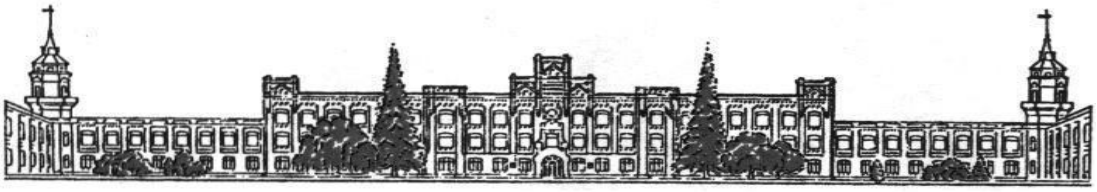

Міністерство освіти і науки України

Національний технічний університет України

«Київський політехнічний інститут ім. Ігоря Сікорського»

Інститут прикладного системного аналізу

Кафедра системного проектування

Лабораторна робота №4

з дисципліни “Паралельні обчислення”

на тему: “**Клієнт-серверний додаток для базових математичних операцій над масивом**”

Виконала:

студентка III курсу, групи ДА-01

Балуматкіна О. В.

Варіант 4

Прийняла(в):

Київ-2023

Зміст

Мета роботи

Реалізувати двосторонній клієнт-серверний додаток знайомою мовою програмування, навчитись розробляти протоколи прикладного рівня з врахуванням умов завдання, набути навичок реалізації підтримки кількох клієнтів одночасно на одному сервері.

Завдання

1. Розробити клієнт-серверний додаток для вирішення завдання з першої лабораторної роботи, передавши масив даних з клієнта на сервер, а потім – отримавши результат назад на сторону клієнта.
2. Розробити протокол прикладного рівня для взаємодії клієнта та сервера, для цього врахувати декілька кроків в процесі взаємодії:
    a. Надсилання даних та конфігурації обчислень;
    b. Надсилання команди для початку обчислень;
    c. Запит результату з двома можливими варіантами відповіді: поточний статус (якщо результат ще обраховується) та отримання остаточного результату.
3. Зробити підтримку декількох клієнтів одночасно.

Варіант 4

Створити вектор з N >= 1000 елементами випадкових чисел. Знайти моду та медіану цього вектора.

Хід роботи

Загальний опис

Одразу можна зазначити, що саме завдання пошуку моди та медіани певного вектору випадкових чисел є доволі простим для реалізації, на відміну від, до прикладу, матриць, тому одразу можна написати клас для реалізації логіки цих обчислень, код якого буде розміщено у файлі Calculation.java (повний код наведено у додатках до протоколу).

Таким чином, у Calculation створюємо звичайну логіку пошуку медіани:

```
private void calcMedian() {
    List<Double> sublist = vector.subList(startIndex, endIndex);
    Double[] vectorChunk = sublist.toArray(new Double[0]);

    Arrays.sort(vectorChunk);
    int middle = vectorChunk.length / 2;
    if (vectorChunk.length % 2 == 0) {
        median = (vectorChunk[middle - 1] + vectorChunk[middle]) / 2.0;
    } else {
        median = vectorChunk[middle];
    }
}
```

А також логіку пошуку моди у послідовності чисел:

```
private void calcMode() {
    List<Double> sublist = vector.subList(startIndex, endIndex); // Add 1 to
include endIndex
    Double[] vectorChunk = sublist.toArray(new Double[0]);

    Map<Double, Double> freqMap = new HashMap<>();
    // Iterate through the array and update the frequency of each number in the
HashMap
    for (double num : vectorChunk) {
        if (freqMap.containsKey(num)) {
            freqMap.put(num, freqMap.get(num) + 1.0);
        } else {
            freqMap.put(num, 1.0);
        }
    }

    // Find the mode and its frequency
    for (Map.Entry<Double, Double> entry : freqMap.entrySet()) {
        if (entry.getValue() > counter) {
            mode = entry.getKey();
            counter = entry.getValue();
        }
    }
}
```

Додатковими методами прописуємо вивід результатів з даного класу, і, в цілому, з найлегшим впорались.

Наступним кроком створюємо клас код для серверу – відповідний файл Server.java. Пам’ятаємо, що одним з завдань даної лабораторної роботи було підтримка декількох клієнтів на одному сервері, тобто на кожного нового клієнта необхідно створювати новий потік, а самого клієнта перенаправляти на опрацювання до класу ClientHandler.

ClientHandler.java містить в собі код повної логіки роботи з, як зрозуміло по назві, клієнтом:

- Працює з сокетом клієнта;
- Наводить список операцій, які може виконати сервер;
- На кожній ітерації отримує повідомлення від клієнта у вигляді номера операції, потрібної клієнту;
- Відповідно до обраної операції повертає клієнту повідомлення-контексти, підтверджуючи правильність виконання задач, а також отримує від клієнта данні (конфігурації, масив чисел);
- У відповідь на запит від клієнта здатен повертати як проміжне значення моди та медіани, так і фінальне (але, зважаючи на завдання з лабораторної роботи у вигляді отримання сервером від клієнта команди на початок обчислень, проміжне значення «схопити» дуже важко);
- Фінально, на запит від клієнта від’єднує клієнта від серверу.

Також клас ClientHandler містить у собі всю логіку для багатопоточного пошуку моди та медіани з отриманого набору чисел, реалізовано це через пул потоків на визначену кількість потоків (встановлюється клієнтом), список завдань на кожен потік, а також подальше обчислення медіани з набору медіан,

що отримуються в результаті роботи кожного потоку, і порівняння лічильників моди з кожного потоку для повернення значення фінальної моди вектору (все це представлено у методі processVector).

Надсилання конфігурацій

Від клієнта вимагається надсилання “1” як бажаної операції на виконання, після чого клієнт також вводить довжину вектора і кількість потоків для ведення розрахунків, а клієнт, власне, зчитує ці значення для подальшого отримання даного вектору випадкових значень та проведень розрахунків.

Зі сторони серверу – код ClientHandler.java.

```
case "1" -> {
    serverResponse(dos, "Option 1. Getting configurations.");

    String[] configs = dis.readUTF().split(" ");
    length = Integer.parseInt(configs[0]);
    nThreads = Integer.parseInt(configs[1]);

    serverResponse(dos, "Received data:\nArray length: "
            + length + "\nThreads amount: " + nThreads + "\n");
}
```

Зі сторони клієнту – код Client.java.

```
case "1" -> {
    serverRead(dis);

    while (!(arrayLength > 0)) {
        System.out.println("Print array length:");
        arrayLength = scanner.nextInt();
    }

    while (!(nThreads > 0)) {
        System.out.println("Print threads amount:");
        nThreads = scanner.nextInt();
    }
    scanner.nextLine();

    dos.writeUTF(arrayLength + " " + nThreads);

    serverRead(dis);
}
```

Надсилання даних у вигляді масиву випадкових чисел

Для надсилання якогось масиву даних його спочатку треба створити, що виконується при активації опції “2” на стороні клієнта – створюється масив

довжини, вказаної в попередній дії (або 0 за замовчанням), його заповнення випадковими числами передається на виконання методу generateRandDoubleArrList, після чого клієнт передає на сервер кожен елемент згенерованого набору.

Зі сторони серверу – код ClientHandler.java.

```
case "2" -> {
    array.clear();
    serverResponse(dos, "Option 2. Getting array.");

    for (int i = 0; i < length; i++) {
        array.add(dis.readDouble());
    }

    serverResponse(dos, "Array successfully received.\n");
}
```

Зі сторони клієнту – код Client.java.

```
case "2" -> {
    List<Double> array = generateRandDoubleArrList(arrayLength);

    serverRead(dis);

    for (double num : array) {
        dos.writeDouble(num);
    }

    serverRead(dis);
}
```

Команда початку обчислень

Як вже згадувалось, також була реалізована команда для початку обчислень, оскільки дані, в цілому, можуть передаватися з клієнта на сервер не за один раз, а за декілька (у випадку, якщо клієнт буде двічі чи більше разів використовувати опції “1” та “2”), реалізовано це було однією з опцій та через switch-case.

Зі сторони серверу – код ClientHandler.java.

```
case "3" -> {
    serverResponse(dos, "Option 3. Starting processing.\n");

    // Future starting
    int finalNumbThreads = nThreads;
    future = CompletableFuture.supplyAsync(() ->
```

```
            processVector(array, finalNumbThreads));
}
```

Зі сторони клієнту – код Client.java.

```
case "3" ->
        serverRead(dis);
```

Проміжний та фінальний результати обчислень

Оскільки код було написано одразу для четвертої та п’ятої лабораторних робіт, для реалізації можливості отримати доступ до проміжних результатів використовується CompletableFuture з логікою, за якою змінна future переходить між потоками, що виконують багатопоточне обчислення моди та медіани отриманого набору чисел, та, відповідно, зберігає у собі результати виконання цих потоків, внаслідок чого в будь який момент можна «вихопити» поточний результат, але виключно, коли цей future є доступним до зчитування та містить в собі якийсь результат, в протилежному випадку сервер просто надасть клієнту відповідь, що обрахунки ще не готові.

Зі сторони серверу – код ClientHandler.java.

```
case "4" -> {
    serverResponse(dos, "Option 4. Getting results.");

    if (future != null && future.isDone()) {
        Double[] result = future.getNow(null);
        serverResponse(dos, "Result:\nMode: " + result[0] +
                "\nMedian: " + result[1] + "\n");
        future = null;
    } else {
        serverResponse(dos, "At the moment results are not ready.\n");
    }
}
```

Зі сторони клієнту – код Client.java.

```
case "4" -> {
    serverRead(dis);
    serverRead(dis);
}
```

Підтримка кількох клієнтів

Про підтримку декількох клієнтів вже було згадано раніше, вся реалізація знаходиться в класі Server, а логіка полягає у назначенні кожному новому клієнту

окремого потоку та відповідного ClientHandler'у для цього клієнта, внаслідок чого набори чисел з різних клієнтів, як і результати обрахунків, не перемішуються між собою, а сервер продовжує свою роботу з першим клієнтом навіть коли другий запросив результати і вже готовий від'єднатись.

```
try {
    ServerSocket serverSocket = new ServerSocket(portNumb);
    System.out.println("Server started: " + serverSocket);
    System.out.println("Waiting for client..\n");

    while (true) {
        Socket clientSocket = null;
        try {
            clientSocket = serverSocket.accept();
            System.out.println("Client connected: " + clientSocket + "\n");

            // Multi client handle
            ClientHandler clientHandler = new ClientHandler(clientSocket);
            Thread thread = new Thread(clientHandler);
            thread.start();

        } catch (Exception e) {
            clientSocket.close();
            e.printStackTrace();
        }
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

Client-Server взаємодія

Клієнт створює сокет для обміну інформацією: Socket: clientSocket.

Сервер створює сокет для обміну інформацією: ServerSocket: serverSocket.

Client → встановлює зв'язок з сервером, сервер та клієнт підтверджують, що зв'язок встановлено;

Server → Надсилає клієнту список команд, номери яких клієнту необхідно передати на сервер для отримання відповідних результатів, очікує наступної команди від клієнта;

Client → Надсилає на сервер номер бажаної команди (почнемо з «1»);

Server → Активує виконання першого блоку, запитує в клієнта довжину масиву та кількість потоків для проведення розрахунків над масивом;

Client → Надсилає на сервер номер два значення – довжину масиву та кількість потоків для обчислень;

Server → Отримує очікувані значення, зберігає їх на своїй стороні, очікує наступної команди від клієнта;

Client → Надсилає на сервер номер бажаної команди («2»);

Server → Активує виконання другого блоку, запитує в клієнта передачу масиву даних, що є в клієнта, саме того розміру, значення якого було передано на сервер при виконанні першого блоку;

Client → Надсилає на сервер весь наявний масив;

Server → Отримує масив та зберігає його на своїй стороні, підтверджує отримання масиву, очікує наступної команди від клієнта;

Client → Надсилає на сервер номер бажаної команди («3»);

Server → Активує виконання третього блоку, надсилає клієнту підтвердження про виконання, починає виконання підрахунків над отриманим масивом даним, використовуючи встановлену клієнтом кількість потоків для обчислень та записуючи результати обрахунків в масив типу CompletableFuture<Double[]>;

Client → Отримує від серверу підтвердження про виконання третього блоку – блоку початку обчислень;

Server → Очікує наступної команди від клієнта;

Client → Надсилає на сервер номер бажаної команди («4»);

Server → Активує виконання четвертого блоку, що відповідає за отримання клієнтом результату обчислень на будь-якому етапі, у випадку, якщо якісь підрахунки будь-яким потоком вже були виконанні і Future об’єкт готовий до зчитування, сервер надає клієнту поточні значення, або, в іншому випадку, надає повідомлення про неготовність результатів на даному моменті, тож клієнту потрібно повторити запит на отримання результатів

Client → Отримує від серверу або результати обрахунків на поточному етапі (який може бути як проміжним, так і фінальним), або повідомлення про необхідність повторити запит на отримання результатів

Server → Очікує наступної команди від клієнта;

Client → Надсилає на сервер номер бажаної команди («5»);

Server → Активує виконання п’ятого блоку, надсилає клієнту підтвердження про активацію цього блоку, закриває зв’язок між собою та клієнтом, що запросив виконання п’ятого блоку;

Client → Отримує від серверу підтвердження про виконання п’ятого блоку, зв’язок з сервером закривається, сокет клієнта закривається;

Server → Очікує наступного клієнта;

Server → При термінації виконання закривається сокет сервера та усі допоміжні механізми припиняють свою роботу.

Висновок

У ході виконання лабораторної роботи було набуто навички по створенню клієнт-серверного додатку мовою програмування Java, набуто знань про особливості передачі даних між сокетом клієнта та сокетом сервера. Також було поглиблено знання про роботу протоколу TCP, механізм Socket, розроблено власний протокол прикладного рівня для взаємодії клієнта з сервером та навпаки, реалізовано підтримку кількох клієнтів на сервері одночасно без необхідності завершувати сесію попереднього клієнта, а також поглиблено загальні знання по влаштуванню комп'ютерних мереж.

Посилання на гіт: https://github.com/balumatkina/parallel-computing-kpi.git

Додаток А

Server.java

```
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

public class Server {

    public Server(int portNumb) {
        try {
            ServerSocket serverSocket = new ServerSocket(portNumb);
            System.out.println("Server started: " + serverSocket);
            System.out.println("Waiting for client..\n");

            while (true) {
                Socket clientSocket = null;
                try {
                    clientSocket = serverSocket.accept();
                    System.out.println("Client connected: " + clientSocket +
"\n");

                    // Multi client handle
                    ClientHandler clientHandler = new
ClientHandler(clientSocket);
                    Thread thread = new Thread(clientHandler);
                    thread.start();

                } catch (Exception e) {
                    clientSocket.close();
                    e.printStackTrace();
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        new Server(6060);
    }
}
```

Додаток Б

ClientHandler.java

```
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.net.Socket;
import java.util.*;
import java.util.concurrent.*;

public class ClientHandler implements Runnable {
    private Socket clSocket;
    private ExecutorService executor;
    private DataOutputStream dos;
    private DataInputStream dis;

    public ClientHandler(Socket clSocket) throws IOException {
        this.clSocket = clSocket;
        dis = new DataInputStream(clSocket.getInputStream());
        dos = new DataOutputStream(clSocket.getOutputStream());
    }

        @Override
        public void run () {
        try {
            CompletableFuture<Double[]> future = new CompletableFuture<>();
            int length = 0, nThreads = 0;
            boolean isConnected = true;
            String clientCommand;
            ArrayList<Double> array = new ArrayList<>();

            String serverCommands = """
                    This server provides calculations of mode
                    and median of a given vector of numbers.
                    Options:
                    1. Setting configurations (vector length, number of
threads);
                    2. Sending a vector to the server;
                    3. Starting processing;
                    4. Getting results of counted mode and median;
                    5. Disconnecting from the server.
                    """;
            dos.writeUTF(serverCommands);

            while (isConnected) {
                clientCommand = dis.readUTF();
                switch (clientCommand) {
                    case "1" -> {
                        serverResponse(dos, "Option 1. Getting
configurations.");

                        String[] configs = dis.readUTF().split(" ");
                        length = Integer.parseInt(configs[0]);
                        nThreads = Integer.parseInt(configs[1]);

                        serverResponse(dos, "Received data:\nArray length: "
                                + length + "\nThreads amount: " + nThreads +
```

```
"\n");
                    }
                    case "2" -> {
                        array.clear();
                        serverResponse(dos, "Option 2. Getting array.");

                        for (int i = 0; i < length; i++) {
                            array.add(dis.readDouble());
                        }

                        serverResponse(dos, "Array successfully received.\n");
                    }
                    case "3" -> {
                        serverResponse(dos, "Option 3. Starting processing.\n");

                        // Future starting
                        int finalNumbThreads = nThreads;
                        future = CompletableFuture.supplyAsync(() ->
                                processVector(array, finalNumbThreads));
                    }
                    case "4" -> {
                        serverResponse(dos, "Option 4. Getting results.");

                        if (future != null && future.isDone()) {
                            Double[] result = future.getNow(null);
                            serverResponse(dos, "Result:\nMode: " + result[0] +
                                    "\nMedian: " + result[1] + "\n");
                            future = null;
                        } else {
                            serverResponse(dos, "At the moment results are not
ready.\n");
                        }
                    }
                    case "5" -> {
                        serverResponse(dos, "Option 5. Breaking connection.");

                        isConnected = false;
                        if (executor != null) {
                            executor.shutdown();
                        }

                        serverResponse(dos, "Connection stopped.\n");
                    }
                    default -> System.err.println("Unknown operation.");
                }
            }

            try {
                if (dis != null) {
                    dis.close();
                }
                if (dos != null) {
                    dos.close();
                }
                if (clSocket != null) {
                    clSocket.close();
                }
                System.out.println("Client closed: " + clSocket);
            } catch (IOException e) {
```

```
                e.printStackTrace();
            }

        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public Double[] processVector(ArrayList<Double> array, int nThreads) {
        Double mode = 0.0, counter = 0.0, finalMedian = 0.0;
        int length = array.size();

        executor = Executors.newFixedThreadPool(nThreads);

        ArrayList<Double> medianArray = new ArrayList<>();
        // Task list for different threads
        List<Calculation> tasks = new ArrayList<>();
        // For Future
        CompletionService<Double[]> completionService = new
ExecutorCompletionService<>(executor);

        for (int i = 0; i < nThreads; i++) {
            int startIndex = length / nThreads * i;
            int endIndex = (i == (nThreads - 1)) ? length : (length / nThreads *
(i + 1));

            endIndex = Math.min(endIndex, length);

            tasks.add(new Calculation(array, startIndex, endIndex));
        }

        try {
            // Submitting tasks for processing
            for (int i = 0; i < nThreads; i++) {
                completionService.submit(tasks.get(i));
            }

            mode = array.get(0);
            finalMedian = array.get(0);
            for (int i = 0; i < nThreads; i++) {
                Future<Double[]> completedFuture = completionService.take();
                Double[] res = completedFuture.get();

                if (res[2] > counter) {
                    mode = res[1];
                    counter = res[2];
                }

                medianArray.add(res[0]);
                Collections.sort(medianArray);
                int middle = medianArray.size() / 2;
                if (medianArray.size() % 2 == 0) {
                    finalMedian = (medianArray.get(middle - 1) +
medianArray.get(middle)) / 2.0;
                } else {
                    finalMedian = medianArray.get(middle);
                }

            }
```

```
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        }

        return new Double[]{mode, finalMedian};
    }

    private static void serverResponse(DataOutputStream dos, String message)
throws IOException {
        dos.writeUTF(message);
        System.out.println(message);
    }
}
```

Додаток В

Calculation.java

```
import java.util.*;
import java.util.concurrent.*;

public class Calculation implements Callable<Double[]> {

    private final ArrayList<Double> vector;
    private final int startIndex;
    private final int endIndex;
    private double mode;
    private double median;
    private Double counter = 0.0;

    public Calculation(ArrayList<Double> vector, int startIndex, int endIndex) {
        this.vector = vector;
        this.startIndex = startIndex;
        this.endIndex = endIndex;
    }

    @Override
    public Double[] call() {
        return getModeMedian();
    }

    private void calcMedian() {
        List<Double> sublist = vector.subList(startIndex, endIndex);
        Double[] vectorChunk = sublist.toArray(new Double[0]);

        Arrays.sort(vectorChunk);
        int middle = vectorChunk.length / 2;
        if (vectorChunk.length % 2 == 0) {
            median = (vectorChunk[middle - 1] + vectorChunk[middle]) / 2.0;
        } else {
            median = vectorChunk[middle];
        }
    }

    private void calcMode() {
        List<Double> sublist = vector.subList(startIndex, endIndex); // Add 1 to
include endIndex
        Double[] vectorChunk = sublist.toArray(new Double[0]);

        Map<Double, Double> freqMap = new HashMap<>();
        // Iterate through the array and update the frequency of each number in
the HashMap
        for (double num : vectorChunk) {
            if (freqMap.containsKey(num)) {
                freqMap.put(num, freqMap.get(num) + 1.0);
            } else {
                freqMap.put(num, 1.0);
            }
        }

        // Find the mode and its frequency
        for (Map.Entry<Double, Double> entry : freqMap.entrySet()) {
```

```
            if (entry.getValue() > counter) {
                mode = entry.getKey();
                counter = entry.getValue();
            }
        }
    }

    private Double[] getModeMedian() {
        calcMode();
        calcMedian();
        return new Double[] {median, mode, counter};
    }
}
```

Додаток Г

Client.java

```
import java.io.*;
import java.net.Socket;
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import java.util.Scanner;

public class Client {
    private Socket socket;
    private DataOutputStream dos;
    private DataInputStream dis;

    public Client (String address, int port) {
        Scanner scanner = new Scanner(System.in);
        int arrayLength = 0, nThreads = 0;
        try {
            socket = new Socket(address, port);
            System.out.println("The client is connected\n" + socket +"\n");

            dos = new DataOutputStream(socket.getOutputStream());
            dis = new DataInputStream(socket.getInputStream());
        } catch (IOException u) {
            u.printStackTrace();
        }

        try {
            serverRead(dis);

            while (true) {
                System.out.println("Enter command number (1-5): ");
                String command = scanner.nextLine();

                dos.writeUTF(command);

                switch (command) {
                    case "1" -> {
                        serverRead(dis);

                        while (!(arrayLength > 0)) {
                            System.out.println("Print array length:");
                            arrayLength = scanner.nextInt();
                        }

                        while (!(nThreads > 0)) {
                            System.out.println("Print threads amount:");
                            nThreads = scanner.nextInt();
                        }
                        scanner.nextLine();

                        dos.writeUTF(arrayLength + " " + nThreads);

                        serverRead(dis);
                    }
                    case "2" -> {
```

```
                        List<Double> array =
generateRandDoubleArrList(arrayLength);

                        serverRead(dis);

                        for (double num : array) {
                            dos.writeDouble(num);
                        }

                        serverRead(dis);
                    }
                    case "3" ->
                            serverRead(dis);
                    case "4" -> {
                        serverRead(dis);
                        serverRead(dis);
                    }
                    case "5" -> {
                        serverRead(dis);
                        serverRead(dis);

                        try {
                            if (dis != null) {
                                dis.close();
                            }
                            if (dos != null) {
                                dos.close();
                            }
                            if (socket != null) {
                                socket.close();
                            }
                            if (scanner != null) {
                                scanner.close();
                            }
                        } catch (IOException e) {
                            e.printStackTrace();
                        }

                        return;
                    }
                    default -> System.out.println("Invalid operation number.
Please try again.");
                }
            }

        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
//        Client client = new Client("localhost", 6060);
        try {
            new Client("localhost", 6060);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

```
    private static List<Double> generateRandDoubleArrList (int arrLength) {
        Random random = new Random();
        List<Double> array = new ArrayList<>();

        for (int i = 0; i < arrLength; i++) {
            array.add(random.nextDouble(100));
        }
        return array;
    }

    private static void serverRead(DataInputStream dis) throws IOException {
        String serverResponse = dis.readUTF();
        System.out.println("SERVER: " + serverResponse);
    }

}
```

Контрольні питання:

1. *Поясніть різницю протоколів TCP та UDP.*
    a. З'єднання та надійність передачі даних:
        i. TCP забезпечує надійну передачу даних, встановлюючи з'єднання між двома комп'ютерами і контролюючи передачу пакетів даних. Він використовує підтвердження та повторну передачу для гарантованої доставки даних. TCP забезпечує послідовну доставку даних та контролює поток, а також виявлення та виправлення помилок.
        ii. UDP є безз'єднаним протоколом і не надає гарантій доставки даних. Він передає пакети даних без будь-якої перевірки на доставку або повторну передачу. UDP підходить для ситуацій, коли швидкість та ефективність важливіші за надійність, наприклад, в реальному часі потокових медіа або онлайн-іграх.
    b. Контроль потоку та розмір пакетів:
        i. TCP дані в такому форматі, в якому отримав, не розбиваючи їх на менші частини. Це дозволяє UDP бути швидшим за TCP, але також може призвести до втрати даних або дублювання пакетів.
    c. Отримання даних та портовий номер:
        i. У TCP передача даних здійснюється відправкою потоку байтів, які можуть бути розділені на пакети під час передачі. Кожен пакет TCP має портовий номер, що дозволяє отримувачеві правильно адресувати та збирати дані.
        ii. У UDP дані передаються у вигляді окремих пакетів. Кожен пакет має портовий номер, який дозволяє програмам розрізняти та обробляти дані. Оскільки UDP є безз'єднаним

протоколом, пакети можуть приходити в різному порядку або бути втрачені.

2. *Що таке протокол прикладного рівня (application protocol)?* Протокол прикладного рівня (application protocol) є частиною стеку протоколів інтернету, яка визначає комунікацію між програмами або пристроями, що працюють на різних комп'ютерах в мережі. Цей рівень стеку протоколів відповідає за обмін даними інтерфейсу програми-клієнта та програми-сервера, які взаємодіють у мережевому середовищі. Протоколи прикладного рівня визначають формати даних, правила передачі та семантику комунікації між програмами. Вони дозволяють програмам виконувати певні функції, такі як обмін повідомленнями електронної пошти, передача файлів, доступ до веб-ресурсів, передача мультимедійних даних тощо. Кожен протокол прикладного рівня має свій власний набір правил та специфікацій, які визначають, як програми повинні обмінюватися даними. Наприклад, HTTP (Hypertext Transfer Protocol) є одним з найбільш відомих протоколів прикладного рівня, використовуваних для передачі веб-сторінок та інших ресурсів у Всесвітній павутині. SMTP (Simple Mail Transfer Protocol) використовується для передачі електронних листів, а FTP (File Transfer Protocol) - для передачі файлів між комп'ютерами у мережі. Протоколи прикладного рівня встановлюються на транспортному рівні (наприклад, використовуючи TCP або UDP), і вони залежать від функцій, які надаються транспортним протоколом
3. *Поясність різницю між big endian та little endian нотаціями.* Big-endian та little-endian - це два різних способи представлення та зберігання числових даних у комп'ютерних системах, особливо в пам'яті комп'ютера. *Big-endian* означає, що найбільш значущі байти (старші байти) числа зберігаються у початкових адресах пам'яті, а найменш значущі байти

(молодші байти) зберігаються у наступних адресах. Значення читаються зліва направо. Це подібно до способу, яким ми записуємо числа - найбільш значуща цифра (старша) знаходиться ліворуч. Наприклад, якщо ми маємо 32-бітне беззнакове ціле число 0x12345678, то його збереження у пам'яті залежить від формату: Big-endian: 12 34 56 78 (початкова адреса - найбільш значущий байт); Little-endian: 78 56 34 12 (початкова адреса - найменш значущий байт). Little-endian означає, що найменш значущі байти зберігаються у початкових адресах пам'яті, а найбільш значущі байти - у наступних адресах. Значення також читаються зліва направо. Цей спосіб збереження байтів є альтернативним до big-endian.

4. *Як на стороні сервера організувати підтримку декількох клієнтів одночасно?*

   Threads: Створення окремих потоків в програмі сервера для обробки кожного клієнта. Кожен новий клієнт, що підключається до сервера, буде обслуговуватися окремим потоком, що дозволяє обробляти їх паралельно. Використання потоків потребує управління синхронізацією та захистом ресурсів від конфліктів.

   Processes: Створення окремих процесів для обслуговування клієнтів. Кожен процес виконується в окремому адресному просторі та має власний набір ресурсів, що дозволяє обслуговувати клієнтів незалежно один від одного. Однак, спілкування між процесами може бути складним, і використання процесів може призвести до великого навантаження на систему.

   Connection Pooling: Створення пулу з'єднань з обмеженою кількістю з'єднань, які можуть бути використані для обслуговування клієнтів. Коли клієнт підключається до сервера, він отримує доступ до вільного з'єднання з пулу. Це може зменшити витрати на створення та розрив з'єднань, але може бути обмеженою щодо кількості одночасних підключень.

5. *Яку інформацію необхідно вказувати при створенні сокетів?*
   Домен (Domain): Домен визначає простір імен, в якому будуть функціонувати сокети. Найпоширеніші домени включають AF_INET для мережевих сокетів IPv4 і AF_INET6 для мережевих сокетів IPv6.
   Тип (Type): Тип сокету визначає характеристики з'єднання, такі як забезпечення надійності, з'єднання чи безз'єднаність. Декілька поширених типів включають SOCK_STREAM для з'єднаної передачі даних (надійний транспортний протокол, такий як TCP), SOCK_DGRAM для безз'єднаної передачі даних (ненадійний транспортний протокол, такий як UDP) та SOCK_RAW для доступу до рівня IP.
   Протокол (Protocol): Протокол вказує конкретний протокол, який буде використовуватися для комунікації. Наприклад, для TCP/IP можна вказати IPPROTO_TCP, а для UDP/IP - IPPROTO_UDP. Зазвичай встановлюється значення 0, що дозволяє системі автоматично вибрати протокол, відповідний до вказаного типу сокету.
   Залежно від мови програмування або бібліотеки сокетів, можуть бути й інші параметри, такі як *адреса та порт, режим блокування, опції сокету* тощо.
6. *Поясніть необхідність вказування порту при створенні сокету.*
   Вказування порту при створенні сокету є необхідним для встановлення комунікації між сокетами на різних комп'ютерах у мережі. Порт є ідентифікатором, який вказує на конкретний мережевий канал або службу, з якою клієнт або сервер бажає взаємодіяти.
   Основні причини вказання порту включають:
   Ідентифікація служби: Порт дозволяє ідентифікувати конкретну службу або програму, з якою клієнт або сервер планує взаємодіяти. Наприклад, HTTP-сервери використовують порт 80, FTP-сервери використовують порт 21, SMTP-сервери використовують порт 25 і т.д. При встановленні

з'єднання клієнт повинен знати, на якому порті служби прослуховують запити, щоб спілкуватися з нею.

Розподіл ресурсів: Використання портів дозволяє серверам розподіляти надходження запитів між різними службами або процесами. Кожен порт може бути призначений для конкретної служби або процесу, і сервер може відправити отримані запити на відповідний порт для обробки.

Багатопроцесові або багатопотокові сервери: В багатопроцесових або багатопотокових серверах, кожний клієнт може бути обслуговований окремим процесом або потоком. Кожен з цих процесів або потоків може використовувати власний порт для взаємодії з клієнтом. Порт дозволяє ідентифікувати, якому процесу або потоку на сервері належить конкретне з'єднання.