Distribuované systémy (cvičení)

Jan Janeček, Jan Kubr, Martin Červený

Červen 1999

Předmluva

Tento text je učební pomůckou pro studenty denního studia Elektrotechnické fakulty ČVUT pro cvičení z předmětu Distribuované systémy. Jeho použití předpokládá základní znalost programování v jazyce C a základní znalost operačního systému UNIX.

Text byl vytvářen jako pomůcka pro programování úloh, které zahrnují komunikaci v prostředí TCP/IP. Popisuje technologii BSD socketů (protokoly UDP a TCP), technologie vzdáleného volání procedur ONC RPC a OSF DCE. Věnuje se začlenění protokolů UDP, TCP a protokolů aplikačních (FTP, HTTP) do jazyka Java, navíc jsou doplněny informace potřebné pro práci se vzdálenými objekty RMI (Remote Method Invocation) a pro komunikaci se servery WWW. Okrajově si všímá i technologie dovolující spolupráci aplikačních komponent CORBA. Našim cílem bylo poskytnout základní informace a uvést příklady použití popisovaných technik, podrobnější a aktuálnější údaje získá čtenář v manuálových stránkách konkrétního systému.

Myslíme si, že na toto místo patří i jazyková poznámka. Náš text se zabývá oblastí, ve které se objevuje a běžně používá řada termínů jazyka současné techniky - angličtiny. Při psaní tohoto textu jsme se snažili respektovat pravidla a duch češtiny. Tam, kde existuje zavedený, nebo dokonce standardizovaný český odborný termín, užíváme ten a vyhýbáme se oborovému slangu (např. používáme standardizovaný termín slabika nebo oktet tam, kde dnes řada publikací používá poměrně nehezký termín „bajt“). Tam, kde alespoň částečně akceptovaný český termín neexistuje, a kde doslovný překlad anglického termínu není dostatečně výstižný a/nebo přetížení českého termínu odlišným významem není výstižné nebo by vedlo ke dvojznačnosti, jsme raději zůstali u původních termínů anglických (pochopitelně bez pokusů o problematický fonetický zápis, české skloňování nebo dokonce časování) a u zkratek. Z čistě praktických důvodů (využitelnost pro výuku v angličtině) jsou anglické termíny použity v obrázcích.

Autoři se o zpracování textu podělili takto: kapitoly 1,2,4.4,5 a 6 napsal Ing. Jan Janeček CSc., kapitolu 3 Ing. Martin Červený a kapitolu 4 Ing. Jan Kubr. Jako autoři chceme poděkovat všem, kteří nám s přípravou textu pomohli, poskytli potřebné materiály a informace.

Text vychází v této formě poprvé a autoři uvítají poznámky pečlivého čtenáře k jeho formě a obsahu.

Praha, červen 1999 autoři

Obsah

1. Úvod

Distribuované zpracování informace přestává být výjimečnou technologií. Komunikace mezi komponentami vyměňujícími si navzájem vstupní data a výsledky zpracování se stává běžnou součástí programových systémů. Pochopení technik počítačové komunikace, znalost základních rozhraní využitelných při tvorbě distribuovaných aplikací a schopnost jejich efektivního využití v praxi patří dnes k běžné výbavě profesionála v oblasti programového vybavení.

Text, který začínáte číst, si klade za cíl poskytnout informace potřebné pro programování současných distribuovaných aplikací. Uváděné technologie jsou vázány na Internet a přes určitou míru systémové nezávislosti na operační systémy vycházející z principů Unixu. Lze předpokládat, že směr rozvoje opírající se o otevřené standardy bude pokračovat a znalosti vázané na tyto technologie budou dlouhodobě užitečné.

Náš text sleduje zmíněný trend a snaží se v pěti následujících kapitolách pokrýt nejdůležitější technologie, s nimiž se dnešní programátor distribuovaných aplikací může setkat. Ty patří převážně do kategorie asymetrické spolupráce procesů, kterou označujeme jako *"klient-server"*).

Následující, *druhá kapitola* je věnována přehledu komunikačních technik, které jsou technologickým základem Internetu (nebo, chcete-li, systému *TCP/IP* nazývaného tak podle jeho nejdůležitějších komunikačních protokolů). Seznámíme se zde s vlastnostmi protokolů IP, ICMP, UDP, TCP a IGMP důležitými pro aplikační procesy, které tyto protokoly používají.

Ve *třetí kapitole* se budeme věnovat klasickému rozhraní *BSD socketů*, jak bylo vytvořeno pro operační systém Unix při jeho doplňování o síťové prostředky. Vzhledem k rozsahu textu jsme se museli omezit na základní rozhraní a nechali jsme stranou jeho modifikaci využívanou v řadě systémů Unix SVR3/4 pod označením TLI (TLI - Transport Layer Interface) a rozšířené rozhraní socketů WinSocks používané v systémech osobních počítačů.

Čtvrtá kapitola je věnována procedurální nadstavbě komunikačního rozhraní, technologiím, které označujeme jako *volání vzdálených procedur* (RPC - Remote Procedure Call). Z řady existujících systémů uvádíme široce využívaný ONC/Sun RPC a OSF DCE. V závěru zmíníme objektově orientovaný a jazykově nezávislý systém CORBA (Common Object Request Brooker Architecture). Proprietární technologie, jako jsou DSOM a DCOM musely zůstat vzhledem k rozsahu textu stranou.

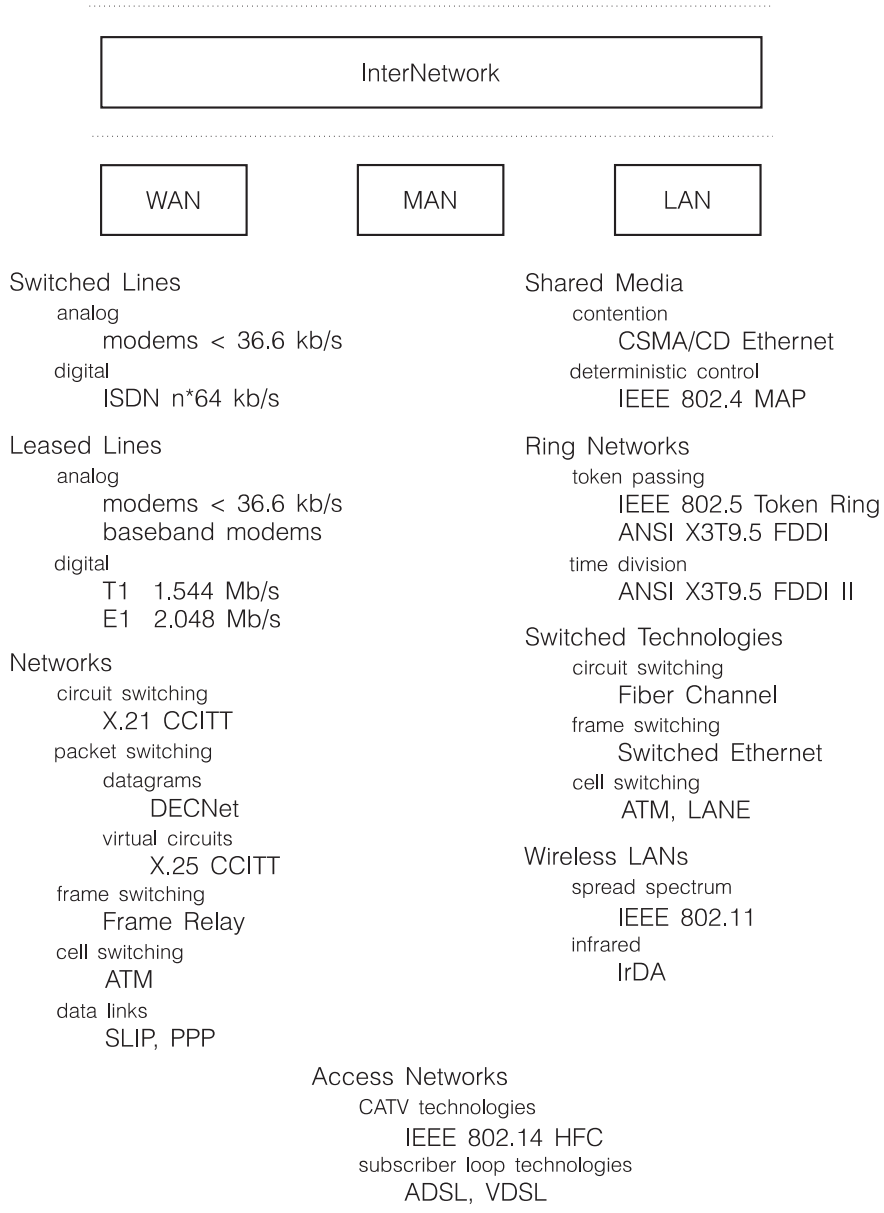
Klasické technologie, tedy BSD sockety a RPC mechanismy, jsou programátorsky poměrně náročné, jejich vazba na programovací jazyk C vyžaduje profesionální přístup a určité zkušenosti. Snížení nároků na programátorskou erudici dovoluje zpřístupnění počítačových komunikací v jazyce, který je schopen vyloučit nejčastejší zdroje chyb. Takovým jazykem se stala *Java*, komunikačním programovým rozhraním tohoto jazyka je věnována *pátá kapitola*. Všimneme si integrace BSD socketů v jejích knihovnách, technologie volání metod ve vzdálených objektech (RMI - Remote Method Invocation) a technologie přenosu souborů adresovaných odkazy URL (URL - Universal Resource Location).

Konečně, současným trendem v přístupu k informačním zdrojům je technologie výměny zvláštních souborů - *stránek HTML* - využívaná v systému "pavučiny" WWW (WWW - World Wide Web). Této technologii a její programové podpoře je věnována poslední, *šestá kapitola*.

Cílem našeho textu je poskytnout podklady pro praktická řešení konkrétních úloh a kromě informací o popisovaných rozhraních proto obsahuje i jednoduché ilustrační příklady.

2. Komunikační protokoly TCP/IP

K nejvýznamnějším protokolům používaných v lokálních a rozsáhlých přepojovacích sítích se řadí rodina protokolů TCP/IP. Začátek vývoje protokolů se datuje do let 1973-74. Tehdy se v experimentech provedených v síti ARPANET ukázala potřeba upravit do té doby používané komunikační mechanismy. Současně se ukázalo jako nutné vytvořit technologii schopnou propojit sítě pracující pod různými proprietárními protokoly (IBM SNA, Decnet) a sítě podstatně se lišící od běžných propojovacích sítí (využívajících pozemních linkových spojů) svými vlastnostmi (lokální, rádiové a satelitní sítě). Výsledkem těchto prací byl návrh protokolů TCP/IP, které dnes překrývají celou řadu síťových technologií (propojování sítí označujeme jako *internetworking*) a využívají jich při vytváření celosvětové sítě *Internet* (obr. 2.1).



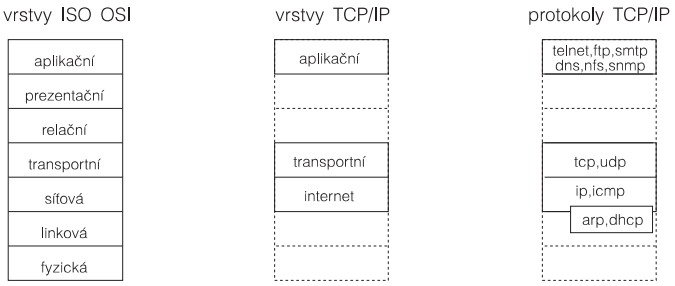
Obr. 2.1: Postavení TCP/IP technologií

Experimenty provedené v roce 1977 firmou Xerox s propojenými sítěmi ARPANET, Ethernet, rádiovou paketovou sítí pozemní a paketovou sítí satelitní ukázaly výhodné vlastnosti nově navržených protokolů TCP/IP. Významným mezníkem pro jejich široké rozšíření bylo

jejich zahrnutí do operačního systému UNIX 4.2BSD realizované vývojovými pracovníky na Universitě v Berkeley (UCB) v roce 1980.

Protokoly TCP/IP sice nejsou definovány klasickou standardizační institucí, ale jejich rozšíření z nich činí *de facto* průmyslový standard. Doporučení s popisem protokolů, poskytovaných služeb a množstvím dalších informací nalezneme v dokumentech volně dostupných na Internetu. Vydávání dokumentů řídí skupina IETF (*Internet Engineering Task Force*) a dělíme je na návrhy doporučení (*Drafts*), vlastní doporučení (RFC - *Request For Comments*) a standardy (STD - *Standards*).

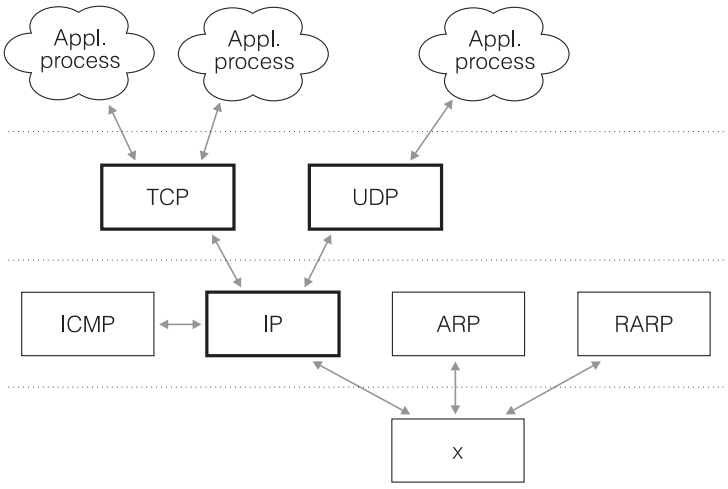
Pozici rodiny protokolů v sedmivrstvové architektuře referenčního modelu ISO OSI si lze znázornit obrázkem 2.2.



Obr. 2.2: Vrstvový model ISO OSI a TCP/IP

Základní protokoly TCP/IP pokrývají síťovou vrstvu, přesněji řečeno, vytvářejí nadstavbu síťové vrstvy propojovaných sítí. Současně vytvářejí transportní rozhraní pro síťové aplikace. Některé z protokolů svázaných s TCP/IP definují i chování, které spadá do vyšších vrstev referenční architektury ISO-OSI (např. vzdálené volání procedur RPC a externí reprezentace dat XDR).

Podrobněji si vazbu jednotlivých protokolů tvořících současné jádro TCP/IP můžeme znázornit na obr. 2.3.



Obr. 2.3: Architektura protokolů TCP/IP

Protokoly TCP/IP se opírají o *přenosový protokol IP*, který lze postavit nad libovolný komunikační kanál (síťové nebo linkové vrstvy) schopný přenést blok oktetů (v obrázku je označený jako x, a může zde být využita libovolná technologie uvedená na obr. 2.3).

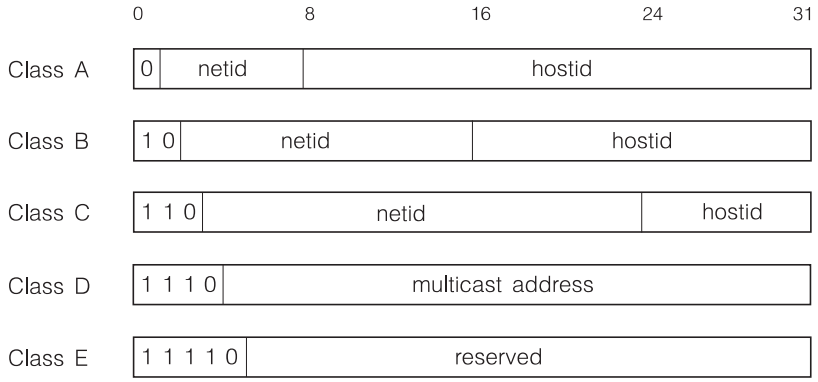
Přenosový protokol IP (verze 4) se liší od protokolů jiných sítí (např. veřejných datových sítí X.25, nebo lokálních sítí NetBIOS a IPX) a lze jej charakterizovat následujícími vlastnostmi:

- Adresace zařízení tvořících síť je nezávislá na adresaci využívaných spojů a propojovaných sítí. Adresa (*IP adresa*) je 32-bitová, jednoznačně přidělená každému síťovému rozhraní zařízení (toto pravidlo dnes porušují vzájemně izolované privátní adresní prostory). Adresní prostor je hierarchicky členěn, zařízení vytvářející podsíť využívají souvislou část adresního prostoru. Členění adresního prostoru podporuje hierarchickou organizaci administrativního přidělování IP adres.

- Přenos IP paketů není potvrzován, pokud existuje vnitřní potvrzování v propojovaných sítích, je samozřejmě využito. Každý paket je směrován nezávisle a je nutné počítat s možností jeho ztráty nebo duplikace a záměny pořadí na straně příjemce.

- Protokol IP byl navržen pro propojování i silně heterogenních sítí. Maximální délka IP paketu je 64 KB, ale běžně pracujeme s pakety kratšími (koncová zařízení musí zvládnout pakety o délce nejméně 576 oktetů). Mezi význačné vlastnosti protokolu IP patří i to, že podporuje dělení paketu na fragmenty při přenosu přes sítě/spoje s limitem délky menším než je délka přenášeného paketu. Jednotlivé fragmenty jsou po rozdělení směrovány nezávisle, pro jejich doručování platí totéž, co jsme si uvedli v předcházejícím bodě pro pakety.

Zastavme se na chvíli u adresace. Každý IP paket obsahuje adresu odesílatele a adresu příjemce, adresy jsou v současné verzi IP protokolu 32-bitové a mají strukturu, která odpovídá obr. 2.4.



Obr. 2.4: Struktura IP adres

Adresa je složena z adresy sítě a adresy zařízení v síti, jednotlivé třídy individuálních (*unicast*) adres (A,B a C) dovolují vhodně přizpůsobit rozsah přiděleného adresního prostoru předpokládanému počtu připojených zařízení. (Současný Internet zvyšuje pružnost přidělování oblastí adres o dělení oblastí třídy A a slučování oblastí třídy C, technika je označována jako CIDR - Classless InterDomain Routing.) Pro lepší přehlednost zapisujeme IP adresy po jednotlivých oktetech dekadicky a jednotlivé oktety oddělujeme tečkou.

Jednotlivé IP pakety/fragmenty jsou sítí směrovány samostatně, výběr cesty se opírá o směrovací tabulky. Ty obsahují údaje pro jednotlivé cílové sítě, případně jejich části - podsítě. Směrovací tabulky mohou být definovány pevně, obsahem souborů jako je */etc/networks*. Mnohem častější, a ve větších sítích praktičtější, je dynamické vytváření směrovacích tabulek.

Protokoly, které výpočet směrovacích tabulek podporují, se opírají o výměnu informací o vzdálenostech v síti (o počtu linek na cestě k cíli, případně o zpoždění na těchto linkách), nebo o výměnu topologických informací a o záplavou šířené informace o změnách vzdáleností.

O první z technik se opírá protokol RIP (*Routing Information Protocol* - RFC 1058) použitelný v malých sítích; o druhou z technik se opírá OSPF (*Open Shortest Path First* - RFC 1583) dovolující hierarchické členění sítě a vhodný i pro sítě rozsáhlé.

Uvedené směrovací protokoly jsou použitelné v sítích (autonomních systémech) připojovaných k páteři/páteřím Internetu a jsou označovány jako vnitřní protokoly autonomních systémů (Intra-Domain Protocols). Funkce samotné páteře je podmíněna existencí dalších směrovacích protokolů (Inter-Domain Protocols) jako jsou EGP (*Exterior Gateway Protocol* - RFC 827), novější BGP (*Border Gateway Protocol* - RFC 1771) a IDRP (Inter-Domain Routing Protocol). Tyto protokoly dovolují oddělit směrování v páteři od směrování v autonomních systémech (ty nemusí být vzájemně kompatibilní). Dovolují směrovat toky mezi autonomními systémy podle pravidel, jejichž cílem může být celková efektivita sítě, využívání určitých páteřních sítí pouze pro určité typy provozu, omezení přenosu mezi autonomními systémy na určité páteřní sítě.

Vedle fyzických IP adres využíváme v síti TCP/IP *adresace jmenné*. Jmenné adresy nám dovolují vytvářet skupiny zařízení - *domény* - nezávisle na jejich IP adresách. Současně zjednodušují i práci s adresami zařízení (jméno se snáze zapamatuje). Jmenná adresace je opět hierarchická, základní domény jsou prvky domén vyšší úrovně.

K převodu jmenné adresy na IP adresu (resp. opačně) používáme jmenné služby. Jmenné služby se v nejjednodušším případech opírají o textové soubory. V rámci Internetu se dnes používají hierarchické jmenné služby s distribuovanou architekturou podporované specializovanými servery - *adresářové systémy* (*Directory Services*). Mezi nejvýznamnější patří služba DNS (Domain Name Service). Kromě ní se můžete setkat se službami NIS (Network Information Services) firmy Sun Microsystems, známé také pod jménem Yellow Pages, NIS+ téže firmy, NDS (Novell Directory Services) firmy Novell, CDS (Cell Directory Services) konsorcia X/Open, postavené pro systém OSF DCE a ISO X.500. Uvedené služby nabízejí většinou i další informace (o uživatelích včetně autentifikace, o skupinách uživatelů, o programech včetně autorizace).

Do IP paketu vkládáme datové bloky protokolů vyšších vrstev. Pro přenos uživatelských dat jsou v současnosti nejdůležitější protokoly UDP (*User Datagram Protocol*) a TCP (*Transmission Control Protocol*). Oba protokoly používají mechanismus zprostředkované adresace zpráv konkrétním aplikačním procesům: zprávy jsou adresovány na porty, na které se připojují aplikace.

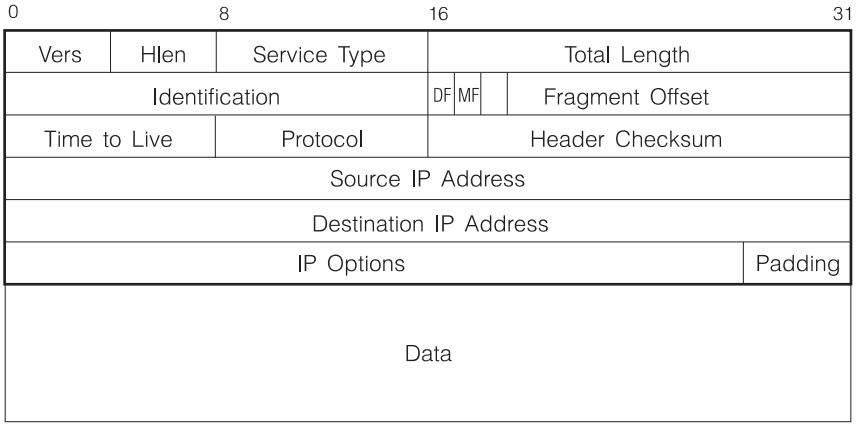
Protokol UDP poskytuje nespojovanou a nepotvrzovanou službu přenosu zpráv a je přímou nadstavbou IP. Rozšiřuje IP protokol pouze o identifikaci zdrojového a cílového portu. Výhodou protokolu UDP je velmi rychlé zpracování systémem. Nevýhody dědí od IP protokolu a pro aplikace je zvláště nevýhodná absence řízení toku (*flow control*). Maximální velikost UDP zprávy je omezena IP vrstvou na méně než 64kB, každá zpráva je přenášena ve zvláštním paketu. Použití UDP se omezuje na aplikace v lokálních sítích (nízká ztrátovost, bez duplikací a záměn), příkladem jsou souborové servery NFS (Network File System) nebo jmenné služby (DNS, NIS). Další oblastí jsou aplikace citlivé na zpoždění - video a audio přenosy (RealAudio).

Protokol TCP poskytuje spojovanou spolehlivou službu. Zajišťuje přenos důsledným potvrzováním přenášených dat. Před začátkem přenosu je mezi aplikacemi navázáno spojení. Detekce chybějících a duplicitních paketů je založena na přenosu sekvenčních čísel. Řídící informace a uživatelská data jsou přenášena v témže paketu (*piggi-backing*) plně duplexně. Pro vyšší efektivitu přenosu je využíván mechanismus okénka (*sliding window*) s automatickým řízením jeho velikosti podle stavu sítě (*flow control*). Na rozdíl od UDP nemusí být zpráva přenesena v jediném paketu (TCP segmentu) a její délka je omezena pouze aplikačním rozhraním.

2.1 Protokol IP (IPv4)

Základní komunikační službou TCP/IP je nezabezpečená datagramová služba - přenos *IP paketů* - popsaná materiálem RFC 791 "Internet Protocol" [1]. Jednotlivé IP pakety jsou sítí směrovány nezávisle, navíc se při na své cestě mohou štěpit na samostatně směrované fragmenty. Důsledkem je skutečnost, že IP pakety nebo jejich fragmenty mohou docházet adresátovi v pořadí odlišném od pořadí jejich vyslání do sítě.

IP pakety mají formát odpovídající obr. 2.5, stručně si uvedeme využití jednotlivých polí jejich formátu:



Obr. 2.5: Struktura IP paketu

Hlavička IP paketu obsahuje informace potřebné pro směrování, případnou fragmentaci a navázání protokolů transportní vrstvy. Úvodní čtyřbitové pole *Vers* označuje použitou verzi IP protokolu, v současné době zde nalezneme čtyřku, u příští verze protokolu IP označované jako IPv6 nebo IPng šestku. Další čtyřbitové pole *Hlen* uvádí délku hlavičky včetně doplňkových polí *Options*, ta podporují mimo jiné informaci o stupni utajení přenášené informace, informaci o řízení sítě (např. směrování po zadané cestě nebo záznam absolvované cesty) a měření. Pole *ServiceType* dovoluje definovat parametry přenosu, které dnes obvykle označujeme jako kvalita služby (QoS - Quality of Service). Lze definovat prioritu, požadavek na minimalizaci doby přenosu, na maximální přenosovou kapacitu a na maximální spolehlivost. Současné směrovače však podporu takto zadaných parametrů běžně neobsahují. Pole *Total Length* uvádí délku IP paketu včetně hlavičky, IP paket může být dlouhý nejvýše 65536 oktetů. Pole *Identification*, příznaky *DF* a *MF* a třináctibitové pole *Fragment Offset* podporují fragmentaci a následné skládání IP paketů. Délka fragmentů musí být násobkem osmi oktetů, pole *Fragment Offset* po doplnění o tři nulové bity udává pozici fragmentu v původním IP paketu.

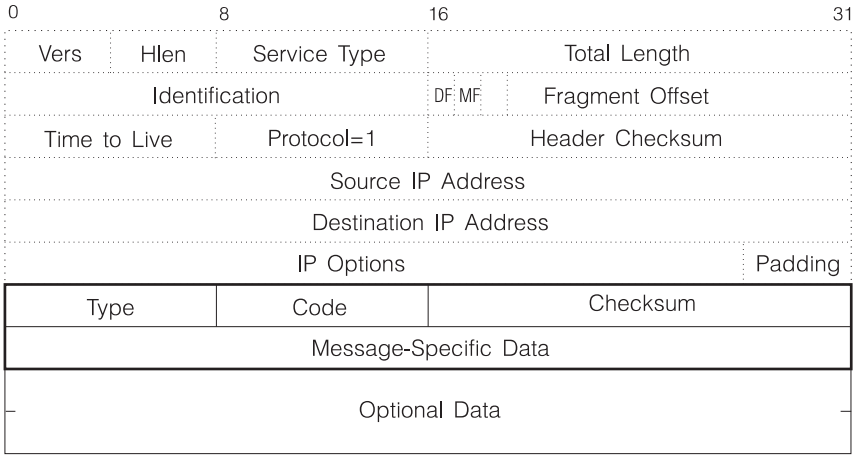
Každá síť nebo dvoubodový spoj má definovánu maximální hodnotu délky paketu označovanou jako *MTU* (*Maximum Transfer Unit*), k fragmentaci dochází při vstupu IP paketu do sítě s MTU menším, než je délka paketu. Skládání fragmentů do IP paketů je možné až u adresáta a je omezené časovým limitem. IP paket nesložený z fragmentů do časového limitu je likvidován. Koncové počítače a směrovače musí být schopné převzít IP pakety s délkou odpovídající parametru MTU připojené sítě/linky, požadovaným minimem je 576 oktetů.

Pole *Time to Live* dovoluje omezit dosah komunikace (pole udává zbývající dobu života paketu ve vteřinách, každý směrovač hodnotu snižuje nejméně o jednotku) a pole *Protocol* určuje protokol transportní vrstvy. Kontrolní součet (modulo 65535) hlavičky IP paketu uložený v poli *Header Checksum* uvažuje při svém výpočtu toto pole (tedy *Header Checksum*) za vynulované. Následují dvaatřicetibitové *IP adresy* adresáta a odesílatele a případné pole *Option* zarovnané na hranici slova výplňovými znaky.

2.1.1 Protokol ICMP

IP protokol nemá vlastní potvrzování a realizuje službu označovanou jako *best-effort* (síť pro přenos IP paketu udělá, co je v jejích možnostech). Přenos dat sítí je však spojen se situacemi, kdy určité informace týkající se přenosu musíme umět předat komunikujícím účastníkům. Jde o situace, kdy je nemožné doručit IP paket, například proto, že zadaná síť nebo počítač jsou nedostupné, že byla překročena doba života IP paketu, že směrovač na cestě neměl dostatek paměti pro uložení IP paketu, nebo že síť nebo počítač leží za bezpečnostní bránou (*firewall*) nedovolující požadovaný přístup.

Takové funkce jsou podporovány specializovaným řídícím protokolem ICMP (*Internet Control Message Protocol*) definovaným v materiálu RFC 792 [2]. Jeho zprávy jsou předávány v běžných IP paketech a jeho dalšími funkcemi je i předávání informací potřebných pro řízení sítě (podpora echo protokolu, podpora směrování a řízení toku, podpora časové synchronizace). Pakety ICMP mají formát odpovídající obr. 2.6:



Obr. 2.6: Struktura paketu ICMP

Osmibitové pole *Type* určuje typ ICMP paketu a najdeme zde následující hodnoty:

0 - Echo Reply,
3 - Destination Unreachable,
4 - Source Quench (oznámení o přetížení směrovače),
5 - Redirect (doporučení na změnu směrování),
8 - Echo Request,
11 - Time Exceeded,
12 - Parameter Problem (),
13 - Timestamp Request,
14 - Timestamp Reply,
17 - Address Mask Request,
18 - Address Mask Reply.

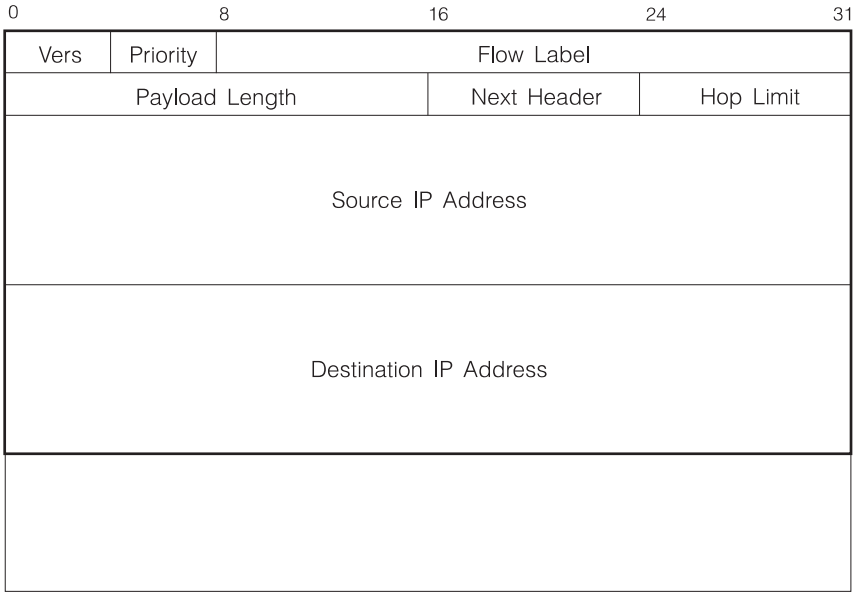
Zajímavé jsou pakety *Source Quench* informující odesítale o přetížení v síti (vyčerpání vyrovnávacích pamětí ve směrovači), které vedlo k likvidaci IP paketu a pakety *Redirect*, které doporučují odesílateli změnit směrovač pro daného adresáta a službu. Pakety *Time Exceeded* indikují překročení doby života IP paketu (využívá je např. služba *traceroute* zjišťující cestu k adresátovi) a vypršení časového limitu pro sestavení IP paketu z fragmentů na straně adresáta.

Doplňkové pole *Code* upřesňuje důvod odeslání ICMP paketu a např. pro nedosažitelného adresáta zde můžeme zjistit přesnější důvod nedoručení IP paketu (nedostupná síť, počítač, protokol nebo port, směrovač nemohl IP paket rozložit pro zadaný zákaz fragmentace).

2.2 Protokol IPv6

Protocol IP byl navrhován pro rozsah sítě Internet předpokládaný počátkem sedmdesátých let. Skutečnost však podstatně představy překonala. Současný počet počítačů již vylučuje možnost jejich adresace původním způsobem při zachování představy hierarchie v adresním prostoru, izolovaných tříd adres a jedinečnosti IP adresy. Naštěstí právě změny v těchto vlastnostech (adresace dovolující využití neobsazených rozsahů adres třídy A , sdružování sousedících adres třídy C, zavedení privátních adresních prostorů a překlad IP adres) přispěly k útlumu požadavků na přidělování nových rozsáhlejších adresních prostorů a k oddálení nástupu technologie, která dovoluje prodloužit adresu z původních 32 bitů na 128 bitů.

Nová technologie označovaná jako IPv6 (IP verze 6) nebo IPng (IP - New Generation) je definována materiály RFC1883 [3] a RFC2373 [4]. Formát paketů IPv6 odpovídá obr. 2.7:



Obr. 2.7: Struktura paketu IPv6

Proti standardní verzi IP protokolu obsahuje hlavička podstatně méně údajů; ty, které v hlavičce zůstávají slouží výhradně směrování, ty, které se směrováním přímo nesouvisí, jsou odsunuty do rozšiřujících hlaviček. Jsou to hlavičky *Hop-by-Hop Options* (informace využívané směrovači, např. informace o extrémně dlouhých - jumbo - paketech), *Routing* (seznam uzlů, kterými má vést cesta paketu), *Fragment* (informace podporující složení paketu z fragmentů, na rozdíl od IPv4 fragmentaci zajišťuje odesílatel), *Destination Options* (podpora funkcí realizovaných na straně adresáta), *Authentication* a *Encapsulating Security Payload* (autentifikace a kryptografická ochrana přenosu). Toto řešení podstatně zvýšilo pružnost architektury s ohledem na budoucí potřeby.

Pole *Vers* hlavičky paketu označuje verzi protokolu (tedy Vers = 6), prvé tři bity pole *Priority* dovolují odlišit různé typy provozu (řízení, interaktivní provoz, dávkový provoz) a nejvyšší bit odlišuje typy provozu s real-time požadavky na přenos. Pole *Flow Label* identifikuje a odlišuje toky dat, odesílatel zde vedle toho může definovat své požadavky na specifickou kvalitu služby - např. pro real-time přenosy. Pole *Payload Length* udává délku pole dat, zajímavostí je uvedení nulové hodnoty pro pakety s délkou pole dat větší než 64 KB (tzv. jumbo pakety), pro které je délka specifikována v následující hlavičce *Hop-to-Hop Options*. Pole *Next Header* identifikuje vyšší protokol (a tedy typ následující hlavičky). Lze zde nalézt hodnoty

0 - data pro IP protokol,
6 - TCP protokol,
17 - UDP protokol,
43 - podpora směrování,
58 - data pro ICMPv6 protokol.

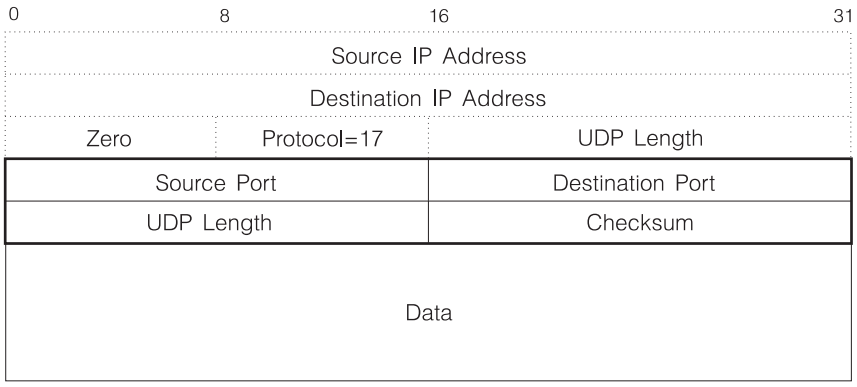
Pole *Hop Limit* v hlavičce IPv6 má funkci shodnou s polem *TTL* protokolu IP s tím, že hodnota je snižována právě o jedničku při průchodu paketu směrovačem. Adresační pole *Source IP Address* a *Destination IP Address* dlouhá 128 bitů dovolují pokrýt potřeby budoucnosti. Do adresního prostoru protokolu IPv6 je vložena adresace původního IP protokolu, ale i protokolů dalších, jako je např. IPX. Jsou pokryty nejen požadavky *unicast* (adresováno konkrétní koncové zařízení) komunikace, ale rozsáhlý prostor je věnován i pro *multicast* (adresovány všechny prvky skupiny) a *anycast* (adresován libovolný z prvků skupiny) komunikaci.

Protokol IPv6 je podporován podobně jako protokol IP speciálním řídícím protokolem. Ten se zde jmenuje ICMPv6 a je definován materiálem RFC1885 [5].

2.3 Protokol UDP

Protokol *UDP* (*User Datagram Protocol*) je jednoduchou nadstavbou protokolu IP a je popsán v materiálu RFC 768 [6].

Zajišťuje transportní multiplex, pro připojení aplikací dává k dispozici 65536 *portů*, z nichž je určitá část vyhrazena pro standardní aplikace. Odesílatelem generovaný kontrolní součet umožňuje detekovat chyby v přenesených datagramech. Strukturu UDP datagramu a pseudohlavičky uvádí obr. 2.8:



Obr. 2.8: Struktura UDP datagramu

Hlavička UDP datagramu obsahuje čísla portů, informaci o délce UDP datagramu (včetně hlavičky) a kontrolní součet (modulo 65535), do kterého jsou zahrnuty i některé údaje z hlavičky IP paketu, tyto přídavné informace označujeme jako pseudohlavičku. Metoda výpočtu kontrolního součtu nechává hodnotu 65535 („záporná“ nula) pole *Checksum* vyhrazenu pro nulový kontrolní součet, hodnota 0 („kladná“ nula) vypovídá o tom, že kontrolní součet není přenášen.

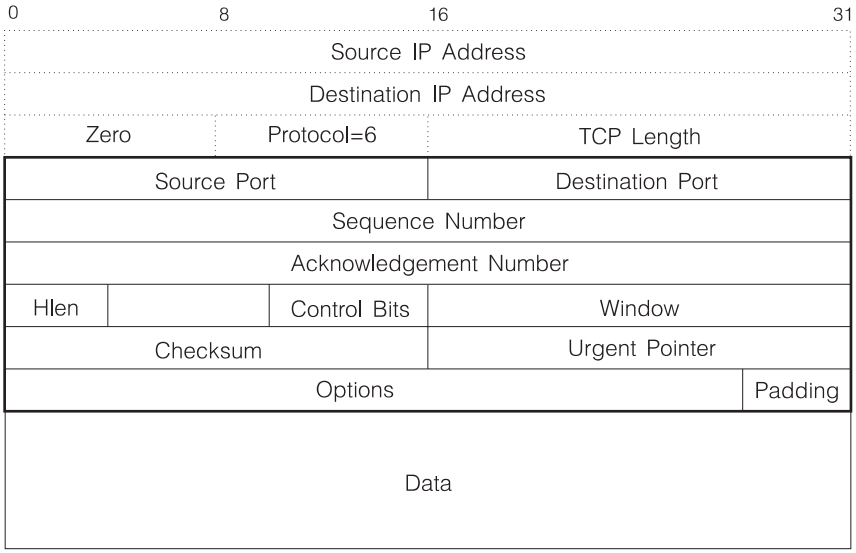
Protokol UDP je typicky využíván službami v lokálních sítích (ty mají relativně malou chybovost a nesetkáváme se se situací, kdy dojde ke změně pořadí paketů) a službami, které dávají přednost rychlosti doručení před požadavkem dodání dat bez chyb a v původním pořadí (například přenos zvuku, videokonference a některé měřící a řídící systémy).

2.4 Protokol TCP

Vážným problémem používání UDP protokolu je potřeba implementovat v aplikaci vlastní detekci chyb a případné potvrzovací schéma (pokud chceme omezit vliv chybovosti přenosu IP paketů), a vyrovnat se se zpracováním datagramů přicházejících v nesprávném pořadí. Výhodou může být používání protokolu, který uvedené funkce zabezpečí automaticky. Takovým protokolem je *TCP* (*Transmition Control Protocol*), který je definován v materiálech RFC 793 [7] a RFC 1122 [8] a který si nyní ve zkratce popíšeme.

TCP protokol na rozdíl od UDP nepřenáší data po blocích definovaných aplikací, zprávy předávané aplikací přebírá jako tok oktetů - *stream* a sám tento tok dělí na *TCP segmenty*. Ty odesílá v IP paketech, jejichž přenos zajišťuje okénkovým potvrzovacím schématem. Protokol nezachovává informaci o hranicích zpráv předávaných odesílatelem, o opětovné rozdělení přijatého toku oktetů na zprávy se musí postarat adresát. Z pohledu aplikace má TCP velice příjemné vlastnosti, realizované spojení se chová jako *plně duplexní virtuální kanál*, existence vnitřních mechanismů je ale zaplacena horším dynamickým chováním než má protokol UDP (to platí pro nepřetížený komunikační systém). Samozřejmou součástí protokolu TCP je zajištění *transportního multiplexu* mechanismem portů a socketů (budeme se mu věnovat v kapitole 3), důležitou součástí jsou vestavěné a relativně složité mechanismy *řízení toku*.

TCP protokol se od potvrzovacích protokolů, jak je známe z linkové vrstvy řady technologií, liší jediným společným formátem TCP segmentů, ten uvádí obr. 2.9:



Obr. 2.9: Struktura TCP segmentu

Pole *Source Port* a *Destination Port* identifikují aplikace na obou koncích TCP spojení a podporují multiplex. Pole *Sequence Number* je číslem prvního oktetu TCP segmentu, pole *Acknowledgement Number* uvádí číslo oktetu očekávaného jako prvý oktet TCP segmentu přenášeného opačným směrem. Pole *HLen* udává délku hlavičky TCP segmentu (včetně případného pole *Options*) v počtu 32-bitových slov. Pole *Control Bits* tvoří šest indikátorů:

- *URG* - přítomnost urgentních dat v TCP segmentu,
- *ACK* - platnost hodnoty v poli *Acknowledgement Number*,
- *PSH* - požadavek na bezprostřední předání TCP segmentu,
- *RST* - žádost o okamžité ukončení spojení (a uvolnění vyrovnávacích pamětí),
- *SYN* - otevírání spojení (synchronizaci klienta a serveru),
- *FIN* - uzavření spojení v jednom směru.

Šestnáctibitové pole *Window* podporuje řízení toku, toho si všimneme dále trochu podrobněji. Kontrolní součet *Checksum* (počítaný modulo 65535) zahrnuje stejně jako u UDP protokolu i údaje z hlavičky IP paketu - pseudohlavičku. Pole *Urgent Pointer* dovoluje označit místo, kde jsou uložena prioritně (mimo pořadí) předávaná data indikovaná příznakem *URG*. Konečně pole *Options* dovoluje nastavit parametry TCP spojení, například maximální délku TCP segmentu (jeho implicitní délka je 536 oktetů).

Potvrzovací mechanismus protokolu TCP se opírá o *pozitivní potvrzovací schéma*, ztráta TCP segmentu (nebo TCP segmentu, který obsahuje potvrzení a není následován dalším TCP segmentem) je překryta *časovým limitem* (timeout). Časový limit však v prostředí Internetu není možné nastavit pevně: u nepřetížených lokálních sítí může být velice malý a relativně konstantní, u rozsáhlých sítí závisí na zatížení i většího množství směrovačů a má podstatně větší střední hodnotu a hlavně velký rozptyl. Řešením, o které se TCP protokol v minulosti opíral, bylo adaptivní nastavování časového limitu *TimeOut*. Ten je nastavován na hodnotu

$$TimeOut = \beta.RTT_{\mathrm{i}},$$

kde RTT_{i} (Round Trip Time) je odhadnuté střední zpoždění získané jako

$$RTT_{\mathrm{i}} = (\alpha.RTT_{\mathrm{i-1}}) + ((1-\alpha).RTS),$$

kde RTS (Round Trip Sample) je konkrétně změřená hodnota doby mezi odesláním TCP segmentu a příjmem potvrzení, hodnota parametru β je obvykle v rozsahu $\beta = 1.3...2.0$, parametr α dovoluje nastavit rychlost reakce RTT_{i} na změny okamžitého zpoždění RTS. Mechanismus je startován s počátečními hodnotami $RTT_0 = 0s$ a $TimeOut = 3s$.

Metoda může selhat, pokud při překročení časového limitu dojde k opakování. V takovém případě neexistuje cesta, jak rozhodnout, zda se přijaté potvrzení vztahuje k originálnímu TCP segmentu nebo k jeho opakování. Řešením je doby přenosu pro opakované TCP segmenty ignorovat a současně prodloužit časový limit při každém opakovaném pokusu, například na dvojnásobek. Uvedený algoritmus je označován jako exponenciální prodlužování časového limitu (exponential back-off) nebo *Karnův mechanismus*.

Současná specifikace adaptivního algoritmu bere kromě zpoždění v úvahu i jeho rozpyl (ten při vyšší zátěži směrovačů roste rychleji než zpoždění samotné). *Jacobsonův mechanismus* vychází ze vztahů

$$Diff = RTS - RTT_{\mathrm{i-1}}$$

$$RTT_{\mathrm{i}} = RTT_{\mathrm{i-1}} - \delta.Diff$$

$$Dev_{\mathrm{i}} = Dev_{\mathrm{i-1}} + \delta.(|Diff| - Dev_{\mathrm{i-1}})$$

Optimální hodnotou parametru δ je (i s ohledem na rychlost výpočtu v pevné řádové čárce) $\delta = 0.125$.

Pro efektivní řízení toku je exponenciální prodlužování časového limitu po ztrátě TCP segmentu nebo potvrzení doplněno o exponenciální *zkracování okénka* potvrzovacího schématu. V poli *Window* zadává příjemce velikost své vyrovnávací paměti. Odesílatel bere v úvahu implicitní nebo dohodnutou maximální délku TCP segmentu, z ní vyplývá možnost odeslání určitého počtu TCP segmentů. Při překročení časového limitu odesílatel spočtený počet TCP segmentů snižuje na polovinu, až se může dostat k jedinému TCP segmentu.

Podstatným důvodem pro zmenšování okénka je i skutečnost, že běžné implementace TCP protokolu nemají schopnost selektivně indikovat chybějící TCP segment, případné další TCP segmenty vyslané v rámci okénka po segmentu poškozeném tedy musí být opakovány.

Mechanismus omezování toku musí být doplněn o opětovné uvolnění toku po zlepšení situace v síti a při startu spojení. Používaný mechanismus je označován jako *pomalý start* (Slow Start)

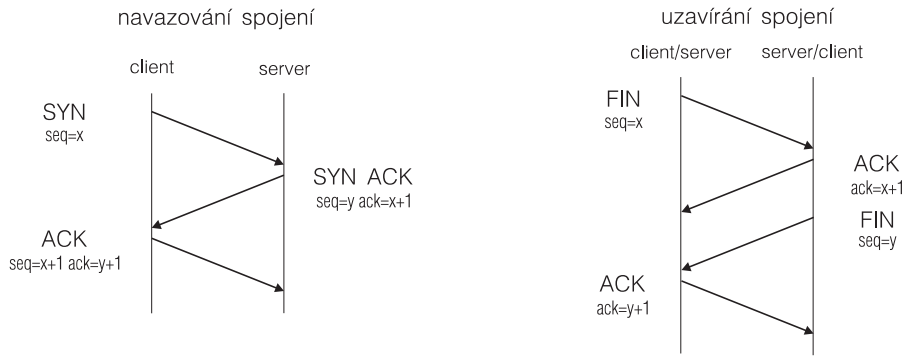
a spočívá v prodloužení okénka (zvýšení počtu odeslatelných TCP segmentů) o jedničku při každém příjmu potvrzení. Aby zvyšování toku nevedlo opětovně k přetížení, je mechanismus ještě doplněn o test, zda při zvětšování okénka došla všechna potvrzení; okénko je zvětšeno pouze v případě, že tomu tak je.

Za určitých okolností může mechanismus okénka vyvolat chování, které označujeme jako syndrom hloupého okénka (*Silly Window Syndrom*). Příjemce posouvá okénko po malých úsecích a tomu odpovídají i krátké odesílané TCP segmenty. Důsledkem je snížení efektivity přenosu. Ochranný mechanismus pracuje tak, že posunutí okénka indikuje až tehdy, když je podstatné (více než 25% jeho celkové šířky).

Stejně nepříjemné snížení efektivity způsobuje i přenos po jednotlivých znacích, se kterým se můžeme setkat například u Telnetu. Zde přináší zlepšení vypínatelný *Nagleův mechanismus*, který nedovoluje odeslat zkrácený TCP segment, dokud není předcházející segment potvrzen.

Vlastní TCP spojení jsou *asymetrická*, jedním z účastníků komunikace je *server*, který připraví svou datovou strukturu pro řízení spojení - *socket* a potom *pasivně* očekává navázání spojení od druhého účastníka komunikace. Tím je *klient*, který po přípravě svého socketu *aktivně* navazuje spojení s čekajícím serverem.

Navázání spojení má formu označovanou jako *třífázový hand-shake* (viz obr. 2.10). Tato technika dovoluje vyhnout se problémům, ke kterým by mohlo dojít při zduplikování aktivity serveru jako důsledku opakování TCP segmentů při pozdě přijatých potvrzeních.



Obr. 2.10: Navazování a uzavírání TCP spojení

O uzavření spojení může požádat libovolný z komunikujících partnerů. Výsledkem žádosti je uzavření spojení v jednom směru, v opačném směru žádá o uzavření spojení druhý partner. (Programové rozhraní dovoluje aplikaci uzavřít kanál v jediném směru nebo si vyžádat uzavření obousměrné.) Uzavírání spojení je poněkud problematičtější aktivitou než jeho otevírání. Připustíme-li možnost ztráty některého z TCP segmentů, které se uzavírání spojení zúčastní, ztrácíme jistotu v tom, že se obě strany na uzavření spojení dohodly. (Problém je označován jako *problém byzantských generálů*.) Nutností je po určitém počtu neúspěšných pokusů prohlásit spojení za ukončené a zrušení záznamů o daném spojení a uvolnění vyrovnávacích pamětí přenechat operačnímu systému.

Každé TCP spojení navázané mezi klientem a serverem je identifikováno následující pětic parametrů:

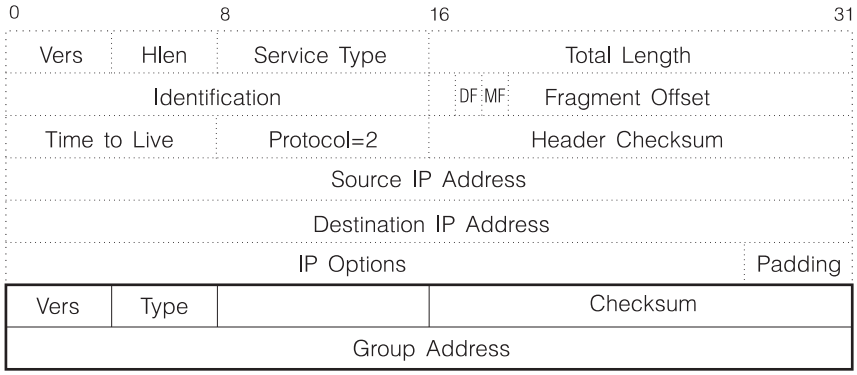
(protokol, adresa klienta, port klienta, adresa serveru, port serveru).

Takováto identifikace TCP spojení dovoluje na straně serveru, který očekává žádost o navázání spojení na portu s pevným číslem, realizovat vícenásobnou obsluhu a to i pro více klientů z téhož počítače. Jediným parametrem, ve kterém se identifikace obou TCP spojení liší, je pak číslo portu klienta, které je přidělováno dynamicky a jednoznačně.

2.5 Protokol IGMP

Protokoly UDP a TCP tvoří základ potřebný pro realizaci dvoubodových spojů (nebo, přesněji, pro realizaci spojů využívajících přímé adresace příjemce). Existují však situace, kdy je taková komunikace neefektivní, jako je otevírání více kanálů přenášejících totožná data k více počítačům vzdálené sítě . Typickým příkladem jsou multimediální přenosy, které poměrně silně zatěžují síť a navíc mají velké požadavky na kvalitu služby (doručování dat do časového limitu, omezený rozptyl dob přenosu). Pro takové aplikace je výhodné, podporuje-li síť komunikaci, kdy k přenosu dat dochází takovým způsobem, že pakety jsou duplikovány ve směrovačích, které tvoří uzly stromu cest mezi odesílatelem a příjemci.

Aplikační data jsou většinou přenášena ve formátu, který odpovídá běžnému UDP datagramu, s tím, že adresa příjemce je *adresou skupiny* (využívají se adresy třídy D, tedy IP adresy začínající kombinací 1110). O administraci skupin ve směrovačích se stará protokol *IGMP* (*Internet Group Management Protocol*). Ten se opírá o skupinovou adresaci a je popsán materiály RFC 1112 [9] a RFC 1122 [8]. Jeho pakety mají formát odpovídající obr. 2.11:



Obr. 2.11: Struktura paketu IGMP protokolu

Standardní hlavička IP paketu obsahuje v poli *Protocol* hodnotu indikující IGMP protokol. V polích IGMP datagramu najdeme informaci o verzi protokolu (pole *Vers*) a o typu paketu (pole *Type*). Pole *Group Address* označuje skupinu, které se IGMP paket týká, a kontrolní součet *Checksum* (počítaný modulo 65535) zajišťuje IGMP paket proti chybám.

Protokol IGMP dovoluje směrovačům dotazovat se připojených koncových zařízení na příslušnost ke konkrétním skupinám (tedy na jejich zájem přijímat IP pakety s konkrétní adresou skupiny) a konstruovat distribuční stromy. Techniky směrování, opírající se o různé metody vytváření distribučních stromů, a jim odpovídající standardy DVMRP (*Distance Vector Multicast Routing Protocol* - RFC 1075), MOSPF (*Multicast OSPF* - RFC 1584) a PIM (*Protocol Independent Multicast*), již leží mimo rozsah tohoto textu.

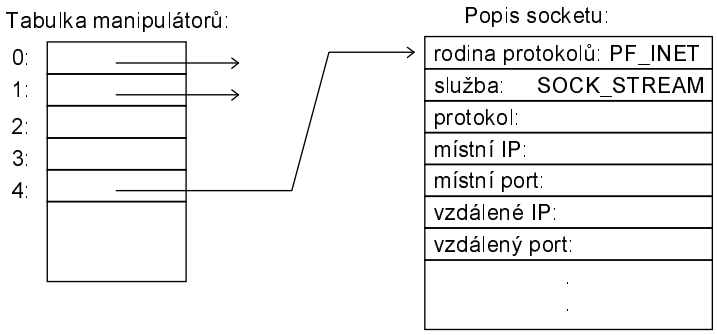
Typickou aplikací skupinové komunikace je distribuce různým způsobem kódovaných multimediálních signálů (zvuku a obrazu). Data jsou dnes s ohledem na časová omezení doručována UDP protokolem, do budoucna se počítá s využitím protokolů, které podporují stromovou distribuci dat a zahrnují i řízení toku, ale nemají podobně nepříjemné časové chování jako protokol TCP. Takovým protokolem je RSVP (*Resource ReSerVation Protocol* - RFC 2205). Data jsou, vedle informace o příslušnosti ke konkrétnímu zdroji signálu a o použitém kódování, doplněna informací o čase, která umožní rekonstrukci signálu na straně příjemce. Protokol, který práci s časově identifikovanými pakety datagramy umožňuje, je standardizován pod označením RTP (*Real Time Protocol* - RFC1889). Vedle vlastního přenosu multimediálních dat protokol podporuje i výstavbu *aplikačních bran*, které realizují převody použitých kódování a dovolují směšovat signály z více zdrojů.

3. Rozhraní BSD socketů

Rozhraní, známé jako *BSD sockety* rozšiřuje operační systém UNIX o přístup k síťovým službám. Bylo vytvořeno v rámci projektu ARPA na kalifornské universitě v Berkeley (UCB) a integrováno do systému BSD UNIX. O obdobné rozhraní byla doplněna i komerční větev operačního systému UNIX SVR3 pod názvem TLI (*Transport Layer Interface*). V dnešní době se setkáme s modifikací rozhraní BSD nejenom v UNIXu, ale i v systémech MS-Windows (knihovna winsock.dll), Novell-Netware a dalších.

Socketové rozhraní bylo navrženo pro protokoly TCP/IP, ale pro své výhodné vlastnosti se dnes využívá i pro další protokoly jako jsou DECnet (v systémech DEC VMS), SPX/IPX (v systémech Novell a Linux) a X.25 (v systému HP-UX).

Prototypy funkcí rozhraní a struktur socketů najdeme v hlavičkových souborech *sys/types.h* a *sys/socket.h*, případně i dalších. Vlastní funkce jsou obvykle uloženy ve standardní knihovně *libc*, na některých platformách ve zvláštní knihovně *libsocket*. Chyby při zpracování jsou indikovány po návratu z funkce, přičemž je lze zjistit v proměnné *errno* a případně vypsat chybové hlášení voláním funkce *perror()*.



Obr. 3.1: Tabulka manipulátorů pro proces

Základem pro identifikaci komunikačního spojení je manipulační číslo socketu (socket descriptor), dále uváděné stručně jako *socket*. Stejně jako manipulační číslo souboru (file descriptor) identifikuje konkrétní soubor po jeho otevření funkcí *open()*, manipulační číslo socketu otevřeného funkcí *socket()* identifikuje konkrétní síťové spojení. Obě manipulační čísla jsou indexy do společné tabulky, vlastní pro konkrétní proces. Volání funkce *socket()* má tvar:

```
int socket(int protocol_family, int service_type, int protocol);
```

Manipulační číslo odkazuje na popis socketu (obr. 3.1), který obsahuje informace o rodině protokolů (konstanta PF_* (protocol_family), např. PF_INET pro rodinu protokolů TCP/IP), typu služby (konstanta SOCK_* (service_type), např. SOCK_STREAM pro protokoly se spojením nebo SOCK_DGRAM pro protokoly bez spojení) a použitém protokolu (hodnota 0 vyjadřuje použití implicitního protokolu pro daný typ služby, může zde být i identifikátor konkrétního protokolu).

Další část popisu socketu obsahuje lokální a vzdálenou adresu ve formátu specifickém pro použitou rodinu protokolů. Obecný formát adresy je definován strukturou *sockaddr*. První položka určuje typ adresy (konstanta AF_* (address family), např. AF_INET pro IP adresu), druhá položka udává hodnotu adresy. Pro rodinu protokolů TCP/IP (rodina protokolů PF_INET) je adresa definována podrobněji jako struktura *sockaddr_in* s pevně nastaveným typem adresy AF_INET, hodnota je tvořena IP adresou a číslem portu.

```
struct sockaddr {
   u_short sa_family;                                   // rodina adres AF_*
   char sa_data[14];                                    // až 14 bytů protokolově závislé adresy
}
struct in_addr {
   u_long s_addr;                                       // 32-bitová IP adresa (uložena v síťovém formátu)
}
struct sockaddr_in {
   u_short sin_family;                                  // vždy AF_INET
   u_short sin_port;                                    // 16-bitová adresa portu (uložena v síťovém formátu)
   struct in_addr sin_addr;                             // IP adresa
   char sin_zero[8];                                    // nevyužito
}
```

Lokální adresu váže na socket obvykle funkce *bind()*. Pokud není funkce *bind()* volána explicitně, mohou ji zastoupit funkce pro otevření spojení, které pak přiřadí implicitní lokální adresu (tento režim je označován jako *autobind*). V takovém případě je implicitní lokální adresa určena jako adresa síťového rozhraní, které odesílá pakety k adresátovi. (Toto řešení má příjemný důsledek; nemusíme se starat o výběr správné IP adresy, je-li náš počítač připojen do více podsítí. Podobného výsledku dosáhneme u funkce *bind()*, pokud do položky *sin_addr* struktury *sockaddr_in* uložíme konstantu INADDR_ANY.) Pokud využijeme automatického přiřazení adresy nebo zadáme jako číslo portu *sin_port* nulu, bude socketu přidělen první volný port v systému (v oblasti čísel >=1024). Hlavička funkce *bind()* má tvar:

```
int bind(int sockfd, struct sockaddr *my_addr, int my_addr_len);
```

Vzdálená adresa je nastavována funkcemi, které otevírají spojení v případě spojované služby TCP aktivně - *connect()* nebo pasivně - *accept()*. V případě nespojované služby UDP je vzdálená adresa nastavována funkcemi pro odeslání a příjem datagramu *sendmsg(), sendto(), recvmsg(), recvfrom()*. Zjištění lokální a vzdálené adresy vázané na socket podporují funkce *getsockname()* a *getpeername()*:

```
int getsockname(int sockfd, struct sockaddr *name, int *namelen);
int getpeername(int sockfd, struct sockaddr *name, int *namelen);
```

Pro vytvoření dvojice provázaných socketů v rámci jednoho systému lze použít funkci *socketpair()*. Implementace je většinou omezena na třídu PF_UNIX (UNIX domain sockets, UNIX internal protocols). Hlavička funkce *socketpair()* má tvar:

```
int socketpair(int protocol_family, int service_type, int protocol, int sockfd[2]);
```

Pro nastavení protokolově nezávislých parametrů socketů lze použít funkce *ioctl()* a *fcntl()*. Protokolově závislé parametry socketů lze nastavit a jejich hodnoty získat funkcemi *setsockopt()* a *getsockopt()*. Vrátíme se k nim v kapitole 3.2.4.

Socket uvolňujeme voláním funkce *close()*:

```
int close(int sockfd);
```

Všechny sockety (stejně jako souborové manipulátory) jsou uvolňovány automaticky při ukončení programu.

3.1 Adresace

Pomocné adresační funkce lze rozdělit do tří oblastí: na oblast databázových funkcí *get*by*()*, na oblast manipulace s IP adresou *inet_*()* a na makra pro převod celých čísel používaných v počítači na síťový formát/notaci a obráceně **to*()*.

Databázové funkce využívají při své práci jmenné služby. Jmennými službami rozumíme souborovou jmennou službu (soubory v /etc/* pro UNIX, sys:\etc* pro Novell Netware, \windows* pro Windows95, \windows\system\etc* pro WindowsNT), nebo *adresářové systémy* DNS, NIS, NIS+, CDS, NDS a další. Využití jmenných služeb závisí na konkrétní databázové funkci a je i implementačně závislé (u starších verzí UNIXu). Nové verze UNIXu nabízejí možnost předdefinování pořadí prohledávání jmenných služeb v souboru /etc/nsswitch.conf.

Prototypy databázových funkcí najdete v hlavičkových souborech *netdb.h*. Funkce patří mezi blokující, tj. pozastaví běh programu na čas potřebný k získání odpovědi případně k určení chyby. Pokud se používají adresářové systémy (např. služba DNS), pozastavení může trvat i několik desítek sekund.

Vyhledání IP jména nebo adresy se opírá o funkce *gethostbyaddr()* resp. *gethostbyname()*. Jako jmenné služby lze použít soubor /etc/hosts, adresářový systém DNS, NIS, nebo NIS+.

```
struct hostent *gethostbyname(char *name);
struct hostent *gethostbyaddr(char *addr, int length, int addrtype);

struct hostent {
    char *h_name;                                   // oficiální jméno
    char **h_aliases;                               // další jména
    int h_addrtype;                                 // typ adresy (konstanty AF_*)
    int h_length;                                   // délka adresační struktury
    char **h_addr_list;                             // výčet adres
};
#define h_addr h_addr_list[0]          // pro snadnější přístup k první položce a kompatibilitu
```

Čísla a jména sítí lze vyhledat pomocí funkcí *getnetbyname()* a *getnetbyaddr()*. Některé systémy využívají při hledání jen část adresy určující síť (neúplná IP adresa), jinde je třeba vynulovat část určující koncové zařízení. Jako jmenné služby lze použít soubor /etc/networks, adresářový systém DNS, NIS, nebo NIS+.

```
struct netent *getnetbyname(char *name);
struct netent *getnetbyaddr(long net, int type);

struct netent {
    char *n_name;                                   // oficiální jméno sítě
    char **n_aliases;                               // další jména
    int n_addrtype;                                 // typ adresy
    long n_net;                                     // číslo sítě
};
```

Pro vyhledání čísla a jména protokolu lze použít funkcí *getprotobyname()* a *getprotobynumber()*. Jako jmenné služby lze použít soubor /etc/protocols, adresářový systém NIS nebo NIS+.

```
struct protoent *getprotobyname(const char *name);
struct protoent *getprotobynumber(int proto);

struct protoent {
    char *p_name;                                   // oficiální jméno protokolu
    char **p_aliases;                                        // další jména
    int p_proto;                                             // číslo protokolu
}
```

Pro vyhledání služeb (*well-known portů*) se používají funkce *getservbyname()* a *getservbyport()*. Jako jmenné služby lze použít soubor /etc/services, adresářový systém NIS nebo NIS+.

```
struct servent *getservbyname(const char *name, const char *proto);
struct servent *getservbyport(int port, const char *proto);

struct servent {
    char *s_name;                                          // oficiální jméno služby
    char **s_aliases;                                               // další jména
    int s_port;                                                     // číslo portu
    char *s_proto;                  // jméno protokolu, který se má používat s touto službou
}
```

Další skupina funkcí a maker provádí jednoduché manipulace s IP adresou. Prototypy těchto funkcí naleznete v hlavičkovém souboru souboru *<arpa/inet.h>*.

```
unsigned long int inet_addr(const char *cp);          // převod z tvaru "x.y.z.w" na adresu
unsigned long int inet_network(const char *cp);    // převod z tvaru "x.y.z.w" na adresu sítě
char *inet_ntoa(struct in_addr in);                    // převod adresy na tvar "x.y.z.w"
struct in_addr inet_makeaddr(int net, int host);     // vytvoří adresu z čísla počítače a sítě
unsigned long int inet_lnaof(struct in_addr in);          // vrátí číslo počítače z adresy
unsigned long int inet_netof(struct in_addr in);            // vrátí číslo sítě z adresy
```

Poslední skupina funkcí převádí celá čísla z formátu využívaného počítačem do formátu síťového a naopak. Tyto funkce musí být použity, pokud nastavujeme číslo portu a IP adresu struktury *sock_addr_in*. Sítový formát celých čísel odpovídá zobrazení *big-endian* (do paměti jsou na nižší adresy ukládány vyšší řády čísla). Makra jsou proto např. pro počítače založené na procesorech SPARC prázdná, pro procesory Intel x86 přehazují pořadí bytů. Prototypy maker nalezneme v souboru *<netinet/in.h>*.

```
// "h" (host) představuje reprezentaci uvnitř počítače , "n" (network) síťovou reprezentaci
// varianty jsou pro "l" long a "s" short int
unsigned long int htonl(unsigned long int hostlong);
unsigned short int htons(unsigned short int hostshort);
unsigned long int ntohl(unsigned long int netlong);
unsigned short int ntohs(unsigned short int netshort);
```

Následující příklad ilustruje použití adresačních funkcí. Dovoluje zjistit jména a adresy zadaného počítače, jména zadaného protokolu a služby a připraví socket pro komunikaci.

```
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/ioctl.h>
struct hostent *ph;                                          // naše pracovní proměnné
struct servent *ps;
struct protoent *pp;
unsigned int inaddr, i;
//------------------------------------ zjištění IP adresy
   if ((inaddr=inet_addr(argv[1]))!=INADDR_NONE)
      ph=gethostbyaddr((char *)&inaddr, sizeof(unsigned int), AF_INET);  // zkusíme přes číslo
   else ph=gethostbyname(argv[1]);                                  // zkusíme přes jméno
   if (ph) {                                                         // vypíšeme výsledky
      printf("\nHostname: %s\n",ph->h_name);                          // jméno počítače
      for (i=0; ph->h_aliases[i]; i++) printf(" %s\n", ph->h_aliases[i]);     // další jména
      printf("Type: %i\nLen: %i\n", ph->h_addrtype, ph->h_length);     // vypíšeme IP adresy
      for (i=0; ph->h_addr_list[i]; i++)
         printf("Addr: %i.%i.%i.%i\n",
         (unsigned char)(ph->h_addr_list[i])[0],
         (unsigned char)(ph->h_addr_list[i])[1],
         (unsigned char)(ph->h_addr_list[i])[2],
         (unsigned char)(ph->h_addr_list[i])[3]);
   }
   else
   { herror("Chyba pri zjisteni adresy - neni registrovana ve jmenne sluzbe"); exit(10); }
//------------------------------------ zjištění protokolu
   if (isdigit(argv[3][0]))
      pp=getprotobynumber(atoi(argv[3]));                           // zkusíme přes číslo
   else pp=getprotobyname(argv[3]);                                 // zkusíme přes jméno
   if (pp) {                                                         // vypíšeme výsledky
      printf("\nProtocol: %s\n", pp->p_name);                                  // jméno
      for (i=0; pp->p_aliases[i]; i++) printf(" %s\n", pp->p_aliases[i]);     // další jména
      printf("Num: %i\n", pp->p_proto);                              // a číslo protokolu
   }
   else
   { printf("Chyba pri zjisteni protokolu\n"); exit(11); }
//------------------------------------ zjištění služby
   if (isdigit(argv[2][0]))
      ps=getservbyport(htons((u_short)atoi(argv[2])), pp->p_name);
   else ps=getservbyname(argv[2], pp->p_name);
   if (ps) {                                                         // vypíšeme výsledky
      printf("\nServname: %s\n",ps->s_name);
      for (i=0; ps->s_aliases[i]; i++)
         printf(" %s\n", ps->s_aliases[i]);
      printf("Port: %hi\nProto: %s\n", ntohs((u_short)ps->s_port), ps->s_proto);
   }
   else
   { printf("Chyba pri zjisteni sluzby\n"); exit(12); }
```

```
//————————————————————————————— otevření socketu
if (strcmp(pp->p_name,"tcp"))
   sockfd = socket(PF_INET, SOCK_DGRAM, pp->p_proto));
else
   sockfd = socket(PF_INET, SOCK_STREAM, pp->p_proto));
rem_addr.sin_family = AF_INET;
rem_addr.sin_port = ps->s_port;
bcopy(ph->h_addr, (char *) &rem_addr.sin_addr, ph->h_length);
```

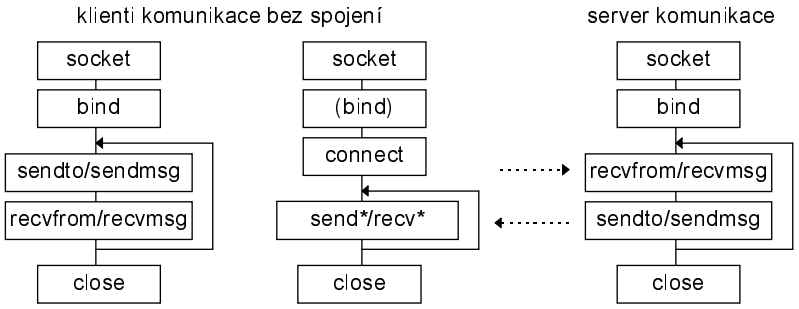
3.2 Komunikace klient – server

Po seznámemí se s datovými strukturami socketů a adres nyní přejdeme k programování aplikací. Nejdříve se seznámíme s principy práce klienta a serveru. Budeme se věnovat jednoduchým ukázkám komunikace UDP a TCP a uvedeme si konkrétní řešení *iterativního* TCP serveru, který obsluhuje klienty sekvenčně. Později si všimneme složitějších implementací serverů, které dovolují souběžně obsluhovat více klientů a které obvykle označujeme jako *paralelní* (tomuto termínu dáme v dalším textu přednost) nebo *konkurentní*.

Všechny uváděné příklady zajišťují jednoduchou komunikaci - službu *echo*; klient zašle zprávu serveru a ten ji beze změny (jako ozvěnu) vrátí klientovi. Pro zjednodušení naše příklady neobsahují převody jmen, adres, čísel portů a služeb, jak byly uvedeny v předchozím příkladu.

3.2.1 Komunikace bez spojení (protokol UDP)

Pro datagramovou komunikaci bez navazování spojení (kdy používáme UDP protokol) využíváme již popsané funkce v sekvenci odpovídající obrázku 3.2. Strana klienta a serveru se liší posloupností jejich volání a parametry použitými při inicializaci. Socket otevřeme, stejným způsobem na straně serveru i na straně klienta, pro rodinu protokolů PF_INET a typ služby SOCK_DGRAM. Potřebné číslo protokolu můžeme zjistit pomocí jmenných služeb nebo dosadíme nulu; pak bude použit implicitní protokol (v našem případě UDP).



Obr. 3.2: Typické posloupnosti volání funkcí při datagramovou komunikaci

V dalším se už klient a server liší. Klient obvykle nenastavuje lokální IP adresu a číslo portu (použije konstantu INADDR_ANY a nulovou hodnotu) ve volání funkce *bind()*. Naopak u serveru je nutné nastavit adresu portu na zvolenou hodnotu. Vzdálenou adresu nastavujeme ve funkcích pro zaslání datagramu *sendmsg()*, *sendto()*. Při příjmu lze vzdálenou adresu zjistit ve výstupních parametrech funkcí *recvmsg()*, *recvfrom()*. Funkce vrací jako výsledek počet odeslaných resp. přijatých znaků, hodnota -1 indikuje chybu. Mezi užitečné příznaky (*flags*) při volání funkcí pro příjem patří MSG_PEEK (datagram nebude vyzvednut ze vstupu a může být

později opět čten). Funkce pracují obvykle v blokujícím režimu, tj. proces nepokračuje, dokud nebude datagram odeslán nebo přijat. Používané funkce se liší v seznamu parametrů a jejich hlavičky jsou uvedeny na následujících řádcích:

```
int sendto(int sockfd, const void *buf, int buf_len, unsigned int flags,
   const struct sockaddr *to, int to_len);
int sendmsg(int sockfd, const struct msghdr *msg, unsigned int flags);
int recvfrom(int sockfd, void *buf, int buf_len, unsigned int flags,
   struct sockaddr *from, int *from_len);
int recvmsg(int sockfd, struct msghdr *msg, unsigned int flags);

struct msghdr {
   caddr_t msg_name;                          // nepovinná cílová adresa
   u_int msg_name_len;                        // a její délka
   struct iovec *msg_iov;     // tabulka s odkazy na datové úseky (pro funkce readv/writev)
   u_int msg_iovlen;                          // množství dat
   caddr_t msg_control;                       // doplňková data pro příslušný protokol
   u_int msg_controllen;                      // velikost příslušné paměti pro tato data
   int msg_flags;                             // příznaky přijaté zprávy
};
```

Alternativní možností je použití funkce *connect()*. Při použití této funkce nedojde k vytvoření spojení (používáme datagramovou službu), ale bude nastavena vzdálená adresa. Pokud před voláním funkce *connect()* nebyla volána funkce *bind()*, je proveden "autobind" - IP adresa a číslo portu jsou přiděleny implicitně. Funkcí *connect()* jsou doplněny všechny potřebné údaje pro přenos a můžeme využít zjednodušeného volání funkcí z předchozího odstavce - odkaz na vzdálenou adresu nastavujeme na NULL. Další možností je možnost použít funkce *recv()*, *send()* nebo běžné funkce pro čtení a zápis *read()*, *write()*.

```
int recv(int sockfd, void *buf, int buf_len, unsigned int flags);
int send(int sockfd, const void *buf, int buf_len, unsigned int flags);
```

Je třeba upozornit, že při každém vysílání a příjmu je přijat právě jeden datagram. Na datagramovou službu se neváže žádná obsluha vyrovnávacích pamětí; pokud je při příjmu datagram delší než vyhrazená přijímací paměť, je zkrácen, pokud datagram není vyzvednut funkcí *recv*()* včas, bude přepsán při příjmu následujícího datagramu.

Ukázkou jednoduché UDP komunikace je následující příklad. Nejdříve si uvedeme klienta komunikace, který odešle textovou zprávu na server komunikující protokolem UDP dostupný na zadané IP adrese a zadaném portu a na standardní výstup vypíše přijatou odpověď:

```
main(int argc, char *argv[])
{
   struct sockaddr_in my_addr, rem_addr;                  // adresační struktury
   int sockfd;                                            // číslo manipulátoru pro socket
   int len, rem_addr_length;                              // délka zprávy resp. adresační struktury
   char sbuf[80], rbuf[80];
   if ((sockfd=socket(PF_INET, SOCK_DGRAM, 0))<0)         // otevíráme socket
      { perror("socket nelze otevrit"); exit(1); }        // (pro implicitní protokol UDP)
   bzero((char *)&my_addr,sizeof(my_addr));               // vyplnění lokální adresy nulami
   my_addr.sin_family=AF_INET;                            // adresace
   my_addr.sin_addr.s_addr=INADDR_ANY;                    // IP adresu přidělí jádro
```

```
    my_addr.sin_port=0;                                          // i port přidělí jádro
    if (bind(sockfd, (struct sockaddr *) &my_addr, sizeof(my_addr)) < 0)  // vážeme se socketem
       { perror("Chyba pri bind"); close(sockfd); exit(2); }
    bzero ((char *) &rem_addr, sizeof(rem_addr));                // vzdálená adresa
    rem_addr.sin_family=AF_INET;
    rem_addr.sin_port=htons(atoi(argv[2]));                  // port přidělíme podle parametru
    rem_addr.sin_addr=inet_addr(argv[1]);                // IP adresu přidělíme podle parametru
    strcpy(sbuf,"Testovaci zprava");                             // připravíme zprávu
    len=strlen(sbuf);
    if ((len=sendto(sockfd, sbuf, len, 0, (struct sockaddr *)&rem_addr, sizeof(rem_addr)))<0)
       { perror("Chyba ve vysilani"); close(sockfd); exit(3); }              // odešleme
    rem_addr_length=sizeof(rem_addr);    // při příjmu zprávy musí být vložena velikost adr. struktury
    if ((len=recvfrom(sockfd, rbuf, sizeof(rbuf), 0, (struct sockaddr *)&rem_addr,
            &rem_addr_length))<0)                                // přijmeme odpověď
       { perror("Chyba pri prijmu"); close(sockfd); exit(4); }
    else
       { rbuf[len]=(char) 0; printf("%s\n",rbuf); }                  // a vypíšeme ji
    close(sockfd);                                               // ukončíme program
}
```

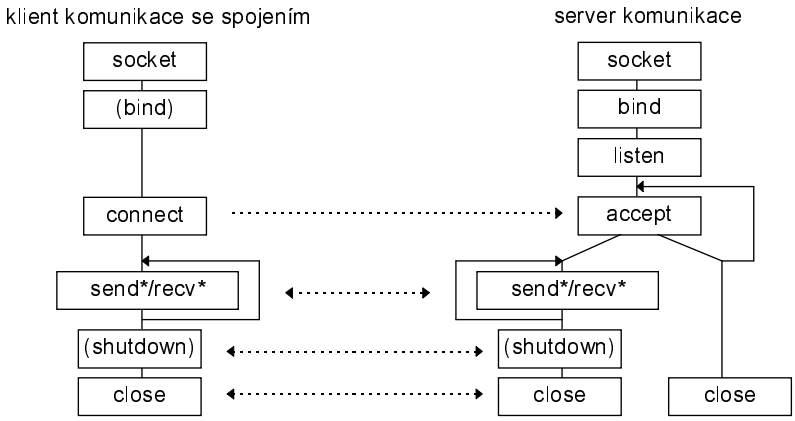
Za zmínku stojí nutnost nastavit délku vyrovnávací paměti *rem_addr_length* před voláním funkce *recvfrom()*.

Server spolupracující s uvedeným klientem protokolem UDP je vázaný na zvolený port, přijímá zprávu a v původním tvaru ji odesílá zpět klientovi. Má tvar:

```
main(int argc, char *argv[])
{
    struct sockaddr_in my_addr, rem_addr;                                // IP adresy
    int sockfd;                                              // číslo manipulátoru pro socket
    struct hostent *hp;                                  // pro zjištění klienta (dotaz podle IP)
    int i, len, rem_addr_length;
    char rbuf[80];                                                   // vyrovnávací paměť
    if ((sockfd=socket(PF_INET, SOCK_DGRAM, 0))<0)                       // otevření socketu
       { perror("Socket nelze otevrit"); exit(1); }
    bzero((char *)&my_addr, sizeof(my_addr));                            // lokální adresa
    my_addr.sin_family=AF_INET;
    my_addr.sin_addr.s_addr=INADDR_ANY;
    my_addr.sin_port=htons(atoi(argv[1]));                   // přidělíme port podle parametru
    if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(my_addr))<0)    // vážeme se socketem
       { perror("Chyba pri bind"); close(sockfd); exit(2); }
    rem_addr_length=sizeof(rem_addr);                                // připravíme se na příjem
    if ((len=recvfrom(sockfd, rbuf, sizeof(rbuf), 0, (struct sockaddr *)&rem_addr,
            &rem_addr_length))<0)                                    // přijmeme zprávu
       { perror("Chyba pri prijmu"); close(sockfd); exit(3); }
    if ((hp=gethostbyaddr((char *)&rem_addr.sin_addr, sizeof(rem_addr.sin_addr),
            AF_INET))!=NULL)
    { rbuf[len]=(char)0; printf("Prijato od %s: %s\n", hp->h_name, rbuf); }
    if ((len=sendto(sockfd, rbuf, len, 0, (struct sockaddr *)&rem_addr, sizeof(rem_addr)))<0)
       { perror("Chyba ve vysilani\n"); close(sockfd); exit(4); }            // zašleme zpět
    close(sockfd);                                                   // ukončíme program
}
```

3.2.2 Komunikace se spojením (protokol TCP)

U spolehlivé komunikace se spojením (používáme protokol TCP) navazujeme spojení a přenášíme data sekvencí volání funkcí podle obrázku 3.3. Socket otevíráme pro rodinu



Obr. 3.3: Typické volání funkcí při komunikaci se spojením

protokolů PF_INET, typ služby SOCK_STREAM, číslo protokolu můžeme zjistit pomocí jmenných služeb nebo dosadíme nulu (TCP je implicitním protokolem pro tuto rodinu a typ služby).

Klient obvykle nenastavuje lokální IP adresu a číslo portu voláním funkce *bind()* , častěji se využívá automatický "autobind" při navazování spojení. Funkce *connect()* zablokuje proces až do potvrzení spojení serverem. Pokud není cílový počítač dostupný nebo na něm není spuštěn požadovaný server, funkce indikuje chybu. Vytvořené spojení je plně obousměrné a jsou ošetřeny všechny problémy vyplývající z použití IP vrstvy. Pro zefektivnění přenosu je pro potvrzování použito okénkové potvrzování (sliding windows), řízení rychlosti přenosu dat (flow control) a další optimalizační mechanismy. Uzavření spojení může být jednosměrné i obousměrné.

```
int connect(int sockfd, struct sockaddr *serv_addr, int serv_addr_len);
int shutdown(int sockfd, int how);          // how=0 ukončuje příjem, 1 vysílání, 2 obousměrné
```

Pro přenos dat lze využít všechny funkce uváděné v předchozím odstavci. Pro komunikaci po vytvořeném spojení lze použít funkce *recv(), send(), read(), write()*. Při zasílání jednotlivých bloků dat si musíme uvědomit, že se jedná o nestrukturovaný tok dat (stream flow). Není zaručeno, že odesílaná data budou vložena do jednoho IP paketu (uplatní se vstupní a výstupní vyrovnávací paměť u klienta resp. serveru). Při čtení dat se nečeká na načtení celého bloku, ale funkce se vrací po uplynutí časového limitu i s částí načtených dat (příslušné hodnoty lze ovlivnit nastavením parametrů socketu - viz str. 36). Pro snadnější vyzvednutí celého bloku dat lze použít ve funkci *recv()* s příznakem MSG_WAITALL, kdy se vyčká, až bude přijato potřebné množství dat. Dalším doplňkem je možnost zaslat prioritní data mimo pořadí (out of band), kdy se vyhnou vyrovnávací paměti. Vyslání prioritních dat resp. schopnost přijmout prioritní data je nutné vyznačit příznakem MSG_OOB ve funkcích *send()* resp. *recv()*.

U serveru obvykle nastavujeme číslo portu funkcí *bind()*. Funkce *listen()* vytvoří v systému frontu požadavků klientů (o zadané délce *backlog*) na navázání spojení. Jednotlivé žádosti o spojení jsou přebírány funkcí *accept()*. Tato funkce vytvoří nový socket pro příslušné spojení (zacházíme s ním nezávisle), na původním socketu můžeme opět zavolat funkci *accept()* a

obsloužit další požadavek. Při překročení limitu na počet neobsloužených požadavků jsou nová spojení odmítána.

```
int listen(int sockfd, int backlog);
int accept(int sockfd, struct sockaddr *rem_addr, int *rem_addrlen);
```

Funkce *accept()* otevřením nového socketu podporuje implementaci paralelní obsluhy více žádostí. Zde si však uvedeme příklad jednoduchého iterativního serveru, který přebírá další žádost až po ukončení obsluhy žádosti předchozí a souběžné obsluhy obou socketů nevyužívá. Podobně jako u protokolu UDP začneme ukázkou jednoduchého TCP klienta:

```
#define BUFFSIZE 2048
main(int argc, char *argv[])
{
   char buf[BUFFSIZE];                                          // data pro příjem
   int sockfd, len, off;
   struct sockaddr_in rem_addr;                                 // adresační struktura
   if ((sockfd=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP))<0)      // otevíráme socket
      { perror("Socket nelze otevrit"); exit(1); }              // (pro protokol TCP)
   bzero((char *)&rem_addr, sizeof(rem_addr));                  // vyplnění vzdálené adresy
   rem_addr.sin_family=AF_INET;                                 // (lokální je "autobind" v connect)
   rem_addr.sin_port=htons(atoi(argv[2]));
   rem_addr.sin_addr.s_addr=inet_addr(argv[1]);
   if (connect(sockfd, (struct sockaddr *)&rem_addr, sizeof(rem_addr))==-1)   // spojení
      { perror("Chyba pri connect"); close(sockfd); exit(1); }
   if (write(sockfd, buf, BUFFSIZE) != BUFFSIZE)                // zašleme data
      { perror("Chyba pri vysilani"); close(sockfd); exit(2); }
   off=0;                                                       // příjem dat (přijímáme nestrukturovaný tok)
   do {
      if((len=read(sockfd, buf+off, BUFFSIZE-off))<=0)  // čteme dostupná data do volné paměti
         { perror("Chyba pri prijmu"); close(sockfd); exit(3); }
      printf("Prijato: %i\n", len);
      off+=len;
   }
   while (off!=BUFFSIZE);                                       // dokud nepřijmeme vše
            // do-while cyklus lze nahradit voláním recv(sockfd, buf, BUFFSIZE, MSG_WAITALL);
   close(sockfd);                                               // ukončíme program
}
```

Nyní si uvedeme server komunikace. Iterativní server dokáže komunikovat pouze s jedním klientem (jedno navázané spojení v daném okamžiku) a uchovává si frontu maximálně *backlog* požadavků.

```
#define BUFFSIZE 1000          // zkusíme jinou velikost vyrovnávací paměti pro názornější práci TCP
int main(int argc, char *argv[])
{
   char buf[BUFFSIZE];
   int sockfd, c_sockfd;                    // manipulátor pro příjem spojení a samotné spojení
```

```
    struct sockaddr_in my_addr, rem_addr;                          // adresační struktury
    int rem_addr_length;
    int len;                                                       // délka přijaté zprávy
    if ((sockfd=socket(PF_INET, SOCK_STREAM, IPPROTO_TCP))<0)
       { perror("Soket nelze otevrit"); exit(1); }
    bzero(&my_addr, sizeof(my_addr));                              // lokální adresa
    my_addr.sin_family=AF_INET;
    my_addr.sin_addr.s_addr=INADDR_ANY;
    my_addr.sin_port=htons(atoi(argv[1]));
    if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(my_addr)) == -1)
       { perror("Chyba pri bind"); close(sockfd); exit(2); }
    if (listen(sockfd, 5)==-1) // začneme poslouchat na příslušném portu (fronta max. 5 požadavků)
       { perror("Nelze provest listen"); close(sockfd); exit(3); }
    while (1) {                                        // nekonečná smyčka pro obsluhy požadavků
       rem_addr_length=sizeof(rem_addr);               // připravíme se na příjem požadavku
       if ((c_sockfd=accept(sockfd, (struct sockaddr *)&rem_addr, &rem_addr_length))==-1)
       { perror("Nelze akceptovat požadavek"); close(sockfd); exit(4); }   // požadavek
       if ((len=recv(c_sockfd, buf, BUFFSIZE, 0))==-1)                     // přijmeme data
          perror("Chyba pri cteni");                   // nelze přijmout - uzavřeme spojení
       else
          while (len) {                                                    // dokud jsou přečtená data
             if (send(c_sockfd, buf, mlen, 0)==-1)                         // zašleme zpět
                { perror("Chyba pri zapisu"); break; }   // nelze zaslat - uzavřeme spojení
             printf("%i byte zkopirovano\n", mlen);
             if ((len=recv(c_sockfd, buf, BUFFSIZE, 0))==-1)               // další data
                { perror("Chyba pri cteni"); break; }    // nelze přijmout - uzavřeme spojení
          }
       close(c_sockfd);          // uzavřeme konkrétní spojení a opět jdeme na příjem požadavku
    }
}
```

3.2.3 Paralelní zpracování

V předchozích příkladech jsme se seznámili se základní iterativní komunikací klient–server. V těchto příkladech server zpracovával pouze jeden datagram (UDP), případně obsluhoval pouze jedno spojení (TCP) současně. Tato kapitola uvádí příklady umožňující souběžné zpracování více požadavků. Podobně lze upravit i klienta a dovolit mu souběžně komunikovat s více servery, ale takové řešení není příliš časté, naše příklady souběžného/paralelního zpracování se proto omezí na stranu serveru.

Jestliže je na straně serveru požadováno zpracovávání více požadavků od různých klientů (případně více požadavků od téhož klienta), musíme si uvědomit zda:

- je možné dopředu určit maximální počet požadavků,
- chceme omezit z hlediska výkonnosti serveru počet požadavků,
- můžeme vložit zpracování do stejného procesu nebo je nutné spustit samostatný proces,
- jaký je požadavek na rychlost reakce serveru na požadavek klienta.

Podle váhy těchto požadavků můžeme použít některý ze základních typů serverů:

- **iterativní** - zpracováván je pouze jeden požadavek, rychlost reakce na něj je vysoká. Řešení je velmi jednoduché a techniku lze využít jak pro třídu SOCK_DGRAM (3.2.1) tak i pro SOCK_STREAM (3.2.2).

- **paralelní s předpřipravenými procesy** - je spuštěn určitý počet procesů. Každý z procesů dokáže zpracovávat jeden požadavek. Techniku lze použít pro obě třídy komunikace, výhodou je rychlá reakce na příchod požadavku.

- **paralelní se společným zpracováním** - pokud lze obsluhovat více klientů v rámci jednoho procesu, můžeme vytvořit server, který bude zpracovávat i více požadavků souběžně. Výhodou tohoto typu serveru je rychlá reakce na příchod požadavku, volnost v počtu obsluhovaných událostí, možnost vzájemného provázání procesů přes globální proměnné. Musíme se ale omezit na obsluhu krátkých požadavků od jednotlivých klientů. Technika se používá pro obě třídy komunikace.

- **paralelní s dynamickým vytvářením procesů** - při příchodu požadavku je spuštěn nový proces pro obsluhu požadavku. Výhodou je šetření zdrojů počítače, když nevíme, kolik souběžných požadavků bude třeba obsloužit. Tento server lze snadno rozšířit o spouštění vnějších procesů, kterým bude předán socket, obvykle jako standardní vstup a výstup. Vnější proces může pracovat i jako samostatný program, lze tak obsluhovat i velmi odlišné typy klientů. Problémem je zpoždění při spouštění procesu. Technika se používá především pro třídu SOCK_STREAM, pro obsluhu datagramů u třídy SOCK_DGRAM je režie spouštění procesů příliš vysoká.

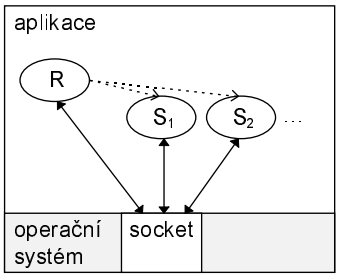
- **paralelní s předběžným dynamickým vytvářením procesů** - tato technika kombinuje určitý počet předpřipravených procesů, který je postupně doplňován o další dynamicky vytvářené tak, aby při příchodu požadavku byl pokud možno k dispozici volný proces. Techniku lze pak využít i pro obsluhu datagramů u třídy SOCK_DGRAM.

V následujících příkladech se podíváme podrobněji na jednotlivé typy paralelních serverů. Ke spouštění procesů se v příkladech používá funkce *fork()*, v novějších operačních systémech lze využít stejně dobře i knihovnu vláken (threads).

Paralelní UDP server s předpřipravenými procesy

Pokud má více procesů využívat stejný port, narazíme na problém ochrany použitého portu. Jestliže jeden proces zavolá *bind()*, jiný proces již nemůže tento port použít (za některých podmínek lze tuto ochranu zrušit). Tuto překážku překonáme, pokud socket vytvoříme v rodičovském procesu. Synovské procesy vytvořené funkcí *fork()* dědí všechny manipulátory, zdědí tedy i tento socket a pro příjem datagramů ze socketu mohou přímo použít funkce *recv*()*. Přijatý datagram obdrží ten synovský proces, který je zablokován funkcí *recv*()* a dovoluje spuštění (je naplánován). Způsob sdílení socketu mezi rodičovským procesem R a procesy synovskými S je znázorněn na obrázku 3.4.

Následující příklad serveru zpracovává v hlavním procesu i v podřízených procesech požadavky od jednotlivých klientů (celkově 5 procesů). Všimněte si úseků programu, které popisují vytváření procesů, zpracování výsledků při ukončení procesu (odstranění procesu typu "zombie") a zrušení všech procesů při ukončení aplikace. Jestliže je přijat signál a vracíme se z jádra operačního systému (kde byl proces zablokován funkcí *recv*()*), musíme takovou situaci ošetřit.



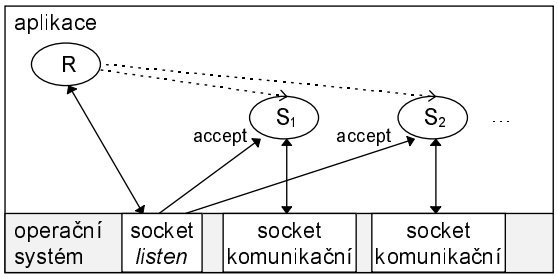
Obr. 3.4: Sdílení socketu pro předpřipravené UDP procesy

```
#define CLIENTS 4                                          // počet předpřipravených klientů
int client=0;                                              // aktuální počet běžících klientů
int clients[CLIENTS];                                      // pole pid klientů
//------------------------------------ obsluha signálů
void reaper()                                              // sběrač koncových stavů klientů
{
   union wait status;
   int pid, i;
   if ((pid=wait3(&status, 0, (struct rusage *)0))>0)      // (ne)vyzvedneme stav klienta
      for (i=0; i<CLIENTS; i++)                            // zrušíme klienta z pole pid klientů
         if (pid==clients[i]) { clients[i]=0; client--; break; }
   signal(SIGCHLD, reaper);                                // znovu zaregistrujeme signál
}
void ender()                                               // ukončovač aplikace (včetně podprocesů)
{
   int i;
   for (i=0; i<CLIENTS; i++)                               // všem zbývajícím klientům zašleme SIGTERM
      if (clients[i]) kill(clients[i], SIGTERM);
}
void stopper()                                             // ukončení aplikace signálem, zavoláme standarní řetězec ukončení
{ exit(0); }
//------------------------------------ hlavní program
main(int argc, char *argv)
{
   int i, pid, clnt_id;
   ....
   socket(...);                                            // stejné jako v předchozích příkladech
   bind(...);
   ....
//------------------------------------ start podprocesů
   for (clnt_id=0; clnt_id<CLIENTS; clnt_id++) {           // nastarujeme klienty
      if (pid=fork())                                      // start
      { client++; clients[clnt_id]=pid; }                  // v serveru si zapamatujeme pid klienta
      else break;                                          // v klientu neděláme nic
   }
   if (pid) {                                              // pro server zaregistrujeme signály
      if (atexit(ender))                                   // ukončovací funkce (zruší klienty) (případně onexit())
         { ender(); perror("Chyba v registraci obsluhy ukonceni"); exit(10); }
      if ((signal(SIGCHLD, reaper)==SIG_ERR) ||            // signál o ukončení činnosti klienta
```

```
        (signal(SIGTERM, stopper)==SIG_ERR))          // signálem "kill" ukončíme klienty
      { perror("Chyba v registraci obsluhy signalu"); exit(11); }
  }
//————————————————————————————— zpracování požadavků
  while (1) {
    rem_addr_length=sizeof(rem_addr);                          // připravíme příjem
    while ((len=recvfrom(sockfd, buf, sizeof(buf)-1, 0,
          (struct sockaddr *)&rem_addr, &rem_addr_length))<0) {
      if (errno==EINTR) continue;      // jedná se o předčasné ukončení (zpracovával se signál)
        perror("Chyba pri prijmu"); exit(3);
    }
    if (sendto(sockfd, buf, len, 0, (struct sockaddr *)&rem_addr, sizeof(rem_addr))<0)
      { perror("Chyba ve vysilani"); exit(4); }
  }
}
```

Paralelní TCP server s předpřipravenými procesy

Pokud použijeme místo protokolu UDP protokol TCP, kostra programu se příliš nezmění. Při inicializaci kromě inicializace socketu (*socket(), bind()*) připravíme frontu požadavků voláním funkce *listen()*. V hlavní smyčce podprocesu při zpracování požadavku počkáme na spojení (*accept()*) a potom přijmeme paket a odešleme odpověď stejně jako u jednoduchého serveru TCP. Pokud i hlavní proces bude obsluhovat klienty, musíme doplnit detekci návratových kódů u funkcí *accept()* a *recv*()* o příjem signálu jako u předchozího příkladu. Sdílení socketu mezi rodičovským procesem serveru R a synovskými procesy S je znázorněno na obrázku 3.5.



Obr. 3.5: Sdílení socketu pro předpřipravené TCP procesy

Paralelní TCP server se společným zpracováním

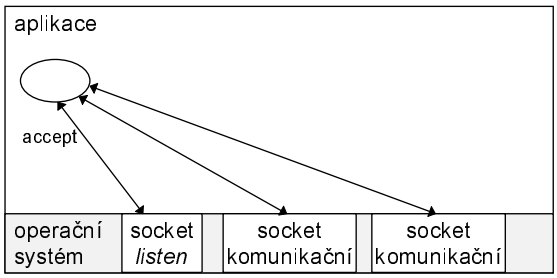
Použití serverů s předpřipravenými procesy omezuje počet současně prováděných obsluh na počet předem připravených procesů. Pokud dopředu nevíme, kolik klientů bude na server přistupovat, může se stát, že server poddimenzujeme (klienti pak budou muset čekat na iterativní zpracování). Naopak zbytečně mnoho procesů přináší zbytečnou režii. Řešením je obsluha podle okamžitých potřeb. Existují dvě možná řešení, zde si uvedeme server se společnou obsluhou požadavků; dynamickému vytváření procesů se budeme věnovat dále (str.33):

První uváděné řešení je vhodné v případě, kde je nutná rychlá reakce serveru a samotné nároky na zpracování požadavků nejsou vysoké. Paralelní zpracování podporuje funkce *select()*, která testuje tři množiny manipulátorů a jež může být navíc ukončena po uplynutí

definovaného času. Funkce zablokuje proces, dokud nebudou přijata data na socketu uvedeném v množině čtení (*readfs*), neskončí vysílání na socketu v množině zápisů (*writefds*), nedojde k výjimce (chybě) na socketu v množině výjimek (*exeptfds*) nebo nevyprší časový limit (timeout). S množinami manipulují makra *FD_*()*. Funkce *select()* sleduje sockety do zadaného maximálního čísla (*maxfd-1*).

```
int select(int maxfd, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
        struct timeval *timeout);
FD_ZERO(fd_set *set);                                    // vynuluje množinu
FD_SET(int fd, fd_set *set);                             // přidá socket do množiny
FD_CLR(int fd, fd_set *set);                             // vyjme socket z množiny
FD_ISSET(int fd, fd_set *set);                           // testuje, zde je socket v množině
```

Činnost serveru je následující: Server otevře socket *sockfd*, na kterém přijímá nové požadavky na spojení ("listen" socket). Zařadí ho do množiny pro čtení a počká, dokud nepřijde požadavek na vytvoření spojení. Po vzbuzení server otestuje, zda se jedná o požadavek na vytvoření spojení; nové spojení přijme funkcí *accept()* a nový socket zařadí do množiny pro čtení a zaregistruje v poli socketů klientů *c_sockfd*. Je-li pole klientů zaplněné, vyjme socket "listen" z množiny pro čtení (další spojení již není možné otevřít a obsloužit, dokud nebude některé z dříve otevřených ukončeno). Nejedná-li se o nový požadavek, server prochází již otevřená spojení a na spojení, které přijalo data, odpovídá klientovi. Pokud byl přijat blok s nulovou délkou (ukončující komunikaci - jako důsledek volání *close()* v kódu klienta), pak spojení uzavírá. Při uzavření spojení je socket vyjmut z pole socketů klientů a z množiny pro čtení. Případně vyjmutý "listen" socket je vrácen zpět do množiny pro čtení. Způsob sdílení socketu je znázorněn na obrázku 3.6.



Obr. 3.6: Sdílení socketů při společné paralelní obsluze TCP spojení

```
#define CLIENTS 4
int main(int argc, char *argv[])
{
   int sockfd, c_sockfd[CLIENTS];                // listen-socket a pole socketů pro klienty
   int c_cnt;                                    // počet připojených klientů
   fd_set fset, t_fset;                          // originál a kopie množiny
   ...
   socket();                                     // stejné jako v předchozích příkladech
   bind(...);
   listen(...);
   FD_ZERO(&fset);                               // vynulujeme množinu socketů
   FD_SET(sockfd, &fset);                        // přidáme listen-socket
```

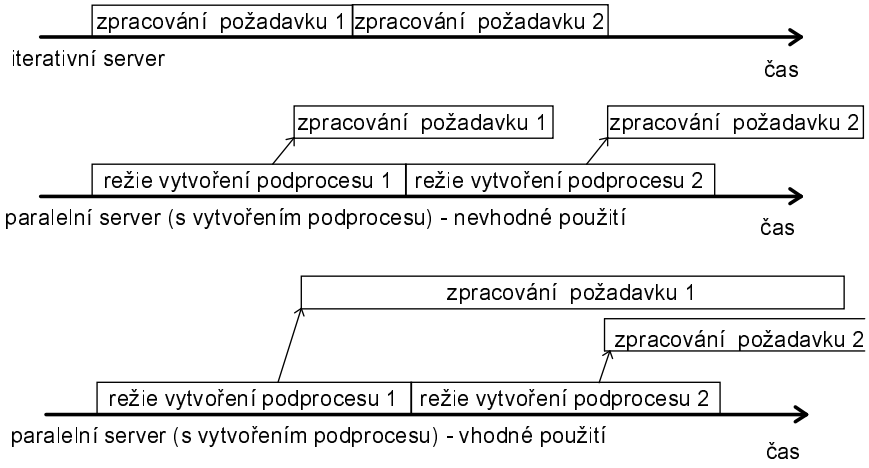
```
    c_cnt=0;                                                    // počet klientů
    while (1) {
       memcpy(&t_fset, &fset, sizeof(fset));                    // uděláme si kopii množiny
       if (select(getdtablesize(), &t_fset, (fd_set *)0, (fd_set *)0, (struct timeval *)0)<0)
          { perror("Chyba v select"); exit(1); }                // čekáme, až se něco stane
       if (FD_ISSET(sockfd, &t_fset)) {                         // testujeme, zda náhodou nešlo o listen-socket
          rem_addr_length=sizeof(rem_addr);                     // připravíme se na žádost o spojení
          if ((c_sockfd[c_cnt]=accept(sockfd, (struct sockaddr *)&rem_addr, &rem_addr_length))<0)
             { perror("Nelze accept"); close(sockfd); exit(1); }// přijmeme spojení
          FD_SET(c_sockfd[c_cnt], &fset);                       // přidáme do množiny
          if (++c_cnt==CLIENTS) FD_CLR(sockfd, &fset);          // pokud již nemáme místo,
       }                                                        // vyjmeme listen-socket z množiny
       for (i=0; i<c_cnt; i++)          // procházíme zbylé sockety a hledáme, zda nepřijaly data
          if (FD_ISSET(c_sockfd[i], &t_fset)) {                                    // přijaly ?
             if ((len=recv(c_sockfd[i], buf, BUFFSIZE, 0))<=0) {                   // převezmeme data
                if (len) perror("Chyba pri cteni");             // len=0 je uzavření spojení
                else {
                   close(c_sockfd[i]);                                             // uzavřeme socket
                   FD_CLR(c_sockfd[i], &fset);                  // vyjmeme z množiny a ze seznamu
                   memcpy(c_sockfd+i, c_sockfd+i+1, sizeof(int)*(CLIENTS-i-1));
                                                                // a pokud už máme volnou pozici,
                   if (c_cnt--==CLIENTS) FD_SET(sockfd, &fset);
                   break;                                       // vrátíme listen-socket a pokračujeme
                }
             }
             else {
                if (send(c_sockfd[i], buf, len, 0)<0)                              // vrátíme odpověď
                { perror("Chyba pri zapisu"); break; }
             }
          }
    }
}
```

Paralelní TCP server s dynamickým vytvářením procesů

Pokud nelze řešit obsluhu více klientů jediným procesem, protože je příliš komplikovaná, můžeme spustit při převzetí požadavku na spojení synovský proces, který se o obsluhu postará. Při otevírání spojení však musíme počítat s dodatečnou režií, potřebnou ke spuštění synovského procesu nebo dokonce samostatného vnějšího programu. Efektivita takového řešení může být často horší, než kdybychom použili čistě iterativní server (obrázek 3.7).

Následující příklad využívá pro obsluhu synovský proces, využití vnějšího programu si uvedeme v dalším příkladě. Všimněte si, že v rodičovském procesu zavíráme komunikační socket. Synovský proces naproti tomu nepotřebuje socket pro příjem spojení. Způsob sdílení socketu je znázorněn na obrázku 3.8.

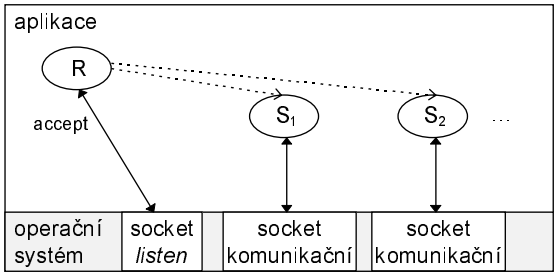
```
void reaper()
{
   union wait status;
   while (wait3(&status, 0, (struct rusage *)0)>0);         // (ne)vyzvednutí stavu podprocesu
```



Obr. 3.7: Porovnání vhodně a nevhodně voleného paralelního zpracování

```
   signal(SIGCHLD, reaper);                                    // opětovná registrace signálu
}
int main(int argc, char *argv[])
{
   int sockfd, c_sockfd;
   ...
   socket();                                                   // stejné jako v předchozích příkladech
   bind(...);
   listen(...);

   if (signal(SIGCHLD, reaper)==SIG_ERR)                       // registrace ukončovací procedury
      { perror("Chyba v registraci obsluhy signalu"); close(sockfd); exit(11); }
   while (1) {
      rem_addr_length=sizeof(rem_addr);                        // připravíme se na příjem požadavku
                                                               // převezmeme spojení
      if ((c_sockfd=accept(sockfd, (struct sockaddr *)&rem_addr, &rem_addr_length))<0) {
         if (errno==EINTR) continue;                           // přerušení při příjmu signálu
            perror("Nelze accept"); close(sockfd); exit(1);
      }
      switch (fork()) {                                        // a starujeme podproces
         case 0:                                               // jedná se o podproces
            close(sockfd);                                     // nepotřebujeme listen-socket
            if ((len=recv(c_sockfd, buf, BUFFSIZE, 0))<0)      // přijmeme zprávu
               perror("Chyba pri cteni");
            else
               while (len) {                                   // opakujeme, dokud jsou data
                  if (send(c_sockfd, buf, mlen, 0)<0)          // zašleme zprávu
                     { perror("Chyba pri zapisu"); break; }
                  if ((len=recv(c_sockfd, buf, BUFFSIZE, 0))<0)  // přijmeme další
                     { perror("Chyba pri cteni"); break; }
               }
            close(c_sockfd);                                   // konec komunikace
            exit(0);                                           // konec podprocesu
```

```
            case -1:
               perror("Chyba pri fork"); exit(20);
            default:                                                    // hlavní proces
               close(c_sockfd);                          // nepotřebujeme komunikační socket na klienta
         }
      }
   }
```



Obr. 3.8: Sdílení socketu pro spouštěné podprocesy TCP spojení

Paralelní TCP server využívající vnějšího procesu

Příklad z předchozí kapitoly si rozšíříme o spuštění vnějšího procesu. Takový postup dovoluje vytvářet univerzální *super–servery*, které na základě požadavku na spojení spouští příslušný výkonný server jako vnější proces. Socket obvykle předáváme jako standardní vstup a výstup. Popsaným způsobem pracuje i standardní internet–démon UNIXu *inetd*. Tento démon podle konfigurace /etc/inetd.conf poslouchá na zadaných portech, a pokud se objeví požadavek na spojení, spustí příslušný výkonný komunikační server. Standardní ARPA servery (in.telnetd, in.ftpd ...) i BSD servery (in.rexecd, in.rlogind, in.rshd ...) jsou také spouštěny tímto způsobem. Internet–démon dokáže registrovat a spouštět i RPC služby (rpc.rusersd, rpc.rwalld ...).

Proti předchozímu příkladu se pouze mění způsob spouštění podprocesu.

```
#define CHILD "tcps-child"                                      // jméno startovaného programu
#define DATAFD 5                                    // manipulátor, přes který se bude komunikovat
      ...                                                       // stejné jako v předchozím příkladu
      switch (fork()) {                                                  // a startujeme podproces
         case 0:                                                          // jedná se o podproces
            close(sockfd);                                      // nepotřebujeme listen-socket
            if (c_sockfd!=DATAFD) {                  // zduplikujeme do požadovaného manipulátoru
               if (dup2(c_sockfd, DATAFD)!=DATAFD)
               { perror("Nelze dup2"); exit(1); }
               close(c_sockfd);                                   // uzavřeme nepotřebný originál
            }
            execl(CHILD, NULL)              // nastartujeme cizí proces (nahradí stávající podproces)
            perror("Nelze exec"); exit(21);                                    // nepodařilo se
         case -1:
            perror("Chyba pri fork"); exit(20);
         default:                                                                // hlavní proces
            close(c_sockfd);                          // nepotřebujeme komunikační socket na klienta
      }
```

Další částí programu je samozřejmě i samotný výkonný proces. Na rozdíl od řešení, které používá *inetd*, není zde využitý pro předávání socketu standardní vstup a výstup (používáme ho pro výpisy na terminál), ale předdefinovaný socket (DATAFD).

```
#define BUFFSIZE 1000
#define DATAFD 5                                    // manipulátor, přes který se bude komunikovat
main()
{
char buf[BUFFSIZE];
struct sockaddr_in rem_addr;
int rem_addr_length;
int mlen;
struct hostent *ph;
rem_addr_length=sizeof(rem_addr);                   // připravíme pro zjištění jména klienta
if (getsockname(DATAFD, (struct sockaddr *)&rem_addr, &rem_addr_length))      // zjistíme adresu
   { perror("Chyba pri zjisteni jmena socketu"); exit(1); }
if (ph=gethostbyaddr((char *)&rem_addr.sin_addr, sizeof(rem_addr.sin_addr), AF_INET))
   printf("Prijato od %s\n", ph->h_name);                  // přes jmennou službu vypíšeme
else
   { herror("Chyba pri zjisteni jmena pocitace"); exit(2); }
if ((len=recv(DATAFD, buf, BUFFSIZE, 0))<0)
   perror("Chyba pri cteni");
else
   while (len) {                                    // opakujeme, dokud jsou data
   if (send(DATAFD, buf, len, 0)<0)                 // zašleme zprávu
   { perror("Chyba pri zapisu"); exit(1); }
   if ((len=recv(DATAFD, buf, BUFFSIZE, 0))<0)      // přijmeme další
   { perror("Chyba pri cteni"); exit(1); }
}
close(DATAFD);                                      // uzavřeme manipulátor a ukončíme program
}
```

3.2.4 Vlastnosti socketů

Předchozí kapitoly obsahovaly základní popis socketového rozhraní, se kterým lze vystačit ve většině programů. V této kapitole se podíváme na některé další vlastnosti, které lze ovlivnit funkcemi *fcntl()*, *ioctl()*, *setsockopt()* a *getsockopt()*. Je ale nutné upozornit, že zde uváděné funkce upravují vlastnosti socketů, které jsou implementačně závislé. Některá nastavení navíc vyžadují spuštění procesu v privilegovaném režimu. Podrobnější informace lze získat z manuálových stránek a z popisu implementace jádra operačního systému.

V úvodu byla zmíněna příbuznost socketů a manipulátorů souborů v rámci operačního systému. Pro ovlivnění některých vlastností socketu (obecné vlastnosti manipulátorů) lze použít funkci *fcntl()* (prototyp funkce je v hlavičkovém souboru <*fcntl.h*>):

```
int fcntl(int fd, int cmd, long arg);
```

K užitečným operacím se socketem (určují je konstanty F_* jako hodnoty argumentu *cmd*) patří zjištění a nastavení parametrů (operace F_GETFL a F_SETFL). Zajímavým příkladem je

parametr O_NONBLOCK, který dovoluje, aby funkce, které by jinak blokovaly výpočet procesu (např. *read()*), vrátily řízení okamžitě nebo po uplynutí určeného časového limitu (tedy např. bez načtení dat), ale s indikací chyby EWOULDBLOCK. To nám dovoluje obejít spouštění paralelně běžících procesů/vláken v případě, kdy chceme kombinovat čekání na výsledek funkce s další aktivitou. Je však nutné si uvědomit, že cenou za jednoduše zapsaný program může být snížení efektivity celého systému (např. při náhradě funkce *select()* cyklem).

Další systémově závislé parametry ovladačů zařízení svázaných se socketem nastavuje funkce *ioctl()* (prototyp funkce je v hlavičkovém souboru <*sys/ioctl.h*>). Funkcí lze například nastavit nebo zjistit směrovací tabulku, tabulku ARP, vlastnosti síťového rozhraní na síťové i linkové vrstvě, apod..

```
int ioctl(int fd, int request, ...);
```

Jako poslední uvedeme funkce *setsockopt()* a *getsockopt()* nastavující a měnící vlastnosti spojené výhradně se sockety.

```
int getsockopt(int sockfd, int level, int optname, void *optval, int *optlen);
int setsockopt(int sockfd, int level, int optname, const void *optval, int optlen);
```

Argument *optname* určuje zjišťovaný/nastavovaný parametr, argument *level* říká, ke které protokolové vrstvě (IP - SOL_IP, TCP - SOL_TCP, socketové rozhraní - SOL_SOCKET) se parametr vztahuje. Další argumenty (*optval* a *optlen*) slouží k předání hodnoty. Seznam nastavitelných parametrů najdeme v hlavičkovém souboru <*sys/socket.h*>. Zde si uvedeme několik zajímavých parametrů (jména parametrů jsou ve formátu vrstva/parametr):

- SOL_SOCKET / SO_REUSEADDR umožní využít již používaného připojení při volání funkce *bind()* (např. sdílet port u TCP nebo UDP protokolu),
- SOL_SOCKET / SO_OOBINLINE umožní přijímat urgentní data mimo hlavní tok dat (příznak MSG_OOB ve funkcích *recv(), send()*),
- SOL_SOCKET / SO_RCVBUF, SO_SNDBUF změní velikost vyrovnávací paměti socketu,
- SOL_SOCKET / SO_RCVTIMEO, SO_SNDTIMEO nastavuje časový limit pro návrat s chybou EWOULDBLOCK,
- SOL_IP (IPPROTO_IP) / IP_TOS, IP_TTL a IP_OPTIONS upraví příslušné položky IP hlavičky (str. 10).

Následující příklad uvádí přehled využití funkcí zmíněných v této kapitole. Příklad je fragmentem zdrojového tvaru programu *ping*, který je, stejně jako jiné programy, volně dostupný ve zdrojovém tvaru na Internetu.

```
#define IPOPT_NROUTES 9                    // počet záznamů v IP_OPTIONS-RR
u_char packet[MAXPACKET], *cp, *packet;
struct sockaddr_in to, from;
struct protoent *proto;
struct icmphdr *icp;                                              // ICMP
struct iphdr *ip;                                                 // IP
char rspace[3 + 4 * IPOPT_NROUTES + 1];   // vytvoříme místo pro záznam cesty
```

```
   int hold, i, j, cc, old_rrlen;
   ....
//———————————— nastavení SOCK_RAW socketu pro ICMP protokol
   if (!(proto = getprotobyname("icmp")))                    // vyhledání protokolu ICMP
   { fprintf(stderr, "ping: Neznam protokol ICMP.\n"); exit(2); }
   if ((s = socket(AF_INET, SOCK_RAW, proto->p_proto)) < 0) {
      if (errno==EPERM) fprintf(stderr, "ping: program vyzaduje privilegovany rezim\n");
      else perror("ping: nepodaril se socket()");
      exit(2);
   }
   ...
//———————————— nastavení parametrů socketu
   hold = 1;                                                 // nastavíme si broadcast
   setsockopt(s, SOL_SOCKET, SO_BROADCAST, (char *)&hold, sizeof(hold));
   hold = 64 * 1024;                        // odpovědí na "broadcast ping" může být více paketů
                                                             // - zvětšíme si vyrovnávací paměť
   (void)setsockopt(s, SOL_SOCKET, SO_RCVBUF, (char *)&hold, sizeof(hold));
   if (?) {                                  // zaznamenáme cestu pomocí record-route v IP-options
      memset(rspace, 0, sizeof(rspace));
      rspace[IPOPT_OPTVAL] = IPOPT_RR;
      rspace[IPOPT_OLEN] = sizeof(rspace)-1;
      rspace[IPOPT_OFFSET] = IPOPT_MINOFF;
      if (setsockopt(s, IPPROTO_IP, IP_OPTIONS, rspace, sizeof(rspace)) < 0)
      { perror("ping: record route"); exit(2); }
   }
   ....
//———————————— sestavení ICMP
   icp = (struct icmphdr *)packet;
   icp->icmp_type = ICMP_ECHO;
   icp->icmp_code = 0;
   icp->icmp_cksum = 0;
   icp->icmp_seq = ...;
   icp->icmp_id = ident;                                     // unikátní identifikace
   ....
//———————————— odeslání dotazu
   i = sendto(s, (char *)packet, sizeof(packet), 0, &to, sizeof(struct sockaddr));
   ....
//———————————— příjem odpovědi
   fromlen = sizeof(from);
   packlen = sizeof(packet);
   if ((cc = recvfrom(s, (char *)packet, packlen, 0,
         (struct sockaddr *)&from, &fromlen)) < 0) {
      if (errno == EINTR) continue;
      perror("ping: recvfrom"); exit(2);
   }
   ....
//———————————— zpracování IP a IP_OPTIONS
   ip = (struct iphdr *)packet;
   hlen = ip->ip_hl << 2;
   cp = (u_char *)buf + sizeof(struct iphdr);                // zobrazíme IP_OPTIONS
   for (; hlen > (int)sizeof(struct iphdr); --hlen, ++cp)
      switch (*cp) {
```

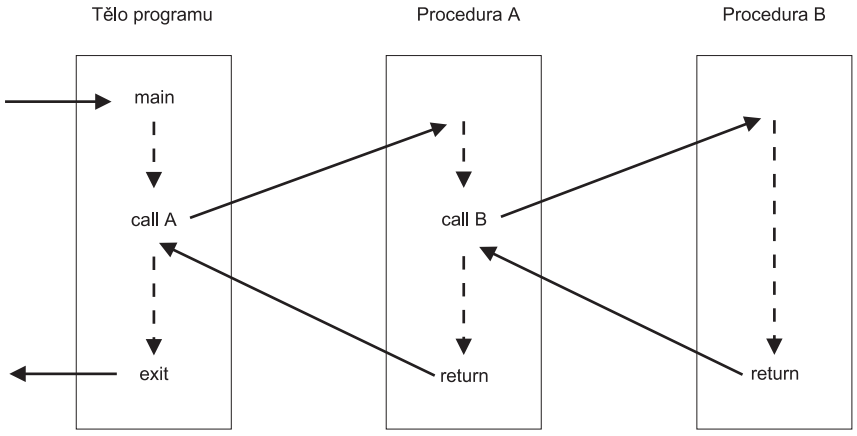
```
        ....
        case IPOPT_RR:
           j = ***cp;                                          // délka
           i = ***cp;                                          // adresa
           hlen -= 2;
           if (i > j) i = j;
           i -= IPOPT_MINOFF;
           if (i <= 0) continue;
           if (i == old_rrlen && cp == (u_char *)buf + sizeof(struct iphdr) + 2
                 && !memcmp((char *)cp, old_rr, i) && !(options & F_FLOOD)) {
              (void)printf("\t(same route)");
              i = ((i + 3) / 4) * 4;
              hlen -= i;
              cp += i;
              break;
           }
           old_rrlen = i;
           memcpy(old_rr, cp, i);
           (void)printf("\nRR: ");
           for (;;) {
              l = ***cp;
              l = (l<<8) + ***cp;
              l = (l<<8) + ***cp;
              l = (l<<8) + ***cp;
              if (l == 0) printf("\t0.0.0.0");
              else printf("\t%s", pr_addr(ntohl(l)));
              hlen -= 4;
              i -= 4;
              if (i <= 0) break;
              putchar('\n');
           }
           break;
        ...
        default:
           (void)printf("\nunknown option %x", *cp);
           break;
        }
//————————————— zpracování ICMP
     cc -= hlen;
     icp = (struct icmphdr *)(buf + hlen);
     if (icp->icmp_type == ICMP_ECHOREPLY) {
        if (icp->icmp_id != ident) return;
        ....
     }
```

Za poznámku v textu příkladu stojí zpracování pole *Options* IP paketu vracejícího odpověď, kde v tomto případě najdeme záznam cesty paketu sítí.

4. RPC systémy

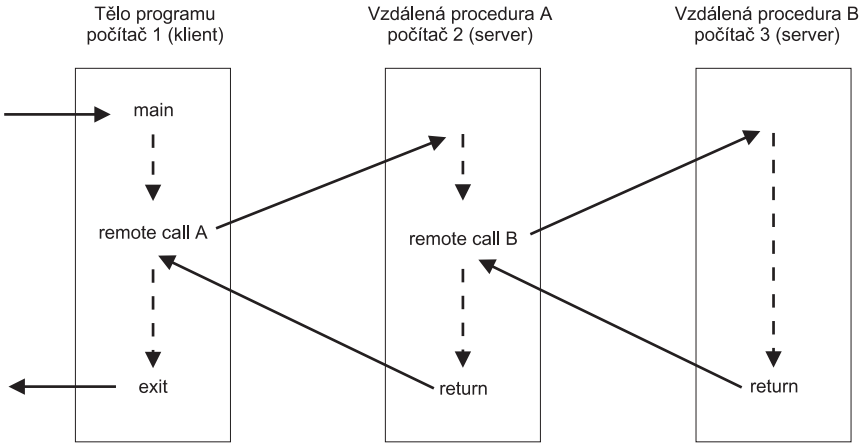
Programování komunikace mezi komponentami aplikace rozložené na více počítačů propojených vhodným komunikačním systémem s běžnými programovacími prostředky (jazyk C a příslušné komunikační a systémové knihovny) vyžaduje značné znalosti a dovednosti. Větší rozšíření technologie *klient-server* si vynutilo vytvořit prostředky, dovolující složitost programování snížit a spolupráci komponent standardizovat (a tak možná i omezit výběr možných řešení). Současné technologie se opírají o model *volání vzdálených procedur* RPC (*Remote Procedure Call*), poprvé popsaný v [10]. Mechanismus má základ v klasických programovacích technikách. Jedná se o jazykovou obdobu lokálního volání procedury nebo funkce. Je reprezentací modelu klient-server, kde je předání požadavku *request* a příslušné odpovědi *response* "zapouzdřeno" do volání procedury s parametry.

Porovnejme si volání funkcí v konvenčním programu s technologií RPC. U konvenčního (lokálního) modelu volání jsou z těla programu volány procedury a funkce, které přebírají vhodně uložené parametry a po provedení požadované akce a uložení výsledků vrací řízení do místa volání. Příklad zřetězeného lokálního volání uvádí obr. 4.1.



Obr. 4.1: Lokální volání procedur/funkcí

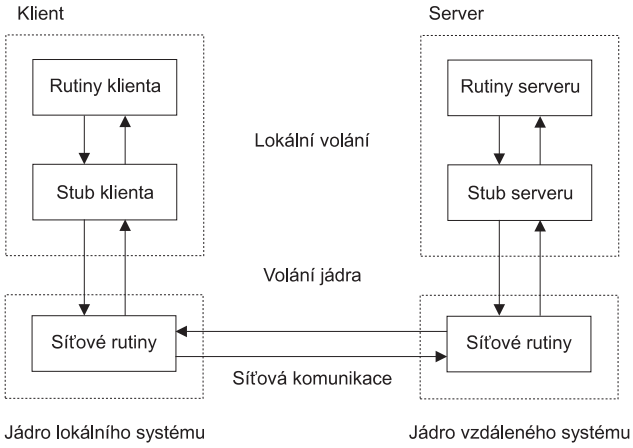
Umístíme-li volané procedury/funkce na jiný počítač a jejich volání zprostředkujeme sítí, nazýváme tyto procedury *vzdálené*. Podobně jako u lokálního volání lze volání vzdálených procedur řetězit (obr. 4.2).



Obr. 4.2: Vzdálené volání procedur/funkcí

Je pochopitelné, že implementace volání vzdálené procedury je odlišná od volání lokálního. Protože volání zprostředkuje síť, musíme zajistit převod volání (tj. předání identifikace procedury a jejích parametrů) na zprávy transportního protokolu (např. TCP, nebo UDP). O převod se stará programová *spojka* (*stub*). Uložení identifikačních údajů (identifikace procedury/funkce, autorizace) a parametrů volání do zprávy *request* se nazývá *marshalling* a realizuje ho *spojka klienta*. Zpětný převod přenesené zprávy *request* na identifikační údaje a parametry volání se nazývá *unmarshalling* a realizuje ho *spojka serveru*. Spojka klienta lokálně "nahrazuje" proceduru, takže v programu může být zápis volání vzdálené procedury velmi podobný zápisu volání lokální procedury. Spojka bývá generována automaticky z popisu rozhraní mezi klientem a serverem.

Vlastní volání vzdálené procedury vypadá následovně: Volání vzdálené procedury ve zdrojovém kódu klienta je přeloženo na lokální volání spojky klienta. Spojka vytvoří zprávu/zprávy daného transportního protokolu a odešle ji serveru. Server zprávu (případně zprávy) přijme a jeho spojka převede přijatou zprávu na formu potřebnou pro volání procedury, která je implementována v kódu serveru. Po ukončení výpočtu vrátí procedura výsledky spojce, která je převede na zprávu/zprávy transportního protokolu a odešle klientovi. Spojka klienta odpověď dekóduje a vrátí výsledek klientovi do místa volání (obr. 4.3).



Obr. 4.3: Volání vzdálené procedury

U komunikace RPC, jak jsme si ji právě popsali, čeká klient na výsledek výpočtu serveru. Takový režim označujeme jako *synchronní*, a odlišujeme ho od *asynchronního* režimu, u kterého povolujeme pokračování ve výpočtu.

4.1 Transparentnost RPC a její zajištění

Při navrhování RPC modelu se obvykle snažíme, aby se volání vzdálené procedury z pohledu vývojáře aplikace podobalo volání procedury lokální. Vzdálené volání však má některé principiální odlišnosti od volání lokálního. Ty se týkají následujících oblastí:

- předávání parametrů,
- sémantiky volání vzdálené procedury,
- reprezentace dat,
- transportního protokolu,
- vazby (binding),
- zpracování výjimek,

- výkonu,

- bezpečnosti.

V následujících odstavcích se o jednotlivých odlišnostech zmíníme podrobněji.

Předávání parametrů

Při předávání parametrů hodnotou nevznikají žádné potíže, spojka klienta pouze vloží hodnotu do zprávy transportního protokolu. Pokud však chceme parametry předat odkazem, musíme si uvědomit, že klient a server mají jiný adresní prostor. Přesto však implementace RPC dovolují i tento způsob předávání parametrů. Obvykle je předávání parametrů odkazem převedeno na předávání parametrů hodnotou.

Některé implementace RPC mají prostředky omezující množství přenášených dat (například pole s přizpůsobitelnou délkou v DCE - str.65).

Sémantika volání

Používáme-li lokální volání procedury, běží hlavní program i volaná procedura na jednom počítači. Předávání parametrů a výsledků probíhá uvnitř systému a zodpovídá za něj systém. Pokud dojde k chybě, měla by být tato chyba signalizována systémem, aby na ní aplikace mohla reagovat. Při použití RPC je aplikace rozdělena na dvě (nebo více) částí. Ztráta požadavku na vykonání vzdálené procedury (včetně parametrů), nebo ztráta odpovědi s výsledky je oproti lokálnímu volání procedury pravděpodobnější a není nijak signalizována.

Nedostaneme-li od vzdálené procedury výsledek, nevíme, zda se ztratil požadavek a procedura tedy neproběhla, zda havaroval server během výpočtu a tato skutečnost nebyla ohlášena výjimkou, nebo zda procedura proběhla a ztratil se výsledek. Potřebujeme-li pokračovat ve výpočtu (který na výsledek nebo výjimku čeká), můžeme omezit čekání *časovým limitem* (*timeout*).

Reakcí na vypršení časového omezení může být pokračování ve výpočtu (jako kdyby výpočet korektně skončil) nebo můžeme volání vzdálené procedury opakovat. Může ale nastat situace, že se pro zpoždění (při komunikaci nebo při výpočtu) nebo pro ztrátu odpovědi spustí již provedená vzdálená procedura opakovaně. Je tedy zřejmé, že je vhodné takové chování uvažovat a rozlišovat mezi sémantikou vzdáleného volání a volání lokálního.

Obvykle definujeme tři základní sémantiky synchronního volání RPC, jejichž výhodnost a použitelnost závisí často i na konkrétní aplikační oblasti:

Právě jednou (*exactly-once)*) - vzdálená procedura se spustila jednou, a ne vícekrát ani méněkrát. Tohoto ideálního stavu nelze v praxi téměř dosáhnout pro možnou ztrátu stavové informaci při havárii serveru. Praktické implementace RPC se této sémantice snaží přiblížit opakováním volání vzdálené procedury. Násobnému spuštění vzdálené procedury se bráníme číslováním požadavků a odpovědí, server si musí udržovat nejen příslušné čítače, ale ukládat i odeslané odpovědi.

Nejvýše jednou (*at-most-once*) - zaručuje, že procedura se buď nespustí, nebo se spustí jen jednou. Pokud obdržíme výsledek, víme, že procedura se spustila právě jednou. Nedostaneme-li výsledek do časového limitu, je jisté, že procedura se buď nespustila, nebo se spustila jednou. Této sémantiky dosáhneme v případě, že požadavek neopakujeme.

Alespoň jednou (*at-least-once*) procedura se spustila jednou, nebo vícekrát. Tento typ sémantiky dostáváme (automatickým) opakováním volání při vypršení časového limitu. Sémantika vyhovuje idempotentním procedurám. Tyto procedury mají tu vlastnost, že je lze

provést několikrát po sobě (patří sem třeba zápis dat od dané pozice, čtení teploty, času). Nevhodná je tato sémantika pro procedury, které nejsou idempotentní. Mezi ně patří např. odečtení částky z bankovního konta nebo zápis na konec souboru.

Sémantiky, které jsme si v našem výčtu uvedli, nepokrývají celé spektrum možností systémů RPC. V praxi se setkáme ještě se sémantikami *broadcast* nebo *multicast* (spuštění funkce na více serverech souběžně jediným voláním) a *anycast* (spuštění funkce na některém z více serverů). Některé systémy dovolují klientovi pokračovat ve výpočtu bezprostředně po odeslání požadavku, mluvíme pak o *asynchronním* a *one-way* volání.

Reprezentace dat

Při volání procedur je většinou potřebné předat vstupní data a získat výsledek. V případě volání vzdálené procedury se objevuje otázka reprezentace dat. Pokud se jedná o počítače stejného typu se stejným operačním systémem, nevzniknou větší potíže. Horší situace nastane v případě komunikace mezi různými typy počítačů a různými operačními systémy. Každá kombinace počítač-operační systém může totiž použít (a většinou používá) jiný typ reprezentace dat. Rozdíly mohou být v potřebné délce (typ integer může mít délku 16, 32, nebo 64 bitů), v kódování informace (znakové kódy ASCII, EBCDIC, čísla v plovoucí řádové čárce) nebo v jejich uložení v paměti (zobrazení little-endian, big-endian).

K převodu mezi reprezentacemi dat používáme množiny konverzních funkcí. Množiny proto, že musíme mít konverzní funkci pro každý zobrazitelný typ v dané reprezentaci. Vyjdeme-li z předpokladu, že existuje N reprezentací, musíme vytvořit $N * (N - 1)$ množin konverzních funkcí. Přidáme-li novou reprezentaci, bude nutné přidat N nových množin konverzních funkcí.

Pokud chceme snížit počet množin konverzních funkcí, můžeme vytvořit speciální reprezentaci pro přenos dat. Potom bude potřeba vytvořit pouze $2 * N$ množin konverzních funkcí a při přidání nového systému stačí přidat pouze dvě množiny, jednu pro překlad do standardní reprezentace a druhou pro překlad zpětný. Nevýhodou tohoto řešení je překlad do speciální reprezentace i v případě, že spolu komunikují dva stejné systémy. Navíc se obvykle zvýší i počet převodů (do speciální reprezentace a ze speciální reprezentace). Na tomto principu vzniklo několik standardů.

Často používaným formátem je standard XDR (eXternal Data Representation) firmy Sun. U této reprezentace je každé položce dat vyhrazeno $4 * n$ oktetů, kde $n \geq 0$. Pokud je pro zobrazení hodnoty potřeba oktetů méně, doplní se pro přenos nevýznamnými znaky. Číselné položky jsou uloženy od nejvíce významného oktetu (MSB - *Most Significant Byte*) po nejméně významný oktet (LSB - *Least Significant Byte*). Tento způsob uložení je označován jako *big-endian*. Po síti se přenášejí pouze hodnoty položek bez informace o typu položky a mluvíme o *implicitním zobrazení*. Typ předávaných dat musí být znám oběma komunikujícím počítačům.

Při vlastní komunikaci mezi dvěma počítači jsou data programu běžícího na jednom počítači převedena do formátu XDR. Po přenosu přes síť jsou data převedena z formátu XDR do formátu používaného na druhém počítači. Ze způsobu přenosu dat vyplývá, že v případě komunikace dvou počítačů využívajících uložení big-endian je režie na převod dat menší než v případě počítačů s uložením little-endian.

Další řešení, které si uvedeme, je formát NDR (Network Data Representation). Stejně jako u formátu XDR nepřenášíme typ dat (používáme implicitní zobrazení). Oproti formátu XDR mohou mít stejná data ve formátu NDR různou reprezentaci (*multicanonical format*). Informace o použité reprezentaci je uložena v NDR hlavičce, což jsou první čtyři oktety zprávy.

Použití formátu NDR redukuje potřebu konverze na stranu příjemce. Program, který data posílá, pouze vloží do hlavičky informaci o formátu. V případě, že spolu komunikují počítače

se stejnou reprezentací dat, konverze se neprovádí.

Jak jsme uvedli, uvedené formáty používají *implicitní zobrazení*, po síti se přenáší pouze hodnoty proměnných, nikoliv údaje o jejich typu. Při *explicitním zobrazení* se naopak přenáší typ proměnné spolu s její hodnotou. Takové zobrazení používá například presentační protokol ASN.1 (*Abstract Syntax Notation One*) definovaný organizací ISO a používaný při správě počítačových sítí.

Transportní protokol

Řada implementací RPC je nezávislá na použitém transportním protokolu. V praxi naprostá většina implementací podporuje protokoly UDP a TCP. Někdy se liší možnosti RPC podle typu použitého transportního protokolu. Například sémantika volání *broadcast* v prostředí DCE (*doručení všem serverům*) je možná pouze při použití datagramové služby (UDP). Použijeme-li však datagramovou službu (UDP) pro běžnou komunikaci klient-server, může být proti virtuálnímu kanálu (TCP) sémantika volání vzdálené procedury odlišná. Nedostane-li klient výsledek do předem nastaveného časového limitu, zavolá vzdálenou proceduru znovu a ta se provede opakovaně (tedy budeme mít sémantiku *alespoň jednou*). Tato situace nastane, pokud se ztratí odpověď serveru, nebo pokud je doba zpracování procedury na serveru příliš dlouhá. Násobnému provedení vzdálené procedury se lze bránit vhodným nastavením časových limitů (delších než doba zpracování a komunikace), použitím spojované služby (TCP automaticky zopakuje ztracené zprávy), nebo použitím vhodného systému RPC (například *Transport-Independent RPC*).

Vazba

Pokud chce klient spustit vzdálenou proceduru, musí ji nejdříve nalézt. Vyhledávání má dvě fáze:

- nalezení hostitelského počítače s požadovaným serverem,
- nalezení správného procesu serveru na daném počítači.

Údaje dovolující vyhledání počítače se serverem může mít klient zadány napevno, nebo mohou být uloženy v nějaké adresační databázi.

Nalezení správného procesu bývá řešeno dynamicky. Na cílovém počítači je spuštěn server, který takové vyhledávání podporuje (např. *portmapper* v ONC RPC, nebo *rpc démon* v DCE). Tento server udržuje informace o všech serverech RPC spuštěných a korektně registrovaných na témže počítači. Informace, které tento server využívá, mohou být modifikovány aplikací, nebo je modifikuje administrátor. Přístupový port takového procesu je vyhrazený (*well-known*) port.

Zpracování výjimek

Při běhu aplikace může dojít ke vzniku řady výjimek (dělení nulou, neplatný odkaz). Pokud k chybě dojde v lokální proceduře, nastaví systém chybové proměnné, které aplikace může zpracovat a ukončí výpočet procedury. Dojde-li k podobné výjimce na vzdáleném systému, je nutné tyto informaci o chybě přenést. Některé implementace RPC dovolí ve spojce automaticky vytvořit spojení pro chybové proměnné.

Výkon

Je pochopitelné, že použití RPC přináší zvýšenou režii. Největším přínosem RPC není zvýšení rychlosti výpočtu (i když i to při vhodném využití může být významné), ale usnadnění programování distribuovaných aplikací. Použití distribuovaných aplikací se vyplatí v případě paralelně běžícího výpočtu s velkou výpočetní a menší komunikační náročností. Distribuované řešení je výhodné, pokud požadujeme větší spolehlivost systému, a jednotlivé servery se mohou zastupovat.

Bezpečnost

Jelikož při použití RPC dochází ke spouštění kódu na vzdáleném počítači, je třeba věnovat zvýšenou pozornost bezpečnosti. Na bezpečnost lze pohlížet z několika úhlů:

- právo spuštění serveru,
- utajení předávaných dat před třetí stranou,
- znemožnění změny dat při přenosu mezi klientem a serverem.

Ne všechna současná řešení RPC podporují všechny výše zmíněné způsoby zabezpečení. Budoucí systémy se již bez řešení bezpečnostních problémů neobejdou.

4.2 ONC RPC

V současnosti existuje celá řada implementací techniky RPC. V tomto textu se budeme zabývat dvěma z nich. První je ONC (*Open Network Computing*) RPC, která také bývá označována jako Sun RPC. Ta je momentálně zřejmě nejrozšířenější a můžeme se s ní setkat na různých systémech. Druhou implementací, kterou si popíšeme později je DCE RPC.

Zkratka ONC je označením pro skupinu produktů. Ta zahrnuje podporu vlastního ONC RPC, bezestavový síťový souborový server NFS (*Network File System*) využívající RPC komunikaci a adresářový systém NIS (*Network Information Service*), známý také jako *Yellow Pages*.

Systém ONC RPC byl vytvořen pro podporu klient-server aplikací zapsaných v jazyce C. Tvoří ho definice formátu XDR a knihovna základních konverzních funkcí, definice jazyka rozhraní RPCL (*RPC Language*) a překladač tohoto jazyka *rpcgen*. Překladač vytváří kostru aplikace klient server, generuje kód spojky klienta, kód spojky serveru a příslušný hlavičkový soubor.

4.2.1 External data representation

Důležitou součástí každého RPC systému pro heterogenní prostředí je definice formátu předávaných dat. ONC RPC využívá formátu označovaného jako XDR (*eXternal Data Representation*). Každé položce dat je vyhrazeno $4 * n$ oktetů, kde $n \geq 0$. Pokud je pro zobrazení položky potřeba jiný počet oktetů, doplní se pro přenos výplňovými znaky. Čísla jsou uložena jako *big-endian*. Po síti se přenášejí pouze hodnoty položek bez informace o typu (výjimkou je diskriminant u variantního záznamu), typ předávaných dat musí tedy být znám oběma komunikujícím aplikacím. Základní datové typy a konstruktory používané XDR jsou:

- integer (32 bitů) - celé číslo se znaménkem,
- unsigned integer (32 bitů) - celé číslo bez znaménka,
- enumeration - výčtový typ (32 bitů),
- boolean (32 bitů, ekvivalentní typu enum FALSE=0, TRUE=1),
- hyper integer (64 bitů) - celé číslo se znaménkem,
- hyper unsigned integer (64 bitů) - celé číslo bez znaménka,
- floating-point (32 bitů) - číslo s plovoucí desetinnou čárkou,
- double precision floating-point (64 bitů) - číslo s plovoucí desetinnou čárkou,
- quadruple precision floating-point (128 bitů) - číslo s plovoucí desetinnou čárkou,
- fixed length opaque data - posloupnost oktetů pevné délky (není na rozhraní interpretována/překládána),
- variable length opaque data - posloupnost oktetů proměnné délky (první 4 oktety udávají délku posloupnosti, následují vlastní data případně doplněná výplňovými oktety),
- string - řetězec znaků, má proměnnou délku a je ukončen prázdným znakem (NULL),
- fixed length array - pole pevné délky,
- variable length array - pole proměnné délky,
- structure - struktura/záznam s pevnou délkou,
- discriminated union - variantní záznam (struktura záznamu je určena diskriminantem, který je uložen v prvních čtyřech oktetech),
- void - prázdný typ (bez hodnoty).

Pro převod mezi reprezentací dat v jazyce C a reprezentací XDR jsou k dispozici konverzní funkce. Tyto funkce jsou dobře popsány v [11] a některé z nich si uvedeme v tomto textu. Podrobný popis reprezentace XDR včetně definičního jazyka, který je velmi podobný jazyku C (pokud jde o definice typů) lze nalézt v [12]. Příklady definic v jazyce XDR si uvedeme v části 4.2.2.

4.2.2 Vývoj aplikací v ONC RPC

Vývoj aplikací využívajících RPC probíhá ve dvou krocích:

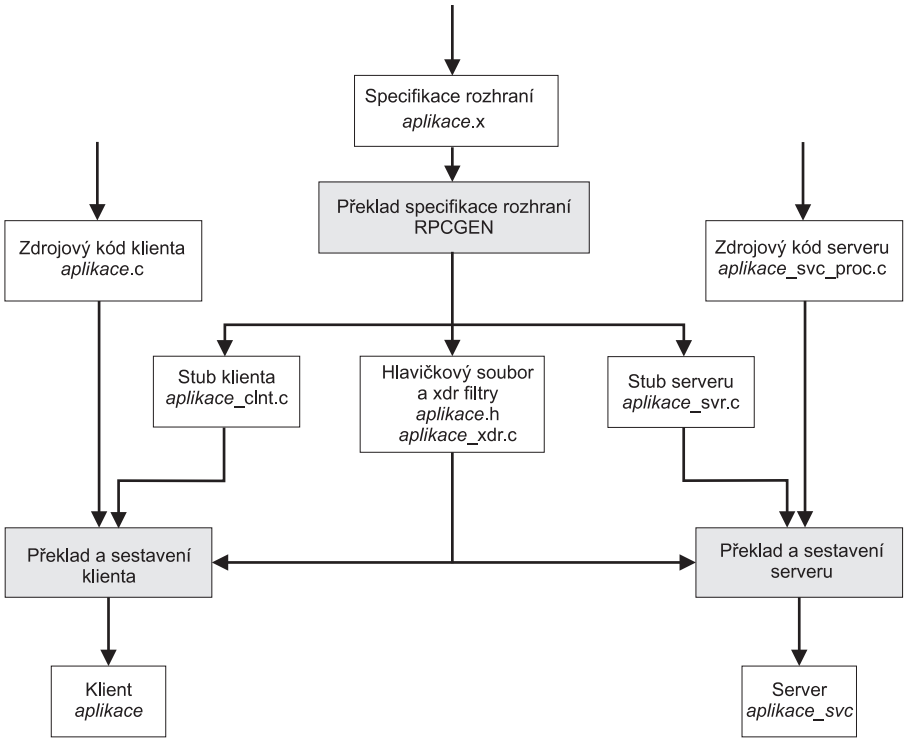
- specifikace rozhraní mezi klientem a serverem,

- vývoj klienta a serveru.

Nejprve je potřeba vytvořit definici rozhraní, ta je uložena v souboru *aplikace.x* (slovo *aplikace* nahrazujeme skutečným jménem vyvíjené aplikace). Z tohoto souboru překladač `rpcgen` vygeneruje spojky klienta a serveru *aplikace_clnt.c* a *aplikace_svc.c*, hlavičkový soubor rozhraní *aplikace.h* a rutiny XDR filtrů *aplikace_xdr.c*. Hlavičkový soubor aplikace je vhodné pojmenovat jménem různým od *aplikace.h*, protože jinak by byl přepsán při generování rozhraní.

Překladač ONC RPC se nazývá `rpcgen`. V modernější verzi TIRPC (Transport-Independent RPC) se překladač nazývá `rpcgen.new`. Pro vývojáře je výhodnější, protože oproti `rpcgen` může automaticky vytvořit šablonu zdrojového textu klienta a serveru.

Dále je potřeba napsat zdrojové texty klienta a serveru většinou pojmenované *aplikace.c* a *aplikace_svc_proc.c*. Na obr. 4.4 je uveden postup při vývoji, překladu a sestavení aplikace.



Obr. 4.4: Vývoj aplikace s použitím *rpcgen*

Definice rozhraní

Jazyk pro definici rozhraní RPCL je rozšířením definičního jazyka XDR a je stejně jako XDR velmi podobný jazyku C. Ukažme si na příkladech způsob zápisu definic a deklarací v tomto jazyce a překlad do jazyka C.

Definice

Jazyk XDR dovoluje definovat konstanty a s použitím konstruktorů `enum`, `struct` a `union` definovat složené datové typy. Jazyk RPCL přidává konstruktor rozhraní `program`.

Konstruktor `const` dovoluje definovat celočíselné konstanty. Příkladem může být definice velikosti bufferu:

```
const MAX_SIZE = 512;
```

rpcgen z tohoto zápisu vytvoří v hlavičkovém souboru definici konstanty:

```
#define MAX_SIZE 512
```

Definice výčtového typu má stejný zápis jako v jazyce C:

```
enum strana {
    SEVER=0,
    JIH=1,
    ZAPAD=2,
    VYCHOD=3
};
```

Výsledkem překladu bude:

```
enum strana {
    SEVER=0,
    JIH=1,
    ZAPAD=2,
    VYCHOD=3
};
typedef enum strana strana;
bool_t xdr_strana();
```

Program rpcgen dále v souboru *aplikace_xdr.c* vytvoří XDR filtr `xdr_strana()`:

```
bool_t xdr_strana(xdrs, objp)
    XDR *xdrs;
    strana *objp;
{
    if (!xdr_enum(xdrs, (enum_t *)objp)) {
        return (FALSE);
    }
    return (TRUE);
}
```

Podrobný popis filtrů XDR přesahuje rámec tohoto textu. Zjednodušeně lze říct, že proměnná `xdrs` ukazuje na strukturu určující typ prováděné operace nad XDR daty. Proměnná `objp` ukazuje na místo, ze kterého má konverzní funkce číst data, nebo kam tato data má uložit.

Struktury definujeme stejně jako v jazyce C. Příkladem může být bod zadaný souřadnicemi:

```
struct point {
    int x;
    int y;
};
```

V generovaném hlavičkovém souboru bude:

```
struct point {
    int x;
    int y;
};
typedef enum point point;
bool_t xdr_point();
```

Tentokrát bude v souboru *aplikace_xdr.c* vytvořen filtr `xdr_point()`:

```
bool_t xdr_point(xdrs, objp)
    XDR *xdrs;
    point *objp;
{
    if (!xdr_int(xdrs, &objp->x)) {
        return (FALSE);
    }
    if (!xdr_int(xdrs, &objp->y)) {
        return (FALSE);
    }
    return (TRUE);
}
```

Na dalším příkladu je vidět způsob zápisu variantního záznamu (`union`):

```
union result switch (int errno) {
    case 0:
        int x;
    default:
        void;
};
```

Překladem vznikne hlavičkový soubor:

```
struct result {
    int errno;
    union {
        int x;
    } result_u;
};
typedef struct result result;
bool_t xdr_result();
```

Pro definici typu budeme potom používat `typedef result`. Položky variantního záznamu budou ve výsledné struktuře označeny jménem `result` s příponou `_u`. Překladač opět generuje filtr `xdr_result()`.

K definici uživatelských typů slouží konstrukce `typedef`. Příkladem může být typ reprezentující čtyřúhelník:

```
typedef point ctyruhelnik[4];
```

V hlavičkovém souboru se objeví:

```
typedef point ctyruhelnik[4];
bool_t xdr_ctyruhelnik();
```

Zajímavější je však filtr `xdr_ctyruhelnik()`.

```
bool_t xdr_ctyruhelnik(xdrs, objp)
    XDR *xdrs;
    ctyruhelnik objp;
{
    if (!xdr_vector(xdrs, (char *)objp, 4, sizeof(point), xdr_point)) {
        return (FALSE);
    }
    return (TRUE);
}
```

Všimněte si, že funkce `xdr_vector` používá dříve vytvořený filtr `xdr_point`.

Poslední definicí je definice programu. Zde uvádíme číslo programu, čísla verzí, názvy a čísla vzdálených procedur. Jako příklad slouží program pro práci s účtem. Program obsahuje dvě funkce. Jedna vrací stav účtu, druhá přidává na účet.

```
program UCET_PROG {
  version UCET_VERS {
    long ZJISTI(void)=1;
    long PRIDEJ(long)=2;
  }=1;
}=0x31234567;
```

Číslo programu, zde 31234567_{H} , slouží k jednoznačnému rozlišení jednotlivých rozhraní. Toto číslo je voleno z intervalu podle tab. 4.1. Každé zaregistrované rozhraní na jednom počítači musí mít jiné číslo. (Výjimkou z tohoto pravidla je možnost registrovat současně více různých verzí jednoho rozhraní).

Rozsah hodnot	Popis
0x00000000 - 0x1FFFFFFF	Definováno firmou Sun
0x20000000 - 0x3FFFFFFF	Definuje uživatel
0x40000000 - 0x5FFFFFFF	Přechodné
0x60000000 - 0xFFFFFFFF	Rezervované

Tab. 4.1: Čísla programů ONC RPC.

V našem příkladě je definována jediná verze rozhraní s číslem 1. Procedura `ZJISTI` má přiřazeno číslo 1 a procedura `PRIDEJ` číslo 2. Nula je vyhrazena pro proceduru `NULLPROC`. Tato procedura je automaticky generovaná překladačem rpcgen. Jedná se o funkci, která umožňuje zjišťovat, zda je server spuštěný. Po překladu vznikne v hlavičkovém souboru tento zápis:

```
#define UCET_PROG ((u_long)0x31234567)
#define UCET_VERS ((u_long)1)
#define ZJISTI ((u_long)1)
extern long *zjisti_1();
#define PRIDEJ ((u_long)2)
extern long *pridej_1();
```

Z překladu je patrné, že jména procedur jsou jiná, než jsme uvedli v definici. Nová jména vznikla převedením velkých písmen v názvu procedury na malá a přidáním čísla verze za podtržítko. Protože se v některých systémech může naznačený způsob převedení jména poněkud lišit, doporučujeme zkontrolovat vzniklý hlavičkový soubor.

Deklarace

V RPCL máme čtyři typy deklarací proměnných, ty se poněkud liší od deklarací v jazyce C:

- simple,
- fixed array,
- variable array,
- pointer.

Jednoduché deklarace a deklarace polí s pevnou délkou jsou stejné jako v jazyce C a překlad je nemění.

```
strana s;
strana trasa[20];
```

ONC RPC nedovoluje použití vícerozměrných polí. Vícerozměrná pole však jde převést na pole jednorozměrná.

Rozšířením oproti jazyku C je použití polí proměnné délky. Takováto pole mohou, nebo nemusí mít určen maximální počet prvků.

```
strana zkratka<MAX_CESTA>;
strana bloudeni<>;
```

V prvním případě je délka pole shora omezena konstantou `MAX_CESTA`, ve druhém případě omezena není. Protože pole s proměnnou délkou jazyk C nezná, překladač vytvoří z první deklarace následující konstrukci:

```
struct {
  u_int  zkratka_len;
  strana *zkratka_val;
} zkratka;
```

V proměnných s koncovkou `_len` je udán počet prvků pole a v proměnných s koncovkou `_val` je ukazatel na toto pole. Rozdíl mezi jednotlivými typy polí je nejvíce patrný z XDR filtrů (pro srovnání uvádíme i XDR filtr pro pole s pevnou délkou):

```
if (!xdr_vector(xdrs, (char *)objp->trasa, 20, sizeof(strana), xdr_strana))
    {return (FALSE);}

if (!xdr_array(xdrs, (char **)&objp->zkratka.zkratka_val,
               (u_int *)&objp->zkratka.zkratka_len,
                MAX_CESTA, sizeof(strana), xdr_strana))
    {return (FALSE);}

if (!xdr_array(xdrs, (char **)&objp->bloudeni.bloudeni_val,
               (u_int *)&objp->bloudeni.bloudeni_len,
               0, sizeof(strana), xdr_strana))
    {return (FALSE);}
```

Použití ukazatelů je opět stejné jako v jazyce C. Ukazatele se používají pro tvorbu strukturovaných dat (stromy, seznamy). Pokud chceme definovat obecný polynom, nabízí se nám následující možnost:

```
struct point {
  int x;
  int y;
};
struct polynom {
  point   *p;
  polynom *pNext;
};
```

V hlavičkovém souboru bude vygenerovaný kód:

```
struct point {
  int x;
  int y;
};
typedef struct point point;
bool_t xdr_point();

struct polynom {
  point *p;
  struct polynom *pNext;
};
typedef struct polynom polynom;
bool_t xdr_polynom();
```

Speciální případy

Protože jazyk C nepoužívá typ boolean, je tento typ přeložen na typ `bool_t`. Například deklarace proměnné `flag`:

```
bool flag;
```

je přeložena na

```
bool_t flag;
```

Typ string je definován jako řetězec znaků ukončený prázdným znakem NULL. V RPCL máme string s neurčenou a maximální délkou:

```
string longBuff<>;
string buffer<20>;
```

Tyto proměnné jsou přeloženy:

```
char *longBuff;
char *buffer;
```

Další specialitou jsou blíže nedefinovaná data `opaque`. Jsou to data bez explicitního typování, při jejich předávání mezi klientem a serverem nejsou překládána. Data `opaque` mohou být pevné nebo proměnné délky:

```
opaque fixData[512];
opaque varData<1024>;
```

Překlad je následující:

```
char fixData[512];
struct {
  u_int varData_len;
  char *varData_val;
} varData;
```

Použití této struktury je stejné jako u polí proměnné délky.

Deklarace typu `void` je použitelná pouze v definici union a program. Obsluhu tohoto typu zajišťuje filtr `xdr_void`.

Knihovna ONC RPC

Pokud použijeme `rpcgen`, je většina volání funkcí skryta ve spojkách klienta a serveru. Přesto si uvedeme některé důležité funkce. Nejdříve uvedeme funkce, které se používají na straně klienta.

Spojení se serverem se vytváří voláním funkce

```
CLIENT *clnt_create(char *host, u_long prognum, u_long versnum,
char *protokol),
```

která vrací ukazatel na objekt typu `CLIENT`. Vstupními parametry je jméno serveru, na kterém hledáme vzdálenou proceduru, číslo programu, číslo verze a transportní protokol. Transportní protokol bývá zadán jako "udp" nebo "tcp". Pokud použijeme protokol UDP, nemůže být předávaný parametr větší než 8KB. Toto omezení není způsobeno samotným protokolem UDP (paket UDP může být dlouhý až 64KB), ale implementací RPC. Délka předávaných parametrů pomocí protokolu TCP je teoreticky neomezena. Funkce `clnt_create` se volá v kódu klienta.

Máme-li navázané spojení, lze jeho vlastnosti měnit pomocí funkce

```
bool_t clnt_control(CLIENT *clnt, int request, char *info).
```

Tato funkce pracuje s oběma transportními protokoly UDP i TCP. Význam požadavku `request` a typ argumentu požadavku je uveden v tab. 4.2.

Požadavek	Typ argumentu	Popis	Poznámka
CLSET_TIMEOUT	struct timeval	nastaví celkový timeout	
CLGET_TIMEOUT	struct timeval	informuje o velikosti celkového timeoutu	
CLGET_FD	int	vrátí ukazatel na připojený socket	
CLSET_FD_CLOSE	void	povolí uzavření socketu při volání clnt_destroy()	
CLSET_FC_NCLOSE	void	zakáže uzavření socketu při volání clnt_destroy()	
CLGET_SERVER_ADDR	struct sockaddr_in	vrací adresu serveru	
CLSET_RETRY_TIMEOUT	struct timeval	nastaví timeout opakování požadavku	pouze UDP
CLGET_RETRY_TIMEOUT	struct timeval	vrátí hodnotu timeoutu opakování požadavku	pouze UDP

Tab. 4.2: Parametry funkce `clnt_control()`.

Funkce volající vzdálenou proceduru identifikovanou číslem `procnum` má deklaraci

```
enum clnt_stat clnt_call(CLIENT *clnt, u_long procnum, xdrproc_t inproc,
char *in, xdrproc_t outproc, char *out, struct timeval timeout).
```

Pomocí XDR procedur `inproc` a `outproc` převádíme vstupní parametry a výsledek. Proměnná `timeout` definuje dobu, po kterou funkce čeká na výsledek. Parametr se ignoruje, byl-li timeout nastaven pomocí funkce `clnt_control`.

Poslední zde zmíněnou funkcí používanou na straně klienta je

```
void clnt_destroy(CLIENT *clnt).
```

Pomocí funkce `clnt_destroy` uvolníme paměť alokovanou `clnt_create`. Pokud jsme pomocí funkce `clnt_control` (request `CLSET_FD_NCLOSE`) nezakázali uzavření socketu asociovaného s objektem `CLIENT`, je touto funkcí uzavřen.

Dále si ukážeme tři funkce sloužící k inicializaci serveru. První dvě slouží k vytvoření obsluhy serveru a třetí je určena k registraci procedury.

Chceme-li vytvořit obsluhu serveru s protokolem UDP, zavoláme funkci:

```
SVCXPRT *svcudp_create(int sock)
```

Funkce vrátí ukazatel na obsluhu serveru. Vstupním parametrem je vytvořený socket, který pro proceduru chceme použít. Použijeme-li namísto socketu konstantu `RPC_ANYSOCK`, vytvoří se socket automaticky.

Obsluha serveru s protokolem TCP se vytváří funkcí:

```
SVCXPRT *svctcp_create(int sock, u_int sendsz, u_int recvsz)
```

Návratová hodnota a parametr `sock` fungují stejně jako v předchozí funkci. Pomocí parametrů `sendsz, recvsz` lze nastavit velikost přijímacího a vysílacího bufferu. Nastavíme-li nulovou hodnotu, použije se implicitní velikost.

Registraci serveru u *portmapperu* (v UNIXu bývá portmapper reprezentován démonem portmap) zajišťuje funkce

```
bool_t svc_register(SVCXPRT *xprt, u_long prognum, u_long versnum,
void (*dispatch)(), u_long protocol)
```

Všechny výše uvedené funkce bývají umístěny v souboru *aplikace_svc.c*, který je automaticky vytvořen překladačem rozhraní rpcgen.

V následujícím textu si informace uvedené v předchozím textu ukážeme na příkladu. Bude jím program, který vypíše obsah adresářů vzdáleného počítače. Jak již bylo řečeno, nejprve nadefinujeme rozhraní mezi klientem a serverem. Z hlediska zadání je zřejmé, že vstup naší procedury bude jméno adresáře, jehož obsah chceme vypsat. Návratovou hodnotou bude výpis adresáře. Protože jednotlivé položky výpisu mají různou délku a jejich počet je neznámý, pro předávání výsledku použijeme typ s dynamickým chováním. Řešení je vidět na příkladu:

```
const MAXNAMELEN = 255;
typedef string nametype<MAXNAMELEN>;
typedef struct namenode *namelist;

struct namenode {
  nametype name;
  namelist pNext;
};

union readdir_res switch (int errno) {
case 0:   namelist list;
default:  void;
};
```

```
program DIRPROG {
  version DIRVERS {
    readdir_res  READDIR(nametype) = 1;
  } = 1;
} = 0x20000001;
```

Definujeme zde typ `nametype` jako string. Proměnná tohoto typu bude sloužit k zaslání jména adresáře, jehož obsah chceme vypsat. Struktura `namenode` reprezentuje výpis obsahu adresáře. Obsahuje položku `name` typu `nametype`, ve které je jedna položka výpisu, a ukazatel `pNext` na následující položku seznamu. Každé položce ve výpisu adresáře tedy odpovídá jedna struktura `namenode`. Pro jednoduchost jsme navíc definovali typ `namelist`, který je ukazatelem na strukturu `namenode`.

Konstanta `MAXNAMELEN` je podobně jako nadefinované datové typy použitelná v kódu klienta i serveru.

Další součástí definice rozhraní jsou hodnoty uvedené v sekci `program`. Zde se uvádí definice jednotlivých procedur. V našem příkladě je to procedura `READDIR`. Procedura přebírá argument typu `nametype` a vrací výsledek typu `readdir_res`. Typ `readdir_res` je variantní záznam s rozhodovací proměnnou `errno`. Pokud je výsledkem chyba, procedura vrací prázdný ukazatel NULL, jinak vrací ukazatel na strukturu `namenode`.

Další informací v sekci `program` je číslo verze. Číslování verzí podporuje postupný vývoj aplikací. Programátor může vytvořit novou verzi serveru. Klient podle svého čísla verze určí, jakou verzi serveru bude využívat. Najednou tak mohou vedle sebe fungovat starší a novější verze téže aplikace.

Posledním údajem je číslo programu. Podle tohoto čísla se identifikuje rozhraní a musí být vždy na jednom počítači jedinečné, protože je možné zaregistrovat několik rozhraní se stejným jménem.

Pro lepší pochopení uvádíme hlavičkový soubor vzniklý překladem definice rozhraní.

```
#include <rpc/types.h>

#define MAXNAMELEN 255

typedef char *nametype;
bool_t xdr_nametype();

typedef struct namenode *namelist;
bool_t xdr_namelist();

struct namenode {
  nametype name;
  namelist pNext;
};
typedef struct namenode namenode;
bool_t xdr_namenode();

struct readdir_res {
  int errno;
  union {
    namelist list;
  } readdir_res_u;
};
typedef struct readdir_res readdir_res;
bool_t xdr_readdir_res();
```

```
#define DIRPROG ((u_long)0x20000001)
#define DIRVERS ((u_long)1)
#define READDIR ((u_long)1)
extern readdir_res *readdir_1();
```

Všimněte si, že každá definice typu je následovaná definicí konverzní funkce. Dále si všimněte, že proměnná union má příponu _u. Pomocí tohoto jména se na union odkazujeme ve vlastním programu.

Pokud máme definované rozhraní, můžeme přistoupit k vývoji vzdálené procedury. Procedura vypíše získaný adresář a vrátí výsledek. Vznikne-li při běhu procedury chyba (špatné jméno adresáře, nedostatečná práva), procedura nevrací žádný výsledek, pouze kód chyby:

```
#include <rpc/rpc.h>
#include <sys/dir.h>
#include "rls.h"

extern int errno;
extern char *malloc();
extern char *strdup();

readdir_res *readdir_1(dirname)
  nametype *dirname;
{
  namelist nl;
  namelist *nlp;
  static readdir_res res;
  static DIR *dirp = NULL;
  struct direct *d;

  dirp = opendir(*dirname);
  if (dirp == NULL) {
    res.errno = errno;
    return(&res);
  }

  if (dirp) xdr_free(xdr_readdir_res, &res);

  nlp = &res.readdir_res_u.list;
  while (d = readdir(dirp)) {
    nl = *nlp = (namenode *)malloc(sizeof(namenode));
    nl->name = strdup(d->d_name);
    nlp = &nl->pNext;
  }
  *nlp = NULL;

  res.errno = 0;
  closedir(dirp);
  return(&res);
}
```

Povšimněme si detailů, které jsou důležité z pohledu používání RPC. Pokud vyvíjíme aplikaci s použitím rpcgen, je třeba vložit hlavičkový soubor *rls.h* a standardní hlavičkové soubory.

Na straně serveru definujeme pouze proceduru (tělo programu `main` je obsaženo v generovaném souboru *aplikace_svc.c*). Jméno procedury vznikne převedením velkých písmen

ve jméně uvedeném v souboru s definicí rozhraní (`READDIR`) na písmena malá a připojením čísla verze na konec jména (`readdir_1`). V Linuxu je převod jména procedury poněkud jiný. Navíc se liší pojmenování procedury v klientu a serveru.

Důležité je, aby proměnná, ve které se vrací výsledek, byla třídy `static`. Pokud by byla jiné třídy, mohla by být v okamžiku předávání hodnoty do spojky nedefinovaná. Z podobného důvodu uvolňujeme paměť přidělenou XDR objektu (`xdr_free()`) až v následujícím volání procedury. Zajímavé je také použití unionu `readdir_res_u`.

Máme-li připravenou proceduru, můžeme vytvořit klientský program. Oproti programu využívajícímu pouze lokálního volání procedur musíme inicializovat spojení použitím funkce `clnt_create`. V našem příkladu vytváříme spojení a volíme transportní protokol TCP. Získaného ukazatel používáme jako poslední parametru ve volání vzdálené procedury `readdir_1`.

K výpisům chybových hlášení lze v prostředí RPC využít řady funkcí. Zde jsme použili funkce `clnt_perror()` a `clnt_pcreateerror()`. Příklad klienta následuje:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include "rls.h"

extern int errno;

main (argc, argv)
  int argc;
  char *argv[];
{
  CLIENT *cl;
  char *server;
  char *dir;
  readdir_res *result;
  namelist nl;

  if (argc != 3) {
    fprintf(stderr, "usage: %s host directory\n", argv[0]);
    exit(1);
  }

  server = argv[1];
  dir = argv[2];

  cl = clnt_create(server, DIRPROG, DIRVERS, "tcp");
  if (cl == NULL) {
    clnt_pcreateerror(server);
    exit(1);
  }

  result = readdir_1(&dir, cl);
  if (result == NULL) {
    clnt_perror(cl, server);
    exit(1);
  }

  if (result->errno != 0) {
    errno = result->errno;
    perror(dir);
    exit(1);
```

```
    }

    for (nl = result->readdir_res_u.list; nl != NULL; nl = nl->pNext) {
      printf("%s\n", nl->name);
    }
    exit(0);
  }
```

V tomto okamžiku je aplikace připravena k překladu. Překlad bývá většinou automatizován programem `make`. Běh tohoto programu je ovlivněn řídícím souborem `makefile`. Pro příklad si takový soubor uvedeme:

```
APPN=rls
#LFLAGS=
CFLAGS= -g -DDEBUG
RPCGEN = rpcgen

CLIENT_AND_SERVER: $(APPN) $(APPN)_svc

all: $(CLIENT_AND_SERVER)

# protokol
$(APPN)_xdr.c $(APPN)_svc.c $(APPN)_clnt.c $(APPN).h: $(APPN).x
  $(RPCGEN) $(APPN).x

# klient
$(APPN): $(APPN).h $(APPN).o $(APPN)_clnt.o $(APPN)_xdr.o
  cc $(CFLAGS) -o $(APPN) $(APPN).o $(APPN)_clnt.o $(APPN)_xdr.o \
  $(LFLAGS)

# server
$(APPN)_svc: $(APPN).h $(APPN)_svc_proc.o $(APPN)_svc.o \
$(APPN)_xdr.o
  cc $(CFLAGS) -o $(APPN)_svc $(APPN)_svc_proc.o $(APPN)_svc.o \
  $(APPN)_xdr.o $(LFLAGS)
```

Struktura souboru *makefile* není složitá, v první části jsou uvedeny konstanty. V druhé jsou uvedeny operace, které se mají s danými soubory provést. Před dvojtečkou jsou uvedena jména souborů, které se v daný okamžik budou měnit. Za dvojtečkou jsou soubory, při jejichž změně se má provést operace uvedená na následující řádce. Změna je vyvolána tím, že některý ze souborů uvedených za dvojtečkou je mladší než soubory před dvojtečkou. Při použití programu `make` se provádějí pouze operace, které jsou nutné.

Po překladu už můžeme aplikaci spustit. Nejdříve na jednom počítači spustíme server. Server se většinou spouští na pozadí příkazem

```
rls_svc &
```

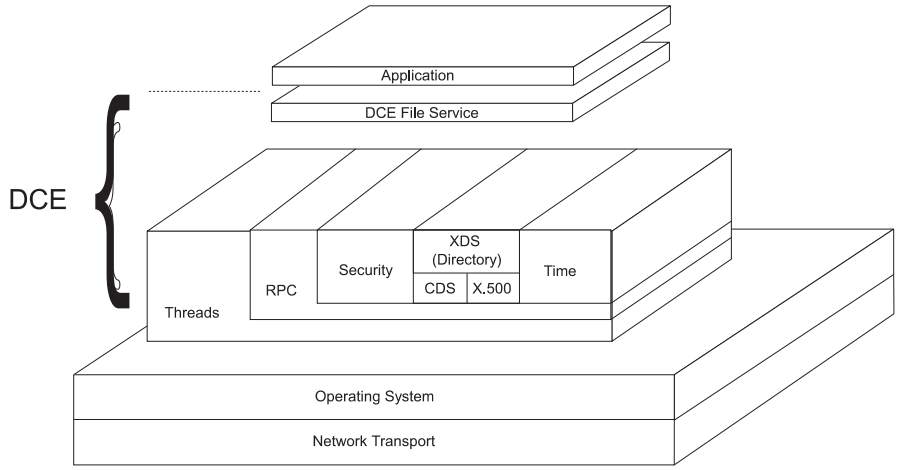
Na dalším počítači (ale klient může být spuštěn i na stejném počítači jako server), spustíme klienta např. příkazem

```
rls hpk ./
```

První parametr je jméno počítače, na kterém je spuštěn server, druhý parametr obsahuje jméno adresáře, jehož obsah chceme vypsat.

4.3 OSF DCE

Podobně jako ONC je DCE (*Distributed Computing Environment*) skupina produktů. Jejich provázanost je však větší a nemohou samostatně fungovat. Lepší představu o DCE získáme z blokového schématu na obr. 4.5.



Obr. 4.5: Model DCE

Network transport je libovolný typ síťového propojení (TCP/IP, DECNET, IPX) včetně programového vybavení.

Operating System je operační systém podporující paralelní zpracování. V současné době se jedná o širokou řadu operačních systémů: různé verze UNIXu, VMS, Windows 3.x, 95, NT i MS DOS. U některých operačních systémů však nejsou implementovány všechny části DCE. Chceme-li tedy používat DCE na těchto systémech, musíme přidat počítače s "dospělejším" systémem (tento rys však u distribuovaných systémů nebývá na závadu), na kterých budou instalovány některé nezbytné části DCE.

Threads (vlákna) je část DCE, která podporuje vytváření, běh a rušení vláken pro možnost vytváření neblokujících RPC. Vlákna jsou obdoby procesů jak je známe z operačního systému UNIX, oproti procesům mají nižší režii a používají společný adresový prostor.

RPC je modul podporující psaní RPC. Zahrnuje API (aplikační rozhraní) a runtime knihovnu. RPC je provázáno se všemi ostatními částmi DCE.

Security umožňuje používání bezpečnostních funkcí na úrovni DCE, k tomuto účelu využívá centralizované replikovatelné databáze. Pomocí speciálních programů je možné v některých systémech zavést tzv. *integrovaný login*. Integrovaný login nahradí autentizační prostředky daného systému prostředky DCE. Objekt se potom nepřihlašuje na jednotlivý počítač, ale do celého DCE systému.

XDS (X.500 Global Directory Service) slouží jako adresářová služba, která umožňuje adresaci podle normy X.500 a pomocí DNS (Domain Name Service). Tato služba dovoluje ukládat a nalézt informace o síťových prostředcích v jedné distribuované databázi. Mohou to být informace o uživatelích, počítačích, službách a dalších prostředcích. Adresářová služba CDS (Cell Directory Service) spravuje jmenný prostor uvnitř buňky. Pokud potřebujeme informace z jiné buňky, kontaktujeme GDS (Global Directory Service), který kontaktuje CDS jiné buňky.

Distributed Time Service (DTS) vytváří podmínky pro používání distribuovaného času. Tento modul je v distribuovaném prostředí velmi důležitý, protože např. modul security nemůže bez stejného času v celé DCE buňce fungovat.

Distributed File System (DFS) zprostředkovává přístup k souborům. Je to obdoba NFS známého z ONC. V DFS je zavedena vyšší bezpečnost a pohled na DFS strom je ze všech stanic totožný. Na rozdíl od NFS má DFS centralizovanou správu souborového stromu.

Vlastní aplikace může, nebo nemusí využívat všechny části DCE.

V předchozím textu jsme použili termín *buňka* (Cell). Buňka je základní prvek pro správu v DCE - skupina počítačů tvořící logický celek. Protože mezi všemi částmi buňky probíhá značná komunikace mohlo by se zdát, že počítače v jedné buňce si musí být komunikačně blízké. Nemusí tomu ale tak být, neboť služby potřebné pro funkci buňky lze replikovat.

Z principu fungování DCE je nutné, aby v každé buňce byly přítomny tyto služby:

- security server,
- CDS server,
- DTS server.

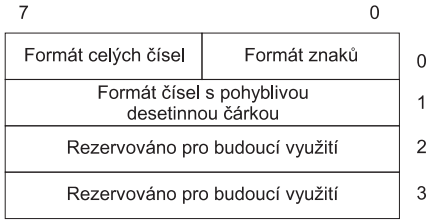
Pochopitelně takto minimálně konfigurovaná buňka neposkytuje všechny možnosti DCE.

4.3.1 Vývoj aplikací v OSF DCE

Na úvod části věnované popisu vývoje aplikací v DCE uvedeme, podobně jako v části věnované ONC RPC, způsob reprezentace dat pro komunikaci.

Network data representation

DCE RPC používá pro reprezentaci dat formát NDR (*Network Data Representation*). Jak jsme již uvedli, mohou mít stejná data ve formátu NDR různou reprezentaci. Informace o použité reprezentaci je uložena v NDR hlavičce (*NDR format label*). NDR hlavička je uvedena na obr. 4.6.



Obr. 4.6: NDR hlavička

V současnosti jsou využívány pouze dva oktety. Nižší čtyři bity prvního oktetu určují kód pro zobrazení znaků (ASCII, EBCDIC), vyšší čtyři bity určují pořadí oktetů v reprezentaci celočíselných hodnot a hodnot s plovoucí desetinnou čárkou (big-endian, little-endian). Druhý oktet určuje reprezentaci čísel v plovoucí desetinné čárce (IEEE 754, VAX, Cray, IBM).

Podobně jako ve formátu XDR máme k dispozici řadu typů:

- boolean (8 bitů, FALSE=0, TRUE≠0),
- character (8 bitů),
- integer,

  - small (8 bitů),
  - short (16 bitů),
  - long (32 bitů),
  - hyper (64 bitů),
- enumerated type (16 bitů),
- floating-point number
  - single (32 bitů),
  - double (64 bitů),
- uninterpreted octet (8 bitů) - typ, pro který není definována formátová konverze,
- union,
- structure,
- array - jedno i vícerozměrná pole, pevné i proměnné délky,
- string - množina znaků (jedno i víceoktetových) ukončená nulou (pokud jsou znaky n-oktetové, je potřeba řetězec ukončit n nulami),
- pipe - sekvence po sobě jdoucích znaků, jejichž množství je definováno dynamicky a teoreticky není limitováno,
- pointer
  - full pointer - může být NULL i alias,
  - reference pointer - nemůže být NULL a alias.

Podrobnější popis jazyka NDR naleznete v [13].

Definice rozhraní

V prostředí DCE definujeme RPC rozhraní pomocí jazyka pro definici rozhraní IDL (*Interface Definition Language*). Definice rozhraní je uložena v souboru *aplikace.idl*. Překladač protokolu se nazývá `idl`. Vytváření aplikace je tedy velmi podobné ONC RPC. Definice rozhraní je tvořena dvěma částmi:

- hlavička rozhraní (je ohraničena hranatými závorkami)
  - atributy hlavičky,
  - jméno rozhraní,
- tělo rozhraní (je ohraničeno složenými závorkami)
  - import z jiných idl souborů,
  - definice konstant,
  - definice datových typů,
  - definice procedur.

V *hlavičce rozhraní* jsou uvedeny následující povinné atributy:

- *UUID (Universal Unique IDentifier)*. UUID je proměnná o velikosti 128 bitů, která slouží k jednoznačné identifikaci rozhraní. Aby byla tato jednoznačnost zaručena, je UUID generované. Generovací algoritmus využívá aktuální čas a číslo síťové karty. Generátor se nazývá `uuidgen`. `uuidgen` může také vytvořit kostru celého idl souboru.

- *jméno rozhraní* je uvedeno za klíčovým slovem `interface`, které se nachází za definicí hlavičky. Jméno může být tvořeno nejvýše 17 znaky a narozdíl od UUID není jedinečné. Je vhodné, aby jméno rozhraní bylo stejné jako jméno idl souboru.

Nejkratší možná definice hlavičky s názvem rozhraní banka je následující:

```
[uuid(C985A380-255B-11C9-A50B-08002B0ECEF1)]
interface banka
```

Nepovinné atributy jsou:

- *verze*, která napomáhá postupnému vývoji aplikace. Verze je tvořena ze dvou čísel, hlavního (major) a vedlejšího (minor), od sebe oddělených tečkou.

```
version(1.0)
```

Při kontrole verze může klient navázat spojení se serverem, jehož hlavní číslo je stejné jako u klienta a vedlejší číslo serveru je větší nebo rovno vedlejšímu číslu klienta. Pokud číslo verze neuvedeme, nastaví se na hodnotu 0.0. Při postupném vývoji bychom měli při přidání nových typů nebo procedur zvýšit vedlejší číslo. Pokud změníme počet nebo význam parametrů v proceduře, zvýšíme hlavní číslo verze.

- *koncový bod (endpoint)*, jehož součástí je typ protokolu a číslo. Neuvedeme-li tento atribut, je koncový bod přidělen automaticky.

- *pointer_default* nastavuje implicitní typ ukazatelů (ref a ptr).

V *těle rozhraní* definujeme konstanty, datové typy a procedury.

Konstanty mohou být typu integer, boolean, character, string a null pointer. Deklarace konstant má tvar:

```
const long MAX=256;
const void* AB=NULL;
```

Deklarace *datových typů* je oproti jazyku C rozšířena o atribut typu uzavřený v hranatých závorkách. Deklarace typů mohou být:

```
typedef long cislo_zbozi;
typedef [ptr] cislo_zbozi *cislo_zbozi_p;
typedef char nazev_zbozi[MAX+1];
```

Atributy datových typů jsou:

- handle - data obsahují informace o vazbě mezi klientem a serverem,
- context_handle - data jsou využívána pro předávání stavové informace stavového serveru,
- transmit_as(typ) - data budou konvertována před posíláním přes síť do daného typu,

- ref - referenční ukazatel,
- ptr - plný ukazatel,
- string - data mají vlastnosti řetězce.

Identifikátory datových typů jsou:

- základní typy - integer, floating point number, character, boolean, byte a void,
- vytvořené typy - pointer, array, string, struct, union, enum a pipe,
- předdefinované typy - error_status_t, ISO_LATIN_1, ISO_MULTI_LINGUAL a ISO_UCS.

V následujících příkladech podrobněji probereme definice některých typů. Začněme definicí *variantního záznamu*. Pokud chceme vytvořit záznam, který bude obsahovat podle typu jednotek (kusy, kilogramy, gramy) jejich množství (celočíselný nebo desetinný údaj):

```
typedef enum {
   KUS, GRAM, KILOGRAM
} polozka_jednotky;

typedef union switch(polozka_jednotky jednotky) celkem {
   case ITEM:     long int pocet;
   case GRAM:
   case KILOGRAM: double   vaha;
} polozka_mnozstvi;
```

V hlavičkovém souboru bude po překladu:

```
typedef enum {ITEM=0,
  GRAM=1,
  KILOGRAM=2} polozka_jednotky;

typedef struct  {
   polozka_jednotky jednotky;
   union  {
     /* case(s): 0 */
     idl_long_int pocet;
     /* case(s): 1, 2 */
     idl_long_float vaha;
   } celkem;
 } polozka_mnozstvi;
```

V programu se takto definovaný typ používá:

```
polozka_mnozstvi mnozstvi;

if(mnozstvi.jednotky == KUS)
    printf("celkem %ld kusu\n", mnozstvi.celkem.pocet);
                else if(mnozstvi.jednotky == GRAM)
                   printf("celkem %10.2f gramu\n", mnozstvi.celkem.vaha);
                else if(mnozstvi.jednotky == KILOGRAM)
                   printf("celkem %10.2f kilogramu\n", mnozstvi.celkem.vaha);
```

Pokud bychom vynechali jméno variantního záznamu (zde `celkem`), idl automaticky dosadí jméno `tagget_union`.

Podobně jako v ONC lze v DCE použít různé typy *polí*:

- konstantní pole, jejichž velikost je dána při překladu,
- proměnná pole, která mají danou velikost, ale po síti se posílá pouze určitý úsek pole,
- přizpůsobitelná pole, jejich velikost je určena za běhu programu a přes síť se posílá pouze potřebná část.

Definice konstantních polí je stejná jako v jazyce C.

```
int pole[256];
```

U proměnných polí se používá deklarace:

```
const long SIZE 256;

void proc(
 [in] long first,
 [in] long length,
 [in, first_is(first), length_is(length)] int pole[SIZE]
);
```

Předání dat proměnného pole se potom v programu provádí voláním definované procedury.

```
long first = 102;
long length = 50;
int pole[size];

proc(first, length, pole);
```

Takto zapsaný požadavek přenese položky 102 až 151 (od délky musíme odečíst 1). Nechceme-li uvádět velikost přenášeného úseku pole, můžeme použít místo velikosti přenášeného úseku `(length_is)` pozici poslední přenesené položky `(last_is)`. Uvedený zápis pomocí definice procedury je možné nahradit zápisem používající definici struktury, který je rovnocenný.

Pole s přizpůsobitelnou délkou můžeme například použít pro uvedení čísel položek, jejichž počet neznáme. Pole s přizpůsobitelnou délkou potom definujeme zápisem:

```
typedef struct polozka_list{
   long                     size;
   [size_is(size)] polozka_cis cisla[*];
} polozka_list;
```

Použití tohoto pole v definici procedury, která vrací skupinu podpoložek, by mohlo vypadat:

```
void jake_jsou_podpolozky([in] polozka_cis cislo,
   [out] polozka_list **podpolozky);
```

Délka pole je skryta ve struktuře `podpolozky`. Když použijeme pole s přizpůsobitelnou délkou jako parametr, musí být délka pole uvedena jako další parametr. Jako příklad uvedeme proceduru, která vrací n podpoložek:

```
void vrat_n_podpolozek([in] polozka_cis cislo, [in] long n,
      [out, size_is(n)] polozka_cis podpolozky[]);
```

Podobně jako u proměnných polí položka `size_is(size)` definuje pole s prvky [0] až [size-1]. Použitím atributu `max_is(max)` definujeme pole s prvky [0] až [max].

V případě použití polí s přizpůsobitelnou délkou musíme pro tato pole alokovat paměť. Pokud budou tato pole uvedena jako součást struktury, musí být pole s přizpůsobitelnou délkou uvedeno jako poslední položka.

Atributy procedury určují atribut výsledku a sémantiku volání vzdálené procedury. Atributy výsledku mohou být string, ptr, contex_handle a není je potřeba vysvětlovat. Atributy sémantiky volání vzdálené procedury jsou:

idempotent - volání nejméně jednou, je vhodné pro idempotentní procedury.

maybe - vhodné pouze pro procedury, které nevracejí výsledek a nemají výstupní [out] parametr. Toto volání předá požadavek a nečeká na odpověď. Není znám výsledek volání, ani zda procedura proběhla.

broadcast - volání je předáno všem počítačům lokální sítě. Zpracována je první odpověď, ostatní se nepoužijí. Tato sémantika je podporována pouze při použití nespojovaného transportního protokolu (UDP). Při použití této sémantiky obdržíme výsledek nejrychleji, jak je to možné, ovšem zvýšíme zatížení sítě.

Implicitní hodnotou je sémantika *nejvýše jednou*. Případné poruchy musíme u této sémantice řešit opakovaným voláním. Duplicitu požadavků lze odstranit číslováním dotazů.

Procedura v DCE může mít několik parametrů, které mohou být vstupní `[in]`, výstupní `[out]` nebo vstupně-výstupní `[in,out]` (bráno z pohledu serveru). Další parametry atributů určují například typ pole a typ ukazatele a byly zmíněny dříve.

Definice rozhraní může být doplněna souborem ACF (Attribute Configuration File). Tímto souborem můžeme ovlivnit některé vlastnosti RPC. Pomocí souboru ACF lze například určovat způsob navazování spojení, ovlivnit překladač, aby negeneroval spojku pro některé procedury, a rozhodovat o umístění rutin pro marshaling a unmarshaling buď ve spojce, nebo v individuálním souboru. ACF soubor má tvar:

```
[explicit_handle]
interface bank
{
}
```

Výše uvedené informace si ukážeme na příkladu. Jako příklad použijeme program, který bude obsluhovat sklad. Každá položka je reprezentována strukturou `polozka_record`. Tato struktura obsahuje číslo, jméno a popis položky. Dále je ve struktuře uvedena cena za měrnou jednotku a množství, které je na skladě. Posledním údajem je pole s přizpůsobitelnou délkou, které obsahuje čísla podpoložek, tedy položek patřících pod daný druh.

Pomocí programu můžeme zjišťovat, zda je položka na skladě, vypsat její popis, zjistit množství a jaká čísla podpoložek položka obsahuje.

```
[
uuid(41cd5a62-f1ee-11d0-ad83-08002bbc062a),
version(1.0),
pointer_default(ptr)
] interface  sklad
{
```

```
    const long MAX_STRING = 30;
    typedef long     polozka_cis;
    typedef [string] char polozka_jmeno[MAX_STRING+1];
    typedef [string, ptr] char *veta;
    typedef enum {
       KUS, GRAM, KILOGRAM
    } polozka_jednotky;
    typedef struct polozka_cena {
       polozka_jednotky jednotky;
       double     cena_jednotky;
    } polozka_cena;
    typedef union switch(polozka_jednotky jednotky) celkem {
       case KUS:     long int pocet;
       case GRAM:
       case KILOGRAM: double   vaha;
    } polozka_mnozstvi;
    typedef struct polozka_list{
       long                      size;
       [size_is(size)] polozka_cis cisla[*];
    } polozka_list;
    typedef struct polozka_record {
       polozka_cis      cislo;
       polozka_jmeno    jmeno;
       veta             popis;
       polozka_cena     cena;
       polozka_mnozstvi mnozstvi;
       polozka_list     podpolozky;
    } polozka_record;

    boolean polozka_je_dostupna([in] polozka_cis cislo);
    veta popis_polozky([in] polozka_cis cislo);
    void jake_je_mnozstvi([in] polozka_cis number,
                          [out] polozka_mnozstvi *mnozstvi);
    void jake_jsou_podpolozky([in] polozka_cis cislo,
                              [out] polozka_list **podpolozky);
}
```

Klient

Kód klienta je ovlivněn způsobem vytvoření vazby se serverem. K vytvoření vazby potřebujeme znát informace o této vazbě (binding information). Informace o vazbě obsahují informaci o komunikačním protokolu, jméno nebo adresu počítače se serverem a adresu serveru na hostitelském počítači (endpoint), strukturu obsahující informace o vazbě nazýváme *binding handle*, v dalším textu budeme namísto tohoto termínu používat termín *ovladač vazby*. V DCE je několik metod vytváření vazby.

Automatická vazba zcela skrývá navázání spojení ve spojce klienta. Spojka automaticky získá informace o spojení z databáze jmen (CDS). Pokud se spojení ztratí, spojka se jej automaticky pokusí obnovit. Automatická vazba je základní metodou navázání spojení a je použita, pokud nespecifikujeme metodu jinou.

Implicitní vazba dovoluje omezenou kontrolu navázání spojení. V klientu definujeme společný ovladač vazby, který je použit pro všechny volané procedury. Klient není schopen automaticky obnovit spojení.

Explicitní vazba poskytuje největší možnost ovládání spojení. Při použití této metody může být každá volaná procedura na jiném počítači. Podobně jako implicitní metoda tato metoda automaticky neobnovuje přerušené spojení.

Pokud v programu použijeme automatickou metodu navazování spojení, je klient naprosto stejný, jako bychom použili lokální volání. Pouze je nutné v systémové proměnné `RPC_DEFAULT_ENTRY` určit místo, kde se nacházejí informace o vazbě. V systému VMS se tato proměnná nastavuje příkazem

```
define RPC_DEFAULT_ENTRY "/.:/sklad_cs.felk.cvut.cz"
```

z příkazového řádku.

Klient bude hledat informace o spojení v kořenovém adresáři buňky */.:/* pod jménem *sklad_cs.felk.cvut.cz*. Pokud bychom chtěli informaci o spojení získat z jiné buňky, musíme jméno zadat od absolutního kořene. Jméno má potom tvar */.../cell.felk.cvut.cz/sklad_cs.felk.cvut.cz*. Absolutní kořen je označen */.../* a jméno *cell.felk.cvut.cz* je jméno buňky.

Použití implicitní metody spojení je uvedeno v ACF souboru:

```
[implicit_handle(handle_t jmeno_global_handle)]
interface  sklad
{
}
```

Při použití implicitní metody spojení musí klient nejprve získat ovladač vazby. ovladač vazby pak spojka použije ve všech voláních vzdálených procedur. Získání ovladače vazby není zcela triviální a je popsáno v [14].

Explicitní navázání spojení můžeme použít pro všechny procedury klienta, nebo pouze pro některé. Pokud zvolíme první možnost, uvedeme v ACF souboru společný ovladač vazby:

```
[explicit_handle]
interface  sklad
{
}
```

Chceme-li použít expicitní vazbu pouze pro vyjmenované procedury uvedeme je v těle definice rozhraní:

```
[auto_handle]
interface  sklad
{
[explicit_handle] polozka_je_dostupna();
}
```

V IDL souboru musíme vyjmenované proceduře s explicitní vazbou přidat jako první parametr ovladač vazby, což nemusíme udělat, použijeme-li explicitní metodu pro všechny procedury rozhraní.

```
boolean polozka_je_dostupna([in] handle_t binding_h, [in] polozka_cis cislo)
```

Podobně jako u implicitní metody získáme ovladač vazby, kterých může být při použití explicitní metody několik. Tyto ovladače použijeme při volání procedur:

```
if polozka_je_dostupna(binding_h, cislo);
    puts("Na sklade: ANO");
  else
    puts("Na sklade: NE");
```

Pokud tedy použijeme automatickou vazbu, bude klientská část našeho programu vypadat takto:

```
#include <stdio.h>
#include <stdlib.h>
#include "sklad.h"

char instrukce[] = "Napis pismeno nasledovane parametry.\n
   Je polozka na sklade?                a  [cislo_polozky]\n\
   Popis polozky                        d  [cislo_polozky]\n\
   Jake mnozstvi polozky je na sklade?  q  [cislo_polozky]\n\
   Jake jsou podpolozky polozky?        s  [cislo_polozky]\n\
   Objednavka polozky                   o   cislo_polozky mnozstvi\n\
   HELP                                 r\n\
   KONEC                                e\n";

main()
{
   polozka_record polozka;
   polozka_list   *podpolozky;
   ucet_cis ucet = 1234;

   int i, num_args, done = 0;
   long result;
   char vstup[100], vyber[20], mnozstvi[20];
   char *strcpy();

   puts(instrukce);
   polozka.cislo = 0;
   strcpy(mnozstvi, "");
   while(!done) {
      printf("Vyber: ");  fflush(stdout);  gets(vstup);
      num_args = sscanf(vstup, "%s%ld%s", vyber, &(polozka.cislo), mnozstvi);

      switch (tolower(vyber[0])) {
      case 'a': if (polozka_je_dostupna(polozka.cislo))
                   puts("dostupna: ANO");
                else
                   puts("dostupna: NE");
                break;
      case 'd': polozka.popis = popis_polozky(polozka.cislo);
                printf("popis:\n%s\n", polozka.popis);
                if(polozka.popis != NULL)
                   free(polozka.popis);
                break;
      case 'q': jake_je_mnozstvi(polozka.cislo, &(polozka.mnozstvi));
                if(polozka.mnozstvi.jednotky == KUS)
                   printf("celkem kusu:%ld\n", polozka.mnozstvi.celkem.pocet);
                else if(polozka.mnozstvi.jednotky == GRAM)
                   printf("celkem gramu:%10.2f\n", polozka.mnozstvi.celkem.vaha);
                else if(polozka.mnozstvi.jednotky == KILOGRAM)
                   printf("celkem kilogramu:%10.2f\n",
                           polozka.mnozstvi.celkem.vaha);
                break;
      case 's': jake_jsou_podpolozky(polozka.cislo, &podpolozky);
                for(i = 0; i < podpolozky->size; i++)
                   printf("%ld  ", podpolozky->cisla[i]);
                printf("\npocet podpolozek:%ld\n", podpolozky->size);
```

```
                free(podpolozky);
                break;

        case 'r':
        default:  puts(instrukce);  break;
        case 'e': done = 1;  break;
        }
    }
}
```

Server

Server je obyčejně tvořen dvěma částmi. V jedné jsou implementované procedury a druhá slouží k inicializaci serveru. Inicializace serveru na rozdíl od ONC RPC není generována překladačem rozhraní, ale musí ji napsat programátor. Inicializace probíhá v následujících krocích:

1. Registrace rozhraní.
2. Příprava binding informací a výběr komunikačního protokolu, nebo protokolů.
3. Zveřejnění informace o umístění serveru. Server může informace umístit do jmenné databáze (CDS nebo jiné), nebo je zveřejnit jinak (např. vytisknout).
4. Zapsání koncových bodů (endpoint) do lokálního seznamu koncových bodů (endpoint map).
5. Čekání na vzdálené volání.

V následujících příkladech si ukážeme jednotlivé kroky inicializace. Nejprve je potřeba připravit datové struktury a připojit hlavičkové soubory.

```
#include <stdio.h>
#include <ctype.h>
#include "sklad.h"
#include "check_status.h"
#define STRINGLEN 50

main (argc, argv)
int argc;
char *argv[];
{
   unsigned32           status;
   rpc_binding_vector_t *binding_vector;
   rpc_protseq_vector_t *protseq_vector;

   char entry_name[STRINGLEN];
   char group_name[STRINGLEN];
   char annotation[STRINGLEN];
   char hostname[STRINGLEN];
   char *strcpy(), *strcat();
....
```

Proměnná `status` se používá pro předávání kódu chyby. `binding_vector` a `protseq_vector` jsou vektory používané funkcemi RPC run-time knihovny. Do ostatních proměnných se ukládají jména využívaná různými RPC službami.

Všechny servery musí mít rozhraní zaregistrované. Při registraci rozhraní využíváme *interface handle*. Je to struktura, která vznikne překladem idl souboru. V našem příkladu se tato struktura nazývá `sklad_v1_0_s_ifspec`. Jméno je složeno ze jména rozhraní `sklad`, čísla verze `1_0` (pokud není v idl souboru číslo verze uvedeno, použije se `0_0`), označení, zda se ukazatel používá na straně serveru `_s` nebo klienta `_c` a přípony `_ifspec`. Registraci provádí funkce

```
void rpc_server_register_if(rpc_if_handle_t handle, uuid_t *mgr_type_uuid,
                            rpc_mgr_epv_t mgr_epv, unsigned32 *status);
```

kde `handle` ukazuje na rozhraní a `status` vrací chybový stav. Proměnné `mgr_type_uuid` a `mgr_epv` se používají při registraci několika rozhraní se stejným jménem, kdy se bere ohled na uuid. Jinak obsahují obě proměnné hodnotu NULL. Registrace má v našem příkladě tvar:

```
rpc_server_register_if(sklad_v1_0_s_ifspec, NULL, NULL, &status);
CHECK_STATUS(status, "Nemohu zaregistrovat rozhrani:", ABORT);
```

Pokud nastane chyba, makro `CHECK_STATUS` vypíše zadaný text a chybové hlášení podle hodnoty proměnné `status`. Nakonec provede námi uvedenou operaci `ABORT` nebo `RESUME`. Makro využívá funkci `dce_error_inq_text()`.

Informace o vazbě (server binding information) vznikají při výběru protokolu v době inicializace serveru. Většina systémů podporuje protokoly UDP a TCP. Tyto protokoly jsou v DCE označeny `ncadg_ip_udp` a `ncacn_ip_tcp`. Při výběru protokolu je registrován koncový bod (port), s jehož pomocí se klient spojuje se severem.

Server může registrovat koncový bod dynamicky nebo staticky. Statická registrace použije při každém spuštění stejný koncový bod, který proto nazýváme dobře známý (well-known endpoints). Použití statické registrace je vhodné především pro široce používané aplikace (ftp, telnet, www ...). My si uvedeme dynamickou registraci, která je častější.

Inicializaci komunikačního protokolu provádíme voláním jedné ze dvou následujících funkcí.

```
void rpc_server_use_protseq(unsigned_char_t *protseq, unsigned32 max_call_req,
                            unsigned32 *status);
void rpc_server_use_all_protseqs(unsigned32 max_call_req, unsigned32 *status);
```

První funkce inicializuje jeden protokol uvedený v proměnné `protseq`. Druhá funkce inicializuje všechny dostupné protokoly. Proměnná `max_call_req` udává, kolik současných volání může server zpracovat.

Po dokončení inicializace protokolu získáme ovladač vazby voláním funkce

```
void rpc_server_inq_binding(rpc_binding_vector_t **binding_vector,
                            unsigned32 *status);
```

která vrací ukazatel na ovladač vazby. Registrace rozhraní potom může být:

```
if(argc > 1) {
   rpc_server_use_protseq((unsigned_char_t *)argv[1],
                          rpc_c_protseq_max_calls_default, &status);
   CHECK_STATUS(status, "Nemohu pouzit protokol:", ABORT);
}
else {
   rpc_server_use_all_protseqs( rpc_c_protseq_max_calls_default, &status);
   CHECK_STATUS(status, "Nemohu pouzit protokoly:", ABORT);
}
```

```
rpc_server_inq_bindings( &binding_vector, &status);
CHECK_STATUS(status, "Nemohu ziskat informace pro spojeni:", ABORT);
```

Studenty katedry počítačů, kde skriptum vzniklo, chceme upozornit, že operační systém VMS u nás standardně nepovoluje použití protokolu sítě DECNET `ncacn_dnet_nsp`, který funkce `rpc_server_use_all_protseqs()` zkouší inicializovat. Protože funkce vrátí chybu, musíme chybu ošetřit (v makru CHECK_STATUS stačí použít operaci RESUME), nebo použít funkci `rpc_server_use_protseq()` a vybrat protokoly UDP a TCP.

Jak jsme se zmínili, může server informace o spojení uložit do databáze CDS. Vložení informace do databáze CDS se provádí voláním funkce:

```
void rpc_ns_binding_export(unsigned32 e_name_syntax, unsigned_char_t *e_name,
            rpc_if_handle_t if_handle, rpc_binding_vector_t *binding_vec,
            uuid_vector_t *obj_uuid_vec, unsigned32 *status);
```

Jména položek vkládaných do databáze CDS mohou mít různou syntaxi. Syntaxe se uvádí v parametru `e_name_syntax`. V současnosti je v DCE podporována pouze syntaxe jmen `rpc_c_ns_syntax_default`. Jméno položky je uvedeno v parametru `e_name`. Parametr `if_handle` obsahuje ukazatel na rozhraní, které chceme exportovat. `binding_vec` je ukazatel na ovladač vazby získaný voláním `rpc_server_inq_binding()`. Používáme-li uuid, uvedeme ukazatel na tento objekt v parametru `obj_uuid_vec`, v opačném případě má proměnná hodnotu NULL. Zapsání binding informací do databáze CDS pod jméno */.:/sklad_cs.felk.cvut.cz* včetně podpůrných operací:

```
strcpy(entry_name, "/.:/sklad_");
gethostname(hostname, STRINGLEN);
strcat(entry_name, hostname);
rpc_ns_binding_export(rpc_c_ns_syntax_default, (unsigned_char_t *)entry_name,
                      sklad_v1_0_s_ifspec, binding_vector, NULL, &status);
CHECK_STATUS(status, "Nemohu zapsat do databaze jmen:", RESUME);
```

Uvědomte si, že na rozdíl od čtení z CDS, které je často povoleno i neautentifikovaným uživatelům, je zápis do CDS vyhrazen pouze objektům již přihlášeným do DCE. Před prvním spuštěním programu (do databáze CDS se informace, pokud nejsou smazány, zapisují pouze jednou) je potřeba se přihlásit do prostředí DCE příkazem

`dce_login` *jméno heslo*

Při volání RPC potřebuje klient znát koncový bod. Použijeme-li dynamické přidělení koncových bodů, není koncový bod uveden v databázi CDS. Server proto musí vložit informaci o koncových bodech do databáze koncových bodů (obdoba portmapperu v ONC RPC). Tato databáze je přítomna na všech počítačích, na kterých lze spustit server. Zápis do databáze zajišťuje funkce:

```
void rpc_ep_register(rpc_if_handle if_handle,
                     rpc_binding_vector_t *binding_vec,
                     uuid_vector_t *obj_uuid_vec,
                     unsigned_char_t *annotation,
                     unsigned32 *status);
```

V této funkci je pro nás jediný nový parametr `annotation`. Tento parametr je nepovinný a obsahuje popis koncového bodu, který zjednodušuje administrátorovi správu koncových bodů:

```
strcpy(annotation, "Rozhrani skladu");
```

```
rpc_ep_register(sklad_v1_0_s_ifspec, binding_vector, NULL,
                (unsigned_char_t *)annotation, &status);
CHECK_STATUS(status, "Nemohu pridat koncovy bod do lokalni databaze:",
             RESUME);

rpc_binding_vector_free(&binding_vector, &status);
CHECK_STATUS(status, "Nemohu uvolnit ukazatel na spojeni:", RESUME);

otevri_sklad();
```

V příkladu je znázorněno přidání koncového bodu do lokální databáze, uvolnění paměti alokované pro ovladač vazby a inicializace skladové databáze.

Inicializace serveru je zakončená uvedením serveru do režimu čekání na vzdálené volání. Režim čekání je spuštěn voláním funkce:

```
void rpc_server_listen(unsigned32 max_call_exec, unsigned32 status);
```

Při volání lze v parametru `max_call_exec` určit maximální počet současně zpracovaných příchozích volání. Server je v režimu čekání, dokud není přerušen jiným procesem, nebo klient nepoužije volání funkce `rpc_mgmt_stop_server_listening()`. V následujícím příkladu je ukázáno uvedení serveru do režimu čekání a některé procedury, které se provádějí při ukončení běhu serveru.

```
TRY
   rpc_server_listen(1, &status);
   CHECK_STATUS(status, "Nemohu prejit do stavu cekani:", RESUME);

FINALLY
   zavri_sklad();
   rpc_server_inq_bindings(&binding_vector, &status);
   CHECK_STATUS(status, "Nemohu ziskat binding informace:", RESUME);

   rpc_ep_unregister(
      sklad_v1_0_s_ifspec,
      binding_vector,
      NULL,
      &status
   );
   CHECK_STATUS(status, "Nelze odstranit koncovy bod z~databaze:", RESUME);

   rpc_binding_vector_free(&binding_vector, &status);
   CHECK_STATUS(status, "Nelze uvolnit pamet binding handle:", RESUME);

   puts("\nServer odstaven!");
ENDTRY
}
```

V této části programu jsou použita makra `TRY`, `FINALLY` a `ENDTRY`. Makra slouží k ošetření výjimek, které mohou nastat při běhu programu. Za makrem `TRY` uvádíme příkazy, při jejichž běhu může dojít k výjimce a ta má být ošetřena. Příkazy, které se provedou při výjimce i při korektním ukončení výpočtu, uvádíme mezi makra `FINALLY` a `ENDTRY`.

Sestavením uvedených úseků programu do jednoho souboru získáte kompletní ukázku fungující registrace serveru.

Na závěr uvedeme kompletní příklad druhé fáze implementace serveru: definice vzdáleně volaných funkcí na straně serveru. Z příkladu je patrné i použití proměnných typu string, polí s přizpůsobitelnou délkou a variantních záznamů.

```
#include <stdio.h>
#include "sklad.h"

idl_boolean polozka_je_dostupna(polozka_cis cislo)
{
   polozka_record *polozka;
   int found;

   found = cti_polozku(cislo, &polozka);
   if(found)
      return(idl_true);
   else
      return(idl_false);
}

veta popis_polozky(polozka_cis cislo)
{
   polozka_record *polozka;
   veta popis;
   int size;
   char *strcpy();

   if( cti_polozku(cislo, &polozka) ) {
      size = strlen((char *)polozka->popis) + 1;
      popis = (veta)rpc_ss_allocate((unsigned)size);
      strcpy((char *)popis, (char *)polozka->popis);
   }
   else
      popis = NULL;
   return(popis);
}

void jake_je_mnozstvi(polozka_cis cislo, polozka_mnozstvi *mnozstvi)
{
   polozka_record *polozka;

   cti_polozku(cislo, &polozka);
   mnozstvi->jednotky = polozka->mnozstvi.jednotky;
   switch(mnozstvi->jednotky) {
      case KUS: mnozstvi->celkem.pocet = polozka->mnozstvi.celkem.pocet;
                break;
      case KILOGRAM:
      case GRAM: mnozstvi->celkem.vaha = polozka->mnozstvi.celkem.vaha;
                break;
   }
   return;
}

void jake_jsou_podpolozky(polozka_cis cislo, polozka_list **podpolozka_ptr)
{
   polozka_record *polozka;
   int i;
   int size;

   cti_polozku(cislo, &polozka);

   size = sizeof(polozka_list) + (sizeof(polozka_cis) *
                                     (polozka->podpolozky.size-1));
```

```
    *podpolozka_ptr = (polozka_list *)rpc_ss_allocate((unsigned)size);

    (*podpolozka_ptr)->size = polozka->podpolozky.size;
    for(i = 0; i < (*podpolozka_ptr)->size; i++)
       (*podpolozka_ptr)->cisla[i] = polozka->podpolozky.cisla[i];
    return;
}
```

Překlad a sestavení programu používajícího DCE RPC je velmi podobné jako v případě ONC RPC. Protože jsme v době psaní tohoto textu měli DCE dostupný pouze na počítači DEC s operačním systémem VMS, uvedeme příklad překladu a spuštění aplikace s ohledem na tento operační systém. Překlad na tomto systému je automatizován použitím DCL (Digital Command Language) skriptu. Tento skript provede překlad idl souboru, zdrojových souborů klienta a serveru a dalších souborů vzniklých překladem idl souboru. Nakonec skript sestaví program klienta a serveru.

```
$@DCE:DCE$CC_TEST
$ cc := cc'f$trnlnm("dce$cc")'
$ IF( f$trnlnm( "dce$cc" ) .eqs. " " )
$ THEN options = "dce:dce.opt/options" + "," + "dce:dce_vaxc.opt/options"
$ ELSE options = "dce:dce.opt/options"
$ ENDIF
$ idl := $sys$system:dce$idl.exe
$ idl sklad.idl -keep all -trace all -trace log_manager
$ cc s_client
$ link/exec=client s_client,sklad_cstub,'options'
$ write sys$output "CLIENT.EXE done."
$ cc s_server,s_procedury,implementace_sklad
$ link/exec=server s_server,s_procedury,sklad_sstub,-
                implementace_sklad,'options'
$ write sys$output "SERVER.EXE done."
```

Ani spouštění programů na systému VMS není stejné jako v operačním systému UNIX. Nejjednodušším způsobem je vytvořit si skript, který nastaví proměnné systému.

```
$define RPC_DEFAULT_ENTRY "/.:/sklad_cs.felk.cvut.cz"
$server:==$user$device:[doktk336.kubr.dce.sklad]server.exe
$client:==$user$device:[doktk336.kubr.dce.sklad]client.exe
```

Skripty musí být uloženy v souboru s příponou *.com* (např. *sklad.com*). Skript se spustí příkazem `@sklad`. Program se spouští nadefinovaným jménem s parametry (např. `client`, `server`, nebo `server ncacn_ip_tcp`). Program se dá spustit i příkazem `run server.exe`, ale v tomto případě nejdou připojit argumenty.

V nové verzi operačního systému VMS (od verze 6.2) je možné nastavit systémovou proměnnou `dcl$path` pomocí příkazu `define dcl$path []`. Potom je možné spouštět programy přímo z příkazového řádku (např. `server ncacn_ip_tcp`). Pochopitelně nesmíme zapomenout nastavit systémovou proměnnou `RPC_DEFAULT_ENTRY`.

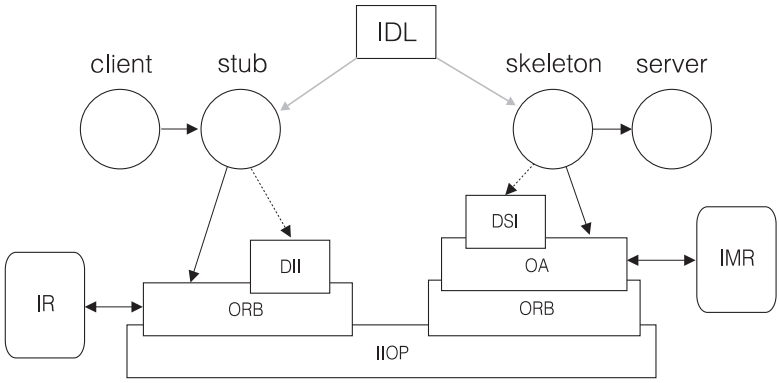
4.4 Technologie CORBA

Na závěr této kapitoly, věnované technologiím označovaným jako *middleware*, se zastavme u technologie, která vznikla s cílem vzájemně sjednotit přístup k rozhraní komponent v rozsáhlých systémech a dovolit spolupráci mezi komponentami vytvářenými v různých programovacích jazycích. Z iniciativy výrobců programových systémů vzniklo v roce 1989 konsorcium OMG (*Object Management Group*), které v průběhu času vytvořilo standard označovaný jako CORBA (*Common Object Request Brooker Architecture*) opírající se o objektový přístup k aplikačním komponentám. V současné verzi standardu CORBA 2.2 jsou podporovány programovací jazyky C, C++, Smalltalk, Cobol, Ada a Java.

Technologie CORBA v mnohém překračuje rozsah funkcí, které najdeme u RPC systémů, a zasloužila by si samostatnou kapitolu. Následující stručný popis slouží spíše jako příklad obecnějšího přístupu, než jaký reprezentují systémy ONC RPC a DCE. Zájemce o bližší informace odkazujeme na literaturu [15] a [16].

Představa, o níž se systém CORBA opírá, je následující: jednotlivé komponenty systému jsou instancemi objektů (označovaných jako *ORB objekty*), které zapouzdřují atributy/proměnné, metody a výjimky aplikačních komponent. Objektové instance mohou být umístěny na témže počítači, stejně jako mohou být umístěny na různých počítačích distribuovaného systému. Systém CORBA zakrývá technické detaily komunikace mezi aplikačními komponentami, které splňují určitá pravidla chování na vzájemném rozhraní.

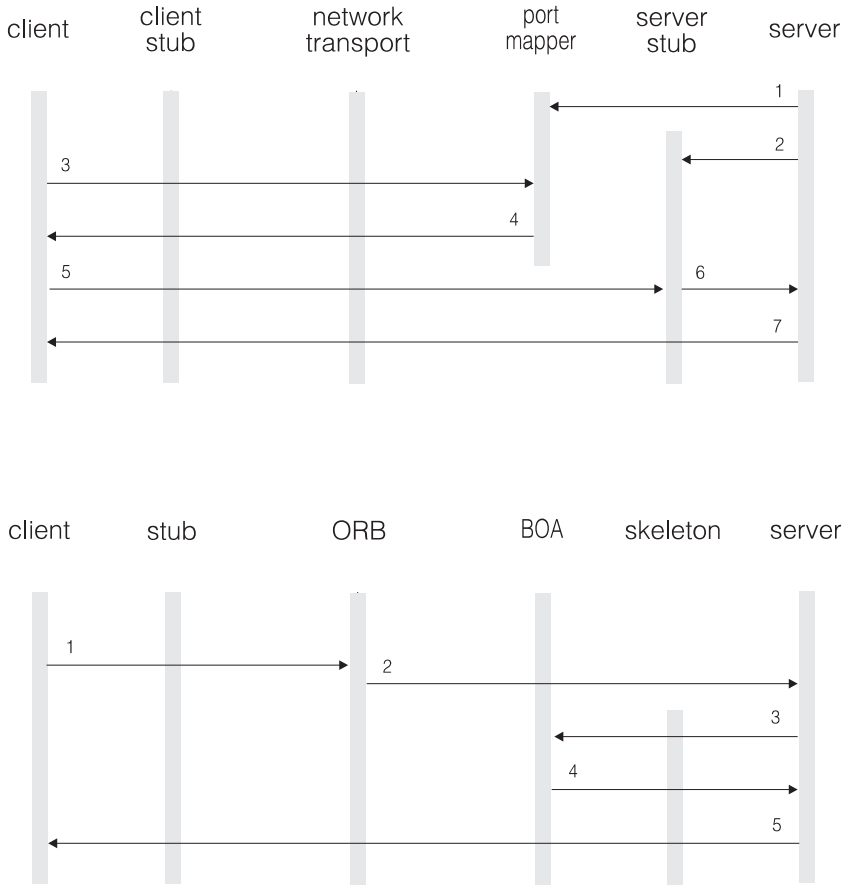
Pro popis rozhraní mezi objekty ORB používá CORBA jazyk IDL (*Interface Definition Language*). Tento jazyk dovoluje definovat konstruované typy využívané rozhraním a dostupné atributy, metody a výjimky. Překladem rozhraní v jazyce IDL získáváme *stub* klienta a *skeleton*, který po doplnění výkonného kódu v konkrétním programovacím jazyce implementuje aplikační komponentu. Na rozdíl od technologií ONC RPC a DCE systém CORBA nerozlišuje mezi objekty realizujícími role klienta a serveru; implementace systémů, ve kterých objektové komponenty realizují obě role (klient i server) nepřináší žádné přídavné problémy.



Obr. 4.7: Struktura systému CORBA

Systém CORBA podporuje všechny funkce potřebné pro spolupráci objektových komponent v distribuovaném prostředí (obr. 4.7). Volání metody objektové komponenty je *stubem* převedeno na zprávu, kterou předá *transportní systém ORB* na počítač, kde je tato komponenta dostupná. Volání je předáno *objektovému adaptéru* (OA - Object Adapter), který případně zavede, zaregistruje a inicializuje *implementaci komponenty* a předá jí volání prostřednictvím kódu realizujícího *skeleton*. Výsledek výpočtu komponenty (výsledek, výstupní parametry, výjimky) jsou předány prostřednictvím *skeletonu*, *transportního systému ORB* a *stubu* zpět klientovi. Objektový adaptér OA tedy plní i funkce, které má v systému ONC RPC *portmapper* a v systému DCE inicializační kód serveru (obr. 4.8). Současné implementace používají

objektové adaptéry, které realizují pouze základní funkce - BOA (*Basic Object Adapter*). Trendem je přechod k přesněji specifikovanému a portabilnímu objektovému adaptéru POA (*Portable Object Adapter*), který dovoluje vázat objektový adaptér na výkonné komponenty (*servant*) s odlišným cyklem "života" (persistentní a transientní komponenty) a zajišťuje paralelismus opírající se o vlákna výpočtu.



Obr. 4.8: Srovnání volání v systému ONC RPC a v systému CORBA

Předkompilovaný stub je ukládán do knihovny rozhraní IR (*Interface Repository* - obr. 4.7), dovolující zavést stub dynamicky v okamžiku volání metody objektu. Protějškem knihovny rozhraní je knihovna implementací IMR (*Implementation Repository*), kterou používá objektový adaptér, když přebírá volání metody.

Kromě základního statického vyvolání metody, které se opírá o předkompilovaný stub, podporuje CORBA dynamické vytváření volání metod DII (*Dynamic Invocation Interface*) pro objekty, jejichž stub pak nemusí (nebo nemůže) být předem předkompilován a uložen do knihovny rozhraní IR.

Dynamické chování trochu jiného typu může být realizováno i na straně implementace, kde je označováno jako dynamický skeleton DSI (*Dynamic Skeleton Inteface*). Volání klienta je po předání objektovým adaptérem zprostředkováno ještě dynamickou implementační rutinou DIR (*Dynamic Implementation Routine*) a dovoluje tak využít různých programovacích jazyků pro zápis komponenty.

Jako příklad použití systému CORBA si můžeme uvést definici rozhraní bankovního systému a soubory vygenerované IDL překladačem pro jazyk C a pro jazyk Java:

```
interface BankAccount {
   boolean deposit(in unsigned long amount);
   boolean withdraw(in unsigned long amount);
   void balance(out long amount);
};
```

Naše ukázkové rozhraní bankovního účtu *BankAccount* dovoluje na účet ukládat a z účtu vybírat (metody *deposit* a *withdraw*), obě operace dovolují i záporný zůstatek na účtu (kredit), znaménko stavu účtu po provedení operace indikuje vracená logická hodnota (např. hodnota *true* reprezentuje kladný zůstatek). Metoda *balance* vrací stav účtu jako svůj výstupní parametr.

Překlad rozhraní *BankAccount* pro jazyk C může mít formu (uloženou např. jako soubor *bankacount.h*):

```
typedef CORBA_Object BankAccount;
extern CORBA_boolean BankAccount_deposit(BankAccount o,CORBA_unsigned_long amount,
      CORBA_Environment *ev);
extern CORBA_boolean BankAccount_withdraw(BankAccount o,CORBA_unsigned_long amount,
      CORBA_Environment *ev);
extern void BankAccount_balance(BankAccount o,CORBA_long *amount,CORBA_Environment *ev);
```

Součástí knihoven podporujících konkrétní implementaci musí být definice základních typů (CORBA_boolean, CORBA_long, CORBA_unsigned_long, ...). Metody objektů systému CORBA jsou implementovány jako funkce jazyka C, první parametr určuje objekt, jehož se vyvolání metody týká, poslední parametr dovoluje implementovat výjimky. Výstupní (*out*) a vstupně-výstupní (*inout*) parametry jsou volány odkazem, vstupní parametry (*in*) jsou předávány hodnotou tam, kde to jazyk C dovoluje.

Překlad našeho rozhraní *BankAccount* pro programovací jazyk Java může mít následující formu:

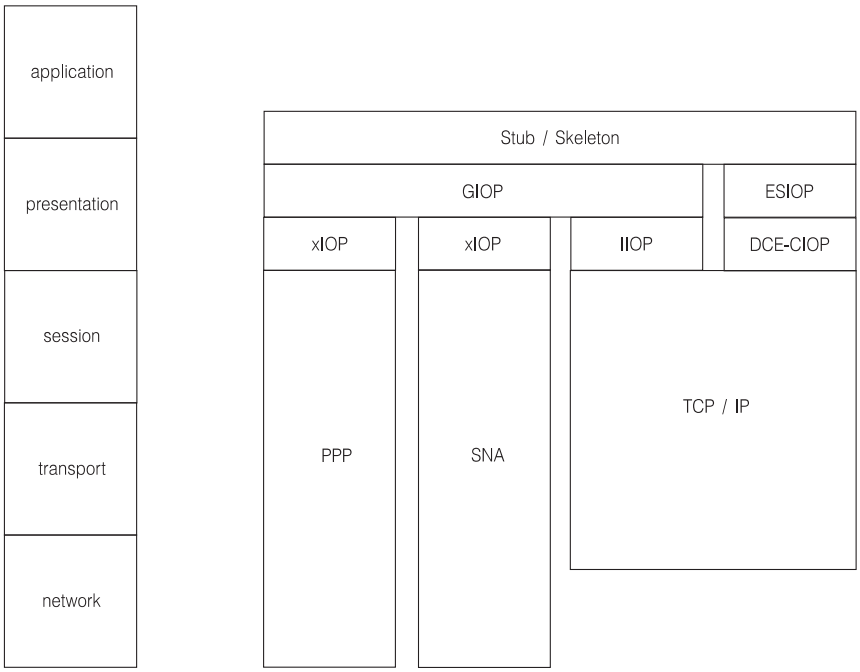
```
package Example;
public interface BankAccount {
   boolean deposit(int amount);
   boolean withdraw(int amount);
   void balance(IntHolder amount);
};
```

Jazyk Java podporuje objekty a využívá rozhraní; překlad je proto poměrně přímočarý. Malý problém představují pouze výstupní a vstupně-výstupní parametry, vracení hodnot je podporováno objekty typu *<type>Holder*. Definice pro základní typy jazyka IDL jsou součástí knihovny *org.omg.CORBA*, pro uživatelem definované typy je překladač IDL generuje. Pro náš příklad má definice knihovní třídy *IntHolder* tvar:

```
final public class IntHolder {
   public Int value;
   public IntHolder() {value = 0;}
   public IntHolder(Int initial) {value = initial;}
};
```

Spolupráce aplikačních komponent běžících na různých počítačích propojených komunikačním podsystémem vyžaduje konsistentní definici formátů předávaných volání a vracených výsledků a výjimek. Takovou definicí se pro prostředí TCP stala definice protokolu IIOP (*Internet Inter-ORB Protocol*). Pro prostředí jiných transportních a relačních protokolů může být definice IOB (*Inter-ORB Protocol*) protokolů odlišná. Překrytí vzájemných rozdílů zajišťuje protokol GIOB (*General Inter-ORB Protocol*), který se opírá o standardní reprezentaci dat CDR (*Common Data Representation*) a definuje formáty předávání zpráv. Z reprezentace CDR a formátů zpráv GIOB pak protokol IIOB (stejně jako protokoly pro jiná transportní prostředí, jako jsou spojení PPP nebo sítě SNA) vychází.

Alternativní možností vazby aplikační komponenty na komunikační systém může být specifický IOB protokol ESIOP (*Environment Specific Inter-ORB Protocol*) postavený nad protokolem jiného middleware systému. Takovým příkladem je DCE-CIOP (*DCE Common Interoperability Protocol*). Architekturu komunikačních protokolů CORBA a jejich místo v sedmivrstvové architektuře ISO-OSI ilustruje obr. 4.9.



Obr. 4.9: Architektura komunikačních protokolů CORBA

Komunikační architektura podporující objektové prostředí ORB se stává zřejmým standardem. Svědčí o tom jednak existence protokolů jako je DCE-CIOP dovolujících spolupráci s průmyslovým standardem OSF DCE, jednak i začlenění protokolu IIOP jako (zatím) alternativního komunikačního prostředí a překladače jazyka IDL systému CORBA do jazykového systému Java.

Systém CORBA v současnosti prodělává prudký rozvoj. Vedle základních funkcí podporujících spolupráci objektových komponent (označujeme je jako *CORBAservices*) zahrnuje i podporu složitějších systémových služeb (označujeme je jako *Horizontal CORBA-facilities* a podporují uživatelské rozhraní, přístup k datovým zdrojům a správu systému). Jsou vytvářeny standardní aplikace pro určité oblasti (označujeme je jako *Vertical CORBAfacilities*). Vedle statického pojetí výpočtu jsou v současnosti vytvářeny základy pro práci s mobilními komponentami. Takový přístup dovolí v řadě případů snížit zatížení transportní sítě tím, že namísto přenosu množství dat ke komponentám přeneseme výpočet komponent k datovým zdrojům. Současná specifikace systému dovoluje definovat objekty, které lze následně přenášet technologií (*Value Objects*).

5. Komunikační prostředky jazyka Java

Programovací jazyk Java (hezkým úvodem do jazyka Java je [17], a v českém překladu [18]) je jeho autory charakterizován jako jednoduchý objektově orientovaný jazyk, podporující výpočetní paralelismus a distribuovaný výpočet, navržený s ohledem na efektivní interpretaci na počítačích s odlišnou architekturou procesoru a podporující tak přenesitelnost vytvářených programů. Cílem jeho tvůrců bylo podpořit dynamičnost výpočtu (včetně zavádění modulů programu na žádost i ze vzdálených prvků systému), zajistit spolehlivost výsledných programů a celkovou bezpečnost systémů v jazyce Java vytvářených.

Řada z uvedených rysů zvýhodňuje tento jazyk proti klasickým programovacím jazykům (např. C, C++). Jeho hlavním problémem je hlavně nižší efektivita, která souvisí s interpretací mezijazyka (bytecode) ve většině současných implementací. V programování distribuovaných aplikací však přináší jazyk Java řadu výhod, za zmínku stojí zvláště podpora výpočetního paralelismu a podpora pro počítačové komunikace. Rozšíření jazyka přispěla jeho otevřenost a volná dostupnost od jeho prvních implementací. V současnosti je kolem jazyka Java vybudován systém podpory pro řadu moderních aplikačních oblastí (např. počítačová telefonie, čipové karty, elektronický obchod).

Jazyk Java dovoluje psát programy, ve kterých spolupracuje více *vláken výpočtu* (multithreading). Přepínání mezi vlákny (*lightweight processes*) je efektivnější než přepínání mezi běžnými procesy, u kterých je potřebné přepínat i příslušný kontext. Zahrnutí vláken mezi základní prvky jazyka (třída *Thread* knihovny *java.lang*) poskytuje slušnou vývojovou podporu a ochranu systému při havárii aplikace.

Synchronizace mezi vlákny výpočtu v Javě se opírá o synchronizační prostředky vyšší úrovně, než jsou zámky a semafory dostupné v knihovnách operačního systému. Specifikace objektu nebo jeho jednotlivých metod jako synchronizovaných (*synchronized*) dovoluje omezit vstup do objektu nebo jeho určených metod na jediné vlákno. Použitý mechanismus vychází z Hoarových monitorů a dovoluje přehlednější omezení přístupu výpočetních vláken ke sdíleným datům/prostředkům.

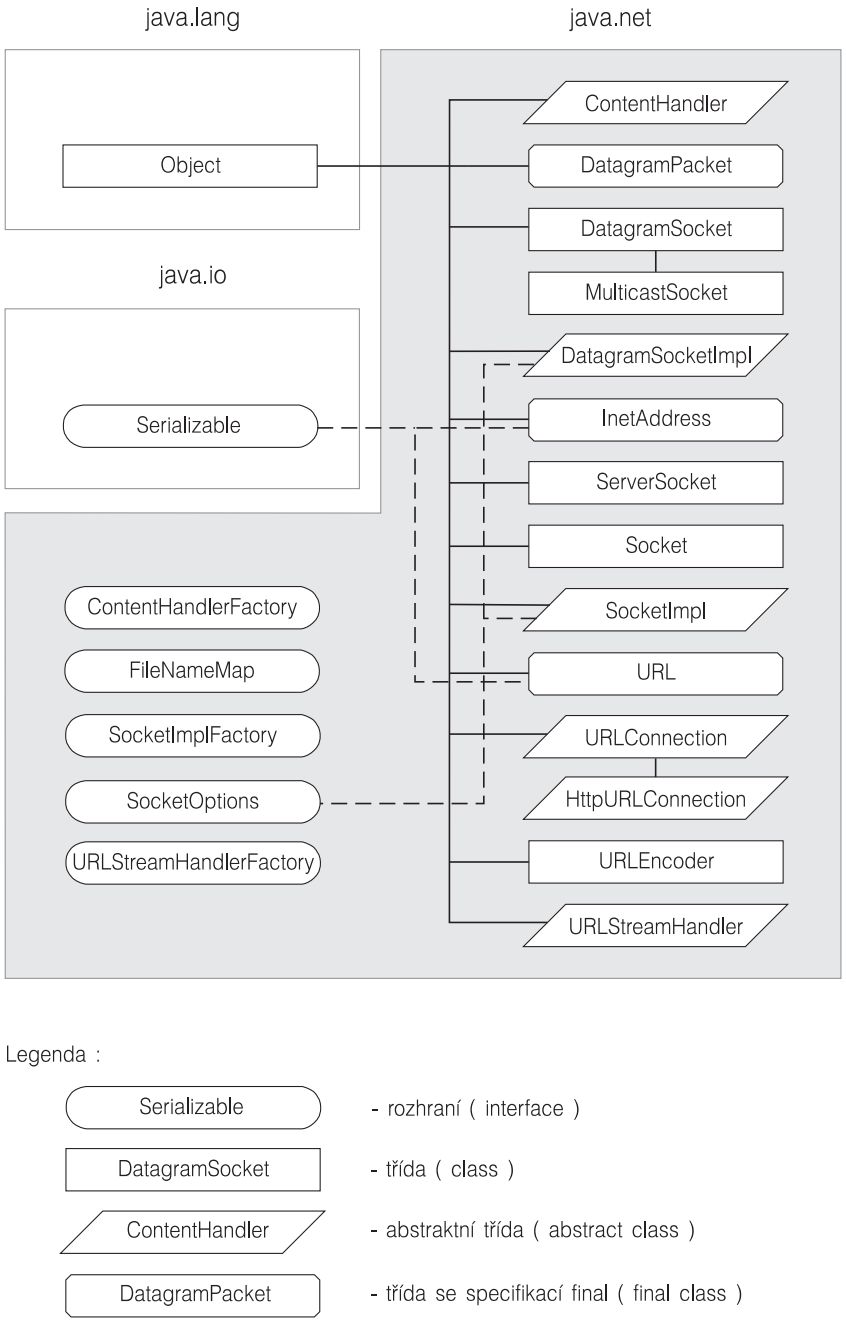
Silnou pozici v oblasti programovacích prostředků pro distribuované a síťové aplikace zajistilo jazyku Java jeho vybavení komunikačními rozhraními, která dovolují využívat běžné typy internetovských transportních kanálů TCP a UDP formou, která je nesrovnatelně jednodušší než jsme poznali u socketových knihoven (BSD, TLI, WinSocks). Doplnění základních funkcí o podporu přenosu souborů identifikovaných odkazy URL (URL - Universal Resource Location) dovolilo začlenit aplikace zapsané v jazyce Java do systému WWW (vložené odkazy na jejich kód do stránek HTML dovolují prohlížečům WWW zavedení a spuštění portabilního kódu) a výrazně rozšířit možnosti, které systém WWW poskytoval.

Základní komunikační prostředky byly později doplněny systémem procedurální komunikace, který rozšiřuje použití v základním jazyce definovaného rozhraní objektu na rozhraní *vzdáleného volání metod* (RMI - Remote Method Invocation).

Objekty, vzdáleně zaváděné prohlížečem WWW a spouštěné pod vestavěným interpretem portabilního kódu (JVM - Java Virtual Machine), známé pod termínem *applety* se staly standardem dovolujícím vytvářet efektivní uživatelské rozhraní WWW aplikací. Výhodnost dynamického vkládání appletů na stranu prohlížeče vedla později k rozšíření aplikačních objektů v jazyce Java i na stranu serverů WWW, zde tyto objekty označujeme jako *servlety*. Možnost dynamického zavádění objektů se nemusí zřejmě omezit na prohlížeče, dynamické zavádění objektů a spouštění jejich kódu na výpočtem zvolených serverech dovoluje zavést aktivní mobilitu do distribuovaného výpočtu realizovanou objekty označovanými jako *mobilní agenti*. Současné systémy podporující takovou formu výpočtu (z nejznámějších můžeme uvést Voyager

a Aglets), již ale leží mimo rozsah našeho textu.

Tato kapitola je věnována popisu rozhraní a objektů podporujících socketovou komunikaci a přístup k souborům prostřednictvím odkazů URL. Jejich definice jsou obsahem knihovny *java.net*, strukturu knihovny uvádí obr. 5.1. Popisu jednotlivých tříd této knihovny a technologii *vzdáleného volání metod* (RMI - Remote Method Invocation) se budeme věnovat podrobněji v jednotlivých částech kapitoly.



Obr. 5.1: Třída *java.net*

Popis tříd knihovny *java.net* a systému RMI je doprovázen ilustračními příklady. Všechny byly ověřeny v prostředí JDK 1.1.4 pod operačními systémy Solaris a Linux.

5.1 Adresace

Jako první si popíšeme třídu *InetAddress*, která podporuje IP adresaci. Její metody dovolují vytvářet objekty reprezentující IP adresy a realizovat běžné operace nad nimi. Veřejně dostupná část signatury třídy *InetAddress* má tvar:

```
public final class InetAddress extends Object implements Serializable {
   public static InetAddress getLocalHost() throws UnknownHostException
   public static InetAddress getByName(String host) throws UnknownHostException
   public static InetAddress[] getAllByName(String host)
               throws UnknownHostException
   public byte[] getAddress()
   public String getHostAddress()
   public String getHostName()
   public boolean equals(Object obj)
   public boolean isMulticastAddress()
   public int hashCode()
   public String toString()
}
```

Třída *InetAddress* nemá uživatelsky přístupný konstruktor, její instance získáváme voláním statických metod (jsou vázány na třídu, nikoliv na konkrétní instanci) *getLocalHost*, *getByName* a *getAllByName*. První z nich, *getLocalHost*, vrací jako výsledek objekt odpovídající IP adrese počítače, na němž aplikace běží. Statická metoda *getByName* vrací objekt reprezentující IP adresu počítače zadaného doménovým jménem a metoda *getAllByName* vrací všechny IP adresy. Tyto metody se opírají o běžné postupy dovolující zjistit vazbu mezi IP adresou a doménovým jménem (soubor /etc/hosts, servery DNS).

Metoda běžné instance *getAddress* vrací IP adresu jako čtyřbytové pole (v konvenci "big endian"), metoda *getHostAddress* jako řetěz znaků (např. "192.160.108.1") a konečně metoda getHostName vrací doménové jméno. Metoda *isMulticastAddress* dovoluje otestovat, zda IP adresa odpovídá skupinové komunikaci (adresa třídy D, hodnoty v intervalu 224.0.0.0 až 239.255.255.255). Konečně, metoda *equals* dovoluje zjistit zda dva objekty *InetAddress* reprezentují tutéž IP adresu, metody *hashCode* a *toString* doplňují informace poskytované odpovídajícími metodami třídy Object.

5.2 Datagramová komunikace

Nejjednodušší formou IP komunikace je nepotvrzovaný přenos uživatelem definovaných datagramů. Nemáme zaručeno doručení a ani pořadí přijatých paketů nemusí odpovídat pořadí paketů odeslaných (pakety nebo jejich fragmenty procházející sítí jsou směrovány nezávisle). Takový přenos známe pod označením *UDP - User Datatagram Protocol*. V jazyce Java jsou základní funkce datagramové komunikace podporovány třídami *java.net.DatagramPacket* a *java.net.DatagramSocket*. Skupinovou komunikaci podporuje třída *java.net.MulticastSocket*. Konečně třída *java.net.DatagramSocketImpl* dovoluje modifikovat metody UDP komunikace v situacích, kdy nám standardní chování nevyhovuje.

5.2.1 Třída DatagramPacket

Třída *DatagramPacket* dovoluje připravovat datagramy (UDP pakety) k odeslání na zadanou adresu, požádat o jejich přenos, přijmout je a po příjmu analyzovat. Signatura třídy má tvar

```
public final class DatagramPacket extends Object {
   public DatagramPacket(byte[] ibuf,int ilength)
   public DatagramPacket(byte[] ibuf,int ilength,InetAddress iaddr,int iport)
   public synchronized InetAddress getAddress()
   public synchronized int getPort()
   public synchronized byte[] getData()
   public synchronized int getLength()
   public synchronized void setAddress(InetAddress iaddr)
   public synchronized void setPort(int iport)
   public synchronized void setData(byte ibuf[])
   public synchronized void setLength(int ilength)
}
```

Konstruktor *DatagramPacket(byte[] ibuf,int ilength)* podporuje příjem paketu, parametry uvádějí adresu a délku paměti pro data. Konstruktor *DatagramPacket(byte[] ibuf,int ilength,InetAddress iaddr,int iport)* připravuje paket k vyslání, parametry určují vedle adresy a délky bloku přenášených dat ještě adresu a číslo portu příjemce.

Metody *getAddress*, *getPort*, *getData* a *getLength* podporují analýzu paketu. Metody *setAddress*, *setPort*, *setData* a *setLength* dovolují jednotlivé parametry paketu nezávisle modifikovat. Za zmínku stojí klausule *synchronized* v hlavičce metod, ta vyjadřuje skutečnost, že metody pro analýzu a modifikaci UDP paketu zahrnují ochranu proti násobnému přístupu.

5.2.2 Třída DatagramSocket

Třída *DatagramSocket* připravuje porty UDP k odesílání a příjmu datagramů vytvářením příslušných řídících struktur - *socketů* a podporuje vlastní UDP komunikaci. Její veřejně dostupná signatura má tvar:

```
public class DatagramSocket extends Object {
   public DatagramSocket() throws SocketException
   public DatagramSocket(int port) throws SocketException
   public DatagramSocket(int port,InetAddress laddr) throws SocketException
```

```
    public void send(DatagramPacket p) throws IOException
    public synchronized void receive(DatagramPacket p) throws IOException
    public void close()
    public synchronized void setSoTimeout(int timeout) throws SocketException
    public synchronized int getSoTimeout() throws SocketException
    public InetAddress getLocalAddress()
    public int getLocalPort()
}
```

Konstruktor *DatagramSocket(int port)* otevírá pro komunikaci zvolený UDP port, konstruktor *DatagramSocket()* nechává volbu portu UDP na systému. Poslední konstruktor *DatagramSocket(int port,InetAddress laddr)* dovoluje otevřít UDP komunikaci na zvoleném portu a zvolené lokální adrese (to má význam u počítačů připojených do sítě více komunikačními kanály). Další konstruktory použitelné pouze v odvozovaných podtřídách (jsou specifikovány jako *protected*) přidávají parametr dovolující určit zvláštní modifikaci chování UDP socketu (parametr odkazuje na objekt třídy DatagramSocketImpl). Metody *send* a *receive* podporují odeslání a příjem UDP paketu, metoda *close* port UDP uzavírá. Metody *getSoTimeout* a *setTimeout* dovolují zjistit a nastavit časový limit pro příjem UDP paketu v milisekundách, nulová hodnota odpovídá neomezovanému čekání na příjem. Pomocné metody *getLocalAddress* a *getLocalPort* dovolují zjistit lokální adresu (prostředky pro práci se vzdálenou adresou poskytuje třída *Datagram*).

5.2.3 Třída MulticastSocket

Třída *MulticastSocket* rozšiřuje definici třídy *DatagramSocket* o metody podporující skupinovou komunikaci (realizovanou protokolem IGMP).

```
public class MulticastSocket extends DatagramSocket {
    public MulticastSocket() throws IOException
    public MulticastSocket(int port) throws IOException
    public void joinGroup(InetAddress mcastaddr) throws IOException
    public void leaveGroup(InetAddress mcastaddr) throws IOException
    public void setTTL(byte ttl) throws IOException
    public byte getTTL() throws IOException
    public InetAddress getInterface() throws SocketException
    public void setInterface(InetAddress inf) throws SocketException
    public synchronized void send(DatagramPacket p,byte ttl) throws IOException
}
```

Konstruktory *Multicast(int port)* a *Multicast()* otevírají pro skupinovou komunikaci zvolený nebo systémem přidělený port.

Metody *joinGroup* a *leaveGroup* slouží pro registraci skupinové adresy na směrovačích, metody *getInterface* a *setInterface* zjišťují a volí IP adresu využívanou pro skupinovou komunikaci na daném UDP portu. Metody *getTTL* a *setTTL* dovolují zjistit a nastavit implicitní hodnotu pole TTL v odesílaných paketech (doba "života" UDP paketu v počtu směrovačů/sekund, lze tak omezit dosah skupinové komunikace). Metoda *send* odesílá UDP paket se zadanou hodnotou TTL, odeslání UDP paketu s implicitní hodnotou TTL zajišťuje metoda *send* třídy *DatagramSocket*.

5.2.4 Třída DatagramSocketImpl

Abstraktní třída *DatagramSocketImpl* zahrnuje implementační metody, na kterých staví třídy *DatagramSocket* a *MulticastSocket*. Její zpřístupnění dovoluje modifikovat základní funkce UDP komunikace, dovolí nám například popsat UDP komunikaci procházející specifickou bezpečnostní bránou (firewall), implicitní šifrování nebo komprimaci, komunikaci nad odlišným síťovým protokolem a podobně.

```
public abstract class DatagramSocketImpl extends Object implements SocketOptions {
   protected int localPort
   protected FileDescriptor fd
   public DatagramSocketImpl()
   protected abstract void create() throws SocketException
   protected abstract void bind(int lport,InetAddress laddr)
               throws SocketException
   protected abstract void send(DatagramPacket p) throws IOException
   protected abstract int peek(InetAddress i) throws IOException
   protected abstract void receive(DatagramPacket p) throws IOException
   protected abstract void setTTL(byte ttl) throws IOException
   protected abstract byte getTTL() throws IOException
   protected abstract void join(InetAddress inetaddr) throws IOException
   protected abstract void leave(InetAddress inetaddr) throws IOException
   protected abstract void close()
   protected int getLocalPort()
   protected FileDescriptor getFileDescriptor()
}
```

Vlastní modifikací třídy *DatagramSocketImpl* lze nahradit implicitní implementaci UDP komunikace *PlainSocketImpl*. Je také možné vytvořit implementaci, která bude sloužit pouze určitým instancím třídy *DatagramSocket*, pro takový případ je třída *DatagramSocket* doplněna o konstruktory využitelné pouze v od ní odvozených podtřídách (*protected* konstruktory mají jako parametr instanci třídy *DatagramSocketImpl*).

Zpřístupnění třídy *DatagramSocketImpl* dává možnost modifikovat funkce *pod* standardním UDP rozhraním, odvozování podtříd třídy *DatagramSocket* dovoluje doplňovat funkce *nad* standardním UDP rozhraním.

5.2.5 Příklad - UDP Echo Server

Jako příklad datagramové komunikace si uvedeme jednoduchý *Echo Server*, schopný informovat o přijatých UDP paketech a vracet je klientům:

```
import java.io.*;
import java.net.*;
public class UDPServer {
   static final int port = 6010;
   public static void printInfo(InetAddress a,int p,String s) {
      System.out.println("Client IP address : "+a.getHostAddress()+":"+p+":");
      System.out.println("Client domain address : "+a.getHostName()+":"+p+":");
      System.out.println("Message echoed : "+"["+s.length()+"] "+s);
```

```
    }
    public static void main(String args[]) throws Exception {
       byte[] buffer = new byte[1024];
       if (args.length!=0) {
          System.out.println("Usage: java UDPServer"); System.exit(1);
       }
       DatagramPacket request = new DatagramPacket(buffer,buffer.length);
       DatagramSocket socket = null;
       try {
          socket = new DatagramSocket(port);
       }
       catch (SocketException e) {
          System.out.println(e); System.exit(2);
       }
       System.out.println("Waiting for request ... ");
       try {
          do {
             request.setData(buffer); request.setLength(buffer.length);
             socket.receive(request);
             String s = new String(request.getData(),0,request.getLength());
             printInfo(request.getAddress(),request.getPort(),s);
             DatagramPacket reply =
                new DatagramPacket(buffer,request.getLength(),
                                   request.getAddress(),request.getPort());
             socket.send(reply);
          } while (true);
       }
       catch (IOException e) {
          System.out.println(e);
       }
       finally {
          socket.close();
       }
    }
}
```

Server si vytváří socket vázaný na port 6010 a očekává příjem UDP paketu, přijatá data ukládá do pole *buffer*. Voláním metody *printInfo* informuje o přijetí UDP paketu na systémovém výstupu *System.out* a vrací obsah UDP paketu klientovi. Pracuje v nekonečném cyklu, který ukončí až případná výjimka.

Odpovídající klient, schopný odeslat paket se zadaným textem může být popsán následujícím programem:

```
import java.net.*;
import java.io.*;
public class UDPClient {
   static final int port = 6010;
   public static void printInfo(InetAddress a,int p,String s) {
      System.out.println("Destination domain address : "+a.getHostName()+":"+p);
      System.out.println("Destination IP address : "+a.getHostAddress()+":"+p);
```

```
        System.out.println("Message sent : "+"["+s.length()+"] "+s);
    }
    public static void main(String args[]) throws Exception {
        if (args.length!=2) {
            System.out.println("Usage: java UDPClient <hostname> <message>");
            System.exit(1);
        }
        InetAddress address = null;
        try {
            address = InetAddress.getByName(args[0]);
        }
        catch (UnknownHostException e) {
            System.out.println(e); System.exit(2);
        }
        printInfo(address,port,args[1]);
        DatagramSocket socket = null;
        try {
            socket = new DatagramSocket();
        }
        catch (SocketException e) {
            System.out.println(e); System.exit(3);
        }
        try {
            byte[] message = args[1].getBytes();
            int msglen = args[1].length();
            DatagramPacket request = new DatagramPacket(message,msglen,address,port);
            socket.send(request);
            byte[] buffer = new byte[1024];
            DatagramPacket reply = new DatagramPacket(buffer,buffer.length);
            socket.receive(reply);
            String s = new String(buffer,0,reply.getLength());
            System.out.println("Message returned : "+"["+s.length()+"] "+s);
        }
        catch (IOException e) {
            System.out.println(e);
        }
        finally {
            socket.close();
        }
    }
}
```

Klient zasílá UDP paket serveru určenému doménovým jménem, informuje o odeslání na systémovém výstupu *System.out* a čeká na odpověď. O příjmu odpovědi informuje opět na systémovém výstupu *System.out* a končí výpočet. Za povšimnutí stojí i použití metod *getHostAddress*, *getHostName* a *getByName* při práci s doménovými jmény a číselnými IP adresami.

5.3 Virtuální kanály TCP

Knihovna *java.net* podporuje vedle rozhraní nezabezpečené datagramové služby UDP i rozhraní virtuálních spojení TCP. To nám dává možnost využít vnitřních potvrzovacích mechanismů TCP, které zajišťují bezpečný přenos dat IP kanálem (a zbavit se tak spousty starostí) a navázat síťovou komunikaci na mechanismus proudů jazyka Java. TCP komunikace je podporována třídami *java.net.ServerSocket* a *java.net.Socket*. Podobně jako u datagramové služby existuje i u virtuálních spojení možnost předefinovat vnitřní chování pomocí třídy *java.net.SocketImpl*.

5.3.1 Třída ServerSocket

Třída *ServerSocket* implementuje funkce koncového místa TCP komunikace - *socketu* na straně serveru spojené s navazováním spojení. Funkce spojené s vlastním přenosem dat jsou implementovány samostatnou třídou *Socket*. Implementace TCP rozhraní v knihovně *java.net* tak na rozdíl od BSD socketů rozlišuje mezi navazováním TCP spojení a vlastním přenosem dat.

```
public class ServerSocket extends Object {
   public ServerSocket(int port) throws IOException
   public ServerSocket(int port,int backlog) throws IOException
   public ServerSocket(int port,int backlog,InetAddress bindAddr)
               throws IOException
   public InetAddress getInetAddress()
   public int getLocalPort()
   public Socket accept() throws IOException
   protected final void implAccept(Socket s) throws IOException
   public void close() throws IOException
   public synchronized void setSoTimeout(int timeout) throws SocketException
   public synchronized int getSoTimeout() throws IOException
   public static synchronized void setSocketFactory(SocketImplFactory fac)
               throws IOException
   public String toString()
}
```

Konstruktor *ServerSocket(int port)* vytváří socket a váže ho na zadaný TCP port, při nulové hodnotě parametru *port* je volba portu přenechána systému. Konstruktor *ServerSocket (int port,int backlog)* dovoluje v parametru *backlog* specifikovat maximální délku fronty pro příchozích volání, implicitní hodnota (při neuvedení parametru) je 50 příchozích volání. Poslední konstruktor *ServerSocket (int port,int backlog,InetAddress bindAddr)* dovoluje omezit příjem volání na určenou lokální adresu, hodnota *null* posledního parametru nebo jeho neuvedení (u předchozích konstruktorů) dovolí příjem volání na libovolné lokální adrese vícenásobně do sítě připojeného (*multi-homed*) systému.

Metoda *getInetAddress* vrací lokální adresu, na níž je vázán zadaný socket, metoda *getLocalPort* vrací číslo lokálního portu. Metoda *accept* nám dovoluje čekat na přicházející volání a vrací instanci (dále popsané) třídy *Socket* pro obsluhu vlastního spojení. Metoda *implAccept (Socket s)* dovoluje vytvářet vlastní modifikace navazování spojení, jako příklad si můžeme uvést zahrnutí autentifikačního dialogu do navazování spojení:

```
class SSLServerSocket extends ServerSocket {
   ...
   public Socket accept() throws IOException {
      SSLSocket s = new SSLSocket(certChain,privateKey);
      implAccept(s);
      s.handshake();
      return s;
   }
   ...
}

class SSLSocket extends Socket {
   ...
   public SSLSocket(CertChain c,PrivateKey k) {
      super();
      ...
   }
   ...
}
```

Na rozdíl od metody *accept*, která si socket pro přenos dat sama vytváří, musíme pro metodu *implAccept* potřebný socket (zde třídy *SSLSocket*) předem připravit.

Metoda *close* uzavírá TCP spojení, metody *setSoTimeout* a *getSoTimeout* dovolují nastavit a zjistit hodnotu časového limit pro převzetí volání (metodou *accept*). Metoda *setSocketFactory* umožňuje volit činnost při vytváření socketů: je vyvolána metoda createSocketImpl objektu *fac*. Metoda *toString* předefinovává odpovídající metodu třídy *Object*.

5.3.2 Třída Socket

Třída *Socket* implementuje funkce vlastního přenosu dat, přičemž poskytuje možnost vlastní implementace vnitřních funkcí určením vhodného objektu *SocketImpl*.

```
public class Socket extends Object {
   protected Socket()
   protected Socket(SocketImpl impl) throws SocketException
   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
   public Socket(InetAddress address,int port,InetAddress localAddr,int localPort)
               throws IOException
   public Socket(String host,int port,boolean stream) throws IOException
   public Socket(InetAddress host,int port,boolean stream) throws IOException
   public InetAddress getInetAddress()
   public InetAddress getLocalAddress()
   public int getPort()
   public int getLocalPort()
   public InputStream getInputStream() throws IOException
   public OutputStream getOutputStream() throws IOException
   public void setTcpNoDelay(boolean on) throws SocketException
```

```
    public boolean getTcpNoDelay() throws SocketException
    public void setSoLinger(boolean on,int val) throws SocketException
    public int getSoLinger() throws SocketException
    public synchronized void setSoTimeout(int timeout) throws SocketException
    public synchronized int getSoTimeout() throws SocketException
    public synchronized void close() throws IOException
    public String toString()
    public static synchronized void setSocketImplFactory(SocketImplFactory fac)
                throws IOException
}
```

Třída má řadu konstruktorů, první z nich *Socket()* vytváří volný socket s implicitní funkcí, zatímco konstruktor *Socket(SocketImpl impl)* dovoluje svázat socket se specifickou implementací vnitřních funkcí. Konstruktory *Socket(String host,int port)* a *Socket(InetAddress host,int port)* vytvářejí socket klienta a otevírají spojení na určený port určeného serveru. Liší se v adresaci serveru (DNS jméno nebo IP adresa) a využívají implicitní funkci socketu. Konstruktory *Socket(String host,int port,InetAddress localAddr,int localPort)* a *Socket(InetAddress host,int port,InetAddress localAddr,int localPort)* dovolují navíc určit i lokální IP adresu a lokální port. Konečně, konstruktory *Socket(String host,int port,boolean stream)* a *Socket(InetAddress host,int port,boolean stream)* vytvářejí socket klienta, určují, zda jde o socket pro datagramovou službu nebo pro TCP spojení, a pro TCP spojení otevírají spojení na zadaný server.

Metody *getInetAddress* a *getLocalAddress* vracejí IP adresu vzdáleného počítače a lokální IP adresu, přes níž je otevřeno TCP spojení, metoda *getPort* vrací číslo vzdáleného a metoda *getLocalPort* číslo lokálního portu. Metody *getInputStream* a *getOutputStream* vrací odkazy na vstupní a výstupní proud spojený s TCP spojením. Metody *setTcpNoDelay* a *getTcpNoDelay* dovolují ovládat a testovat stav Nagleova mechanismu. Metody *setSoLinger* a *getSoLinger* dovolují pozdržet uzavření socketu do předání všech odeslaných dat. Metody *setSoTimeout* a *getSoTimeout* nastavují časový limit pro metodu read() na zadaný počet milisekund a dovolují nastavenou hodnotu zjistit. Vypršení časového limitu je indikováno výjimkou *InterruptedIoException*, nulová hodnota časového limitu odpovídá neomezenému čekání na příjem zprávy. Metoda *close()* uzavírá TCP spojení a socket a metoda *toString* předefinovává odpovídající metodu třídy *Object*.

Konečně, metoda *setSocketImplFactory* dovoluje definovat implicitní implementaci pro vytvářené sockety. S vytvářením socketů je spojeno vyvolání metody *createSocketImpl* zvolené instance třídy implementující rozhraní *SocketImplFactory*.

5.3.3 Třída SocketImpl

Třída *SocketImpl* je společným předkem všech tříd implementujících sockety UDP a TCP komunikace.

```
public abstract class SocketImpl extends Object implements SocketOptions {
    protected FileDescriptor fd
    protected InetAddress address
    protected int port
    protected int localport
    public SocketImpl()
    protected abstract void create(boolean stream) throws IOException
```

```
    protected abstract void connect(String host,int port) throws IOException
    protected abstract void connect(InetAddress address,int port)
                throws IOException
    protected abstract void bind(InetAddress host,int port) throws IOException
    protected abstract void listen(int backlog) throws IOException
    protected abstract void accept(SocketImpl s) throws IOException
    protected abstract InputStream getInputStream() throws IOException
    protected abstract OutputStream getOutputStream() throws IOException
    protected abstract int available() throws IOException
    protected abstract void close() throws IOException
    protected FileDescriptor getFileDescriptor()
    protected InetAddress getInetAddress()
    protected int getPort()
    protected int getLocalPort()
    public String toString()
}
```

Atributy a metody třídy *SocketImpl* odpovídají funkcím BSD socketů jak je známe z knihoven TCP/IP UNIXu. Atributy *fd*, *address*, *port* a *localport* odpovídají deskriptoru, adrese vzdáleného serveru, a číslům obou portů TCP spojení.

Metoda *create* vytváří socket, parametr *stream* určuje, zda půjde o socket pro UDP (stream==false) nebo TCP (stream==true) komunikaci. Metody *connect* slouží klientu k navázání TCP spojení se vzdáleným serverem. Metoda *bind* podporuje vazbu na lokální IP adresu a port. Metody *listen* a *accept* dovolují serveru určit délku fronty pro příchozí volání (implicitní hodnotou je 50) a převzít volání klienta na socketu zadané třídy. Metody *InputStream* a *OutputStream* vrací odkaz na odpovídající vstupní a výstupní proud, metoda *available* informuje o počtu znaků, které můžeme bez čekání z TCP spojení odebrat a metoda *close* TCP spojení a socket uzavírá. Metody *getFileDescriptor*, *get InetAddress*, *getPort* a *getLocalPort* vrací hodnoty atributů *fd*, *address*, *port* a *localport*. Konečně, metoda *setSocketImplFactory* dovoluje definovat implicitní implementaci pro vytvářené sockety.

5.3.4 Rozhraní SocketImplFactory

Rozhraní *SocketImplFactory* je využíváno třídami Socket a ServerSocket při vytváření socketů.

```
public interface interface SocketImplFactory {
    public abstract SocketImpl createSocketImpl()
}
```

Má jedinou metodu *createSocketImpl*, která vytváří instance typu SocketImpl.

5.3.5 Příklad - TCP Echo Server

Jako příklad TCP komunikace si uvedeme podobně jako u datagramové služby UDP jednoduchý *Echo Server* informující o přijatých zprávách a vracející je klientům.

```
import java.net.*;
import java.io.*;
public class TCPServer {
   static final int port = 7777;
   public static void printInfo(InetAddress a,int p,String s) {
      System.out.println("Client IP address : "+a.getHostAddress()+":"+p+":");
      System.out.println("Client domain address : "+a.getHostName()+":"+p+":");
      System.out.println("Message echoed : "+"["+s.length()+"] "+s);
   }
   public static void main(String args[]) throws Exception {
      ServerSocket control = null;
      try {
         control = new ServerSocket(port);
      }
      catch (IOException e) {
         System.out.println(e); System.exit(1);
      }
      Socket data = null;
      System.out.println("Waiting for connection ...");
      do {
         try {
            data = control.accept();
         }
         catch (IOException e) {
            System.out.println(e); break;
         }
         try {
            BufferedReader in = new BufferedReader(
                              new InputStreamReader(data.getInputStream()));
            OutputStream out = data.getOutputStream();
            String inputLine = in.readLine();
            printInfo(data.getInetAddress(),data.getPort(),inputLine);
            inputLine = inputLine + "\n";
            out.write(inputLine.getBytes()); out.flush();
            in.close(); out.close();
         }
         catch (IOException e) {
            System.out.println(e); data.close(); break;
         }
         finally {
            data.close();
         }
      } while (true);
      control.close();
   }
}
```

Server využívá dva sockety - *control* pro navazování TCP spojení a *data* pro přenos zpráv. Socket *control* je při vytváření navázán na port 7777, socket *data* vytváří a na stejný port váže metoda *accept*. Výstup z TCP spojení je navázán na vstupní proud *in* a spolu s údajem

o IP adrese a čísle portu klienta zveřejněn na systémovém výstupu *System.out*, a přes výstupní proud *out* odeslán zpět klientovi. Na závěr jsou proudy *in* a *out* a socket *data* uzavřeny. Server svou činnost v nekonečném cyklu opakuje, výjimka při navazování spojení nebo při přebírání, výstupu a odesílání zprávy způsobí ukončení výpočtu.

```
import java.net.*;
import java.io.*;
public class TCPClient {
   static final int port = 7777;
   public static void printInfo(InetAddress a,int p,String s) {
      System.out.println("Destination domain address : "+a.getHostName()+":"+p);
      System.out.println("Destination IP address : "+a.getHostAddress()+":"+p);
      System.out.println("Message sent : "+"["+s.length()+"] "+s);
   }
   public static void main(String args[]) throws Exception {
      if (args.length!=2) {
         System.out.println("Usage: java TCPClient <hostname> <message>");
         System.exit(1);
      }
      InetAddress address = null;
      try {
         address = InetAddress.getByName(args[0]);
      }
      catch (UnknownHostException e) {
         System.out.println(e); System.exit(2);
      }
      printInfo(address,port,args[1]);
      String s = args[1] + "\n";
      byte[] message = s.getBytes();
      int msglen = args[1].length();
      Socket data = null;
      try {
         data = new Socket(args[0],port);
      }
      catch (IOException e) {
         System.out.println(e); System.exit(3);
      }
      try {
         OutputStream out = data.getOutputStream();
         BufferedReader in = new BufferedReader(
                           new InputStreamReader(data.getInputStream()));
         out.write(message); out.flush();
         String response = in.readLine();
         System.out.println("Message returned : "+
                           "["+response.length()+"] "+response);
         in.close(); out.close();
      }
      catch (IOException e) {
         System.out.println(e);
      }
      finally {
```

```
            data.close();
        }
    }
}
```

Činnost klienta je velice jednoduchá. Po zkontrolování počtu parametrů je spolu s vytvořením socketu navázáno TCP spojení k serveru. Na vstup TCP spojení je navázán proud *os* na jeho výstup proud *is*. Předávaná zpráva (řetěz) je odeslána a po příjmu odpovědi a jejím výstupu na *System.out* jsou vstupní a výstupní proud a TCP socket uzavřeny a výpočet ukončen.

Výjimky v programu klienta vedou na výstup chybového hlášení na *System.out* a na ukončení výpočtu. Za povšimnutí stojí doplnění přechodu na nový řádek na konec zprávy, ten vyžaduje použití metody readLine na straně příjemce.

Příklad, který jsme si uvedli, odpovídá schématu obsluhy na straně serveru, kterou známe pod názvem *iterativní server*. Na rozdíl od implementace v jazyce C si všimneme rozdílu v tom, že pečlivě odlišujeme socket třídy *ServerSocket* sloužící pro navazování spojení a socket třídy Socket pro vlastní přenos dat.

Implementace paralelního serveru (*concurrent server*) v jazyce Java (na rozdíl od jazyka C, kde se musíme opřít o knihovny pro práci s procesy) je celkem jednoduchá a výsledný program je nezávislý na cílovém počítači/operačním systému. Možnou paralelní verzi *Echo Serveru* si uvedeme, klient může zůstat shodný s předcházejícím řešením

```
import java.net.*;
import java.io.*;
public class TCPCServer extends Thread {
   public static final int port = 7777;
   protected ServerSocket control = null;
   public TCPCServer() {
      try {
         control = new ServerSocket(port);
      }
      catch (IOException e) {
         System.out.println(e); System.exit(1);
      }
      System.out.println("Waiting for connection ...");
      this.start();
   }
   public void run() {
      do {
         try {
            Socket data = control.accept();
            Connection c = new Connection(data);
         }
         catch (IOException e) {
            System.out.println(e); break;
         }
      } while (true);
      try {
         control.close();
```

```
        }
        catch (IOException e) {
           System.out.println(e); System.exit(1);
        }
     }
     public static void main(String args[]) throws Exception {
        new TCPCServer();
     }
  }
  class Connection extends Thread {
     protected Socket data = null;
     BufferedReader in = null;
     OutputStream out = null;
     public Connection(Socket data_socket) {
        data = data_socket;
        try {
           in = new BufferedReader(
              new InputStreamReader(data.getInputStream()));
           out = data.getOutputStream();
           this.start();
        }
        catch (IOException e) {
           System.out.println(e);
        }
     }
     public static void printInfo(InetAddress a,int p,String s) {
        System.out.println("Client IP address : "+a.getHostAddress()+":"+p+":");
        System.out.println("Client domain address : "+a.getHostName()+":"+p+":");
        System.out.println("Message echoed : "+"["+s.length()+"] "+s);
     }
     public void run() {
        try {
           String inputLine = in.readLine();
           printInfo(data.getInetAddress(),data.getPort(),inputLine);
           inputLine = inputLine + "\n";
           out.write(inputLine.getBytes()); out.flush();
           in.close(); out.close();
        }
        catch (IOException e) {
           System.out.println(e);
        }
        try {
           data.close();
        }
        catch (IOException e) {
           System.out.println(e);
        }
     }
  }
```

5.4 Podpora URL přenosů

5.4.1 Třída URL

Třída *URL* dovoluje vytvářet odkazy na objekty v síti známé jako *URL odkazy* (Universal Resource Location) a efektivně s takto zpřístupněnými objekty pracovat. URL odkazy, které využíváme při práci s třídou *URL* vycházejí z materiálu RFC 1738, ale omezují se na odkazy, které mají syntaktickou formu
<protocol>: //<host>:<port>/<url-path>#<anchor>.

Metody třídy *URL* dovolují vytvořit spojení s URL objektem a s použitím odpovídajícího přenosového protokolu URL objekt přenést ve formě proudu oktetů. Metoda *getContent* dokonce dovoluje přenesený objekt rekonstruovat.

```
public final class URL extends Object implements Serializable {
   public URL(String protocol,String host,int port,String file)
                     throws MalformedURLException
   public URL(String protocol,String host,String file)
                     throws MalformedURLException
   public URL(String spec) throws MalformedURLException
   public URL(URL context,String spec) throws MalformedURLException
   protected void set(String protocol,String host,int port,String file,
                     String ref)
   public int getPort()
   public String getProtocol()
   public String getHost()
   public String getFile()
   public String getRef()
   public boolean equals(Object obj)
   public int hashCode()
   public boolean sameFile(URL other)
   public String toString()
   public String toExternalForm()
   public URLConnection openConnection() throws IOException
   public final InputStream openStream() throws IOException
   public final Object getContent() throws IOException
   public static synchronized void setURLStreamHandlerFactory
                     (URLStreamHandlerFactory fac)
}
```

Konstruktor *URL(String protocol,String host,int port,String file)* dovoluje vytvořit URL odkaz z jednotlivých složek. Protože mechanismus přenosu a rekostrukce URL objektů musí být schopen zvládnout nejen všechny dnes používané, ale i ještě neznámé standardní protokoly a protokoly nestandardní, je jeho řešení poměrně komplikované:

Pro první vytvářený URL odkaz daného protokolu v aplikaci je vnitřně vytvářena instance třídy *URLStreamHandler - URL ovladač* (URL handler). Ten lze získat prostřednictvím dříve vytvořené instance třídy implementující rozhraní *URLStreamHandlerFactory*, přesněji jako výsledek vyvolání metody *createURLStreamHandler* takového objektu. Pokud taková instance neexistuje, nebo pokud její metoda *createURLStreamHandler* vrátí *null*, je potřebný URL ovladač vyhledáván v knihovnách (package) vyjmenovaných v systémovém parametru

java.handler.protocol.pkgs (jako seznam jmen oddělených znakem |). Pokud v těchto knihovnách není potřebný URL ovladač pod jménem *<package>.<protocol>.Handler* nalezen, je hledán pod jménem *sun.net.www.protocol.<protocol>.Handler*. Teprve nevede-li ani poslední krok k nalezení URL ovladače pro zadaný protokol, končí vytváření instance URL odkazu výjimkou *MalformedURLException*.

Konstruktor *URL(String protocol,String host,String file)* předpokládá použití standardního portu pro zadaný protokol (například port *80* pro protokol *http*). Stejný význam má uvedení hodnoty -1 jako čísla portu v konstruktoru předcházejícím. Konstruktor *URL(String spec)* dovoluje zadat kompletní URL odkaz jako řetěz. Konečně, konstruktor *URL(URL context,String spec)* dovoluje odvodit URL odkaz z jiného URL odkazu *context* doplněním prvků uvedených v parametru *spec*.

Privátní metodu *set* vyžívají vnitřně URL ovladače. Metody *getPort*, *getProtocol*, *getHost*, *getFile*, *getRef* vrací číslo portu, označení protokolu, jméno cílového počítače, textový řetězec popisující cestu k souboru a odkaz v rámci souboru (využívaný uvnitř HTML stránek). Metoda *equals* porovnává URL odkaz na shodu s jiným URL odkazem. Metoda *sameFile* oproti metodě *equals* ignoruje odkaz v rámci souboru. Metody *hashCode* a *toString* redefinují odpovídající metody třídy Object. Metoda *toString* přitom využívá výsledku metody *toExternalForm* poskytující textový tvar URL odkazu.

Další metody souvisí s otevřením spojení k URL objektu a s jeho přenosem. Metoda *openConnection* otevírá spojení vyvoláním metody *openConnection* příslušného URL ovladače. Metoda *openStream* otevírá spojení k URL objektu a váže na něj vstupní proud, jde o zkrácený zápis pro *openConnection().getInputStream()*. Metoda *getContent* otevírá spojení k URL objektu a vrací hodnotu tohoto objektu, jde o zkrácený zápis pro *openConnection().getContent()*. Úkolem posledních dvou metod je zjednodušení přístupu ke vzdálenému objektu.

Statická metoda *setURLStreamHandlerFactory* nám dovoluje zvolit instanci dále popsané třídy *URLStreamHandlerFactory*, její metoda *createURLHandler* vrací URL ovladač vybíraný při vytváření URL odkazu.

5.4.2 Třída URLConnection

Třída *URLConnection* podporuje přenos stránek HTML protokolem http a v obráceném směru zadávání parametrů pro výběr nebo generování stránek dalších (např. parametry CGI skriptů). Některé metody mají význam pouze pro http protokol, řada metod je však použitelná i pro jiné protokoly.

```
public abstract class URLConnection extends Object {
   protected URL url
   protected boolean doInput
   protected boolean doOutput
   protected boolean allowUserInteraction
   protected boolean useCaches
   protected long ifModifiedSince
   public static FileNameMap fileNameMap
   protected boolean connected
   protected URLConnection(URL url)
   public abstract void connect() throws IOException
   public URL getURL()
   public int getContentLength()
```

```
    public String getContentType()
    public String getContentEncoding()
    public long getExpiration()
    public long getDate()
    public long getLastModified()
    public String getHeaderField(String name)
    public int getHeaderFieldInt(String name,int Default)
    public long getHeaderFieldDate(String name,long Default)
    public String getHeaderFieldKey(int n)
    public String getHeaderField(int n)
    public Object getContent() throws IOException
    public InputStream getInputStream() throws IOException
    public OutputStream getOutputStream() throws IOException
    public String toString()
    public void setDoInput(boolean doinput)
    public boolean getDoInput()
    public void setDoOutput(boolean dooutput)
    public boolean getDoOutput()
    public void setAllowUserInteraction(boolean allowuserinteraction)
    public boolean getAllowUserInteraction()
    public static void setDefaultAllowUserInteraction
                      (boolean defaultallowuserinteraction)
    public static boolean getDefaultAllowUserInteraction()
    public void setUseCaches(boolean usecaches)
    public boolean getUseCaches()
    public void setIfModifiedSince(long ifmodifiedsince)
    public long getIfModifiedSince()
    public boolean getDefaultUseCaches()
    public void setDefaultUseCaches(boolean defaultusecaches)
    public void setRequestProperty(String key,
    public String getRequestProperty(String key)
    public static void setDefaultRequestProperty(String key,String value)
    public static String getDefaultRequestProperty(String key)
    public static synchronized void setContentHandlerFactory
                      (ContentHandlerFactory fac)
    protected static String guessContentTypeFromName(String fname)
    public static String guessContentTypeFromStream(InputStream is)
                      throws IOException
}
```

Třída má řadu privátních atributů a metod pro jejich čtení a modifikaci. Atribut *URL* reprezentuje URL odkaz, kterého se URL spojení týká, atributy *doInput* a *doOutput* indikují požadovaný směr přenosů. Atribut *allowUserInteraction* povoluje interakci s uživatelem v protokolech, které toho mohou využít (např. autentifikace). Atributy *useCaches* a *ifModifiedSince* dovolují povolit nebo zakázat využití lokálně schraňovaných kopií a omezit přístup ke starým objektům. Atribut *connected* indikuje otevřené spojení s URL objektem.

Třída má jediný konstruktor *URLConnection(URL url)* dovolující vytvořit spojení k objektu zadanému URL odkazem.

Metoda *connect* otevírá URL spojení. Před otevřením spojení je možné měnit hodnoty některých atributů, např. atributu *useCaches*. Metoda *getURL* vrací URL odkaz, metody

getContentLength, *getContentType*, *getContentEncoding*, *getExpiration*, *getDate*, *getLastModified*, *getHeaderField*, *getHeaderFieldInt*, *getHeaderFieldDate*, *getHeaderFieldKey* a *getHeaderField* vrací položky z hlavičky URL objektu.

Metoda *getContent* přenáší hodnotu URL objektu po otevřeném spojení (pokud dosud otevřené nebylo, pak si ho otevře) a po příjmu ho rekonstruuje. Protože je nutné zvládnout nejen všechny dnes používané, ale i ještě neznámé standardní formáty a formáty nestandardní, je činnost této metody (podobně jako u vyhledáváni URL ovladače pro zadaný přenosový protokol) poměrně komplikovaná:

Metoda *getContent* se opírá o zjištěný typ URL objektu, případně si ho voláním metody *getContentType* sama zjistí. Pro první práci s daným typem URL objektu v aplikaci je vnitřně vytvářena instance třídy *ContentHandler*. Takový objekt lze získat prostřednictvím dříve vytvořené instance třídy implementující rozhraní *ContentHandlerFactory*, přesněji jako výsledek vyvolání metody *createContentHandler* této instance. Pokud taková instance neexistuje, nebo pokud její metoda *createContentHandler* vrátí *null*, je potřebný ovladač hledán pod jménem *sun.net.www.content.<contentType>* (po náhradě lomítek tečkami v řetězu *contentType*). Teprve nevede-li ani poslední krok k nalezení ovladače, končí pokus o přenos hodnoty URL objektu výjimkou *UnknownServiceException*.

Metody *getInputStream* a *getOutputStream* vrací odkazy na proudy pro vstup z URL spojení a výstup do URL spojení. Metoda *toString* redefinuje odpovídající hodnotu třídy *Objekt*.

Třída dále implementuje řadu metod dovolujících zjistit a modifikovat hodnoty atributů. Konečně metoda *setContentHandlerFactory* vybírá objekt s implementovaným rozhraním *ContentHandlerFactory* vytvářející ovladače pro přenos a rekonstrukci URL objektů. Metody *guessContentTypeFromName* a *guessContentTypeFromStream* podporují zjištění typu URL objektu ze jména souboru a z hodnoty URL objektu.

5.4.3 Třída URLEncoder

Pomocná třída *URLEncoder* obsahuje metodu dovolující překódovat libovolný řetěz do formátu MIME "x-www-form-encoded".

```
public class URLEncoder extends Object {
   public static encode (String s)
}
```

Její metoda *encode* realizuje transformaci, při níž jsou mezery nahrazeny znakem '+' a všechny znaky s výjimkou písmen a číslic svou hexadecimální reprezentací prefixovanou apostrofem.

5.4.4 Třída URLStreamHandler

Třída *URLStreamHandler* je společným předkem všech ovladačů pro přenos URL objektů (URLStreamHandler). Ovladače jsou k programu dynamicky přisestavovány způsobem, uvedeným v popisu třídy URL.

```
public abstract class URLStreamHandler extends Object {
   public URLStreamHandler()
   protected abstract URLConnection openConnection(URL u)
```

```
    protected void parseURL(URL u,String spec,int start,int limit)
    protected String toExternalForm(URL u)
    protected void setURL(URL u,String protocol,String host,int port,String file,
                          String ref)
}
```

Třída má pouze základní konstruktor *URLStreamHandler()*. Metoda *openConnection* otevírá spojení s URL objektem. Metoda *parseURL* překládá řetěz na URL odkaz, parametry *start* a *limit* omezují zpracování na výřez řetězu *string*. Metoda *toExternalForm* převádí URL odkaz do textové formy. Metoda *setURL* vytváří URL odkaz z jednotlivých položek.

5.4.5 Rozhraní URLStreamHandlerFactory

Rozhraní *URLStreamHandlerFactory* dovoluje vyhledat k danému typu přenosového protokolu příslušný ovladač. Má jedinou metodu:

```
public interface interface URLStreamHandlerFactory {
    public abstract URLStreamHandler createURLStreamHandler(String protocol)
}
```

Metoda *createURLStreamHandler* má na starosti instalaci ovladače pro protokol využívaný pro přenos objektu.

5.4.6 Třída ContentHandler

Třída *ContentHandler* je společným předkem všech ovladačů pro přenos objektů (ContentHandler). Ovladače jsou k programu dynamicky přisestavovány způsobem, uvedeným v popisu třídy URLConnection.

```
public abstract class ContentHandler extends Object {
    public ContentHandler()
    public abstract Object getContent(URLConnection urlc) throws IOException
}
```

5.4.7 Rozhraní ContentHandlerFactory

Rozhraní *ContentHandlerFactory* dovoluje vyhledat k danému typu přenášeného objektu příslušný ovladač. Má jedinou metodu:

```
public interface interface ContentHandlerFactory {
    public abstract ContentHandler createContentHandler(String mimetype)
}
```

Metoda *createContentHandler* má na starosti instalaci ovladače pro typ objektu, který má být přenesen.

5.4.8 Rozhraní FileNameMap

Rozhraní *FileNameMap* dovoluje vyhledat k danému jménu souboru obsahujícího URL objekt přiřadit odpovídající typ MIME. Má jedinou metodu:

```
public interface interface FileNameMap {
   public abstract String getContentTypeFor(String fileName)
}
```

Metoda *getContentTypeFor* vrací informaci o typu MIME odvozenou ze jména soboru.

5.4.9 Příklady - přenos URL objektu

Jako ukázku použití prostředků pro přístup k URL objektům si uvedeme dva příklady. První příklad používá pro přenos URL objektu metody *getContent* třídy *URL*. Opírá se přitom o existenci vestavěného ovladače (ContentHandler), který umí přijímat "prosté texty" (objekty typu "text/plain").

```
import java.net.*;
import java.io.*;
public class URLGet_1 {
   public static String fetch(String address)
      throws MalformedURLException, IOException {
      URL url = new URL(address);
      System.out.println(url.toExternalForm());
      System.out.println(" Protocol : " + url.getProtocol());
      System.out.println(" Host : " + url.getHost());
      System.out.println(" File : " + url.getFile());
      return (String) url.getContent();
   }
   public static void main(String args[])
      throws MalformedURLException, IOException {
      if (args.length!=1) {
         System.out.println("Usage: java URLGet_1 <url>");
         System.exit(0);
      }
      System.out.println(fetch(args[0]));
   }
}
```

Náš druhý příklad otevírá URL spojení, zveřejňuje atributy URL objektu a hodnotu URL objektu na systémovém výstupu *System.out*. Výhodou tohoto přístupu je, že můžeme přenést libovolný objekt nezávisle na jeho typu, o jeho vytvoření z URL reprezentace se ale musí postarat vlastní aplikace.

```
import java.net.*;
import java.io.*;
import java.util.*;
public class URLGet_2 {
```

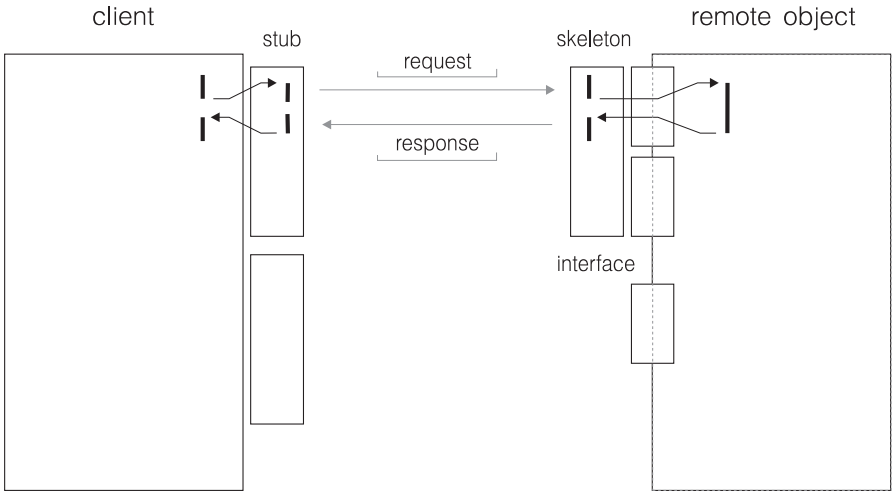
```
    public static void printinfo(URLConnection u) throws IOException {
       System.out.println(u.getURL().toExternalForm() + ":");
       System.out.println(" Content Type : " + u.getContentType());
       System.out.println(" Content Length : " + u.getContentLength());
       System.out.println(" Last Modified : " + new Date(u.getLastModified()));
       System.out.println(" Expiration : " + u.getExpiration());
       System.out.println(" Content Encoding : " + u.getContentEncoding());
       try {
          BufferedReader dis = new BufferedReader
                                    (new InputStreamReader(u.getInputStream()));
          String inputLine;
          while ((inputLine = dis.readLine()) != null) {
          System.out.println(inputLine);
          }
          dis.close();
       } catch (IOException me) {
          System.out.println("IOException A: " + me);
       }
    }
    public static void main(String args[])
       throws MalformedURLException, IOException {
       if (args.length!=1) {
          System.out.println("Usage: java URLGet_2 <url>");
          System.exit(0);
       }
       try {
          URL url = new URL(args[0]);
          URLConnection connection = url.openConnection();
          printinfo(connection);
       } catch (MalformedURLException me) {
          System.out.println("MalformedURLException: " + me);
       } catch (IOException me) {
          System.out.println("IOException B: " + me);
       }
    }
}
```

5.5 Procedurální komunikace - RMI

Rozhraní standardních komunikačních kanálů UDP a TCP a možnost přístupu k vyšším protokolům prostřednictvím URL knihovny nejsou jedinou podporovanou formou komunikace. Současná implementace jazyka Java zahrnuje také procedurální objektově orientovanou komunikaci, ta je označována jako *RMI - Remote Method Invocation*.

Ve srovnání s jinými, dříve uvedenými, systémy SunRPC/ONC, OSF DCE nebo ještě výrazněji ve srovnání se systémem CORBA (Common Object Request Brooker Architecture) je RMI jednodušší - opírá se totiž o objektový přístup podporovaný samotným jazykem Java a omezuje se pouze na tento jazyk.

Strukturu mechanismu RMI si lze popsat na obr. 5.2.



Obr. 5.2: Komunikace RMI

Mechanismus RMI zajišťuje komunikaci mezi *klientem* (klientskou částí aplikace), kterým může být libovolný objekt jazyka Java (nebo lépe vlákno výpočtu, které interpretuje kód nějakého objektu) a *serverem*, kterým může být libovolný *vzdálený objekt* realizující určitou obsluhu. Ten se liší od běžného objektu (z kterého je odvozen) v některých předefinovaných metodách; základním rozdílem je zpřístupnění vzdáleného objektu pro volání klientem.

Komunikaci mezi klientem a vzdáleným objektem definuje *rozhraní RMI*. To je výčtem metod, které vzdálený objekt pro klienta zajišťuje, a které musí být vzdáleným objektem implementovány. Vzdálený objekt může současně implementovat více různých rozhraní RMI.

Klient může získat odkaz na zpřístupněný vzdálený objekt prostřednictvím systémové aplikace *registry* dovolující registrovat vzdálený objekt na daném počítači pod zvoleným jménem, další možností je získat odkaz jako výsledek jiného vzdáleného volání.

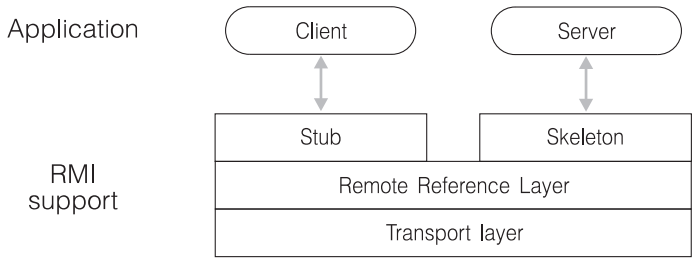
Klient spolupracuje s lokálním zástupcem vzdáleného objektu - ten označujeme jako *stub*. Aktivaci metod rozhraní RMI (označovanou jako *Remote Method Invocation*) budeme označovat jako *vzdálená volání*. Na rozdíl od běžné aktivace metody mohou vzdálená volání překlenout hranici mezi logickými stroji (Java Virtual Machine); běh klienta a serveru na různých strojích však není podmínkou.

Při vzdáleném volání jsou předávány parametry a výsledek jako posloupnosti oktetů. O transformaci běžné aktivace metody na takovou posloupnost (používáme zde termínů *marshalling* a *unmarshalling*) se stará *stub* na straně klienta, odpovídající stub na straně vzdáleného objektu (ten převádí posloupnost zpět na volání) je označován jako *stub serveru* nebo *skeleton*. Stuby mohou předávat primitivní typy, vzdálené objekty (přesněji odkazy na

ně) a objekty (jako hodnoty), které implementují rozhraní *java.io.Serializable* (a jsou tedy převeditelné na posloupnost oktetů). Pokud parametr nebo výsledek vzdáleného volání nepatří do některé z uvedených skupin, končí vzdálené volání výjimkou.

Schéma na obr. 5.2 kromě synchronního charakteru volání RMI zdůrazňuje skutečnost, že skeleton je svázán s třídou popisující vzdálený objekt (zahrnuje všechna implementovaná rozhraní) a že klient může spolupracovat s více vzdálenými objekty jedné nebo více tříd. O vytvoření stubu a skeletonu se postará generátor *rmic*.

Mechanismus komunikace mezi klientem a vzdáleným objektem je nejviditelnější vrstvou RMI systému, celkovou strukturu RMI systému si můžeme popsat oblíbeným schématem vrstev - obr. 5.3.



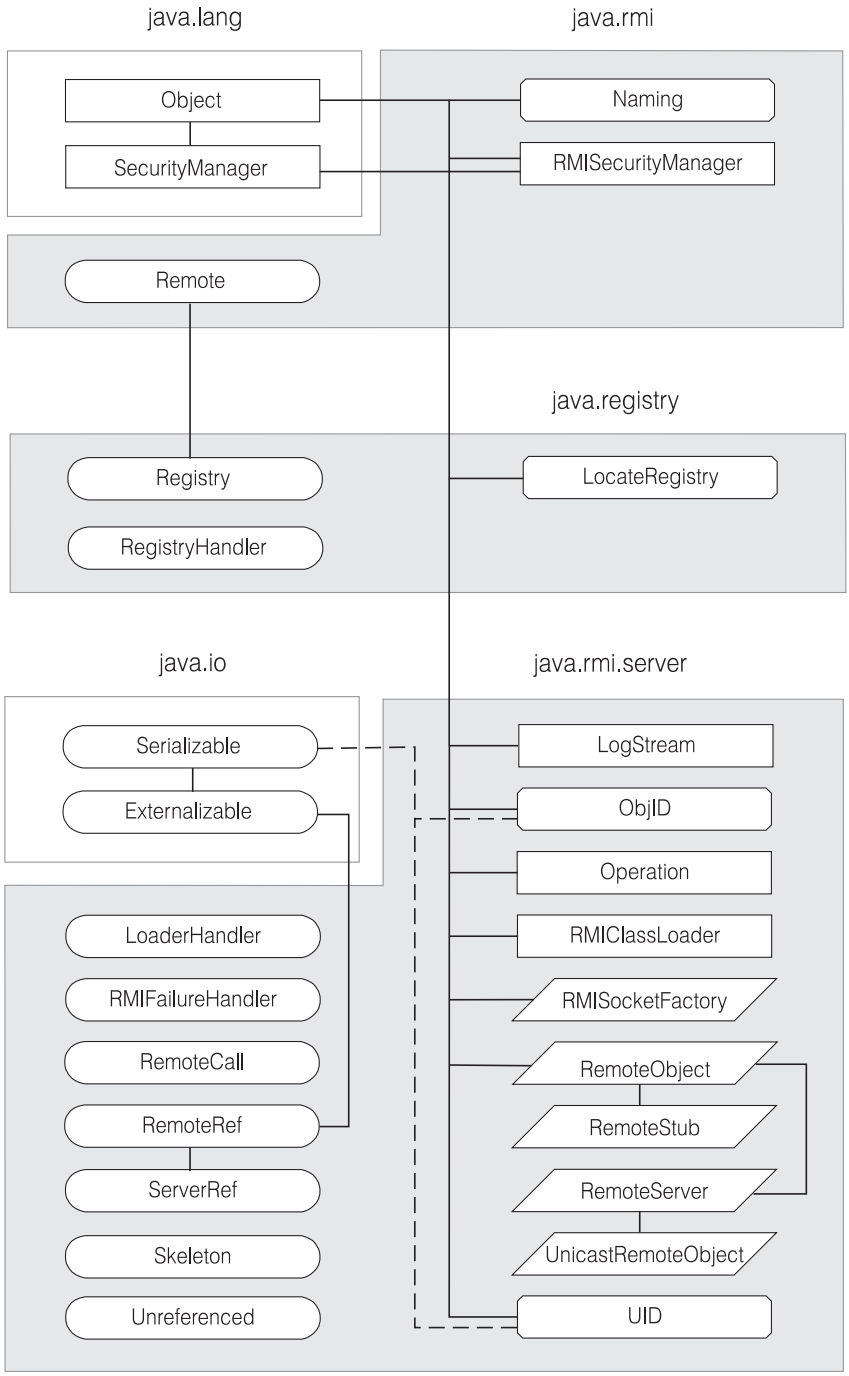
Obr. 5.3: Architektura systému RMI

Pod vrstvou komunikace mezi stubem klienta a skeletonem vzdáleného objektu se skrývá velice zajímavá vrstva označovaná jako *Remote Reference Layer*. Ta je zodpovědná za *sémantiku* vzdálených volání, tedy za způsob zajištění proti výpadku komunikace a havárii výpočtu vzdáleného objektu (sémantiky *"at-most-once"*, *"at-least-once"* a *"exactly- once"*), za způsob synchronizace klienta a vzdáleného objektu (standardem je *synchronní* vyvolání vzdálené metody, *asynchronní* vyvolání není v současném systému podporováno) a za způsob implementace vzdáleného objektu (trvalá nebo dočasná *aktivita* vzdáleného objektu, správa *persistentních* vzdálených objektů, *replikační strategie* a *mobilita*).

Konečně, celý RMI mechanismus je podporován *transportní službou* zodpovědnou za navazování a správu komunikačních spojení pod vhodným protokolem. K podpoře RMI mechanismu konečně patří i registrace vzdálených objektů (v rámci jednoho stroje) schopných přijímat RMI volání - program *registry*.

Kromě dosud uvedených prvků, které přímo podporují spolupráci částí distribuované aplikace, zahrnuje RMI systém prostředky podporující uvolňování paměti po vzdálených objektech (distribuované rozšíření čističe paměti - *garbage collectoru*), dynamické zavádění definic tříd (pro stuby, skeletony a vzdálená rozhraní) nejenom z lokálních systémů souborů (*java.lang.ClassLoader*), ale i ze sítě (*java.rmi.server.RMIClassLoader*) a zabezpečení systému proti vnějšímu narušení (*java.rmi.RMISecurityManager*).

Ale vraťme se nyní k základnímu mechanismu RMI, ten je podporován třídami systémových knihoven *java.rmi*, *java.rmi.server*, *java.rmi.registry* a *java.rgi.DGM*. Strukturu těchto knihoven uvádí obr. 5.4. Popisu jednotlivých tříd této knihovny se nyní budeme věnovat podrobněji.



Obr. 5.4: Třídy RMI

5.5.1 Rozhraní RMI

Rozhraní RMI pro konkrétní aplikaci, zpřístupňující metody vzdáleného objektu, odvodíme z rozhraní *java.rmi.remote*. Rozhraní *java.rmi.remote* má velice stručnou signaturu:

```
package java.rmi;
public interface Remote { }  ,
```

nezahrnuje tedy žádné veřejně přístupné metody.

Rozhraní RMI aplikace, z rozhraní *java.rmi.remote* odvozené, musí být veřejně dostupné (*public*). Každá z jeho metod musí mít v seznamu výjimečných ukončení (klauzule *throws*) uvedenu výjimku *java.rmi.RemoteException*, která je základem pro všechny výjimky související s RMI komunikací. To dovolí klientské části aplikace reagovat specifickým způsobem na výpadky v síťové komunikaci nebo na problémy vzniklé na vzdáleném objektu/serveru.

Na parametry a výsledek metod rozhraní RMI jsou kladena určitá, již uvedená, omezení. Nejdůležitějším je zřejmě skutečnost, že objekty lokálně dostupné klientské části aplikace jsou předávány *hodnotou* (kopírovány, do kopií pouze nejsou zahrnuty položky se specifikacemi *static* a *transient*). Vzdálené objekty jsou naproti tomu identifikovány *odkazem*, odkaz musí být navíc směrován na určité rozhraní RMI vzdáleného objektu a ne na vzdálený objekt samotný.

Jako ilustrační příklad definice rozhraní RMI si uvedeme uživatelské rozhraní bankovního účtu dovolující uložit (metoda *deposit*) nebo vyzvednout (metoda *withdraw*) zadaný obnos a zjistit (metoda *balance*) okamžitý stav účtu.

```
package bankaccount;
public interface BankAccount extends java.rmi.Remote {
   public void deposit(float amount)
         throws java.rmi.RemoteException;
   public void withdraw(float amount)
         throws OverDrawnException, java.rmi.RemoteException;
   public float balance()
         throws java.rmi.RemoteException;
}  .
```

Klientská část aplikace musí být schopna získat odkaz na RMI rozhraní vzdáleného objektu identifikovaného doménovým jménem nebo IP adresou počítače a výlučným jménem tohoto objektu na tomto počítači. Vytvoření potřebné vazby vzdáleného objektu na jméno a získání odkazu při zadání jména podporují metody třídy *Naming*. Třída má signaturu:

```
public final class Naming {
   public static Remote lookup(String name) throws NotBoundException,
            MalformedURLException, UnknownHostException, RemoteException;
   public static void bind(String name,Remote obj) throws AlreadyBoundException,
            MalformedURLException, UnknownHostException, RemoteException;
   public static void rebind(String name,Remote obj) throws
            MalformedURLException, UnknownHostException, RemoteException;
   public static void unbind(String name) throws NotBoundException,
            MalformedURLException, UnknownHostException, RemoteException;
   public static String[] list(String name) throws
            MalformedURLException, UnknownHostException, RemoteException;
}  .
```

Použití metod třídy Naming si můžeme ilustrovat na jednoduchých ukázkách. O registraci pod zadaným jménem (*"xyz"*) žádá vzdálený objekt *remObj* voláním metody *bind* (nebo *rebind*, pokud chceme předefinovat předchozí vazbu jména na jiný vzdálený objekt):

```
Naming.rebind("xyz",remObj);  .
```

Vyhledání tohoto serveru (budeme předpokládat, že k jeho registraci došlo na počítači *java*) klientem pak může mít formu:

```
BankAccount remObj = (BankAccount)Naming.lookup("//java/xyz");  .
```

Systém RMI vyžaduje použití vhodné bezpečnostní strategie, která nám zaručí, že objekt (například klientský applet načtený ze sítě prohlížečem) bude mít přístup k systémovým prostředkům (lokálním souborům, komunikačním kanálům) vhodně omezený. Konkrétní bezpečnostní strategii pro RMI aplikace volíme výběrem objektu třídy *RMISecurityManager* statickou metodou *System.setSecurityManager*.

```
System.setSecurityManager(new RMISecurityManager());  .
```

V systému zahrnutá třída *java.rmi.RMISecurityManager* (je odvozená ze základní třídy *java.lang.SecurityManager*) povoluje jen nejnutnější funkce potřebné pro přenos RMI odkazů, serializovaných reprezentací objektů a pro vlastní RMI volání. Implicitně použitý *java.lang.SecurityManager* dovoluje přístup jen k lokálním souborům. Rozšíření povolených funkcí oproti strategii *RMISecurityManager* lze dosáhnout definováním vlastní odvozené bezpečnostní strategie.

5.5.2 Vzdálené objekty

Třídy definující vzdálené objekty odvozujeme ze tříd *java.rmi.server.RemoteObject*, *java.rmi.server.RemoteServer* a *java.rmi.server.UnicastRemoteObject*. Termínem *vzdálený objekt* označujeme takto získaný objekt, bez ohledu na jeho skutečné umístění.

Třída *RemoteObject* modifikuje základní metody třídy *Object* pro vzdáleně přístupné objekty. Má signaturu

```
package java.rmi.server;
public abstract class RemoteObject
       implements java.rmi.Remote, java.io.Serializable {
   public int HashCode();
   public boolean equals(Object obj);
   public String toString();
}  .
```

Třída *RemoteServer* odvozená ze třídy *RemoteObject* vytváří základ pro implementaci metod vzdálených objektů. Signatura zahrnuje několik služebních metod:

```
package java.rmi.server;
public class RemoteServer extends RemoteObject {
   public static String getClientHost() throws ServerNotActiveException;
   public static void setLog(java.io.OutputStream out);
   public static java.io.PrintStream getLog();
}
```

Metoda *getClientHost* dovoluje zjistit počítač klienta, který vyvolal právě prováděnou metodu;

výjimka *ServerNotActiveException* indikuje situaci, kdy akce nebyla vyvolána vzdáleným voláním. Metoda *setLog* dovoluje zvolit výstupní kanál pro monitorování (parametrem *null* lze záznam vypnout), metoda *getLog* předává tento kanál ke zpracování.

Třída *RemoteServer* slouží jako základ, nad nímž lze vystavět mechanismus umožňující specifickou formu komunikace se vzdáleným objektem. Lze tak podpořit práci s replikovanými objekty, s objekty persistentními a podobně. V současnosti (verze 1.1.4) je běžně podporována pouze synchronní komunikace s jednoduchým objektem třídou *UnicastRemoteServer*. Ta má signaturu:

```
package java.rmi.server;
public class UnicastRemoteServer extends RemoteServer {
   protected UnicastRemoteObject() throws java.rmi.RemoteException;
   public Object clone() throws java.lang.CloneNotSupportedException;
   public static void exportObject(java.rmi.Remote obj)
         throws java.rmi.RemoteException;
}
```

Konstruktor *UnicastRemoteServer* vytváří vzdálený objekt. Výsledný objekt je zpřístupněn (*exported*) a může být použit jako parametr nebo výsledek vzdáleného volání. Metoda *clone* vytváří kopii vzdáleného objektu (třída *RemoteObject* totiž neimplementuje rozhraní *java.lang.Clonable*). Metoda *exportObject* zpřístupňuje objekt, který byl vytvořen jinak než konstruktorem *UnicastRemoteServer*. Použití nezpřístupněného objektu pro vzdálené volání vyvolá výjimku *java.rmi.server.StubNotFoundException*.

Jako příklad si uvedeme definici vzdáleného objektu BankAccountImpl implementujícího naše rozhraní bankovního účtu BankAccount:

```
package bankaccount;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
public class BankAccountImpl extends UnicastRemoteObject implements BankAccount {
   public void deposit(float amount) throws java.rmi.RemoteException {
   }
   public void withdraw(float amount)
         throws OverDrawnException, java.rmi.RemoteException {
   }
   public float balance() throws java.rmi.RemoteException {
   }
}
```

Třídy, které jsme si zatím uvedli vytváří kostru RMI aplikací a u jednoduchých aplikací s nimi bez problémů vystačíme. Chceme-li podrobně porozumět činnosti mechanismů skrývajících se za kódem generovaným překladačem *rmic* neobejdeme se bez pochopení některých podpůrných rozhraní a systémových tříd. Těm jsou věnovány následující odstavce.

Instance tříd implementující rozhraní *RemoteRef* odkazují na vzdálené objekty (na straně klienta). Protějšky těchto odkazů na straně vzdálených objektů jsou popsány třídami implementujícími rozhraní *ServerRef*. Jednotlivá volání na straně klienta zajišťují instance třídy implementující rozhraní *RemoteCall*, na straně serveru se o volání metod stará rozhraní *Skeleton*. Třída *RemoteStub* je kostrou všech klientských stubů.

Rozhraní *LoaderHandler* podporuje vzdálené dynamické přisestavování tříd, rozhraní RMIFailureHandler dovoluje informovat o problémech při přípravě TCP kanálu (přípravě socketů tříd Socket a ServerSocket). Konečně, rozhraní *java.rmi.server.Unreferenced* indikuje,

že na objekt již neexistuje žádný vzdálený odkaz. Jeho metoda *unreferenced* je aktivována při zrušení posledního ze vzdálených odkazů (to může být za dobu existence vzdáleného objektu i vícekrát).

Třída *RMISocketFactory* vytváří sockety pro RMI spojení. Funkce jejích metod *createSocket* a *createServerSocket* je poměrně zajímavá: pokouší se otevřít běžný TCP kanál, pokud neuspěje (např. pokud navázání TCP spojení nedovolí bezpečnostní brána (firewall) zkouší spojení HTTP protokolem po vlastním kanále, a pokud ani zde neuspěje, vytváří spojení přes implicitní kanál HTTP (TCP:80) a využívá funkce POST tohoto protokolu. Třída *RMIClassLoader* se stará o vzdálené dynamické přisestavování tříd. Třídy *UID* a *ObjID* vytvářejí jedinečné identifikátory v rámci jednoho virtuálního stroje. Třída *Operation* uchovává informace o metodách RMI rozhraní. Na závěr, třída *LogStream* podporuje monitorování chyb.

5.5.3 Příklad - jednoduchá aplikace klient-server

Jako reálný příklad použití technologie RMI si uvedeme jednoduchou aplikaci tvořenou klientem, který umí požádat o zopakování textového řetězce doplněného o pozdravení ("Hello"), a vzdálený objekt, který na požádání zaslaný řetězec doplněný o pozdravení ("Hello") vrátí.

Prvním krokem při programování aplikace RMI je definice RMI rozhraní, v našem případě toto rozhraní zahrnuje jedinou metodu, která vrací textový řetězec.

```
package hello;
public interface Hello extends java.rmi.Remote {
   String sayHello(String s) throws java.rmi.RemoteException;
}
```

Server, který rozhraní *Hello* implementuje a je typicky rozšířením některé z podtříd *java.rmi.server.RemoteObject*, musí kromě konstruktoru a vlastního kódu metody *sayHello* definovat bezpečnostní strategii a svou službu zaregistrovat.

```
package hello;
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
public class HelloImpl extends UnicastRemoteObject implements Hello {
   private String greeting = "Hello";
   public HelloImpl(String s) throws RemoteException {
      super();
      greeting = s;
   }
   public String sayHello(String s) throws RemoteException {
      return greeting+" "+s+"!";
   }
   public static void main(String args[]) {
      try {
         System.setSecurityManager(new RMISecurityManager());
      }
      catch(Exception e) {
         System.out.println(e); return;
      }
      try {
         HelloImpl obj = new HelloImpl("Salut");
         Naming.rebind("//java/HelloServer",obj);
         System.out.println("HelloServer bound in registry");
      }
      catch(Exception e) {
         System.out.println(e);
      }
   }
}
```

Explicitní konstruktor třídy *RemoteObject* (dovoluje zaměnit formu pozdravení) je uveden spíše pro ilustraci (implicitní konstruktor je aktivován automaticky). Výjimku *java.rmi.RemoteException* konstruktoru může vyvolat chyba v komunikačním systému indikovaná při instalaci vzdáleného objektu.

Prvním krokem statické metody *main* třídy definující vzdálený objekt/objekty je instalace objektu *RMISecurityManager* (případně jiného objektu odvozeného ze třídy *SecurityManager*, ale spíše ze třídy *RMISecurityManager*). Bez něj nelze zavést RMI objekty (stuby, skeletony) ani lokálně, *RMISecurityManager* dovoluje vzdálené zavádění stubů pro parametry a výsledek. Teprve potom lze instalovat jeden nebo více vzdálených objektů.

Aby bylo možné vzdálený objekt zpřístupnit klientovi, musíme mu umět poskytnout odkaz na tento objekt. Takovou službu poskytuje aplikace *registry*, metoda *Naming.rebind* v našem příkladě zpřístupňuje stub objektu *HelloImpl* pod jménem *HelloServer* na počítači *java*. Formát jména ve volání se řídí pravidly pro vytváření URL odkazů s tím, že nemusíme definovat protokol (tedy *rmi*), jméno počítače (jde-li o *localhost*) a můžeme využít implicitní port 1099, na němž je služba *registry* běžně dostupná (plný formát URL odkazu by pro náš příklad byl *rmi://java:1099/HelloServer*) Klient, který metodu *sayHello* volá musí nejprve zjistit odkaz na vzdálený objekt třídy *HelloImpl*, v našem příkladě vytvořením URL odkazu a jeho použitím jako parametru metody Naming.lookup.

```
package hello;
import java.rmi.*;
public class HelloClient {
   static String message = "";
   public static void main(String[] args) {
      try {
         Hello obj = (Hello)Naming.lookup("//java/HelloServer");
         message = obj.sayHello("client");
      }
      catch(Exception e) {
         System.out.println(e);
      }
      System.out.println(message);
   }
}
```

Po získání odkazu na stub vzdáleného serveru (v proměnné *obj*) už můžeme používat metody vzdáleného objektu běžným způsobem, pouze se musíme postarat o zpracování případných výjimek vyvolaných problémy při komunikaci.

Pro doplnění našeho přehledu si ještě uvedeme klienta, který má formu appletu zaváděného HTTP protokolem ze serveru WWW.

```
package hello;
import java.awt.*;
import java.rmi.*;
public class HelloApplet extends java.applet.Applet {
   String message = "";
   public void init() {
      try {
         Hello obj = (Hello)Naming.lookup("//"+getCodeBase().getHost()
                           +"/HelloServer");
         message = obj.sayHello("client");
      }
      catch(Exception e) {
         System.out.println(e);
      }
```

```
    }
    public void paint(Graphics g) {
       g.drawString(message,25,50);
    }
```

Spíše jako poznámku si uveďme, že při vyhledávání odkazu na stub je zde nutné uvést jméno počítače, z něhož byl zaveden applet, protože standardní AppletSecurityManager nedovolí přístup k lokálním prostředkům.

Příslušná WWW stránka, která dovolí náš applet zavést a spustit může mít například tvar:

```
<HTML>
<title>Hello World</title>
<center><h1>Hello World</h1></center>
The message from the HelloServer is:
<p>
<applet codebase="../.."
   code="examples.hello.HelloApplet"
   width=500 height=120>
</applet>
</HTML>
```

Je zřejmé, že na počítači, na kterém běží server naší aplikace musí být v tomto případě funkční HTTP server, který poskytne uvedenou WWW stránku.

Pokud jde o vytvoření aplikace, prvním krokem je překlad potřebných zdrojových textů. V našem případě, kdy jsou zdrojové soubory *Hello.java*, *HelloImpl.java* a *HelloClient.java* uloženy v podadresáři *Hello* aktuálního adresáře, bude mít volání překladače tvar:

```
javac Hello.java hello/HelloImpl.java hello/HelloClient.java
```

Uvedený příkaz předpokládá zpřístupnění překladače *javac* v systémové proměnné *PATH* (to platí i pro další binární soubory jako jsou *java*, *rmic* a *registry*). Překládané soubory máme v podadresáři *hello* adresáře aktuálního. Knihovny překladače musí být zpřístupněné systémovou proměnnou *CLASSPATH*, případně lze cestu k nim zadat explicitně parametrem *-classpath*. Pokud chceme přeložené soubory aplikace ukládat do jiného adresáře, než kde jsou soubory zdrojové, musíme takový adresář specifikovat parametrem *-d*.

Po překladu musíme vygenerovat soubory *HelloImpl_Stub.class* a *HelloImpl_Skel.class*. Příslušný příkaz pro překladač *rmic* může mít tvar:

```
rmic -d . -keepgenerated hello.HelloImpl
```

Parametrem *-d* opět můžeme definovat adresář, kam budou uloženy generované soubory, parametrem *-keepgenerated* žádáme i o vytvoření jejich textové formy (*HelloImpl_Stub.java* a *HelloImpl_Skel.java*). Textovou formu vygenerovaného stubu a skeletonu se na závěr našeho příkladu uvedeme.

V dalším kroku se musíme případně postarat o uložení vytvořených *bytekódů* - souborů *.class* do adresářů na počítači klienta i serveru (pokud je tam neuložil přímo překladač *javac* a generátor *rmic*). Před spuštěním serveru aplikace musí na tomto počítači běžet aplikace *registry*, její instalaci na portu 2001 lze zajistit příkazem

```
registry 2001 &
```

Při neuvedení čísla portu použije *registry* implicitní port *1099*. Potom již pouze stačí spustit server příkazem:

```
java -cs hello.HelloImpl
```

a klienta příkazem:

```
java -cs hello.HelloClient
```

Slíbili jsme si, že uvedeme textové tvary generovaného stubu a skeletonu. Nejprve se podívejme na stub *HelloImpl_Stub.java*:

```
// Stub class generated by rmic, do not edit.
// Contents subject to change without notice.

package hello;

public final class HelloImpl_Stub
    extends java.rmi.server.RemoteStub
    implements hello.Hello, java.rmi.Remote
{
    private static java.rmi.server.Operation[] operations = {
        new java.rmi.server.Operation
            ("java.lang.String sayHello(java.lang.String)")
    };

    private static final long interfaceHash = 4088309404432474598L;

    // Constructors
    public HelloImpl_Stub() {
        super();
    }
    public HelloImpl_Stub(java.rmi.server.RemoteRef rep) {
        super(rep);
    }
    // Methods from remote interfaces

    // Implementation of sayHello
    public java.lang.String sayHello(java.lang.String $_String_1)
        throws java.rmi.RemoteException {
        int opnum = 0;
        java.rmi.server.RemoteRef sub = ref;
        java.rmi.server.RemoteCall call = sub.newCall
            ((java.rmi.server.RemoteObject)this, operations, opnum, interfaceHash);
        try {
            java.io.ObjectOutput out = call.getOutputStream();
            out.writeObject($_String_1);
        } catch (java.io.IOException ex) {
            throw new java.rmi.MarshalException("Error marshaling arguments", ex);
        };
        try {
            sub.invoke(call);
        } catch (java.rmi.RemoteException ex) {
            throw ex;
```

```
        } catch (java.lang.Exception ex) {
            throw new java.rmi.UnexpectedException("Unexpected exception", ex);
        };
        java.lang.String $result;
        try {
            java.io.ObjectInput in = call.getInputStream();
            $result = (java.lang.String)in.readObject();
        } catch (java.io.IOException ex) {
            throw new java.rmi.UnmarshalException("Error unmarshaling return", ex);
        } catch (java.lang.ClassNotFoundException ex) {
            throw new java.rmi.UnmarshalException("Return class not found", ex);
        } catch (Exception ex) {
            throw new java.rmi.UnexpectedException("Unexpected exception", ex);
        } finally {
            sub.done(call);
        }
        return $result;
    }
}
```

Generovaný stub se opírá o třídu *java.rmi.server.RemoteStub*, která podporuje základní funkce volání metod vzdálených objektů. Doplněn je seznam konkrétních realizovaných metod *operations* a identifikační kód *interfaceHash*.

Pro jednotlivé metody stubu jsou vytvářeny odkazy třídy *RemoteRef* (*sub*) zpřístupňující vzdálený objekt, proměnná *call* se stará o volání vzdálené procedury, vlastní volání realizuje metoda *sub.invoke*. Výsledek volání je zpřístupněn proudem *in* v proměnné *$result*. Metoda *sub.done* volání ukončuje.

Skeleton, který vygeneroval program *rmic* do souboru HelloImpl_Skel.java má tvar:

```
// Skeleton class generated by rmic, do not edit.
// Contents subject to change without notice.

package hello;


public final class HelloImpl_Skel
    extends java.lang.Object
    implements java.rmi.server.Skeleton
{
    private static java.rmi.server.Operation[] operations = {
        new java.rmi.server.Operation("java.lang.String sayHello(java.lang.String)")
    };

    private static final long interfaceHash = 4088309404432474598L;

    public java.rmi.server.Operation[] getOperations() {
        return operations;
    }

    public void dispatch(java.rmi.Remote obj,
                java.rmi.server.RemoteCall call, int opnum, long hash)
```

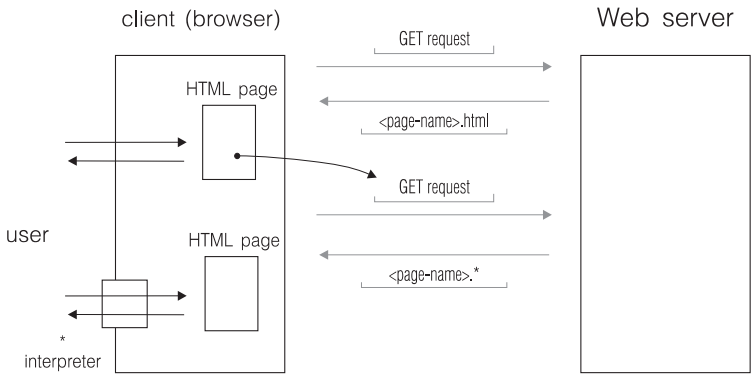
```
            throws java.rmi.RemoteException, Exception {
        // Exceptions pass through, to be caught, identified and marshalled

        if (hash != interfaceHash)
           throw new java.rmi.server.SkeletonMismatchException("Hash mismatch");
        hello.HelloImpl server = (hello.HelloImpl)obj;
        switch (opnum) {
        case 0: { // sayHello
           java.lang.String $_String_1;
           try {
              java.io.ObjectInput in = call.getInputStream();
              $_String_1 = (java.lang.String)in.readObject();
           } catch (java.io.IOException ex) {
              throw new java.rmi.UnmarshalException("Error unmarshaling args", ex);
           } finally {
              call.releaseInputStream();
           };
           java.lang.String $result = server.sayHello($_String_1);
           try {
              java.io.ObjectOutput out = call.getResultStream(true);
              out.writeObject($result);
           } catch (java.io.IOException ex) {
              throw new java.rmi.MarshalException("Error marshaling return", ex);
           };
           break;
           }
        default:
           throw new java.rmi.RemoteException("Method number out of range");
        }
     }
  }
```

Generovaný skeleton je objekt implementující rozhraní *java.rmi.server.Skeleton*, které podporuje základní funkce spojené s voláním metod vzdáleného objektu. Podobně jako u aplikačního stubu je doplněn seznam konkrétních realizovaných metod *operations* a identifikační kód *interfaceHash*. Navíc zde najdeme definici metody *getOperations*. Jádrem skeletonu je metoda dispatch, která po otestování identifikačního kódu *hash*, adresuje odpovídající vzdálený objekt *obj*, vybírá příslušnou metodu a volá ji lokálně. Pro výsledek je vytvořen proud *out*, zápisem výsledku obsluha volání končí.

6. Komunikace v systému WWW

6.1 Stránky HTML, protokol HTTP

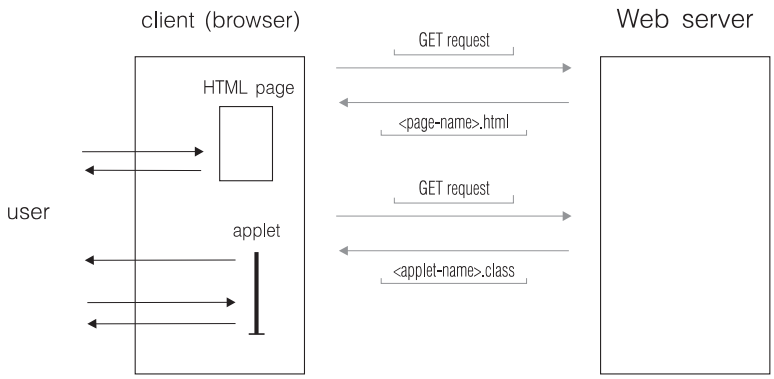
Současné systémy přístupu k informacím se opírají o přenos vyžádaných stránek HTML (HTML - HyperText Markup Language) ze *serveru* WWW (WWW - World Wide Web) na *prohlížeč* WWW (browser). Obsahem těchto stránek je textová, grafická, ale i zvuková informace, přenos stránek zajišťuje protokol HTTP (HyperText Transfer Protocol). Základní formu spolupráce prohlížeče se serverem uvádí obr. 6.1.



Obr. 6.1: Komunikace prohlížeče a serveru WWW

Prohlížeč ve své žádosti identifikuje požadovanou stránku HTML pomocí URL odkazu (URL - Universal Resource Locator), odpovědí serveru je text stránky ve vhodném formátu. Náš příklad uvádí vedle předávání *statických stránek* zapsaných v jazyce HTML, i předávání předem připravených dat v jiných formátech (např. gif, pdf, waw). Přenesená data musí být uživateli zpřístupněna přídavným programem.

Velice zajímavou možností je předávání kódu, který je na straně prohlížeče následné interpretován. Problémem, kterému se lze vyhnout interpretací zdrojového tvaru, jsou rozdílné strojové kódy různých procesorů. Plná interpretace je ale časové náročná a přináší i bezpečnostní rizika. Rozumným kompromisem je překlad zdrojového textu do mezijazyka interpretovatelného na různých procesorech, nebo přeložitelného do jejich kódu. Tato myšlenka se stala základem pro technologii využívající portability mezijazyka (*bytecode*) jazyka Java. Základním kamenem této technologie je *applet* - fragment programu zavedený ze serveru WWW a spuštěný na prohlížeči (obr. 6.2).



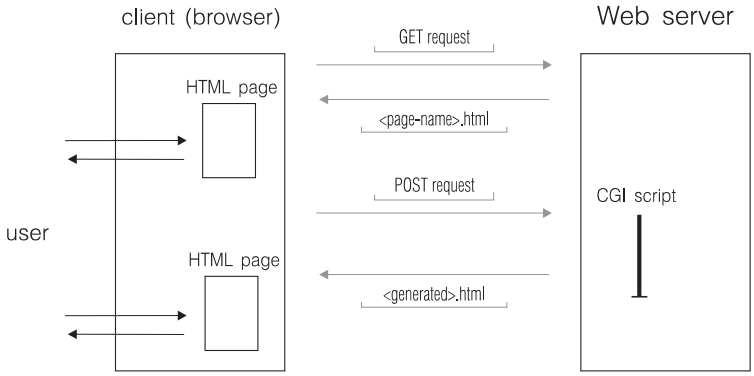
Obr. 6.2: Zavedení appletu ze serveru WWW

Jedinou podmínkou pro využití technologie appletů je doplnění prohlížeče o interpret mezijazyka Javy (JVM - Java Virtual Machine). Výhody jazyka Java v této oblasti jsou výrazné, alternativní technologie jsou buď strojově závislé (ActiveX) nebo pomalejší (JavaScript, Tcl/Tk).

applety dovolují i velice složitým způsobem presentovat na straně prohlížeče data získaná ze serverů. Dovolují vytvořit i velice kvalitní uživatelské rozhraní optimálně využívající možností klientského zařízení a nepřetěžující při tom komunikační prostředky.

6.2 Skripty CGI

Další technologie, které jsou důležité hlavně pro výstavbu podnikových informačních systémů, dovolují vytvářet stránky HTML nebo data v jiných formátech na serverech WWW *dynamicky* (obr. 6.3).

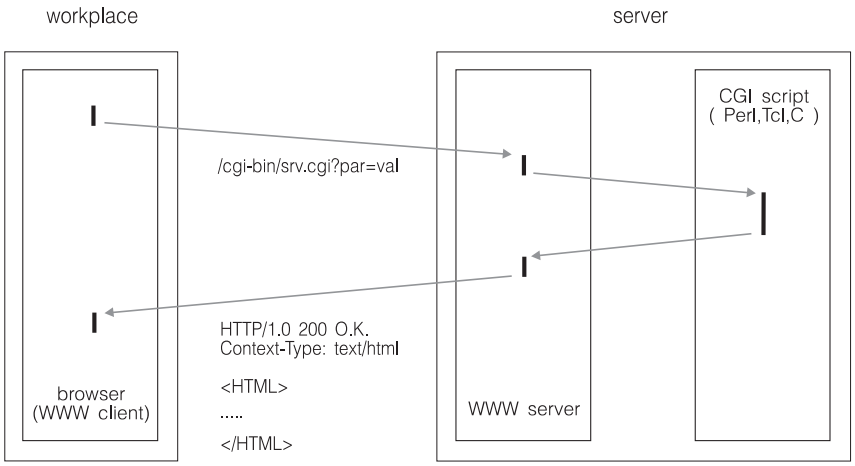


Obr. 6.3: Generování stránek HTML skriptem CGI

Nejčastěji v současné době používanou technikou, dovolující dynamické vytváření stránek HTML a souborů dat určených pro prohlížeče HTML, je technika *skriptů CGI* (Common Gateway Interface). Princip této techniky je jednoduchý: požadavek prohlížeče určuje namísto stránky HTML nebo souboru program, který má být na serveru spuštěn. Spuštěnému programu jsou na standardním vstupu předány i případné parametry. Standardní výstup programu vytváří data, která jsou následně přenesena na prohlížeč.

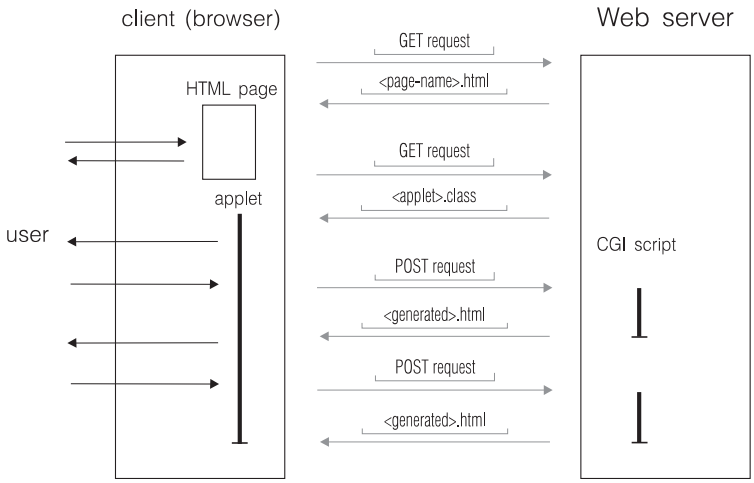
Žádosti prohlížeče mají formát HTTP příkazů, odpovědí jsou data ve formátu MIME, který dovoluje jejich přenos jako textů (obr. 6.4).

Technika skriptů CGI se opírá o programy, schopné přijímat data na standardním vstupu a generovat data na standardním výstupu. Takové programy lze vytvářet prakticky v libovolném jazyce, nejčastěji využívanými jsou skriptový jazyk Perl a jazyk C.



Obr. 6.4: Spolupráce prohlížeče se skriptem CGI

Kombinace obou uvedených technologií, tedy appletů na straně prohlížeče a skriptu CGI na straně serveru dovoluje spojit výhody aktivity na straně prohlížeče (efektivní presentace) s výhodami aktivity na straně serveru (dynamicky vytvářená data). Možnou komunikaci appletu se skripty CGI ilustruje obr. 6.5.



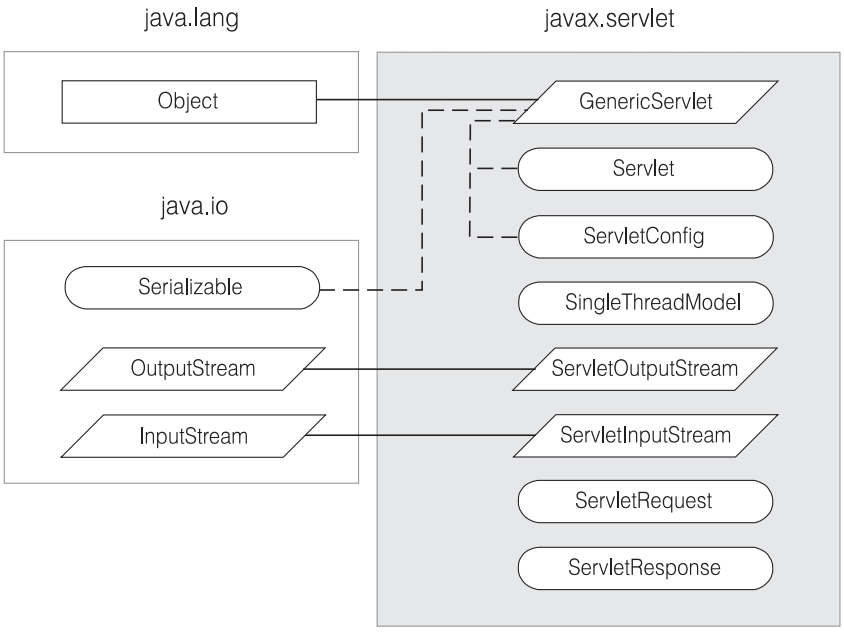
Obr. 6.5: Komunikace appletu se skriptem CGI

Spojení technologie appletů se skripty CGI sice dovoluje realizovat i složité činnosti, určitou nevýhodou je však nízká efektivita spouštění a případné interpretace skriptů CGI. Podstatného zlepšení lze dosáhnout technologií servletů, kterým se budeme věnovat v následující části.

6.3 Servlety - aplikační komponenty serveru WWW

Při programování aplikací realizujících CGI si sice můžeme zvolit programovací jazyk, nevýhodou mechanismu je však malá efektivita a chybějící programátorská podpora. Realizace funkcí CGI objekty v jazyce Java spouštěnými v prostředí serveru WWW zlepšuje ve srovnání s klasickým přístupem efektivitu a díky podpoře standardizovaných knihoven i zjednodušuje programování. Objekty jazyka Java, které jsou spouštěné na serveru WWW a dovolují podpořit prohlížeče způsobem obdobným rozhraní CGI, známe pod jménem *servlety*.

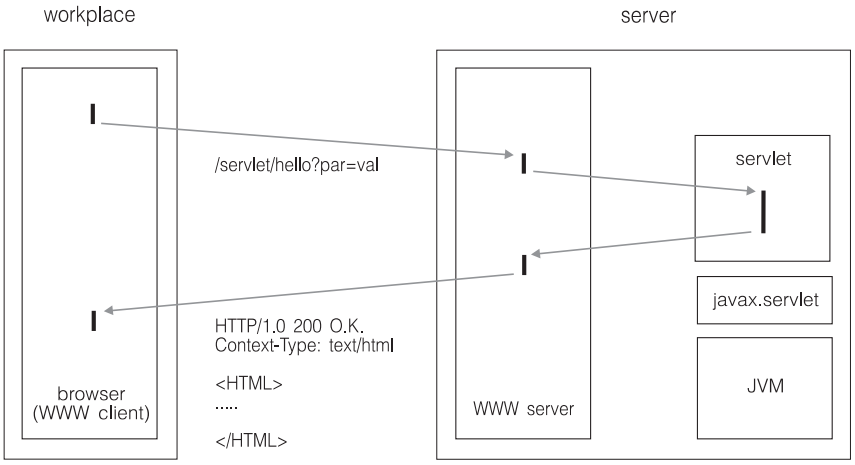
Servlet je objekt jazyka Java, schopný spolupráce se serverem WWW. Taková spolupráce je podpořena knihovnou *javax.servlet* (obr. 6.6), její využití na serveru WWW se opírá o JVM podobně, jako se o JVM opírá applet v prohlížeči.



Obr. 6.6: Knihovna *javax.servlet*

Knihovnu tvoří vedle definic rozhraní *Servlet*, *ServletConfig* a rozhraní a tříd, která tato základní rozhraní podporují, ještě prototypová implementace servletu *GenericServlet*.

Servlet se svou činností podobá skriptu CGI, způsob komunikace prohlížeče s aplikací spouštěnou na WWW serveru ilustruje obr. 6.7. Servlet umí převzít případné parametry příkazu (potřebné metody zpřístupňuje rozhraní *ServletRequest*) a vytvořit správně formátovanou odpověď (potřebné metody zpřístupňuje rozhraní *ServletResponse*). Na rozdíl od skriptu CGI je servlet instalován na delší dobu a může zodpovědět víc dotazů, jak sekvenčních, tak i souběžných.



Obr. 6.7: Komunikace prohlížeče se servletem

Každý servlet implementuje rozhraní *Servlet* knihovny *javax.servlet*. Rozhraní má poměrně jednoduchou signaturu:

```
public interface Servlet {
   public abstract void init(ServletConfig config) throws ServletException;
   public abstract ServletConfig getServletConfig();
   public abstract void service(ServletRequest req,ServletResponse res)
         throws ServletException, IOException;
   public abstract String getServletInfo();
   public abstract void destroy();
}
```

Metoda *init* je spuštěna při instalaci (při prvním vyvolání) servletu, metoda *destroy* je spuštěna před jeho ukončením. Ukončení může být explicitně vyžádáno klientem (v kódu metody *service*) nebo serverem WWW při uzavírání práce). Jednoduché servlety mohou vystačit s implicitní definicí těchto metod, v reálných aplikacích můžeme v metodě *init* například navázat déle trvající spojení s databází využívanou více dotazy.

Metoda *getServletConfig* zpřístupňuje parametry předávané servletu při jeho instalaci. Definice parametrů je spolu s vazbou mezi jménem, pod nímž je servlet znám klientům, a jménem, pod nímž je definován, uvedena v souboru *servlets properties*. Metoda *getServletInfo* dovoluje identifikovat servlet pro potřeby správy serveru WWW.

Vlastní chování servletu definuje metoda *service*. Ta je spouštěna při každém vyvolání klientem, současné vyvolání metody *service* více klienty vede na souběžný výpočet několika vláken v prostředí servletu. Vzájemnou synchronizaci vláken (například při sdílení dat nebo jiných prostředků) musíme zajistit explicitně. V jazyce Java máme pro tento účel k dispozici synchronizované objekty a synchronizované metody.

Nevyžadujeme-li spolupráci mezi jednotlivými voláními a chceme-li se vyhnout komplikacím spojeným s explicitní synchronizací, můžeme servlet doplnit o rozhraní *SingleThreadModel*.

```
public interface SingleThreadModel {
}
```

Uvedení rozhraní *SingleThreadModel* mezi rozhraními servletu způsobí, že souběžná volání budou používat nezávislé kopie dat servletu, výsledná obsluha je však podstatně pomalejší.

U většiny servletů jsou instalace i ukončení obdobné a navíc je potřeba, aby servlety respektovaly řadu dalších pravidel dovolujících jejich začlenění do práce serveru. Toho lze dosáhnout přidržíme-li se při programování servletů prototypu definovaného třídou *GenericServlet*. Tato třída byla vytvořena pro servlety komunikující protokolem HTTP a má signaturu:

```
public abstract class GenericServlet
          extends Object implements Servlet, ServletConfig, Serializable {
   public GenericServlet();
   public ServletContext getServletContext();
   public String getInitParameter(String name);
   public Enumeration getInitParameterNames();
   public void log(String msg);
   public String getServletInfo();
   public void init(ServletConfig config) throws ServletException;
   public ServletConfig getServletConfig();
   public abstract void service(ServletRequest req,ServletResponse res)
          throws ServletException, IOException;
   public void destroy()
}
```

Kromě implicitní implementace metod rozhraní *Servlet* a *ServletConfig* zpřístupňuje *GenericServlet* kontext servletu. Kontextem se zde rozumí skupina servletů, které mohou během své činnosti vzájemně komunikovat voláním svých metod. Poslední implementovanou metodou je *log*, ta podporuje záznam poznámek o práci servletu do příslušného souboru.

Vytvoření servletu, kterému postačuje komunikace protokolem HTTP, pak pouze vyžaduje v rozšíření třídy *GenericServlet* definovat chování při příchodu požadavku (metoda *service*) a případně definovat chování při spouštění a ukončování práce (metody *init* a *destroy*). Jako příklad si uvedeme velice jednoduchý servlet reagující na volání bez parametrů vygenerováním textového řetězu "Hello Client!":

```
package hello;
import javax.io.*;
import javax.servlet.*;
public class HelloClientServlet extends GenericServlet {
   public void service(javax.servlet.ServletRequest req,
         javax.servlet.ServletResponse res) {
      try {
         PrintStream out = new PrintStream (res.getOutputStream());
         out.println("Content-Type: text/plain");
         out.println("");
         out.println("Hello Client!");
         out.close();
      }
      catch(java.io.Exception e) {
         System.out.println(e);
      }
   }
}
```

Příjem požadavků servletem a vytváření odpovědí je podporováno rozhraními *ServletRequest* a *ServletResponse*, zde si uvedeme pouze jejich zjednodušenou signaturu:

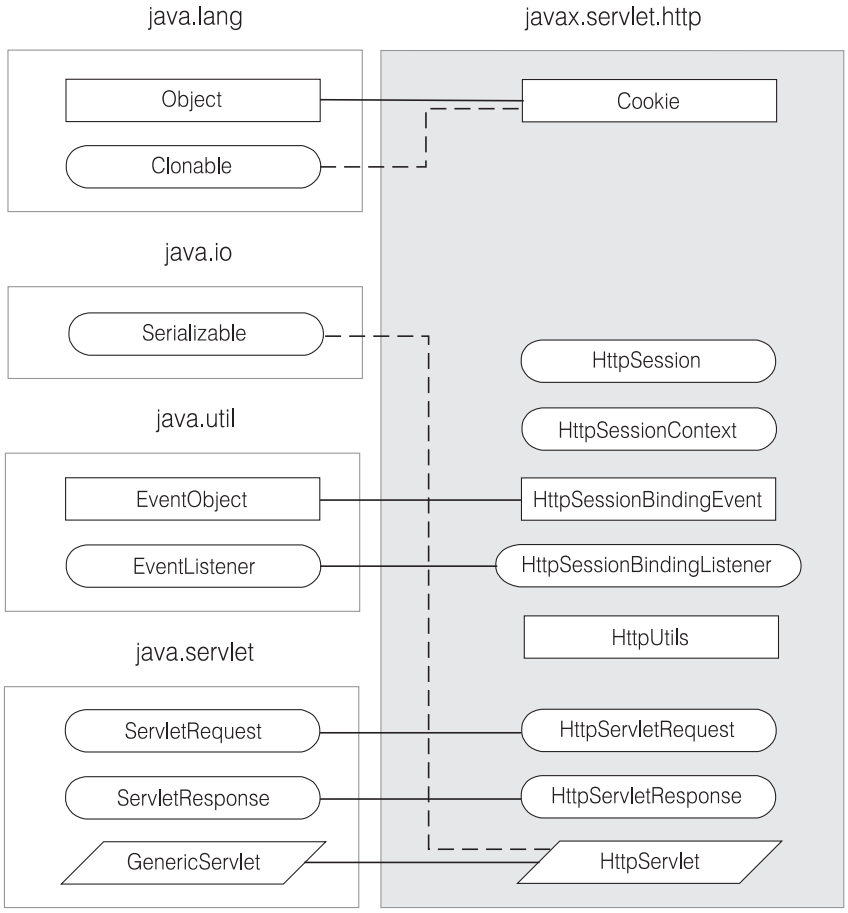
```
public interface ServletRequest {
   public abstract int getContentLength();
   public abstract String getContentType();
   ...
   public abstract ServletInputStream getInputStream() throws IOException;
   public abstract BufferedReader getReader() throws IOException;
   public abstract String getCharacterEncoding();
   }
```

Rozhraní *ServletRequest* dovoluje předat požadavek ve formě vstupního proudu typu *ServletInputStream* (pro binární data) nebo *BufferedReader* (pro textová data). Metody *getContentLength* a *getContentType* dovolují získat informaci o formátu dotazu (jak ho definuje MIME), metoda *getCharacterEncoding* dovoluje zjistit použité kódování. Další zde neuvedené metody informují o síťové adrese klienta a serveru (IP adresa, doménové jméno, číslo portu), o použitém protokolu (HTTP, HTTPS, FTP, ...).

```
public interface ServletResponse {
   public abstract void setContentLength(int len);
   public abstract void setContentType(String type);
   public abstract ServletOutputStream getOutputStream() throws IOException;
   public abstract PrintWriter getWriter() throws IOException;
   public abstract String getCharacterEncoding();
   }
```

Rozhraní *ServletResponse* dovoluje předat odpověď ve formě výstupního proudu typu *ServletOutputStream* (pro binární data) nebo *PrintWriter* (pro textová data). Metody *setContentLength* a *setContentType* dovolují definovat formát dotazu (implicitním formátem je "text/plain"), metoda *getCharacterEncoding* dovoluje zjistit kódování použité při daném formátu výstupu.

Uvedený způsob komunikace sice dovoluje vytvářet servlety schopné spolupracovat s prohlížeči, dalšího zjednodušení přípravy aplikací komunikujících protokolem HTTP však lze dosáhnout použitím standardizovaného rozhraní *javax.servlet.http.HtttServlet* podporovaného knihovnou *javax.servlet.http* (obr. 6.8).



Obr. 6.8: Knihovna *javax.servlet.http*

Knihovnu tvoří prototypový servlet *HttpServlet* a podpůrná rozhraní *HttpServletRequest* a *HttpServletResponse*. Doplněna jsou rozhraní a třídy podporující správu relací: relací se rozumí posloupnost operací mezi jedním klientem a servletem. Identifikace klienta v bezestavovém protokolu HTTP se opírá o mechanismus výměny *cookies* (proměnných identifikovaných jménem a nabývajících jako hodnot textových řetězců) mezi servletem a klientem. Pokud použití cookies klient nepovoluje, je k dispozici alternativní mechanismus dynamické modifikace URL odkazů.

Rozhraní *HtttServlet* knihovny *javax.servlet.http* má následující signaturu:

```
public abstract class HttpServlet
          extends GenericServlet implements Serializable {
   public HttpServlet();
   protected void doGet(HttpServletRequest req,HttpServletResponse resp)
          throws ServletException, IOException;
   protected long getLastModified(HttpServletRequest req);
   protected void doPost(HttpServletRequest req,HttpServletResponse resp)
          throws ServletException, IOException;
   protected void doPut(HttpServletRequest req,HttpServletResponse resp)
          throws ServletException, IOException;
   protected void doDelete(HttpServletRequest req,HttpServletResponse resp)
          throws ServletException, IOException;
   protected void doOptions(HttpServletRequest req,HttpServletResponse resp)
          throws ServletException, IOException;
   protected void doTrace(HttpServletRequest req,HttpServletResponse resp)
          throws ServletException, IOException;
   protected void service(HttpServletRequest req,HttpServletResponse resp)
          throws ServletException, IOException;
   public void service(ServletRequest req,HttpServletResponse resp)
          throws ServletException, IOException;
}
```

Třída *HttpServlet* dovoluje rozdělit přicházející přikazy HTTP (GET, POST, ...) a obsloužit je odpovídajícími metodami (*doGet*, *doPost* , ...). Řada z těchto metod má implicitní chování, u nejjednodušších aplikací postačí definovat metody *doGet* a případně *doPost*.

Jako příklad komunikace, která využívá pouze příkazu GET si uvedeme, podobně jako u třídy *GenericServlet*, servlet *HelloClientServlet*:

```
package hello;
import javax.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class HelloClientServlet extends HttpServlet {
   public void doGet(HttpServletRequest req,HttpServletResponse res)
              throws ServletException, IOException {
          res.setContentType("text/html");)
          PrintWriter out = res.getWriter();
          out.println("<HTML><HEAD><TITLE>Hello Client!</TITLE>"+
                "</HEAD><BODY>Hello Client!</BODY></HTML>");
          out.close();
   }
   public String getServletInfo() {
      return "HelloClientServlet";
   }
}
```

Náš servlet pouze definuje formát výstupních dat ("text/html") a generuje vlastní stránku HTML. Metoda *getServletInfo* definuje řetězec, pod kterým se servlet identifikuje správě serveru.

Složitější situace nastává u příkazu POST, u kterého analyzujeme text požadavku a podle jeho obsahu generujeme odpověď. Jednoduchým příkladem může být opět *HelloClientServlet*, který se ale nejdříve dotáže na identitu klienta (zadávanou jako textový řetězec) a následně vygeneruje stránku s odpovědí:

```
package hello;
import javax.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class HelloClientServlet extends HttpServlet {
   public void doGet(HttpServletRequest req,HttpServletResponse res)
         throws ServletException, IOException {
      res.setContentType("text/html");)
      res.setHeader("pragma","no-cache");
      PrintWriter out = res.getWriter();
      out.print("<HTML><HEAD><TITLE>Hello Client</TITLE></HEAD>");
      out.print("<BODY><H3>HelloClientServlet</H3><HR>");
      out.print("<FORM METHOD=POST>");
      out.print("Enter your name, please: <INPUT TYPE=TEXT NAME=id><BR>")
      out.print("<INPUT TYPE=SUBMIT NAME=action VALUE=send>")
      out.print("</FORM></BODY></HTML>");
      out.close();
   }
   public void doPost(HttpServletRequest req,HttpServletResponse res)
         throws ServletException, IOException {
      String name = req.getParameter("id");
      res.setContentType("text/html");)
      res.setHeader("pragma","no-cache");
      PrintWriter out = res.getWriter();
      out.print("<HTML><HEAD><TITLE>Hello Client</TITLE></HEAD><BODY>");
      if (name == null)
         out.print("Hello anonymous client");
      else
         out.print("Hello "+name);
      out.print("!</BODY></HTML>");
      out.close();
   }
   public String getServletInfo() \{
      return "HelloClientServlet";
   }
}
```

V odpovědi na příkaz GET je generována stránka HTML obsahující dotaz, prohlížeč po vyplnění identifikačního řetězce <*name*> odešle příkaz POST, na který reaguje náš servlet vytvořením stránky s pozdravem "Hello <*name*>!" (případně "Hello anonymous client!").

7. Závěr

Náš text je průvodcem technologiemi, podporujícími distribuovaný výpočet, využívanými v praxi počátkem roku 1999. Jeho hlavním cílem bylo uvést jednoduché příklady, usnadňující vytváření distribuovaných komponent, využívající jak běžné BSD sockety, tak prostředky moderní procedurální komunikace.

Vzhledem k uvažovanému využití a rozsahu nebylo možné zahrnout řadu modifikací uváděných technologií (jako např. TLI sockety, WinSocks a DCOM firmy Microsoft). Text si všímá zvláště komunikačních aspektů popisovaných technologií, poněkud stranou zůstávají systémové služby.

Do textu nebyly zahrnuty moderní nástroje podporující mobilitu procesů. Jejich první generace se opírala o interpretované jazyky (Telescript, Tacoma, AgentTcl), modernější řešení využívají portability, kterou přinesla Java (Voyager, Aglets). Dosud neuzavřenou oblastí je integrace objektových rozhraní serverů, opírajících se o technologii CORBA, s portabilními clienty, opírajícími se o technologii Java.

Autoři věří, že text alespoň po určitou dobu podpoří praktická cvičení v předmětu Distribuované systémy, jejichž cílem je seznámit se s problematikou programování distribuovaných aplikací v reálném prostředí.

Literatura a odkazy

[1] Postel J.: *Internet Protocol, RFC 791.* September 1981.

[2] Postel J.: *Internet Control Message Protocol, RFC 792.* September 1981.

[3] Hinden R., Deering S.: *Internet Protocol Version 6 (IPv6) Specification, RFC 1883.* December 1995.

[4] Deering S., Hinden R.: *IP Version 6 Addressing Architecture, RFC 2373.* July 1998.

[5] Deering S., Conta A.: *Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification, RFC 1885.* December 1995.

[6] Postel J.: *User Datagram Protocol, RFC 768.* August 1980.

[7] Postel J.: *Transmission Control Protocol, RFC 793.* September 1981.

[8] Braden R.: *Requirements for Internet Hosts - Communication Layers, RFC 1122.* October 1989.

[9] Deering S.: *Host Extensions for IP Multicasting, RFC 1112.* August 1989.

[10] A. D. Birrel, Nelson B. J.: Implementation remote procedure call. *ACM Trans. Computer Systems*, 2:39–59, 1984.

[11] John Bloomer.: *Power Programming with RPC.* O'Reilly, September 1992.

[12] R. Srinivasan.: *XDR: External Data Representation Standard, RFC 1832.* Sun Microsystems, August 1995.

[13] Fagan B.: *Distributed Computing RPC Volume.* Prentice Hall, 1994.

[14] Shirley J., Wei Hu, Magid D.: *Guide to Writing DCE Applications.* O'Reilly, 1994.

[15] Pope A.: *The CORBA Reference Guide.* Addison-Wesley, 1997.

[16] *The Common Object Request Broker Architecture and Specification.* February 1998.

[17] Flanagan D.: *Java in a Nutshell.* O'Reilly, 1996.

[18] Flanagan D.: *Programování v jazyce Java.* Computer Press, 1997.

[19] Tanenbaum A.S.: *Computer Networks - 3rd ed.* Prentice Hall, 1996.

[20] Braden R.: *Requirements for Internet Hosts - Application and Support, RFC 1123.* October 1989.

[21] Comer D.E.: *Internetworking with TCP/IP. Vol I: Principles, Protocols and Architecture.* Prentice Hall, 1991.

[22] Stevens D.L., Comer D.E.: *Internetworking with TCP/IP. Vol II: Design, Implementation and Internals.* Prentice Hall, 1991.

[23] Stevens D.L., Comer D.E.: *Internetworking with TCP/IP. Vol III: Client-Server Programming and Applications.* Prentice Hall, 1993.

[24] Comer D.E.: *Internet Book.* Prentice Hall, 1995.

[25] Stevens W.R.: *UNIX Network Programming.* Prentice Hall, 1990.

[26] Stevens W.R.: *TCP/IP Illustrated.* Addison-Wesley, 1994.

[27] Rudolf V., Šmrha P.: *Internetworking pomocí TCP/IP.* Kopp, 1994.

[28] Rosenberry W.: *Understading DCE.* O'Reilly, 1992.

[29] Srinivasan R.: *RPC: Remote Procedure Call Protocol Specification Version 2, RFC 1831.* Sun Microsystems, August 1995.

[30] Stevens W.R.: *Programování sítí operačního systému UNIX.* Science, 1994.

Literatura a odkazy

Janecek

[19] [6] [1] [2] [7] [9] [8] [20] [5] [4] [21] [22] [23] [24] [25] [26] [27] [15] [16] [17] [18]

Kubr

[13] [14] [28] [11] [29] [12] [30]