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 розроблення програмного забезпечення
«РІЗНІ ВИДИ ВЗАЄМОДІЇ ДОДАТКІВ: CLIENT-SERVER, PEER-TO-PEER, SERVICE-ORIENTED ARCHITECTURE»

Виконала:

студентка групи ІА-23

Єрмак Д. Р

Перевірив:

Мягкий М. Ю.

Київ 2024

Тема лабораторних робіт:

IRC client (singleton, builder, abstract factory, template method, composite, client-server)

Клієнт для IRC-чатів з можливістю вказівки порту і адреси з'єднання, підтримка базових команд (підключення до чату, створення чату, установка імені, реєстрація, допомога і т.д.), отримання метаданих про канал.

Завдання:

1. Ознайомитися з короткими теоретичними відомостями.
2. Реалізувати частину функціонала робочої програми у вигляді класів і їх взаємодій для досягнення конкретних функціональних можливостей.
3. Реалізувати взаємодію програми в одній з архітектур відповідно до обраної теми

Зміст

Хід роботи:

Крок 1. Теоретичні відомості

CLIENT-SERVER

Клієнт-серверні додатки являють собою найпростіший варіант розподілених додатків, де виділяється два види додатків: клієнти і сервери. Вони поділяються на два типи клієнтів: тонкі, які виконують мінімум обчислень і передають основну логіку на сервер, та товсті, де більша частина обробки відбувається на клієнтській стороні, що зменшує навантаження на сервер. Такі системи зазвичай мають три рівні: клієнтську частину для взаємодії з користувачем, загальну частину (middleware) для обміну даними, та серверну частину, яка реалізує бізнес-логіку. Моделі взаємодії можуть включати підписку/видачу для ефективного оновлення даних на клієнтах.

PEER-TO-PEER

Модель взаємодії типу peer-to-peer передбачає рівноправ'я клієнтських програм, де відсутня серверна частина, і всі клієнти взаємодіють між собою для досягнення спільних цілей. Проблеми, що виникають у таких мережах, включають синхронізацію даних та пошук клієнтських застосувань, для чого використовуються методи, як-от структуровані однорангові мережі та

алгоритми синхронізації даних. Для ефективного обміну повідомленнями часто розробляються спеціалізовані протоколи та формати даних.

SERVICE-ORIENTED ARCHITECTURE

Сервіс-орієнтована архітектура передбачає використання модульних компонентів з незалежними інтерфейсами для взаємодії через стандартизовані протоколи, що забезпечує масштабованість та незалежність від платформ. Модель SaaS (Software as a service) ґрунтується на SOA і дозволяє користувачам орендувати програмне забезпечення через Інтернет без необхідності в установці та підтримці, з можливістю гнучкої оплати і оновлень.

MICROSERVICE

Мікросервісна архітектура дозволяє створювати серверні додатки як набір незалежних служб, кожна з яких реалізує свою бізнес-логіку та розгортається автономно. Це забезпечує високу масштабованість і гнучкість, дозволяючи швидше вносити зміни та оновлення в окремі частини системи без впливу на інші. Такий підхід також полегшує супроводження великих і складних систем у довгостроковій перспективі.

Крок 2. Реалізація клієнт-серверної архітектури

Для реалізації програми IRClient використано архітектуру клієнт-сервер, де сервер відповідає за обробку запитів від кількох клієнтів, а клієнт здійснює взаємодію через сокетне з'єднання. Клієнтська частина працює з сервером, надсилаючи запити через потоки вводу та виводу, а серверна частина керує підключеннями клієнтів і транслює повідомлення всім підключеним користувачам. Основна роль сервера — це обробка запитів, передача повідомлень між клієнтами та підтримка одночасної взаємодії кількох користувачів.

Такий підхід дозволяє масштабувати систему, додаючи нових клієнтів без необхідності змінювати серверну логіку, забезпечуючи при цьому ефективну обробку багатьох підключень. Через збереження списку активних клієнтів сервер має можливість транслювати повідомлення на всі підключення, що дає змогу всім клієнтам бачити нові повідомлення в реальному часі. Цей клієнт-серверний підхід забезпечує зручність масштабування, дозволяючи гнучко додавати новий функціонал без змін в основній архітектурі програми.

```java
package org.example.ClientServer;

import java.io.*;
import java.net.*;
import java.util.Scanner;

public class Client {
    private static final String SERVER_ADDRESS = "localhost"; 1 usage
    private static final int SERVER_PORT = 8080; 1 usage

    public static void main(String[] args) {
        try (Socket socket = new Socket(SERVER_ADDRESS, SERVER_PORT);
             BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
             PrintWriter out = new PrintWriter(socket.getOutputStream(), autoFlush: true);
             Scanner scanner = new Scanner(System.in)) {

            Thread receiverThread = new Thread(() -> {
                try {
                    String message;
                    while ((message = in.readLine()) != null) {
                        System.out.println(message);
                    }
                } catch (IOException e) {
                    System.out.println("Error connecting to the server");
                }
            });
            receiverThread.start();

            System.out.println("The connection to the IRC server is established");
            System.out.print("Enter your nickname: ");
            String username = scanner.nextLine();
            out.println(username);

            while (true) {
                String userInput = scanner.nextLine();
                if ("/quit".equalsIgnoreCase(userInput)) {
                    break;
                }
                out.println(userInput);
            }
        } catch (IOException e) {
            System.out.println("Error connecting to the server: " + e.getMessage());
        }
    }
}
```

Рис. 1 – Код класу Client

```java
package org.example.ClientServer;

import java.io.*;
import java.net.*;
import java.util.*;

public class Server {
    private static final int PORT = 8080;  2 usages
    private static final Set<PrintWriter> clientWriters = new HashSet<>();  6 usages

    public static void main(String[] args) {
        System.out.println("The IRC server is running on the port " + PORT);
        try (ServerSocket serverSocket = new ServerSocket(PORT)) {
            while (true) {
                try {
                    Socket clientSocket = serverSocket.accept();
                    new ClientHandler(clientSocket).start();
                } catch (IOException e) {
                    System.err.println("Error connecting the client: " + e.getMessage());
                }
            }
        } catch (IOException e) {
            System.err.println("Failed to start server: " + e.getMessage());
        }
    }


    private static class ClientHandler extends Thread {  1 usage
        private final Socket socket;  2 usages
        private PrintWriter out;  4 usages
        private BufferedReader in;  3 usages

        public ClientHandler(Socket socket) {  1 usage
            this.socket = socket;
            try {
                in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
                out = new PrintWriter(socket.getOutputStream(),  autoFlush: true);
            } catch (IOException e) {
                System.err.println("Error initializing client handler: " + e.getMessage());
                e.printStackTrace(System.err);
            }
        }

        public void run() {
            try {
                out.println("Enter your nickname:");
                String username = in.readLine();
                synchronized (clientWriters) {
                    clientWriters.add(out);
                }
                broadcastMessage("User " + username + " joined the chat");

                String message;
                while ((message = in.readLine()) != null) {
                    if (message.equalsIgnoreCase( anotherString: "/quit")) {
                        break;
                    }
                    broadcastMessage(username + ": " + message);
                }

                synchronized (clientWriters) {
                    clientWriters.remove(out);
                }
                socket.close();
                broadcastMessage("User " + username + " left the channel");
            } catch (IOException e) {
                System.err.println("Error handling client: " + e.getMessage());
            }
        }

        private void broadcastMessage(String message) {  3 usages
            synchronized (clientWriters) {
                for (PrintWriter writer : clientWriters) {
                    writer.println(message);
                }
            }
        }
    }
}
```

Рис. 2 – Код класу Server

У даній реалізації використано архітектурний підхід клієнт-сервер, де сервер відповідає за обробку запитів від кількох клієнтів, а клієнт взаємодіє з сервером через сокетне з'єднання. Клієнт та сервер працюють за допомогою потоків вводу та виводу, що забезпечує ефективне управління комунікацією в реальному часі.

Переваги використання цієї архітектури:

1. Централізоване управління з’єднаннями: Сервер управляє всіма підключеними клієнтами, що дозволяє централізовано керувати усіма запитами та транслювати повідомлення між користувачами, забезпечуючи зручність масштабування та обробки багатьох підключень.
2. Гнучкість у додаванні нових клієнтів: Завдяки серверу, кожен новий клієнт може підключатися без необхідності змінювати серверну частину коду, що забезпечує гнучкість і зручність в управлінні числом підключених користувачів.
3. Спрощення взаємодії: Завдяки використанню окремих потоків для кожного клієнта сервер може ефективно обробляти кожне підключення незалежно, дозволяючи кожному клієнту отримувати повідомлення без впливу на інших. Це дозволяє зменшити складність коду та покращити продуктивність системи.
4. Масштабованість: Сервер здатен обробляти багатокористувацькі запити без змін у клієнтському коді, що дозволяє ефективно масштабувати додаток з часом, додаючи нових користувачів без негативного впливу на поточну архітектуру.
5. Легка модифікація та підтримка: Додавання нових функцій може бути здійсненим без зміни клієнтського коду, що забезпечує простоту в підтримці та розширенні системи.

Цей підхід дозволяє ефективно обробляти запити від кількох клієнтів та забезпечує централізовану комунікацію, що значно спрощує підтримку та розширення додатка.

Крок 3. Зображення клієнт-серверної структури архітектури

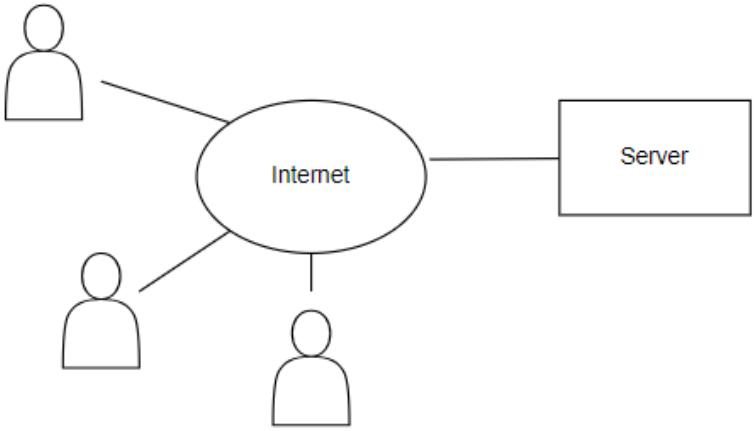

Рис. 3 – Діаграма клієнт-серверної архітектури

Архітектура клієнт-сервер забезпечує розподіл задач між двома основними компонентами: клієнтом, який ініціює запити, і сервером, який їх обробляє. У цьому прикладі сервер визначає єдиний інтерфейс для роботи з запитами від клієнтів, обробляючи їх в окремих потоках. Клієнтські компоненти, відправка запитів і приймання відповідей, реалізують базову функціональність, що дозволяє надсилати дані на сервер, але не підтримують логіку обробки запитів безпосередньо на клієнті.

Сервер, у свою чергу, зберігає логіку обробки запитів у вигляді потоків, які можуть обробляти кілька запитів одночасно. Ці компоненти відповідають за приймання запитів від клієнтів, їх обробку і відправлення результатів назад. Загальна структура забезпечує уніфікований підхід до роботи з запитами незалежно від їх складності. Кожен запит обробляється сервером незалежно, а для клієнта важливо лише відправити запит і отримати відповідь.

Такий підхід спрощує масштабованість, оскільки сервер може обробляти нові типи запитів або додавати нові обробники без зміни клієнтської частини програми. Також модель клієнт-сервер знижує дублювання коду, дозволяючи централізовано керувати логікою обробки запитів на сервері і розподіляти обов'язки між клієнтом і сервером відповідно до їх ролей.

Код можна переглянути в даному репозиторії:
https://github.com/konrelityw/irc_chat

Висновок: Під час виконання даної лабораторної роботи було досліджено структуру, призначення та основні властивості різних видів взаємодії додатків: CLIENT-SERVER, PEER-TO-PEER, SERVICE-ORIENTED, MICROSERVICE

ARCHITECTURE. Наведені архітектури мають свої особисті переваги та недоліки, а також використовуються для досягнення різних цілей. Для реалізації частини майбутньої системи було обрано архітектуру клієнт-сервер, яка найкраще підходить у даному випадку. Далі було реалізовано обрану архітектуру, розглянуто її призначення, структуру та досліджено її переваги та недоліки у використанні.