Sokety a síťování

Jiří Vokřínek

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

Přednáška 9

B0B36PJV – Programování v JAVA

Část 1 – Síťování

Síťování

- Způsoby a modely komunikace
- Síťové modely a Internet
- Transportní a aplikační protokoly

Síťová API

- Soket

Síťování v Javě

- Třídy UDP a TCP soketů

Část 2 – Příklad – jednoduchý klient a server

Příklad – jednoduchý klient a server

Část 3 – Příklad aplikace – Boss/Worker

Příklad vláknové aplikace – Server

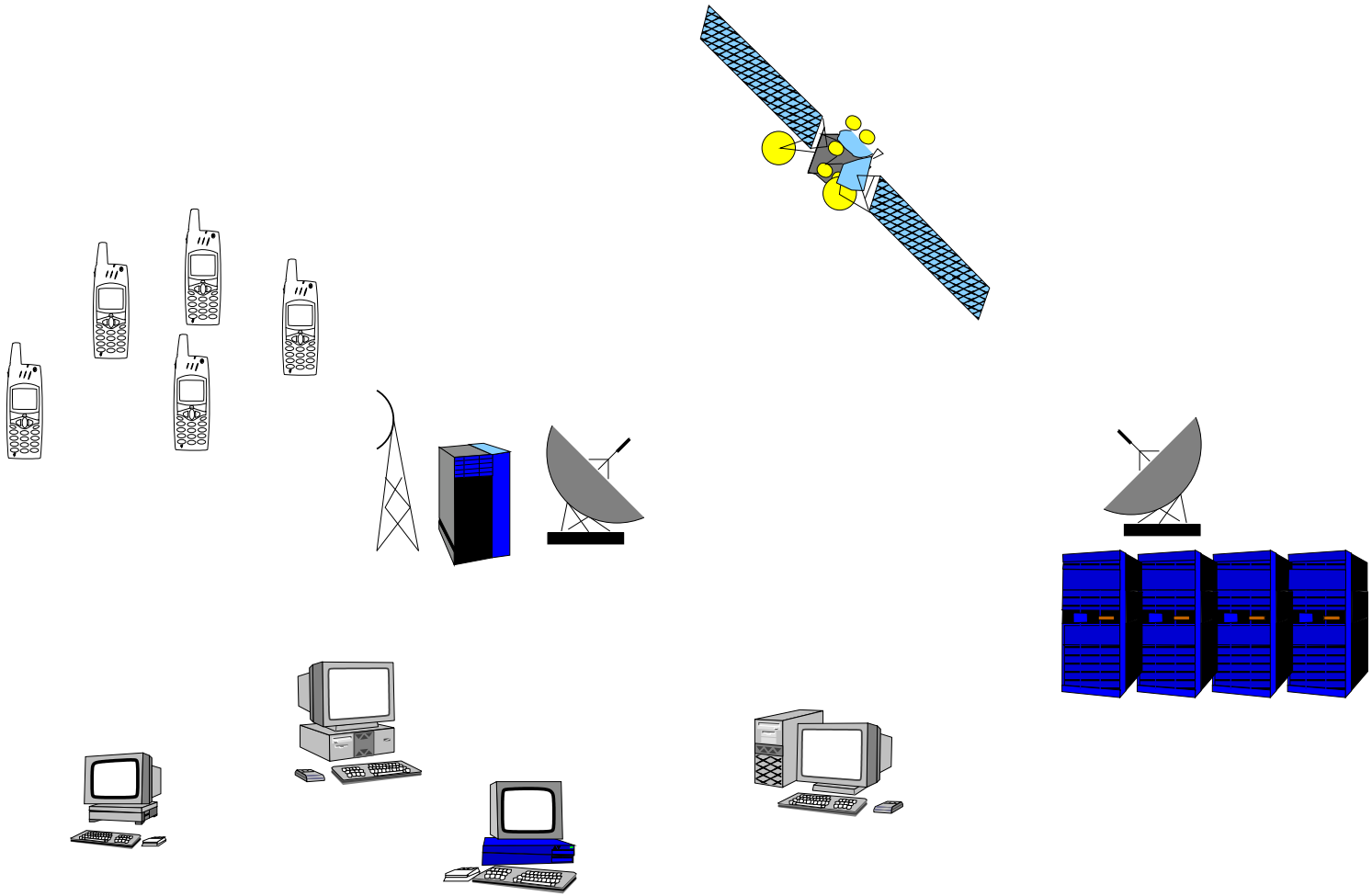
Příklad vláknové aplikace – Klient

Spuštění jiného programu z procesu

Část I

Síťování

Co je síťování?



Zdroje


Jiří Peterka,
`http://www.earchiv.cz/i_prednasky.php3`


RFC - Request for Comments,
série poznámek o Internetu.


Martin Majer,
`http://www.root.cz/clanky/sitovani-v-jave-uvod/`


W. Richard Stevens,
UNIX Network Programming.
Prentice Hall.


W. Richard Stevens and Stephen A. Rago,
Advanced Programming in the UNIX Environment.
Addison Wesley.

Motivace

Síťování je z pohledu vývoje aplikace (programování) technická realizace komunikace vzdálených výpočetních systémů.

Komunikace je přirozenou součástí distribuovaných aplikací.

- Aplikace nabízející služby uživateli (např. webový server, databáze).
- Uživatelské aplikace zprostředkovávající uživateli přístup ke službám (např. webový prohlížeč).
- Sdílení zdrojů - distribuované výpočetní systémy.
- Sběr dat (data acquisition) - zpracování dat z mnoha měřících míst, senzorické sítě.
- Distribuované řízení - například kooperující skupina mobilních robotů.

Vývoj aplikace

- Při vývoji aplikace využíváme zpravidla nějakého rozhraní (API) pro realizaci komunikačního spojení se vzdáleným systémem.
- Z tohoto pohledu síťové spojení slouží ke čtení a zápisu posloupnosti bytů.
- Podle typu aplikace je rozhodující:
  - spolehlivost přenosu dat,
  - přenosová rychlost,
  - zpoždění (latence) přenosu,
  - způsob předávání dat,
- Vlastnosti souvisejí s konkrétní realizací síťového spojení a je nutné je respektovat.

Komunikace

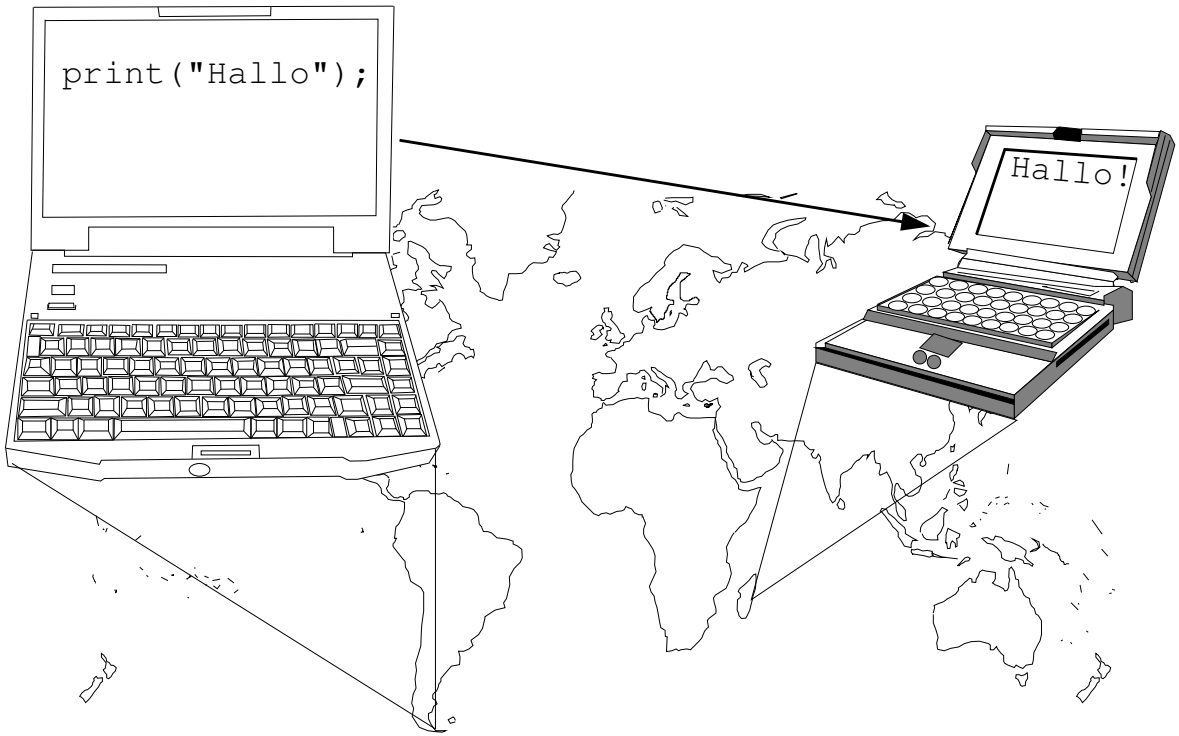
- Komunikace slouží k přenosu informace.
- Přenos informace se děje výměnou zpráv.
- Mechanismus výměny zpráv musí mít definovaná pravidla.
- Typicky lze definovat:
  - zahájení komunikace,
  - předání zprávy,
  - reakce na zprávu,
  - ukončení komunikace.

Protokol

- Způsob komunikace definuje komunikační protokol.
- Protokol definuje:
  - formát zpráv,
  - pořadí výměny zpráv,
  - syntaxi zpráv,
  - sémantiku zpráv,
  - chování při přijmu a vyslání zprávy.

Přenos bitů/bytů

Z uživatelského hlediska jde o přenos obsahu sdělení.



Přenos však vyžaduje další informace související s přenosovou cestou. „Výsledná velikost přenášených dat je vyšší."

Počet účastníku komunikace

Komunikaci může rozdělit podle počtu účastníků.

- Dvou bodové (point-to-point) - jeden přijímací bod, jeden vysílací bod.
  - duplexní přenos - komunikace může probíhat oběma směry současně.
- Vícesměrové - více přijímacích stran, jedna vysílací. Vhodné pro multimediální aplikace, přenos videa, tele-konference.
  - všesměrové - broadcast,
  - vícesměrové - multicast.

My se budeme zabývat převážně dvou bodovou komunikací.

Modely komunikace

Typická síťová aplikace se skládá ze dvou částí:

- Server - reprezentuje služby.
- Klient - reprezentuje poptávku po službě.

Modely komunikace jsou:

- klient/server - klient žádá o službu server. *Webový server, poštovní server, Instant Messaging (IM), vzdálená sezení.*
- peer-to-peer (P2P) - každý účastník vystupuje jako klient i jako server. *Služby sdílení souborů, bittorrent, ...*

Způsoby komunikace

Kritéria dělení komunikace.

- Podle způsobu navazování spojení:
  - spojovaná komunikace,
  - nespojovaná komunikace.
- Podle způsobu přenosu data:
  - proudový přenos,
  - blokový přenos.
- Podle kvality přenosu a garance kvality přenosu:
  - spolehlivý,
  - nespolehlivý,
  - s garantovanou kvalitou,
  - bez řízení kvality.

Spojovaná komunikace

Spojovaná komunikace (Connection oriented).
Skládá se ze tří kroků:

1. Obě strany nejdříve navazují spojení.

   Obě strany potvrdí zájem o komunikaci případně upřesní parametry vzájemné komunikace.

2. Vlastní výměna sdělení.
3. Ukončení spojení.

Vlastnosti spojované komunikace

- Součástí komunikace je přechod stavů účastníků.
- Přechody mezi stavy musí být koordinované.

  „Obě strany musí být v kompatibilním stavu, aby se domluvily."
- Musí být ošetřovány nestandardní situace.

  Například rozpad spojení.
- Při přenosu zpráv je zachováno pořadí vysílaných zpráv.

  Přijímací strana obdrží zprávy ve stejném pořadí v jakém je poslala vysílací strana.

Nespojovaná komunikace

- Komunikující strany nenavazují spojení.

  Nedochází k ověřování existence druhé strany.
- Komunikace probíhá zasíláním samostatných zpráv (datagramů).

  Adresování zprávy
- Není nutné komunikaci ukončovat.

Vlastnosti:

- Komunikace je bezstavová.
- Zprávy jsou přenášeny v samostatném bloku dat (datagramu), které jsou samostatně přenášeny.
- Není zaručené pořadí zpráv.

Způsoby přenosu

Proudový přenos (stream)

- Data jsou přenášena po bytech (bitech).
- Data nemusí být přenášena po větších blocích.
- Předpokládá spojovaný typ komunikace.
- Data nejsou adresována.

Blokový přenos

- Data jsou přenášena po blocích.
- Blok je přenesen jako celek.
- Spojovaný i nespojovaný typ komunikace.
- Data jsou adresována podle typu komunikace.

Složky síťování

Propojování systémů se skládá z:

1. přenosového média,
2. řízení přístupu k přenosovému médiu,
3. rozlišení (adresace) fyzického prostředku připojení,
4. přenosových pravidel (*jak jsou data přenášena*),
5. komunikačních pravidel (*jak je definované spojení*),
6. aplikačního rozhraní,
7. aplikačního protokolu.

Síťové modely

- Modely jsou vícevrstvé.
- Vrstva definuje vlastnosti příslušné části komunikace.
- Vrstvený model je abstrakcí jednotlivých stupňů realizace sítě.
- Modely:
  - ISO-OSI - 7-vrstvý obecný model.
  - TCP/IP - 4-vrstvý model, nejpoužívanější v rámci Internetu, základem je jednoduchý protokol IP.

„Pokud víte co děláte, 4 vrstvy stačí, pokud ne, ani 7 vrstev vám nepomůže.”

ISO-OSI síťový model

1. Fyzická vrstva - médium (kabel, vzduch), jejím úkolem je přenos bitů po vedení.
2. Linková vrstva - pravidla pro časový multiplex paketů.
3. Síťová vrstva - pravidla pro HW adresování.
4. Transportní vrstva - pravidla pro zasílání paketů.
5. Komunikační vrstva - pravidla pro komunikační spojení mezi dvěma počítači.
6. Prezentační vrstva - síťové API

   („*a mnohem víc, ”*).
7. Aplikační vrstva.

TCP/IP síťový model

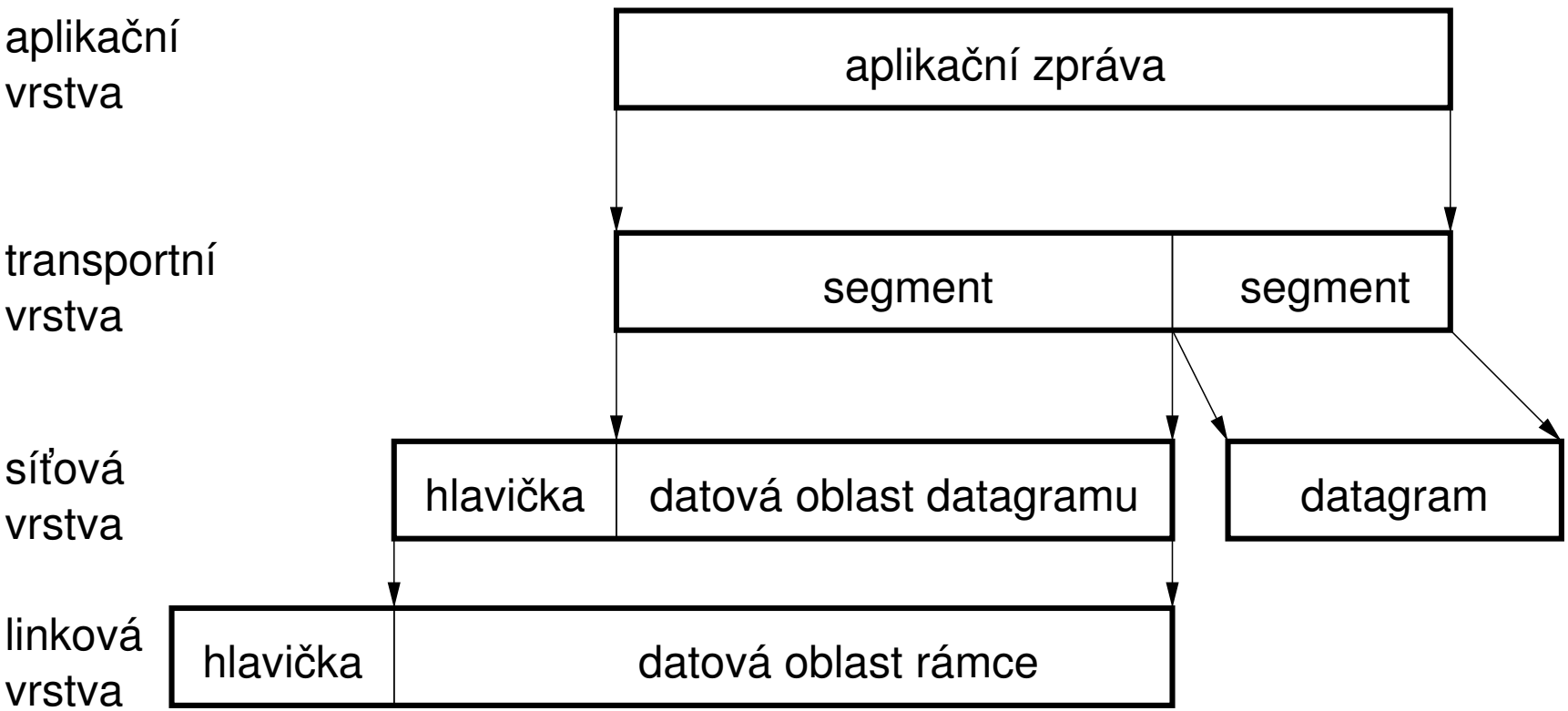
- Linková vrstva - přenos bitů.
- Síťová vrstva (Internet nebo též IP vrstva) - cesta datagramů z jednoho hosta na jiný.
- Transportní vrstva - TCP zprávy transportní vrstvy.
- Aplikační vrstva - síťová aplikace.

Jednotným prvek modelu TCP/IP je přenosový protokol IP, který má všude (linková vrstva) stejné vlastnosti.

Přenosový protokol IP předepisuje jednotný způsob adresování:

- inet - 32 bitová adresa,
- inet6 - 128 bitová adresa.

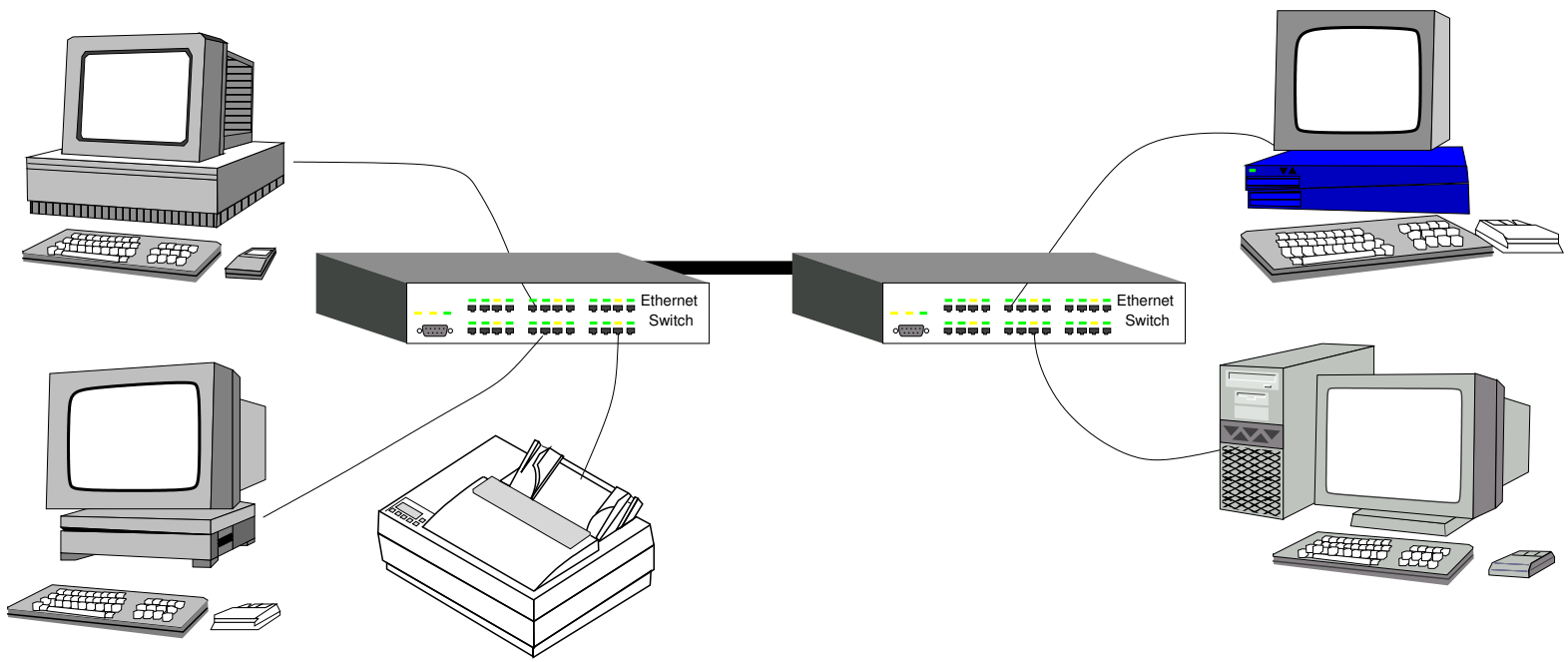
Zapouzdřování datových jednotek



Síť TCP/IP (Internet)

Složena ze soustavy dílčích sítí, které jsou vzájemně propojeny. Rozlišujeme dva typu uzlů:

- Hostitelský počítač (host) - koncový uzel sítě, má svoji adresu.
- Směrovač (router) - propojuje nejméně dvě IP sítě.



Adresování v TCP/IP a transportní protokoly

- Vrstva IP definuje síťovou adresu síťového rozhraní (host).
- Z pohledu více procesů na hostitelském počítači je výhodné zavést další rozlišovací údaj, (*v rámci transportního protokolu*).
  Nejznámější transportní protokoly jsou
  - UDP - User Datagram Protocol,
  - TCP - Transport Control Protocol.

  Oba protokoly používají číslo portu.
- Adresa je složena z
  1. IP adresy (host address), 32bitů nebo 128bitů,
  2. čísla portu (16bitů).

  Některé IP adresy mají vyhrazené použití, např. 127.0.0.1 - localhost.
  Určité aplikace používají „standardních" portů, 80 - www, 21 - ftp, 23 - telnet, 22 - ssh.

UDP

User Datagram Protocol

- Nespojovaný a nespolehlivý způsob komunikace.
- Blokový přenos zpráv (datagramů).
- Rozšiřuje přenosový protokol IP (RFC 791) síťové vrstvy o číslo portu.

  „Přidává pouze tenkou vrstvu nad IP."
- Přenos každého datagramu je nezávislý.
- RFC 768.

UDP vlastnosti

- Není nutné navazovat spojení, které může být časové náročné

  zvyšuje zpoždění
- Hlavička datagramu je malá.
- Žádná kontrola propustnosti, maximální rychlost přenosu.
- UDP datagram je přenesen najednou.
- Hodí se pro přenos dat, které mohou obsahovat výpadky (*nedoručení zprávy*), ale vyžadující minimalizaci časových zpoždění.

  např. data měřená opakovaně
- Spolehlivost lze řešit v aplikační vrstvě.

  (opakovaný přenos zpráv)

TCP

Transmission Control Protocol

- Spojovaný způsob komunikace - dochází k výměně řídicích zpráv, (*vyžaduje model klient/server pro navázání spojení*).
- Dvou bodové obousměrné spojení.
- Proudový přenos zpráv - zachování pořadí. *Pipelined přenos s vyrovnávací pamětí, zapsaná data nemusí být ihned poslána.*
- Tok dat je řízen - nemůže dojít k zahlcení přijímací strany, (*řízení toku je realizováno potvrzení příjetí*).
- Spolehlivost přenosu je zajištěna potvrzováním přijatých dat. *Řídicí zpráva ACK potvrzující příjem zprávy (packetu).*
- Spojení je udržováno kontrolními zprávami i v případě, že nedochází k přenosu aplikačních dat (*detekce rozpadu spojení*).
- RFC - 793, 1323, 2018, 2581.

TCP - Potvrzování přijatých dat

- Potvrzení přijetí každého vyslaného paketu musí být přijato v daném časovém intervalu (TCP timeout).
- Data jsou posílána za sebou, současně se čeká na více potvrzení.
- Doba přenosu zprávy (paketu) tzv. Round Trip Time RTT (RTT) - doba přenesení vlastních dat a potvrzovací zprávy.
  - Závisí na konkrétním zatížení sítě.
  - TCP timeout může být příliš krátký pro konkrétní RTT.

  lze nastavit
- Využívá se kumulativních *ack* potvrzení více přijatých paketů najednou.
- V případě obousměrné komunikace jsou *ack* potvrzení přenášena s daty ve společném paketu.

TCP řízení toku proudu

- Vysílací strana využívá přijatých ACK k řízení toku dat, aby nedocházelo k přehlcovaní přijímací strany (sítě).
  - Problém řízení toku se uplatňuje při pomalém zpracování na přijímací straně.
  - V případě nedostatečné kapacity sítě, dochází k zahlcování sítě, vedoucí k delším zpožděním a případně ke ztrátám paketů.

    konečná velikost vyrovnávací paměti směrovačů
- Velikost okénka pro maximální počet přenášených se dynamicky mění, podle zatížení koncových bodů spojení.

Poznámka - spolehlivost přenosu:

garantovaná, ale ne zas tak moc, větší stupeň garance lze dosáhnout dalšími aplikačními úrovněmi.

HTTP

- HTTP - HyperText Transfer Protocol pro webové aplikace.
- Aplikační protokol nad přenosovým protokolem TCP z rodiny protokolů TCP/IP.
- Klient/Server model komunikace:
  - client - webový prohlížeč, žádá o webové objekty, které chce zobrazit.
  - server - webový server, posílá webové objekty jako odpověď na žádost.
- Bezstavová komunikace, request/response model.
- HTTP 1.0: RFC 1945, HTTP 1.1:RFC 2068.

HTTP - request/response

Postup vyřízení žádosti o webový objekt (HTML stránka):

1. Klient iniciuje TCP spojení k serveru.
2. Server přijímá TCP spojení (spojení je navázáno).
3. Klient posílá HTTP zprávu s žádostí o webový objekt.
4. Server posílá HTTP zprávu s webovým objektem.
5. Klient přijímá HTTP zprávu s webovým objektem.
6. Server uzavírá spojení.

Navazování spojení (TCP handshaking) řeší transportní vrstva, při navazování zpravidla dojde k výměně několika zpráv.

TCP protokol je potvrzovaný, server uzavírá spojení, až když klient potvrdí příjem zprávy. Potvrzení opět řeší transportní vrstva.

HTTP příklad

```
telnet cs.felk.cvut.cz 80
Trying 2001:718:2:1611::16...
Trying 147.32.80.16...
Connected to cs.felk.cvut.cz.
Escape character is '^]'.
GET /index.html

<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <meta name="description" content="">
        <meta name="robots" content="noindex">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width, initial-scale=1">

        <base href="">

        <title>Department of Computer Science</title>
```

Soket

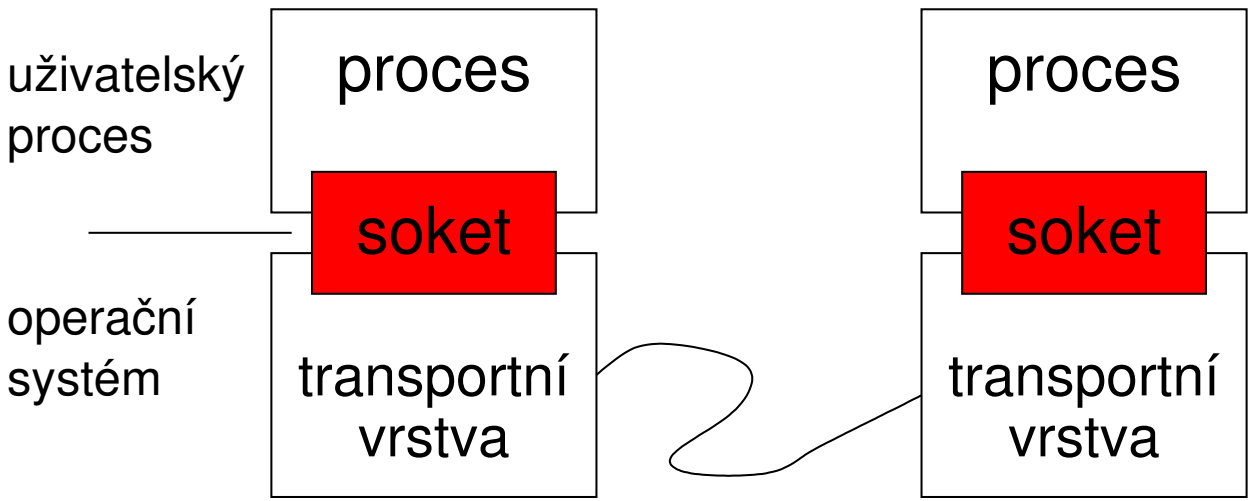
Soket je objekt, který propojuje aplikaci s nějakým „síťovým" protokolem.

- 1981 BSD4.1 Unix.
- Soket je softwarová komponenta.
- Soket je obecný objekt komunikace mezi dvěma procesy.

  Není omezen pouze na TCP/IP.
- Soket API konvertuje obecnou aplikační vrstvu na specifický protokol transportní vrstvy.
- Soket API definuje operace nad soketem (primitiva).
- Soket reprezentuje koncový bod komunikace.

Soket - aplikace a OS

Síťové rozhraní patří mezi sdílené prostředky, proto přístup k němu řídí OS.



Soket a TCP/IP

- Dva základní transportní protokoly TCP a UDP.
- Plná specifikace soketu (ve spojení):
  - protokol,
  - lokální adresa,
  - vzdálená adresa (může být na témže hostu).
- Adresa se skládá z IP adresy a čísla portu.

Příklad obsahu Soket deskriptoru

```
Family:        PF_INET
Service:       SOCK_STREAM
Local IP:      147.32.85.234
Remote IP:     147.321.85.85
Local Port:    55515
Remote Port:   22
```

Základní primitiva soketu

- create - vytvoření soketu.
- bind - přiřazení portu soketu na lokální adrese.
- connect - iniciace spojení na vzdálený soket.
- listen - čekání na iniciaci spojení.
- accept - přijmutí spojení.
- send - poslání zprávy.
- receive - přijmutí zprávy.
- shutdown - uzavření spojení.
- close - uvolnění soketu.

Z pohledu vlastní komunikace jsou nejdůležitější primitiva send a receive.
Ostatní souvisí s navazováním a ukončováním spojení.

Soket a klient/server

TCP protokol vyžaduje model komunikace klient/server.

1. Server očekává žádost o spojení.
2. Klient navazuje spojení se serverem.
3. Po navázání spojení probíhá komunikace.
4. Uzavření komunikace.

Soket serveru čekající na žádost o spojení není aktivní (neprobíhá komunikace). Říkáme, že je v pasivním režimu.

Soket je

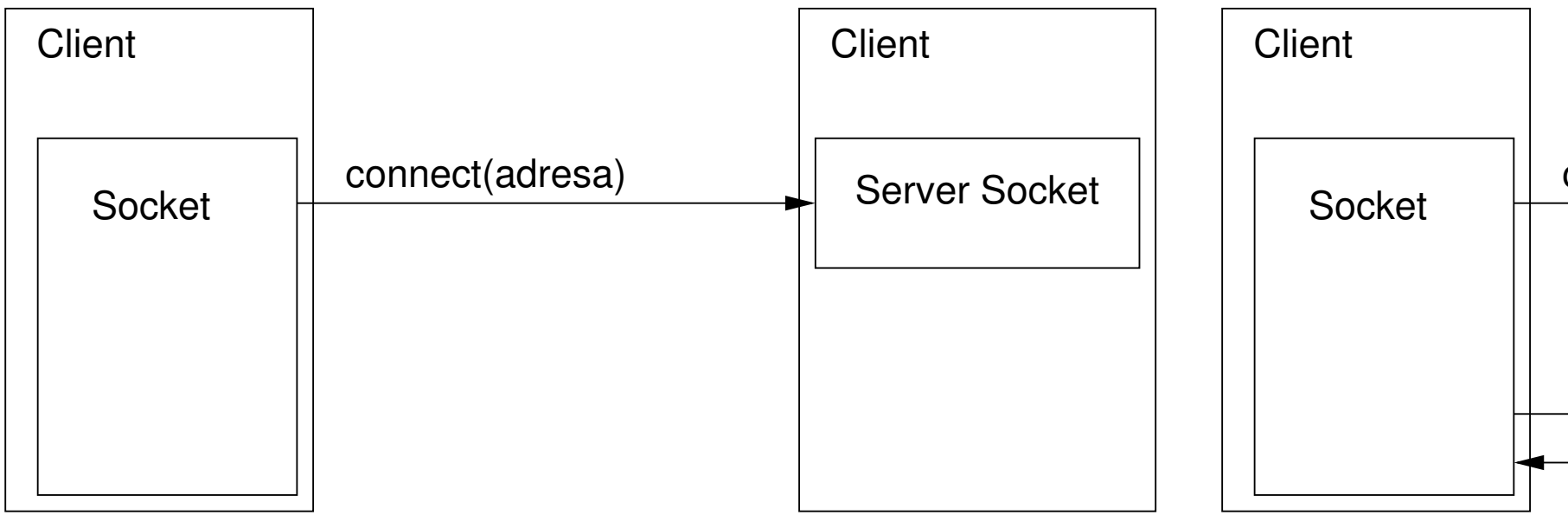
- aktivní - slouží k inicializaci spojení a komunikaci.
- pasivní - čeká na příchozí spojení.

Nepřipojený soket konvertujeme na pasivní soket primitivem `listen`. Alternativně používáme tzv. *Serverový soket*.

Accept - přijmutí spojení

Primitivum `accept` způsobí:

1. čekání na příchozí žádost o spojení,
2. příjem spojení, vrací nový soket deskriptor.



Bind - nastavení adresy

Přiřadí soketu lokální adresu (IP, port).

- Nastavení IP adresy je užitečné v případě více síťových rozhraní.
  - TCP Server přijímá spojení pouze z nastavené síťové adresy (sítě).
  - TCP Client posílá datagramy přes nastavené síťové rozhraní.
- Server nastavuje port na „standardní" hodnotu (80 http, 22 ssh. ...), tak aby se klient mohl připojit.
- TCP klient nemusí nastavovat port.
- V případě, že není adresa ani port specifikován vybere OS volnou hodnotu.

Čísla portů

BSD rozdělení:

- 1-1023 - BSD rezervované,
- 1024-5000 - BSD dočasné porty, s krátkou dobou použití, např. spojení s webovým serverem,
- 5001-65535 - BSD servery, neprivilegované.

IANA: Internet Assigned Numbers Authority

- 0-1023 - IANA známé porty,
- 1024-49151 - IANA registrované porty,
- 49152-65535 - IANA dynamické nebo privátní porty.

TCP Příklad

Příklad server

```
servsock=sock();
bind(servsock, port);
listen(servsock);
//servsock je v pasivním režimu
sock=accept(servsock);
//spojení navázáno
msg=receive(sock);
send(sock, msg2);
close(sock);
```

Příklad klient

```
sock = sock();
add = address(
        servip, port);
connect(sock, add);
//spojení navázáno
send(sock, msg);
msg = receive(sock);
close(sock);
```

UDP soket

- UDP komunikace neobsahuje spojení mezi serverem a klientem.
- Vysílací strana explicitně stanovuje cílovou IP adresu a port.
- Přijímací strana musí získat IP adresu a port vysílací strany z přijatého datagramu.
- Vysílaná data mohou být:
  - přijata v jiném pořadí (datagramů),
  - ztracena.

Server čeká, že něco přijme od nějakého klienta.

UDP Příklad

Příklad server

```
sock=sock();
bind(sock, port);
//sock je připraven na přijmutí dat
od klienta
msg=receive(sock);
clAdd = address(msg);
//poslání dat na klientovu adresu
sendTo(sock, msg2, clAdd);
close(sock);
```

Příklad klient

```
sock = sock();
add = address(
        servip, port);
//poslání dat na adresu serveru
sendTo(sock, msg, add);
//doufám, že mne server vyslyší a
pošle odpověď
msg = receive(sock);
close(sock);
```

Server může komunikovat s více klienty prostřednictvím jediného soketu (multiplexing).

Síťování a paralelismus (vlákna)

Doporučení pro běžné aplikace, pokud to programové prostředí dovoluje.

- *TCP server realizuje obsluhu klienta samostatným vláknem, Boss/-Worker model.*
- *UDP server komunikující prostřednictvím jediného soketu je výhodné realizovat multiplexem I/O operací.*
- *Klient, u kterého je soket jedním z mnoha generátorů událostí, realizujeme více vláknově.*
- *Asynchronní model I/O operací je typický v embedded aplikacích.*
  - *Nárůst výpočetních výkonu embedded procesorů vede na používání OS s podporou paralelismu (pseudoparalelismu).*
- *Aktivnímu čekání se snažíme pokud možno vyhnout.*

Čtení proudu

Proud je „nekonečná" posloupnost bytů.

- Jediné čtení zprávy z proudu, může vrátit celou zprávu nebo pouze jediný byte.
- Aplikační protokol musí definovat jak rozpoznat zprávu
- Identifikace začátku a konce zprávy je zpravidla kontextově závislá (stavová komunikace).
- TCP proud používáme jako „spolehlivého" kanálu pro přenos zpráv definované délky (např. HTTP).

Příklad
První dva byty obsahují velikost n zprávy v bytech. Začátek následující zprávy se nachází v proudu o n bytů dále.

- V případě chybné detekce nebo nepřijetí zprávy včas
  - ukončujeme spojení,
  - pokusíme se identifikovat začátek další zprávy.

SLIP - Serial Line Internet Protocol, RFC 1055.

Sokety v Javě

Lesson: All About Sockets

http://docs.oracle.com/javase/tutorial/networking/sockets/index.html

- UDP soket `java.net.DatagramSocket`
- TCP sokety:
  - `java.net.ServerSocket`
  - `java.net.Socket`
- Adresa
  - `String` host, `int` port,
  - `java.net.InetAddress`.
  - `java.net.SocketAddress`.

UDP soket

- Datová jednotka `java.net.DatagramPacket`.
  - `DatagramPacket(byte[] buf, int length)`
  - `DatagramPacket(byte[] buf, int length, InetAddress address, int port)`
  - `byte[] getData()`
- Primitiva
  - **connect**`(InetAddress address, int port)`
  - **bind**`(SocketAddress addr)`
  - **disconnect**`()`
  - **close**`()`
  - **receive**`(DatagramPacket p)`
  - **send**`(DatagramPacket p)`
    Cílová adresa je součástí datagramu.

TCP soket

- Server soket primitiva
  - **bind**(SocketAddress endpoint)
  - Socket **accept**()
  - **close**()
- Soket (klientský) primitiva
  - **connect**(SocketAddress endpoint)
  - **connect**(SocketAddress endpoint, int timeout)
  - **bind**(SocketAddress bindpoint)
    Jaké rozhraní a jaký port chceme použít pro spojení (null).
  - **close**()
  - Zápis a čtení je realizováno proudy.
    - OutputStream getOutputStream()
    - InputStream getInputStream()

Ošetření výjimečných stavů

Mechanismem výjimek `java.net.SocketException`, resp. `java.io.IOException`.

- `BindException`
- `ConnectException`
- `NoRouteToHostException`
- `ProtocolException`
- `SocketException`
- `SocketTimeoutException`
- `UnknownHostException`

Část II

Příklad – jednoduchý klient a server

Popis činnosti

- Jednoduchý *telnet* server s dvěma příkazy.
  - `time` pošle aktuální čas serveru.
  - `bye` ukončení spojení.
- Textově orientované spojení.
- Po navázání spojení (TCP) musí klient poslat uživatelské jméno a heslo.

Definice protokolu

1. Po navázání spojení server posílá výzvu '`Username:`'.
2. Klient odpovídá posláním uživatelského jména zakončeného znakem '`\n`'.
3. Server posílá výzvu '`Password:`'.
4. Klient odpovídá posláním hesla zakončeného znakem '`\n`'.
5. Server odpovídá zprávou '`Welcome`\n'.
6. Klient může posílat serveru příkazy v libovolném pořadí.

Definice protokolu - příkazy

- Příkaz se skládá ze jména příkazu a znaku konce řádky '\n'.
- Server odpovídá textovou zprávou závislou na příkazu, ukončenou '\n'.
- Příkazy:
  - Žádost o zaslání aktuálního času.
    1. Klient: 'time\n'.
    2. Server: posílá aktuální čas ve formátu 'time is: E M d hh:mm:ss zzz yyyy\'.
  - Ukončení spojení.
    1. Klient: 'bye\n'.
    2. Server: posílá konec proudu a zavírá soket.

Implementace

Implementaci rozdělíme na třídy:

- `ParseMessage` - realizuje čtení a zápis textové zprávy z/do proudu.
  - Obsah textové zprávy může začínat a nebo končit zadanou sekvencí znaků.
- `Server` - otevírá serverový soket na zadaném portu, po přijetí klienta vytváří ovladač klientského spojení.
- `ClienHandler` - realizuje obsluhu klientského spojení v samostatném vlákně.
- `Client` - testovací klient, který se připojí k serveru na zadané adrese a portu.

  Pošle uživatelské jméno, heslo a žádost o aktuální čas, který vypíše na obrazovku (pouze časový údaj) a skončí. Vše proběhne bez interakce uživatele.

ParseMessage

```
1  class ParseMessage {
2    void write(String msg) throws IOException {
3      out.write(msg.getBytes());
4    }
5    String read(String startStr, String endStr) throws
       IOException {
6      byte[] start = startStr.getBytes();
7      byte[] end = endStr.getBytes();
8      int sI = 0; int eI = 0; byte r;int count = 0;
9      while((sI < start.length)
10         && ((r = (byte)in.read()) != -1)) {
11       sI = (r == start[sI]) ? sI+1 : 0;
12     }
13     while ((eI < end.length) && (count < BUFFSIZE)
14         && ((r = (byte)in.read()) != -1)) {
15       buffer[count++] = r;
16       eI = (r == end[eI]) ? eI+1 : 0;
17     }
18     return new String(buffer, 0,
19         count > end.length ? count-end.length : 0);
20   }
21 }
```

ClientHandler 1/3

```
1  class ClientHandler extends ParseMessage implements
       Runnable {
2      static final int UNKNOWN = -1;
3      static final int TIME = 0;
4      static final int BYE = 1;
5      static final int NUMBER = 2;
6      static final String[] STRCMD = {"time", "bye"};
7
8      static int parseCmd(String str) {
9          int ret = UNKNOWN;
10         for (int i = 0; i < NUMBER; i++) {
11     if (str.compareTo(STRCMD[i]) == 0) {
12         ret = i;
13         break;
14     }
15         }
16         return ret;
17     }
18
19     Socket sock; //klientsky soket
20     int id; //cislo klienta
```

ClientHandler 2/3

```
1       ClientHandler(Socket iSocket, int iID) throws
         IOException {
2          sock = iSocket;
3          id = iID;
4          out = sock.getOutputStream();
5          in = sock.getInputStream();
6       }
7
8       public void start() { new Thread(this).start(); }
9       void log(String str) { System.out.println(str);  }
10
11      public void run() {
12         String cID = "client["+id+"] ";
13         try {
14            log(cID + "Accepted");
15            write("Login:");
16            log(cID + "Username:" + read("", "\n"));
17            write("Password:");
18            log(cID + "Password:" + read("", "\n"));
19            write("Welcome\n");
```

ClientHandler 3/3

```
1       ... //run pokracování
2     boolean quit = false;
3     while (!quit) {
4        switch(parseCmd(read("", "\n"))) {
5         case TIME:
6           write("time is:"+ new Date().toString() + "\n");
7           break;
8         case BYE:
9           log(cID + "Client sends bye");
10          quit = true;
11          break;
12        default:
13          log(cID + "Unknown message, disconnect");
14          quit = true;
15          break;
16        } }
17
18      sock.shutdownOutput();
19      sock.close();
20    } catch (Exception e) {
21     System.out.println(cID + "Exception:" + e.getMessage())
        ;
22     e.printStackTrace();
23    } }
```

Server

```
1  public class Server {
2    public Server(int port) throws IOException {
3      int i = 0;
4      ServerSocket servsock = new ServerSocket(port);
5      while (true) {
6        try {
7          new ClientHandler(servsock.accept(), i++);
8        } catch (IOException e) {
9          System.out.println("IO error in new client");
10       } }
11   } // Server()
12
13   public static void main(String[] args) {
14     try {
15       new Server(args.length > 0 ?
16           Integer.parseInt(args[0]) : 9000);
17     } catch (Exception e) {
18       e.printStackTrace();
19     }
20   }
21 }
```

Client 1/2

```
1  public class Client extends ParseMessage {
2    Socket sock;
3    public static void main(String[] args) {
4      Client c = new Client(
5          args.length > 0 ? args[0] : "localhost",
6          args.length > 1 ? Integer.parseInt(args[1]) : 9000
7          );
8    }
9
10   Client(String host, int port) {
11     try {
12       sock = new Socket();
13       sock.connect(new InetSocketAddress(host, port));
14       out = sock.getOutputStream();
15       in = sock.getInputStream();
```

Client 2/2

```
1           //Client konstruktor pokračování
2           write("user\n");
3           read("", "Password:");
4           System.out.println("Password prompt readed");
5           write("heslo\n");
6           read("", "Welcome\n");
7           write("time\n");
8           out.flush();
9           System.out.println("Time on server is " + read("time
          is:", "\n"));
10          write("bye\n");
11          sock.shutdownOutput();
12          sock.close();
13          System.out.println("Communication END");
14      } catch (Exception e) {
15          System.out.println("Exception:" + e.getMessage());
16      }
17  }}
```

Ukázka činnosti

Příklad Telnet

```
1 oredre$ java Telnet
2 Login:telnet
3 Password:tel
4 Welcome
5 time
6 time is:Tue Nov 28 09:56:49
      CET 2006
7 time
8 time is:Tue Nov 28 09:56:50
      CET 2006
9 bye
```

Příklad Server

```
1 oredre$ java Server
2 client[0] Accepted
3 client[0] Username:telnet
4 client[1] Accepted
5 client[1] Username:user
6 client[1] Password:heslo
7 client[1] Client sends bye
8 client[0] Password:tel
9 client[0] Client sends bye
```

Příklad Klient

```
1 oredre$ java Client
2 Password prompt readed
3 Time on server is Tue Nov 28 09:56:40 CET 2006
4 Communication END
```

Část III

Příklad aplikace – Boss/Worker

Příklad vícevláknové aplikace klient/server

- K serveru se připojí klient, který obdrží parametry pro generování grafu.
- Klient vrací serveru hash hodnotu řešení nejkratších cest z uzlu 0.
- Během vlastního řešení musí klient komunikovat se serverem definovaným způsobem a udržovat spojení.
- Po odeslání výsledku server posílá potvrzení správného / špatného řešení a ukončuje spojení.
- Není-li spojení správně udržované (periodické), posíláním `alive` zpráv, ukončuje server spojení.

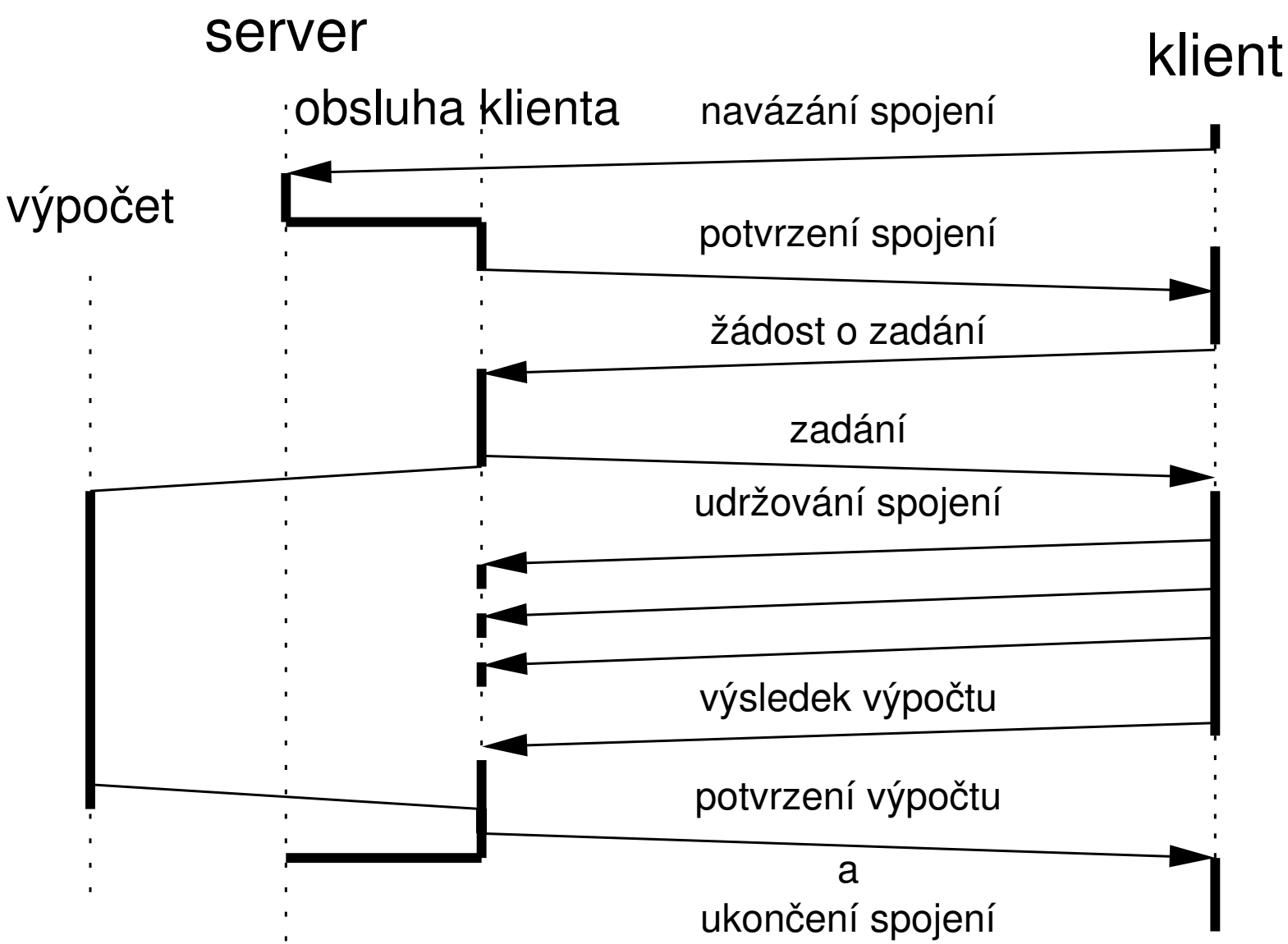
Popis činnosti serveru

Po přijetí klienta jsou provedeny následující operace.

1. Poslání parametrů spojení klientu.
2. Čekání na žádost o parametry grafu.
3. Poslání parametrů grafu.
4. Generování grafu, hledání řešení a výpočet hash.
5. Příjem klientského řešení.
6. Porovnání vlastního a klientského řešení.
7. Poslání výsledku porovnání klientu.
8. Ukončení spojení.

Během bodu 4 a 5 je nutné hlídat, zda klient udržuje spojení.
Klient může poslat řešení dříve než jej nalezne server.

Průběh činnosti



Základní architektura

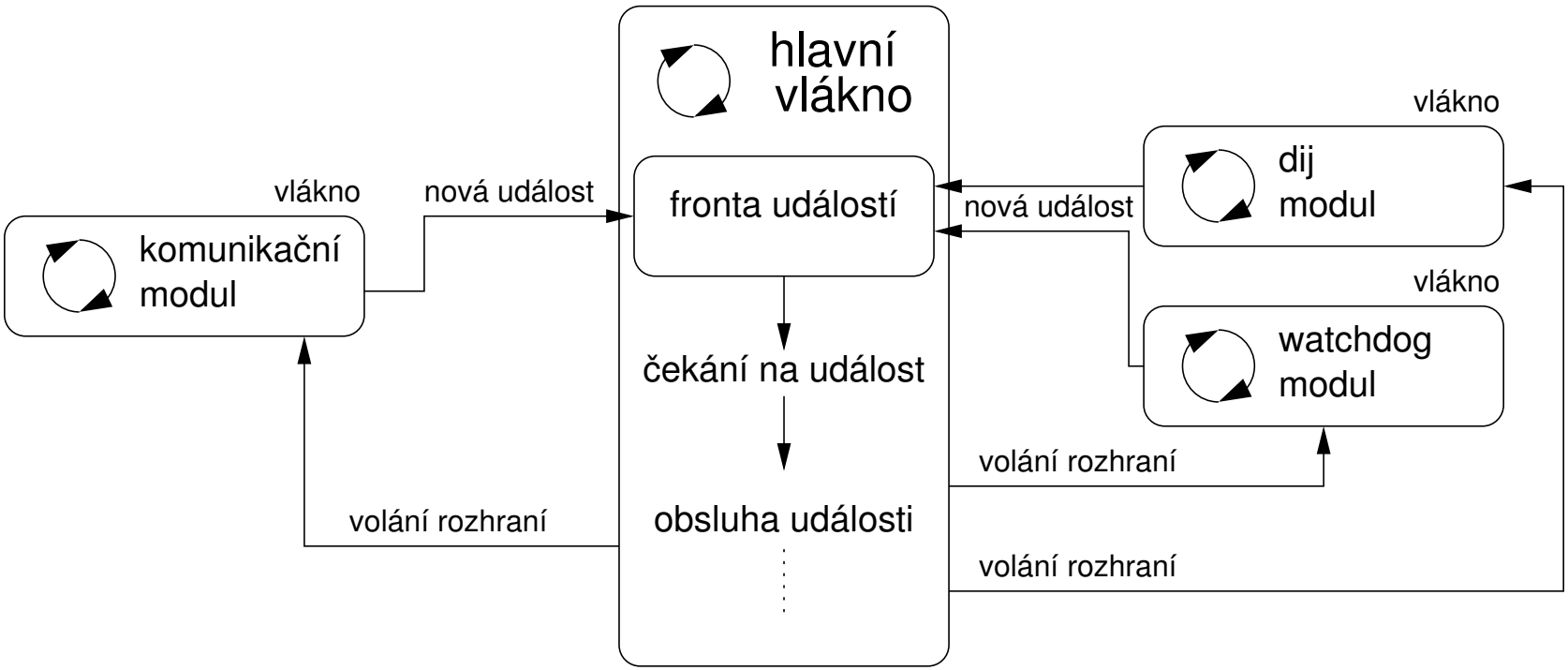
Pro každého připojeného klienta vytvoříme `clientHandler`, který obsahuje

- komunikační modul `comm`,
- watchdog modul `watchdog`, který hlídá udržování komunikace s klientem,
- modul pro řešení zadaní `dij`, který volá program `tdijkstra`.

Synchronizační mechanismy:

- fronta událostí,
- exkluzivní přístup ke sdíleným proměnným.

Propojení modulů



Fronta událostí

```
1  class EventQueue {
2      Queue queue;
3      Object cond;
4
5      EventQueue() {
6          queue = new Queue();
7          cond = new Object();
8      }
9
10     synchronized void addEvent(int event) {
11         queue.push(event);
12         cond.notify();
13     }
14
15     synchronized int getEvent() {
16         if (queue.size() == 0) {
17             cond.wait();
18         }
19         return queue.pop();
20     }
21 }
```

Modul Dij

```
1  class Dij extends Thread {
2      EventQueue queue;
3      int number; int seed; int from;
4      int hash;
5      Dij(EventQueue iQueue) {
6          queue = iQueue;
7      }
8      public void run() {
9          hash = callTdijkstra(number, seed, from);
10         queue.addEvent(SOLUTION_FOUND);
11     }
12     synchronized int getHash() {
13         return hash;
14     }
15     synchronized void solve(int n, int s, int f) {
16         if (!isAlive()) {
17             number = n; seed = s; from = f;
18             start();
19     } }
20     synchronized void shutdown() {
21         join();
22     }
23 }
```

Modul Watchdog 1/2

```
1  public class Watchdog extends Thread {
2      EventQueue queue;
3      int period;
4      int ping;
5      Watchdog(EventQueue iQueue, int iPeriod) {
6          queue = iQueue;
7          period = iPeriod;
8      }
9
10
11     synchronized void alive() {
12         ping++;
13     }
```

Modul Watchdog 2/2

```
1       public void run() {
2           boolean q = false;
3           while (!q) {
4               sleep(period);
5               synchronized(this) {
6                   q = quit;
7                   if (ping == 0) {
8                       eventqueue.add(WATCHDOG_TIMEOUT);
9                   }
10                  ping = 0;
11              }
12          }
13      }
14
15      void shutdown() {
16          synchronized(this) {
17              quit = true;
18          }
19          join();
20      }
21  }
```

Modul Comm 1/2

```
1  class Comm extends Thread {
2    NetConnection conn;
3    EventQueue queue;
4    Comm(EventQueue iQueue, NetConnection iConn) {
5      queue = iQueue;
6      conn = iCon;
7    }
8    synchronized void shutdown() {
9      if (!isAlive()) {
10       conn.stop(); //stop network (blocking receiving)
11       join();
12   } }
13   private synchronized int parse(CommMessage msg) {
14     ...
15     return msgType;
16   }
17   int synchronized getHash() {
18     //client hash from network connection
19     return hash;
20   }
```

Modul Comm 2/2

```
1  public void run() {
2      CommMessage msg;
3      boolean quit = false;
4      //blocking receive
5      while(!quit && (msg = conn.receive() != null)) {
6          switch(parse(msg)) {
7              case MSG_GETPARAM:
8                  queue.addEvent(CLIENT_GETPARAM);
9              case MSG_ALIVE:
10                 queue.addEvent(CLIENT_ALIVE);
11                 break;
12             case MSG_SOLUTION:
13                 queue.addEvent(CLIENT_SOLUTION);
14             default:
15                 //unknown message
16                 quit = true;
17                 break;
18         }
19     }
20     //inform main thread
21     queue.addEvent(COMM_TERMINATED);
22 }
```

Hlavní vlákno 1/3

ClientHandler je vytvořen po navázání spojení.

```
1  public class ClientHandler extends Thread {
2      EventQueue queue;
3      Watchdog watchdog;
4      Dij dij;
5      Comm com;
6      ClientHandler(NetConnection conn) {
7          queue = new EventQueue();
8          watchdog = new Watchdog(queue, PERIOD);
9          dij = new Dij(queue);
10         comm = new Comm(queue, conn);
11         start();
12     }
13
14     void shutdown() {
15         watchdog.shutdown();
16         comm.shutdown();
17         dij.shutdown();
18         queue.addEvent(STOP);
19         join();
20     }
```

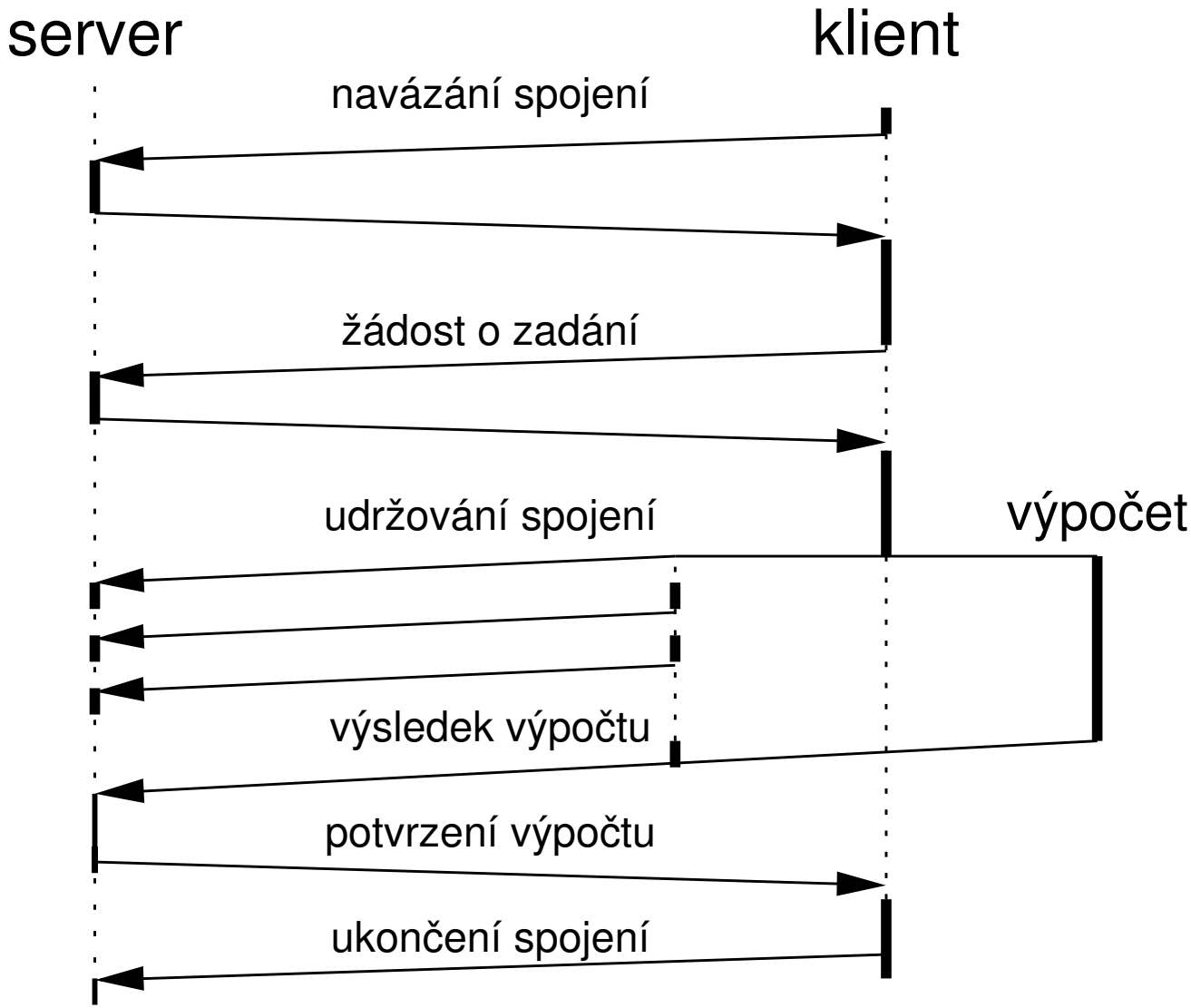
Hlavní vlákno 2/3

```
1  public void run() {
2    int ev;
3    boolean clientSolution; boolean mySolution;
4    Param par = new Param();
5    while (ev = queue.getEvent() != STOP) {
6      switch(ev) {
7        case CLIENT_GETPARAM:
8          par = generateParam();
9          dij.solve(par.number, par.seed, par.from);
10         comm.sendParam(param);
11         watchdog.start();
12         break;
13       case CLIENT_ALIVE:
14         watchdog.alive();
15         break;
16       case CLIENT_SOLUTION:
17         clientSolution = true;
18         queue.addEvent(COMPARE);
19         break;
20       case COMM_TERMINATED:
21         queue.addEvent(STOP);
22         break;
```

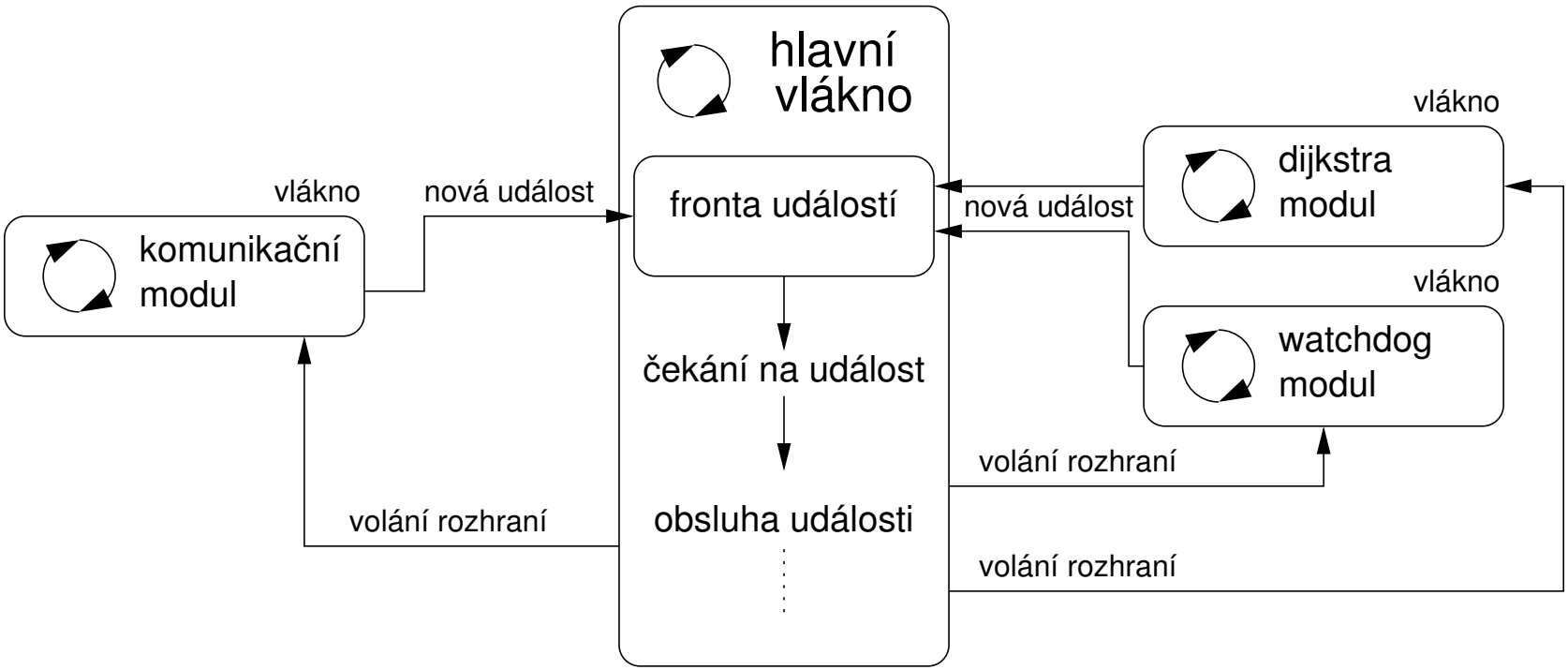
Hlavní vlákno 3/3

```
1          case SOLUTION_FOUND:
2            mySolution = true;
3            queue.addEvent(COMPARE);
4            break;
5          case COMPARE:
6            if (clientSolution && mySolution) {
7              if (dij.getHash() == comm.getHash()) {
8                comm.sendDisconnect("Solution OK");
9              } else {
10               comm.sendDisconnect("Solution WRONG");
11             }
12             queue.addEvent(STOP);
13           }
14           break;
15         case WATCHDOG_TIMEOUT:
16           comm.sendDisconnect("Alive timeout");
17           queue.addEvent(STOP);
18           break;
19         default://unknown event
20           queue.addEvent(STOP);
21           break;
22       }
23 } } }
```

Jak implementovat klientskou část?

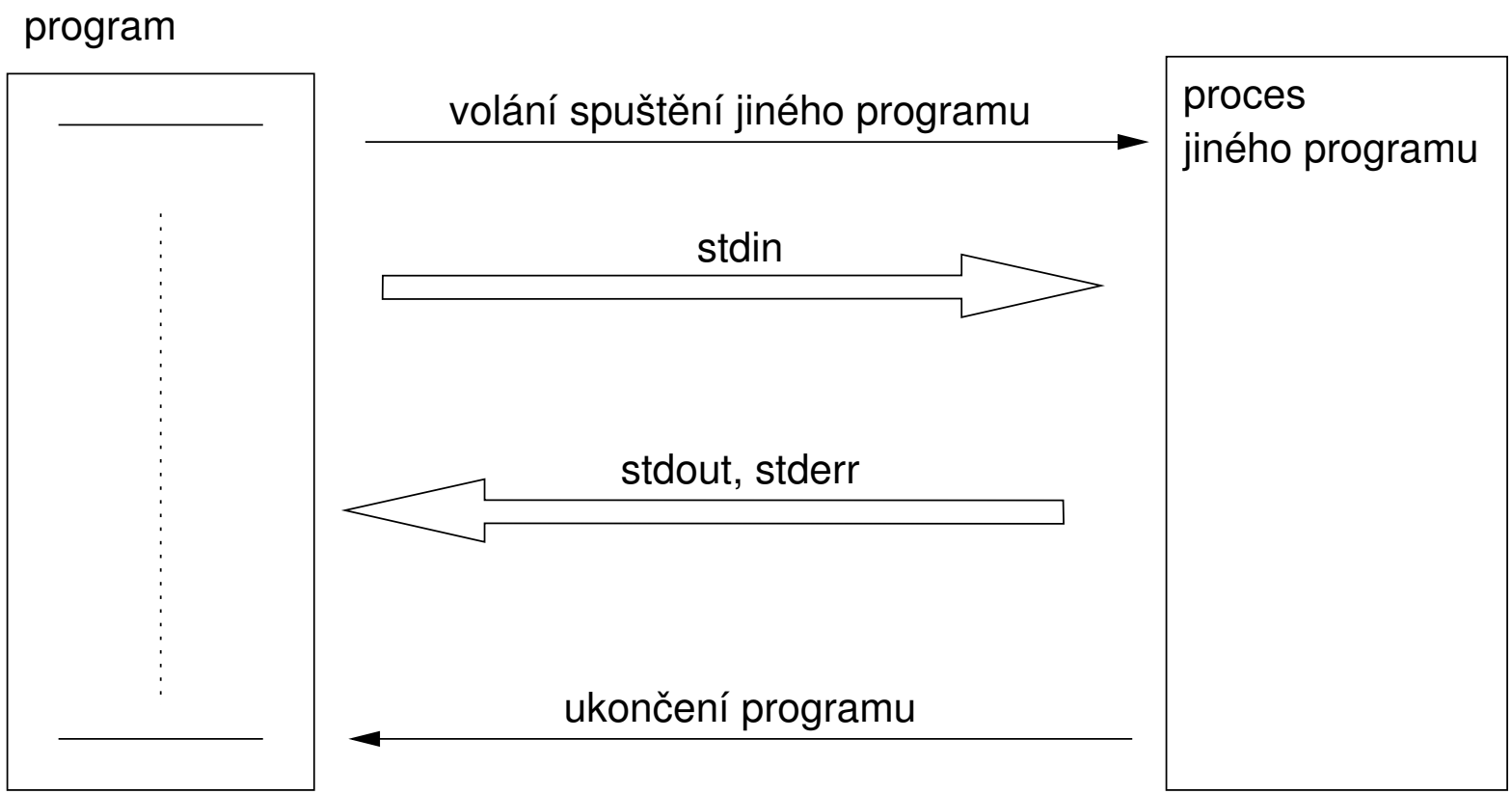


Propojení modulů



Příklad Spuštění jiného programu z procesu

Příklad spuštění jiného programu z Javy



- Identická architektura aplikace (Boss/Worker)
- Hlavní logika programu je v obsluze hlavní smyčky zpráv

Příklad - spuštění jiného programu z procesu

Příklad - spuštění programu `tdijkstra` z Javy

```
1  private boolean callDijkstra() {
2    boolean ret = false;
3    try {
4      String cmd = "./tdijkstra -h -n " + size + " -s " + seed;
5      Process child = Runtime.getRuntime().exec(cmd);
6      child.waitFor();
7      if (child.exitValue() == 0) {
8        BufferedReader out = new BufferedReader(new
       InputStreamReader(child.getInputStream()));
9        hash = Integer.parseInt(out.readLine());
10       ret = true;
11     } else {
12       System.out.println("Error in dijsktra");
13     }
14   } catch (Exception e) {
15     System.out.println("Error: Exception : " + e.getMessage());
16   }
17   return ret;
18 }
```

Co se stane při nedostatečné vyrovnávací paměti pro stdout.

Shrnutí přednášky

Diskutovaná témata

- Síťování
- Síťová API – Soket
- Síťování v Javě
- Příklady
  - Jednoduchý klient a server
  - Boss/Worker Server

- Příště: Výkon a profilování