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»

«Веб-браузер»

Виконав:
студент групи ІА-32
Самойленко С. Д.

Перевірив:
Мягкий Михайло Юрійович

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 Іраклі Надарейшвілі, Ронні Мітра, Метта Маклarті 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.

Архітектура: P2P

Посилання на Github: https://github.com/brxhh/TRPZ

Хід роботи

Діаграма класів

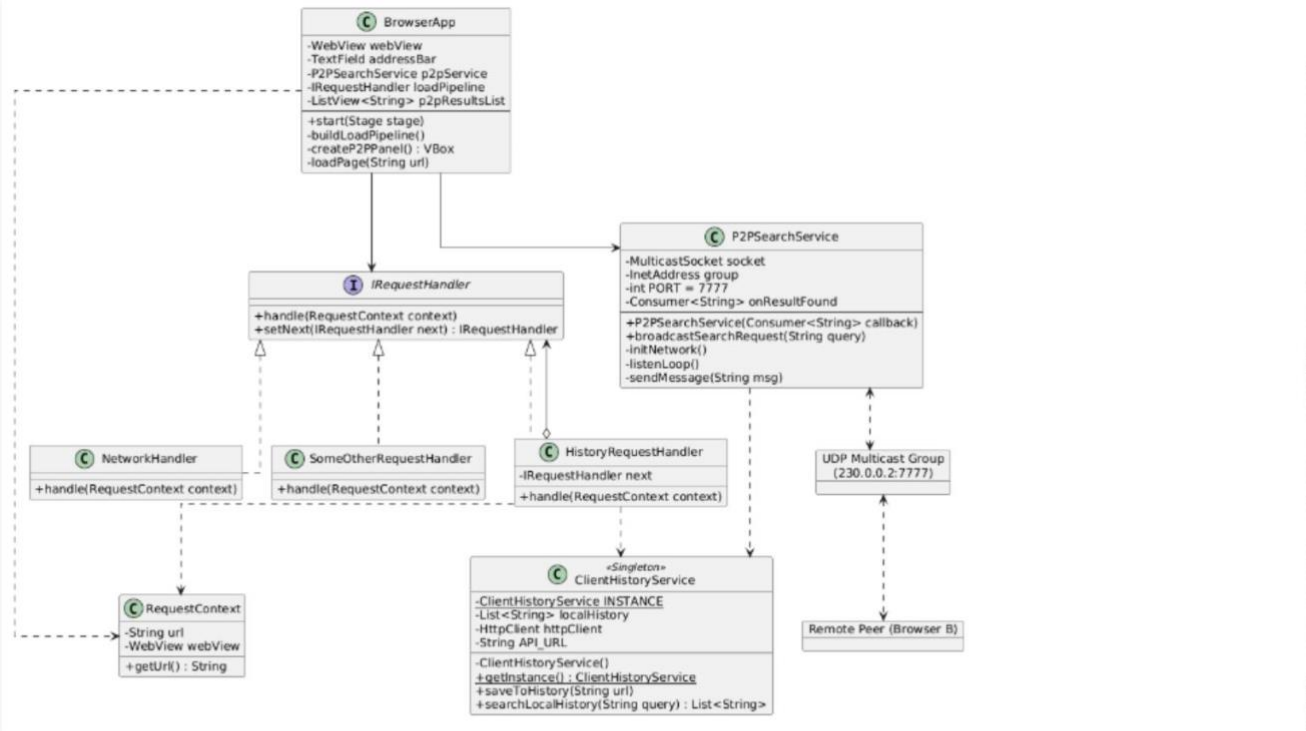

Рис.1 – Діаграма класів архітектури P2P

Опис діаграми класів

Ця діаграма класів ілюструє архітектуру розподіленого пошуку в локальній мережі, реалізовану на базі гібридної P2P-моделі. Вона демонструє відокремлення мережевої взаємодії (UDP Multicast) від бізнес-логіки збереження історії та інтерфейсу користувача. Замість того, щоб UI-компоненти напряму працювали з сокетами, ця відповідальність делегована спеціалізованому сервісу, який взаємодіє з глобальним сховищем історії через патерн "Одинак" (Singleton).

Учасники архітектури P2P та взаємодії

На діаграмі чітко видно розподіл відповідальності між ключовими класами системи:

1. **Головний Клієнт (Client Application / Controller)**
   - Клас: BrowserApp
   - Роль: Точка входу та координатор. Відповідає за ініціалізацію UI, налаштування ланцюжка обробки запитів (loadPipeline) та запуск P2P сервісу.
   - Взаємодія:
     - Ініціалізує P2PSearchService, передаючи йому логіку оновлення інтерфейсу через Callback (лямбда-вираз).
     - Обробляє дії користувача (натискання кнопки "Знайти у пірів") та делегує їх сервісам.
2. **Мережевий Сервіс P2P (P2P Network Service)**
   - Клас: P2PSearchService
   - Роль: Реалізує низькорівневу мережеву взаємодію. Він слухає UDP-порт у фоновому потоці та розсилає запити.
   - Ключова деталь:
     - Використовує MulticastSocket для приєднання до групової адреси (230.0.0.2).
     - Є повністю асинхронним: прослуховування відбувається в окремому потоці (listenLoop), а результати передаються в UI через Platform.runLater.
3. **Сервіс Даних (Data Service / Singleton)**
   - Клас: ClientHistoryService
   - Роль: "Єдине джерело правди" для історії відвідувань. Реалізує патерн Singleton, що дозволяє доступ до одних і тих самих даних з різних потоків (UI потік для запису, P2P потік для читання).
   - Взаємодія:
     - Надає метод saveToHistory() для запису нових URL.
     - Надає метод searchLocalHistory() для пошуку, який використовує P2P сервіс при отриманні запиту ззовні.
4. **Інтегратор Історії (Integration Point)**
   - Клас: HistoryRequestHandler (частина Chain of Responsibility)
   - Роль: Автоматично перехоплює навігацію користувача.
   - Взаємодія: При обробці запиту викликає ClientHistoryService.saveToHistory(), тим самим наповнюючи локальну

базу, яка стає доступною для інших учасників мережі.

5. Мережеве Оточення (Network Environment)

- Об'єкти: UDP Multicast Group, Remote Peer
- Роль: Зовнішнє середовище. Групова адреса виступає каналом зв'язку, через який повідомлення потрапляють до інших екземплярів програми (Remote Peer).

Як працює потік (The Flow)

Сценарій: Користувач А хоче знайти інформацію, яка може бути у Користувача Б.

1. Ініціація: У класі BrowserApp (Користувач А) натискається кнопка пошуку. Викликається метод p2pService.broadcastSearchRequest("query").
2. Розсилка: P2PSearchService формує пакет SEARCH_REQ:query і відправляє його в UDP Multicast Group.
3. Отримання (на іншому пристрої): На пристрої Користувача Б метод listenLoop (у фоновому потоці P2PSearchService) перехоплює цей пакет.
4. Пошук даних: Сервіс на пристрої Б звертається до свого ClientHistoryService.getInstance().searchLocalHistory("query").
5. Відповідь: Якщо дані знайдено, пристрій Б відправляє пакет SEARCH_RES:foundUrl назад у мережу.
6. Відображення: Пристрій А отримує пакет SEARCH_RES. P2PSearchService викликає збережений callback (onResultFound), який оновлює список p2pResultsList у графічному інтерфейсі.

Частина коду програми з використанням патерну Visitor

P2PSearchService.java

```
package com.edu.web.browserapp.service;

import com.edu.web.browserapp.service.impl.ClientHistoryService;
import javafx.application.Platform;
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.InetAddress;
import java.net.MulticastSocket;
import java.nio.charset.StandardCharsets;
import java.util.List;
```

```
import java.util.function.Consumer;

public class P2PSearchService {
    private static final String GROUP_ADDRESS = "230.0.0.2";
    private static final int PORT = 7777;

    private MulticastSocket socket;
    private InetAddress group;
    private boolean running = true;
    private final Consumer<String> onResultFound;

    public P2PSearchService(Consumer<String> onResultFound) {
        this.onResultFound = onResultFound;
        initNetwork();
    }

    private void initNetwork() {
        try {
            socket = new MulticastSocket(PORT);
            group = InetAddress.getByName(GROUP_ADDRESS);
            socket.joinGroup(group);

            Thread listener = new Thread(this::listenLoop);
            listener.setDaemon(true);
            listener.start();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public void broadcastSearchRequest(String query) {
        sendMessage("SEARCH_REQ:" + query);
    }

    private void sendMessage(String msg) {
        try {
            byte[] buf = msg.getBytes(StandardCharsets.UTF_8);
            DatagramPacket packet = new DatagramPacket(buf, buf.length, group, PORT);
            socket.send(packet);
        } catch (IOException e) {
            e.printStackTrace();
```

```
        }
    }

    private void listenLoop() {
        byte[] buffer = new byte[2048];
        while (running) {
            try {
                DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
                socket.receive(packet);
                String message = new String(packet.getData(), 0, packet.getLength(),
StandardCharsets.UTF_8);
                handleMessage(message);
            } catch (IOException e) {
                if(running) e.printStackTrace();
            }
        }
    }

    private void handleMessage(String message) {
        if (message.startsWith("SEARCH_REQ:")) {
            String query = message.substring(11);

            List<String> results =
ClientHistoryService.getInstance().searchLocalHistory(query);

            for (String res : results) {
                sendMessage("SEARCH_RES:" + res);
            }

        } else if (message.startsWith("SEARCH_RES:")) {
            String foundUrl = message.substring(11);
            Platform.runLater(() -> onResultFound.accept(foundUrl));
        }
    }

    public void stop() {
        running = false;
        if (socket != null && !socket.isClosed()) {
            socket.close();
        }
    }
```

}

ClientHistoryService.java

```
package com.edu.web.browserapp.service.impl;

import com.edu.web.browserapp.service.IHistoryService;

import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.stream.Collectors;

public class ClientHistoryService implements IHistoryService {

   private static final ClientHistoryService INSTANCE = new ClientHistoryService();

   private final HttpClient httpClient;
   private final String API_URL = "http://localhost:8080/browser-service-
1.0/api/history";

   private final List<String> localHistory;

   private ClientHistoryService() {
      this.httpClient = HttpClient.newBuilder()
            .version(HttpClient.Version.HTTP_1_1)
            .build();
      this.localHistory = Collections.synchronizedList(new ArrayList<>());
   }

   public static ClientHistoryService getInstance() {
      return INSTANCE;
   }

   @Override
   public void saveToHistory(String url) {
      if (!localHistory.contains(url)) {
```

```
            localHistory.add(url);
            System.out.println("[History] Saved locally for P2P: " + url);
        }

        try {
            String jsonPayload = "{\"url\":\"" + url.replace("\"", "\\\"") + "\"}";

            HttpRequest request = HttpRequest.newBuilder()
                    .uri(URI.create(API_URL))
                    .header("Content-Type", "application/json")
                    .POST(HttpRequest.BodyPublishers.ofString(jsonPayload))
                    .build();

            httpClient.sendAsync(request, HttpResponse.BodyHandlers.ofString())
                    .thenAccept(this::handleResponse)
                    .exceptionally(this::handleError);

        } catch (Exception e) {
            System.err.println("[Proxy] Error during request: " + e.getMessage());
        }
    }

    public List<String> searchLocalHistory(String query) {
        String lowerQuery = query.toLowerCase();
        synchronized (localHistory) {
            return localHistory.stream()
                    .filter(url -> url.toLowerCase().contains(lowerQuery))
                    .collect(Collectors.toList());
        }
    }

    private void handleResponse(HttpResponse<String> response) {
        if (response.statusCode() == 201 || response.statusCode() == 200) {
        } else {
            System.err.println("[Proxy] Server error: " + response.statusCode());
        }
    }

    private Void handleError(Throwable ex) {
        System.err.println("[Proxy] Warning: Central History Server unavailable (" +
ex.getMessage() + ")");
```

```
        return null;
    }
}
```

BrowserApp.java

```
package com.edu.web.browserapp;

import com.edu.web.browserapp.requestHandle.IRequestHandler;
import com.edu.web.browserapp.requestHandle.RequestContext;
import com.edu.web.browserapp.requestHandle.handler.HistoryRequestHandler;
import com.edu.web.browserapp.requestHandle.handler.NetworkHandler;
import com.edu.web.browserapp.requestHandle.handler.SomeOtherRequestHandler;
import com.edu.web.browserapp.service.P2PSearchService;
import javafx.application.Application;
import javafx.concurrent.Worker;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.input.KeyCode;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.Priority;
import javafx.scene.layout.VBox;
import javafx.scene.web.WebView;
import javafx.stage.Stage;

public class BrowserApp extends Application {

    private WebView webView;
    private TextField addressBar;
    private Label statusLabel;
    private IRequestHandler loadPipeline;

    private P2PSearchService p2pService;
    private ListView<String> p2pResultsList;

    @Override
    public void start(Stage stage) {

        webView = new WebView();
```

```
        addressBar = new TextField("https://www.google.com");
        statusLabel = new Label("Готовий");

        buildLoadPipeline();

        p2pService = new P2PSearchService(foundUrl -> {
            if (!p2pResultsList.getItems().contains(foundUrl)) {
                p2pResultsList.getItems().addFirst(foundUrl);
            }
        });

        addressBar.setOnKeyPressed(e -> {
            if (e.getCode() == KeyCode.ENTER) {
                loadPage(addressBar.getText());
            }
        });

        webView.getEngine().getLoadWorker().stateProperty().addListener((_, _,
newState) -> {
            statusLabel.setText(newState.toString());
            if (newState == Worker.State.SUCCEEDED) {
                String loadedUrl = webView.getEngine().getLocation();
                statusLabel.setText("Завантажено: " + loadedUrl);
            }
        });

        HBox topBar = new HBox(10, new Label("URL:"), addressBar);
        topBar.setPadding(new Insets(10));
        HBox.setHgrow(addressBar, Priority.ALWAYS);

        VBox rightPanel = createP2PPanel();

        BorderPane root = new BorderPane();
        root.setTop(topBar);
        root.setCenter(webView);
        root.setRight(rightPanel);
        root.setBottom(statusLabel);
        BorderPane.setMargin(statusLabel, new Insets(5));

        loadPage(addressBar.getText());
```

```
    Scene scene = new Scene(root, 1200, 768);
    stage.setTitle("Web Browser + P2P Search");
    stage.setScene(scene);
    stage.show();
}

private VBox createP2PPanel() {
    TextField searchField = new TextField();
    searchField.setPromptText("Пошук у мережі...");

    Button btnSearch = new Button("Знайти у пірів");
    p2pResultsList = new ListView<>();

    btnSearch.setOnAction(_ -> {
        String query = searchField.getText();
        if (!query.isEmpty()) {
            p2pResultsList.getItems().clear();
            p2pResultsList.getItems().add("Шукаю: " + query + "...");
            p2pService.broadcastSearchRequest(query);
        }
    });

    p2pResultsList.setOnMouseClicked(_ -> {
        String selected = p2pResultsList.getSelectionModel().getSelectedItem();
        if (selected != null && selected.startsWith("http")) {
            addressBar.setText(selected);
            loadPage(selected);
        }
    });

    VBox panel = new VBox(10,
            new Label("P2P Історія"),
            searchField,
            btnSearch,
            new Separator(),
            p2pResultsList
    );
    panel.setPadding(new Insets(10));
    panel.setPrefWidth(250);
    panel.setStyle("-fx-border-color: #ccc; -fx-border-width: 0 0 0 1px;");
    return panel;
```

```
    }

    private void loadPage(String url) {
        if (!url.startsWith("https://") && !url.startsWith("http://")) {
            url = "https://" + url;
        }

        var requestContext = new RequestContext(webView, url);

        loadPipeline.handle(requestContext);
    }

    private void buildLoadPipeline() {
        IRequestHandler historyHandler = new HistoryRequestHandler();
        IRequestHandler someOtherHandler = new SomeOtherRequestHandler();
        IRequestHandler networkHandler = new NetworkHandler();

        historyHandler.setNext(someOtherHandler)
                .setNext(networkHandler);

        loadPipeline = historyHandler;
    }

    @Override
    public void stop() throws Exception {
        if (p2pService != null) {
            p2pService.stop();
        }
        super.stop();
    }
}
```

Опис коду з використання патерну Chain of responsibility

Цей код демонструє архітектурне рішення для реалізації розподіленого пошуку інформації (Distributed Search) у локальній одноранговій мережі (P2P). Замість використання централізованого сервера для обробки пошукових запитів, логіка пошуку делегована кожному окремому клієнту. Система базується на чіткому розділенні відповідальності між шаром представлення (UI), шаром мережевої взаємодії та шаром даних, що забезпечує гнучкість та потокобезпечність.

Учасники архітектури P2P

Реалізація спирається на асинхронну обробку повідомлень та патерн "Одинак" (Singleton) для забезпечення узгодженості даних між різними потоками виконання.

1. Мережевий Сервіс (Network Service)

- Клас: P2PSearchService
- Роль: Інкапсулює низькорівневу логіку роботи з мережею (UDP Multicast) та протокол обміну повідомленнями.
- Деталі реалізації:
    - Асинхронність: Сервіс запускає окремий потік-демон (listenLoop), який безперервно слухає мережевий порт, не блокуючи основний інтерфейс користувача.
    - Протокол: Реалізує обробку текстових команд (SEARCH_REQ для запитів та SEARCH_RES для відповідей), діючи як маршрутизатор повідомлень.
    - Взаємодія з UI: Використовує функціональний інтерфейс Consumer<String> (callback) для передачі знайдених результатів назад у BrowserApp. Це дозволяє сервісу бути незалежним від конкретної реалізації графічного інтерфейсу.

2. Сервіс Даних (Shared Data Service)

- Клас: ClientHistoryService
- Роль: Виступає єдиним джерелом правди (Single Source of Truth) для історії відвідувань, реалізуючи патерн Singleton.
- Призначення: Забезпечує централізований доступ до даних історії як для локального користувача (через UI), так і для віддалених запитів (через P2P сервіс).
- Ключовий момент (Потокобезпечність):
    - Оскільки до списку localHistory одночасно звертаються потік JavaFX (при записі нових URL) та потік P2P (при читанні/пошуку), використовується синхронізована колекція Collections.synchronizedList. Це запобігає стану гонитви (Race Conditions) та помилкам конкурентного доступу.

3. Клієнт-Координатор (Client / Coordinator)

- Клас: BrowserApp
- Роль: Точка входу в застосунок, яка об'єднує UI, бізнес-логіку та мережеві сервіси.
- Взаємодія:

1. Ініціалізація: Створює екземпляр P2PSearchService та визначає поведінку при отриманні результату (додавання URL у ListView).
2. Запуск подій: Транслює дії користувача (натискання кнопки "Знайти у пірів") у виклики методу broadcastSearchRequest.
3. Інтеграція: Забезпечує наповнення історії через ланцюжок обов'язків (Chain of Responsibility), гарантуючи, що кожен відвіданий сайт автоматично потрапляє в ClientHistoryService.

Як працює потік (The Flow)

Взаємодія компонентів відбувається за сценарієм **"Запит-Відповідь" (Request-Response)** у розподіленому середовищі:

1. Користувач А (BrowserApp) вводить пошуковий запит. Додаток викликає p2pService.broadcastSearchRequest().
2. P2PSearchService формує UDP-пакет і розсилає його на групову адресу (Multicast).
3. Користувач Б (P2PSearchService) перехоплює пакет у фоновому потоці.
4. Сервіс на стороні Б звертається до свого ClientHistoryService (searchLocalHistory), щоб перевірити наявність збігів у локальній базі.
5. Якщо дані знайдено, P2PSearchService відправляє пакет-відповідь назад у мережу.
6. BrowserApp (на стороні А) отримує відповідь через callback і оновлює список результатів, використовуючи Platform.runLater для безпечної роботи з графічним інтерфейсом JavaFX.

Переваги архітектурного підходу

1. Слабка зв'язність (Loose Coupling): Клас P2PSearchService нічого не знає про структуру BrowserApp або ListView. Він просто передає рядок через інтерфейс Consumer. Це дозволяє легко змінити UI без переписування мережевого коду.
2. Відмовостійкість (Fault Tolerance): Завдяки використанню локального кешу в ClientHistoryService, система P2P продовжує працювати навіть якщо центральний сервер історії недоступний. Кожен клієнт є автономним носієм даних.
3. Масштабованість пошуку: Навантаження на пошук розподіляється між усіма учасниками мережі. Додавання нових клієнтів збільшує загальну базу знань мережі без необхідності модернізації центрального сервера.

Контрольні запитання

1. Що таке клієнт-серверна архітектура?

Клієнт-серверна архітектура являє собою модель взаємодії в комп'ютерній мережі, де завдання розподіляються між постачальниками ресурсів або послуг, яких називають серверами, та замовниками цих послуг, яких називають клієнтами. У цій моделі сервер є потужнішим вузлом, який зберігає дані, керує доступом до мережевих ресурсів, виконує обчислювальні операції та очікує на запити. Клієнт, у свою чергу, є ініціатором взаємодії, відправляючи запит на сервер для отримання певної інформації або виконання дії, після чого очікує на відповідь. Головною особливістю є централізація зберігання даних та управління на стороні сервера, що дозволяє багатьом клієнтам користуватися спільними ресурсами одночасно, не піклуючись про їхню внутрішню реалізацію чи підтримку цілісності.

2. Розкажіть про сервіс-орієнтовану архітектуру.

Сервіс-орієнтована архітектура (SOA) — це архітектурний підхід до розробки програмного забезпечення, який базується на використанні сервісів як основних компонентів для побудови додатків. У цій моделі програмне забезпечення розбивається на окремі, незалежні функціональні одиниці, які називаються сервісами, що виконують конкретні бізнес-завдання та взаємодіють між собою через мережу. Кожен сервіс є самодостатнім і надає свою функціональність іншим компонентам через чітко визначені інтерфейси, незалежно від того, на якій платформі або мові програмування він реалізований. Це дозволяє компаніям створювати гнучкі системи, де нові бізнес-процеси можна формувати шляхом комбінування та повторного використання вже існуючих сервісів без необхідності переписувати код з нуля.

3. Якими принципами керується SOA?

SOA керується набором ключових принципів, серед яких найважливішим є слабка зв'язність, що означає мінімальну залежність сервісів один від одного та від деталей їхньої реалізації. Іншим важливим принципом є стандартизація сервісного контракту, який чітко описує, що робить сервіс і як з ним взаємодіяти, приховуючи при цьому внутрішню логіку роботи за принципом абстракції. Також архітектура передбачає повторне використання сервісів у різних бізнес-процесах та їхню автономність, що дозволяє кожному модулю контролювати власне середовище виконання та ресурси. Крім того, важливим є принцип відсутності стану, коли сервіс не зберігає інформацію про попередні запити клієнта між викликами, що значно підвищує масштабованість системи, та

принцип можливості виявлення, завдяки якому споживачі можуть автоматично знаходити необхідні сервіси через спеціальні реєстри.

4. Як між собою взаємодіють сервіси в SOA?

Взаємодія сервісів у SOA відбувається через обмін повідомленнями за визначеними протоколами, що дозволяє їм спілкуватися незалежно від базових платформ. Зазвичай для цього використовується проміжний шар, відомий як сервісна шина підприємства (Enterprise Service Bus, ESB), яка бере на себе завдання маршрутизації повідомлень, перетворення форматів даних та забезпечення безпеки. Сервіси можуть взаємодіяти синхронно, очікуючи негайної відповіді, або асинхронно через черги повідомлень, коли відправник не блокує свою роботу в очікуванні результату. Сама комунікація найчастіше базується на стандартах на кшталт SOAP (Simple Object Access Protocol) з використанням XML або більш легковагових REST-архітектурах з використанням JSON, але головне, що взаємодія завжди відбувається через строго визначені інтерфейси-контракти.

5. Як розробники взнають про існуючі сервіси і як робити до них запити?

Розробники дізнаються про наявні сервіси та їхні можливості за допомогою механізму виявлення сервісів, який часто реалізується через спеціалізовані реєстри або репозиторії сервісів, наприклад, UDDI (Universal Description, Discovery, and Integration). У такому реєстрі зберігається інформація про розташування сервісу, його призначення та посилання на його технічний опис. Щоб зробити запит, розробник повинен ознайомитися з сервісним контрактом, який зазвичай представлений у вигляді WSDL-файлу (для SOAP) або специфікації OpenAPI/Swagger (для REST), де детально описані доступні методи, структура вхідних та вихідних даних, а також мережева адреса кінцевої точки. На основі цього контракту часто генеруються клієнтські проксі-класи, які дозволяють звертатися до віддаленого сервісу так само просто, як до локального методу в коді.

6. У чому полягають переваги та недоліки клієнт-серверної моделі?

Перевагою клієнт-серверної моделі є централізоване управління даними та безпекою, оскільки всі ресурси знаходяться на сервері, що полегшує резервне копіювання, контроль доступу та адміністрування. Також це дозволяє розвантажити клієнтські машини, переклавши важкі обчислення на потужний сервер, і спрощує оновлення програмного забезпечення, адже зміни часто достатньо внести лише на серверній стороні. Однак суттєвим недоліком є наявність єдиної точки відмови: якщо сервер виходить з ладу, робота всієї

мережі клієнтів зупиняється. Крім того, при великій кількості одночасних запитів сервер може стати "вузьким місцем", що призводить до падіння продуктивності, а масштабування системи часто вимагає значних фінансових витрат на модернізацію апаратного забезпечення сервера.

7. У чому полягають переваги та недоліки однорангової моделі взаємодії?

Головною перевагою однорангової (P2P) моделі є висока масштабованість та відмовостійкість, оскільки в мережі відсутній центральний сервер, і вихід з ладу одного або декількох вузлів не зупиняє роботу всієї системи. Кожен новий учасник мережі додає свої обчислювальні ресурси та пам'ять, що робить мережу потужнішою зі зростанням кількості користувачів, а також знижує витрати на утримання дорогого серверного обладнання. Недоліками цієї моделі є складність управління та адміністрування, оскільки дані децентралізовані, і немає єдиного центру для забезпечення безпеки чи оновлення. Також у P2P мережах важче гарантувати швидкість доступу до даних та їх цілісність, оскільки ресурси вузлів можуть бути нестабільними, а їхня доступність — непередбачуваною.

8. Що таке мікро-сервісна архітектура?

Мікросервісна архітектура — це підхід до розробки програмного забезпечення, при якому додаток будується як сукупність невеликих, незалежних сервісів, кожен з яких працює у власному процесі та виконує одну конкретну бізнес-функцію. Ці сервіси слабо пов'язані між собою і можуть розроблятися, розгортатися та масштабуватися абсолютно незалежно, часто різними командами та з використанням різних технологій або мов програмування. На відміну від монолітної архітектури, де всі компоненти тісно переплетені в єдину програму, мікросервіси мають власні бази даних і спілкуються між собою за допомогою легковагових протоколів, що дозволяє досягти високої гнучкості та швидкості впровадження змін у великих системах.

9. Які протоколи використовуються для обміну даними в мікросервісній архітектурі?

Для обміну даними в мікросервісній архітектурі найчастіше використовуються легковагові протоколи, які забезпечують швидку та просту комунікацію. Найпопулярнішим є протокол HTTP/HTTPS у поєднанні з архітектурним стилем REST, де дані передаються у зручному форматі JSON, що є стандартом для синхронної взаємодії. Для більш високопродуктивних синхронних викликів також використовується gRPC, який базується на HTTP/2 та форматі Protocol Buffers. У випадках, коли необхідна асинхронна взаємодія, широко застосовуються протоколи черг повідомлень, такі як AMQP (використовується в

RabbitMQ) або специфічні бінарні протоколи брокерів повідомлень на кшталт Kafka, що дозволяє сервісам обмінюватися подіями без прямого зв'язку один з одним.

10. Чи можна назвати підхід сервіс-орієнтованою архітектурою, коли ми в проєкті між шаром веб-контролерів та шаром доступу до даних реалізуємо шар бізнес-логіки у вигляді сервісів?

Ні, такий підхід не можна назвати повноцінною сервіс-орієнтованою архітектурою (SOA), це скоріше реалізація патерну "Шар сервісів" (Service Layer) у рамках монолітної архітектури або модульного моноліту. Ключова відмінність полягає в тому, що в SOA сервіси є фізично відокремленими компонентами, які можуть працювати на різних серверах, мають власні життєві цикли та спілкуються через мережу за допомогою визначених контрактів. У описаному ж випадку "сервіси" є просто класами або модулями в межах одного процесу пам'яті та однієї програми, які взаємодіють через прямі виклики методів мови програмування, а не через мережеві протоколи, тому це є лише способом структурування коду, а не розподіленою архітектурою SOA.

Висновки

У ході виконання лабораторної роботи було закріплено навички проєктування та реалізації розподілених програмних систем на прикладі веб-браузера з гібридною архітектурою. Практична розробка засвідчила ефективність поєднання різних архітектурних стилів (клієнт-серверного та однорангового) для досягнення функціональної гнучкості та надійності. Зокрема, реалізація клієнт-серверної взаємодії дозволила централізувати зберігання історії відвідувань через REST API, забезпечивши персистентність даних. Водночас впровадження архітектури P2P на базі протоколу UDP Multicast вирішило задачу децентралізованого пошуку інформації, дозволивши клієнтам обмінюватися ресурсами напряму, без участі центрального сервера, що значно підвищило відмовостійкість системи. Внутрішню логіку додатку було структуровано за допомогою патерну Chain of Responsibility, який уніфікував процес обробки URL-запитів, валідацію та обробку HTTP-помилок, а використання Singleton гарантувало узгодженість даних між мережевими потоками та UI. У результаті було створено масштабований додаток, який відповідає принципам сервіс-орієнтованої архітектури, забезпечує чітке розмежування локальної та мережевої логіки, а також демонструє переваги роботи в розподіленому середовищі.