ДОДАТОК А

Графічний матеріал кваліфікаційної роботи

Харківський національний університет радіоелектроніки

Кваліфікаційна робота

«Моделі підвищення ефективності взаємодії програм у розподілених системах»

Виконав:
ст. гр. КСМм-23-1
Бондар І.І.

Керівник:
проф. Міхаль О.П.

Мета та завдання кваліфікаційної роботи

Актуальність теми дослідження полягає у необхідності розв'язання проблем ефективної взаємодії програмних компонентів у розподілених системах.

Мета кваліфікаційної роботи: розробка та дослідження моделей підвищення ефективності взаємодії програм у розподілених системах шляхом удосконалення механізмів передачі даних, балансування навантаження, кешування та інтеграції сучасних технологій у системну архітектуру.

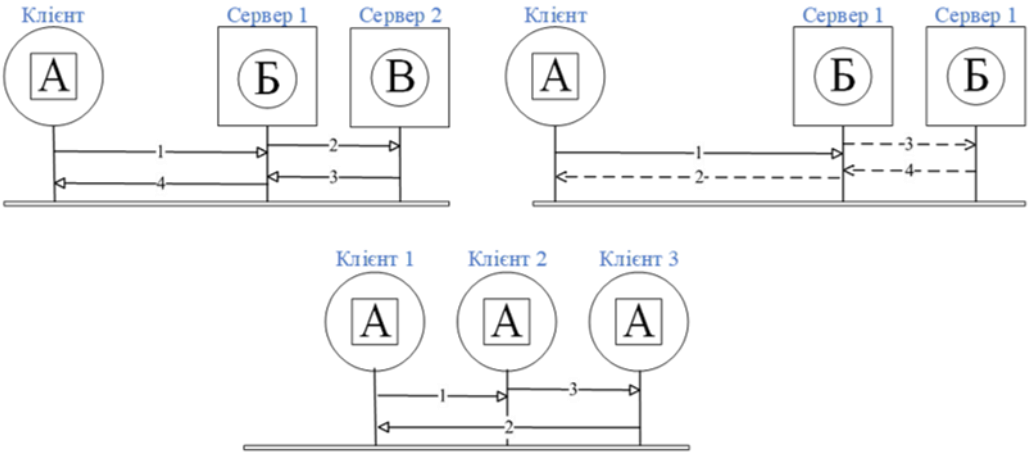
Завдання:

- ❑аналіз існуючих підходів та моделей взаємодії програм в розподілених системах;
- ❑аналіз моделей оптимізації передачі даних, балансування навантаження та управління кешуванням;
- ❑проведення експериментальних досліджень ефективності запропонованих моделей у порівнянні з існуючими рішеннями.

Об'єкт дослідження: взаємодія програмних компонентів у розподілених системах.

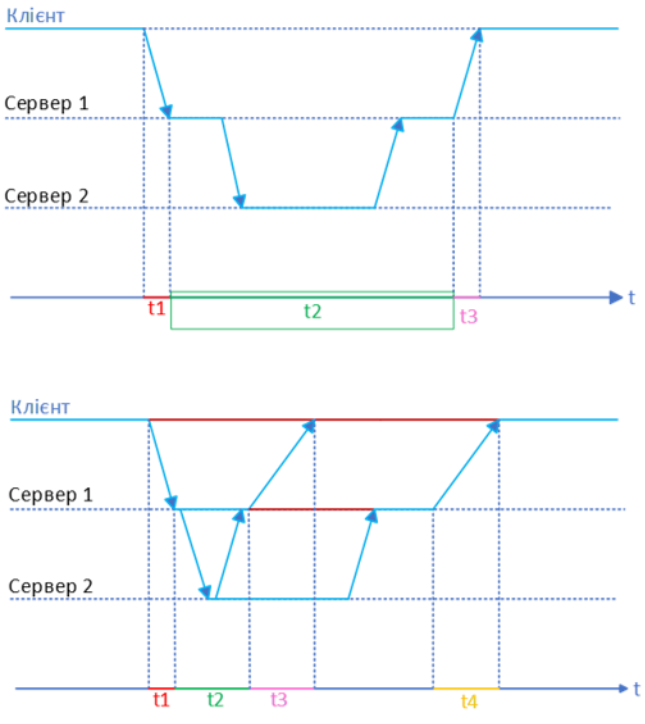
Взаємодія на основі моделі «клієнт-сервер»

3



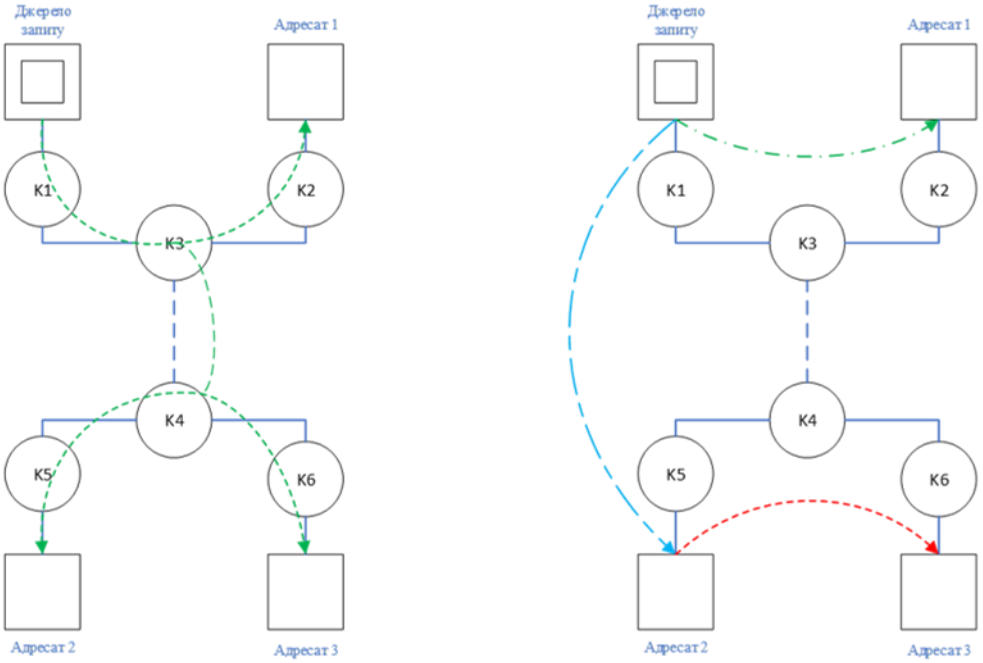
Час очікування процесом-клієнтом відповіді від процесу-сервера

4



Реалізація групового мовлення на мережевому та прикладному рівнях

5



Надійність взаємодії програм

6

7

Порівняння

Модель	Продуктивність	Надійність	Недоліки
Централізована	Висока при низькому навантаженні, але обмежена потужністю центрального вузла	Низька, оскільки відмова центрального вузла зупиняє всю систему	Обмежена масштабованість, залежність від центрального вузла
Децентралізована	Висока при масштабуванні завдяки розподілу навантаження	Висока, оскільки відмова одного вузла не зупиняє систему	Складність узгодження даних між вузлами, додаткові витрати на синхронізацію
Модель з чергами повідомлень	Висока завдяки асинхронній взаємодії та паралельній обробці	Висока, оскільки повідомлення зберігаються у чергах навіть при збоях	Додаткові накладні витрати на управління чергами та моніторинг

8

Формалізація критеріїв ефективності взаємодії

$$T\_response = T\_network + T\_processing + T\_storage$$

$$Throughput = N / T\_total$$

$$R(t) = e^{(-\lambda t)}$$

$$S(n) = P(n) / P(1)$$

$$Latency = (\Sigma T\_i) / N$$

$$\Lambda_1 = \Lambda r \sum_{i=1}^{\infty} i(1-r)^{i-1} = \frac{\Lambda}{1-(1-p)^k}.$$

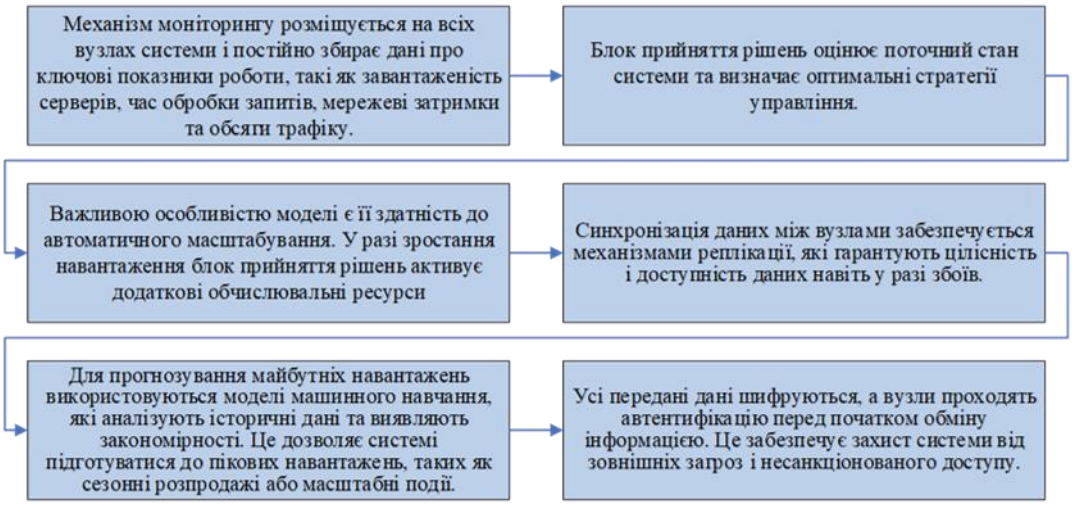
Пропонована модель адаптивного управління взаємодією програм

9

Модель адаптивного управління взаємодією програм у розподілених системах спрямована на підвищення ефективності комунікації між компонентами системи шляхом динамічного налаштування параметрів і адаптації до змінного навантаження. Основу моделі складає механізм моніторингу, який відстежує стан системи в реальному часі, і блок прийняття рішень, що аналізує отримані дані та оптимізує роботу системи.

Механізм функціонування розроблювальної моделі

10



Сценарії симуляції запропонованої моделі

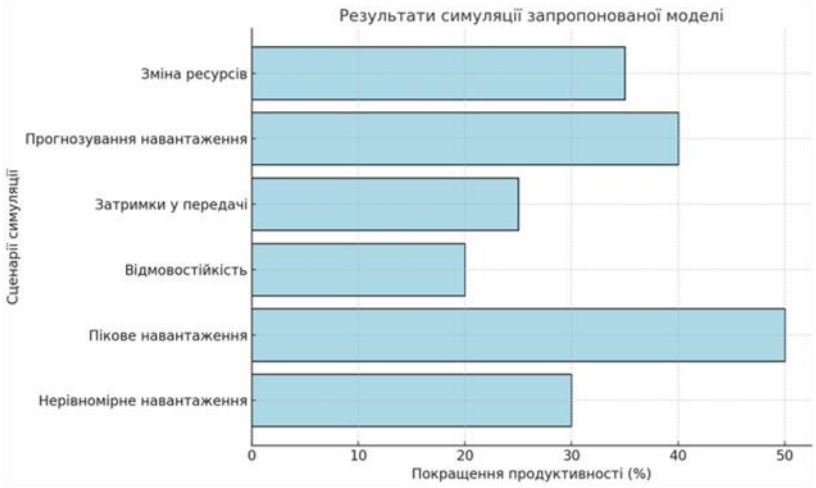
11

Сценарій 1	*Сценарій 2*	*Сценарій 3*
Нерівномірне навантаження на систему	Пікове навантаження	Відмовостійкість системи
Затримки у передачі даних	Прогнозування навантаження	Зміна доступних ресурсів
Сценарій 4	*Сценарій 5*	*Сценарій 6*

Симуляція всіх сценаріїв підтвердила ефективність і гнучкість запропонованої моделі. Модель адаптивно реагувала на зміни в умовах роботи, забезпечуючи високу продуктивність, надійність і стійкість системи до зовнішніх і внутрішніх факторів. Верифікація результатів за допомогою порівняння з еталонними значеннями підтвердила відповідність моделі поставленим цілям.

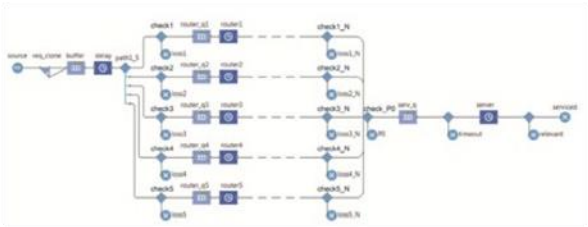
Результати симуляції запропонованої моделі

12



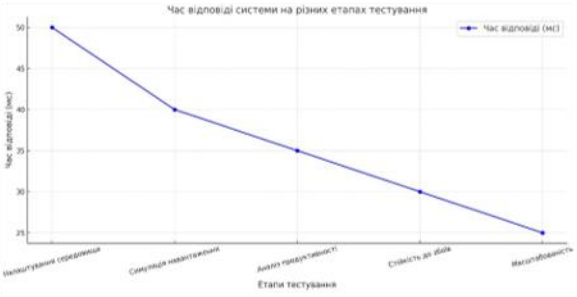
Архітектура ПЗ. Фрагмент моделі розподіленої системи.

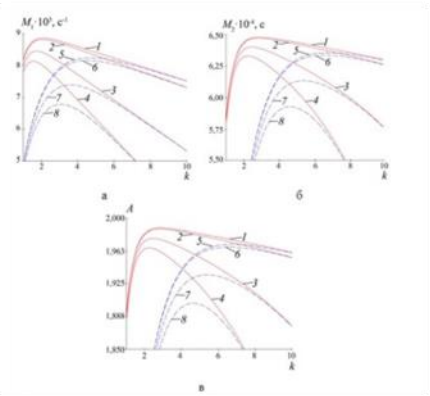
13

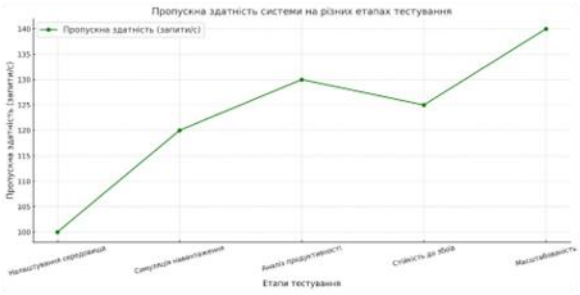


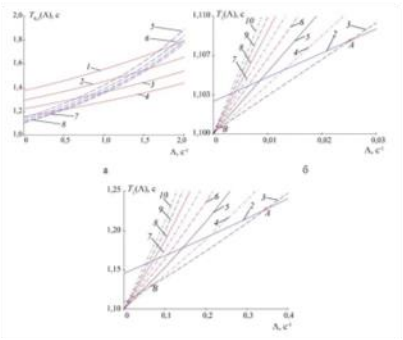
Результати роботи

14









Висновки

За час підготовки кваліфікаційної роботи опубліковани тези доповідей за даною темою. Проведений аналіз існуючих підходів і моделей взаємодії в розподілених системах показав, що традиційні методи мають обмеження у масштабованості, продуктивності та адаптивності до змінних умов роботи. Це підкреслило необхідність розробки нових підходів, орієнтованих на оптимізацію роботи таких систем. Розроблені моделі оптимізації передачі даних, балансування навантаження та управління кешуванням базуються на застосуванні інтелектуальних алгоритмів та динамічної адаптації до умов роботи. Ці моделі враховують реальний стан вузлів системи, прогнозують можливі навантаження та автоматично адаптують ресурси для забезпечення стабільної роботи. Експериментальне дослідження підтвердило ефективність запропонованих моделей; тестування прототипу в умовах розподіленої системи показало зменшення середнього часу відповіді на 20-30% у порівнянні з традиційними підходами; підвищення стійкості системи до збоїв завдяки автоматичному відновленню роботи та перенаправленню запитів на активні вузли; ефективне використання ресурсів із можливістю їх динамічного масштабування, що знизило витрати на інфраструктуру на 15-25%; здатність прогнозувати пікові навантаження та завчасно активувати додаткові ресурси, що дозволило уникнути перевантажень.

ДОДАТОК Б

Програмні коди сценаріїв

Б.1 Код для балансування навантаження

```
import asyncio
import random
# Черга для балансування навантаження
queue = asyncio.Queue()
# Сервер: отримує повідомлення та обробляє їх
async def server(reader, writer):
    addr = writer.get_extra_info('peername')
    print(f"Підключено клієнта: {addr}")
    while True:
        data = await reader.read(100)
        if not data:
            break
        message = data.decode()
        print(f"Отримано від {addr}: {message}")
        # Додаємо завдання до черги
        await queue.put((addr, message))
    print(f"Клієнт відключився: {addr}")
    writer.close()
    await writer.wait_closed()
# Обробник завдань із черги
async def task_processor():
    while True:
        addr, message = await queue.get()
        print(f"Обробляємо повідомлення від {addr}: {message}")
        await asyncio.sleep(random.uniform(0.5, 2.0))  #
Симуляція обробки
        print(f"Завдання завершено для {addr}")
# Клієнт: надсилає повідомлення серверу
async def client(message):
    reader, writer = await asyncio.open_connection('127.0.0.1',
8888)
    print(f"Відправляємо: {message}")
    writer.write(message.encode())
    await writer.drain()
    print("Повідомлення надіслано")
    writer.close()
    await writer.wait_closed()
# Основна функція запуску сервера та клієнтів
async def main():
    # Запускаємо сервер
    server_coroutine = await asyncio.start_server(server,
'127.0.0.1', 8888)

    print("Сервер запущено на 127.0.0.1:8888")
```

```
    # Запускаємо обробник завдань
    asyncio.create_task(task_processor())
    # Симуляція клієнтів
    messages = ["КСМм-23-1", "Бондар І.І?", "Виконання
сценаріїв!"]
    for msg in messages:
        asyncio.create_task(client(msg))
        await asyncio.sleep(random.uniform(0.1, 0.5))  #
Невелика затримка між клієнтами
    async with server_coroutine:
        await server_coroutine.serve_forever()

# Запуск програми
asyncio.run(main())
```

Б.2 Робота з чергами

```
import asyncio
import random
from heapq import heappush, heappop
# Черга з пріоритетами для завдань
class PriorityQueue:
    def __init__(self):
        self._queue = []
        self._counter = 0  # Для унікальності кожного запису
    def put(self, priority, item):
        heappush(self._queue, (priority, self._counter, item))
        self._counter += 1
    def get(self):
        return heappop(self._queue)[-1]
    def is_empty(self):
        return len(self._queue) == 0
# Пріоритетна черга для балансування
priority_queue = PriorityQueue()
# Сервер: отримує повідомлення та додає їх у чергу із
зазначенням пріоритету
async def server(reader, writer):
    addr = writer.get_extra_info('peername')
    print(f"Підключено клієнта: {addr}")
    while True:
        data = await reader.read(100)
        if not data:
            break
        message, priority = data.decode().split(':')  #
Повідомлення з пріоритетом
        priority = int(priority)
        print(f"Отримано від {addr}: {message} з пріоритетом
{priority}")
        # Додаємо завдання до пріоритетної черги
        priority_queue.put(priority, (addr, message))
    print(f"Клієнт відключився: {addr}")
```

```
    writer.close()
    await writer.wait_closed()
# Обробник завдань із пріоритетної черги
async def task_processor():
    while True:
        if not priority_queue.is_empty():
            addr, message = priority_queue.get()
            print(f"Обробляємо повідомлення від {addr}:
{message}")
            await asyncio.sleep(random.uniform(0.5, 2.0))  #
Симуляція обробки
            print(f"Завдання завершено для {addr}")
        else:
            await asyncio.sleep(0.1)  # Чекати нових завдань
# Клієнт: надсилає повідомлення серверу з пріоритетом
async def client(message, priority):
    reader, writer = await asyncio.open_connection('127.0.0.1',
8888)
    message_with_priority = f"{message}:{priority}"
    print(f"Відправляємо: {message_with_priority}")
    writer.write(message_with_priority.encode())
    await writer.drain()
    print("Повідомлення надіслано")
    writer.close()
    await writer.wait_closed()
# Основна функція запуску сервера та клієнтів
async def main():
    # Запускаємо сервер
    server_coroutine = await asyncio.start_server(server,
'127.0.0.1', 8888)
    print("Сервер запущено на 127.0.0.1:8888")
    # Запускаємо обробник завдань
    asyncio.create_task(task_processor())

    # Симуляція клієнтів із різними пріоритетами
    tasks = [
        ("Завдання 1", 3),
        ("Термінове завдання", 1),
        ("Рядове завдання", 5),
        ("Важливе завдання", 2),
        ("Малозначне завдання", 6)
    ]
    for msg, prio in tasks:
        asyncio.create_task(client(msg, prio))
        await asyncio.sleep(random.uniform(0.1, 0.5))  #
Невелика затримка між клієнтами
    async with server_coroutine:
        await server_coroutine.serve_forever()
# Запуск програми
asyncio.run(main())
```