Харківський національний університет імені В. Н. Каразіна
Факультет комп'ютерних наук
Кафедра штучного інтелекту та програмного забезпечення

# ЗВІТ
# З ЛАБОРАТОРНОЇ РОБОТИ №5

дисципліна: «Крос-платформне програмування»

Виконав: студент групи КС23
Травченко Сергій Миколайович

Перевірив: доцент кафедри ШІтаПЗ
Споров Олександр Євгенович

Харків
2024

**Завдання №2,1** Напишіть простий розподілений клієнт / серверний додаток за допомогою TCP

сокетів. У цьому додатку сервер приймає завдання від клієнтів, виконує ці завдання, визначає

час їх виконання і потім повертає всю цю інформацію клієнту. При цьому саме клієнти

створюють свої власні завдання і відправляють їх на сервер для виконання (клас завдання

повинен реалізовувати інтерфейс, визначений відповідно до договору із сервером).

Визначення класу завдання відправляється клієнтом на сервер і, щойно class - файл стає

доступним, сервер може виконувати отримане завдання. Аналогічно, сервер створює об’єкт

класу результату і відправляє його разом із визначенням класу клієнта. При цьому клас

результату реалізує інтерфейс, відомий клієнту.

В якості клієнтського завдання можна взяти завдання обчислення факторіала

достатньо великого числа.

Коротко розглянемо основні моменти щодо організації цього додатку.

Особливістю цього додатку є те, що сервер може виконувати завдання клієнтів без будь-якого попереднього знання про клас завдання, а клієнти можуть отримувати результат

без будь-яких попередніх знань про клас результату. Необхідні для роботи серверної та

клієнтської сторони класи передаються через сокет.

Цей додаток повинен включати клієнтську, серверну сторони та інтерфейси, які будуть

доступні як серверу, так і клієнтам.

Спочатку у новому проекті створюємо пакет для зберігання інтерфейсів interfaces.

У цьому пакеті спочатку визначимо інтерфейс Executable, який визначає те, які методи має

виконувати завдання. Клієнти створюватимуть свої класи завдань, які мають реалізовувати

цей інтерфейс, і надсилатимуть їх за допомогою механізму серіалізації серверу для

виконання. Сервер десеріалізує об’єкт завдання та викличе метод, визначений в інтерфейсі

Executable.

```
public interface Executable {

public Object execute();

}
```

Цей інтерфейс містить єдиний метод execute(). Це той метод, який виконуватиме

сервер після отримання завдання від клієнта. Цей метод повертатиме результат, як об’єкт

типу Object.

За умовою завдання сервер виконує завдання клієнта, визначає час виконання,

створює об’єкт результату та за допомогою механізму серіалізації відправляє його назад

клієнту. Для цього слід визначити інтерфейс Result, який описує структуру результату, що

повертається після виконання завдання.

```
public interface Result {

public Object output();

public double scoreTime();
```

}

Даний інтерфейс визначає два методи: output(), що повертає результат виконання

завдання, та scoreTime(), що повертає час виконання завдання. Клас об’єкта-результату

виконання завдання має реалізувати цей інтерфейс.

Оскільки об’єкти типів Executable і Result передаються через мережу із

застосуванням механізму серіалізації об’єктів Java, то класи, що реалізують інтерфейси

Executable і Result, повинні ще реалізовувати якийсь із інтерфейсів серіалізації,

наприклад java.io.Serializable.

Створимо два пакети server та client для розміщення серверних та клієнтських

класів, відповідно.

У пакеті для класів клієнта створимо клас завдання, що має структуру простого

JavaBeans компонента.

```
public class JobOne implements Executable, Serializable {

private final static long serialVersionUID = -1L;

private int n;

.....

@Override

public Object execute() {

BigInteger res = BigInteger.ONE;

.....
```

```
return res;

}

}
```

Аналогічно, у пакеті для класів сервера створимо клас результату.

```
public class ResultImpl implements Result, Serializable {

Object output;

double scoreTime;

public ResultImpl(Object o, double c) {

output = o;

scoreTime = c;

}

public Object output() {

return output;

}

public double scoreTime() {

return scoreTime;

}

}
```

Після цього можна переходити до створення сервера та клієнта. Структура клієнта та

сервера аналогічна тій, що була розглянута в попередньому завданні. Розглянемо основні дії,

які потрібно виконати з боку сервера:

```
//Створюємо об’єктний потік вводу для прийому інформації від клієнта
```

```
ObjectInputStream in = new ObjectInputStream(clientSocket.getInputStream());

//Отримуємо ім’я class файлу завдання і зберігаємо його у файл

//Важливо: зберегти його в “правильне” місце, щоб він був знайдений завантажувачем

String classFile = (String) in.readObject();

classFile = classFile.replaceFirst("client", "server"); //Важливо: підправити повне

ім’я

byte[] b = (byte[]) in.readObject();

FileOutputStream fos = new FileOutputStream(classFile);

fos.write(b);

//Отримуємо об’єкт - завдання і обчислюємо задачу

Executable ex = (Executable) in.readObject();

//Початок обчислень

double startTime = System.nanoTime();

Object output = ex.execute();

double endTime = System.nanoTime();

double completionTime = endTime - startTime;

//Завершення обчислень

//Формування об’єкта — результату і відправка class файлу і самого об’єкта клієнту

ResultImpl r = new ResultImpl(output, completionTime);

ObjectOutputStream out = new ObjectOutputStream(clientSocket.getOutputStream());

classFile = ".....class";

out.writeObject(classFile);
```

```
FileInputStream fis = new FileInputStream(classFile);
byte[] bo = new byte[fis.available()];
fis.read(bo);
out.writeObject(bo);
out.writeObject(r);
```

Аналогічно представимо основні дії, які має виконати клієнт:

```
//Встановлюємо з’єднання з сервером
Socket client = new Socket(host, port);
//Створюємо об’єктний потік виводу для передачі завдання серверу
ObjectOutputStream out = new ObjectOutputStream(client.getOutputStream());
//Передаємо ім’я class – файлу і сам class – файл на сервер
String classFile = ".....class";
out.writeObject(classFile);
FileInputStream fis = new FileInputStream(classFile);
byte[] b = new byte[fis.available()];
fis.read(b);
out.writeObject(b);
//Формуємо об’єкт — завдання для обчислень
int num = Integer.parseInt(textFieldN.getText());
JobOne aJob = new JobOne(num);
//Передаємо об’єкт — завдання на сервер
out.writeObject(aJob);
```

//Створюємо об’єктний потік вводу для отримання результату

ObjectInputStream in = new ObjectInputStream(client.getInputStream());

//Отримуємо ім’я class – файлу результату та сам class – файл і зберігаємо його у файл

//Важливо: зберегти його у “правильне” місце, щоб він був знайдений завантажувачем

classFile = (String) in.readObject();

.....

b = (byte[]) in.readObject();

FileOutputStream fos = new FileOutputStream(classFile);

fos.write(b);

//Отримуємо і десеріалізуємо об’єкт - результат

Result r = (Result) in.readObject();

//Виводимо результати розрахунків і час обчислення

showText("result = " + r.output() + ", time taken = " + r.scoreTime() + "ns");

Як уже згадувалося, схема побудови клієнта та сервера аналогічна попередньому

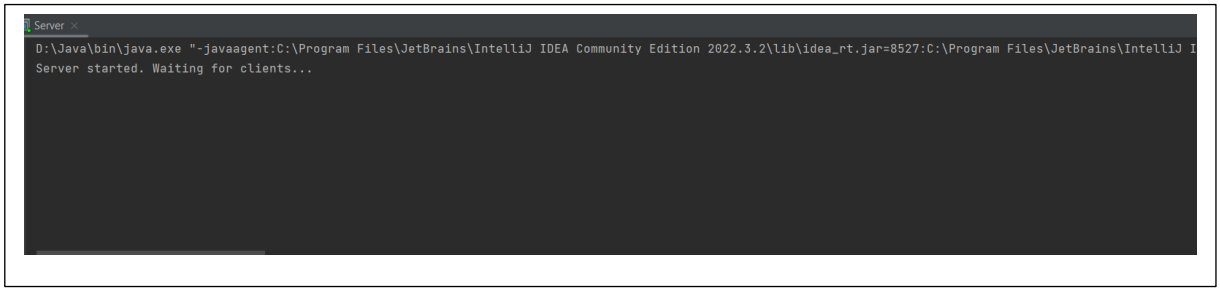
завданню. Такий розподілений додаток можна оформити як консольну програму, а можна як

додаток з графічним інтерфейсом користувача.

Лістинги на гітхабі.

Результати виконання завдання №2 наведено:

1. На малюнку 1.1 – результат виконання програми.




Малюнок 1.1 – результат виконання програми

**Завдання 1** Розглянемо взаємодію двох комп'ютерів у мережі. Перш ніж розпочати роботу,
комп'ютери повинні обмінятися IP-адресами. Це може стати досить непростим завданням.
Створимо спеціальний UDP сервер, який допоможе комп'ютерам обмінятися
«координатами»: IP-адресами та номерами портів. Потім створимо UDP клієнтів, які
перевірять роботу сервера: відправлять запит, який реєструє клієнта (комп’ютер) на сервері, і
отримають відповідь сервера — список вже зареєстрованих комп'ютерів.

Результати виконання завдання №1 наведено:

1. На малюнку 2.1 – результат виконання програми.

```java
import java.io.*;
import java.net.*;

// Клас, який реалізує UDP сервер
no usages
class UDPServer {
    no usages
    public static void main(String args[]) throws Exception {
        DatagramSocket serverSocket = new DatagramSocket( port: 9876); // Створення сокету для сервера з портом 9876

        byte[] receiveData = new byte[1024];
        byte[] sendData = new byte[1024];

        while (true) {
            DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);
            serverSocket.receive(receivePacket); // Очікування отримання даних від клієнта

            String clientData = new String(receivePacket.getData()); // Отримання даних від клієнта
            InetAddress clientIP = receivePacket.getAddress(); // Отримання IP-адреси клієнта
            int clientPort = receivePacket.getPort(); // Отримання порту клієнта

            System.out.println("Received from client: " + clientData);

            String response = "Registration successful. Registered clients:\n"; // Формування відповіді сервера
            // Тут можна додати код для збереження IP-адреси та порту клієнта у серверній базі даних або списку

            sendData = response.getBytes();
            DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, clientIP, clientPort);
            serverSocket.send(sendPacket); // Відправлення відповіді клієнту
```

Малюнок 2.1 – результат виконання програми

Для того, щоб вирішити задачу, в новому проекті Java спочатку створимо пакет для
зберігання створюваних класів. Цей пакет можна назвати updWork. У цьому пакеті слід
спочатку створити допоміжні класи, призначені для зберігання інформації. Це може бути
клас User, призначений для зберігання інформації про зареєстрований комп'ютер. Цей клас
повинен зберігати Інтернет адресу зареєстрованого комп'ютера, представлену класом
InetAddress, і номер порту int port, з якого відправлявся пакет. Для зручності подальшої
роботи цей клас слід зробити серіалізованим (що реалізує інтерфейс Serializable). При
цьому рекомендується оголосити константу serialVersionUID, призначену для керування
версіями серіалізації. У даному класі потрібно оголосити конструктор за замовчуванням,
гетери та сеттери для кожного поля, а також перевизначити метод toString для зручного
відображення інформації про підключення. Крім того, слід визначити клас, який зберігає
інформацію про всі зареєстровані на даному сервері комп'ютери: клас ActiveUsers.
Інформація про зареєстровані комп'ютери може зберігатися у списку ArrayList<User>
users. Цей клас також можна зробити серіалізованим. Крім того, в ньому слід визначити
конструктор за умовчанням та методи призначені для:

- додавання інформації про підключення комп'ютера (метод add);
- організації перевірки «сховища» на пустоту (метод isEmpty);
- отримання кількості вже зареєстрованих комп'ютерів (метод size);
- перевірки, чи зареєстрований даний комп'ютер (метод contains);
- отримання зареєстрованого комп'ютера за індексом (метод get).

Крім того, слід перевизначити метод toString для зручного відображення інформації
про всі зареєстровані підключення. Наприклад, так:

```
@Override
public String toString() {
```

```
StringBuilder buf = new StringBuilder();
for (User u : users)
buf.append(u+"\n");
return buf.toString();
}
```

Можна додати й інші методи (очищення сховища тощо).

Далі створимо клас UPDServer, який представляє сервер UDP. Протокол UDP (User Datagram Protocol) є протоколом, що використовується для передачі датаграм. Це ненадійний протокол, оскільки повідомлення, що передаються за його допомогою, можуть губитися або приходити в послідовності, що відрізняється від послідовності їх відправки. При використанні цього протоколу для забезпечення надійної передачі необхідно організовувати спеціальну надбудову над цим протоколом, що забезпечує перевірку доставки, наприклад, нумерацію пакетів, повторну передачу пакетів при закінченні часу очікування і т.д. Зазвичай, застосування протоколу UDP означає, що перевірка помилок і їх виправлення просто не потрібні. Так, протокол UDP часто використовують додатки реального часу. У цьому випадку краще скинути пакети, що затрималися або втрачені, оскільки їх отримання заново може виявитися неможливим. Також протокол UDP застосовується для серверів, що відповідають на невеликі запити від величезної кількості клієнтів, наприклад DNS, для організації роботи потокових мультимедійних додатків на кшталт IPTV, Voice over IP, протоколу TFTP (Trivial File Transfer Protocol — простий протокол передачі файлів) і багатьох онлайн-ігор.

Довжина одного повідомлення (однієї датаграми) при використанні цього протоколу

обмежена 65536 байтами (причому багато реалізацій обмежують розмір датаграми 8К).

Якщо потрібно переслати дані більшого розміру, вони мають бути розбиті на шматки
відправником і знову зібрані одержувачем. Передача повідомлення — не блокуюча,
прийом — блокуючий, з можливістю переривання після закінчення часу очікування.

Для роботи з протоколом UDP у пакеті java.net визначено такі класи:

• DatagramPacket (датаграма). Конструктор цього класу приймає масив байтів і
адресу отримувача (IP-адресу вузла та порт). Клас призначений для представлення
одиничної датаграми. Цей клас використовується як при створенні повідомлення для
передачі, так і для прийому повідомлення.

• DatagramSocket. Призначений для передачі / прийому UDP датаграм. Один із
конструкторів приймає в якості аргументу порт, з яким зв’язується сокет, інший
конструктор, без аргументів, задіює в якості порту деякий вільний порт. Клас містить
методи send та receive, для, відповідно, передачі та прийому датаграм. Метод
setSoTimeout встановлює обмеження по часу (в мілісекундах) для операцій сокета.

Наш UDP сервер є простою мережевою програмою, що використовує протокол UDP
для обслуговування запитів клієнтських додатків. Для створення UDP сервера
використовується об'єкт DatagramSocket, який приймає об'єкти DatagramPacket від
клієнтів, а також формує відповідь і відправляє об'єкти DatagramPacket з результатом

запиту назад клієнтам. Для створення сервера UDP необхідно виконати такі кроки:

- створити сокет, використовуючи об'єкт DatagramSocket;
- створити об'єкт класу DatagramPacket і застосувати метод receive() для отримання запиту клієнта;
- опрацювати запит клієнта, створити об'єкт класу DatagramPacket, упакувати в нього результат запиту і застосувати метод send() для передачі повідомлення клієнту.

Цей клас буде містити метод main(), в якому буде створюватися і запускатися UDP сервер.

Таким чином, клас UPDServer може містити декілька закритих полів:

```
public class UPDServer {
private ActiveUsers userList = null; // список зареєстрованих
// комп’ютерів
private DatagramSocket socket = null; // датаграмний сокет для
// взаємодії комп’ютерів по мережі
private DatagramPacket packet = null; // датаграмний пакет для
// отримання і відправки інформації
private InetAddress address = null; // клас, який представляє мережеву
// адресу комп’ютера
private int port = -1; // номер порту
}
```

Відкритий конструктор класу UPDServer отримує номер порту, на якому працюватиме сервер, та створює датаграмний сокет для мережевої взаємодії та порожнє «сховище» зареєстрованих користувачів userList. При створенні сокету слід якось обробити можливе виключення типу SocketException, яке викидається у разі виникнення помилок під час створення сокету чи доступу до нього. Конструктор сервера може мати такий вигляд:

```
public UPDServer(int serverPort) {
try {
socket = new DatagramSocket(serverPort);
} catch(SocketException e) {
```

```
System.out.println("Error: " + e);
}
userList = new ActiveUsers();
}
```

Далі можна створити єдиний відкритий метод нашого сервера, який організовуватиме

цикл роботи з клієнтами: «отримання запиту клієнта» – «обробка запиту клієнта» –

«формування та надсилання відповіді клієнту». Як аргумент цей метод отримуватиме ціле

число, що визначає розмір внутрішнього буфера для зберігання інформації UDP

повідомлення. Таким чином, цей параметр визначатиме розмір датаграми. При створенні

методу слід обробити можливі виняткові ситуації та, у будь-якому разі, після закінчення

роботи сервера закрити сокет. У нашому схематичному прикладі сервер працює «постійно» і

сокет закритий не буде. Самостійно реалізувати припинення роботи сервера за спеціальним

повідомленням клієнта. Метод може мати такий вигляд:

```
public void work(int bufferSize) {
try {
System.out.println("Server start...");
while (true) { // безкінечний цикл роботи з клієнтами
getUserData(bufferSize); // отримання запиту клієнта
log(address, port); // вивід інформації про клієнта на екран
sendUserData(); // формування та відправка відповіді
// клієнту
}
} catch(IOException e) {
System.out.println("Error: " + e);
} finally {
System.out.println("Server end...");
socket.close();
}
}
```

Функціональність сервера реалізується закритими методами. Спочатку реалізуємо

службові методи: метод ведення «лога» сервера (вивід інформації про підключення):

```
private void log(InetAddress address, int port) {
System.out.println("Request from: " + address.getHostAddress() +
" port: " + port);
}
```

Потім створимо метод, призначений для очищення байтового масиву — інформаційного буфера датаграми: private void clear(byte[] arr).

Тепер можна написати метод, призначений для отримання датаграм клієнтів. У цьому методі слід створити об'єкт класу DatagramPacket, призначений для отримання датаграм.

Він створюється на основі байтового масиву – буфера для зберігання отриманої від клієнтів інформації. Потім за допомогою методу сокета receive() переведемо сокет у режим очікування на підключення клієнта. Після підключення клієнта визначаємо адресу та порт, звідки здійснювалося підключення і, якщо цього клієнта в списку немає, додаємо його до списку-сховища. Далі очищаємо буфер для підготовки до наступного запиту. Метод має якось обробити можливі виняткові ситуації. Таким чином метод getUserData може мати такий вигляд:

```
private void getUserData(int bufferSize) throws IOException {
byte[] buffer = new byte[bufferSize];
packet = new DatagramPacket(buffer, buffer.length);
socket.receive(packet);
address = packet.getAddress();
port = packet.getPort();
User usr = new User(address, port);
if (userList.isEmpty()) {
userList.add(usr);
} else if (!userList.contains(usr)) {
userList.add(usr);
}
clear(buffer);
```

}

Слід створити останній метод: «упаковка» відповіді в датаграму та відправка її клієнту за тією адресою і на той самий порт, звідки надійшов запит. Тут слід зазначити два моменти: оскільки датаграма не може бути занадто великою, а список зареєстрованих клієнтів може бути дуже довгим, необхідно розбити його на частини: кожна частина — інформація про одного клієнта. Для кожної частини створимо окремий об'єкт-датаграму для надсилання цієї інформації клієнту. Ознакою кінця надсилання всього списку клієнтів буде датаграма з буфером нульової довжини.

Другий момент: для того, щоб перетворити об'єкт класу User – інформацію про окремого клієнта – у байтовий масив скористаємося стандартним механізмом Java – серіалізацією об'єктів. Серіалізацію виконаємо в потік ByteArrayOutputStream, який дуже легко перетворити на масив байт, як того вимагає конструктор датаграми (https://docs.oracle.com/javase/8/docs/api/java/io/ByteArrayOutputStream.html). Слід зазначити, що датаграма для надсилання повідомлення має включати адресу та порт призначення. Таким чином, метод відправлення датаграми користувачеві sendUserData() може мати такий вигляд:

```
private void sendUserData() throws IOException {
byte[] buffer;
for (int i = 0; i < userList.size(); i++) {
ByteArrayOutputStream bout = new ByteArrayOutputStream();
ObjectOutputStream out = new ObjectOutputStream(bout);
out.writeObject(userList.get(i));
buffer = bout.toByteArray();
packet = new DatagramPacket(buffer, buffer.length, address, port);
socket.send(packet);
```

```
}
buffer = "end".getBytes();
packet = new DatagramPacket(buffer, 0, address, port);
socket.send(packet);
}
```

І, нарешті, слід написати метод main(), який створить об'єкт-сервер і викличе метод
work для роботи сервера та очікування запитів клієнтів:

```
public static void main(String[] args) {
(new UPDServer(1501)).work(256);
}
```

У цьому випадку сервер підключено до порту 1501 на тому комп'ютері, де він буде
запущений і може приймати датаграми довжиною до 256 байт.

Тепер слід створити клас, який представляє клієнта UDP. Клієнт UDP є додатком,
який за своєю структурою нагадує сервер: він використовує протокол UDP для надсилання
запитів на сервер та отримання відповідей від серверної програми. У клієнтському UDP
додатку необхідно створити об'єкт класу DatagramSocket, який буде відправляти
повідомлення-запит на сервер і потім буде приймати повідомлення від сервера UDP. Для
цього необхідно виконати такі кроки:

- створити сокет (об'єкт класу DatagramSocket) для установки з'єднання з сервером;
- створити повідомлення для сервера — об'єкт класу DatagramPacket і застосувати
метод сокета send() для передачі цього повідомлення на сервер;
- створити об'єкт класу DatagramPacket для зберігання отриманого повідомлення і
застосувати метод сокета receive() для прийому датаграм, відправлених сервером.

Цей клас міститиме метод main(), у якому створюватиметься і запускатиметься UDP
клієнт.

За своєю структурою клас UDPClient буде аналогічним класу UPDServer. Він може
містити аналогічні закриті поля.

```
public class UDPClient {
private ActiveUsers userList = null;
private DatagramSocket socket = null;
private DatagramPacket packet = null;
private int serverPort = -1;
private InetAddress serverAddress = null;
}
```

Однак, конструктор класу трохи відрізнятиметься: при створенні об'єкта-клієнта слід
вказати адресу та порт сервера, з яким працюватиме клієнт. Крім того, зручно встановити
обмеження за часом, протягом якого клієнт чекатиме відповіді від сервера (метод
setSoTimeout). У випадку, якщо вказана межа часу буде перевищена, клієнт вважатиме, що
сервер недоступний і зможе якось обробити цю ситуацію. У цьому випадку буде викинуто
виключення SocketTimeoutException. Таким чином, конструктор клієнта може виглядати
так:

```
public UDPClient(String address, int port) {
userList = new ActiveUsers();
serverPort = port;
try {
serverAddress = InetAddress.getByName(address);
socket = new DatagramSocket();
socket.setSoTimeout(1000);
} catch (UnknownHostException e) {
System.out.println("Error: " + e);
} catch (SocketException e) {
System.out.println("Error: " + e);

}
}
```

Далі можна створити єдиний відкритий метод клієнта — метод work(). Цей метод

дуже схожий на однойменний метод сервера, тільки клієнт спочатку відправляє запит на

сервер

```
packet = new DatagramPacket(buffer, buffer.length,
serverAddress, serverPort);
socket.send(packet);
```

а потім отримує повідомлення від сервера з інформацією про зареєстрованих клієнтів,

заповнює свій список можливих клієнтів та завершує прийом повідомлень після отримання

датаграми з нульовою довжиною буфера. Після цього сокет клієнта закривається та список

зареєстрованих клієнтів виводиться на екран. Слід зазначити, що для відновлення об'єкта

використовується стандартний механізм десеріалізації Java. Для цього використовується

вхідний об'єктний потік, створений на основі ByteArrayInputStream (https://docs.oracle.com/javase/8/docs/api/java/io/ByteArrayInputStream.html). Можливий вигляд

методу work() наведений нижче:

```
public void work(int bufferSize) throws ClassNotFoundException {
byte[] buffer = new byte[bufferSize];
try {
packet = new DatagramPacket(buffer, buffer.length,
serverAddress, serverPort);
socket.send(packet);
System.out.println("Sending request");
while (true) {
packet = new DatagramPacket(buffer, buffer.length);
socket.receive(packet);
if (packet.getLength()==0) break;
ObjectInputStream in = new ObjectInputStream(
new ByteArrayInputStream(
packet.getData(), 0, packet.getLength()));
User usr = (User) in.readObject();
userList.add(usr);
clear(buffer);
}
```

```
} catch(SocketTimeoutException e) {
System.out.println("Server is unreachable: " + e);
} catch (IOException e) {
System.out.println("Error: " + e);
}
finally {
socket.close();
}
System.out.println("Registered users: " + userList.size());
System.out.println(userList);
}
```

Залишилось додати закритий метод очищення буфера private void clear(byte[]

arr) і метод main(), який створює і запускає клієнта:

```
public static void main(String[] args) throws ClassNotFoundException {
(new UDPClient("127.0.0.1", 1501)).work(256);
}
```

Слід зазначити, що в нашому випадку сервер та клієнт запускаються на одній і тій самій локальній машині як дві віртуальні Java-машини та взаємодіють між собою за допомогою мережевих інтерфейсів.