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

ЗВІТ
З ЛАБОРАТОРНОЇ РОБОТИ №6

дисципліна: «Крос-платформне програмування»

Виконав: студент групи КС23
Травченко Сергій Миколайович

Перевірив: доцент кафедри ШІтаПЗ
Споров Олександр Євгенович

Харків
2024

Завдання №2,3 Напишіть набір програм (UDP ехо-клієнт і UDP ехо-сервер), який можна

використовувати для тестування роботи мережі. Для того, щоб переконатися, що мережа

функціонує належним чином між двома комп'ютерами, можна запустити ехо-клієнт на

одному комп'ютері та підключитися до ехо-сервера на другому комп'ютері.

Для вирішення цього завдання можна в поточному проекті Java створити пакет для

зберігання класів, що створюються. Цей пакет можна назвати echoServer. У цьому пакеті слід

спочатку створити деякі допоміжні класи.

Так, наприклад, різні сервери, що працюють відповідно до правил UDP, мають дуже

схожу структуру. Вони всі чекають на прибуття датаграми на заздалегідь заданому порту і

відповідають датаграмою на кожну отриману датаграму від клієнтів. Їхня відмінність полягає

лише у вмісті датаграми, яку вони надсилають клієнтам. Створимо простий абстрактний клас

UDPServer, який реалізує вказану функціональність.

Цей клас обов'язково міститиме два поля: int bufferSize (задає максимальний розмір

датаграми, яку може прийняти сервер) і int port (номер порту, на якому очікується

надходження датаграми); дані поля не повинні змінюватися після створення сервера. Для

того, щоб створюваний сервер міг працювати в окремому потоці виконання, його клас

реалізовуватиме інтерфейс Runnable.

```
package echoServer;

public abstract class UDPServer implements Runnable {

private final int bufferSize;

private final int port;

@Override

public void run() {

// TODO Auto-generated method stub

}

}
```

Тепер до класу додамо конструктори, які надають задані значення вказаним

фінальним полям класу.

```
public UDPServer(int port, int bufferSize) {

this.bufferSize = bufferSize;

this.port = port;

}

public UDPServer(int port) {

this(port, 8192);

}

public UDPServer() {

this(12345, 8192);

}
```

Метод run(), вказаний у класі, повинен містити цикл, у якому багаторазово отримує

датаграму від клієнтів, а потім формує та надсилає відповідь, передаючи її абстрактному

методу response(). Цей метод визначає реакцію сервера на датаграму, отриману від клієнта,

і буде перевизначений у конкретних підкласах для реалізації різних типів серверів.

```
public void run() {

byte[] buffer = new byte[bufferSize];

try (DatagramSocket socket = new DatagramSocket(port)) {

.....

while (true) {

.....

DatagramPacket incoming = new

DatagramPacket(buffer, buffer.length);

try {

socket.receive(incoming);

this.respond(socket, incoming);

} catch (...) {}

}
```

Для зручності подальшої роботи нам потрібен спосіб зупинки сервера. Для цього

можна створити метод shutDown(), який встановлює прапорець isShutDown. В

основному циклі на кожній ітерації виконуватиметься перевірка значення цього прапорця для

визначення моменту, коли слід завершити цей цикл. Оскільки у випадку, коли взагалі немає

клієнтів, виклик методу receive() може бути заблокованим необмежено довго,

рекомендується встановити час очікування на сокеті. Це призведе до того, що у разі

відсутності клієнтів за заданий час, буде викинуто виключення SocketTimeoutException і

сервер розблокується і вийде з режиму очікування клієнтів.

Так, створимо поле класу для прапорця isShutDown.

```
private volatile boolean isShutDown = false;
```

Додамо метод зупинки сервера shutDown():

```
public void shutDown() {

this.isShutDown = true;

}
```

Додамо вказаний абстрактний метод respond():

```
public abstract void respond(DatagramSocket socket, DatagramPacket request)

throws IOException;
```

І завершимо реалізацію основного методу класу — методу run().

```
byte[] buffer = new byte[bufferSize];

try (DatagramSocket socket = new DatagramSocket(port)) {

socket.setSoTimeout(10000);

while (true) {

if (isShutDown)

return;

DatagramPacket incoming = new DatagramPacket(buffer,
```

```
buffer.length);

try {

socket.receive(incoming);

this.respond(socket, incoming);

} catch (SocketTimeoutException ex) {

if (isShutDown)

return;

} catch (IOException ex) {

System.err.println(ex.getMessage() + "\n" + ex);

}

} // end while

} catch (SocketException ex) {

System.err.println("Could not bind to port: " + port + "\n" + ex);

}
```

При реалізації підкласів класу UDPServer слід врахувати, що якщо для відповіді на

пакет від клієнта потрібна довготривала обробка, тоді метод response() може породжувати

новий потік виконання, щоб виконати це довге завдання. Однак, як правило, UDP сервери не

виконують довгої взаємодії з клієнтом. Кожен вхідний пакет – датаграма обробляється

незалежно від інших пакетів, тому відповідь клієнту зазвичай можна формувати у методі

response(), не створюючи новий потік виконання.

Тепер створимо клас exo-сервера UDPEchoServer, який виконуватиме серверну

частину поставленого завдання.

```
package echoServer;

import java.io.*;

import java.net.*;

public class UDPEchoServer extends UDPServer {

@Override

public void respond(DatagramSocket socket, DatagramPacket request)

throws IOException {

// TODO Auto-generated method stub

}

public static void main(String[] args) {

// TODO Auto-generated method stub

}

}
```

Оскільки, зазвичай, для роботи exo-сервера використовується порт 7

(https://en.wikipedia.org/wiki/Echo_Protocol), у класі можна вказати фінальне поле, яке

визначає порт з цим номером і конструктор, який передає його батьківському класу.

```
public final static int DEFAULT_PORT = 7;



public UDPEchoServer() {

super(DEFAULT_PORT);

}
```

Метод response() виконуватиме просте завдання: формувати датаграму — копію

датаграми клієнта та відправляти її туди, звідки надійшов запит.

```
DatagramPacket reply = new DatagramPacket(request.getData(),

request.getLength(), request.getAddress(), request.getPort());

socket.send(reply);
```

Тепер залишилось додати до методу main() команди створення і запуску сервера:

```
UDPServer server = new UDPEchoServer();

Thread t = new Thread(server);

t.start();
```

А також, за бажанням, можна додати команди для зупинки сервера по закінченню

заданого часу роботи (у прикладі 20 сек.):

```
System.out.println("Start echo-server...");

try {

Thread.sleep(20000);

} catch (InterruptedException e) {

e.printStackTrace();

}

server.shutDown();

System.out.println("Finish echo-server...");
```

Тепер перейдемо до створення класів ехо-клієнта. Ехо-клієнт, розроблений на базі

TCP сокетів, може встановити з'єднання з ехо-сервером, відправити повідомлення і

дочекатися відповіді на нього. На відміну від цього, ехо-клієнт, розроблений на базі UDP

сокетів, не може гарантувати, що надіслане повідомлення буде отримано сервером. Тому він

не може просто чекати на відповідь; клієнт повинен реалізувати асинхронну відправку та

отримання даних.

Таку поведінку можна реалізувати за допомогою потоків виконання. Один потік

виконання буде обробляти ввід користувача та відправляти його на ехо-сервер, а інший потік

виконання отримуватиме дані від сервера та відображатиме їх для користувача. Таким чином,

ехо-клієнт можна розділити на три класи: основний клас UDPEchoClient, клас

SenderThread і клас ReceiverThread.

Основний клас програми-клієнта дуже простий. На основі введеної інформації

створюється DatagramSocket, створюються та запускаються потоки для відправлення та

отримання інформації від сервера. Крім того, опрацьовуються відповідні виключення.

```
public class UDPEchoClient {

public final static int PORT = 7;

public static void main(String[] args) {

String hostname = "localhost";

if (args.length > 0) {

hostname = args[0];

}
```

```
try {



InetAddress ia = InetAddress.getByName(hostname);

DatagramSocket socket = new DatagramSocket();

Thread sender = new SenderThread(socket, ia, PORT);

sender.start();

Thread receiver = new ReceiverThread(socket);

receiver.start();

} catch (UnknownHostException ex) {

System.err.println(ex);

} catch (SocketException ex) {

System.err.println(ex);

}

}

}
```

Тут важливо відзначити, що для створення об'єктів класів SenderThread і

ReceiverThread використовувався один і той самий DatagramSocket.

Клас SenderThread повинен організувати читання даних (у нашому прикладі у

вигляді рядків) з консолі, сформувати з них датаграму та відправити її на ехо-сервер.

```
public class SenderThread extends Thread {

@Override

public void run() {
```

}

}

У нашому прикладі читання даних буде виконано з потоку System.in, але можна

зробити і інший клас клієнта, який читатиме дані з іншого потоку (наприклад, з файлового

потоку FileInputStream).

Цей клас може містити поля, що визначають адресу exo-сервера, з яким буде

взаємодіяти даний клієнт, номер порту на цьому сервері, а також об'єкт типу

DatagramSocket, який виконуватиме прийом та передачу датаграм.

private InetAddress server;

private int port;

private DatagramSocket socket;

Цей клас повинен включати конструктор, який приймає зазначені дані, і присвоює їх

відповідним полям класу. Крім того, рекомендується «встановити з'єднання» з сервером за

допомогою методу connect, щоб бути впевненим, що датаграми спрямовуються тільки на

потрібний сервер і приймаються також тільки з вказаного сервера. Звичайно, малоймовірно,

що якийсь інший сервер у мережі докучатиме своїми датаграмами нашому клієнту, але з

точки зору безпеки роботи це («встановити з'єднання») рекомендується зробити.

SenderThread(DatagramSocket socket, InetAddress address, int port) {

this.server = address;

this.port = port;

this.socket = socket;

this.socket.connect(server, port);

}

Спочатку в методі run() створюється потік читання BufferedReader userInput

для взаємодії з користувачем. Потім в основному циклі користувач може ввести рядок для

передачі на сервер. Рядок, що складається лише з точки, сигналізує про закінчення роботи.

Потім, введений рядок за допомогою методу getBytes() перетворюється на масив байтів, на

основі якого створюється датаграма і відправляється на сервер. І після цього цей потік

виконання дає шанс іншому потоку на виконання.

try {

BufferedReader userInput = new BufferedReader(

new InputStreamReader(System.in));

while (true) {

.....

String theLine = userInput.readLine();

if (theLine.equals("."))

break;

byte[] data = theLine.getBytes("UTF-8");

DatagramPacket output = new DatagramPacket(data, data.length, server,

```
port);

socket.send(output);

Thread.yield();

}

} catch (IOException ex) {

System.err.println(ex);

}
```

Для керування цим клієнтом можна додати ще одне поле

```
private volatile boolean stopped = false;
```

перевірку значення цього поля відразу після заголовка основного циклу методу run():

```
if (stopped)

return;
```

та метод встановлення значення цього поля для виходу з циклу:

```
public void halt() {

this.stopped = true;

}
```

Клас ReceiverThread може мати аналогічну структуру, тільки він не отримує дані

від користувача, а приймає датаграму, отримує з неї передану інформацію та виводить її на

екран:

```
class ReceiverThread extends Thread {

private DatagramSocket socket;

private volatile boolean stopped = false;

ReceiverThread(DatagramSocket socket) {
```

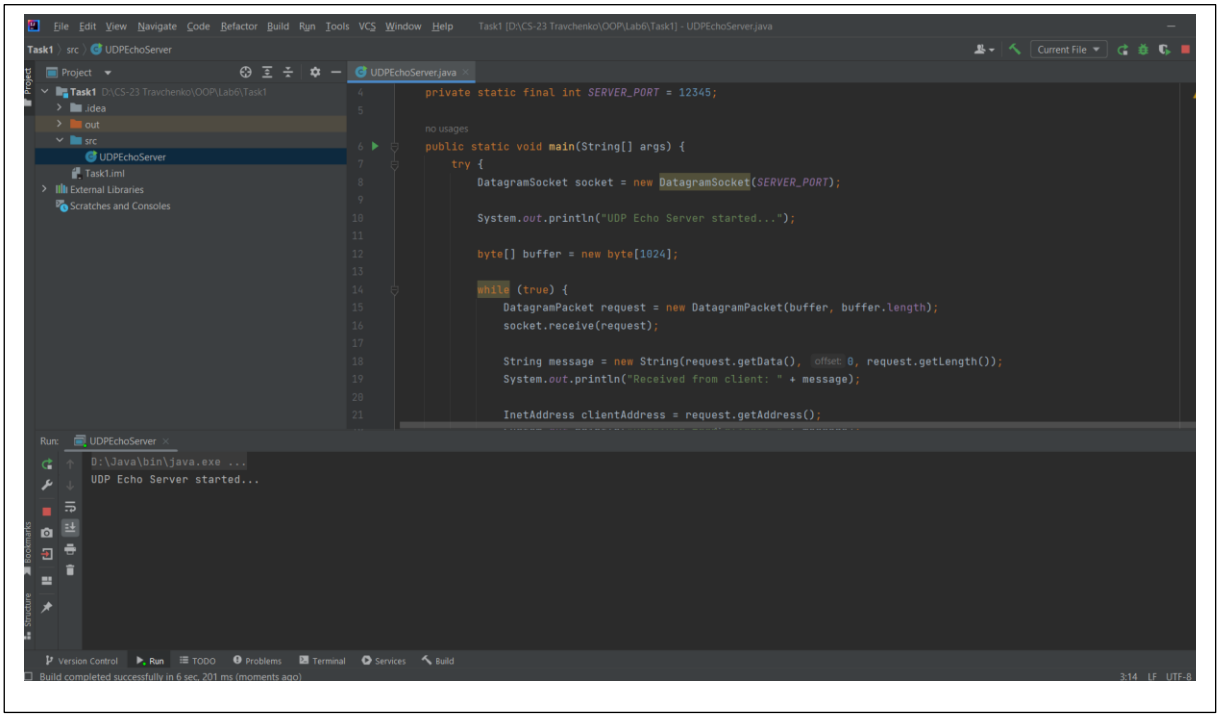
```
this.socket = socket;
}
@Override
public void run() {
byte[] buffer = new byte[65507];
while (true) {
if (stopped)
return;


DatagramPacket dp = new DatagramPacket(buffer, buffer.length);
try {
socket.receive(dp);
String s = new String(dp.getData(), 0, dp.getLength(), "UTF-8");
System.out.println(s);
Thread.yield();
} catch (IOException ex) {
System.err.println(ex);
}
}
}
public void halt() {
this.stopped = true;
}
```

}Лістинги на гітхабі.

Результати виконання завдання №2 наведено:

1. На малюнку 1.1 – результат виконання програми.



Малюнок 1.1 – результат виконання програми

Завдання 1 Текстові конференції (text mode conferencing) та електронні дошки оголошень (bulletin
board system, BBS) є найстарішими способами мережного спілкування. Відповідно до цих
моделей спілкування користувачі обмінюються текстовими повідомленнями через термінал.
На основі Multicast сокетів розробіть простий додаток, який підтримує текстову
конференцію. Додаток виконує два завдання:

- відправлення текстових повідомлень якогось одного учасника конференції всім іншим учасникам;
- отримання всіма учасниками конференції таких текстових повідомлень.

Розробіть як консольну версію програми, так і версію із графічним інтерфейсом
користувача.

```
// Сервер
import ...

no usages
public class TextConferenceServer {
    2 usages
    private static final int PORT = 12345;
    1 usage
    private static final String GROUP_ADDRESS = "230.0.0.0";

    no usages
    public static void main(String[] args) {
        try {
            InetAddress group = InetAddress.getByName(GROUP_ADDRESS);
            MulticastSocket socket = new MulticastSocket(PORT);
            socket.joinGroup(group);

            Thread receiverThread = new Thread(() -> {
                try {
```

Малюнок 2.1 – результат виконання програми

Власне та частина програми, що буде відповідати за проведення текстової конференції, виконуватиметься в окремому потоці виконання. На наступному етапі можна
визначити інтерфейси: які операції графічний інтерфейс користувача вимагає від фонового
потоку виконання та ті операції, що фоновий потік вимагає від графічного інтерфейсу
користувача.
Створимо інтерфейс Messanger, що визначає основні операції, які потрібні графічному інтерфейсу для керування текстовою конференцією, що виконується в окремому
потоці:

```
public interface Messanger {
void start();
void stop();
void send();
}
```

Цих операцій небагато: запуск фонового потоку, зупинення його та відправлення

повідомлення за призначенням. Наступний інтерфейс UITasks визначає ті операції, які
графічний інтерфейс користувача повинен надати фоновому потоку: одержати з текстового
поля текст повідомлення для надсилання та занесення в текстову область повідомлення,
отриманого через мережу.

```
public interface UITasks {
String getMessage();
void setText(String txt);
}
```

Далі перейдемо реалізації заданих інтерфейсів. Інтерфейс UITasks найпростіше
реалізовувати як внутрішній клас основного вікна – таким чином буде отримано повний
доступ до всіх потрібних полів основного вікна програми. Але, для коректної роботи
програмного рішення, оскільки методи цього інтерфейсу можуть викликатися як з Event
Dispatching Thread, так і ззовні нього, необхідно подбати про те, щоб робота з компонентами
інтерфейсу користувача йшла тільки в Event Dispatching Thread (в Java цей потік ще має
назву AWT-EventQueue). Зробити це необхідно для кожного методу, що вказаний в інтерфейсі.
Для покращення структури програми можна створити динамічний проксі. З його допомогою
можна динамічно, в процесі роботи програми створити об'єкт, який буде реалізовувати

потрібний інтерфейс, і всі виклики методів цього інтерфейсу будуть перенаправлені
створеному нами обробнику. А цей обробник подбав про те, щоб кожен метод викликався з
Event Dispatching Thread. Таким чином, код програми буде менш громіздким і буде простіше,
при необхідності, модифікувати нашу програму.
Створимо клас обробника EDTInvocationHandler, що буде відповідати за організацію виклику методу графічного інтерфейсу користувача в потоці обробки подій.

```
public class EDTInvocationHandler implements InvocationHandler {
private Object invocationResult = null;
private UITasks ui;
public EDTInvocationHandler(UITasks ui) {
```

```
this.ui = ui;
}
@Override
public Object invoke(Object proxy, final Method method, final Object[] args)
throws Throwable {
if (SwingUtilities.isEventDispatchThread()) {
invocationResult = method.invoke(ui, args);
} else {
Runnable shell = new Runnable() {
@Override
public void run() {
try {
invocationResult = method.invoke(ui, args);
} catch (Exception ex) {
throw new RuntimeException(ex);
}
}
};
SwingUtilities.invokeAndWait(shell);
}
return invocationResult;
}
}
```

В нашому обробнику ми використали вбудований клас javax.swing.SwingUtilities.

Крім усього іншого, він містить два корисні для нас статичні методи – invokeAndWait(Runnable) та invokeLater(Runnable). Обидва ці методи виконують метод run() переданого їм об'єкту Runnable в GUI-потоці. Відмінність між методами у тому, перший метод синхронний – тобто. викликаючий його потік чекає на завершення виконання методу run(), а другий асинхронний – тобто цей метод run() виконується гарантовано, але тоді, коли це буде зручно віртуальній машині.

Крім цих двох методів клас SwingUtilities містить корисний для нашої програми метод – isEventDispatchThread, – що дозволяє визначити, у якому потоці виконується поточний код – в GUI чи ні. Таким чином, якщо це GUI-потік, то у використанні методів invokeAndWait(Runnable) та invokeLater(Runnable) немає необхідності. Ще одне зауваження: існує клас java.awt.EventQueue, що містить ті ж три методи (з тією

різницею, що метод isEventDispatchThread в цьому класі має називу isDispatchThread). Справа в тому, що методи класу SwingUtilities просто є оболонками
для цих методів і можна використовувати будь-які з цих.
Тепер можна створити клас UITasksImpl, що реалізує інтерфейс UITasks, як

внутрішній клас у класі основного вікна програмного рішення.

```
private class UITasksImpl implements UITasks {
@Override
public String getMessage() {
String res = textFieldMsg.getText();
textFieldMsg.setText("");
return res;
}
@Override
public void setText(String txt) {
textArea.append(txt + "\n");
}
}
```

Залишилося лише скористатись створеним обробником, створивши об'єкт динамічного проксі. Це можна зробити в обробнику натискання кнопки З'єднати.

```
UITasks ui = (UITasks) Proxy.newProxyInstance(getClass().getClassLoader(),
new Class[]{UITasks.class},
new EDTInvocationHandler(new UITasksImpl()));
```

Тепер можна перейти до реалізації фонового потоку. Визначимо клас MessanderImpl, що реалізує інтерфейс Messander. Слід зазначити, що крім інформації,
необхідної організації зв'язку (адреси групи багатоадресного спілкування, номери порту,
імені користувача) цей клас має містити полі типу UITasks, якому буде переданий
динамічний проксі-об'єкт для взаємодії з графічним інтерфейсом користувача.
Структура класу дуже схожа на відповідні класи з попередніх завдань: у конструкторі
створюється об'єкт для організації багатоадресного спілкування (типу MulticastSocket),
реалізуються методи, що визначені в інтерфейсі Messander, для чого створюються два
внутрішні класи Receiver (для організації постійного очікування повідомлень від

співрозмовників) та Sender (для відправлення багатоадресного повідомлення по мережі).

```
public class MessanderImpl implements Messanger {
private UITasks ui = null;
private MulticastSocket group = null;
private InetAddress addr = null;
private int port;
private String name;
private boolean canceled = false;
public MessanderImpl(InetAddress addr, int port, String name, UITasks ui) {
this.name = name;
this.ui = ui;
this.addr = addr;
this.port = port;
try {
group = new MulticastSocket(port);
group.setTimeToLive(2);
group.joinGroup(addr);
} catch (Exception e) {
e.printStackTrace();
}
}
```

```
@Override
public void start() {
Thread t = new Receiver();
t.start();
}
@Override
public void stop() {
cancel();
try {
group.leaveGroup(addr);
} catch (IOException e) {
JOptionPane.showMessageDialog(null, "Помилка від'єднання\n" +
e.getMessage());
} finally {
group.close();
}
}
@Override
public void send() {
new Sender().start();
}
```

```
private class Sender extends Thread {
public void run() {
try {
String msg = name + ": " + ui.getMessage();
byte[] out = msg.getBytes();
DatagramPacket pkt = new DatagramPacket(out, out.length, addr,
port);
group.send(pkt);
} catch (Exception e) {
JOptionPane.showMessageDialog(null, "Помилка відправлення\n" +
e.getMessage());
}
}
}
private class Receiver extends Thread {
public void run() {
try {
byte[] in = new byte[512];
DatagramPacket pkt = new DatagramPacket(in, in.length);
while (!isCanceled()) {
group.receive(pkt);

ui.setText(new String(pkt.getData(), 0, pkt.getLength()));

}
} catch (Exception e) {
if (isCanceled()) {
JOptionPane.showMessageDialog(null, "З'єднання завершено");
} else {
JOptionPane.showMessageDialog(null, "Помилка прийому\n" +
e.getMessage());
}
}
}
}
private synchronized boolean isCanceled() {

return canceled;
}
public synchronized void cancel() {
canceled = true;
}
}
```

Об'єкт Messanger зручно зробити закритим полем класу – основного вікна програми

private Messanger messanger = null;
та створити у обробнику натискання кнопки З'єднати об'єкт типу MessanderImpl за
інформацією, вказаною користувачем у відповідних текстових полях.
messanger = new MessanderImpl(addr, port, name, ui);
Тут ui — створений трохи раніше об'єкт динамічного проксі для коректної взаємодії
мережевої частини програми з графічним інтерфейсом користувача.
Далі необхідно закінчити нашу програму — вказати команди, що повинні бути
виконані при натисканні на кожну кнопку. Вони природні та зрозумілі.
Запустимо наше програмне рішення на виконання. Слід зазначити, що в нашому
випадку всі учасники текстової конференції запускаються на одній і тій же локальній машині
як різні віртуальні Java-машини та взаємодіють між собою за допомогою мережевих
інтерфейсів. Приклад можливого поточного стану текстової конференції з графічним
інтерфейсом користувача з чотирма учасниками наведений на рис. нижче.