Programowanie sieciowe w Javie

- Wprowadzenie
- Przegląd klas pakietu java.net
- Reprezentacje adresów
- Połączenia gniazdowe
- Datagramy

Protokoły sieciowe - podstawy

Protokół sieciowy jest to zbiór procedur oraz reguł rządzących komunikacją, między co najmniej dwoma urządzeniami sieciowymi. Istnieją różne protokoły, lecz nawiązujące w danym momencie połączenie urządzenia muszą używać tego samego protokołu, aby wymiana danych pomiędzy nimi była możliwa.

Podstawowe rodziny protokołów sieciowych:

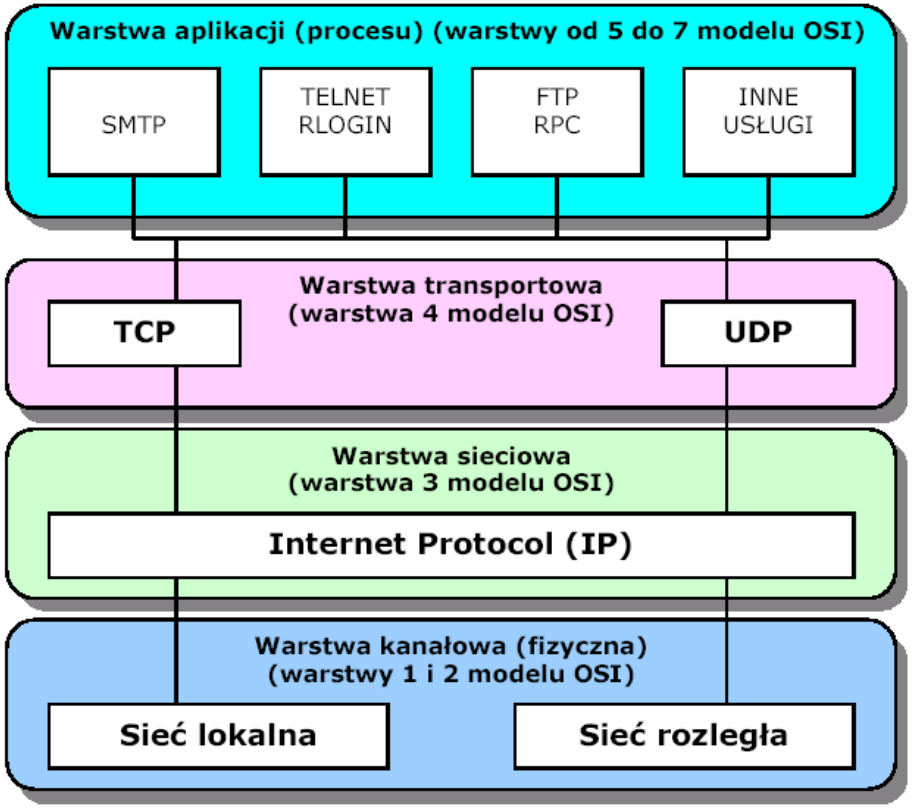
- **NetBEUI** – opracowany w 1985 r. przez firmę IBM. Używany w małych, odizolowanych sieciach LAN typu peer-to-peer.
- **IPX/SPX** – opracowany w latach 70-tych przez firmę Novell, może być wykorzystywany do budowy złożonych sieci.
- **TCP/IP** – najczęściej stosowany zestaw protokołów sieciowych.

Łączą komputery pracujące na różnych platformach sprzętowych i systemowych. Specyfikacje protokołów można znaleźć w dokumentach **RFC** (Request for Comments)

Teoretyczne modele sieciowe

OSI (Open Systems Interconnection)

DoD (US Department of Defense)/TCP/IP



Wprowadzenie do TCP/IP

- zestaw protokołów komunikacyjnych (RFC 793), którego nazwa pochodzi od dwóch najważniejszych (spośród wielu):
  - kontroli transmisji (*Transmission Control Protocol*, TCP),
  - protokołu internetowego (*Internet Protocol*, IP)
- TCP/IP udostępnia metody transmisji informacji pomiędzy poszczególnymi hostami w sieci, obsługując pojawiające się błędy oraz tworząc wymagane do transmisji informacje dodatkowe.
- otwarty standard, dostępny bezpłatnie i stworzony niezależnie od platformy sprzętowej czy programowej co służy integracji różnych sieci,
- IP zapewnia jednolity system adresowania, pozwalający w identyczny sposób zaadresować każde urządzenie w sieci, nawet tak dużej jak Internet,

Transmisja TCP

- Protokół połączeniowy,
- Niezawodne i wiarygodne przesyłanie danych (segmenty są dostarczone do miejsca przeznaczenia w kolejności, w jakiej zostały wysłane, bez uszkodzeń i strat)
- Oparty na sygnale ACK potwierdzenia z retransmisją,
- Kontrola przepływu, korekcja błędów,
- Skierowany na strumieniową (jednolitą) transmisję danych (przesyłanie dużych ilości ciągłych informacji – np. pliki)
- RFC 793.

Przesył UDP (*User Datagram Protocol*)

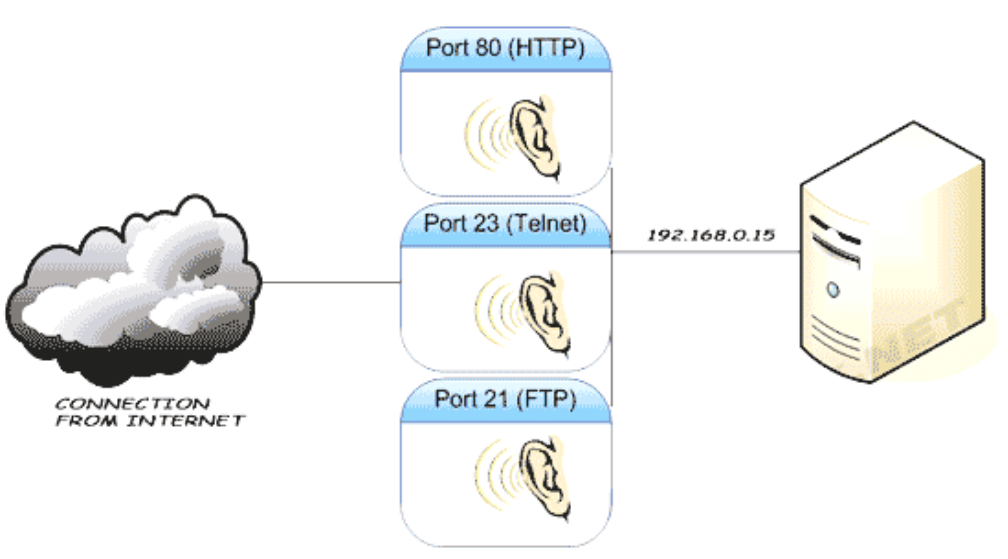
- Protokół bezpołączeniowy, w którym pojedynczy pakiet (datagram) integralną informację.
- Bez narzutu na nawiązywanie połączenia i śledzenie sesji (vide TCP),
- Nie korzysta z mechanizmów kontroli przepływu i retransmisji – zawodny
- Duża szybkość transmisji danych,
- Dla zastosowań gdzie dane muszą być przesyłane możliwie szybko, a poprawianiem błędów mogą zajmować się inne warstwy modelu OSI, np. wideokonferencje, strumienie dźwięku w Internecie, gry sieciowe, itp.
- RFC 768.

Adresy sieciowe

- **adres fizyczny (MAC)** - nadawany przez producentów kart sieciowych i należący do warstwy fizycznej DoD np. ***00-50-56-C0-0A-01***,
- **adres domenowy** - łatwiejszy do zapamiętania słowny odpowiednik adresu IP, wykorzystujący serwery DNS i należący do warstwy aplikacji DoD np. ***www.issi.uz.zgora.pl***
- **adres IP** - unikatowy identyfikator nadawany komputerowi w sieci IP, należący do warstwy komunikacyjnej DoD i pozwalający na lokalizację komputera w sieci, np. ***84.10.191.2***

Port protokołu

- Adres wewnętrzny zapewniający interfejs pomiędzy aplikacjami sieciowymi, a warstwą transportową sieci,
- **Numer portu źródłowego** procesu, który wysłał dane oraz **numer portu docelowego** procesu, który ma dane otrzymać, są zawarte w pierwszym słowie nagłówka każdego segmentu TCP i UDP



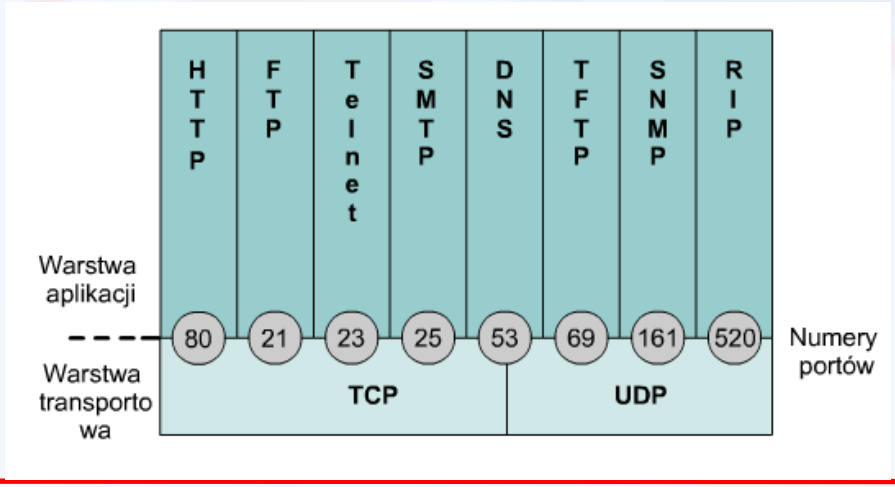
Reprezentacja i wartości portów

- Port reprezentowany jest przez liczbę z zakresu 0-65535 (16 bitów)
- Numery portów mają przydzielone następujące zakresy:
  - numery poniżej 1024 są uważane za dobrze znane numery portów,
  - 1024 – 49151 zarejestrowane porty. Mogą z nich korzystać programy i procesy zwykłych użytkowników,
  - 49152 – 65535 porty dynamiczne i/lub prywatne.
- Numery protokołów i portów przyporządkowane znanym usługom zdefiniowane są w RFC 1700
- Host źródłowy dynamicznie przydziela numery portów źródła rozpoczynającego transmisję. Numery te są zawsze większe od 1023.

Reprezentacja i wartości portów

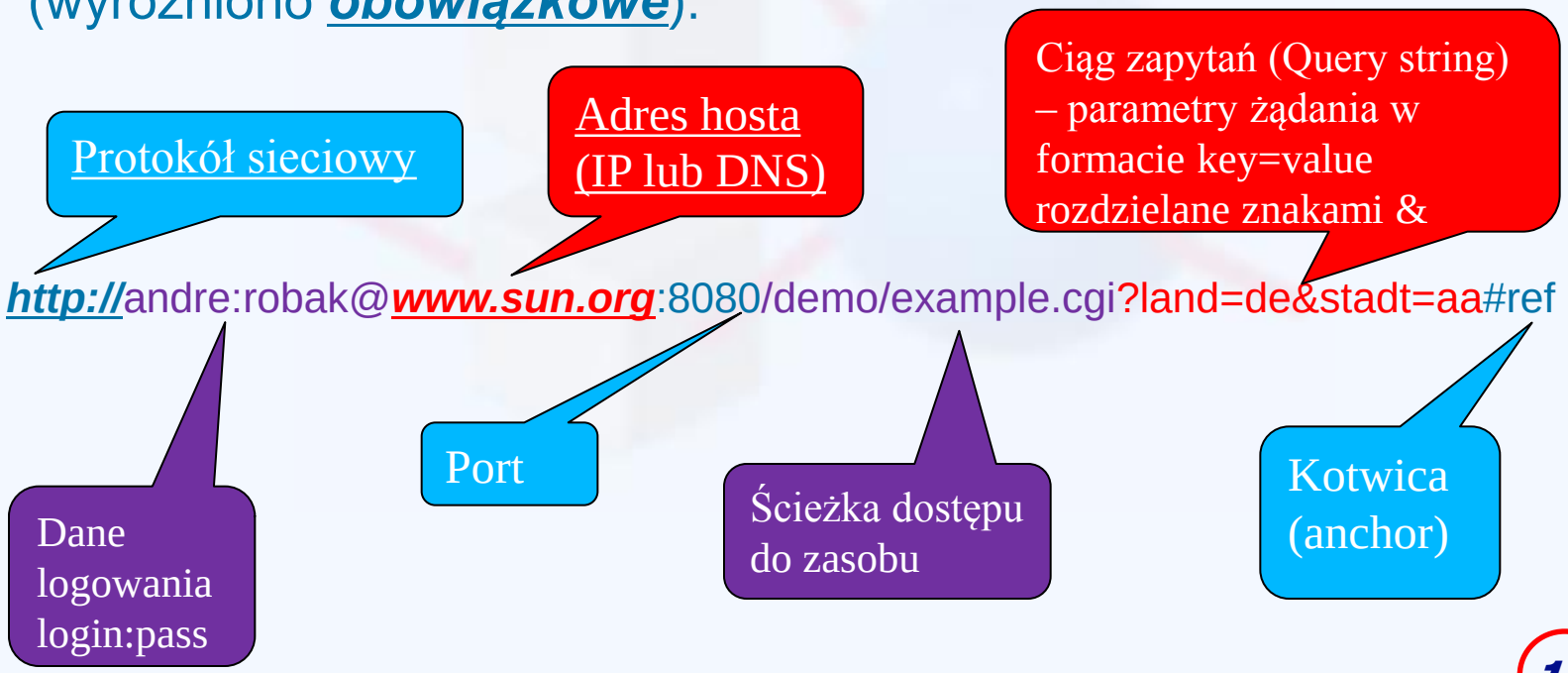
- Różne usługi mogą używać tego samego numeru portów, pod warunkiem że korzystają z innego protokołu (TCP lub UDP), chociaż istnieją także usługi korzystające jednocześnie z jednego numeru portu i obu protokołów, np. DNS - korzysta z portu 53 za pomocą TCP i UDP jednocześnie.

- Zdarza się także, że jedna usługa może korzystać z dwóch różnych portów używanych do innych zadań (np. FTP, SNMP).



Adres URL

URL (*Uniform Resource Locator*) jest adresem zasobów w Internecie opisanym w specyfikacji RFC1738. URL jest używany przez przeglądarki internetowe w celu lokalizacji i dostępu do informacji na stronach WWW. Przykładowy adres zasobu (wyróżniono ***obowiązkowe***):



Java i programowanie sieciowe

- Javę zaprojektowano do tworzenia programów sieciowych
- Java dostarcza wiele interfejsów niskiego i wysokiego poziomu dostępnych w pakiecie *java.net* oraz pakietach uzupełniających (np. *javax.servlet, javax.smtp, etc.*) , m.in.:
  - Klasy reprezentujące adresy
    - *InetAddress*,
    - *URL + URLConnection*
  - Klasy połączeń opartych na TCP:
    - *Socket*
    - *ServerSocket*
  - Klasy połączeń opartych na UDP:
    - *DatagramSocket*
    - *DatagramPacket*

Klasa InetAddress

- reprezentuje istniejący adres (adresy) IP komputera (hosta) w sieci
- nie posiada publicznych konstruktorów - utworzenie obiektu możliwe jest dzięki statycznym metodom fabrycznym (*factory methods*) :

public static InetAddress getByName(String host) throws UnknownHostException
tworzy obiekt adresu hosta o podanej nazwie – IP lub DNS (jeśli istnieje)

public static InetAddress getByAddress(byte[] addr) throws UnknownHostException
tworzy obiekt adresu hosta na podstawie tablicy bajtów np. new byte[] {127, 0, 0, 1}

public static InetAddress[] getAllByName(String host) throws UnknownHostException
tworzy i inicjalizuje tablicę obiektów adresów hosta o podanej nazwie IP lub DNS

public static InetAddress getLocalHost() throws UnknownHostException
tworzy obiekt adresu lokalnego hosta

InetAddress – przykład użycia

```
try {
InetAddress adresIP = InetAddress.getByName("www.issi.uz.zgora.pl");
System.out.println("Nazwa hosta: " + adresIP.getHostName());

System.out.println("Numer IP: " + adresIP.getHostAddress());

System.out.println("?Grupowy (rozgłoszeniowy): " +
adresIP.isMulticastAddress());

System.out.println("?Osiągalne Echo: " + adresIP.isReachable(10));

System.out.println("?Prywatny: " + adresIP.isSiteLocalAddress());

System.out.println(adresIP);

} catch (UnknownHostException e) {
System.err.println(e);
}
```

Klasa URL

- Reprezentuje abstrakcyjny adres zasobu sieciowego.
- Adres URL może składać się z następujących elementów (dostępnych przy użyciu metod dostępowych *get*) :
  - protokół (wymagane) – *getProtocol()*
  - host (wymagane) – *getHost()*
  - port – *getPort()*
  - plik – *getFile()*
  - identyfikator fragmentu (np. ref, section, anchor) – *getRef()*
  - ciąg zapytania (query) – *getQuery()*
  - dane uwierzytelniające – *getAuthority(),*
- Obiekty tworzone są wyłącznie przy użyciu konstruktorów, ww. elementy adresu nie posiadają metod ustawiających *set-*

Konstruktory URL

- Konstruktor adresu bezwzględnego (absolute URL):

public URL(String spec) throws MalformedURLException

np.
```
URL urlAddress=new URL("http://www.issi.uz.zgora.pl");
```

- Konstruktor adresu względnego (relative URL):

public URL(URL host, String spec) throws MalformedURLException

np.
```
new URL(urlAddress,"?action=111&id=6");
new  URL(urlAddress, "#bottom");
```

Istnieją także konstruktory, umożliwiające podanie kolejnych komponentów adresu URL np. adres :

```
new URL("http", "www.gamelan.com", "/pages/Gamelan.html");
```

jest równoznaczny z:

```
new URL ("http://www.gamelan.com/pages/Gamelan.html");
```

oraz

```
new URL("http", "www.gamelan.com", 80,
"pages/Gamelan.html");
```

i reprezentuje on następujący adres URL:

```
http://www.gamelan.com:80/pages/Gamelan.html
```

Każdy z konstruktorów może wyrzucić wyjątek MalformedURLException, gdy przekazane argumenty odnoszą się do złego protokołu:

```
try{
URL myURL = new URL(...)
}
catch (MalformedURLException e)
{...
//obsługa wyjątku
...}
```

Rozpoznawane protokoły:
- file, ftp, http,
- gopher, mailto,
- appletresource, doc, netdoc, systemresource, verbatim

Elementy URL - przykład

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

public class ParseURL {
    public static void main(String[] args) throws Exception {
        URL aURL = new URL("http://java.sun.com:80/docs/
                           + "tutorial/index.html#DOWNLOADING");
System.out.println("protocol = " +aURL.getProtocol());
System.out.println("host = " + aURL.getHost());
System.out.println("filename = " + aURL.getFile());
System.out.println("port = " + aURL.getPort());
System.out.println("ref = " + aURL.getRef());    }}
```

Powyższy kod zwróci następujące wyniki:

```
protocol = http
host = java.sun.com
filename = /docs /tutorial/index.html
port = 80
ref = DOWNLOADING
```

URL – odczyt z zasobu sieciowego

Obiekt URL udostępnia obiekt binarnego strumienia wejściowego (metoda openStream() otwiera połączenie do zasobu).

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

public class URLReader{
public static void main(String[] args) throws Exception{
URL myURL = new URL("http://www.issi.uz.zgora.pl/");
BufferedReader in = new BufferedReader(
        new InputStreamReader(myURL.openStream()));
    String inpuLine;
    while((inputLine= i.readLine())!=null)
        System.out.println(inputLine);
    in.close();
}
}
```

Po wykonaniu powinien ukazać się cały dokument HTML, którego adres przekazaliśmy do konstruktora obiektu klasy URL. Może się zdarzyć, że system operacyjny będzie wymagał ustawienia Proxy:

```
java -Dhttp.proxyHost=proxyhost URLReader
```

Klasa URLConnection

Abstrakcyjna klasa URLConnection zapewnia metody komunikacji z zasobem URL - przede wszystkim pozwalające pobrać strumienie do zapisu i odczytu. Podklasy abstrakcyjne: HttpURLConnection, JarURLConnection

Obiekt URLConnection tworzony jest metodą `openConnection()` klasy URL, której użycie inicjuje połączenie komunikacyjne (HTTP lub JAR). Gdy połączenie jest niemożliwe zwracany jest wyjątek *IOException.*

```
try {
    URL myURL = new URL("http://www.issi.zgora.pl /");
    URLConnection urlConnection = myURL.openConnection();

} catch (MalformedURLException e) { // błąd tworzenia obiektu URL
    . . .
} catch (IOException e) {           // błąd metody openConnection()
    . . .
}
```

URLConnection - odczyt

Poniższy program wykonuje analogiczne zadania jak wcześniej omówiony odczyt z obiektu URL. Różnice między programami zaznaczono wytłuszczoną czcionką.

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

public class URLConnectionReader {
    public static void main(String[] args) throws Exception {
        URL myURL = new URL("http://www.issi.uz.zgora.pl/");
        URLConnection urlConn = myURL.openConnection();
        BufferedReader in = new BufferedReader(
                                new InputStreamReader(
                                urlConn.getInputStream()));
        String inputLine;

        while ((inputLine = in.readLine()) != null)
            System.out.println(inputLine);
        in.close();
    }}
```

URLConnection - zapis

- Aplikacje internetowe zawierają formularze- pola tekstowe i inne elementu GUI, umożliwiające użytkownikowi wprowadzić dane i wysłać je na serwer. Po wypełnieniu formularza i potwierdzeniu wysłania przeglądarka wysyła dane do URL. Następnie dane te są przetwarzane po stronie serwera przez aplikacje (skrypty CGI, serwlety), w wyniku czego użytkownik otrzymuje odpowiedź w formie nowej strony.

- Programy Javy mogą współpracować ze skryptami CGI lub serwletami po stronie serwera poprzez zapis informacji do strumienia wyjściowego dostępnego z obiektu URLConnection.

Zapis do URLConnection - schemat

1. Utworzenie obiektu URL;
2. Otwarcie połączenia URLConnection;
3. Ustawienie pojemności wyjściowej połączenia;
4. Pobranie strumienia wyjściowego z połączenia. Jest on zazwyczaj połączony ze strumieniem wejściowym skryptu CGI, serwletu bądź strony PHP na serwerze;
5. Zapisanie do strumienia wyjściowego;
6. Zamknięcie strumienia wyjściowego.

Poniższy program przesyła łańcuch znakowy do skryptu CGI umieszczonego na stronie firmy Sun, który odwraca kolejność znaków i odsyła go z powrotem.

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

public class Reverse {
   static final  String ADDRESS= " http://java.sun.com/cgi-bin/backwards ";
   public static void main(String[] args) throws Exception {
       if (args.length != 1) {  // tylko jedno słowo
          System.err.println("Usage:  java Reverse  string_to_reverse");     System.exit(1);
       }

       String stringToReverse = URLEncoder.encode(args[0], "UTF-8");  //kodowanie

       URL url = new URL(ADDRESS);
       URLConnection connection = url.openConnection();
       connection.setDoOutput(true); //ustawienie w tryb zapisu

       OutputStreamWriter out = new OutputStreamWriter(connection.getOutputStream()); //pobranie strumienia
       out.write("string=" + stringToReverse); //zapis
       out.close(); //zamknięcie strumienia wyjściowego do zasobu

       BufferedReader in = new BufferedReader(new InputStreamReader(connection.getInputStream()));
       String decodedString;
       while ((decodedString = in.readLine()) != null) {
              System.out.println(decodedString);     }  //odczytanie ze strumienia wejściowego
       in.close();
   }
}
```

Gniazdo (socket)

- URL i URLConnection dostarczają interfejs sieciowy względnie wysokiego poziomu. Czasami wymagany jest niższy poziom komunikacji sieciowej, np. połączenia klient-serwer.
- Gniazdo jest zdefiniowanym programowo jednym z końców dwu-kierunkowego połączenia pomiędzy dwoma programami pracującymi w sieci.
- Stanowi kombinację adresu IP oraz numeru portu (terminy gniazdo i numer portu często używane są zamiennie),
- Gniazdo jednoznacznie określa każdy program sieciowy w Internecie.

Rozróżniamy:

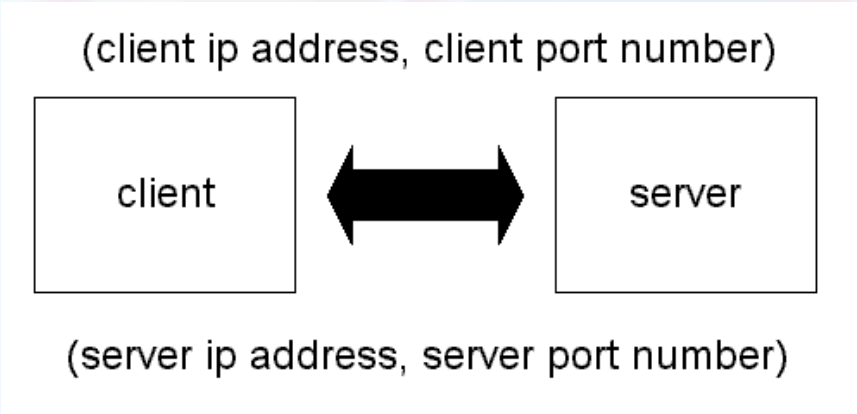
- Gniazda potokowe (TCP)
- Gniazda datagramowe (UDP)

Połączenie gniazdowe

Klient wysyła w sieci TCP/IP żądanie połączenia z innym węzłem, przekazując numer gniazda (portu). Jeśli węzeł odbiorcy może przyjąć połączenie, zwraca numer gniazda (nowo tworzonego) zawierający adres IP odbiorcy i numer portu usługi, która będzie obsługiwać zadanie.

Zgodnie z RFC 793 połączenie gniazdowe definiuje para gniazd czyli czwórka:

(remoteAddress, remotePort, localAddress, localPort).

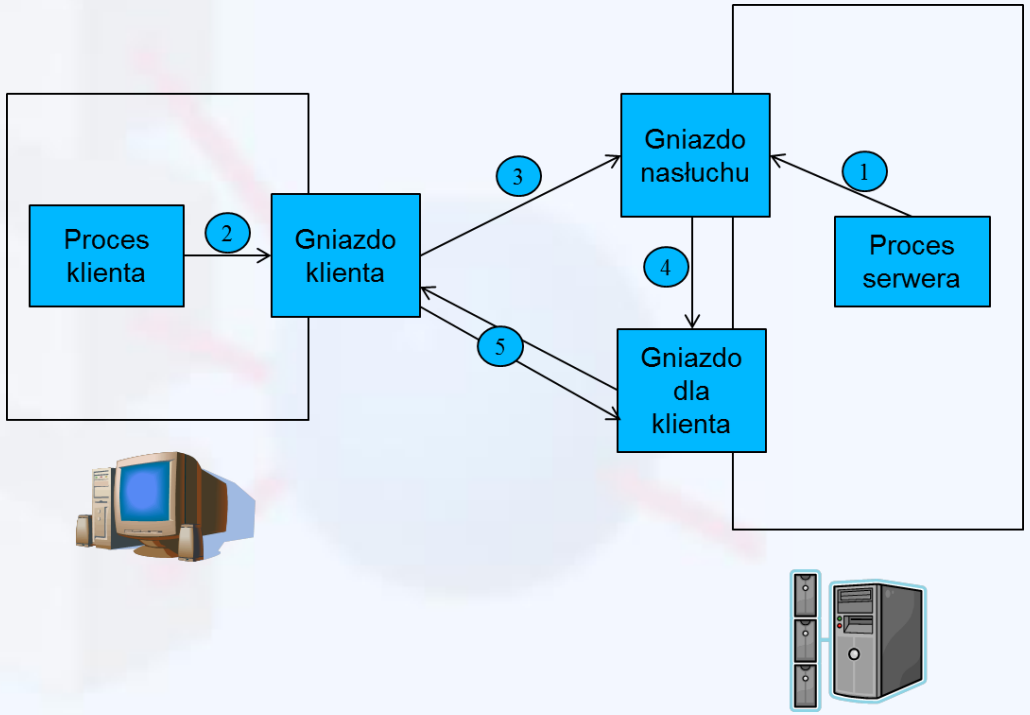


Połączenie klient - serwer

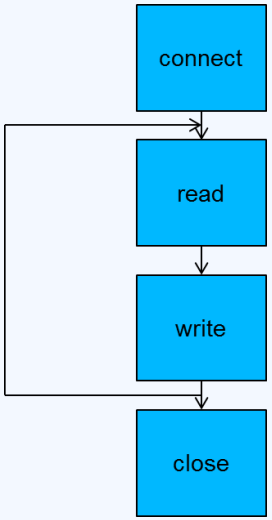
- Serwer świadczy usługi lub udostępnia zasoby czekając i nasłuchując na określonym porcie.
- Klient natomiast jest stroną żądającą dostępu do danej usługi lub zasobu. Klient musi znać adres gniazda serwera.
- Klient i serwer mogą pracować na tym samym komputerze, używając mechanizmów komunikacji lokalnej, lub na różnych komputerach, wykorzystując komunikację przez sieć.

Połączenie klient - serwer

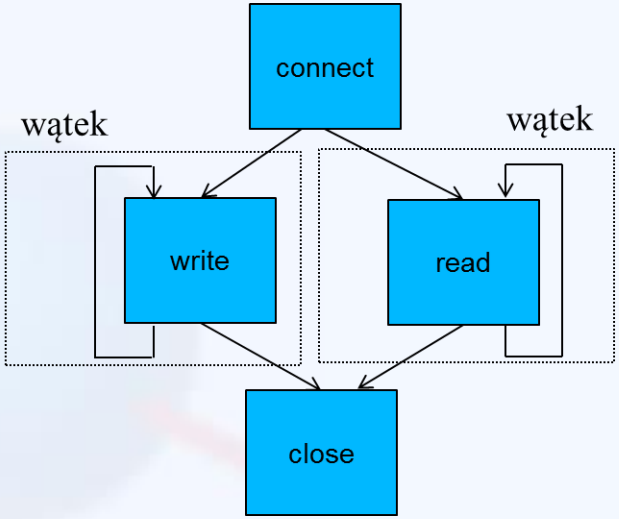
1. Proces serwera tworzy gniazdo nasłuchiwania klientów
2. Proces klienta tworzy gniazdo klienta do połączenia
3. Klient inicjuje połączenie i przesyła dane swojego gniazda
4. Serwer **akceptuje** żądanie i tworzy nowe gniazdo do komunikacji z klientem, uwalniając gniazdo nasłuchu dla innych połączeń
5. Klient i serwer komunikują się z użyciem ustalonego protokołu



Implementacja klienta – zadania



1. utworzenie gniazda;
2. otwarcie strumieni IO dla gniazda;
3. odczyt i zapis z/do strumieni zgodnie z wykorzystywanym protokołem przez serwer (komunikacja **synchroniczna** lub **asynchroniczna**);
4. zamknięcie strumieni;
5. zamknięcie gniazda.



Implementacja klienta – klasa Socket

- konstruktor służy do nawiązywania połączenia - należy **obowiązkowo** podać stację zdalną i jej port;
  - `public Socket(String host, int port) throws UnknownHostException, IOException`
  - `public Socket(InetAddress address, int port) throws IOException`
  - `public Socket(String host, int port, InetAddress localAddr, int localPort) throws IOException`
- konstruktor pozwala wskazać także adres i port lokalny z którego będziemy się łączyć, ale uwaga - w danej chwili na jednym porcie może być realizowane tylko jedno połączenie
- obiekt Socket dostarcza referencje do podstawowych strumieni binarnych, które można dowolnie opakowywać (np. DataInputStream, ObjectOutputStream, BufferedReader, etc.)

Implementacja gniazda - wyjątki

```
try {
        Socket gniazdo = new Socket("www.issi.uz.zgora.pl", 21);
        System.out.println("Nawiązano połączenie");
        gniazdo.close();
    } catch (UnknownHostException e) {
        System.out.println("komputer nie jest znany");
    } catch (NoRouteToHostException e) {
        System.out.println("komputer jest niedostępny, firewall");
    } catch (ConnectException e) {
        System.out.println("komputer odrzucił polaczenie lub/i nie
nasłuchuje na porcie");
    } catch (PortUnreachableException e) {
        System.out.println("nieosiągalny port");
    } catch (IOException e) {
        System.out.println("Wystąpił błąd IO"+e);
    }
```

Połączenia - przykład

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

public class Skaner {
public static void main(String[] args) {
for (int port = 1; port < 65536; port++) {
        try {
        Socket s = new Socket("localhost", port);
        System.out.println("otwarty port:"+port);
        s.close();
        } catch (IOException e) {
        System.out.println("zamkniety:"+port);
        }
}}}
```

Komunikacja potokami - przykład

Przykładem ilustrującym nawiązanie połączenia z serwerem, a następnie wysyłanie i odbiór danych jest aplikacja ***EchoClient***.

Program klienta ***EchoClient*** łączy się z usługą ***Echo*** - znaną usługą działającą na porcie 7, która "odbija" odebrane komunikaty z powrotem do klienta.

EchoClient czyta strumień wejściowy użytkownika, a potem wysyła go do serwera. Po „odbiciu” informacji klient czyta dane i wyświetla je na konsoli.

Uwaga: Aby przetestować działanie aplikacji, usługa Echo systemu operacyjnego serwera musi być włączona (Windows IIS domyślnie wyłącza usługę Echo).

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

public class EchoClient {
    public static void main(String[] args) throws IOException {

        Socket echoSocket = null;
        PrintWriter out = null;
        BufferedReader in = null;

        try {
            echoSocket = new Socket("localhost", 7);
            out = new PrintWriter(echoSocket.getOutputStream(),
                                                    true);
            in = new BufferedReader(new InputStreamReader(
                                    echoSocket.getInputStream()));
        } catch (UnknownHostException e) {
            System.err.println("Don't know about host: localhost.");
            System.exit(1);
        } catch (IOException e) {
            System.err.println("Couldn't get I/O for "
                               + "the connection to: localhost.");
            System.exit(1);
        }
```

```
BufferedReader stdIn = new BufferedReader(
                                   new InputStreamReader(System.in));
String userInput;

while ((userInput = stdIn.readLine()) != null) {
    out.println(userInput);
    System.out.println("echo: " + in.readLine());
}

out.close();
in.close();
stdIn.close();
echoSocket.close();
   }
}
```

Kluczowymi są linie:

```
echoSocket = new Socket("localhost", 7);
out = new PrintWriter(echoSocket.getOutputStream(),
                                          true);
in = new BufferedReader(new
InputStreamReader(echoSocket.getInputStream()));
```

Użyty konstruktor wymaga podania nazwy hosta i portu na którym mają się komunikować. Dwie pozostałe komendy otwierają strumienie wejściowe i wyjściowe gniazda.

Wyjątki

- **BindException** – próba utworzenia obiektu ***Socket*** lub ***ServerSocket*** na używanym porcie lokalnym lub brak uprawnień;
- **ConnectException** – zdalna stacja odmówiła połączenia np. z powodu zajętości stacji lub braku procesu nasłuchu na wskazanym porcie;
- **NoRouteToHostException** – połączenie przekroczyło przydzielony czas;
- **SecurityException** – wyjątek związany z próbą wykonania akcji naruszającej bezpieczeństwo;
- **InterruptedIOException** – wystąpienie „timeout'u" po ustawienie czasu oczekiwania.

Implementacja gniazd serwera

- serwer działa w nasłuchu na konkretnym porcie;
- nie jest znany adres klienta
- gniazdo nasłuchu implementowane jest przez obiekt klasy ServerSocket, która dostarcza:
  - konstruktory obiektów;
  - metody do nasłuchu oraz przyjęcia przychodzącego połączenia;
  - metody parametryzujące działanie np. podstawowy czas życia serwera.

Schemat implementacji serwera

1. utworzenie obiektu ***ServerSocket***, związanego z konkretnym portem na lokalnej stacji;
2. ***ServerSocket*** nasłuchuje na połączenia za pomocą metody ***accept()***; metoda ta blokuje się do wystąpienia próby połączenia; wówczas ***accept()*** zwraca obiekt ***Socket***, łączący serwer z klientami;
3. W zależności od typu serwera, są wywoływane odpowiednie metody do utworzenia strumieni wejściowych i wyjściowych – komunikacja odbywa się już na zwykłych ***Socketach***
4. Serwer i klient działają w oparciu o ustalony protokół aż do momentu zamknięcia połączenia;
5. Połączenie zamyka serwer, klient lub obie strony jednocześnie
6. Serwer wraca do pozycji 2.

Konstruktory ServerSocket

```
public ServerSocket(int port)
public ServerSocket(int port, int queue)
public ServerSocket(int port, int queue, InetAddress bindAddr)
```

- tworzą obiekty gniazd nasłuchujących związane ze wskazanym **portem** lokalnego komputera;
- **port=0** oznacza, że wybór portu należy do systemu (port anonimowy);
- mogą generować ***IOException***, ***BindException***;
- ***queue*** określa długość kolejki pamiętanych połączeń oczekujących; system operacyjny kolejkuje (FIFO) połączenia dla każdego portu, po zapełnieniu kolejki kolejne są odrzucane, chyba że zwolni się miejsce,
- konstruktor z argumentem ***bindAddr*** oznacza, że serwer jest dowiązany (binding) do danego portu; konstruktor przydatny dla hostów posiadających wiele adresów IP (interfejsów), pozwalający wskazać konkretny interfejs, do którego może się podłączyć klient.

Serwer akceptuje połączenie poprzez zastosowanie metody ***accept()***:

- blokuje ona wątek serwera w oczekiwaniu na połączenie;
- po połączeniu metoda zwraca obiekt ***Socket***, który służy do komunikacji.

```
try{ ServerSocket s = new ServerSocket(8877);
while (true){
  Socket conn = s.accept();
  PrintStream p = new PrintStream (conn.getOutputStream());
  p.println("Połączyłeś sie z serwerem."+"BYE");
  conn.close();
}
}catch(IOException){....}
```

Przykład - KnockKnockServer

Tworzenie obiektu typu ServerSocket i nasłuchiwanie na przykładowym porcie:

```
try
{
 serverSocket = new ServerSocket(4444);
}
 catch (IOException e)
{
System.out.println("Could not listen on port: 4444"); System.exit(-1);
}
```

Akceptowanie połączenia od klienta:

```
Socket clientSocket = null;
try {
 clientSocket = serverSocket.accept();
}
catch (IOException e) { System.out.println("Accept failed:
4444");
System.exit(-1);
}
```

Otwieranie strumieni

```
PrintWriter out= new PrintWriter(clientSocket.getOutputStream(),true);
BufferedReader in = new BufferedReader(new InputStreamReader(
clientSocket.getInputStream()));
String inputLine, outputLine;
```

Inicjowanie połączenia z klientem poprzez protokół **KnockKnockProtocol()**.

```
KnockKnockProtocol kkp = new KnockKnockProtocol();
outputLine = kkp.processInput(null);
out.println(outputLine);
```

Odsyłanie danych do klienta

```
while ((inputLine = in.readLine()) != null){
    outputLine = kkp.processInput(inputLine);
    out.println(outputLine);
    out.flush();
    if outputLine.equals("Bye.")) break; }
```

zamykanie strumieni i gniazd

```
out.close(); in.close();clientSocket.close();
serverSocket.close();
```

Przykład - KnockKnockClient

Klient otwiera gniazdo połączone z działającym już serwerem:

```
kkSocket = new Socket("taranis", 4444);
out = new PrintWriter(kkSocket.getOutputStream(), true);
in = new BufferedReader(new InputStreamReader(
kkSocket.getInputStream()));
```

Klient wysyła tekst do serwera poprzez strumień wyjściowy skojarzony z gniazdem:

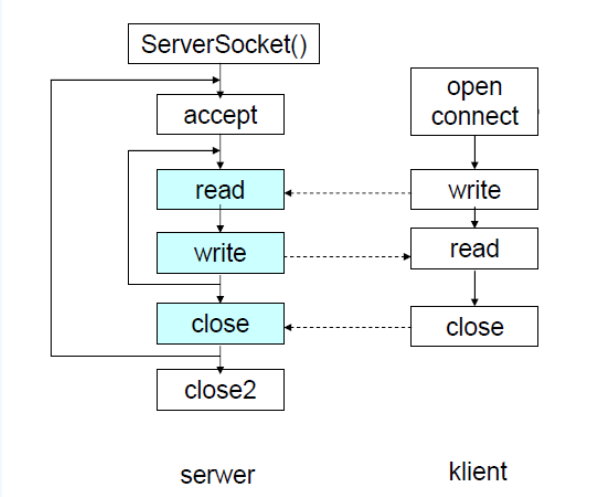
```
while ((fromServer = in.readLine()) != null)
{
 System.out.println("Server: " + fromServer);
 if (fromServer.equals("Bye.")) break;
 fromUser = stdIn.readLine(); if (fromUser != null) {
 System.out.println("Client: " + fromUser);
 out.println(fromUser); }
}
```

Na zakończenie klient zamyka wszystkie strumienie oraz gniazdo:

```
out.close();
in.close();
stdIn.close();
kkSocket.close();
```

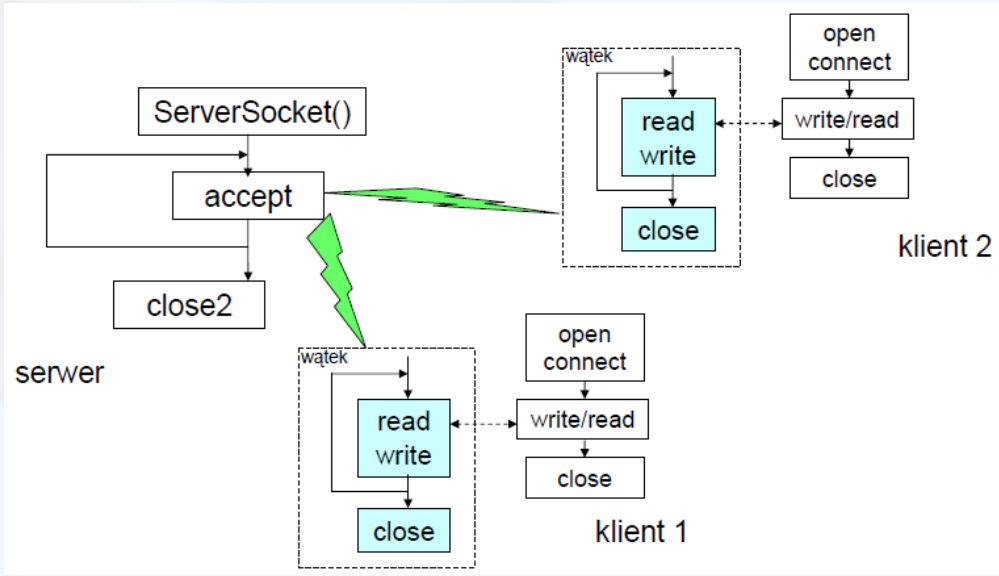
Kod źródłowy znajduje się w liście nr 5

Schematy obsługi klienta



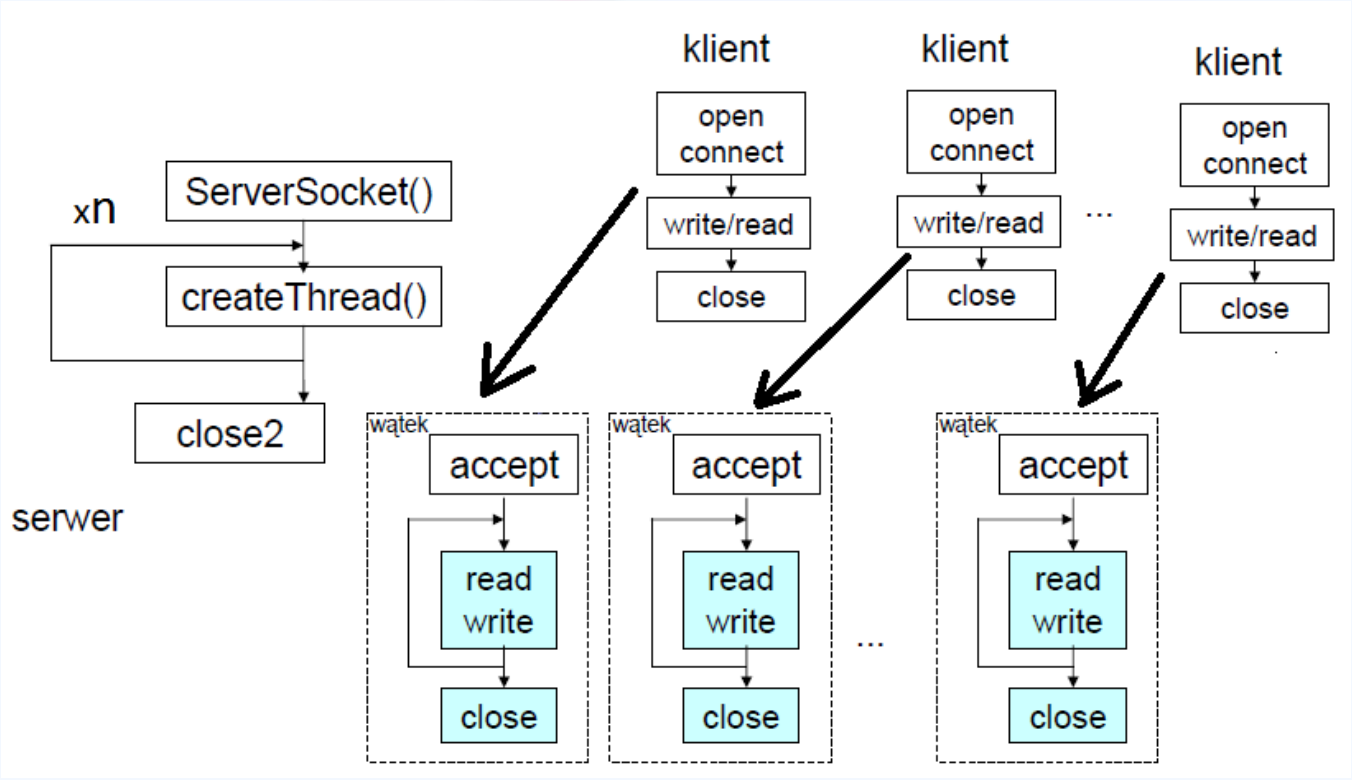
iteracyjny – proces serwera zajmuje się obsługą jednego klienta, stosowany jest gdy czas obsługi klienta jest krótki lub kiedy używa go tylko jeden klient.

współbieżny – dla obsługi klienta, proces serwera uruchamia nowy wątek.



Schematy obsługi klienta cd.

współbieżny z wykorzystaniem puli **n** wątków
(najefektywniejszy z punktu widzenia czasu obsługi klienta)



Schematy obsługi klienta - przykład

```
public static void main(String[] args) {
ServerSocket serverSocket = null;
Socket clientSocket = null;
boolean listening = true; //unreacheable code below while otherwise

try {
serverSocket = new ServerSocket(4444);
} catch (IOException e) {
System.err.println("Unavailable port: 4444.");System.exit(1);
}
try {
while (listening) {
        clientSocket = serverSocket.accept();
        new ServiceThread(clientSocket).start();
}
serverSocket.close();
} catch (IOException e) {
System.err.println("Accept failed.");System.exit(1);
}
}
```

Przykład wątku obsługi żądania

Wątek obsługi żądania dla serwera udostępniającego screenshoty.

```
class ServiceThread extends Thread {
private Socket client;

public ServiceThread(Socket client) {this.client = client;
}

@Override
public void run() {

// Tutaj ciało metody run (następny slajd)
}
}
```

```
@Override
public void run() {
OutputStream out;
try {
out = client.getOutputStream();
ByteArrayOutputStream memBuffer = new ByteArrayOutputStream();//buffer in
memory
Robot robot = new Robot();
BufferedImage screenShot = robot
         .createScreenCapture(new Rectangle(Toolkit
         .getDefaultToolkit().getScreenSize()));//creating screenshot
ImageIO.write(screenShot, "JPG", memBuffer);//and writing to buffer
byte buffer[] = memBuffer.toByteArray();
out.write(buffer); //sending buffer to remote client
System.out.println("Screenshot of " +buffer.length
                  + " bytes  sent to: " + client.getInetAddress());
out.close();
client.close();
} catch (IOException e) {
System.err.println("Streaming problem.");System.exit(1);
} catch (AWTException e) {
System.err.println("Problem with image capturing.");System.exit(1);
}
}
```

Datagramy

Istnieją dwa rodzaje konstruktorów klasy *DatagramPacket*:

- do odbioru:

public DatagramPacket(byte[] buf, int offset, int length)

public Datagrampocket(byte[] buf, int length)

buf – bufor do odbioru danych;
offset – początek bufora buf;
length – max. długość pakietu;

- do nadawania:

public DatagramPacket(byte[] buf, int offset, int length, InetAddress address, int port)

public DatagramPacket(byte[] buf, int length, InetAddress address, int port)

buf – bufor z pakietem UDP;
offset – początek bufora buf;
length – długość danych;
address – adres danych;
port – numer portu adresata danych.

Datagram może mieć maksymalnie 65.535 bajtów, w tym 8 bajtów na nagłówek UDP i min 20 bajtów na nagłówek IP. Zastosowanie większych pakietów wiąże się ze wzrostem efektywności transferu, mniejsze datagramy są natomiast bardziej niezawodne.

Nagłówek UDP składa się z :

- **2 bajty** – unsigned int – numer portu źródłowego;
- **2 bajty** – numer portu docelowego;
- **trzecia para bajtów** – rozmiar datagramu razem z nagłówkiem;
- **ostatnia para** – suma kontrolna.

Konstruktory

Obiekty klasy ***DatagramSocket*** służą do nadawania i odbioru danych. Są one dowiązane do lokalnego portu, na którym oczekują na dane. Gniazda używane przez klienta i serwer są identyczne.

Konstruktory:

- ***public DatagramSocket()*** – tworzy gniazdo UDP dowiązane do portu anonimowego;
- ***public DatagramSocket(int port)*** – gniazdo UDP dowiązane do konkretnego portu;
- ***public DatagramSocket(int port, InetAddress iaddr)*** – gniazdo UDP dowiązane do podanego adresu lokalnego.

Konstruktory te zwracają wyjątki:

- ***SocketException*** – gniazdo nie może zostać otwarte, nie jest możliwe dowiązanie do danego portu;
- ***SecurityException*** – gdy istnieje tzw. ***SecurityManager*** i odpowiednie sprawdzenie zabroniło realizacji operacji.

Wysyłanie i odbiór datagramów:

- ***public void send(DatagramPacket p)***
- ***public void receive(DatagramPacket p)*** – odbiera z sieci datagram UDP i konwertuje go do postaci obiektu *DatagramPocket*.

Przykład wysyłania datagramu

```
public static void main(String[] args) {
try {
InetAddress host = InetAddress.getByName("127.0.0.1");
int port = 250;
byte[] b = "Hello".getBytes();
DatagramPacket dp =
        new DatagramPacket(b, b.length, host, port);

DatagramSocket ds = new DatagramSocket();

ds.send(dp);
}
catch (UnknownHostException ex) {System.err.println(ex);}
catch (IOException e) {System.out.println(e);}
}
```

Przykład odbierania datagramu

```
public static void main(String[] args) {
try{
        DatagramSocket ds = new DatagramSocket(250);
        byte buffer[] = new byte[10];
        DatagramPacket dp = new
                DatagramPacket(buffer,buffer.length);
        ds.receive(dp);
        byte[] data = dp.getData();
        String s = new String(data, 0, data.length);
        System.out.println(s);
        System.out.println(dp.getPort());
}
catch(IOException e){System.err.println(e);}
}
```

Gniazda rozgłoszeniowe

Dane wysyłane przez stacje do grupy multicastowej są umieszczone w ***datagramie multicastowym*** – jest to datagram zaadresowany do grupy multicastowej. Wysłane dane multicastowe dochodzą do wszystkich członków grupy w sieci lokalnej. Transmisje mulitcastową obsługuje klasa ***MulticastSocket***.

Konstruktory :

- ***public MulticastSocket()*** – tworzy gniazdo dowiązane do portu anonimowego. Jest użyteczny po stronie klienta, która nie musi znać portu usługi;
- ***public MulticastSocket(int port)*** – tworzy gniazdo, które odbiera dane na ogólnie znanym porcie.

Odbieranie danych multicastowych

Odbieranie danych multicastowych:

- utworzenie obiektu ***MulticastSocket*** za pomocą konstruktora tej klasy;
- dołączenie do grupy multicastowej – metoda ***joinGroup();***
- po dołączeniu do grupy dane są odbierane jak dla zwykłych datagramów UDP;
- opuszczenie grupy multicastowej – metoda ***leaveGroup()***.

Wysyłanie danych na adres multicastowy

Wysyłanie danych na adres multicastowy:

- podobne do wysyłania datagramów UDP;
- można ale nie trzeba dołączać do grupy multicastowej (wystarczy adres rozgłoszeniowy klasy D) ;
- tworzymy obiekt ***DatagramPacket***, umieszczamy dane w pakiecie i adres grupy ;
- podajemy odpowiedni TTL w pakiecie. ***TTL Time to live*** limituje zasięg pakietu np. do sieci lokalnej, kraju. Zawiera maksymalną liczbę router'ów przez która pakiet może przejść zanim nie zostanie skasowany;
- metoda ***send()*** wysyła pakiet.

Wysyłanie danych na adres multicastowy -przykład

```
String msg = "Hello";
InetAddress group = InetAddress.getByName("228.5.6.7");
MulticastSocket s = new MulticastSocket(250);
s.joinGroup(group);
DatagramPacket hi = new DatagramPacket(msg.getBytes(),
msg.length(),group, 250);
s.send(hi);
```

Podsumowanie TCP vs UDP

- TCP posiada oddzielną klasę do nasłuchu (serwera);
- UDP nie stosuje pojęcia gniazda serwera - to samo gniazdo może przyjmować połączenia i wysyłać dane;
- TCP wysyła dane przez strumień skojarzony z gniazdem;
  a UDP nie stosuje strumieni, wysyła pakiety (datagramy);
- Konkretne gniazdo typu DatagramSocket może odbierać dane od wielu niezależnych stacji i nie jest dedykowane jednemu konkretnemu połączeniu,
- Metody strumieni łącza: *read* i *write* (TCP) mogą **blokować** swój wątek, np. przy zapisie write do strumienia wyjściowego gdy zostanie już osiągnięty rozmiar bufora gniazda, a odbiorca nie odbiera danych (domyślnie bufor gniazda ma poj. 8KB, ale klasa Socket pozwala ustawić inny).
- W UDP metoda zapisu **nie blokuje** bieżącego wątku, ale pakiety mogą zostać utracone gdy wystąpi awaria sieci