Programowanie współbieżne i rozproszone

mgr inż. Marcin Mrukowicz

Instrukcja laboratoryjna IX

Programowanie z udziałem gniazd (ang. sockets)

W języku programowania Java gniazda w protokole UDP są reprezentowane przez klasę *DatagramSocket*. W przeciwieństwie do gniazd w protokole TCP nie ma tutaj wyraźnego podziału na role klienta i serwera (choć te role mogą występować). Oznacza to, że istnieje możliwość uruchomienia współpracujących procesów w dowolnej kolejności; może się jednak zdarzyć, że programy zadziałają poprawnie tylko w przypadku, gdy któryś z procesów zostanie uruchomiony jako pierwszy. Protokół UDP nie gwarantuje, że dane zostaną dostarczone; istnieje możliwość, że dany pakiet (datagram) nigdy nie dotrze do hosta docelowego. Jest to spowodowane tym, że w przeciwieństwie do protokołu TCP, protokół UDP nie ma mechanizmu kontroli przepływu danych, nie występuje w nim również pojęcie połączenia (jest to protokół bezpołączeniowy). Oprócz oczywistych wad (niepewność transmisji), protokół ten posiada jedną kluczową zaletę: szybkość działania. Powoduje to, że jest on stosowany w sytuacjach, w których zależy nam na szybkim przesłaniu danych i jednocześnie dopuszczamy że czasami protokół ten zawiedzie (nasze dane nie zostaną dostarczone). Najczęściej protokół ten stosuje się do strumieniowania danych multimedialnych (np. odtwarzanie muzyki albo filmów przez sieć), gdzie zgubienie kilku pakietów spowoduje co najwyżej, że transmisja zostanie przerwana na chwilę, być może nawet niezauważalną dla odbiorcy (rzędu kilkudziesięciu milisekund), natomiast sam proces strumieniowania będzie zachodził dużo szybciej niż w przypadku zastosowania protokołu TCP. Innym zastosowaniem protokołu UDP mogą być gry sieciowe, rozgrywane w czasie rzeczywistym, gdzie zgubienie kilku pakietów może spowodować lekkie opóźnienie reakcji gracza, jednak całościowo rozgrywka będzie zapewne bardziej płynna, niż w przypadku stosowania protokołu TCP, gdzie błędy transmisji wymuszałyby przerwy w rozgrywce.

1. Przepisz, następującą klasę w języku Java:

```
package sockets.helloworld;

import java.io.BufferedReader;
import java.io.DataOutputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.Socket;
import java.net.SocketException;

public class ClientKeyboard {
    public static void main(String[] args) throws IOException {
        BufferedReader keyboardBufferedReader = new BufferedReader(new InputStreamReader(System.in));

        Socket socket = new Socket("127.0.0.1", 6000);
        DataOutputStream output = new DataOutputStream(socket.getOutputStream());
        BufferedReader socketBufferedReader = new BufferedReader(new
InputStreamReader(socket.getInputStream()));

        String response = socketBufferedReader.readLine();
        System.out.println("Sever responsed: " + response);
        while (true) {
            try {
                String command = keyboardBufferedReader.readLine();
                switch (command) {
                    case "end": {
                        socket.close();
```

```
                        System.out.println("stopped!");
                        return;
                    }
                    case "all files": {
                        output.writeBytes("all files" + System.lineSeparator());
                        output.flush();

                        response = socketBufferedReader.readLine();
                        System.out.println("Sever responsed: " + response);
                    }
                }
            }
            catch (SocketException ex) {
                System.out.println("Server stopped connection!");
                break;
            }
            catch (IOException e) {
                System.out.println("Input output exception!");
                break;
            }

        }
    }
}
```

klasa *ClientKeyboard.java*

Klasa ta jest klientem dla klasy *ServerLoop.java* z poprzedniej instrukcji laboratoryjnej, z tą różnicą, że tutaj wczytujemy polecenia z klawiatury. Implementacja tej klasy nie jest zupełnie skończona: na razie możliwe jest jedynie zakończenie połączenie z serwerem oraz wyświetlenie informacji o plikach txt.

- **Dokończ implementację tej klasy, tak, aby można było użyć jeszcze poleceń: *new file* i *get file*, które niech działają tak samo, jak w poprzedniej instrukcji laboratoryjnej.**

2. Przepisz następujące klasy w języku Java:

```
package sockets.helloworld;

import java.io.BufferedReader;
import java.io.DataOutputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.ArrayList;

public class WorldThreadServer {

    static class ServerThread extends Thread {
        Socket connection;

        public ServerThread(Socket connection) {
            this.connection = connection;
        }

        @Override
        public void run() {
            BufferedReader br = null;
            try {
                br = new BufferedReader(new InputStreamReader(connection.getInputStream()));
                DataOutputStream dataOutputStream = new DataOutputStream(connection.getOutputStream());
                int i = 0;
                while (i < 10000) {
                    String hello = br.readLine();
                    System.out.println("Client sends: " + hello);

                    dataOutputStream.writeBytes(" world!" + System.lineSeparator());
                    i++;
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
```

```
    public static void main(String [] args) throws IOException {
        ServerSocket serverSocket = new ServerSocket(6000);

        System.out.println("Started server at: " + serverSocket.getLocalSocketAddress());
        System.out.println(serverSocket.getInetAddress());
        int noClients = 0;
        ArrayList<Socket> connections = new ArrayList<>();
        while (noClients < 10) {
            Socket connection = serverSocket.accept();
            connections.add(connection);
            new ServerThread(connection).start();
            noClients++;
        }
        for (Socket s: connections) {
            s.getOutputStream().close();
            s.getInputStream().close();
            s.close();
        }
        serverSocket.close();
    }
}
```

Klasa *WorldThreadServer.java*

```
package sockets.helloworld;

import java.io.BufferedReader;
import java.io.DataOutputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.Socket;

public class HelloClient {
    public static void main(String[] args) throws IOException, InterruptedException {

        Socket socket = new Socket("127.0.0.1", 6000);
        DataOutputStream output = new DataOutputStream(socket.getOutputStream());
        BufferedReader socketBufferedReader = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
        for (int i = 0; i < 10000; i++) {
            output.writeBytes("Hello, " + System.lineSeparator());
            output.flush();

            String response = socketBufferedReader.readLine();
            System.out.println("Sever responsed: " + response);
        }
        output.close();
        socketBufferedReader.close();
        socket.close();
    }
}
```

klasa *HelloClient.java*

Klasa serwera różni się zasadniczo od dotychczas stosowanych, gdyż potrafi ona obsługiwać więcej niż jednego klienta jednocześnie. Jest to osiągane za pomocą stosowania wątków: połączenie z każdym klientem jest delegowane do osobnego wątku, któremu przekazujemy w konstruktorze obiekt gniazda, tak, aby mógł on z nim pracować. Warto zauważyć, że istnieje tutaj założenie o tym, że każdy taki wątek będzie działać dokładnie tak samo.

- **Uruchom serwer, po czym uruchom równocześnie co najmniej 5 klientów (do maksymalnie 10). Obserwuj pracę każdego z klientów; czy występuje sytuacja, że klient oczekuje na obsłużenie przez serwer?**
- **Zmodyfikuj klasę *ServerLoop.java* z poprzedniego laboratorium, tak, aby działała ona wielowątkowo. Za pomocą klasy *ClientKeyboard.java* przetestuj działanie wielowątkowej klasy *ServerLoop.java:* utwórz dwie instancje klasy *ClientKeyboard.java,* wywołaj w obydwu polecenie *all files*, po czym za pomocą jednego klienta dodaj nowy plik, a następnie sprawdź w drugim kliencie, czy widać ten plik (ponownie wywołaj *all files*), po czym wywołaj polecenie *get file* na tym pliku.**

3. Przepisz następujące klasy w języku Java:

```
package sockets.advanced;
```

```java
import sun.misc.Signal;

import java.io.*;
import java.net.*;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.atomic.AtomicBoolean;

public class UDPSender {
    File f = new File("C:\\PinkPanther30.wav");
    BufferedReader fileReader;
    FileInputStream fs;
    int BUFFER = 64000;
    byte[] buf = new byte[BUFFER];
    DatagramSocket ds;
    InetAddress ip;
    static List<String> connected;

    public UDPSender() throws FileNotFoundException, SocketException, UnknownHostException {
        fileReader = new BufferedReader(new FileReader(f));
        ds = new DatagramSocket(3001);
        ip = InetAddress.getByName("127.0.0.1");
        connected = new ArrayList<>();
    }

    static class UDPServerThread extends Thread {
        FileInputStream fs;
        DatagramSocket ds;
        InetAddress ip;
        int port;
        int BUFFER = 64000;
        byte[] buf = new byte[BUFFER];

        public UDPServerThread(File f, DatagramSocket ds, InetAddress ip, int port) throws
FileNotFoundException {
            this.fs = new FileInputStream(f);;
            this.ds = ds;
            this.ip = ip;
            this.port = port;
        }

        public void sendPart() throws IOException {
            fs.read(buf, 0, BUFFER);
            ds.send(new DatagramPacket(buf, BUFFER, ip, port));
        }

        public void waitForResponse() throws IOException {
            byte [] received = new byte[1];
            ds.receive(new DatagramPacket(received, 1));
        }

        @Override
        public void run() {
            int sendedParts = 0;
            try {
                int parts = fs.available() / BUFFER;
                System.out.println(parts);
                this.sendPart();
                this.sendPart();
                sendedParts += 2;
                System.out.println("I sended 2 parts");

                while (sendedParts < parts) {
                    System.out.println("i wait for response");
                    waitForResponse();
                    sendPart();
                    sendedParts++;
                    System.out.println(sendedParts);
                }
                System.out.println("I ended work!");
                connected.remove(ip + String.valueOf(port));
                fs.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
```

```
    }

    public static void main(String [] args) throws IOException, InterruptedException {
        UDPSender udpSender = new UDPSender();
        AtomicBoolean serverStopped = new AtomicBoolean(false);

        Signal.handle(new Signal("INT"),
                signal ->
                {
                    System.out.println("Interrupted by Ctrl+C");
                    serverStopped.set(true);
                    try {
                        udpSender.ds.close();
                        udpSender.fileReader.close();
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                    System.out.println("stopped!");
                });

        while (!serverStopped.get()) {
            byte [] start = new byte[2];
            DatagramPacket dp = new DatagramPacket(start, 2);
            udpSender.ds.receive(dp);
            if (start[0] == 100 && !connected.contains(dp.getAddress() + String.valueOf(dp.getPort()))) {
                System.out.println("I get connection from new client: " + dp.getAddress() + ':'
+dp.getPort());
                new UDPSender.UDPServerThread(udpSender.f, udpSender.ds, dp.getAddress(),
dp.getPort()).start();
                connected.add(dp.getAddress() + String.valueOf(dp.getPort()));
            } else {
                udpSender.ds.send(new DatagramPacket(new byte[]{1}, 1, InetAddress.getByName("127.0.0.1"),
3001));
                System.out.println("I get an invalid connection from client: " + dp.getAddress() + ':' +
dp.getPort());
            }
        }
    }
}
```

klasa *UDPSender.java*

```
package sockets.advanced;

import javax.sound.sampled.*;
import java.io.*;
import java.net.*;

public class UDPReceiver {
    DatagramSocket ds;
    byte[] buf;
    int BUFFER = 64000;
    SourceDataLine sourceLine;
    ByteArrayInputStream byteArrayInputStream;

    public UDPReceiver(int port) throws SocketException, LineUnavailableException {
        ds = new DatagramSocket(port);
        buf = new byte[BUFFER];

        AudioFormat format = new AudioFormat(AudioFormat.Encoding.PCM_SIGNED, 22050.0f, 16, 1, 2,22050.5f,
false);
        DataLine.Info info = new DataLine.Info(SourceDataLine.class, format);
        sourceLine = (SourceDataLine) AudioSystem.getLine(info);
        System.out.println("Started client " + port);
    }

    public void sendReceived() throws IOException {
        ds.send(new DatagramPacket(new byte[]{1}, 1, InetAddress.getByName("127.0.0.1"), 3001));
    }

    public void getPart() throws IOException {
        ds.receive(new DatagramPacket(buf, BUFFER));
    }

    public void playPart() {
        sourceLine.write(buf, 0, 64000);
```

```
    }

    public void startStream() throws IOException {
        ds.send(new DatagramPacket(new byte[]{100, 100}, 2, InetAddress.getByName("127.0.0.1"), 3001));
    }

    public void getFirstParts() throws IOException {
        byte [] buf2 = new byte[BUFFER];
        ds.receive(new DatagramPacket(buf2, BUFFER));
        byte [] buf3 = new byte[BUFFER];
        ds.receive(new DatagramPacket(buf3, BUFFER));
        byte [] buf4 = new byte[128000];
        System.arraycopy(buf2, 0, buf4, 0, buf2.length);
        System.arraycopy(buf3, 0, buf4, buf2.length, buf3.length);
        byteArrayInputStream = new ByteArrayInputStream(buf4);
        sendReceived();
    }

    public void play() throws IOException {
        AudioFormat format = new AudioFormat(AudioFormat.Encoding.PCM_SIGNED, 22050.0f, 16, 1, 2,22050.5f,
false);
        AudioInputStream audioStream = new AudioInputStream(byteArrayInputStream, format, 661500);
        DataLine.Info info = new DataLine.Info(SourceDataLine.class, format);
        final int BUFFER_SIZE = 128000;
        AudioFormat audioFormat = audioStream.getFormat();
        int nBytesWritten = sourceLine.write(byteArrayInputStream.readAllBytes(), 0, 128000);
        try {
            sourceLine.open(audioFormat);
            sourceLine.start();

            for (int i = 3; i < 20; i++) {
                System.out.println("i want part no " + i);
                getPart();
                System.out.println("received " + i);
                sendReceived();
                playPart();
            }
            sourceLine.drain();
            sourceLine.close();
            System.out.println("I quit");
        } catch (LineUnavailableException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) throws Exception {
        UDPReceiver udpReceiver = new UDPReceiver(3003);
        udpReceiver.startStream();
        udpReceiver.getFirstParts();
        udpReceiver.play();

        UDPReceiver udpReceiver2 = new UDPReceiver(3002);
        udpReceiver2.startStream();
        udpReceiver2.getFirstParts();
        udpReceiver2.play();

        udpReceiver.ds.close();
        udpReceiver2.ds.close();
    }
}
```

klasa *UPDReceiver.java*

```
package sockets.advanced;

import javax.sound.sampled.LineUnavailableException;
import java.io.IOException;

public class UDPReceiver2 {
    public static void main(String [] args) throws IOException, LineUnavailableException {
        UDPReceiver udpReceiver2 = new UDPReceiver(3002);
        udpReceiver2.startStream();
        udpReceiver2.getFirstParts();
        System.out.println("2 gets first parts");
        udpReceiver2.play();
    }
```

```
}
```

klasa *UDPReceiver.java*

Powyższy przykład pokazuje wykorzystanie gniazd UPD do strumieniowania pliku audio. Dla uproszczenia wykorzystano nieskompresowany plik .wav i wbudowany w Javę mechanizm odtwarzania dźwięków, niewymagający stosowania żadnych bibliotek. Przykładowy plik pobrano ze strony: [https://www2.cs.uic.edu/~i101/SoundFiles/].

Można zauważyć również, że komunikacja z gniazdami UDP nie opiera się o strumienie, ale o przesyłanie wprost pakietów, które w uproszczeniu są po prostu tablicami, zawierającymi dane. W języku Java dane muszą być typu byte []. Zatem obiekty i inne typy, będą musiały być zamienione na tablicę bajtów, aby można było je przesłać za pomocą UDP.

Jak już wspomniano UDP jest protokołem bezpołączeniowym i nie ma tutaj podziału na rolę serwera i klienta. Zatem klasa *UDPSender.java* tylko ze względu na specyficzne napisanie programu, może być postrzegana jako „serwer", jednakże warto zwrócić uwagę na kilka szczegółów. W przeciwieństwie do TCP, istnieje tutaj tylko jedno aktywne połączenie, zatem nie ma tutaj wbudowanego mechanizmu, który odróżniałby kto wysyła dane do serwera. W istocie to każde inne gniazdo UDP może w dowolnej chwili przesłać dane do naszego „serwera" UDP. Jeżeli programista sam nie rozwiąże problemu potencjalnych konfliktów pomiędzy gniazdami, to mogą one sobie wzajemnie przeszkadzać w komunikacji.

W powyższym programie założono, że gniazdo „serwera" nasłuchuje na nowe połączenia jeżeli dostanie tablice dwóch bajtów, gdzie pierwszy bajt ma wartość równą 100. Klasa posiada również listę aktualnie połączonych „klientów". Jeżeli inne gniazdo wyśle dwubajtowy komunikat, zgodny z oczekiwaniem i dodatkowo nie jest zarejestrowane jako aktywny „klient", to zostanie ono dodane do listy „klientów", oraz zostanie uruchomiony dedykowany dla tego „klienta" wątek, który będzie strumieniował plik audio dla tego „klienta". Co ciekawe, jeżeli dowolne gniazdo wyśle jednobajtowy komunikat, to również zostanie on potraktowany jako dwubajtowy; w tej sytuacji następuje ponowne wysłanie tego komunikatu (gniazdo przesyła go samo sobie!), tak aby mógł być on odczytany przez odpowiedni wątek.

Zasadniczo „klient" najpierw wysyła rzeczywiście dwubajtowy komunikat do gniazda „serwera", po czym oczekuje na otrzymanie dwóch pakietów z fragmentami pliku audio. Po pomyślnym otrzymaniu pakietów, „klient" wysyła jednobajtowe potwierdzenie, że otrzymał pakiety (jest to ogólna zasada w naszym własnym protokole). „Serwer" z kolei oczekuje na to potwierdzenie, zanim odeśle kolejny fragment pliku audio. Wątek „serwera" oblicza na ile części będzie dzielił się dany plik, po czym pracuje, aż prześle do danego „klienta" cały plik. Z kolei klient również pracuje, aż otrzyma wszystkie części (które odtwarza na bieżąco). Wątek „serwera" poa zakończeniu wysyłania pliku, usuwa „klienta" z listy aktywnych „klientów", co pozwoli mu w przyszłości ponownie odpytać nasz serwer. **Warto zwrócić uwagę na fakt, że „klient", nie pobiera najpierw pliku, po czym odtwarza go w całości, ale raczej po otrzymaniu pierwszych dwóch pakietów, rozpoczyna już odtwarzanie pliku audio; jednocześnie „klient" na bieżąco otrzymuje w kolejnych pakietach kolejne fragmenty pliku audio. Jest to zatem prawdziwe strumieniowanie pliku audio. Na marginesie warto dodać, że rozmiar bufora (64 000 bajtów), przesyłającego fragment pliku ustalono zupełnie arbitralnie. Pakiet mógłby być mniejszy lub jeszcze odrobinę większy (w wersji protokołu IPV4 istnieje limit wielkości pakietu 65 527 bajtów).**

- **Uruchom klasę UDPSender, po czym od razu uruchom klasy UDPReceiver i UDPReceiver2. Co obserwujesz?**
- **Wykonaj następujące modyfikacje: usuń w klasie UDPSender metodę *waitForResponse* i jej wywołania. Usuń też instrukcję else w pętli głównej tej klasy. Równocześnie usuń w klasie UDPReceiver metodę *sendReceived* i wszystkie jej wywołania. Ponownie uruchom klasy jak w punkcie 1. Czy wszystko działa tak, jak poprzednio?**

- **Powróć do pierwotnej wersji programu. Zauważ, że klasa *UDPReceiver* posiada na sztywno podane, że ma otrzymać 20 pakietów. Spróbuj tak zmodyfikować program, aby po uzyskaniu połączenia z *UDPSender*, najpierw otrzymać liczbę pakietów, jakie mają zostać wysłane. Przetestuj działanie programu.**
- **Na koniec uruchom serwer na klastrze ICMK a klienta na własnym komputerze. Wykonaj konieczne zmiany, aby program działał i odtwarzał muzykę.**