Міністерство освіти і науки України Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського” Факультет
інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Лабораторна робота № 9

Технології розробки програмного забезпечення

“ Взаємодія компонентів системи.”

“Download manager”

Виконав студент групи ІА–33:
Ященко К.А.

Київ 2025

Тема: Взаємодія компонентів системи
Мета: Вивчити види взаємодії додатків (Client-Server, Peer-to-Peer, Serviceoriented Architecture), та реалізувати в проєктованій системі одну із архітектур.

Зміст

Завдання

• Ознайомитись з короткими теоретичними відомостями.

• Реалізувати частину функціоналу робочої програми у вигляді класів та їхньої взаємодії для досягнення конкретних функціональних можливостей.

• Реалізувати функціонал для роботи в розподіленому оточенні відповідно до обраної теми.

• Реалізувати взаємодію розподілених частин:

– Для клієнт-серверних варіантів: реалізація клієнтської і серверної частини додатків, а також загальної частини (middleware); зв'язок клієнтської і серверної частин за допомогою WCF, TcpClient, .NETRemoting на розсуд виконавця.

– Для однорангових мереж: реалізація взаємодії клієнтських додатків за допомогою WCF Peer to peer channel.

– Для SOA додатків: реалізація сервісу, що надає послуги клієнтським застосуванням; викладання сервісу в хмару або підняття у вигляді Web Service на локальній машині; використання токенів для передачі даних про автентифікації, двостороннє шифрування.

• Підготувати звіт щодо виконання лабораторної роботи. Поданий звіт повинен містити: діаграму класів, яка представляє спроєктовану архітектуру. Навести фрагменти програмного коду, які є суттєвими для відображення реалізованої архітектури.

Теоретичні відомості

Клієнт-серверна архітектура
Клієнт-серверні додатки являють собою найпростіший варіант розподілених додатків, де виділяється два види додатків: клієнти (представляють додаток користувачеві) і сервери (використовується для зберігання і обробки даних). Розрізняють тонкі клієнти і товсті клієнти.
Тонкий клієнт – клієнт, який повністю всі операції (або більшість, пов'язаних з логікою роботи програми) передає для обробки на сервер, а сам зберігає лише візуальне уявлення одержуваних від сервера відповідей. Грубо кажучи, тонкий клієнт – набір форм відображення і канал зв'язку з сервером. Прикладом тонкого клієнта є класичні Web-застосунки.
У такому варіанті використання майже все навантаження лягає на сервер або групу серверів.
Перевагою таких моделей є простота розгортання, тому що оновлювати потрібно лише сервери і в результаті клієнти з наступними запитами автоматично будуть працювати з оновленою системою.
Товстий клієнт – антипод тонкого клієнта, більшість логіки обробки даних містить на стороні клієнта. Це сильно розвантажує сервер. Сервер в таких випадках зазвичай працює лише як точка доступу до деякого іншого ресурсу (наприклад, бази даних) або сполучна ланка з іншими клієнтськими комп'ютерами. Перевагою такого підходу є менші вимоги до серверної частини. Також перевагою, при певному підході до реалізації, є можливість працювати клієнтам без тимчасового доступу до серверу. Прикладом товстого клієнта можна назвати мобільні застосунки, або десктоп застосунки. Наприклад, Evernote, Viber, MS Outlook, комп’ютерні антивіруси, ігри, що потребують інсталяції (The Sims,

GTA, …) та інші.
Проміжним варіантом можна назвати SPA (Single Page Application) – це товсті Web-клієнти, які при старті кожен раз завантажуються з сервера, а надалі працюють з сервером через web-API. З одного боку більшу частину логіки вони відпрацьовують на клієнтський стороні, за рахунок чого зменшується серверне навантаження. Також оновлення простіше ніж для товстих клієнтів. Але такі застосунки не працюють, якщо сервер не доступний.
Клієнт-серверна взаємодія, як правило, організовується за допомогою 3-х рівневої структури: клієнтська частина, загальна частина, серверна частина. Оскільки велика частина даних загальна (класи, використовувані системою), їх прийнято виносити в загальну частину (middleware) системи.
Клієнтська частина містить візуальне відображення і логіку обробки дії користувача; код для встановлення сеансу зв'язку з сервером і виконання відповідних викликів.

Серверна частина містить основну логіку роботи програми (бізнес-логіку) або ту її частину, яка відповідає зберіганню або обміну даними між клієнтом і сервером або клієнтами.

Peer-to-Peer архітектура
Peer-to-Peer (P2P) архітектура – це модель мережевої взаємодії, в якій кожен вузол (комп'ютер або пристрій) є одночасно клієнтом і сервером. У цій архітектурі всі вузли мають рівні права та можливості для обміну даними, ресурсами або виконання завдань. На відміну від клієнт-серверної моделі, де є чітке розділення на клієнти й сервери, P2P-мережа дозволяє учасникам взаємодіяти безпосередньо, без необхідності в централізованому сервері.
Основними принципами P2P-архітектури є:
- Децентралізація – відсутність центрального сервера, що зменшує залежність від одного вузла, підвищуючи стійкість мережі до збоїв і атак.
- Рівноправність вузлів – кожен вузол може виконувати одночасно функції клієнта (отримувати ресурси) і сервера (надавати ресурси).

- Розподіл ресурсів – вузли надають доступ до своїх власних ресурсів, таких як обчислювальна потужність, дисковий простір або файли.

Основними сферами де peer-to-peer архітектура знайшла широке застосування є файлообмінники (BitTorrent), криптовалюти та інші блокчейнтехнології, інтернет телефонія та відеоконференції (Skype, Zoom), розподілені обчислення (SETI@home, BOINC).
До основних проблемних зон можна віднести безпеку, синхронізацію даних та пошук ресурсів. Через централізацію складно контролювати дані, які передаються. Ефективність пошуку даних знижується зі збільшенням кількості вузлів у мережі і для підвищеня ефективності пошуку потрібно застосовувати спеціальні алгоритми.

Сервіс-орієнтована архітектура Сервіс-орієнтована архітектура (SOA, англ. service-oriented architecture) – модульний підхід до розробки програмного забезпечення, заснований на використанні розподілених, слабо пов’язаних (англ. Loose coupling) сервісів або служб, оснащених стандартизованими інтерфейсами для взаємодії за стандартизованими протоколами [11].
Історично сервіс-орієнтована архітектура появилася як альтернатива монолітній архітектурі, в які вся система розроблялася та розгорталася як одне ціле. Програмні комплекси, розроблені відповідно до сервіс-орієнтованою архітектурою, зазвичай реалізуються як набір веб-служб (або веб-сервісів), які, як правило, взаємодіють по HTTP з використанням SOAP або REST. Ці служби надають певні бізнес-функції, наприклад, отримання інформації про наявність матеріалів на складі.
Сервіси взаємодіють між собою тільки за рахунок обміну повідомленнями, без створення спеціальних інтеграцій для доступу до однієї інформації, наприклад, до однієї бази даних. Сервіси також можуть бути реалізовані як обгортки навколо застарілої системи. Це робиться для зменшення вартості переробки системи, а також спрощення інтеграції існуючих монолітних систем в нову архітектуру.

Згідно SOA сервіси реєструються на спеціальних сервісах і будь-яка команда розробників, якій потрібен доступ може знайти їх та використовувати.

Часто реалізація SOA покладається на використання централізованого програмного компонента для обміну даними – шину даних (Enterprise Service Bus).
Мікросервісна архітектура є подальшим розвитком сервіс-орієнтованої архітектури з використанням нових напрацювань у інформаційних технологіях.

Мікро-сервісна архітектура.
Сама назва дає зрозуміти, що мікро-сервісна архітектура є підходом до створення серверного додатку як набору малих служб [11]. Це означає, що архітектура мікро-сервісів головним чином орієнтована на серверну частину, не дивлячись на те, що цей підхід так само використовується для зовнішнього інтерфейсу, де кожна служба виконується в своєму процесі і взаємодіє з іншими службами за такими протоколами, як HTTP/HTTPS, WebSockets чи AMQP. Кожен мікросервіс реалізує специфічні можливості в предметній області і свою бізнес-логіку в рамках конкретного обмеженого контексту, повинна розроблятись автономно і розвертатись незалежно.
Визначення мікросервісів із книги Ірaклі Надарейшвілі, Ронні Мітра, Метта Макларті та Майка Амундсена (О'Рейлі) «Архітектура мікросервісів»: «Мікросервіс – це компонент із чітко визначеними межами, який можна розгортати незалежно, і підтримує взаємодію за допомогою зв'язку на основі повідомлень. Архітектура мікросервісів – це стиль розробки високоавтоматизованих систем програмного забезпечення, що легко розвивати та яке складається з мікросервісів, орієнтованих на певні можливості».
Мікросервіси забезпечують чудові можливості супроводження в величезних комплексних системах з високою масштабуємістю за рахунок створення додатків, заснованих на множині незалежно розгортуючих служб з автономними життєвими циклами.

Тема проєкту

26. Download manager (iterator, command, observer, template method, composite, p2p)
Інструмент для скачування файлів з інтернету по протоколах http або https з можливістю продовження завантаження в зупиненому місці, розподілу швидкостей активним завантаженням, ведення статистики завантажень, інтеграції в основні браузери (firefox, opera, internet explorer, chrome).

Паттерн: P2P
Посилання на GitHub: https://github.com/fl1ckyexe/trpz

Хід роботи

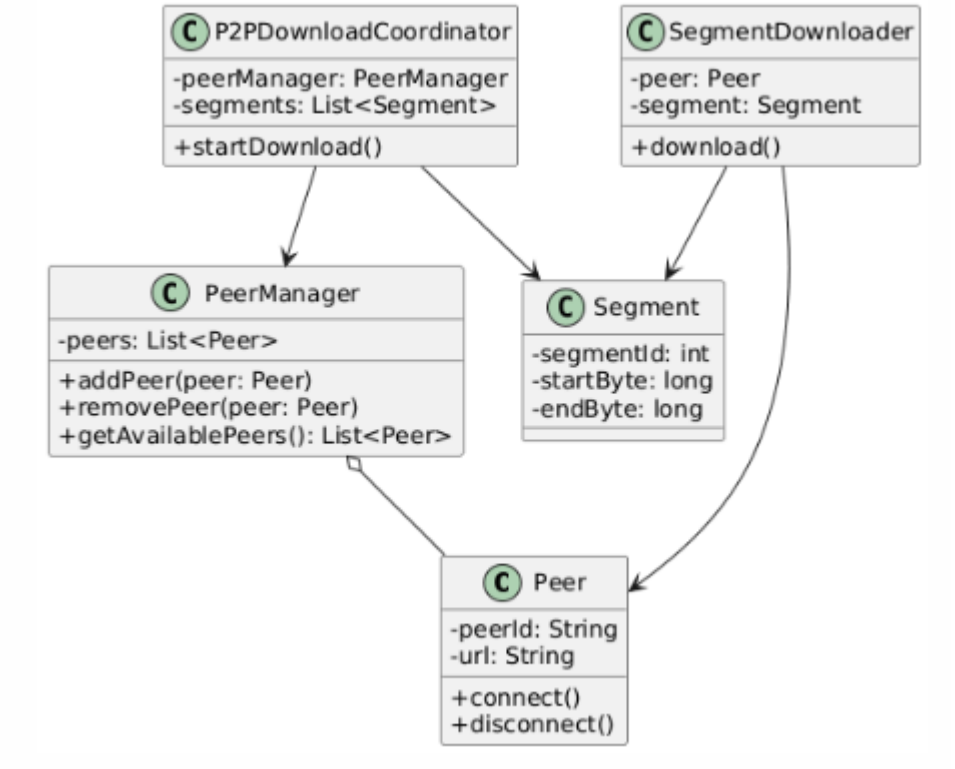

Рис. 1 – Діаграма класів для P2P

Діаграма класів відображає організацію peer-to-peer взаємодії у менеджері завантажень файлів. Основна ідея полягає у використанні декількох віддалених джерел для паралельного завантаження сегментів одного файлу, що підвищує швидкість та надійність процесу.

Основні елементи діаграми

Клас Peer представляє віддалене джерело даних, з якого можуть завантажуватися сегменти файлу. Він містить інформацію про адресу та методи встановлення з’єднання.

Клас PeerManager відповідає за зберігання та керування списком доступних peer-вузлів. Він дозволяє додавати, видаляти та отримувати список доступних джерел.

Клас Segment описує окремий фрагмент файлу, визначаючи діапазон байтів для завантаження.

Клас SegmentDownloader інкапсулює процес завантаження конкретного сегмента з конкретного peer-вузла.

Клас P2PDownloadCoordinator координує загальний процес P2P-завантаження, розподіляючи сегменти між доступними peer-вузлами.

Зв’язки та взаємодія між елементами

PeerManager містить колекцію об’єктів Peer та забезпечує доступ до них для інших компонентів системи.

P2PDownloadCoordinator використовує PeerManager для отримання доступних peer-вузлів та список сегментів для завантаження.

Для кожного сегмента координатор створює об’єкт SegmentDownloader, який встановлює з’єднання з конкретним peer та виконує HTTP range-завантаження відповідної частини файлу.

Усі peer-вузли розглядаються як рівноправні джерела даних, що відповідає принципам peer-to-peer архітектури.

Переваги використання шаблону P2P у проєкті

У межах нашого проєкту peer-to-peer підхід використовується для реалізації сегментного завантаження файлів. Файл розбивається на незалежні сегменти, які можуть паралельно завантажуватися з різних HTTP/HTTPS-джерел.

Такий підхід дозволяє підвищити швидкість завантаження, рівномірно розподіляти навантаження між джерелами та забезпечує відмовостійкість у разі недоступності окремого peer-вузла.

Застосування P2P-архітектури логічно поєднується з використанням шаблонів Composite та Template Method і створює масштабовану основу для подальшого розвитку менеджера завантажень.

Частина коду програми з P2P

Peer-to-Peer implementation for Download Manager

```
// Peer.java
public class Peer {

    private String peerId;
    private String url;

    public Peer(String peerId, String url) {
        this.peerId = peerId;
        this.url = url;
    }

    public String getPeerId() {
        return peerId;
    }

    public String getUrl() {
        return url;
    }

    public void connect() {
        // Встановлення з’єднання з peer (HTTP/HTTPS)
```

```
    }

    public void disconnect() {
        // Завершення з’єднання
    }
}

// PeerManager.java
import java.util.ArrayList;
import java.util.List;

public class PeerManager {

    private List<Peer> peers = new ArrayList<>();

    public void addPeer(Peer peer) {
        peers.add(peer);
    }

    public void removePeer(Peer peer) {
        peers.remove(peer);
    }

    public List<Peer> getAvailablePeers() {
        return peers;
    }
}

// Segment.java
public class Segment {

    private int segmentId;
    private long startByte;
    private long endByte;

    public Segment(int segmentId, long startByte, long endByte) {
        this.segmentId = segmentId;
        this.startByte = startByte;
        this.endByte = endByte;
    }

    public int getSegmentId() {
        return segmentId;
    }

    public long getStartByte() {
        return startByte;
    }
```

```
    public long getEndByte() {
        return endByte;
    }
}

// SegmentDownloader.java
public class SegmentDownloader {

    private Peer peer;
    private Segment segment;

    public SegmentDownloader(Peer peer, Segment segment) {
        this.peer = peer;
        this.segment = segment;
    }

    public void download() {
        peer.connect();
        // HTTP range-запит для завантаження байтів segment.startByte -
segment.endByte
        peer.disconnect();
    }
}

// P2PDownloadCoordinator.java
import java.util.List;

public class P2PDownloadCoordinator {

    private PeerManager peerManager;
    private List<Segment> segments;

    public P2PDownloadCoordinator(PeerManager peerManager, List<Segment>
segments) {
        this.peerManager = peerManager;
        this.segments = segments;
    }

    public void startDownload() {
        List<Peer> peers = peerManager.getAvailablePeers();

        int peerIndex = 0;
        for (Segment segment : segments) {
            Peer peer = peers.get(peerIndex % peers.size());
            SegmentDownloader downloader = new SegmentDownloader(peer, segment);
            downloader.download();
            peerIndex++;
        }
    }
```

```
}

// Main.java
import java.util.Arrays;

public class Main {
   public static void main(String[] args) {

      PeerManager peerManager = new PeerManager();
      peerManager.addPeer(new Peer("peer1", "https://mirror1.example.com/file"));
      peerManager.addPeer(new Peer("peer2", "https://mirror2.example.com/file"));

      Segment s1 = new Segment(1, 0, 1023);
      Segment s2 = new Segment(2, 1024, 2047);
      Segment s3 = new Segment(3, 2048, 3071);

      P2PDownloadCoordinator coordinator =
            new P2PDownloadCoordinator(peerManager, Arrays.asList(s1, s2, s3));

      coordinator.startDownload();
   }
}
```

Опис програмного коду

Поданий програмний код реалізує peer-to-peer підхід у межах десктопного менеджера завантажень файлів. P2P-механізм базується на сегментному завантаженні, при якому файл розбивається на незалежні частини, що можуть завантажуватися паралельно з різних джерел.

Клас Peer представляє віддалене HTTP/HTTPS-джерело даних, яке розглядається як peer-вузол. Клас PeerManager відповідає за керування списком доступних peer-вузлів.

Клас Segment описує окремий сегмент файлу, визначаючи діапазон байтів для завантаження. Клас SegmentDownloader інкапсулює логіку завантаження конкретного сегмента з конкретного peer-вузла.

Клас P2PDownloadCoordinator координує загальний процес завантаження, розподіляючи сегменти між доступними peer-вузлами за принципом рівномірного навантаження. Такий підхід забезпечує паралельність та підвищення швидкості завантаження.

Висновок

У ході виконання лабораторної роботи було реалізовано peer-to-peer підхід у менеджері завантажень файлів на основі сегментного HTTP-завантаження.

Запропонована архітектура дозволяє використовувати декілька віддалених джерел даних як рівноправні peer-вузли, що відповідає принципам P2P-систем.

Використання сегментного завантаження забезпечує підвищення швидкості, ефективний розподіл навантаження та відмовостійкість у разі недоступності окремих джерел. Реалізоване рішення логічно поєднується з іншими шаблонами проєктування, використаними у проєкті, та створює масштабовану основу для подальшого розвитку менеджера завантажень.

Відповіді на контрольні питання

1. Що таке клієнт-серверна архітектура?
Клієнт-серверна архітектура — це модель взаємодії програмних компонентів, у якій система поділяється на клієнтів та сервери. Клієнт ініціює запити на отримання даних або виконання операцій, а сервер приймає ці запити, обробляє їх і повертає результат. Сервер зазвичай централізовано зберігає дані та бізнес-логіку, а клієнт відповідає за взаємодію з користувачем.

2. Розкажіть про сервіс-орієнтовану архітектуру.
Сервіс-орієнтована архітектура (SOA) — це архітектурний підхід, за яким система складається з незалежних сервісів, що надають певну функціональність через чітко визначені інтерфейси. Кожен сервіс є автономним, може розгортатися та розвиватися окремо і взаємодіє з іншими сервісами через стандартизовані протоколи.

3. Якими принципами керується SOA?
SOA базується на принципах слабкого зв’язування, повторного використання сервісів, автономності, стандартизованих інтерфейсів, незалежності від платформи, а також можливості композиції сервісів у складні бізнес-процеси.

4. Як між собою взаємодіють сервіси в SOA?
Сервіси в SOA взаємодіють шляхом обміну повідомленнями через мережу, зазвичай використовуючи HTTP, SOAP або REST. Взаємодія відбувається через чітко описані контракти (інтерфейси), що визначають формат запитів і відповідей.

5. Як розробники взнають про існуючі сервіси і як робити до них запити?
Інформація про сервіси зазвичай зберігається у сервісних реєстрах або документації. У сучасних системах часто використовуються OpenAPI (Swagger) або WSDL для опису сервісів. Запити до сервісів виконуються через HTTP-клієнти, використовуючи визначені URL, методи та формати даних.

6. У чому полягають переваги та недоліки клієнт-серверної моделі?
Перевагами клієнт-серверної моделі є централізоване управління даними, простота адміністрування та підвищена безпека. Недоліками є залежність клієнтів від сервера, можливі проблеми масштабованості та ризик відмови всієї системи у разі збою сервера.

7. У чому полягають переваги та недоліки однорангової моделі взаємодії?
Перевагами однорангової (P2P) моделі є відсутність центрального вузла, краща масштабованість та розподілене навантаження. Недоліками є складність керування, проблеми безпеки, синхронізації та пошуку ресурсів.

8. Що таке мікросервісна архітектура?
Мікросервісна архітектура — це архітектурний стиль, у якому система складається з набору дрібних, незалежних сервісів, кожен з яких реалізує одну бізнес-функцію. Кожен мікросервіс може розгортатися, масштабуватися та оновлюватися незалежно від інших.

9. Які протоколи використовуються для обміну даними в мікросервісній архітектурі?
У мікросервісній архітектурі найчастіше використовуються HTTP/HTTPS з REST API, а також протоколи gRPC, AMQP, Kafka або інші брокери повідомлень для асинхронної взаємодії.

10. Чи можна назвати підхід сервіс-орієнтованою архітектурою, коли між шаром веб-контролерів та шаром доступу до даних реалізується шар бізнес-логіки у вигляді сервісів?
Такий підхід не є повноцінною сервіс-орієнтованою архітектурою, а скоріше є багатошаровою архітектурою. Хоча використовується поняття «сервісів», вони не є автономними, не взаємодіють через мережу та не мають окремих контрактів, що є обов’язковими ознаками SOA.