Seminář Java

IX

Java a sítě

Nutné znalosti

Předpokládané znalosti:

- Základy jazyka Java
- Práce s proudy
- Threads (Vlákna)

Síťová problematika :

- Pouze přehledově
- Vybrané partie pro potřeby této přednášky
- Volitelné kurzy na FITu

OSI model

(OSI - Open System Interconect)

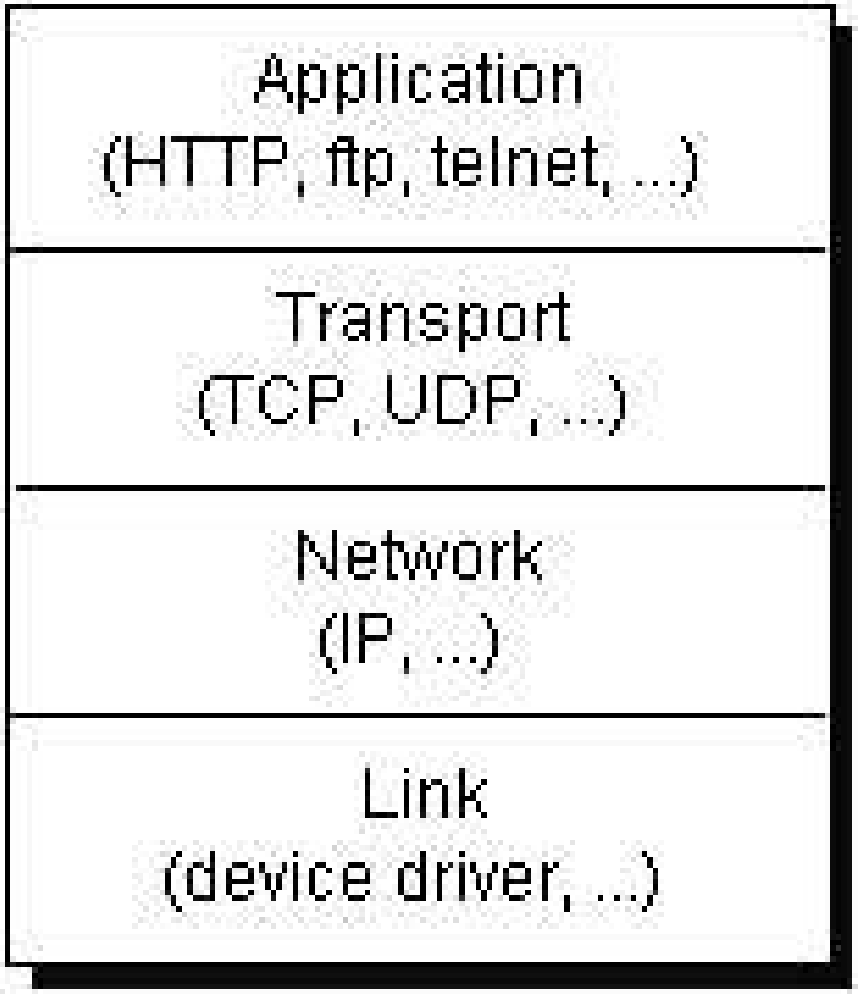
- Standard, který popisuje síťovou komunikaci
- Důvod $\rightarrow$ spojení mezi navzájem si nepříliš podobných počítačů
- Jedná se o doporučení
- Několik vrstev
- Vrstvy jsou na sobě nezávislé
- Vrstva komunikuje pouze s bezprostředně sousedními (?úplnost implementace?)

OSI model II – obrázek

OSI model III – vrstvy

- Fyzická
  - Vlastní fyzické propojení, hardware
- Linková
  - řídí tok dat na přenosovém médiu, pakety → rámce
- Síťová
  - směrování dat, adresování, protokoly (IP, IPX, ICMP)
- Transportní
  - zajišťuje spojení, data → pakety, protokoly (TCP,UDP ,SPX)
- Relační
  - řídí tok mezi aplikačními procesy
- Prezentační
  - vhodná forma pro přenos, transformace dat, komprese, šifrování apod.
- Aplikační
  - služby viditelné uživateli (http, ftp,. . .)

Protokoly IP, TCP, UDP



Protokoly IP, TCP, UDP

- IP (Internet Protocol)
  - Protokol síťové vrstvy (adresování – IP adresa)
  - Zajištění správného pořadí paketů
- TCP (Transmission Control Protocol)
  - Protokol transportní vrstvy
  - Spojově orientovaný, záruka přenosu
  - „telefon"
- UDP (User Datagram Protocol)
  - Protokol transportní vrstvy
  - Nespolehlivý
  - Rychlejší než TCP
  - „pošta"
  - datagramy

Porty

- Obvykle jedno fyzické připojení k síti $\rightarrow$ všechna data tímto připojením
- Na jednom počítači více aplikací $\rightarrow$ pro jakou aplikaci jsou data určena???
- Porty
- *Protokoly TCP a UDP používají porty pro mapování příchozích dat na jednotlivé procesy běžící na počítači*
- přenášená data sebou nesou adresu+port
- port = 16bit číslo (0-65 535)
- 1-1023 vyhrazené pro „známé služby" (http, ftp,. . .)

Co je pro nás důležité

- Čím musí data projít na cestě k adresátovi
- Adresace
  - IP adresa pro určení cíle
  - port pro určení běžícího procesu
- Protokol TCP
  - spojově orientovaný
  - zaručen přenos
  - vyšší režie přenosu
- Protokol UDP
  - nespojově orientovaný
  - není zaručen přenos
  - nezávislé balíčky dat - *datagramy*
  - nižší režie přenosu

Základní balíček – java.net

- Podpora pro komunikaci pomocí TCP a UDP
- Pro TCP
  - *URL*
  - *URLConnection*
  - *Socket*
  - *ServerSocket*
- Pro UDP
  - *DatagramPacket*
  - *DatagramSocket*
  - *MulticastSocket*

URL (Uniform Resource Locator)

- Reference (adresa) na zdroj v internetu
- Třída *java.net.URL*
- URL je složena ze dvou základních komponent : **identifikátor protokolu**, **název zdroje**
- Název zdroje
  - Závisí na použitém protokolu
  - **HostName** – název stroje na kterém je zdroj uložen
  - **FileName** – cesta ke zdroji
  - **PortNumber** – číslo portu
  - **Reference** – reference na pojmenovanou část zdroje

```
http://www.w3.org:80/MarkUp/recommendations
```

URL - konstruktory

- Nejjednodušší je vytvoření URL ze stringu

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

- „Write-once" objekty
- Pro **všechny** konstruktory je nutné odchytit MalformedURLException

Další konstruktory

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

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

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

Práce s URL - relativní odkazy

- Nejjednodušší je vytvoření URL ze stringu

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

- `URL(URL baseURL, String relativeURL)`

- Relativní URL k URL *gamelan*

```
URL gamelanGames = new URL(gamelan, "Gamelan.game.html");
URL gamelanNetwork = new URL(gamelan, "Gamelan.net.html");
```

- Výsledné relativní URL jsou

```
http://www.gamelan.com/pages/Gamelan.game.html
http://www.gamelan.com/pages/Gamelan.net.html
```

Práce s URL - parsování

- Při konstrukci objektu → MalformedURLException
- Několik metod pro získání informací o příslušném URL
  - *getProtocol*, *getHost*
  - *getPort* , *getFile*, *getRef*
- Příklad :

```
URL aURL = new URL("http://java.sun.com:80"
                   +"/docs/books/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());
```

- Ne každá URL má všechny části!

Práce s URL - přímé čtení

- Čtení z *URL*

```
URL yahoo = new URL("http://www.yahoo.com/");
BufferedReader in = new BufferedReader(
                            new InputStreamReader
                                yahoo.openStream()));
String inputLine;
while ((inputLine = in.readLine()) != null)
      System.out.println(inputLine);
in.close();
```

Práce s URL - vytvoření spojení

- Metoda *openConnection* pro připojení
- Vytvoří komunikační linku mezi programem a *URL*
- Úspěšné navázání spojení $\rightarrow$ objekt třídy *URLConnection*
- Čtení dat pomocí *URLConnection* = přímému čtení dat z *URL*
- Zápis dat pomocí *URLConnection*
  - Zápis dat na server (html-formuláře), na straně serveru obvykle bývá cgi skript
    1. Vytvoření *URL*
    2. Otevření spojení (*openConnection*)
    3. Získání (*getOutputStream, ...*)
    4. Zápis do výstupního proudu (např. pomocí *PrintWriter*)
    5. Uzavření výstupního proudu

URLConnection - čtení

```
...
URL myURLConnection = new URL("http://www.seznam.cz/");
URLConnection myURLConnection = myURL.openConnection();
BufferedReader in = new BufferedReader(
        new InputStreamReader(
                myURLConnection.getInputStream()));

String inputLine;

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

URLConnection - zápis

- CGI *http://java.sun.com/cgi-bin/backwards* reverzuje řetězec
- Formát vstupu *string="řetězec"*

```
...
String stringToReverse = "PrikladIJA";

URL url = new URL("http://java.sun.com/cgi-bin/backwards");
URLConnection connection = url.openConnection();
connection.setDoOutput(true);

PrintWriter out = new PrintWriter(
        connection.getOutputStream());
out.println("string=" + stringToReverse);
out.close();
...
```

Rekapitulace

- *URL* = Reference (adresa) na zdroj v internetu
- z *URL* lze získat *URLConnection*
- Pomocí *URLConnection* je výhodné např. pro čtení obsahu www stánek
- Pomocí *URLConnection* lze komunikovat s cgi skripty na straně serveru

Sokety v Jave

- Sokety pro komunikaci klient–server
- Základní operace
  - Vytvoření spojení
  - Zaslání dat
  - Příjem dat
  - Uzavření spojení
- Pouze jeden host
- Klienti se připojují na server, zasílají a čtou zprávy
- Server čeká na spojení
- $\longrightarrow$ dva typy
  - *Socket* klient, aktivní (vytváří spojení)
  - *ServerSocket* server, pasivní „naslouchá"

Hosts, Ports and Sockets

- **Host** - počítač jedinečně identifikován názvem (hostname) a/nebo IP adresou
  - hostname: jméno ve formě Stringu (např. www.fit.vutbr.cz)
  - IP adresa: 4 (6 pro IPv6) byty (např. 147.229.9.22)
  - utility *ipconfig*,*winipcfg* pro zjištění adresy

- **Port** - číslo specifikující typ připojení
  - Například port 80 je rezervovaný pro HTTP, port 23 pro telnet
  - server může „naslouchat" na více portech

- **Soket** - obaluje komunikační proces

Třída Socket - *konstruktory*

- Několik konstruktoru (API)
- **Socket(String host, int port)**
- **Socket(InetAddress address, int port)**

- Třída *InetAddress* (viz. API)
  - Reprezentace adresy Internet protokolu (IP)
  - 32-bit i 128-bit (IPv6)
  - unicast i multicast
  - statické metody
    - **InetAddress.getByName(String host)**
    - **InetAddress.getByAddress(byte[] addr)**
  - UnknownHostException, SecurityException

Třída Socket - *důležité metody*

- API (*isConnected*, *getTcpNoDelay*, . . .)
- **InputStream getInputStream()**
- **OutputStream getOutputStream()**
- **void close()**
- Pomocí získaných proudů lze zasílat/získávat data
  - Čtení z *InputStream* = příjem dat z hosta
  - Zápis do *OutputStream* = zasílání dat do hosta
  - Práce s proudy v předchozích přednáškách (*BufferedReader*,*PrintWriter*)

Třída Socket - *postup*

1. Vytvoření objektu
   - Specifikace hosta nebo adresy, specifikace portu
2. Získání I/O proudů
   - *getInputStream()*, *getOutputStream()*
   - „Vylepšení" typu proudu – *BufferedReader*,*PrintWriter*
3. Komunikace pomocí proudů
4. Uzavření spojení – *close()*

Třída Socket - *ukázka*

Pouze ukázka (chybí odchytávání výjimek apod.)

```
// Pokus o připojení na ,,myserver" na port 8888, čeká
Socket s = new Socket("myserver", 8888);

// Získání proudů pro komunikaci
BufferedReader in = new BufferedReader(
     new InputStreamReader(s.getInputStream()));
PrintStream out = new PrintStream(s.getOutputStream());

// Komunikace
out.print("Hello");
out.flush();
String line = in.readLine();
System.out.println(line);

in.close();
out.close();
s.close();
```

Třída Socket - *rekapitulace*

1. Pro komunikaci klient–server
2. Implementace strany klienta
3. Definice cíle v konstruktoru – host+port
4. Komunikace přes dvojici proudů

Třída ServerSocket

- Špatné označení
  - Není potomek třídy *Socket*! Nemá stejné metody
  - Vytváří objekty třídy *Socket* jako odpověď na klientovo připojení
  - Lepší název by byl „ConnectionListener"

- Konstruktoru předáváme port, na kterém má server naslouchat (pozn. port 0)

  ServerSocket(int port);
  ServerSocket(int port, int backlog, InetAddress bindAddr)

- Metoda **accept()** pro získání objektu třídy *Socket*
  - Thread, který zavolá **accept()** čeká dokud se nepřipojí klient
  - Pokračuje ve své činnosti, po připojení klienta

Třída ServerSocket - *důležité metody*

- API
- **ServerSocket(int port)**
- **ServerSocket(int port, int backlog, InetAddress bindAddr)**
- int getLocalPort()
- InetAddress getInetAddress()
- **Socket accept()**
- void close()

Třída ServerSocket - *postup*

1. Vytvoření objektu
   - specifikace portu na kterém bude server naslouchat
2. Volání metody **accept()**
   - Blokuje thread
   - Po připojení klienta vrací objekt třídy *Socket*
3. Získání I/O proudů z vráceného objektu
   - *getInputStream()*, *getOutputStream()*
   - „Vylepšení" typu proudu – *BufferedReader*,*PrintWriter*
4. Komunikace pomocí proudů s klientem
5. Uzavření spojení
6. Opakujeme – volání metody **accept()**

Třída ServerSocket - *ukázka*

Pouze ukázka (chybí odchytávání výjimek apod.), kolik obslouží klientů?

```
ServerSocket ss = new ServerSocket(8888);
Socket s = ss.accept();

// Získání proudů pro komunikaci
BufferedReader in = new BufferedReader(
     new InputStreamReader(s.getInputStream()));
PrintStream out = new PrintStream(s.getOutputStream());

String line = in.readLine();
System.out.println(line);
out.print("Hi");
out.flush();

in.close();
out.close();
s.close();
ss.close();
```

Třída ServerSocket - více klientů I.

- Ukázka je velice primitivní
- Obslouží pouze jednoho klienta a skončí
- → nekonečná smyčka

```
ServerSocket ss = new ServerSocket(8888);
while (true) {
    Socket s = ss.accept();
    metoda_pro_obsluhu_jednoho_klienta(s);
}
ss.close();
```

- Server během svého běhu obslouží více klientů
- Je toto řešení dobře napsané?? Co může být problém??

Třída ServerSocket - více klientů II.

- Server během svého běhu obslouží více klientů
- Pokud ale komunikuje s klientem nemůže akceptovat další připojení
- Řešení tohoto problému = využití threadů

- Jak na to?
  1. Hlavní thread vytvoří *ServerSocket*, volá metodu *accept()*
  2. Pro nově připojeného klienta vytvoří nový thread
  3. Nový thread obslouží komunikaci s klientem
  4. Hlavní thread souběžně opět volá volá metodu *accept()*

Třída ServerSocket - více klientů III.

- → nekonečná smyčka+nový thread

```
ServerSocket ss = new ServerSocket(8888);
while (true) {
    Socket s = ss.accept();
    vytvořit nový thread(s) a spustit jej;
}
ss.close();
```

- Server potom může obsluhovat několik klientů zároveň
- Čeká na další připojení
- Implementace threadu → další slajd

Třída ServerSocket - thread pro komunikaci

```
public class MyClientThread extends Thread {
  private Socket socket = null;

  public MyClientThread(Socket socket) {
    super("MyClientThread");
    this.socket = socket;
  }

  public void run() {
    try {
      /** Získání streamu z objektu socket **/
      /** Komunikace **/
      /** Konec komunikace (uzavřeni apod.) **/
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Ukázka implementace

- Jednoduchý server, který bude čekat na připojení na portu 8080
- Klientovy vrátí html (bez http hlaviček – neúplná implementace)
- Odpověď bude obsahovat čas na serveru v milisekundách
- Obslouží 5 klientů a skončí
- Může obsluhovat klienty souběžně

Implementace serveru - DummyWebServer

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

public class DummyWebServer {

  public static void main(String[] args) {
    try {
      ServerSocket ss = new ServerSocket(8080);
      int maxConnections = 0;
      while (maxConnections++ <5) {
        Socket s = ss.accept();
        (new DummyWebServerClientThread(s)).start();
      }
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }
}
```

Implementace serveru - DummyWebServer

```
class DummyWebServerClientThread extends Thread {
  protected Socket s = null;

  public DummyWebServerClientThread(Socket s) {
    super("DummyWebServerClientThread");
    this.s = s;
  }

  public void run() {
    try {
      PrintStream out = new PrintStream(s.getOutputStream());
      out.print("<html>");
      out.print("<head><title>DummyWebServe</title></head>");
      out.print("<body><h1>DummyWebServer response</h1>");
      out.print("<h4>Server time : "+System.currentTimeMillis
      ...
    } catch (Exception ex) {
      ...
```

Implementace serveru - výsledek

Třída ServerSocket - *rekapitulace*

1. Pro komunikaci klient–server
2. Implementace strany serveru
3. Tato třída není potomkem třídy *Socket*, jiné metody
4. Čeká na spojení (metoda *accept*), při spojení klienta vytvoří objekt třídy *Socket*
5. Nejčastěji „nekonečná smyčka"
6. Je dobré využít multithreading
7. Otázky?

Základní balíček – java.net

- Podpora pro komunikaci pomocí TCP a UDP
- Pro TCP
  - *URL*
  - *URLConnection*
  - *Socket*
  - *ServerSocket*
- Pro UDP
  - *DatagramPacket*
  - *DatagramSocket*
  - *MulticastSocket*

Protokol UDP - opakování

- Protokol transportní vrstvy
- Nespolehlivý
- Rychlejší než TCP
- „pošta"
- datagramy

- Co je datagram?
  - Je nezávislý balíček dat, jehož doručení, čas doručení a obsah není zaručen

Třída DatagramPacket

- Reprezentace datagramu pro UDP protokol

- Konstruktory (pouze některé, další API)
  - **DatagramPacket(byte[] buf, int length)** – pro příjem datagramu
  - **DatagramPacket(byte[] buf, int length, InetAddress address, int port)** – pro posílání na specifický port a hosta

- Metody
  - **InetAddress getAddress(), int getPort()**
  - **byte[] getData()**
  - **int getLength()**
  - setData(byte[] buf), setLength(int length)

Třída DatagramSocket

- Třída reprezentuje soket pro příjem a posílání datagramů
- Každý datagram je individuálně adresován a routován
  - Co to znamená pro vlastní komunikaci?

- Konstruktory (pouze některé, další API)
  - **DatagramSocket(int port)**
  - **DatagramSocket(int port, InetAddress laddr)**

- Metody
  - **send(DatagramPacket p)**
  - **receive(DatagramPacket p)**
  - isClosed(), close()

Použití - ukázka serveru a klienta

- Server
  1. Vytvoří socket na portu 4444 a čeká na datagram
  2. Příjme datagram, který obsahuje jméno klienta
  3. Odpoví „Hello ", sys. čas a číslo klienta
  4. Obslouží max. 5 klientů a pak skončí

- Klient
  1. Pošle datagram se svým jménem
  2. Čeká na odpověď
  3. Vypíše odpověď
  4. Konec

- Poznámka: Je „dobré" spustit server jako první

Implementace UDP serveru

```
DatagramSocket ds = new DatagramSocket(4444);
int responsesCount = 0;
while (responsesCount++ < 5 ) {
    byte[] buff = new byte[256];
    DatagramPacket packet =
        new DatagramPacket(buff, buff.length);
    ds.receive(packet);
    String clientName =
        new String(packet.getData(),0, packet.getLength());

.... Operace s řetězci, výsledek bude v String response ...

    buff = response.getBytes();
    InetAddress address = packet.getAddress();
    int port = packet.getPort();
    packet = new DatagramPacket(buff, buff.length,
                                address, port);
    ds.send(packet);
}
```

Implementace UDP klienta

```
byte[] buff = "Bobik".getBytes();

InetAddress address = InetAddress.getByName("localhost");
DatagramPacket packet =
        new DatagramPacket(buff, buff.length, address, 4444);
DatagramSocket ds = new DatagramSocket();
ds.send(packet);

buff = new byte[256];
packet = new DatagramPacket(buff, buff.length);
ds.receive(packet);

String message =
         new String(packet.getData(), 0, packet.getLength());
System.out.println("Received message : "+message);

ds.close();
```

Rekapitulace k UDP protokolu

1. Pro komunikaci klient–server
2. *DatagramPacket* – reprezentace datagramu
3. *DatagramSocket* – soket pro příjem/posílání datagramu
4. Není zaručeno doručení datagramu, pořadí apod.
5. Nižší režie než TCP
6. Nejčastěji „nekonečná smyčka"
7. Multithreading obdobně jako u TCP
8. Otázky?

Vysílání UDP datagramů více klientům

- Server
  1. Nečeká na datagram od klienta
  2. Datagramy vysílá na adresu (viz. třídy IP)
     InetAddress group = InetAddress.getByName("230.0.0.1");

- Klient
  1. Využívá *MulticastSocket*
  2. Skupina pro příjem (adresa na kterou vysílá server)
  3. Přihlášení do skupiny : *socket.joinGroup(group);*
  4. Příjem dat, zpracování dat
  5. Odhlášení ze skupiny : *socket.leaveGroup(group);*
  6. Uzavření multicastsoketu

Rekapitulace – co jsme se naučili?

- Stahovat obsahy www stánek (*URL*, *URLConnection*)
- Komunikovat s cgi skripty na straně serveru (*URLConnection*)

- Komunikace klient–server
- Spojově orientovaná komunikace (*Socket*, *ServerSocket*)
- Datagramově orientovaná komunikace (*DatagramPacket*, *DatagramSocket*)
- Vysílání více klientům současně (multicasting) (*MulticastSocket*)

- Otázky?

- Posuneme se „výše" – RMI

Některé vyšší technologie

- Nejnižší úroveň : Berkeley sockets
- Střední úroveň : Java networking library
  - Java skrývá detaily
  - Abstrakce
- Vysoká úroveň: RMI, COBRA, apod.
  - Protokol komunikace je oddělen od aplikace

RMI

- Co je RMI?

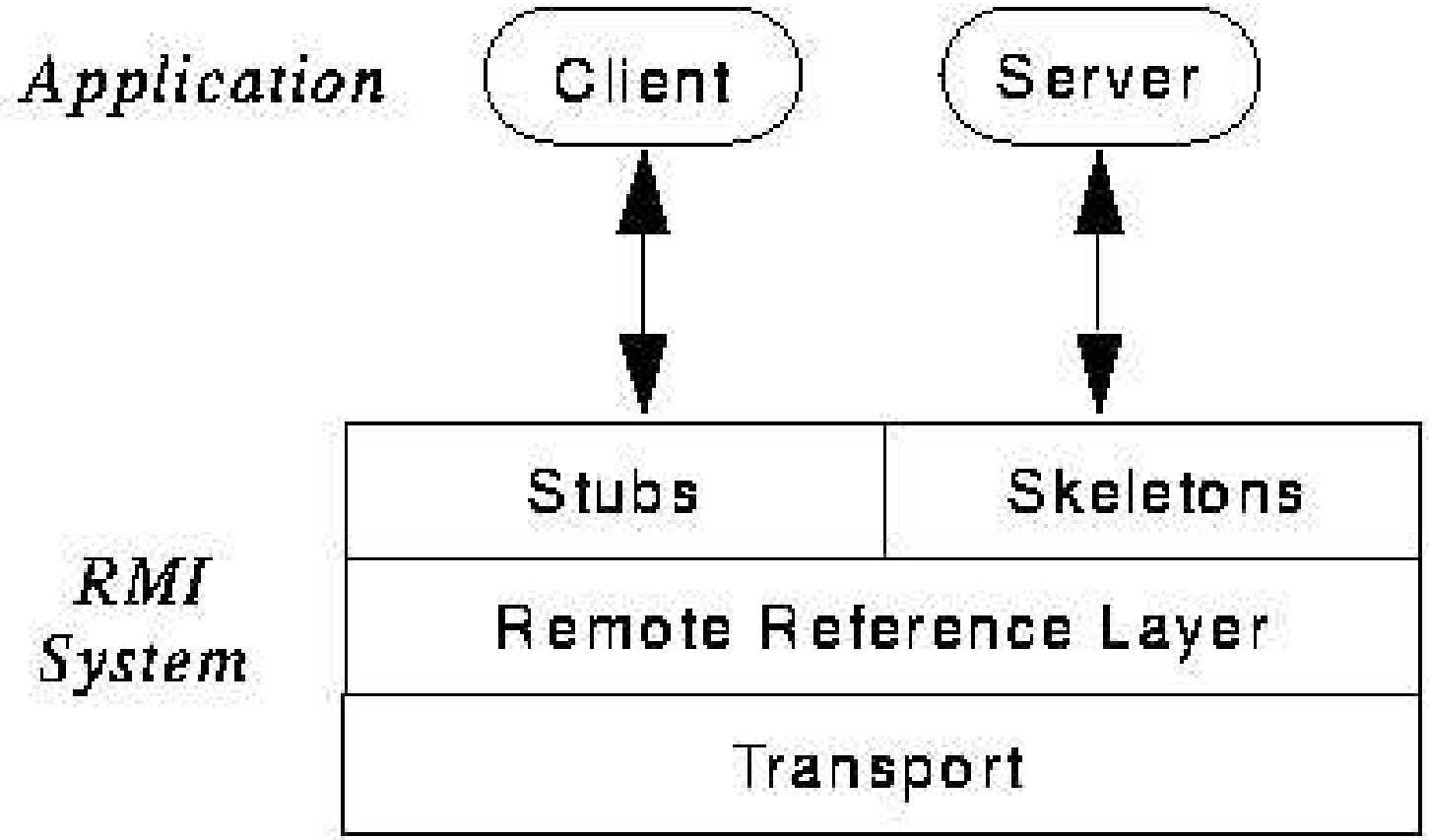
  RMI je systém, který umožňuje objektu běžícím na jednom JVM volat metodu objektu, který běží na jiném JVM

- Systém se skládá z klienta a serveru :
  - Server = vzdálený objekt, přístupné reference, čeká
  - Klient = reference na vzdálený objekt, volání metod vzdáleného objektu
- RMI je mechanismus ve kterém server a klient komunikují a předávají si data

Vzdálené rozhraní, objekty a metody

- Jako ostatní aplikace v Jave – rozhraní(def.) a třídy(impl.)
- Objekty s metodami, které mohou být volány mezi JVM = *vzdálené objekty*
- Vzdálené objekty = implementují vzdálené rozhraní
  - Vzdálené rozhraní potomek *java.rmi.Remote*
  - Každá metoda rozhraní – *java.rmi.RemoteException*

Struktura RMI systému



Stubs

- Raději než lokální kopii implementace RMI předá *stub* (lokální reprezentace vzdáleného objektu)
- *stubs* implementují všechna rozhraní poskytovaná implementací vzdáleného objektu
- Odpovědné za
  1. Inicializace a volání vzdáleného objektu
  2. Serializace argumentů
  3. Informování o začátku volání
  4. Deserializace výsledků či výjimek
  5. Informování o dokončení volání

Skeleton

- Entita na straně serveru
- Směrování na příslušné implementace vzdálených objektů
- Odpovědné za
  1. Deserializace argumentů
  2. Volání metod
  3. Serializace výsledků či výjimek

Příklad

- Přiklad převzat z

  http://java.sun.com/docs/books/tutorial/rmi/designing.html

- Detaily, kompilace, apod.
- Výpočet čísla π

Příklad - definice vzdáleného rozhraní

- Rozhranní definuje metody, které lze volat vzdáleně

```
package compute;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Compute extends Remote {
       Object executeTask(Task t) throws RemoteException;
}
```

Příklad - definice typu pro argument

- Rozhraní definuje typ *Task*, který je předán jako argument pro vzdálené volání

```
package compute;

import java.io.Serializable;

public interface Task extends Serializable {
        Object execute();
}
```

Příklad - implementace serveru I.

```
package engine;

import java.rmi.*;
import java.rmi.server.*;
import compute.*;

public class ComputeEngine extends UnicastRemoteObject
                           implements Compute
{
    public ComputeEngine() throws RemoteException {
        super();
    }

    public Object executeTask(Task t) {
        return t.execute();
    }

    << METODA MAIN - další slajd >>
}
```

Příklad - implementace serveru II.

```
public static voverbatimid main(String[] args) {
  if (System.getSecurityManager() == null) {
     System.setSecurityManager(new RMISecurityManager());
  }
  String name = "//host/Compute";
  try {
     Compute engine = new ComputeEngine();
     Naming.rebind(name, engine);
     System.out.println("ComputeEngine bound");
  } catch (Exception e) {
     e.printStackTrace();
  }
}
```

Příklad - implementace klienta

```
package client;
import java.rmi.*;
import java.math.*;
import compute.*;

public class ComputePi {
    public static void main(String args[]) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(
                new RMISecurityManager());
        }
        try {
            String name = "//" + args[0] + "/Compute";
            Compute comp = (Compute) Naming.lookup(name);
            Pi task = new Pi(Integer.parseInt(args[1]));
            BigDecimal pi = (BigDecimal)
                comp.executeTask(task);
            System.out.println(pi);
        } catch (Exception e) {
```

Rekapitulace RMI, co dále?

- RMI umožňuje volat metody objektů, které jsou spuštěny na jiném JVM
- Jiné JVM může být na jiném počítači v sítí
- Distribuované aplikace
- Omezeno na Java–Java

Odstranění omezení Java-Java $\rightarrow$ CORBA

- Framework pro distribuované objekty
- Nezávislost na jazyce
- Vhodné pro velmi velké projekty

Konec dnešní slideshow

- Otázky?