Programowanie współbieżne i rozproszone

mgr inż. Marcin Mrukowicz

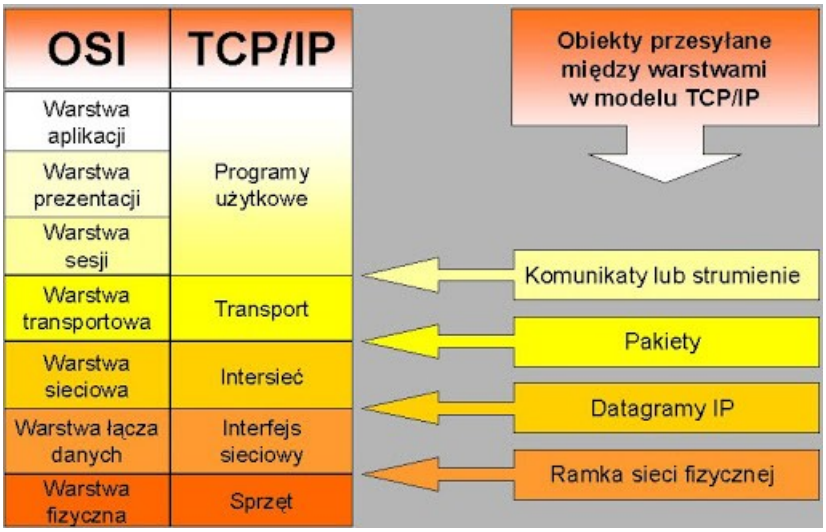
Instrukcja laboratoryjna VIII

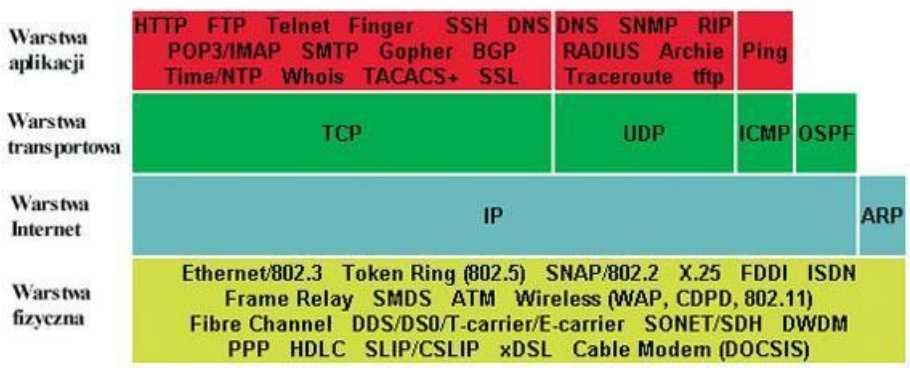
Programowanie z udziałem gniazd (ang. sockets)

Gniazdo (ang. socket) jest pojęciem dość abstrakcyjnym i w największej ogólności oznacza punkt końcowy dwukierunkowej komunikacji pomiędzy procesami (które może odbywać się w obrębie jednej maszyny lub pomiędzy odrębnymi komputerami).

Gniazda kojarzą się obecnie z gniazdami sieciowymi (ang. network socket), gdzie również oznaczają punkt końcowy dwukierunkowej komunikacji między procesami, ale przy założeniu, że komunikacja ta będzie przebiegać poprzez sieć komputerową. Oznacza to też, że mamy założenie, że procesy mogą pracować na różnych komputerach (oczywiście możliwe jest też, aby dwa procesy na jednym komputerze również komunikowały się przez sieć, ale jest to wykorzystywane raczej do specyficznych celów np. testowania wytworzonego oprogramowania niż docelowego sposobu działania). **Zatem korzystanie z gniazd sieciowych będzie wykorzystywane w programowaniu rozproszonym.**

W systemie operacyjnym UNIX i jego sukcesorach występują gniazda zarówno do komunikacji międzyprocesowej (IPC), jak i gniazda sieciowe, które posiadają bardzo podobny interfejs programistyczny. W tym systemie gniazdo jest interpretowane jako rodzaj pliku, więc możliwe jest pisanie i odczytywanie z niego jako strumienia, odpowiednio wyjściowego i wejściowego. Gniazdo do komunikacji międzyprocesowej jest obsługiwane przez jądro systemu, natomiast gniazdo sieciowe wymaga wykorzystania protokołów sieciowych (jednakże z warstwy transportowej, a nie warstwy aplikacji).





Z uwagi na fakt, że obecnie dominującą siecią komputerową jest Internet i powiązany z nim model TCP/IP, to obecnie w praktyce najczęściej wykorzystuje się gniazda internetowe (ang. internet socket). Gniazdo jest wtedy jednoznacznie identyfikowane poprzez: protokół transportowy, adres IP i numer portu. Niektórzy nawet utożsamiają tę trójkę jako synonim gniazda, jednak jest to określenie lekko kolokwialne, opisujące właściwość, jaką ma dane gniazdo, a nie definiujące pojęcie gniazda. **Jednakże w ogólności gniazdo sieciowe mogłoby działać za pomocą innych protokołów niż IP i powiązane z nim protokoły transportowe (TCP, UDP). Przykładem może być IPX/SPX, czyli zestaw protokołów sieciowych firmy Novell. IPX jest protokołem wartwy sieciowej (odpowiednik protokołu IP), natomiast SPX jest protokołem warstwy transportowej (odpowiednik protokołu TCP).** Protokoły te były bardzo popularne jeszcze w latach dziewięćdziesiątych, po czym obecnie wychodzą już z użycia; niektórzy stosują je jeszcze w celach utrudnienia przeprowadzenia ataku sieciowego (pewna część złośliwego oprogramowania nie potrafi działać w tej technologii, co w konsekwencji udaremnia wykonanie ataku).

W dalszej części instrukcji będziemy pracować z gniazdami internetowymi z poziomu języka programowania Java. W języku Java gniazda TCP są obsługiwane poprzez dwie klasy: *ServerSocket* i *Socket*. Klasa *ServerSocket* umożliwia nasłuchiwanie na odpowiednim adresie i porcie (otwiera połączenie do komunikacji); klasa ta posiada metodę *accept*, która pozwala na połączenie się innego procesu z gniazdem (zwracana jest instancja klasy *Socket*). Jednakże *accept* nie tworzy intancji tej klasy, ale raczej wstrzymuje wykonanie wątku głównego aplikacji, do czasu, aż inny proces połączy się z gniazdem; wtedy właśnie zwracana jest instancja *Socket.* Klasa ta oznacza połączenie z już otwartym (zainicjowanym) gniazdem, jeżeli spróbujemy połączyć się z gniazdem, które jeszcze nie istnieje, to rzucony zostanie wyjątek (oznacza to, że z punktu widzenia języka Java, nie można utworzyć instancji klasy *Socket*, jeżeli nie utworzono wcześniej instancji klasy *ServerSocket*, która nasłuchuje na danym adresie i porcie). Klasa *Socket* posiada przede wszystkim strumienie wejściowy i wyjściowy (podobnie jak gniazda w systemie UNIX). Cała komunikacja pomiędzy procesami będzie sprowadzać się do pisania i odczytywania ze strumieni.

1. Przepisz, następujące klasy w języku Java:

```java
package sockets.helloworld;

import java.io.BufferedReader;
import java.io.DataOutputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.Inet4Address;
import java.net.ServerSocket;
import java.net.Socket;

public class WorldServer {

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

        System.out.println("Started server at: " + serverSocket.getLocalSocketAddress());
        Socket connection = serverSocket.accept();
        BufferedReader br = new BufferedReader(new InputStreamReader(connection.getInputStream()));
        String hello = br.readLine();
        System.out.println("Client sends: " + hello);

        DataOutputStream dataOutputStream = new DataOutputStream(connection.getOutputStream());
        dataOutputStream.writeBytes(" world!\n");
        System.out.println("Client answers: " + br.readLine());
        dataOutputStream.close();
        br.close();
        connection.close();
        serverSocket.close();
    }
}
```

klasa WorldServer.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()));

        output.writeBytes("Hello, " + "\n");
        output.flush();

        String response = socketBufferedReader.readLine();
        System.out.println("Sever responsed: " + response);
        output.writeBytes("Thank you!\n");
        output.flush();
        output.close();
        socketBufferedReader.close();
        socket.close();
    }
}
```

Klasa HelloClient.java

Powyższe klasy otwierają gniazdo sieciowe i następnie wymieniają się komunikatami tekstowymi. Jest to oczywiście wariant klasycznego programu Hello World!, tutaj dostosowany do obsługi gniazd sieciowych. Warto zwrócić uwagę, że kluczowe dla obsługi komunikacji międzyprocesowej są strumienie: wejściowy i wyjściowy gniazda. Znaki końca linii są tutaj potrzebne, ponieważ odczytujemy komunikaty za pomocą metody *readLine* (znak nowej linii zastosowano jako naturalne zakończenie dla komunikatu, zamiast stosować konkretną liczbę bajtów). Co ważne, ponieważ komunikacja jest dwukierunkowa to strumień wyjściowy serwera, jest strumieniem wejściowym klienta i strumień wejściowy serwera, jest jednocześnie strumieniem wyjściowym klienta. Metoda *flush* upewnia się, że *wszystkie* dane zostały już przesłane, po czym czyści zawartość strumienia. Po wymienieniu wszystkich komunikatów, zamykamy strumienie i samo połączenie z gniazdem, na koniec serwer przestaje nasłuchiwać na nowe połączenia, zwalniając zajęty port. Warto zauważyć, że komunikacja odbywa się tutaj naprzemiennie: to znaczy wyraźnie określono, że komunikację rozpoczyna klient, po czym następuje odpowiedź serwera i tak dalej, nie ma sytuacji w której klient i serwer jednocześnie próbują wysyłać sobie komunikaty.

- **Uruchom najpierw klasę *WorldServer.java*, a potem klasę *HelloClient.java*. Następnie spróbuj wykonać to w odwrotnej kolejności, co się stało?**
- **Do klasy serwera dopisz instrukcje, w której po otrzymaniu podziękowania od klienta, serwer odpowie jeszcze: „You are welcome!”.**
- **Zmodyfikuj klasę klienta, tak, aby otwierała gniazdo, na innym adresie IP, np. 127.0.0.2, ale nadal porcie 6000. Co obserwujesz?**
- **Zmodyfikuj konstruktor klasy ServerSocket na następujący `ServerSocket serverSocket = new ServerSocket(6000, 1, Inet4Address.getLocalHost());` Uruchom klasę i spróbuj uruchomić klienta, co się stało?**
- **Zmodyfikuj klasę klienta, tak, aby wysyłała dwa razy komunikat „Hello,” i uruchom obydwa programy. Co się stało? Spróbuj naprawić działanie programu, modyfikując klasę serwera.**

2. Przepisz następujące klasy:

```
package sockets.helloworld;
```

```java
import sun.misc.Signal;

import java.io.*;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketException;
import java.util.concurrent.atomic.AtomicBoolean;

public class ServerLoop {
    Socket connection;
    ServerSocket serverSocket;

    public ServerLoop() throws IOException {
        serverSocket = new ServerSocket(6000);
        connection = null;
    }

    public void close() throws IOException {
        if (connection != null) {
            this.connection.close();
        }
        if (serverSocket != null) {
            serverSocket.close();
        }
    }

    public static void main(String[] args) throws IOException {
        ServerLoop sl = new ServerLoop();

        System.out.println("Started server at: " + sl.serverSocket.getLocalSocketAddress());
        System.out.println(sl.serverSocket.getInetAddress());
        AtomicBoolean serverStopped = new AtomicBoolean(false);
        AtomicBoolean connectionNonEnded = new AtomicBoolean(true);

        Signal.handle(new Signal("INT"),
                signal ->
                {
                    System.out.println("Interrupted by Ctrl+C");
                    connectionNonEnded.set(false);
                    serverStopped.set(true);
                    try {
                        if (sl.connection != null) {
                            sl.connection.shutdownInput();
                            sl.connection.shutdownOutput();
                            sl.connection.close();
                        }
                        sl.close();
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                    System.out.println("stopped!");
                });

        while (!serverStopped.get()) {
            try {
                sl.connection = sl.serverSocket.accept();
                System.out.println("Accepted connection from: " + sl.connection.getLocalSocketAddress());
                connectionNonEnded.set(true);
            } catch (IOException e) {
                System.out.println("Server was stopped by user!");
            }
            DataOutputStream dataOutputStream = new DataOutputStream(sl.connection.getOutputStream());
            dataOutputStream.writeBytes("Accepted connection" + System.lineSeparator());
            dataOutputStream.flush();

            BufferedReader br = new BufferedReader(new InputStreamReader(sl.connection.getInputStream()));

            while (connectionNonEnded.get()) {
                if (serverStopped.get()) {
                    System.out.println("Server stopped");
                    break;
                }
                if (sl.connection.isClosed() || sl.connection.isInputShutdown()) {
                    System.out.println("Client closed connection!");
                    break;
                }
```

```
                String request;
                try {
                    request = br.readLine();
                } catch (SocketException e) {
                    System.out.println("Client closed connection!");
                    break;
                }
                if (request == null) {
                    break;
                }

                switch (request) {
                    case "all files": {
                        StringBuilder stringBuilder = new StringBuilder();
                        String[] txts = new File(".").list((file, s) -> s.endsWith(".txt"));

                        if (txts != null && txts.length > 0) {
                            System.out.println("There are txts");
                            for (String f : txts) {
                                stringBuilder.append(f);
                                stringBuilder.append(", ");
                            }
                            System.out.println(stringBuilder.toString());
                            dataOutputStream.writeBytes(stringBuilder.toString() + System.lineSeparator());
                            dataOutputStream.flush();
                        } else {
                            dataOutputStream.writeBytes("No files found!" + System.lineSeparator());
                            dataOutputStream.flush();
                        }
                    }
                    break;
                    case "new file": {
                        try {
                            String fileName = br.readLine();
                            String fileContent = br.readLine();
                            FileWriter fw = new FileWriter(new File(fileName));
                            fw.write(fileContent);
                            fw.close();
                            dataOutputStream.writeBytes("successfully transfered file" +
System.lineSeparator());
                            dataOutputStream.flush();
                        } catch (SocketException e) {
                            System.out.println("Client closed connection!");
                        }
                    }
                    break;
                    case "end": {
                        dataOutputStream.close();
                        br.close();
                        sl.connection.close();
                        connectionNonEnded.set(false);
                    }
                    break;
                    default: {
                        System.out.println("Bad request " + request);
                        for (char c : request.toCharArray())
                            System.out.println((int) c);
                        dataOutputStream.writeBytes("bad request!" + System.lineSeparator());
                        dataOutputStream.flush();
                    }
                }
            }
        }
    }
}
```

klasa *ServerLoop.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 ClientLoop {
    public static void main(String[] args) throws IOException {
        Socket socket = new Socket("127.0.0.1", 6000);
        DataOutputStream output = new DataOutputStream(socket.getOutputStream());
        BufferedReader socketBufferedReader = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
        System.out.println(socketBufferedReader.readLine());
        while (true) {
            try {
                output.writeBytes("all files" + System.lineSeparator());
                output.flush();

                String response = socketBufferedReader.readLine();
                System.out.println("Sever responsed: " + response);

                output.writeBytes("new file" + System.lineSeparator());
                output.flush();

                output.writeBytes("nowy.txt" + System.lineSeparator());
                output.flush();

                output.writeBytes("This is a content of a file, which will be send to a server!" +
System.lineSeparator());
                output.flush();

                response = socketBufferedReader.readLine();
                System.out.println("Server responded: " + response);
            } catch (SocketException e) {
                System.out.println("Serwer closed a connection!");
                break;
            }
        }
    }
}
```

klasa *ClientLoop.java*

Celem tego przykładu jest pokazanie implementacji własnego protokołu sieciowego warstwy aplikacji. Oczywiście protokół ten jest uproszczony i ma charakter demonstracyjny. Jest to w istocie system podobny do FTP, umożliwiający przesyłanie jednak jedynie plików tekstowych. Zarówno serwer, jak i klient przesyłają sobie dane w postaci łańcuchów znaków. Serwer obsługuje kilka komend: wyświetlenie wszystkich plików tekstowych, dodanie nowego pliku wraz z zawartością oraz zakończenie połączenia (zamknięcie otwartego gniazda). Klient musi znać te komendy i wysyłać je do serwera. Powyższe klasy działają w pętlach nieskończonych (możliwe jest jednak opuszczenie tych pętli). Klasa serwera nasłuchuje na połączenie z klientem; jeżeli ono nastąpi, to dopóki klient nie zakończy połączenia (albo za pomocą komendy end albo w wyniku np. zakończenia wykonywania jego procesu) to serwer będzie nasłuchiwał na komendy od klienta, w otwartym dla niego gnieździe (sewer jest tutaj jednowątkowy i nie potrafi obsługiwać więcej niż jednego klienta). Dodatkowo serwer nasłuchuje na zdarzenie naciśnięcia ctrl+c, tożsame z poleceniem zakończenia wykonywania procesu. Jest to zaimplementowane za pomocą sygnałów systemu operacyjnego (są to sygnały zasadniczo raczej systemu operacyjengo UNIX i jego sukcesorów, które jednak w języku Java działają na każdej platformie). *System.lineSeparator()* pozwala zdefiniować poprawnie znak końca linii zarówno w systemie Linux, jak i Windows (ma to znacznie korzystając z konsoli, w tym telnetu).

- **Zbuduj klasę serwera jako jar i uruchom tę klasę w konsoli (mogą być potrzebne uprawnienia administratora). Uruchom również klasę klienta. Co stanie się po naciśnięciu ctrl+c w konsoli serwera (z perspektywy zarówno klienta, jak i serwera)?**
- **Ponownie uruchom klasę serwera i spróbuj uruchomić np. 3 instancję klasy klienta. Co się stało? Zakończ działanie jednego z procesów klienta i obserwuj pozostałe 2 procesy, co się stało?**
- **Zresetuj klasę serwera (na klastrze ICMK) i upewnij się, że nie jest uruchomiony żaden klient. Uruchom na klastrze ICMK telnet (będzie tutaj służył po prostu do**

połączenia się z gniazdem, a niekoniecznie do komunikacji ze zdalnym hostem), za pomocą komendy:
telnet 127.0.0.1 6000
W ten sposób połączysz się z serwerem i będziesz mógł wywoływać komendy w naszym przykładowym protokole. Spróbuj najpierw wpisać komendę: *all files*<*enter*>. Po czym dodaj nowy plik tekstowy za pomocą komend: *new file*<*enter*> <*nazwapliku*><*enter*> <*zawartość pliku*><*enter*>. Połączenie zakończ wpisując komendę *end.*

- **Wzorując się na przykładach dodaj komendę *get file*, która pozwoli pobrać plik tekstowy z serwera. Przetestuj działanie tej komendy za pomocą telnetu.**
- **Następnie połącz się z serwerem uruchomionym w klastrze ICMK z poziomu innego komputera za pomocą telnetu; wywołaj all files, po czym get file <plik>.**
- **Zmień implementację klasy *ClientLoop.java* tak aby połączyła się z poprawnie klastrem ICMK.**