуміння принципів роботи мереж та їхніх алгоритмів є ключовим для розробки високоякісного програмного забезпечення, що може бути використане у різних сферах діяльності, від інформаційних технологій до наукових досліджень.

Код для програми включає кілька заголовкових файлів стандартної бібліотеки для введення/виведення — iostream, векторів — vector, управління пам'яттю — memory, черги — queue та неупорядкованих множин — unordered set.

Взагалі бібліотеки в C++ та програмуванні загалом відіграють критичну роль у розробці програмного забезпечення. Вони забезпечують заздалегідь написаний та готовий код, який розробники можуть використовувати для виконання загальних завдань, що допомагає спростити розробку, покращити якість коду та підвищити продуктивність. Бібліотеки надають багаторазовий код для виконання поширених функцій, що заощаджує час розробників, оскільки їм не потрібно писати код з нуля. Наприклад, у стандартній бібліотеці C++ є контейнери (такі як std::vector std::map), алгоритми (такі як std::sort) та утиліти (такі як розумні вказівники).

Бібліотеки часто містять високоефективний код, який зазвичай працює краще, ніж власні реалізації. Використання цих бібліотек може покращити продуктивність додатків.Вони сприяють стандартизації і це гарантує, що код, написаний із використанням стандартних бібліотек, є портативним та легко зрозумілим та підтримуваним іншими розробниками.

Бібліотеки абстрагують складні деталі реалізації, дозволяючи розробникам зосередитися на розв'язанні вищого рівня проблем. Наприклад, у стандартній

бібліотеці шаблонів (STL) в C++ абстрагуються складні структури даних та алгоритми.

Використання бібліотек прискорює процес розробки. Наприклад, замість того, щоб реалізовувати власні структури даних чи алгоритми, можна скористатися тими, що надаються STL. Добре відомі бібліотеки ретельно тестуються та відлагоджуються, крім того, використання таких бібліотек знижує ймовірність помилок у самотужки написаному коді.

Для успішної та оптимізованої роботи програмного забезпечення мультикомп'ютерної системи з топологією решітка використовувалися такі бібліотеки як: iostream, vector, memory, queue, unordered_map, unordered_set, stack.

Бібліотека iostream призначена для надання можливості для проведення операцій введення та виведення даних. Зазвичай використовується для читання зі стандартного вводу (std::cin) і запису до стандартного виводу (std::cout) та стандартної помилки (std::cerr). Без цієї бібліотеки не обходиться майже жоден код, адже кожному програмісту потрібно бачити роботу коду та його успішний або негативний результат. Ключовими елементами використання цієї бібліотекати можуть бути такі коди std::cin — стандартний потік вводу. std::cout — стандартний потік виводу. std::cerr — стандартний потік помилок. Прикладом коду де використовується цей код є звичайна програма Hello, world:

```
std::cout << "Hello, World!" << std::endl;
std::cin >> userInput;
```

В програмному забезпечені мультикомп'ютерні системі з топологією решітка дана бібліотека використовується та для зчитування розміру сітки, ідентифікатора відправника та ідентифікатора одержувача від користувача для друку повідомлень на консоль, включаючи макет сітки, статус доставки та шлях.

```
#include <iostream>
// ...
int main() {
int gridSize;
```

```
std::cout << "Enter the grid size (N for an N x N grid): ";
std::cin >> gridSize;
// ...
std::cout << "Grid Layout:" << std::endl; printGrid(grid, gridSize);
// ... std::cout << "Enter the ID of the sender node: ";
std::cin >> senderId;
std::cout << "Enter the ID of the receiver node: ";
std::cin >> receiverId;
}.
```

Бібліотека vector реалізує контейнер std::vector, який є динамічним масивом, що може збільшуватись у розмірі. Використовується, коли потрібен масив зі змінним розміром, який буде забезпечувати випадковий доступ до елементів. Приклад коду на мові програмування C++:

```
std::vector<int> numbers = {1, 2, 3, 4, 5};
numbers.push_back(6);.
```

В програмному забезпечені мультикомп’ютерні системі з топологією решітка дана бібліотека використовується для створення 2D сітки вузлів та для зберігання сусідніх вузлів кожного елемента.

```
#include <vector>
// ...
std::vector<std::vector<std::shared_ptr<Node>>> grid(gridSize, std::vector<std::shared_ptr<Node>>(gridSize));
std::vector<std::shared_ptr<Node>> neighbors;
```

Бібліотека memory використовується для надання засобів для керування динамічною пам'яттю, зокрема розумними вказівниками типу std::shared_ptr та

std::unique_ptr і використовується для автоматичного та безпечного для винятків керування динамічно виділеною пам'яттю. Приклад:

```
std::shared_ptr<int> p = std::make_shared<int>(10);
```

В програмному забезпечені мультикомп’ютерні системі з топологією решітка дана бібліотека використовується для автоматичного керування часом життя вузлів, гарантуючи їх звільнення, коли вони більше не потрібні.

```
#include <memory>
// ... int nodeId = 1;
for (int i = 0; i < gridSize; i++) {
for (int j = 0; j < gridSize; j++) {
grid[i][j] = std::make_shared<Node>(nodeId++);
}
}.
```

Бібліотека queue еалізує адаптер контейнера std::queue, який надає функціональність черги FIFO (first-in, first-out). Використовується, коли потрібна структура даних черги для обробки елементів у порядку їх додавання. Прикладом може слугувати звичайни код:

```
std::queue<int> q;
q.push(1);
q.push(2);
int front = q.front();
q.pop();
```

В програмному забезпечені мультикомп’ютерні системі з топологією решітка дана бібліотека використовується в алгоритмі BFS для обходу вузлів рівень за рівнем.

```
#include <queue>
//...
std::queue<std::shared_ptr<Node>> queue;
queue.push(shared_from_this());
```

Бібліотека unordered_map реалізує контейнер std::unordered_map, який є хеш-таблицею, що зберігає пари ключ-значення з середньою складністю O(1) для пошуку. Використовується, коли потрібен швидкий пошук, вставка та видалення з використанням унікальних ключів.

В програмному забезпечені мультикомп'ютерні системі з топологією решітка дана бібліотека використовується для відстеження попередників вузлів під час BFS для реконструкції шляху.

```
#include <unordered_map>
// ...
std::unordered_map<int, int> predecessor;
predecessor[id] = -1;
```

Бібліотека unordered_set створює контейнер std::unordered_set, який є хеш-таблицею, що зберігає унікальні елементи з середньою складністю пошуку O. Використовується, коли потрібна колекція унікальних елементів і швидкий пошук. Прикладом коду є:

```
std::unordered_set<int> uniqueNumbers = {1, 2, 3};
uniqueNumbers.insert(4);
```

В програмному забезпечені мультикомп'ютерні системі з топологією решітка дана бібліотека використовується для відстеження відвіданих вузлів під час BFS, щоб уникнути багаторазової обробки одного і того ж вузла.

```
#include <unordered_set>
// ...
std::unordered_set<int> visited;
visited.insert(id);
```

Бібліотека stack Реалізує адаптер контейнера std::stack, який забезпечує функціональність стеку LIFO (last-in, first-out). Використовується, коли потрібна стекова структура даних для обробки елементів у зворотному порядку їх додавання. Наочним прикладом може бути:

```
std::stack<int> s;
s.push(1);
```

```
s.push(2);
int top = s.top();
s.pop();
```

В програмному забезпечені мультикомп’ютерні системі з топологією решітка дана бібліотека використовується для зберігання і друку шляху від вихідного вузла до цільового.

```
#include <stack>
// ...
std::stack<int> path;
while (current != -1) { path.push(current); current = predecessor.at(current);
}.
```

Коротко підсумовуючи, iostream використовується для читання з консолі та запису до неї. Vector використовується для зберігання сітки вузлів у динамічному масиві. Memory використовується для керування розумними вказівниками на об'єкти Node. Queue використовується в алгоритмі BFS для обходу вузлів. Unordered_map використовується для відстеження попередників вузлів під час BFS. Unordered_set використовується для відстеження відвіданих вузлів у BFS. Stack використовується для зберігання шляху від вузла-джерела до вузла-приймача для друку.

Ці бібліотеки в сукупності надають необхідні структури даних та функції вводу/виводу для реалізації сітки, виконання обходу BFS та ефективного керування динамічною пам'яттю.

3.2 Логіка роботи проміжного програмного забезпечння мультикомп’ютерної системи з топологією «Решітка»

Програма моделює багатокомп'ютерну систему, організовану за сітковою топологією. Кожен комп'ютер (вузол) може надсилати повідомлення іншим вузлам, використовуючи Breadth-First Search (BFS), щоб забезпечити доставку

повідомлень без переповнення стеку рекурсії. Розмір сітки визначається користувачем, а вузли з'єднуються зі своїми безпосередніми сусідами. Програма також надає візуальне представлення сітки і відображає шлях, який проходить повідомлення від відправника до одержувача.

На початку програми увімкнуто декілька часто використовуваних бібліотек: стандартні бібліотеки, необхідні для деяких функцій, таких як ввід/вивід, та бібліотеки структур даних і інтелектуальних вказівників. В класі «Node» чітко визначено, що має робити кожен вузол у мережі з топологією «Решітка» — надавати унікальний ID вузла для ідентифікації, бути проінформованим про список своїх сусідніх вузлів та уміти підключати і надсилати повідомлення до потрібниз вузлів.

Метод «connect» дозволяє вузлу встановити з'єднання з іншим вузлом, додавши його до списку своїх сусідів. Наступним методом який доаоможе програмі надсилати повідомлення є «sendMessage» метод. Цей метод написаний для надсилання повідомлення з одного вузла-комп'ютера до іншого використовуючи пошук у ширину, у справжній системі програмісти використовують сокети для цього. Метод пошуку у ширину дозволяє уникнути рекурсії, тобто нескінченності програми та перенавантаження буферу та пам'яті.

Отже, як цей шматок програми працює, перш за все ініціалізується черга та починається створення черги. Під час роботи відстежуються відпрацьованні вузли, щоб запобігти обробці одних і тих же вузлів, що веде за собою нескінченний цикл. Наступним важливим етапом є збереження шляху для запам'ятовування пройдених вузлів. Якщо потрібний вузол був знайдений, тоді метод підтверджує, що повідомлення, було доставленно та відображає шлях повідомлення з вузла-відправника до кінцевого вузла-отримувача. Але якщо потрібного вузла не було знайдено, то буде відображатися повідомлення для користувача про неможливість відправлення повідомлення.

Подальшою на черзі є функція «printPath», яка існує для виведення шляху повідомлення з вузла-відправника до вузла-отримувача з допомогою

запам’ятовування попередніх кроків. Функція «getId» існує для простого повернення індитифікатора вузла.

З допоміжними елементами коду завершено, далі працює основна частина коду. Першим елементом основної частини є введення розміру потрібного для роботи з системою і обмежень у розмірі вузлів дана система немає. Для вводу в консолі одразу після запуску програми з’являється стрічка «Enter the size of grid (N x N grid): ». У даному проміжному програмному забезпечені мультикомп’ютерної системи з топологією «Решітка» у коді створюється двовимірна сітка з вузлів і присвоюється кожному вузлу свій унікальний ідентифікатор. Після цієї невеличної процедури відбувається підключення вузлів один до одного, у кожного вузла є 4 підключення, тобто вузол має 4 сусідів з кожної сторони: зверху, знизу, зліва, справа.

Для зручності користувача програма налаштована на виведення остаточної сітки, щою далі користувач зміг обрати з якого вузла відправити повідомлення і який вузол отримає дане повідомлення. Цю дію користувач також робить у консолі одразу після виведення всіх доступних вузлів. Далі програма працює настуним чином, знаходить вузол-відправник і використовує функцію «sendMessage», щоб доставити повідомлення до вузла-одержувача. Від одержувача до відправника знаходиться вузол-одержувача і використовує «sendMessage» для відправки повідомлення назад до вузла-відправника.

На данному етапі повідомлення є максимально простими та текстовими, від вузла-відправника до отримувача надсилається повідомлення з текстом «Hello, target node!» та від вузла-отрмувача до вузла-відправника надходить повідомлення «Hello, sender node!».

Також на випадок, якщо вузол-відправника або одержувача не знайдено, виводиться повідомлення про помилку і завершується робота.

Ця програма ефективно створює комунікаційну мережу вузлів у сітці, дозволяючи надсилати повідомлення між будь-якими двома вузлами.

3.6. Результати роботи проміжного програмного забезпечння мультикомп’ютерної системи з топологією «Решітка» та експеременти

Для перевірки програми потрібно провести декілька експерементів з проміжним програмним забезпеченням мультикомп’ютерної системи з топологією «Решітка».

Отже, проведемо перший експеремент для програми. Розмір самої системи обремемо 12 і маємо в результаті 144 вузла готових до роботи з повідомленнями. Вузлом який буде відправляти повідомлення стає 16 вузол, в вузлом-отримувачем буде 144 вузол. І отримаємо наступний результат в консолі рисунок 3.1.

```
Microsoft Visual Studio Debu
Enter the size of grid (N x N grid): 12
Grid Layout:
1       2       3       4       5       6       7       8       9   10      11      12
13      14      15      16      17      18      19      20      21  22      23      24
25      26      27      28      29      30      31      32      33  34      35      36
37      38      39      40      41      42      43      44      45  46      47      48
49      50      51      52      53      54      55      56      57  58      59      60
61      62      63      64      65      66      67      68      69  70      71      72
73      74      75      76      77      78      79      80      81  82      83      84
85      86      87      88      89      90      91      92      93  94      95      96
97      98      99      100     101     102     103     104     105 106     107     108
109     110     111     112     113     114     115     116     117 118     119     120
121     122     123     124     125     126     127     128     129 130     131     132
133     134     135     136     137     138     139     140     141 142     143     144


Enter the ID of the sender node: 16
Enter the ID of the receiver node: 144

Message delivered to target node 144
Path: 16 -> 28 -> 40 -> 52 -> 64 -> 76 -> 88 -> 100 -> 112 -> 124 -> 136 -> 137 -> 138 -> 139 -> 140 -> 141 -> 142 -> 143 -> 144

Message delivered to target node 16
Path: 144 -> 132 -> 120 -> 108 -> 96 -> 84 -> 72 -> 60 -> 48 -> 36 -> 24 -> 23 -> 22 -> 21 -> 20 -> 19 -> 18 -> 17 -> 16

C:\Users\julia\source\repos\con2\x64\Debug\con2.exe (process 11536) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when deb
ugging stops.
Press any key to close this window . . .
```

Рисунок 3.1 — Результати першого експерименту проміжного програмного забезпечення мультикомп’ютерної системи з топологіє «Решітка»

На рисунку 3.1 можна побачити всю систему мультикомп’ютерної системи, вузол-відправник та вузол-отримувач, а також досить важливим є рядок зоображення шляху повідомлення, адже саме в таким спосіб ми можемо відслідкувати чи дотримана структура відправлення повідомлення за топологією «Решітка».

На рисунку 3.2 представлено алгоритм роботи першого експерименту проміжного програмного забезпечення мультикомп’ютерної системи з топологіє «Решітка».

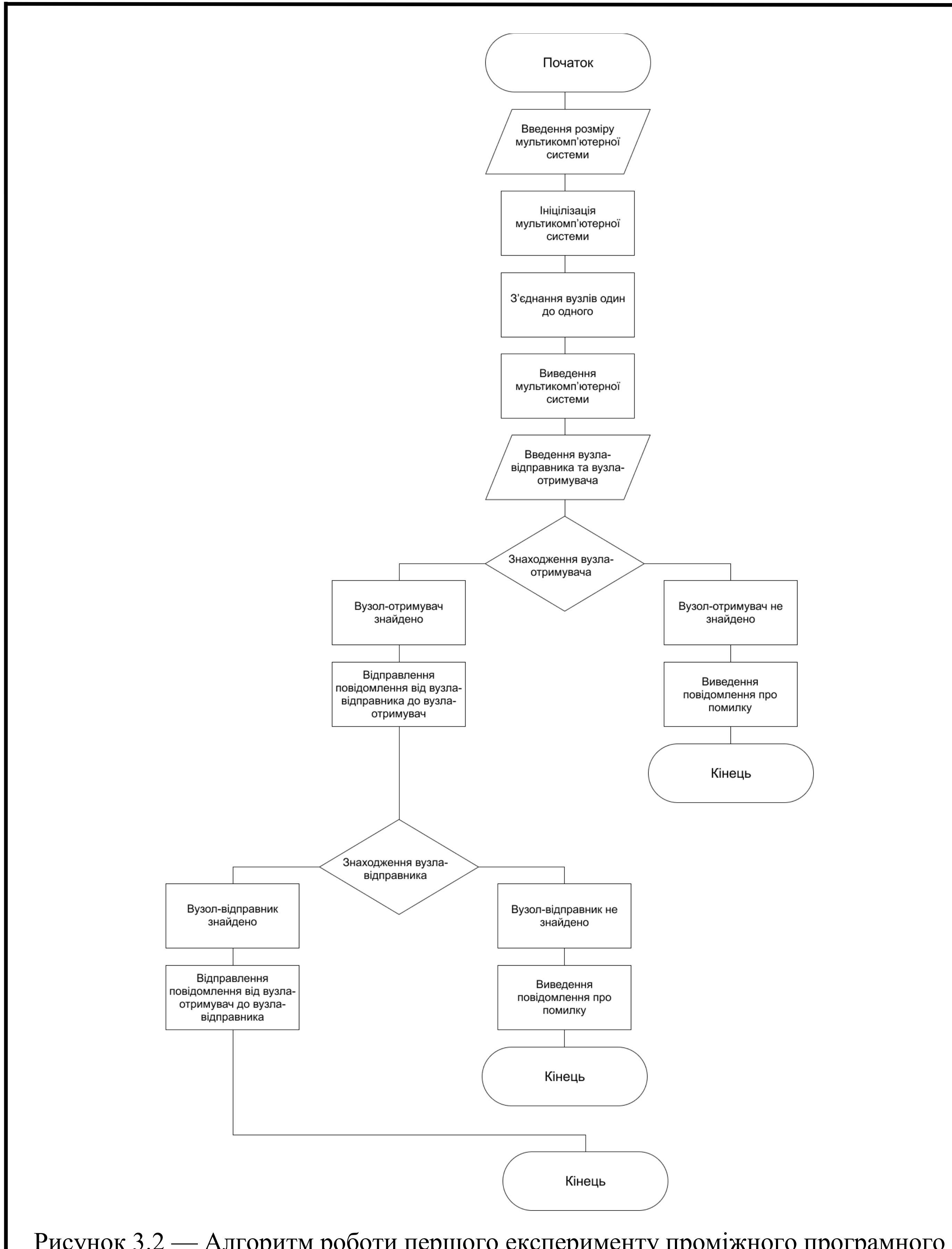


Рисунок 3.2 — Алгоритм роботи першого експерименту проміжного програмного забезпечення мультикомп'ютерної системи з топологіє «Решітка»

Для експерименту номер два, щоб переконатися, що все працює належно і з великими мультикомп'ютерними системами було обрано розмір системи 30, де є доступних для комунікації 900 вузлів. На рисунку 3.3 можна побачити усі доступні вузли для обміну повідомленнями. І після виведених вузлів є можна спостерігати, що повідомлення було успішно доставлене та вказаний шлях, який було пройдено в двох напрямках.

```
Enter the size of grid (N x N grid): 30
Grid Layout:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60
61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90
91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120
121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150
151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180
181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210
211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240
241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270
271 272 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287 288 289 290 291 292 293 294 295 296 297 298 299 300
301 302 303 304 305 306 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323 324 325 326 327 328 329 330
331 332 333 334 335 336 337 338 339 340 341 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359 360
361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377 378 379 380 381 382 383 384 385 386 387 388 389 390
391 392 393 394 395 396 397 398 399 400 401 402 403 404 405 406 407 408 409 410 411 412 413 414 415 416 417 418 419 420
421 422 423 424 425 426 427 428 429 430 431 432 433 434 435 436 437 438 439 440 441 442 443 444 445 446 447 448 449 450
451 452 453 454 455 456 457 458 459 460 461 462 463 464 465 466 467 468 469 470 471 472 473 474 475 476 477 478 479 480
481 482 483 484 485 486 487 488 489 490 491 492 493 494 495 496 497 498 499 500 501 502 503 504 505 506 507 508 509 510
511 512 513 514 515 516 517 518 519 520 521 522 523 524 525 526 527 528 529 530 531 532 533 534 535 536 537 538 539 540
541 542 543 544 545 546 547 548 549 550 551 552 553 554 555 556 557 558 559 560 561 562 563 564 565 566 567 568 569 570
571 572 573 574 575 576 577 578 579 580 581 582 583 584 585 586 587 588 589 590 591 592 593 594 595 596 597 598 599 600
601 602 603 604 605 606 607 608 609 610 611 612 613 614 615 616 617 618 619 620 621 622 623 624 625 626 627 628 629 630
631 632 633 634 635 636 637 638 639 640 641 642 643 644 645 646 647 648 649 650 651 652 653 654 655 656 657 658 659 660
661 662 663 664 665 666 667 668 669 670 671 672 673 674 675 676 677 678 679 680 681 682 683 684 685 686 687 688 689 690
691 692 693 694 695 696 697 698 699 700 701 702 703 704 705 706 707 708 709 710 711 712 713 714 715 716 717 718 719 720
721 722 723 724 725 726 727 728 729 730 731 732 733 734 735 736 737 738 739 740 741 742 743 744 745 746 747 748 749 750
751 752 753 754 755 756 757 758 759 760 761 762 763 764 765 766 767 768 769 770 771 772 773 774 775 776 777 778 779 780
781 782 783 784 785 786 787 788 789 790 791 792 793 794 795 796 797 798 799 800 801 802 803 804 805 806 807 808 809 810
811 812 813 814 815 816 817 818 819 820 821 822 823 824 825 826 827 828 829 830 831 832 833 834 835 836 837 838 839 840
841 842 843 844 845 846 847 848 849 850 851 852 853 854 855 856 857 858 859 860 861 862 863 864 865 866 867 868 869 870
871 872 873 874 875 876 877 878 879 880 881 882 883 884 885 886 887 888 889 890 891 892 893 894 895 896 897 898 899 900

Enter the ID of the sender node: 128
Enter the ID of the receiver node: 771

Message delivered to target node 771
Path: 128 -> 158 -> 188 -> 218 -> 248 -> 278 -> 308 -> 338 -> 368 -> 398 -> 428 -> 458 -> 488 -> 518 -> 548 -> 578 -> 608 -> 638 -> 668 -> 698 -> 728 -> 758 -> 759 -> 760 -> 761 -> 762 -> 763 -> 764 -> 765 -> 766 -> 767 -> 768 -> 769 -> 770 -> 771

Message delivered to target node 128
Path: 771 -> 741 -> 711 -> 681 -> 651 -> 621 -> 591 -> 561 -> 531 -> 501 -> 471 -> 441 -> 411 -> 381 -> 351 -> 321 -> 291 -> 261 -> 231 -> 201 -> 171 -> 141 -> 140 -> 139 -> 138 -> 137 -> 136 -> 135 -> 134 -> 133 -> 132 -> 131 -> 130 -> 129 -> 128

C:\Users\julia\source\repos\con2\x64\Debug\con2.exe (process 20996) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Рисунок 3.3 —Результати першого експерименту проміжного програмного забезпечення мультикомп'ютерної системи з топологіє «Решітка»

Для перевірки роботи проміжного програмного забезпечення було проведено і дослід, коли вузол-відправник хоче передати повідомлення на неіснуючий вузол-тримувач. В таком разі, з'являється повідомлення про те, що повідомлення не було надіслано, та отримувача не існує.

На рисунку 3.4 можна побачити на малюнку результати першого експерименту проміжного програмного забезпечення мультикомп'ютерної системи з топологіє «Решітка».

```
Microsoft Visual Studio Debug
96   97   98   99   100  101  102  103  104  105  106  107  108  109  110  111  112  113  114
115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133
134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152
153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171
172  173  174  175  176  177  178  179  180  181  182  183  184  185  186  187  188  189  190
191  192  193  194  195  196  197  198  199  200  201  202  203  204  205  206  207  208  209
210  211  212  213  214  215  216  217  218  219  220  221  222  223  224  225  226  227  228
229  230  231  232  233  234  235  236  237  238  239  240  241  242  243  244  245  246  247
248  249  250  251  252  253  254  255  256  257  258  259  260  261  262  263  264  265  266
267  268  269  270  271  272  273  274  275  276  277  278  279  280  281  282  283  284  285
286  287  288  289  290  291  292  293  294  295  296  297  298  299  300  301  302  303  304
305  306  307  308  309  310  311  312  313  314  315  316  317  318  319  320  321  322  323
324  325  326  327  328  329  330  331  332  333  334  335  336  337  338  339  340  341  342
343  344  345  346  347  348  349  350  351  352  353  354  355  356  357  358  359  360  361

Enter the ID of the sender node: 48
Enter the ID of the receiver node: 390

Message not delivered
Receiver node not found!

C:\Users\julia\source\repos\con2\x64\Debug\con2.exe (process 7560) exited with code 1.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Рисунок 3.4 — Результати першого експерименту проміжного програмного забезпечення мультикомп’ютерної системи з топологіє «Решітка»

Для експерименту номер три вирішено додадати таймер, щоб відслідкувати час, який система витрачає для відправлення повідомлення між потрібними вузлами.

Перш за все, для успішності даної ідеї потрібно провести певні модифікації в уже наявному коді. Спочатку потрібно підключити ще одну бібліотеку.

Бібліотека «chrono» в C++ є частиною стандартної бібліотеки C++11, розробленої для роботи з датами та часом у типобезпечний та ефективний спосіб. Вона надає набір інструментів для роботи з часовими точками, тривалістю та годинниками. Дана бібліотека дає можливість працювати з декількома типами годинників для вимірювання часу, тому програміст може вибрати для себе потрібний. Клас `std::chrono::system_clock` представляє загальносистемний реальний час. Цей годинник підходить для вимірювання реального часу і може використовуватися для отримання поточного часу. Клас `std::chrono::steady_clock` представляє собою монотонний годинник, який гарантовано ніколи не коригується. Данний годинник ідеально підходить для вимірювання інтервалів. Клас `std::chrono::high_resolution_clock` — годинник з найменшим можливим періодом тіку. Він може бути синонімом steady_clock або system_clock. Клас std::chrono::duration — це шаблонний клас,

який використовується для представлення тривалості часу, такої як мілісекунди, секунди, хвилини тощо.

Отже, бібліотека chrono є потужним та гнучким способом обробки операцій, пов'язаних з часом у мові програмування C++. Забезпечуючи типобезпечні часові точки, тривалості та годинники, вона дозволяє розробникам та програмістам виконувати точні вимірювання часу, отримувати поточний час та легко маніпулювати тривалістю. Незалежно від того, чи потрібно виміряти час виконання функції, обробити затримки або працювати з реальним часом, chrono надає необхідні інструменти для виконання цих завдань.

В модифікації цього коду до функції «sendMessage» додався шматок коду: `auto startTime = std::chrono::high_resolution_clock::now().` Таймер в даному коді починається одразу працювати, але не виводить системний час, а поточний для обчислення часу.

Таймер зупитяється коли програмне забезпечення знайшло вузол-отримувач, тому додано наступний код:

```
if (currentNode->id == targetNodeId) {
auto endTime = std::chrono::high_resolution_clock::now(); // End timing
std::chrono::duration<double> duration = endTime - startTime;
std::cout << "\nMessage delivered to target node " << targetNodeId << std::endl;
printPath(predecessor, targetNodeId);
std::cout << "Time taken to deliver the message: " << duration.count() << " seconds" << std::endl;
return;
}
```

З даними змінами результат роботи проміжного програмного забезпечення виглядає наступним чином, що після виведення шляху повідомлення в консолі перед є наступний рядок: «Time taken to deliver the message:».

```
Microsoft Visual Studio Debu
Enter the size of grid (N x N grid): 20
Grid Layout:
1    2    3    4    5    6    7    8    9    10   11   12   13   14   15   16   17   18   19   20
21   22   23   24   25   26   27   28   29   30   31   32   33   34   35   36   37   38   39   40
41   42   43   44   45   46   47   48   49   50   51   52   53   54   55   56   57   58   59   60
61   62   63   64   65   66   67   68   69   70   71   72   73   74   75   76   77   78   79   80
81   82   83   84   85   86   87   88   89   90   91   92   93   94   95   96   97   98   99   100
101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120
121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140
141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160
161  162  163  164  165  166  167  168  169  170  171  172  173  174  175  176  177  178  179  180
181  182  183  184  185  186  187  188  189  190  191  192  193  194  195  196  197  198  199  200
201  202  203  204  205  206  207  208  209  210  211  212  213  214  215  216  217  218  219  220
221  222  223  224  225  226  227  228  229  230  231  232  233  234  235  236  237  238  239  240
241  242  243  244  245  246  247  248  249  250  251  252  253  254  255  256  257  258  259  260
261  262  263  264  265  266  267  268  269  270  271  272  273  274  275  276  277  278  279  280
281  282  283  284  285  286  287  288  289  290  291  292  293  294  295  296  297  298  299  300
301  302  303  304  305  306  307  308  309  310  311  312  313  314  315  316  317  318  319  320
321  322  323  324  325  326  327  328  329  330  331  332  333  334  335  336  337  338  339  340
341  342  343  344  345  346  347  348  349  350  351  352  353  354  355  356  357  358  359  360
361  362  363  364  365  366  367  368  369  370  371  372  373  374  375  376  377  378  379  380
381  382  383  384  385  386  387  388  389  390  391  392  393  394  395  396  397  398  399  400


Enter the ID of the sender node: 45
Enter the ID of the receiver node: 367

Message delivered to target node 367
Path: 45 -> 65 -> 85 -> 105 -> 125 -> 145 -> 165 -> 185 -> 205 -> 225 -> 245 -> 265 -> 285 -> 305 -> 325 -> 345 -> 365 -> 366 -> 367
Time taken to deliver the message: 0.0013431 seconds

Message delivered to target node 45
Path: 367 -> 347 -> 327 -> 307 -> 287 -> 267 -> 247 -> 227 -> 207 -> 187 -> 167 -> 147 -> 127 -> 107 -> 87 -> 67 -> 47 -> 46 -> 45
Time taken to deliver the message: 0.0013016 seconds

C:\Users\julia\source\repos\app with timer\x64\Debug\app with timer.exe (process 5192) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Рисунок 3.5 — Результат першого запуску модифікованого проміжного програмного забезпечення мультикомп'ютерної системи з топологіє «Решітка»

При першому запуску програми з таймером на рисунку 3.5 можемо бачити наступні результати, що відправлення повідомлення від вузла-відправника до вузла-отримувача займає трошки більше часу, а ніж надсилання відповіді від вузла-отримувача до вузла-відправника.

```
Microsoft Visual Studio Debu
Enter the size of grid (N x N grid): 14
Grid Layout:
1    2    3    4    5    6    7    8    9    10   11   12   13   14
15   16   17   18   19   20   21   22   23   24   25   26   27   28
29   30   31   32   33   34   35   36   37   38   39   40   41   42
43   44   45   46   47   48   49   50   51   52   53   54   55   56
57   58   59   60   61   62   63   64   65   66   67   68   69   70
71   72   73   74   75   76   77   78   79   80   81   82   83   84
85   86   87   88   89   90   91   92   93   94   95   96   97   98
99   100  101  102  103  104  105  106  107  108  109  110  111  112
113  114  115  116  117  118  119  120  121  122  123  124  125  126
127  128  129  130  131  132  133  134  135  136  137  138  139  140
141  142  143  144  145  146  147  148  149  150  151  152  153  154
155  156  157  158  159  160  161  162  163  164  165  166  167  168
169  170  171  172  173  174  175  176  177  178  179  180  181  182
183  184  185  186  187  188  189  190  191  192  193  194  195  196


Enter the ID of the sender node: 3
Enter the ID of the receiver node: 178

Message delivered to target node 178
Path: 3 -> 17 -> 31 -> 45 -> 59 -> 73 -> 87 -> 101 -> 115 -> 129 -> 143 -> 157 -> 171 -> 172 -> 173 -> 174 -> 175 -> 176 -> 177 -> 178
Time taken to deliver the message: 0.0014508 seconds

Message delivered to target node 3
Path: 178 -> 164 -> 150 -> 136 -> 122 -> 108 -> 94 -> 80 -> 66 -> 52 -> 38 -> 24 -> 10 -> 9 -> 8 -> 7 -> 6 -> 5 -> 4 -> 3
Time taken to deliver the message: 0.0008879 seconds

C:\Users\julia\source\repos\app with timer\x64\Debug\app with timer.exe (process 22576) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Рисунок 3.6 — Результат другого запуску модифікованого проміжного програмного забезпечення мультикомп'ютерної системи з топологією «Решітка»

При другому запуску програми на рисунку 3.6 було вирішено зменшити розмір системи з 20 х 20 на 14 х 14. Результат попереднього випробування повторився і від вузла-відправника до вузла-отримувача надсилання повідослення займає більше часу, а ніж надсилання відповіді від вузла-отримувача до вузла-відправника.

```
Microsoft Visual Studio Debu
Enter the size of grid (N x N grid): 20
Grid Layout:
1    2    3    4    5    6    7    8    9    10   11   12   13   14   15   16   17   18   19   20
21   22   23   24   25   26   27   28   29   30   31   32   33   34   35   36   37   38   39   40
41   42   43   44   45   46   47   48   49   50   51   52   53   54   55   56   57   58   59   60
61   62   63   64   65   66   67   68   69   70   71   72   73   74   75   76   77   78   79   80
81   82   83   84   85   86   87   88   89   90   91   92   93   94   95   96   97   98   99   100
101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120
121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140
141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160
161  162  163  164  165  166  167  168  169  170  171  172  173  174  175  176  177  178  179  180
181  182  183  184  185  186  187  188  189  190  191  192  193  194  195  196  197  198  199  200
201  202  203  204  205  206  207  208  209  210  211  212  213  214  215  216  217  218  219  220
221  222  223  224  225  226  227  228  229  230  231  232  233  234  235  236  237  238  239  240
241  242  243  244  245  246  247  248  249  250  251  252  253  254  255  256  257  258  259  260
261  262  263  264  265  266  267  268  269  270  271  272  273  274  275  276  277  278  279  280
281  282  283  284  285  286  287  288  289  290  291  292  293  294  295  296  297  298  299  300
301  302  303  304  305  306  307  308  309  310  311  312  313  314  315  316  317  318  319  320
321  322  323  324  325  326  327  328  329  330  331  332  333  334  335  336  337  338  339  340
341  342  343  344  345  346  347  348  349  350  351  352  353  354  355  356  357  358  359  360
361  362  363  364  365  366  367  368  369  370  371  372  373  374  375  376  377  378  379  380
381  382  383  384  385  386  387  388  389  390  391  392  393  394  395  396  397  398  399  400


Enter the ID of the sender node: 50
Enter the ID of the receiver node: 381

Message delivered to target node 381
Path: 50 -> 70 -> 90 -> 110 -> 130 -> 150 -> 170 -> 190 -> 210 -> 230 -> 250 -> 270 -> 290 -> 310 -> 330 -> 350 -> 370 -> 390 -> 389 -> 388 -> 387 -> 386 ->
 385 -> 384 -> 383 -> 382 -> 381
Time taken to deliver the message: 0.0195168 seconds

Message delivered to target node 50
Path: 381 -> 361 -> 341 -> 321 -> 301 -> 281 -> 261 -> 241 -> 221 -> 201 -> 181 -> 161 -> 141 -> 121 -> 101 -> 81 -> 61 -> 41 -> 42 -> 43 -> 44 -> 45 -> 46
-> 47 -> 48 -> 49 -> 50
Time taken to deliver the message: 0.0023631 seconds

C:\Users\julia\source\repos\app with timer\x64\Debug\app with timer.exe (process 2304) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Рисунок 3.7 — Результат третього запуску модифікованого проміжного програмного забезпечення мультикомп'ютерної системи з топологією «Решітка»

При третьому запуску програми було вирішено повернутися до розміру системи з 20 х 20. Результат випробування повторився і від вузла-відправника до вузла-отримувача надсилання повідослення займає значно більше часу, а ніж надсилання відповіді від вузла-отримувача до вузла-відправника.

Отже, можна зробити висновок, що у дані програмі існує тенденція, що надсилання відповіді від вузла-отримувача до вузла-відправника є меншою не зважаючи на розмір мультикомп'ютерної системи.

Суть четвертого експерименту проведення елементарних математичних експериментів, таких як додавання, віднімання, множення та ділення. Але особливістю цього експерименту є вибір математичної операції та введення двох

потрібних чисел в вузлі-відправнику та передача їх до вузла-отримувача для оброблення результату та надсилання відповіді до вузла-відправника.

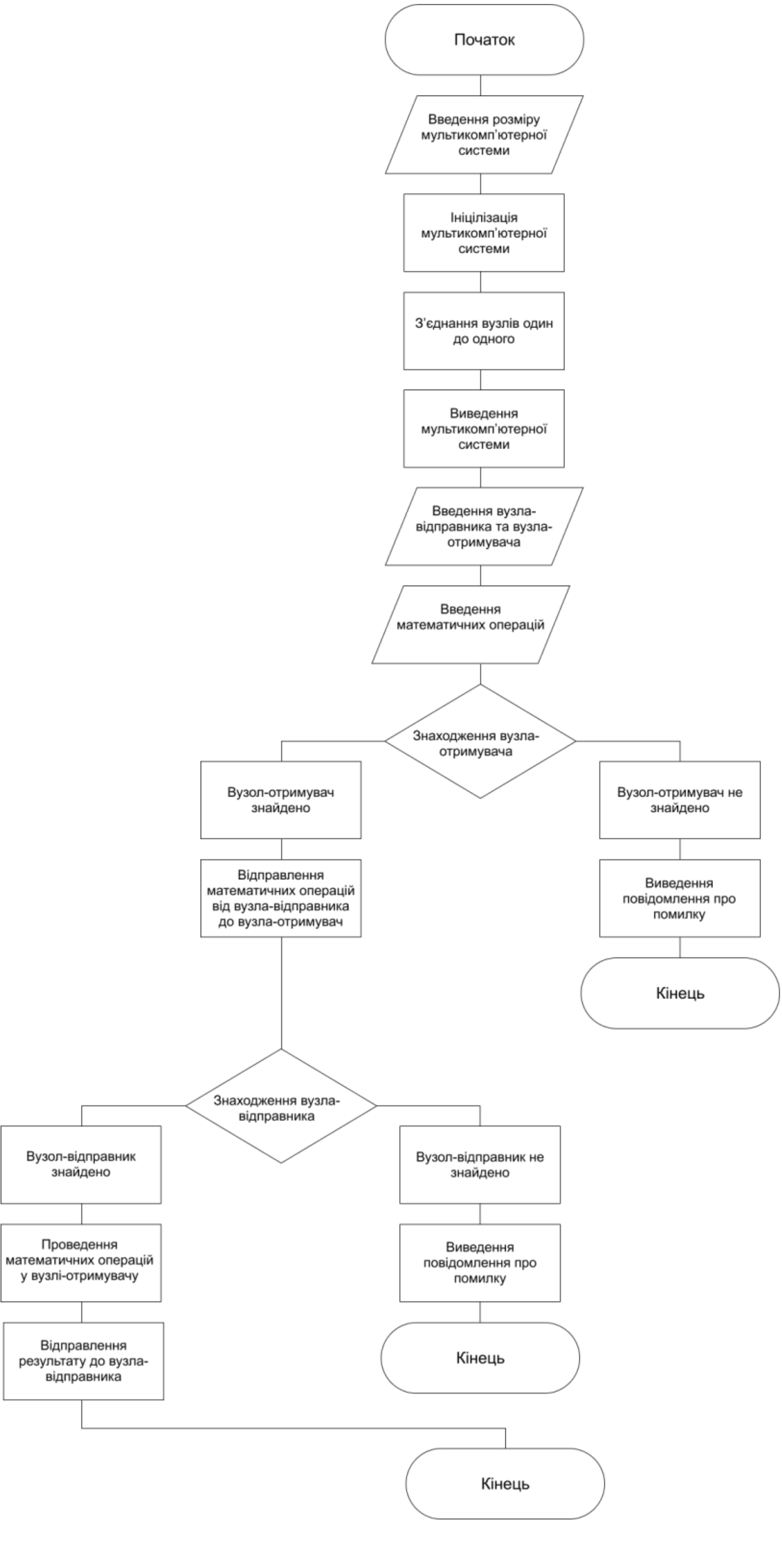


Рисунок 3.8 — Алгоритм роботи четвертого експерименту проміжного програмного забезпечення мультикомп'ютерної системи з топологіє «Решітка»

```
Microsoft Visual Studio Debu
Enter the size of grid (N x N grid): 20
Grid Layout:
1    2    3    4    5    6    7    8    9    10   11   12   13   14   15   16   17   18   19   20
21   22   23   24   25   26   27   28   29   30   31   32   33   34   35   36   37   38   39   40
41   42   43   44   45   46   47   48   49   50   51   52   53   54   55   56   57   58   59   60
61   62   63   64   65   66   67   68   69   70   71   72   73   74   75   76   77   78   79   80
81   82   83   84   85   86   87   88   89   90   91   92   93   94   95   96   97   98   99   100
101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120
121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140
141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160
161  162  163  164  165  166  167  168  169  170  171  172  173  174  175  176  177  178  179  180
181  182  183  184  185  186  187  188  189  190  191  192  193  194  195  196  197  198  199  200
201  202  203  204  205  206  207  208  209  210  211  212  213  214  215  216  217  218  219  220
221  222  223  224  225  226  227  228  229  230  231  232  233  234  235  236  237  238  239  240
241  242  243  244  245  246  247  248  249  250  251  252  253  254  255  256  257  258  259  260
261  262  263  264  265  266  267  268  269  270  271  272  273  274  275  276  277  278  279  280
281  282  283  284  285  286  287  288  289  290  291  292  293  294  295  296  297  298  299  300
301  302  303  304  305  306  307  308  309  310  311  312  313  314  315  316  317  318  319  320
321  322  323  324  325  326  327  328  329  330  331  332  333  334  335  336  337  338  339  340
341  342  343  344  345  346  347  348  349  350  351  352  353  354  355  356  357  358  359  360
361  362  363  364  365  366  367  368  369  370  371  372  373  374  375  376  377  378  379  380
381  382  383  384  385  386  387  388  389  390  391  392  393  394  395  396  397  398  399  400


Enter the ID of the sender node: 55
Enter the ID of the receiver node: 397


Enter the mathematical operation (+, -, *, /): *
Enter the first operand: 24
Enter the second operand: 5


Path: 55 -> 75 -> 95 -> 115 -> 135 -> 155 -> 175 -> 195 -> 215 -> 235 -> 255 -> 275 -> 295 -> 315 -> 335 -> 355 -> 375 -> 395 -> 396 -> 397
Time taken to receive the result: 0.0021854 seconds
Node 397 processed the operation 24 * 5 = 120


Path: 397 -> 377 -> 357 -> 337 -> 317 -> 297 -> 277 -> 257 -> 237 -> 217 -> 197 -> 177 -> 157 -> 137 -> 117 -> 97 -> 77 -> 57 -> 56 -> 55
Time taken to receive the result: 0.0012202 seconds
```

Рисунок 3.9 — Результат роботи четвертого експерименту проведення елементарних математичних операцій

Додавково до попередніх напрацювань програми додалися ще поля для вводу чисел та поле для вибору математичної операції. Також в консолі виводиться лінійка, яка вказує, що саме вузл-отримувач опрацював цей запит — «Node 18 processed the operation 4+6=13». Час оброблення повідомлення у вузла-отримувача є меншим, не зважаючи на опрацювання математичної операції.

Для перевірки програми було проведено ще один запуск данної програми.

Аналізуюри отримані данні другого запуску четвертого експерименту проведення елементарних математичних операцій було обрано для перевірки роботи іншу операцію, а саме множення та створено більшу мультикомп’ютерну систему. Час який витратився для опрацювання повідомлень є в досить великі різниці з перевагою вузла-отримувача.

```
Microsoft Visual Studio Debu
Enter the size of grid (N x N grid): 15
Grid Layout:
1       2       3       4       5       6       7       8       9       10      11      12      13      14      15
16      17      18      19      20      21      22      23      24      25      26      27      28      29      30
31      32      33      34      35      36      37      38      39      40      41      42      43      44      45
46      47      48      49      50      51      52      53      54      55      56      57      58      59      60
61      62      63      64      65      66      67      68      69      70      71      72      73      74      75
76      77      78      79      80      81      82      83      84      85      86      87      88      89      90
91      92      93      94      95      96      97      98      99      100     101     102     103     104     105
106     107     108     109     110     111     112     113     114     115     116     117     118     119     120
121     122     123     124     125     126     127     128     129     130     131     132     133     134     135
136     137     138     139     140     141     142     143     144     145     146     147     148     149     150
151     152     153     154     155     156     157     158     159     160     161     162     163     164     165
166     167     168     169     170     171     172     173     174     175     176     177     178     179     180
181     182     183     184     185     186     187     188     189     190     191     192     193     194     195
196     197     198     199     200     201     202     203     204     205     206     207     208     209     210
211     212     213     214     215     216     217     218     219     220     221     222     223     224     225


Enter the ID of the sender node: 8
Enter the ID of the receiver node: 211


Enter the mathematical operation (+, -, *, /): *
Enter the first operand: 6
Enter the second operand: 8


Path: 8 -> 23 -> 38 -> 53 -> 68 -> 83 -> 98 -> 113 -> 128 -> 143 -> 158 -> 173 -> 188 -> 203 -> 218 -> 217 -> 216 -> 215 -> 214 -> 213 -> 212 -> 211
Time taken to receive the result: 0.0211256 seconds
Node 211 processed the operation 6 * 8 = 48


Path: 211 -> 196 -> 181 -> 166 -> 151 -> 136 -> 121 -> 106 -> 91 -> 76 -> 61 -> 46 -> 31 -> 16 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8
Time taken to receive the result: 0.0015841 seconds

C:\Users\julia\source\repos\calculates\x64\Debug\calculates.exe (process 14544) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Рисунок 3.10 — Результат другого запуску четвертого експерименту проведення елементарних математичних операцій

3.6. Висновки

У третьому розділі даної кваліфікаційної роботи було описано структуру програми та доцільність використання мови програмування C++. Бібліотеки для програмного забезпечення . Розібрано та детально описано логіку роботи проміжного програмного забезпечння мультикомп’ютерної системи з топологією «Решітка». Також важливим пунктом є результати проміжного програмного забезпечення та експерименти які були проведені, щоб протестувати створену мультикомп’ютерну систему з топологією «Решітка».

ВИСНОВКИ

Під час виконання кваліфікованої роботи були проведені чисельні як теоретичні, так і практичні дослідження стосовно теми: «Проміжне програмне забезпечення мультикомп'ютерної системи «Решітка»» та реалізовано саму мультикомп'ютерну систему з заданою топологією.

У першому розділі було визначено різницю між типами комп'ютерних систем, а саме розглядалися однокомп'ютерні системи та мультикомп'ютерні системи (обмеження в кількості комп'ютерів немає). Також важливим етапом під час виконання роботи було детальне вивчення поняття вузол та все про нього, адже саме вузол є одним із ключових елементів у даній роботі, всі комунікації відбуваються між вузлами із вузлами.

У другому розділі було засвоєно поняття топології не лише заданої варіантом, а ї часто використовувані топології також бути описані та опрацьовані. Для кожної топології було розібрано на плюси та мінуси її використання. Важливим етапом в даному розділі є поняття сокетів для з'єднання між вузлами та описані алгоритми виявлення та виправлення помилок. Для успішної реалізації проєкту було опрацьовано механізми та алгоритми, підтримка цілісності та забезпечення функціоналу мультикомп'ютерної системи.

У третьому розділі розповідається про обрану, для реалізації проміжного програмного забезпечення, мову програмування C++ з детальним описом кожної використаної допоміжної бібліотеки. Розібрано логіку роботи проміжного програмного забезпечення мультикомп'ютерної системи з топологією «Решітка» та детально описано результати проведених експерементів для перевірки належної роботи.

1. Tang X., Yao Y., Yu Z., Liu L. Multi-computer Communication Reliability Evaluation System Based on 8051 Microcontroller. *Advances in Computer, Signals and Systems*. 2023. Vol.7(6). P. 36-44.

2. Silvano C., Ielmini D., Ferrandi F., Fiorin L., Curzel S., Benini L., Birke R. A. Survey on deep learning hardware accelerators for heterogeneous hpc platforms. *arXiv preprint arXiv:2306.15552*. 2023. P. 2.

3. Lee J. K., Jamieson M., Brown N., Jesus R. Test-driving RISC-V Vector hardware for HPC. *arXiv preprint arXiv:2304.10319*.2023. P. 419-432.

4. Mira D., Pérez-Sánchez E. J., Borrell R., Houzeaux G. HPC-enabling technologies for high-fidelity combustion simulations. *Proceedings of the Combustion Institute*. 2023 Vol.39(4), P. 5091-5125.

5. Enes J., Expósito R. R., Fuentes J., Cacheiro J. L., Touriño J. A pipeline architecture for feature-based unsupervised clustering using multivariate time series from HPC jobs. *Information Fusion*. 2023. Vol.93, P. 1-20.

6. Rahul P., Kaarthick B. An optimal cluster head and gateway node selection with fault tolerance. *Published in Intelligent Automation & Soft Computing*. 2023. Vol.35(2). P. 1595-1609.

7. Lin K., Fan Z., Liu B., Chen Y., Liu Z. Design of gateway nodes for wireless sensor networks based on microservice architecture. *In Third International Symposium on Computer Engineering and Intelligent Communications (ISCEIC 2022)*. SPIE. February. 2023. Vol.12462. P. 30-36.

8. Panse T., Panse P. An Efficient Gateway Node Selection Method for Clustering in Heterogeneous Mobile Ad-hoc Networks. 2023. P. 3687-3718.

9. Bilgili M., Canpolat C., Pinar E., Sahin B. Analysis of heating degree-days (HDD) data using machine learning and conventional time series methods. *Theoretical and Applied Climatology*. 2023. P. 1-20.

10. Muto R., Uchimura Y. Controller design for HDD benchmark problem using RNN-based reinforcement learning. *IFAC-PapersOnLine*. 2023. Vol.56(2). P. 4424-4429.

11. Seng[1] N. M., Al-Nahari[1] A., Ismail N. A., Ahmad A. Check for updates Big Data Application on Prediction of HDD Manufacturing Process Performance. *Data Science and Emerging Technologies: Proceedings of DaSET 2022*. 2023. Vol.165. P. 222.

12. Wang L., Shi W., Tang Y., Liu Z., He X., Xiao H., Yang Y. (). Transfer Learning-Based Lightweight SSD Model for Detection of Pests in Citrus. *Agronomy.* 2023. Vol.13(7). P. 1710.

13. Huo B., Li C., Zhang J., Xue Y., Lin Z. SAFF-SSD: Self-Attention Combined Feature Fusion-Based SSD for Small Object Detection in Remote Sensing. *Remote Sensing*. (2023). Vol.15(12). P. 3027.

14. Yang H., Wang W., Chen M., Lin B., He T., Chen H., Ouyang W. PVT-SSD: Single-Stage 3D Object Detector with Point-Voxel Transformer. *In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition.* 2023. P. 13476-13487.

15. Cohen J., Crispim-Junior C., Chiappa J. M., Rodet L. T. Industrial object detection with multi-modal SSD: closing the gap between synthetic and real images. *Multimedia Tools and Applications*. 2024. Vol.83(4). P. 12111-12138.

16. Olson K., Wampler J., Keller E. Doomed to Repeat with IPv6? Characterization of NAT-centric Security in SOHO Routers. *ACM Computing Surveys*. 2023.

17. Nassar R., Elhajj I. H., Kayssi A., Salam S. A Generalizable Machine Learning Model for NAT Detection. *In 2023 International Conference on Intelligent Computing, Communication, Networking and Services (ICCNS). IEEE*. June 2023. P. 44-49.

18. Packard N. The ARPANET into the Internet: A tale of two networks. 2023.

19. Packard N. Internet prehistory: ARPANET chronology. *Cogent Social Sciences*. 2023. Vol.9(2).P. 1-47.

20. Mohichehra M. From arpanet to www, or history of the secret network. *best scientific research-2023*. 2023. Vol. 2(1). P. 162-171.

21. Kizi T. S. G. Ethernet and Fast Ethernet network architecture. *Best Journal of Innovation in Science, Research and Development*. 2023. P. 175-179.

22. Loveless A., Phan L. T. X., Dreslinski R., Kasikci B. PCSPOOF: Compromising the safety of time-triggered ethernet. *In 2023 IEEE Symposium on Security and Privacy (SP).* IEEE. May 2023. P. 3193-3208.

23. Yang Z., Zhu W. Improvement and Optimization of Vulnerability Detection Methods for Ethernet Smart Contracts. *IEEE Access*. 2023. P. 78207-78223

24. Wen J., Yang J., Wang T., Li Y., Lv Z. Energy-efficient task allocation for reliable parallel computation of cluster-based wireless sensor network in edge computing. *Digital Communications and Networks*. 2023 Vol. 9(2). P. 473-482.

25. Miao X., Nie X., Zhang H., Zhao T., Cui B. Hetu: A highly efficient automatic parallel distributed deep learning system. 2023. P. 1-2.

26. Suleman M. J. The Use of High-Performance Computing Services in University Settings: A Usability Case Study of the University of Cincinnati's High-Performance Computing Cluster. Master's thesis. University of Cincinnati. 2023. 57 p.

27. Aparecida Silva Camacho T., Martins do Rosario V., Oliveira Napoli O., Borin E. PB3Opt: Profile-based biased Bayesian optimization to select computing clusters on the cloud. *Concurrency and Computation: Practice and Experience e7540*. 2023. Vol. 35(18). P. 777-787.

28. Grinstein J. D. CZI to Build Massive GPU Cluster for Decoding Biology with AI: The Chan Zuckerberg Initiative announced that it will build one of the most powerful high-performance computing systems for non-profit life science research in the world and develop machine learning technology and tools to support scientists solving biology's greatest challenges. *GEN Edge*. 2023. Vol. 5(1). P. 629-632.

29. Roberts M. K., Ramasamy P. An improved high performance clustering based routing protocol for wireless sensor networks in IoT. *Telecommunication Systems*. 2023. Vol.*82*(1). P. 45-59.

30. Duerrwaechter J., Kuhn T., Meyer F., Beck A., Munz C. D. PoUnce: A framework for automatized uncertainty quantification simulations on high-performance clusters. *Journal of Open Source Software*. 2023. Vol. 8(82). P. 4683.

31. Wittig A., Wittig M. Amazon Web Services in Action: An in-depth guide to AWS. Simon and Schuster. 2023. P. 5-171.

32. Yang S., Jin W., Yu Y., Hashim K. F. Optimized hadoop map reduce system for strong analytics of cloud big product data on amazon web service. *Information Processing & Management*. 2023. Vol. 60(3). P. 57-67.

33. Boneder S. Evaluation and comparison of the security offerings of the big three cloud service providers Amazon Web Services, Microsoft Azure and Google Cloud Platform. Bachelor Thesis. Technische Hochschule Ingolstadt.. 2023. 58 p.

34. Singh S., Ramkumar K. R., Kukkar A. (). Analysis and Implementation of Microsoft Azure Machine Learning Studio Services with Respect to Machine Learning Algorithms. *In Modern Electronics Devices and Communication Systems: Select Proceedings of MEDCOM 2021.* Singapore: Springer Nature Singapore. 2023. P. 91-106.

35. Manzato A. Implementing the Zero Trust model through Microsoft Azure Technologies for Enterprise Security. 2023. P. 2-8.

36. Wicaksono A. B., Munadi R. Cloud server design for heavy workload gaming computing with Google cloud platform. *International Journal of Electrical & Computer Engineering*. 2023. Vol. 13(2). P. 2197.

37. Subashini P., Dhivyaprabha T. T., Krishnaveni M., Jennyfer Susan M. B. Smart Intelligent System for Cervix Cancer Image Classification Using Google Cloud Platform. *In Enabling Technologies for Effective Planning and Management in Sustainable Smart Cities*. Cham: Springer International Publishing. 2023. P. 245-281.

38. Gupta U., Sharma R. Apache Hadoop framework for big data analytics using AI. *In Artificial Intelligence and Blockchain in Industry 4.0.* CRC Press. 2024. P. 130-140.

39. Ibtisum S., Bazgir E., Rahman S. A., Hossain S. S. A comparative analysis of big data processing paradigms: Mapreduce vs. apache spark. *World Journal of Advanced Research and Reviews*. 2023. Vol. 20(1). P. 1089-1098.

40. Truong T. X., Nhu V. H., Phuong D. T. N., Nghi L. T., Hung N. N., Hoa P. V., Bui D. T. A New Approach Based on TensorFlow Deep Neural Networks with ADAM Optimizer and GIS for Spatial Prediction of Forest Fire Danger in Tropical Areas. *Remote Sensing*. 2023. Vol. 15(14). P. 3458.

41. Pattanayak S. Introduction to deep-learning concepts and TensorFlow. *In Pro Deep Learning with TensorFlow 2.0: A Mathematical Approach to Advanced Artificial Intelligence in Python.* Berkeley, CA: Apress. 2023. P. 109-197.

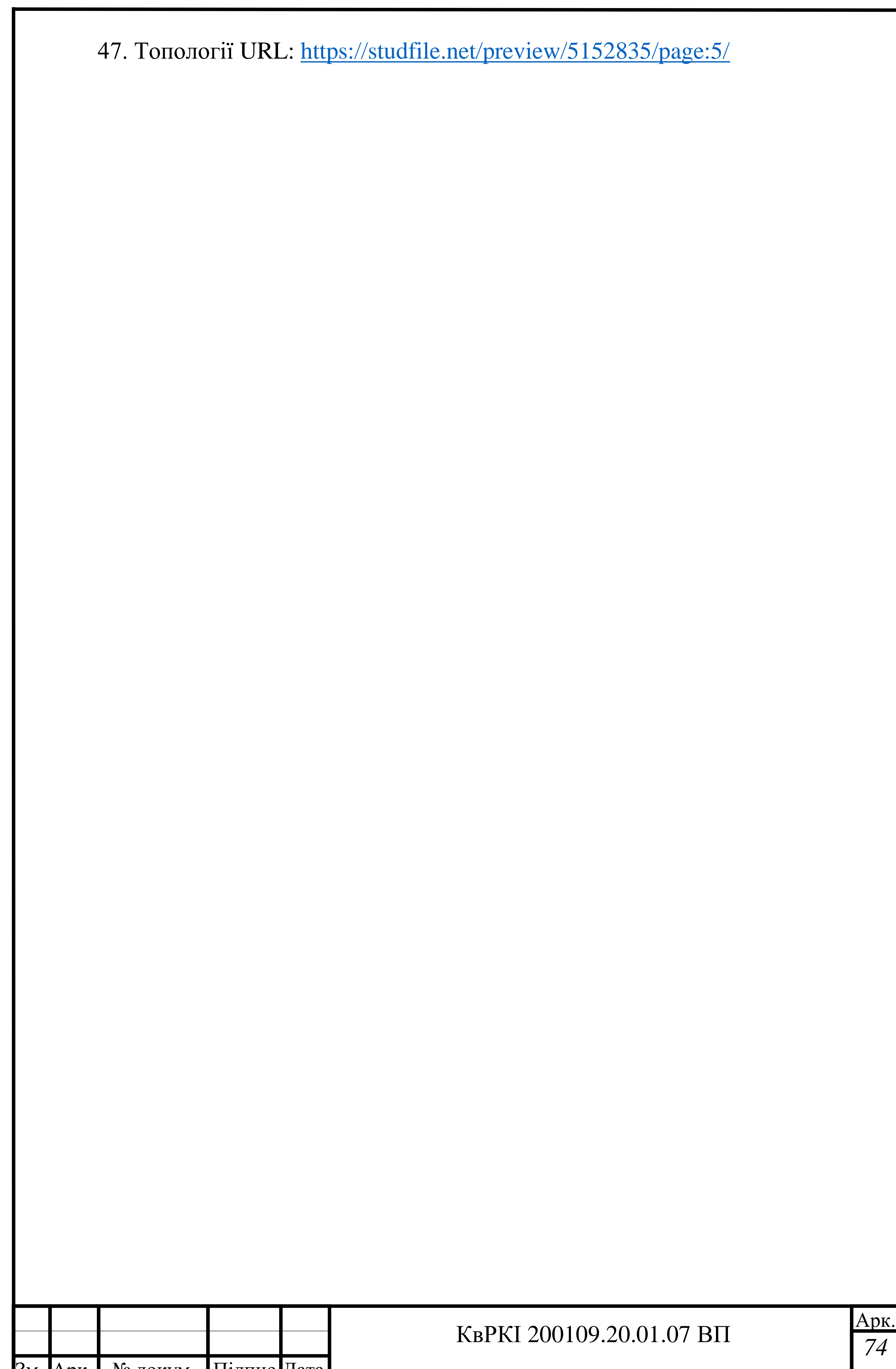
42. Kunduru A. R. Blockchain Technology for ERP Systems: A Review. *American Journal of Engineering, Mechanics and Architecture*. 2023. Vol. 1(7).P. 56-63.

43. Bandara F., Jayawickrama U., Subasinghage M., Olan F., Alamoudi H., Alharthi M. Enhancing ERP responsiveness through big data technologies: an empirical investigation. *Information Systems Frontiers*. 2023.P.1-25.

44. Panjaitan R. A. W. N., Purba M. J. FORENSIC NETWORK ANALYSIS AND IMPLEMENTATION OF SECURITY ATTACKS ON VIRTUAL PRIVATE SERVERS. *Jurnal Sistem Informasi dan Ilmu Komputer Prima* (JUSIKOM PRIMA), 2023. Vol. 6(2). P. 28-34.

45. Durga Sowjanya Kolluru, P. Bhaskara Reddy. IP to IP Calling Through Socket Programming. *2021 Asian Conference on Innovation in Technology (ASIANCON). 2021*

46. Топології багатопроцесорних систем. URL: chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://moodle.znu.edu.ua/pluginfile.php?file=/486895/mod_resource/content.pdf (дата звернення 05.05.2024)

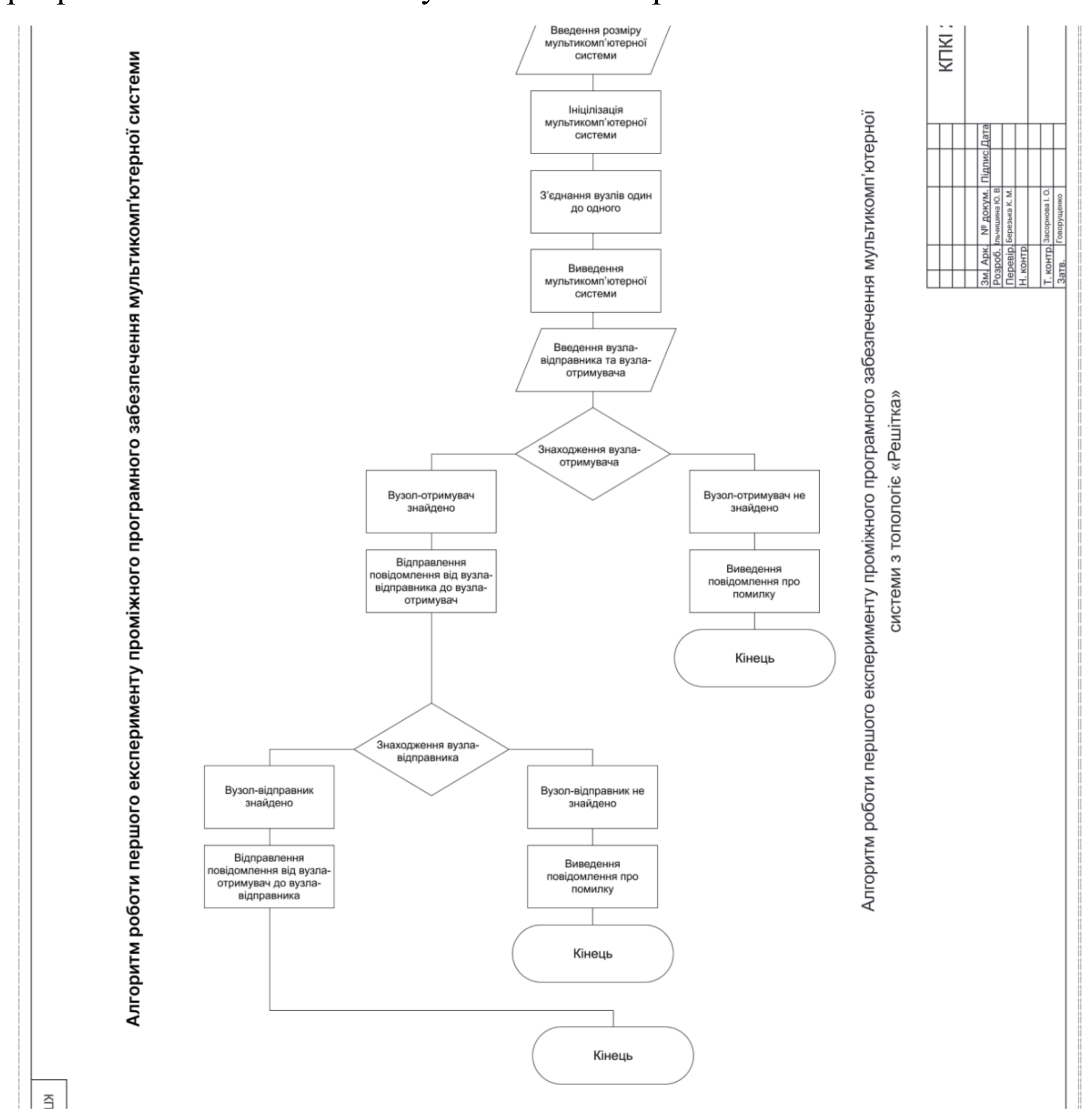


47. Топології URL: https://studfile.net/preview/5152835/page:5/

Додаток А

(обов’язковий)

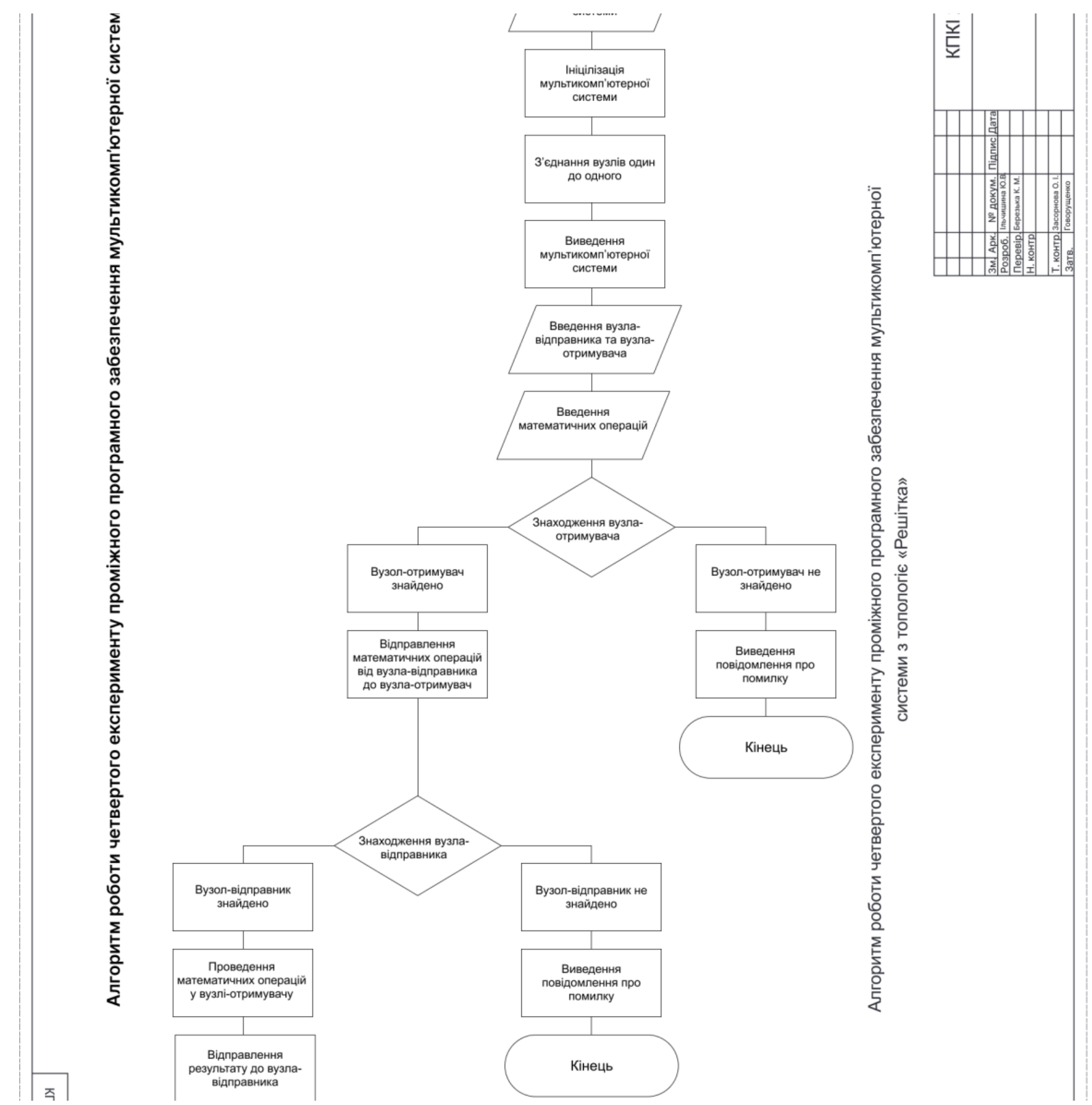
Копія креслення « Алгоритм роботи першого експерименту проміжного програмного забезпечення мультикомп’ютерної системи»



Додаток Б

(обов'язковий)

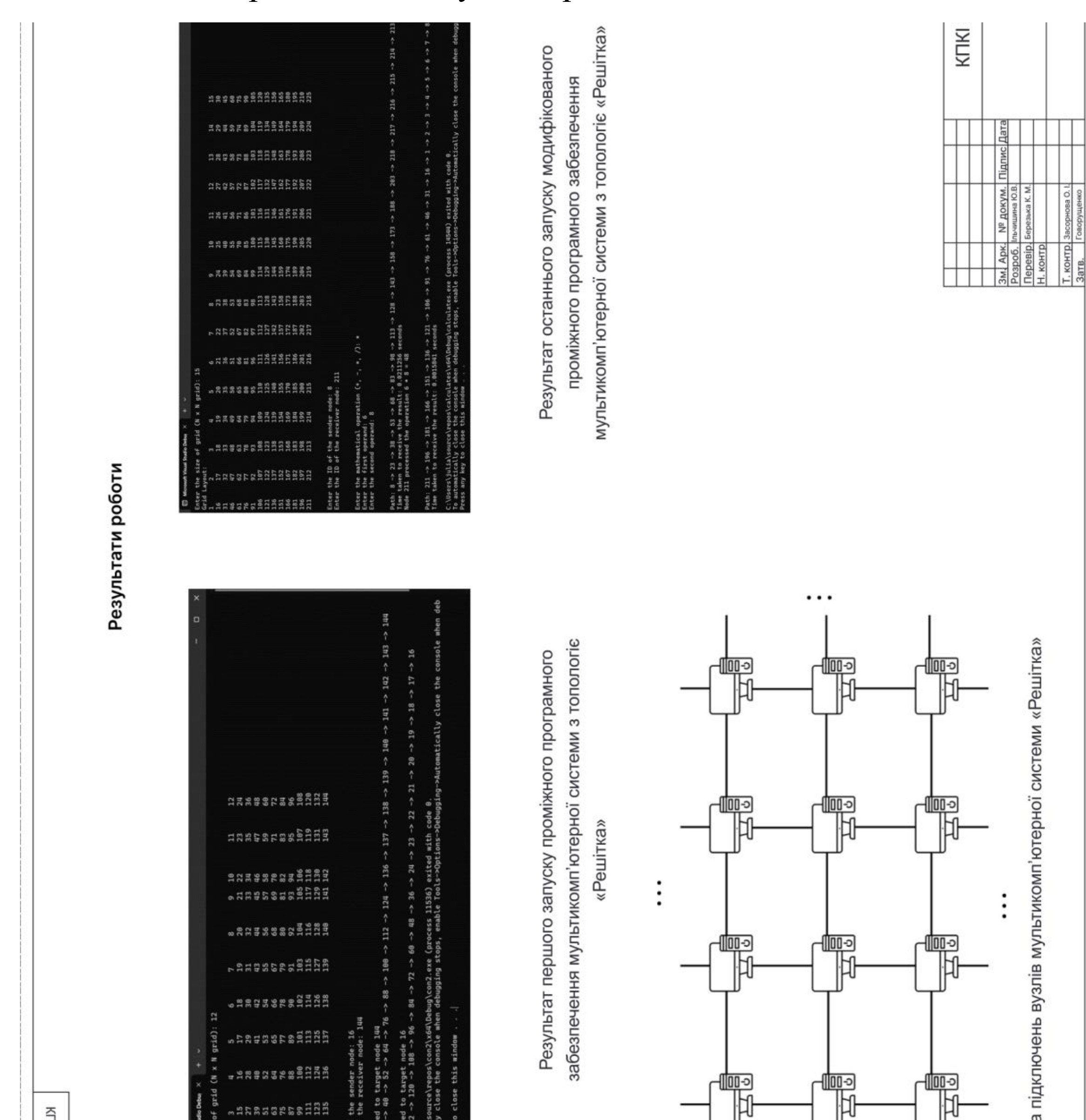
Копія креслення «Алгоритм роботи четвертого експерименту проміжного програмного забезпечення мультикомп'ютерної системи»



Додаток В

(обов'язковий)

Копія креслення «Результат роботи»



Додаток Г

Код програми першого експерименту

```
#include <iostream>
#include <vector>
#include <memory>
#include <queue>
#include <unordered_map>
#include <unordered_set>
#include <stack>
#include <cstdlib>
```

```cpp
class Node : public std::enable_shared_from_this<Node> {
public:
    // Constructor to initialize node ID
    Node(int id) : id(id) {}

    // Connect the node to a neighboring node
    void connect(std::shared_ptr<Node> neighbor) {
        neighbors.push_back(neighbor);
    }

    // Send a message using BFS to prevent recursion and
stack overflow
    void sendMessage(const std::string& message, int
targetNodeId) {
        std::queue<std::shared_ptr<Node>> queue;
        std::unordered_map<int, int> predecessor;  // To
track the path
        std::unordered_set<int> visited;

        queue.push(shared_from_this());
        visited.insert(id);
        predecessor[id] = -1;  // Start node has no
predecessor

        while (!queue.empty()) {
            auto currentNode = queue.front();
            queue.pop();

            if (currentNode->id == targetNodeId) {
                std::cout << " " << std::endl;
                std::cout << "Message delivered to target
node " << targetNodeId << std::endl;
                printPath(predecessor, targetNodeId);
                return;
            }

            for (auto& neighbor : currentNode->neighbors) {
                if (visited.find(neighbor->id) ==
visited.end()) {
                    visited.insert(neighbor->id);
                    queue.push(neighbor);
                    predecessor[neighbor->id] =
currentNode->id;  // Set predecessor
                }
```

```
            }
        }
        std::cout << " " << std::endl;
        std::cout << "Message not delivered" << std::endl;
    }

    // Print the path from source to target
    void printPath(const std::unordered_map<int, int>&
predecessor, int targetNodeId) {
        std::stack<int> path;
        int current = targetNodeId;

        while (current != -1) {
            path.push(current);
            current = predecessor.at(current);
        }

        std::cout << "Path: ";
        while (!path.empty()) {
            std::cout << path.top();
            path.pop();
            if (!path.empty()) {
                std::cout << " -> ";
            }
        }
        std::cout << std::endl;
    }

    // Get the node ID
    int getId() const {
        return id;
    }

private:
    int id;  // Node ID
    std::vector<std::shared_ptr<Node>> neighbors;  // List
of neighboring nodes
};

// Function to print the grid
void printGrid(const
std::vector<std::vector<std::shared_ptr<Node>>>& grid, int
gridSize) {
    for (int i = 0; i < gridSize; ++i) {
        for (int j = 0; j < gridSize; ++j) {
```

```
            std::cout << grid[i][j]->getId() << "\t";
        }
        std::cout << std::endl;
    }
}

int main() {
    system("color F0");
    int gridSize;
    std::cout << "Enter the size of grid (N x N grid): ";
    std::cin >> gridSize;

    int totalNodes = gridSize * gridSize;
    std::vector<std::vector<std::shared_ptr<Node>>>
grid(gridSize,
std::vector<std::shared_ptr<Node>>(gridSize));

    // Initialize nodes and assign IDs
    int nodeId = 1;
    for (int i = 0; i < gridSize; i++) {
        for (int j = 0; j < gridSize; j++) {
            grid[i][j] = std::make_shared<Node>(nodeId++);
        }
    }

    // Connect each node to its immediate neighbors
    for (int i = 0; i < gridSize; i++) {
        for (int j = 0; j < gridSize; j++) {
            if (i > 0) grid[i][j]->connect(grid[i - 1][j]);
// Connect to the node above
            if (i < gridSize - 1) grid[i][j]-
>connect(grid[i + 1][j]); // Connect to the node below
            if (j > 0) grid[i][j]->connect(grid[i][j - 1]);
// Connect to the node on the left
            if (j < gridSize - 1) grid[i][j]-
>connect(grid[i][j + 1]); // Connect to the node on the
right
        }
    }

    // Print the grid
    std::cout << "Grid Layout:" << std::endl;
    printGrid(grid, gridSize);
```

```
    int senderId, receiverId;
    std::cout << " " << std::endl;
    std::cout << " " << std::endl;
    std::cout << "Enter the ID of the sender node: ";
    std::cin >> senderId;
    std::cout << "Enter the ID of the receiver node: ";
    std::cin >> receiverId;

    // Find the sender node and send the message
    bool senderFound = false;
    for (int i = 0; i < gridSize && !senderFound; i++) {
        for (int j = 0; j < gridSize && !senderFound; j++)
{
            if (grid[i][j]->getId() == senderId) {
                grid[i][j]->sendMessage("Hello, target
node!", receiverId);
                senderFound = true;
            }
        }
    }

    if (!senderFound) {
        std::cout << "Sender node not found!" << std::endl;
        return 1;
    }

    // Find the receiver node and send the message back
    bool receiverFound = false;
    for (int i = 0; i < gridSize && !receiverFound; i++) {
        for (int j = 0; j < gridSize && !receiverFound;
j++) {
            if (grid[i][j]->getId() == receiverId) {
                grid[i][j]->sendMessage("Hello, sender
node!", senderId);
                receiverFound = true;
            }
        }
    }

    if (!receiverFound) {
        std::cout << "Receiver node not found!" <<
std::endl;
        return 1;
    }
```

```
    return 0;
}
```

Додаток Д

Код програми першого експерименту

```
#include <iostream>
#include <vector>
#include <memory>
#include <queue>
#include <unordered_map>
#include <unordered_set>
#include <stack>
#include <chrono>
#include <functional>
#include <cstdlib>

class Node : public std::enable_shared_from_this<Node> {
public:
    // Constructor to initialize node ID
    Node(int id) : id(id) {}

    // Connect the node to a neighboring node
    void connect(std::shared_ptr<Node> neighbor) {
        neighbors.push_back(neighbor);
    }

    // Send a message using BFS to prevent recursion and
stack overflow
    void sendMessage(const std::string& message, int
targetNodeId) {
        auto startTime =
std::chrono::high_resolution_clock::now(); // Start timing

        std::queue<std::shared_ptr<Node>> queue;
        std::unordered_map<int, int> predecessor;  // To
track the path
        std::unordered_set<int> visited;

        queue.push(shared_from_this());
        visited.insert(id);
        predecessor[id] = -1;  // Start node has no
predecessor

        while (!queue.empty()) {
            auto currentNode = queue.front();
```

```
            queue.pop();

            if (currentNode->id == targetNodeId) {
                auto endTime =
std::chrono::high_resolution_clock::now(); // End timing
                std::chrono::duration<double> duration =
endTime - startTime;

                // std::cout << "\nMessage delivered to
target node " << targetNodeId << " from node " << id <<
std::endl;
                printPath(predecessor, targetNodeId);

                std::cout << "Time taken to deliver the
message: " << duration.count() << " seconds" << std::endl;
                return;
            }

            for (auto& neighbor : currentNode->neighbors) {
                if (visited.find(neighbor->id) ==
visited.end()) {
                    visited.insert(neighbor->id);
                    queue.push(neighbor);
                    predecessor[neighbor->id] =
currentNode->id;  // Set predecessor
                }
            }
        }
        std::cout << "\nMessage not delivered" <<
std::endl;
    }

    // Send a mathematical operation request to the target
node
    void sendMathOperation(const std::string& operation,
double operand1, double operand2, int targetNodeId) {
        auto startTime =
std::chrono::high_resolution_clock::now(); // Start timing

        std::queue<std::shared_ptr<Node>> queue;
        std::unordered_map<int, int> predecessor;  // To
track the path
        std::unordered_set<int> visited;

        queue.push(shared_from_this());
```

```
        visited.insert(id);
        predecessor[id] = -1;  // Start node has no
predecessor

        while (!queue.empty()) {
            auto currentNode = queue.front();
            queue.pop();

            if (currentNode->id == targetNodeId) {
                double result = performOperation(operation,
operand1, operand2);

                auto endTime =
std::chrono::high_resolution_clock::now(); // End timing
                std::chrono::duration<double> duration =
endTime - startTime;

               /* std::cout << "\nMath operation result
received by node " << id << " from node " << targetNodeId
<< ": " << result << std::endl;*/
                printPath(predecessor, targetNodeId);

                std::cout << "Time taken to receive the
result: " << duration.count() << " seconds" << std::endl;
                return;
            }

            for (auto& neighbor : currentNode->neighbors) {
                if (visited.find(neighbor->id) ==
visited.end()) {
                    visited.insert(neighbor->id);
                    queue.push(neighbor);
                    predecessor[neighbor->id] =
currentNode->id;  // Set predecessor
                }
            }
        }
        std::cout << "\nMath operation request not
delivered" << std::endl;
    }

    // Process a mathematical operation request and send
back the result
    void processMathOperation(const std::string& operation,
double operand1, double operand2, int senderNodeId) {
```

```
        double result = performOperation(operation,
operand1, operand2);

        // Simulate sending the result back to the sender
node
        std::cout << "Node " << id << " processed the
operation " << operand1 << " " << operation << " " <<
operand2 << " = " << result << std::endl;
        sendMathOperation(operation, result, 0,
senderNodeId);
    }

    // Print the path from source to target
    void printPath(const std::unordered_map<int, int>&
predecessor, int targetNodeId) {
        std::stack<int> path;
        int current = targetNodeId;

        while (current != -1) {
            path.push(current);
            current = predecessor.at(current);
        }

        std::cout << "\n\nPath: ";
        while (!path.empty()) {
            std::cout << path.top();
            path.pop();
            if (!path.empty()) {
                std::cout << " -> ";
            }
        }
        std::cout << std::endl;
    }

    // Get the node ID
    int getId() const {
        return id;
    }

private:
    int id;  // Node ID
    std::vector<std::shared_ptr<Node>> neighbors;  // List
of neighboring nodes

    // Perform the mathematical operation
```

```
    double performOperation(const std::string& operation,
double operand1, double operand2) {
        if (operation == "+") {
            return operand1 + operand2;
        }
        else if (operation == "-") {
            return operand1 - operand2;
        }
        else if (operation == "*") {
            return operand1 * operand2;
        }
        else if (operation == "/") {
            return operand1 / operand2;
        }
        else {
            std::cerr << "Unknown operation: " << operation
<< std::endl;
            return 0;
        }
    }
};

// Function to print the grid
void printGrid(const
std::vector<std::vector<std::shared_ptr<Node>>>& grid, int
gridSize) {
    for (int i = 0; i < gridSize; ++i) {
        for (int j = 0; j < gridSize; ++j) {
            std::cout << grid[i][j]->getId() << "\t";
        }
        std::cout << std::endl;
    }
}

int main() {
    system("color F0");
    int gridSize;
    std::cout << "Enter the size of grid (N x N grid): ";
    std::cin >> gridSize;

    int totalNodes = gridSize * gridSize;
    std::vector<std::vector<std::shared_ptr<Node>>>
grid(gridSize,
std::vector<std::shared_ptr<Node>>(gridSize));
```

```
    // Initialize nodes and assign IDs
    int nodeId = 1;
    for (int i = 0; i < gridSize; i++) {
        for (int j = 0; j < gridSize; j++) {
            grid[i][j] = std::make_shared<Node>(nodeId++);
        }
    }

    // Connect each node to its immediate neighbors
    for (int i = 0; i < gridSize; i++) {
        for (int j = 0; j < gridSize; j++) {
            if (i > 0) grid[i][j]->connect(grid[i - 1][j]);
// Connect to the node above
            if (i < gridSize - 1) grid[i][j]-
>connect(grid[i + 1][j]); // Connect to the node below
            if (j > 0) grid[i][j]->connect(grid[i][j - 1]);
// Connect to the node on the left
            if (j < gridSize - 1) grid[i][j]-
>connect(grid[i][j + 1]); // Connect to the node on the
right
        }
    }

    // Print the grid
    std::cout << "Grid Layout:" << std::endl;
    printGrid(grid, gridSize);

    int senderId, receiverId;
    std::cout << "\n\nEnter the ID of the sender node: ";
    std::cin >> senderId;
    std::cout << "Enter the ID of the receiver node: ";
    std::cin >> receiverId;

    std::string operation;
    double operand1, operand2;
    std::cout << "\n\nEnter the mathematical operation (+,
-, *, /): ";
    std::cin >> operation;
    std::cout << "Enter the first operand: ";
    std::cin >> operand1;
    std::cout << "Enter the second operand: ";
    std::cin >> operand2;

    // Find the sender node and send the math operation
    bool senderFound = false;
```

```
    for (int i = 0; i < gridSize && !senderFound; i++) {
        for (int j = 0; j < gridSize && !senderFound; j++)
{
            if (grid[i][j]->getId() == senderId) {
                grid[i][j]->sendMathOperation(operation,
operand1, operand2, receiverId);
                senderFound = true;
            }
        }
    }

    if (!senderFound) {
        std::cout << "Sender node not found!" << std::endl;
        return 1;
    }

    // Find the receiver node and process the math
operation
    bool receiverFound = false;
    for (int i = 0; i < gridSize && !receiverFound; i++) {
        for (int j = 0; j < gridSize && !receiverFound;
j++) {
            if (grid[i][j]->getId() == receiverId) {
                grid[i][j]->processMathOperation(operation,
operand1, operand2, senderId);
                receiverFound = true;
            }
        }
    }

    if (!receiverFound) {
        std::cout << "Receiver node not found!" <<
std::endl;
        return 1;
    }

    return 0;
}
```

Додаток Е

Код програми третього експерименту

```
#include <iostream>
#include <vector>
#include <memory>
#include <queue>
#include <unordered_map>
#include <unordered_set>
#include <stack>
#include <chrono>
#include <functional>
#include <cstdlib>

class Node : public std::enable_shared_from_this<Node> {
public:
    // Constructor to initialize node ID
    Node(int id) : id(id) {}

    // Connect the node to a neighboring node
    void connect(std::shared_ptr<Node> neighbor) {
        neighbors.push_back(neighbor);
    }

    // Send a message using BFS to prevent recursion and
stack overflow
    void sendMessage(const std::string& message, int
targetNodeId) {
        auto startTime =
std::chrono::high_resolution_clock::now(); // Start timing

        std::queue<std::shared_ptr<Node>> queue;
        std::unordered_map<int, int> predecessor;  // To
track the path
        std::unordered_set<int> visited;

        queue.push(shared_from_this());
        visited.insert(id);
        predecessor[id] = -1;  // Start node has no
predecessor

        while (!queue.empty()) {
            auto currentNode = queue.front();
```

```
            queue.pop();

            if (currentNode->id == targetNodeId) {
                auto endTime =
std::chrono::high_resolution_clock::now(); // End timing
                std::chrono::duration<double> duration =
endTime - startTime;

               // std::cout << "\nMessage delivered to
target node " << targetNodeId << " from node " << id <<
std::endl;
                printPath(predecessor, targetNodeId);

                std::cout << "Time taken to deliver the
message: " << duration.count() << " seconds" << std::endl;
                return;
            }

            for (auto& neighbor : currentNode->neighbors) {
                if (visited.find(neighbor->id) ==
visited.end()) {
                    visited.insert(neighbor->id);
                    queue.push(neighbor);
                    predecessor[neighbor->id] =
currentNode->id;  // Set predecessor
                }
            }
        }
        std::cout << "\nMessage not delivered" <<
std::endl;
    }

    // Send a mathematical operation request to the target
node
    void sendMathOperation(const std::string& operation,
double operand1, double operand2, int targetNodeId) {
        auto startTime =
std::chrono::high_resolution_clock::now(); // Start timing

        std::queue<std::shared_ptr<Node>> queue;
        std::unordered_map<int, int> predecessor;  // To
track the path
        std::unordered_set<int> visited;

        queue.push(shared_from_this());
```

```
        visited.insert(id);
        predecessor[id] = -1;  // Start node has no
predecessor

        while (!queue.empty()) {
            auto currentNode = queue.front();
            queue.pop();

            if (currentNode->id == targetNodeId) {
                double result = performOperation(operation,
operand1, operand2);

                auto endTime =
std::chrono::high_resolution_clock::now(); // End timing
                std::chrono::duration<double> duration =
endTime - startTime;

               /* std::cout << "\nMath operation result
received by node " << id << " from node " << targetNodeId
<< ": " << result << std::endl;*/
                printPath(predecessor, targetNodeId);

                std::cout << "Time taken to receive the
result: " << duration.count() << " seconds" << std::endl;
                return;
            }

            for (auto& neighbor : currentNode->neighbors) {
                if (visited.find(neighbor->id) ==
visited.end()) {
                    visited.insert(neighbor->id);
                    queue.push(neighbor);
                    predecessor[neighbor->id] =
currentNode->id;  // Set predecessor
                }
            }
        }
        std::cout << "\nMath operation request not
delivered" << std::endl;
    }

    // Process a mathematical operation request and send
back the result
    void processMathOperation(const std::string& operation,
double operand1, double operand2, int senderNodeId) {
```

```
        double result = performOperation(operation,
operand1, operand2);

        // Simulate sending the result back to the sender
node
        std::cout << "Node " << id << " processed the
operation " << operand1 << " " << operation << " " <<
operand2 << " = " << result << std::endl;
        sendMathOperation(operation, result, 0,
senderNodeId);
    }

    // Print the path from source to target
    void printPath(const std::unordered_map<int, int>&
predecessor, int targetNodeId) {
        std::stack<int> path;
        int current = targetNodeId;

        while (current != -1) {
            path.push(current);
            current = predecessor.at(current);
        }

        std::cout << "\n\nPath: ";
        while (!path.empty()) {
            std::cout << path.top();
            path.pop();
            if (!path.empty()) {
                std::cout << " -> ";
            }
        }
        std::cout << std::endl;
    }

    // Get the node ID
    int getId() const {
        return id;
    }

private:
    int id;  // Node ID
    std::vector<std::shared_ptr<Node>> neighbors;  // List
of neighboring nodes

    // Perform the mathematical operation
```

```
    double performOperation(const std::string& operation,
double operand1, double operand2) {
        if (operation == "+") {
            return operand1 + operand2;
        }
        else if (operation == "-") {
            return operand1 - operand2;
        }
        else if (operation == "*") {
            return operand1 * operand2;
        }
        else if (operation == "/") {
            return operand1 / operand2;
        }
        else {
            std::cerr << "Unknown operation: " << operation
<< std::endl;
            return 0;
        }
    }
};

// Function to print the grid
void printGrid(const
std::vector<std::vector<std::shared_ptr<Node>>>& grid, int
gridSize) {
    for (int i = 0; i < gridSize; ++i) {
        for (int j = 0; j < gridSize; ++j) {
            std::cout << grid[i][j]->getId() << "\t";
        }
        std::cout << std::endl;
    }
}

int main() {
    system("color F0");
    int gridSize;
    std::cout << "Enter the size of grid (N x N grid): ";
    std::cin >> gridSize;

    int totalNodes = gridSize * gridSize;
    std::vector<std::vector<std::shared_ptr<Node>>>
grid(gridSize,
std::vector<std::shared_ptr<Node>>(gridSize));
```

```
    // Initialize nodes and assign IDs
    int nodeId = 1;
    for (int i = 0; i < gridSize; i++) {
        for (int j = 0; j < gridSize; j++) {
            grid[i][j] = std::make_shared<Node>(nodeId++);
        }
    }

    // Connect each node to its immediate neighbors
    for (int i = 0; i < gridSize; i++) {
        for (int j = 0; j < gridSize; j++) {
            if (i > 0) grid[i][j]->connect(grid[i - 1][j]);
// Connect to the node above
            if (i < gridSize - 1) grid[i][j]-
>connect(grid[i + 1][j]); // Connect to the node below
            if (j > 0) grid[i][j]->connect(grid[i][j - 1]);
// Connect to the node on the left
            if (j < gridSize - 1) grid[i][j]-
>connect(grid[i][j + 1]); // Connect to the node on the
right
        }
    }

    // Print the grid
    std::cout << "Grid Layout:" << std::endl;
    printGrid(grid, gridSize);

    int senderId, receiverId;
    std::cout << "\n\nEnter the ID of the sender node: ";
    std::cin >> senderId;
    std::cout << "Enter the ID of the receiver node: ";
    std::cin >> receiverId;

    std::string operation;
    double operand1, operand2;
    std::cout << "\n\nEnter the mathematical operation (+,
-, *, /): ";
    std::cin >> operation;
    std::cout << "Enter the first operand: ";
    std::cin >> operand1;
    std::cout << "Enter the second operand: ";
    std::cin >> operand2;

    // Find the sender node and send the math operation
    bool senderFound = false;
```

```
    for (int i = 0; i < gridSize && !senderFound; i++) {
        for (int j = 0; j < gridSize && !senderFound; j++)
{
            if (grid[i][j]->getId() == senderId) {
                grid[i][j]->sendMathOperation(operation,
operand1, operand2, receiverId);
                senderFound = true;
            }
        }
    }

    if (!senderFound) {
        std::cout << "Sender node not found!" << std::endl;
        return 1;
    }

    // Find the receiver node and process the math
operation
    bool receiverFound = false;
    for (int i = 0; i < gridSize && !receiverFound; i++) {
        for (int j = 0; j < gridSize && !receiverFound;
j++) {
            if (grid[i][j]->getId() == receiverId) {
                grid[i][j]->processMathOperation(operation,
operand1, operand2, senderId);
                receiverFound = true;
            }
        }
    }

    if (!receiverFound) {
        std::cout << "Receiver node not found!" <<
std::endl;
        return 1;
    }

    return 0;
}
```