BSD socket interface

Vladimír Rudolf
dolf@zcu.cz

1 Programová rozhraní

Protokoly *TCP/IP* jsou popsány a definovány v dobře známých dokumentech *RFC*. Pro aplikační programy je však nutné definovat i programové rozhraní, které umožní aplikaci využívat a používat protokoly *TCP/IP*. Standardy, které definují komunikační protokoly, nespecifikují, jak aplikační vrstva komunikuje se spodními vrstvami. Je k tomu dobrý důvod, neboť implementace protokolů je úzce spojena s operačním systémem (většinou bývá dokonce jeho součástí) a programové rozhraní je tím pádem závislé na daném operačním systému.

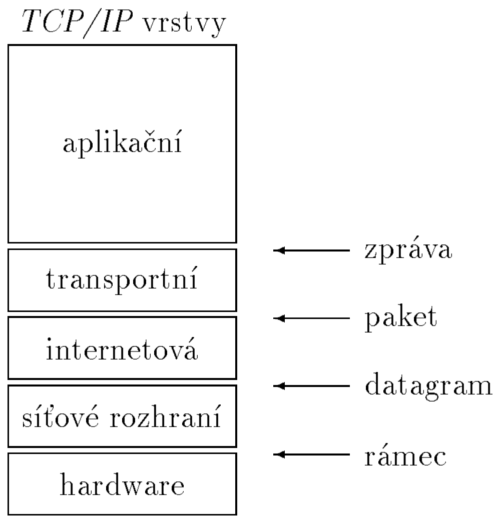
V tomto příspěvku se pokusíme objasnit, jak využívat služby nabízené rodinou protokolů *TCP/IP* a provést základní klasifikaci jednotlivých aplikací a přístupů.

Pro přiblížení problematiky programového rozhraní a využívání nabízených služeb, jsme vybrali programové rozhraní – *sockets*, které je pravděpodobně nejrozšířenějssí a nejpopulárnější.

2 Model klient–server

Protokoly *TCP/IP* spolupracují na bázi vrstveného modelu, který je zobrazen na obr. 1. Odpovídající jednotlivé vrstvy modelu komunikují s odpovídajícími vrstvami svého partnera a využívají přitom služeb, které jim nabízí spodní vrstva modelu. Tomuto způsobu komunikace se říká tzv. *peer-to-peer* komunikace, což značí komunikaci mezi stejnolehlými vrstvami ve vrstveném modelu.

Nás bude zajímat hlavně programové rozhraní mezi aplikační vrstvou a spodními vrstvami. Aplikace, využívající protokoly *TCP/IP*, jsou založeny na modelu *client-server*. Znamená to, že existují aplikace, které poskytují nějaké služby (servery), a ostatní aplikace (klienti) tyto nabízené služby využívají. Tento princip je základem pro aplikace využívající *TCP/IP*. V tomto okamžiku je nutné, abychom zdůraznili i další princip. *TCP/IP* neposkytuje mechanismus na vytvoření procesu při příchodu zprávy. Proto musí existovat mechanismus, který umožní úloze pasivně čekat na příchod zprávy.



Obr. 1: *TCP/IP* vrstvený model internetu.

Z tohoto principu vyplývá, že servery, které nabízejí služby *TCP/IP*, musí být předem spuštěny, nelze je vyvolat na základě požadavků klienta (nenechte se mýlit skutečností, že v operačním systému UNIX se vyvolá např. program `telnetd` až v okamžiku požadavku klienta. Tento požadavek je zpracován programem `inetd`, který musí být aktivní, a ten pak zajistí vyvolání programu `telnetd`).
Z předchozích řádek vyplývá, že aplikace *TCP/IP* se rozpadají na dvě hlavní třídy:

- **klient** – aplikace inicializující komunikaci (obvykle dotaz),
- **server** – aplikace očekávající komunikaci, na jejímž základě vyvolává patřičnou akci (obvykle odpověď).

Jakmile server akceptuje požadavek klienta, je navázáno plně duplexní spojení a procesy mezi sebou komunikují. Je-li aplikace napsána tak, že libovolný z partnerů může zastávat funkci servera i klienta, jedná se o *symetrickou* aplikaci. Je-li předem určeno, kdo je server a kdo klient, hovoříme o *nesymetrické* aplikaci. Typickým představitelem asymetrické aplikace je *ftp*.

2.1 Standardní a nestandardní klient

Standardní klient pracuje s tzv. *well-known port*, což značí předem dohodnuté číslo portu pro standardně zavedené služby. Dále budeme tyto porty nazývat *rezervovanými porty.* Jedná se např. o služby *mail, telnet, ftp* apod. Ostatní software typu klient se považuje za *nestandardní.* Z hlediska uživatele se nejeví v používání žádný podstatný rozdíl.

Rezervované porty jsou zavedeny hlavně z důvodů navázání spojení, kdy klient musí znát číslo portu, na kterém nabízí server určitou službu, např. *telnet.* Po navázání spojení a po dohodě obou partnerů již není třeba využívat rezervovaný port (dokonce to není ani žádoucí) a komunikace probíhá na náhodně vybraných nekonfliktních číslech portů. Standardní rezervovaný port je uvolněn pro komunikaci s příštím potenciálním klientem. Podrobněji si tento mechanismus probereme v dalších kapitolách.

2.2 Úlohy serveru

Server má často privilegia, která jsou nutná pro jeho provoz, ale z důvodů bezpečnosti není vhodné, aby byla běžně přístupná všem uživatelům. Např. server nabízející službu *ftp* musí mít přístup ke všem souborům v systému. Takováto privilegia však zaručeně nechceme dovolit každému uživateli. Musí zde existovat mechanismus, který rozliší práva jednotlivých uživatelů a podle jejich práv jim umožní přístup pouze k povoleným souborům.

Program realizující funkce serveru proto musí obsahovat kód, který zabezpečuje splnění některých nebo všech následujících bodů.

Ověření identity uživatele (podle jména a hesla).

Autorizace uživatele, tj. zda daný uživatel má právo využívat požadovanou službu.

Bezpečnost dat, kde server zaručuje, že jistá data nelze úmyslně či neúmyslně zničit, modifikovat nebo i číst neoprávněnou osobou.

Privátnost dat – server musí zaručit vzájemnou ochranu dat uživatelů.

Ochrana zdrojů – server zabezpečuje systémové zdroje před zneužitím přes komunikační síť.

Implementace těchto požadavků záleží na typu služeb, které server nabízí. Nabízí-li server např. službu, kterou je možné získat pouze aktuální čas, je zřejmé, že nemusíme uvažovat o aplikaci žádného z uvedených požadavků bezpečnosti.

2.3 Spojované versus nespojované služby serverů

Jedno ze základních rozhodnutí, které musí programátor provést, je, zda bude využívat spojované nebo nespojované služby. Jedná se vlastně o to, zda použije protokol *TCP* nebo *UDP*. Podívejme se blíže na rozdíly mezi těmito přístupy.

TCP zajišťuje spolehlivost, tj. kontroluje *checksum*, používá sekvenční čísla ke garanci správného pořadí či eliminaci případné ztráty paketu a řídí tok dat (*flow control*), aby vysílající nevysílal rychleji než je příjemce schopen přijímat.

UDP se naproti tomu pouze snaží o bezchybný přenos (*best effort delivery*). Pakety však mohou být ztraceny, duplikovány, zpožděny nebo přijímány v jiném pořadí.

Je jasné, že programování využitím spojovaných služeb (*TCP*) je pro programátora nepoměrně jednodušší. Proto je většina aplikací napsána s využitím spojovaných služeb, kdy je programátor zbaven netriviální úlohy s ošetřením indikovaných chyb. Zabezpečení spolehlivosti, kterou *TCP* zabezpečuje, však přináší i jistou režii zpracování.

Nespojované služby oproti tomu nabízí rychlejší dobu odezvy, možnosti použití *broadcast* a *multicast* (zpráva určená více adresátům, ne však všem). U nespojovaných služeb pracují aplikace ve spolehlivých lokálních sítích většinou bez problémů. Programátoři často odladí úlohu využívající nespojované služby na lokální síti, kde je provoz téměř bezchybný, avšak nasazení do prostředí *internetu* je pro ně zklamání. Je proto nutné ošetřit veškeré možné chyby, pokud chceme vytvořit spolehlivou aplikaci schopnou nasazení i do prostředí rozsáhlých méně spolehlivých sítí.

Protokol *UDP* používáme obvykle pouze tehdy, je-li nutné splnit některý z následujících požadavků.

- aplikační protokol vyžaduje přímo použití *UDP*,
- aplikační protokol je založen na používání *broadcast* a *multicast*,
- potřebujeme odstranit časovou náročnost zpracování (*overhead*) vrstvy *TCP*.

2.4 Stavové a bezstavové servery

Informaci, která je udržována o spojení klient–server, budeme nazývat *stavová informace*. Jestliže server neudržuje v paměti žádnou stavovou informaci, budeme jej nazývat *bezstavový*, v opačném případě *stavový*. Stavová informace může zefektivnit aplikační protokol a redukovat množství předávaných dat.

Jako příklad si představme aplikaci, která klientovi umožňuje číst resp. zapisovat do souborů na serveru. Při bezstavovém serveru je nutné, aby každý požadavek sebou nesl i jméno souboru, pozici, odkud se mají data číst a počet požadovaných bytů. Stavový server si naopak může spojit tyto informace s danou relací a v následujících požadavcích již není třeba specifikovat některé údaje, jako např. jméno souboru. Je zřejmé, že stavový server pracuje v ideálním prostředí efektivněji. Bohužel ideální prostředí není k dispozici. Proto se musíme vypořádat s mnoha problémy. Patří mezi ně řada asynchronních událostí, které mohou během komunikace nastat, jako jsou *reboot* serveru, nestandardní ukončení relace klientem (jak dlouho mám držet stavovou informaci?), duplikace požadavků (problémové při sekvenčním přístupu) apod.

Navrhnout stavový server v prostředí internetu je úloha komplexní a složitá. Navrhnout bezstavový server je jednodušší, aplikační protokol musí však dovolovat použití takovéhoto serveru. Znamená to např., že protokol nesmí specifikovat obsah nějaké zprávy v závislosti na předchozí, že stejný požadavek vždy vyvolá stejnou odezvu apod.

2.5 Prostředky pro konkurentní zpracování

Jestliže předpokládáme, že server má sloužit více klientům najednou, je třeba navrhnout filosofii serveru tak, aby umožňoval současnou obsluhu více klientů, ať už skutečným paralelním zpracováním (více procesorů) nebo jinými prostředky, např. sdílením času.

V operačním systému UNIX jsou standardně k dispozici prostředky dovolující vytváření samostatných procesů, což je ideální prostředek pro současnou obsluhu více klientů. Je to především systémová funkce `fork()`, která rozdělí běžící program na dva téměř identické procesy. Další vhodné prostředky jsou volání `execve()`, které dovolí změnit (přepsat) kód procesu jiným obrazem a volání `select()`, které slouží pro obsluhu několika současných I/O aktivit.

Pomocí těchto prostředků je možné vytvořit pro každé spojení (dvojice klient a server) samostatný proces a o současnou obsluhu požadavků těchto procesů se postará sám operační systém. Tak je naprogramována většina používaných serverů, např. *telnet, ftp* apod.

2.6 Programové rozhraní k *TCP/IP*

TCP/IP je na většině systémů součástí operačního systému. Aplikační programy komunikují s jádrem a předávají mu své požadavky a získávají odpovědi.

Komunikační protokoly *TCP/IP* jsou definovány poměrně volně a ve standardu nejsou specifikovány detaily softwarového rozhraní.

To má samozřejmě své výhody i nevýhody. Výhodou je flexibilita umožňující implementaci *TCP/IP* na počítačích od PC po superpočítače. Nevýhodou je ztráta přenositelnosti programů, neboť je výběr programového rozhraní ponechán na aplikačním programátorovi.

Každé programové rozhraní k *TCP/IP* musí splňovat jisté základní požadavky:

- alokace lokálních zdrojů nutných pro komunikaci,
- specifikace lokálního a vzdáleného komunikačního uzlu,
- inicializace spojení (klient),
- čekání na požadavek spojení (server),
- čtení a zápis dat,
- indikace příchozích dat,
- vytváření a zpracování urgentních zpráv,
- ukončení komunikace,
- zpracování ukončení komunikace od vzdáleného uzlu,
- násilné přerušení komunikace,

- zpracování chybových stavů,
- uvolnění alokovaných zdrojů.

Mezi nejvíce používaná programová rozhraní patří rozhraní definované univerzitou v Berkeley pro BSD UNIX známé jako *socket interface*, *sockets* a rozhraní vyvinuté v AT&T pro System V známé jako *TLI*(*Transport Layer Interface*).

3 Sockets interface

Jedná se o *de facto* standard. Autoři si vybrali obecný přístup, který lze použít nejen protokoly *TCP/IP*, ale i v jiných případech. Tento obecný přístup má své výhody, především jednotný přístup při řešení více problémů, na druhé straně musejí uživatelé tohoto rozhraní specifikovat více parametrů, kterými určují konkrétní použití *sockets*. V dalším textu se budeme zabývat použitím programového rozhraní pouze pro *TCP/IP* protokoly.

3.1 Socket

Socket je definován jako koncový komunikační uzel (*communication endpoint*). Jedná se o abstrakci, vlastně datovou strukturu, ve které jsou definovány potřebné údaje pro komunikaci.

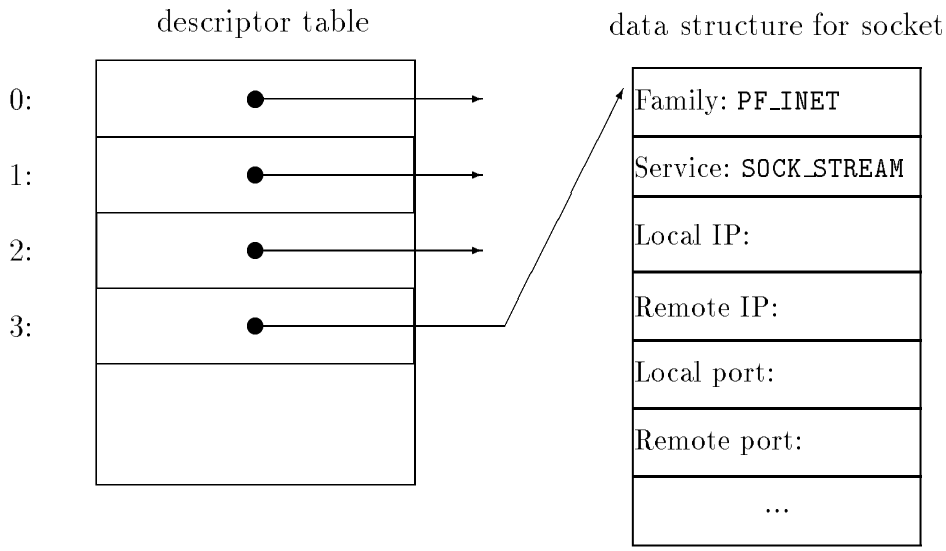
Pro protokoly *TCP/IP* je socket plně identifikován IP adresou a číslem protokolového portu. Socket nese dostatečnou informaci, která přesně identifikuje ať už příjemce nebo odesilatele zprávy. IP adresa určuje konkrétní stroj, konkrétní počítač a číslo portu určuje aplikaci, pro kterou je zpráva určena.

3.2 Deskriptor socketu

Deskriptor souboru je znám v operačním systému UNIX v souvislosti s řádnými soubory, kde funkce `open()` vrací index do tabulky deskriptorů souborů, v níž jsou uloženy potřebné informace pro práci s daným souborem. Aplikace pracuje nadále pouze s deskriptorem souboru, tj. s indexem do tabulky deskriptorů. Sockety využívají shodnou tabulku a shodnou filosofii. Znamená to, že se v systému vytvoří datová struktura (po volání funkce `socket()`), nesoucí informace potřebné pro komunikaci a ukazatel na tuto strukturu se uloží do tabulky deskriptorů. Aplikaci je vrácen index do této tabulky, tzv. *socket deskriptor*. Je využívána stejná tabulka deskriptorů, nelze tedy mít deskriptor souboru a deskriptor socketu se stejnou hodnotou.

Při vytváření socketu nejsou vyplněny všechny položky, je vytvořena pouze základní struktura zaplněná základními údaji. Ta se postupně vyplňuje podle dalších požadavků aplikace.

Takto vytvořený socket může být použit buď jako *pasivní*, tj. čekající na aktivizaci příchozí zprávou nebo jako *aktivní*, tj. socket je použit pro inicializaci spojení. Pasivní



Obr. 2: Tabulka descriptorů pouze se standardními soubory a jedním otevřeným socketem.

sockety jsou používány většinou v serverech, zatímco aktivní používají jak klienti tak i servery pro již navázaná spojení.

3.3 Rodina protokolů

Sockety definují tzv. *rodinu protokolů*, která specifikuje a zahrnuje příbuzné protokoly. Pro všechny protokoly *TCP/IP* je definována jedna rodina: `PF_INET`. Tato symbolická konstanta zahrnuje celou rodinu protokolů *TCP/IP*. Výběr použitého protokolu se v programovém rozhraní *sockets* zadává *typem požadované služby*, nikoliv jménem protokolu. Tak např. protokol *UDP* se vyžádá jako služba datagramového typu (viz kap. 3.5.1).

3.4 Adresát

TCP/IP protokoly definují adresáta pomocí *IP* adresy a čísla protokolového portu, neboli socketu. Aby byla zachována obecnost programového rozhraní, jsou zavedeny *rodiny adres*, které specifikují reprezentaci adres. Každá *rodina protokolů* může využívat jednu nebo více *rodin adres*.

TCP/IP protokoly používají všechny shodnou reprezentaci adres a tato rodina adres je označena symbolickou konstantou `AF_INET`. Často se tato konstanta zaměňuje s konstantou rodiny protokolů, což lze nalézt v programové dokumentaci programového rozhraní i u komerčních produktů (např. Interactive UNIX, HP UX apod.). Pro *TCP/IP* jsou naštěstí obě konstanty co do hodnoty shodné (obě konstanty mají hodnotu 2), takže chyba se projeví pouze chybným formálním zápisem.

Některé rutiny manipulují s adresou bez ohledu na její obsah a přiřazení k příslušné rodině adres. Zavádí se proto obecný formát adresy, který je definován strukturou `sockaddr`:

```
struct sockaddr {
      u_short sa_family;      /* type of address  */
      char    sa_data[14];    /* value of address */
};
```

Bohužel, ne všechny adresy vyhovují této specifikaci a položka `sa_data` je pro některé rodiny adres příliš krátká (např. pro komunikaci prostřednictvím pojmenované roury je adresa jméno souboru, které snadno přesáhne délku 14 bytů). Každá rodina adres si definuje přesnou specifikaci adres a má-li být program portabilní, je nutné používat pro definici proměnných příslušnou strukturu. Jedná se přitom pouze o rozdílný pohled na stejná data. Pro rodinu adres `AF_INET` je formát adresy specifikován následovně:

```
struct sockaddr_in {
      u_short sin_family; /* type of address */
      u_short sin_port;   /* protocol port number */
      u_long sin_addr;    /* IP address */
      char sin_zero[8];   /* unused (zero) */
};
```

3.5 Hlavní systémová volání

Jednotlivá volání *sockets* lze principiálně rozdělit do dvou oblastí.

- volání, která zpřístupňují služby nižších vrstev,
- pomocná volání pro ulehčení práce programátora.

Nejprve se budeme věnovat první oblasti, která je základní pro pochopení tohoto rozhraní. Jednotlivé funkce nejsou popsány do podrobností, neboť detaily je možné nalézt v každé dokumentaci příslušného operačního systému.

3.5.1 `socket()`

Základní volání pro vytvoření struktury *socket*. Používá se jak **klientem** tak i **serverem**. Argument specifikuje rodinu protokolů (pro *TCP/IP* je to `PF_INET`) a specifikaci použitého protokolu prostřednictvím typu používané služby. Protokol *TCP* se volí typem služby stream (konstanta `SOCK_STREAM`), protokol *UDP* typem služby datagram (konstanta `SOCK_DGRAM`). Proběhlo-li volání bez chyb, funkce vrací deskriptor pro nový socket (obr.2).

3.5.2 connect()

Volání `connect()` používá **klient** k vytvoření spojení ke vzdálenému serveru. Parametry specifikují IP adresu a protokolový port, se kterým chceme navázat spojení. Volání `connect()` lze použít jak pro spojení se spojovanými službami (*TCP*) tak i pro spojení s nespojovanými službami (*UDP*). Po vytvoření spojení může klient zahájit výměnu dat mezi ním a specifikovaným serverem.

3.5.3 write()

Klient i **server** používají `write()` pro posílání dat. Klienti obvykle posílají dotazy a požadavky, server obvykle odpovědi. Při volání se specifikuje deskriptor socketu, adresa dat a délka. Volání `write()` předá data jádru operačního systému a vrátí okamžitě řízení zpátky aplikaci, zatímco jsou data přenášena. Je-li linka přeplněna, může volání `write()` aplikaci pozastavit.

3.5.4 read()

Klient i **server** používají `read()` pro příjem dat. Klient čte obvykle odpovědi na své dotazy, zatímco server pasivně očekává klientovy dotazy. Volání `read()` specifikuje opět deskriptor socketu, adresu vyrovnávací paměti a její délku.

Volání má rozdílné chování pro *TCP* a pro *UDP*. Pro *TCP* `read()` přečte data přišlá na socket a zkopíruje je do uživatelovy vyrovnávací paměti. Je-li dat méně než délka vyrovnávací paměti, zkopíruje všechna, je-li dat více, zaplní vyrovnávací paměť a ostatní data předá při dalším volání `read()`.

Pro *UDP* každé volání přečte **jeden** datagram. Je-li datagram delší než vyrovnávací paměť, je předán pouze datagram v délce vyrovnávací paměti a zbytek je zahozen.

V obou případech vrací `read()` počet úspěšně přečtených bytů.

3.5.5 close()

Volání slouží k dealokaci socketu. Používá-li socket více procesů, funkce `close()` způsobí pouze dekrementaci počitadla procesů a socket zanikne až tehdy, dosáhne-li počitadlo procesů nulu.

3.5.6 bind()

Volání `bind()` doplní do socketu identifikaci lokálního uzlu, tj. IP adresu a číslo protokolového portu. Pro spojované služby toto volání používá většinou **server**, který specifikuje *well known port*, na kterém očekává požadavky klientů. Pro definici koncového komunikačního uzlu se využívá struktura `sockaddr_in` (viz kap.3.4).

3.5.7 `listen()`

Zakládáme-li socket, nespecifikujeme, zda se jedná o aktivní nebo pasivní socket. K vytvoření pasivního socketu slouží právě volání `listen()`, které se používá u spojovaných služeb. Pasivní socket je využíván **serverem** na rezervovaných portech. Server musí očekávat komunikaci na rezervovaném portu, neboť komunikaci zahajuje klient a ten musí umět vyplnit kompletně strukturu `sockaddr_in`, která určuje adresáta (a tedy i číslo portu). Funkce `listen()` zároveň umožňuje specifikovat velikost fronty požadavků, které nelze v daném okamžiku zpracovat a které vyčkají ve frontě na pozdější zpracování.

3.5.8 `accept()`

Server pomocí funkcí `socket()`, `bind()` a `listen()` může vytvořit pasivní socket, který očekává požadavky na známém portu a specifikované IP adrese. Přijde-li požadavek, funkce `accept()` (pro spojované služby) vytvoří nový socket, jehož deskriptor je vrácen jako návratový kód. Server používá nově vzniklý socket pro komunikaci s klientem, zatímco originální socket je nadále v pasivním stavu a očekává další požadavky nových klientů.

3.5.9 `recvfrom()`, `recvmsg()`

Při nespojovaných službách jsou tyto funkce používány jak **serverem** tak i **klientem** pro příjem datagramů. Funkce `recvfrom()` navíc zaznamenává odesílatele datagramu.

3.5.10 `sendto()`, `sendmsg()`

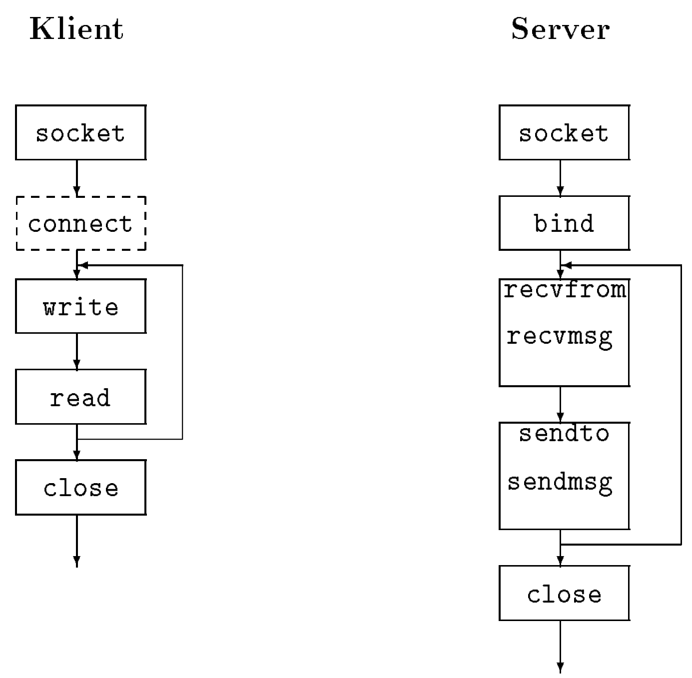
Funkce se používají pro vyslání datagramu jak **serverem**, tak i **klientem**. Funkce `sendto()` může využívat ke specifikaci adresáta datové struktury, která byla vytvořena funkcí `recvfrom()`.

3.5.11 `getsockopt()`, `setsockopt()`

Každý socket má nastaveny vlastnosti, které se mohou lišit případ od případu. Tyto volitelné parametry – *options* – dovolují nastavovat např. velikost vyrovnávací paměti, timeout apod. Každá úroveň protokolu má své volitelné parametry, nejvyšší úroveň je úroveň socketů. Funkce `getsockopt()` dovoluje programátorovi zjistit vlastnosti a hodnoty volitelných parametrů, zatímco funkce `setsockopt()` umožňuje jejich nastavení.

3.6 Typické použití

Použití se liší samozřejmě podle případu, programujeme-li aplikaci jako server nebo jako klient. Další rozdíly jsou při použití spojovaných nebo nespojovaných služeb. Pro protokol *TCP* ukazuje typické použití obr. 4, pro protokol *UDP* obr. 3 (čárkovaně vyznačený blok `connect()` je vysvětlen v kap.**??**).



Obr. 3: Použití funkcí pro nespojované služby.

3.7 Symbolické konstanty

Prototypy funkcí a symbolické konstanty, které mají vztah k programovému rozhraní *sockets*, se vkládají do programu pomocí *include* příkazů. Příkazy *include* mají obvykle formu

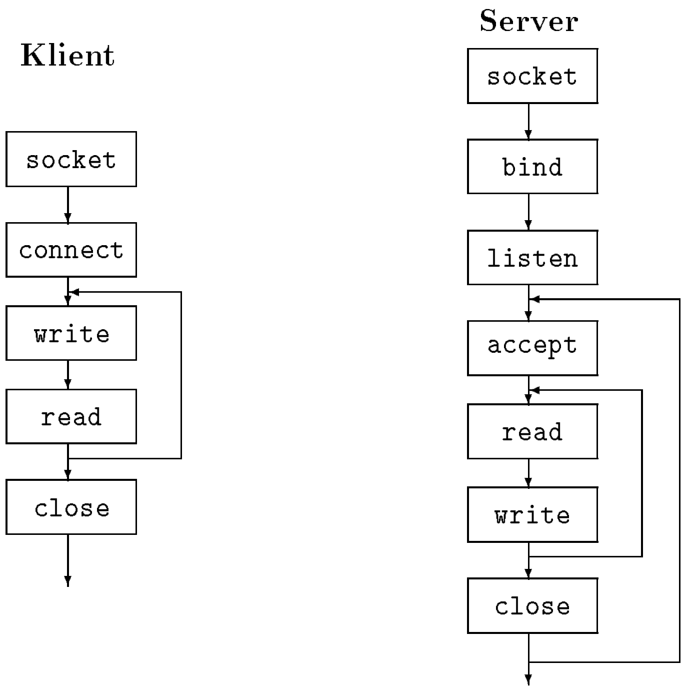
```
#include<sys/types.h>
#include<sys/socket.h>
```

Další hlavičkový soubor, který se úzce vztahuje k programovému rozhraní socketů, je soubor `netinet/in.h`. Kdy který hlavičkový soubor použít je posáno v referenčních popisech každé funkce.

4 Algoritmy, programování

Při psaní aplikace potřebuje programátor znát hlavně algoritmické použití programového rozhraní. Detaily jednotlivých implementací vždy najde v příslušném manuálu, popisující danou implementaci. Přes snahu o unifikaci se implementace na různých operačních systémech liší drobnými detaily. Je proto téměř nutné vždy konsultovat příslušné manuály.

V následujících odstavcích se budeme zabývat základními algoritmy pro tvorbu klienta a jednotlivých druhů serverů.



Obr. 4: Použití funkcí pro spojované služby.

4.1 Aplikace klienta

Software psaný pro klienta je jednodušší než software psaný pro server. Důvody lze shrnout do následujících bodů.

1. Klient nekomunikuje většinou s více servery a nemusí proto obhospodařovat paralelní procesy.

2. Většinou nevyžaduje speciální privilegia ani nepracuje s privilegovanými porty.

3. Při otázkách bezpečnosti dat a případných právech uživatele jsou dostatečné možnosti poskytované operačním systémem.

V dalším nastíníme některé problémy a jejich řešení, které se mohou vyskytnout při používání programového rozhraní socket.

Definované pořadí při přenosu objektů typu integer.

TCP/IP definuje reprezentaci celých čísel, která se vyskytují v hlavičkách protokolů. Tato reprezentace – *network byte order* – říká, že celé číslo je posíláno po síti s nejvýznamnějším bytem jako první.

Vzhledem k tomu, že tato reprezentace nemusí souhlasit s vnitřní reprezentací čísel na všech počítačích, jsou do socketů zařazeny konverzní rutiny, které převádí lokální reprezentaci celých čísel (host) na reprezentaci vyžadovanou *TCP/IP* (network). Jedná se o čtyři funkce – `htons()`, `htonl()`, `ntohs()` a `ntohl()`. Mnemonika je snadná, např. `htons()` převádí host na (*to*) network reprezentaci pro *short integer*. Písmeno `l` je označením pro *long integer*. Při psaní aplikací bychom se vždy měli řídit následujícím pravidlem:

I když je reprezentace celých čísel na vašem počítači shodná s reprezentací požadovanou *TCP/IP*, vždy použijte konverzní funkce (které v tomto případě budou prázdné). Neomezíte tak přenositelnost vašeho programu.

Identifikace serveru.

Existuje několik způsobů, jak určit partnera, se kterým chceme komunikovat. Základní možnosti jsou určit partnera doménovým jménem nebo číselnou IP adresou.

Na rozdíl od zvyklostí (běžně se používá pro specifikaci IP adresy tečková notace např. 147.228.1.1) je nutné použít IP adresu ve formátu celého čísla (binární reprezentace). Pro převod IP adresy reprezentované číselně do binární 32-bitové adresy nám slouží rutina `inet_addr()`, která má argument ASCII řetězec IP adresy a vrací binární ekvivalentní adresu.

Pro převod adresy určené doménovým jménem je k dispozici rutina `gethostbyname()`, která vrací adresu struktury `hostent`. Pro použití těchto prostředků je nutné uvést *include* ve formě

```
#include<netdb.h>
```

Struktura `hostent` je definována

```
struct hostent {
  char *hname;         /* official host name */
  char **h_aliases;    /* other aliases */
  int  h_addrtype;     /* address type */
  int  h_length;       /* address length */
  char **h_addr_list; /* list of addresses */
};
#define h_addr h_addr_list[0];
```

Je-li jméno zadáno pomocí IP adresy, je k dispozici funkce `gethostbyaddr()`, která má shodnou funkci s `gethostbyname()`, tj. naplňuje strukturu `hostent` tentokrát na základě IP adresy. Parametr, kterým se IP adresa zadává, očekává binárně reprezentovanou adresu, převod zajistí již zmíněná funkce `inet_addr()`.

Známe-li tyto funkce, je značně triviální napsat část programu `nslookup` (program `nslookup` nepoužívá programové rozhraní *sockets*, jeho chování se v některých detailech může lišit od našeho příkladu), který převádí IP adresu na doménovou a opačně. Program byl odladěn na HP UNIX.

```
/*
 *   Interactive resolution program.
 *   It is possible to give host address either by name
 *   or by IP address. The answer will be either the IP address
 *   or the textname of the host.
 *   Copyright (c), V.Rudolf, 1993/3
 */

#include<netdb.h>
#include<stdio.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>

extern int h_errno;

char *inp_s() {/*
***************/
static char answ[50];
   printf("\nWhat? -:\n");
   if (scanf("%s", answ) == EOF) exit(0);
     if(answ[0] == '?') {
     printf("\nYou have  the following possibilities:\
```

```
            \n NameAddress - the answer  will be IPAddress\
            \n IPAddress   - the answer will be NameAddress\
            \n CTRL-D      - exit \n");
     return NULL;
   } else
     return answ;
}

void main() { /*
***************/
unsigned long         addr;
struct hostent     *hst;
char               *answ;

   while ((answ = inp_s()) != NULL) {
     if (answ[0] >= '0' && answ[0] <= '9')
/* Numeric character indicated as the leading character */
/* We suppose IPAddress is given                        */
       if ((addr = inet_addr(answ)) == -1)
         printf("Invalid syntax\n");
       else {
         addr = htonl(addr);
         if ((hst = gethostbyaddr(&addr,
                                  sizeof(struct in_addr),
                                  AF_INET)) == NULL)
           switch (h_errno) {
             case HOST_NOT_FOUND : printf("Host not found.\n");
                                   break;
             case TRY_AGAIN      : printf("Try later.\n");
                                   break;
             case NO_RECOVERY    : printf("Irrecoverable error.\n");
                                   break;
             case NO_ADDRESS     : printf("No IP host\n");
                                   break;
           }
         else {
           printf("Host name: %s\n",hst->h_name);
           printf("Aliases   : %s\n",hst->h_aliases[0]);
         }
       }
     else
       if (hst = gethostbyname(answ) ) {
         printf("IP address: %u.%u.%u.%u\n",
```

```
                (unsigned char)(hst->h_addr[0]),
                (unsigned char)hst->h_addr[1],
                (unsigned char)hst->h_addr[2],
                (unsigned char)hst->h_addr[3]);
         printf("Aliases   : %s\n",hst->h_aliases[0]);
       }
       else  printf("Not found.\n");
   }
}
```

Vyhledání specifického portu.

Je mnohem příjemnější, můžete-li zadat službu, kterou požadujete, jménem a požadavek pro server programově převést na určený rezervovaný port – *well known port*. Nemusíme se zpaměti učit, jaké číslo portu je přiřazeno např. službě *ftp*, stačí znát jméno služby. K získání čísla portu slouží funkce `getservbyname()`, které je nutno zadat jako znakový řetězec požadovanou službu a jméno protokolu. Funkce vrátí ukazatel na strukturu `servent`, ve které najdeme i rezervovaný port pro tuto službu.

```
struct  servent {
    char   *s_name;     /* official service name */
    char   **s_aliases; /* other aliases */
    int    s_port;      /* port for this service */
    char   *s_proto;    /* protocol to use       */
};
```

Vyhledání čísla protokolu

Stejná pravidla, jako pro rezervovaná čísla portů, platí i pro protokoly. Také zde existuje mechanismus převodu mnemotechnického znakového řetězce na celé číslo, které je protokolu přiřazeno. Funkce `getprotobyname()` vrací ukazatel na strukturu `protoent`, jejíž definice je:

```
struct protoent {
    char *p_name;     /* official protocol name  */
    char **p_aliases; /* list of aliases allowed */
    int  p_proto;     /* official protocol number */
}
```

4.1.1 *TCP* klient

Základní algoritmus pro *TCP* klienta je možné specifikovat následovně.

⇒ 1. Najdi IP adresu a číslo rezervovaného portu pro službu a server, se kterým chceš komunikovat.

2. Otevři socket.
3. Požádej systém o volný port pro lokální koncový bod.
4. Vytvoř virtuální spoj se serverem.
5. Výměň si své požadavky a odpovědi se serverem.
6. Ukonči spojení.

První bod algoritmu byl již prodiskutován v kap. 4.1.

Otevření socketu je jednoduchá záležitost, jedná se o volání funkce `socket()` s příslušnými parametry.

```
s = socket(PF_INET, SOCK_STREAM, 0);
```

Další úloha, která nás čeká, je výběr lokálního portu. Klient si musí vybrat takový port, který není již používán jiným socketem nebo není vyhrazen jako rezervovaný port. Lokální port je většinou náhodný port, který není v kolizi s předchozími podmínkami. K úplné identifikaci lokálního koncového bodu je třeba ještě udat lokální IP adresu. K problému lokální IP adresy se dostaneme za okamžik. Otevřený socket spojíme se zvoleným portem a IP adresou pomocí funkce `bind()`. Ta se však na obr. 4 nevyskytuje a skutečně, její úkol u klienta přebírá funkce `connect()`, která jako vedlejší efekt doplní do datové struktury socketu (pokud již není doplněn) vybraný port (systémem) a lokální IP adresu. Kromě toho samozřejmě doplní adresu vzdáleného koncového bodu, která je zadána jako parametr, a zinicializuje *TCP* spojení – vytvoří virtuální spoj.

Funkce `connect()` řeší i problém se zadáním lokální IP adresy. Je-li klient spuštěn na počítači s jediným komunikačním rozhraním a tedy s jedinou IP adresou, je situace jednoduchá – lokální IP adresa je jediná a nemáme jiný výběr. Komplikovanější je situace, kdy počítač zastává funkci routeru a má více IP adres. Kterou z lokálních IP adres máme použít? Měla by to být ta, přes kterou protokol IP směruje datagram ke vzdálenému koncovému bodu komunikace. Vzhledem k tomu, že nemáme přímý přístup do směrovacích tabulek, nevíme v daném okamžiku, přes kterou adresu budou naše datagramy směrovány. Abychom měli možnost vyřešit stávající problém, funkce `connect()` nejen zajistí volný port, ale doplní do datových struktur socketu i správnou lokální IP adresu. Funkce `connect()` tedy řeší nejen navázání spojení, ale i automatickou identifikaci lokálního komunikačního koncového uzlu.

Po vytvoření virtuálního spoje je možné začít s výměnou zpráv. Zaslání zprávy serveru je jednoduché a zprostředkuje nám jej funkce `write()`. Adresu připravené zprávy pro server zadáme spolu s délkou jako parametry funkce `write()` a odešleme na patřičný socket. Řízení je předáno programu bezprostředně po volání, odeslání zprávy zařizují spodní vrstvy a nepotřebují již k tomu vlastní program.

Čtení zpráv od serveru je trochu komplikovanější. Vzhledem k možné různorodosti trasy, kudy cestuje datagram, není možné zaručit, že odpověď, kterou očekáváme, bude celá v jednom paketu. *TCP* může data rozdělit na libovolné části, které přicházejí s různým zpožděním. Co nám *TCP* zaručuje, je správnost dat a správné pořadí předávaných dat. Funkce `read()` předává programu data okamžitě po příjmu a zpracování a tak program přijímá data v úsecích, jejichž počet a délku nelze odhadnout předem. Použití funkce `read()` nejlépe demonstruje následující výsek programu.

```
#define LENGTH 100
char *pozadavek = "Dotaz x";
char *ptr;
int  s,buflen;
char buf[LENGTH];
  .
  .
ptr = buf;
buflen = LENGTH;

/* vyšli zprávu */

write(s, pozadavek, strlen(pozadavek));

/* přečti odpověď */

while ((n = read(s, ptr, buflen) > 0) {
   ptr += n;    /* posuň pointer */
   buflen -= n; /* zmenši velikost možné odpovědi */
}
```

Po výměně všech zpráv a uspokojení našich požadavků je nutné relaci správně ukončit. K tomu slouží funkce `close()`, která řeší ukončení spojení a dealokaci socketu. Zamyslíme-li se však nad podstatou *TCP*, nevíme, zda přerušujeme spojení ve správný okamžik, neboť není jisté, zda jsme přijali všechny zprávy od serveru. Proto se mnohem častěji používá funkce `shutdown()`, která k uzavření spojení používá jednoduchý mechanismus. Jedná se o částečné uzavření spojení. Klient specifikuje, že ukončuje relaci a pošle o tom signál serveru pomocí funkce `shutdown()`. Tím se zároveň zaručuje, že již neodešle žádnou zprávu (z plného duplexu se přejde na simplexní spojení). Server akceptuje volání a od této chvíle ví, že již nedostane od klienta další zprávy. Dokončí pouze odeslání případných připravených zpráv a po odeslání poslední již může s klidným svědomím uzavřít spojení a dealokovat socket.

Funkce `shutdown()` má jako parametr deskriptor socketu a hodnotu, která určuje směr uzavření (0 – input, 1 – output a 2 v obou směrech).

```
errcode = shutdown(s, direction);
```

4.1.2 *UDP* klient

Základní algoritmus pro *UDP* klienta je možné specifikovat následovně.

⇒

1. Najdi IP adresu a číslo rezervovaného portu pro službu a server, se kterým chceš komunikovat.
2. Otevři socket.
3. Požádej systém o volný port pro lokální koncový bod.
4. Specifikuj server, na který má být poslán datagram.
5. Výměň si své požadavky a odpovědi se serverem.
6. Ukonči spojení.

Vidíme, že algoritmus je téměř shodný s algoritmem pro *TCP* klienta. Na obr. 3 si můžeme všimnout, že použití funkce `connect()` je čárkované. Rozlišujeme totiž *spojené* (nezaměňujte se spojovaným) nebo *nespojené UDP* sockety.

Ve spojeném socketu klient volá funkci `connect()` a specifikuje vzdálený koncový bod. Tím je jako partner zaznamenán do datových struktur socketu konkrétní vzdálený komunikační bod. Klient se tím zavazuje, že bude komunikovat pouze s tímto uzlem. Pak nemá smysl, aby klient specifikoval při každém vyslání zprávy adresu koncového bodu a komunikuje obdobně jako *TCP* klient prostřednictvím funkcí `write()` a `read()`.

Při nespojeném *UDP* socketu klient nepoužívá funkci `connect()` a při každém vyslání datagramu musí specifikovat koncový uzel, kam má být datagram vyslán. Nevystačí zde již s funkcí `write()`, ale musí použít funkce `sendto()` nebo `sendmsg()`. Výhoda je větší flexibilnost takto napsaného klienta, neboť je možné komunikovat s více servery v jednom běhu programu.

Funkce `connect()` pro *UDP* nezahajuje žádnou komunikaci. Pozor tedy na fakt, že i úspěšná operace `connect()` neznamená, že koncový bod je dosažitelný. Funkce `connect()` proběhne pro *UDP* většinou bez chyb a chyba se vyskytne až při prvním zaslání datagramu k serveru.

UDP klient zasílá i přijímá zásadně celé datagramy. Není tedy třeba opakovaně používat funkci `read()`, neboť funkce přečte vždy celý datagram. Musíme však volit velmi pečlivě velikost vyrovnávací paměti, neboť při nedostatečné velikosti je přijmuta zpráva pouze ve velikosti této paměti a zbytek zprávy ztracen.

Pro uzavření spojení lze opět použít funkce `close()` a `shutdown()`. Funkce `shutdown()` však neinformuje vzdálený server o ukončení komunikace, pouze označí lokální socket jako "neochotný" komunikovat ve specifikovaném směru. Je tedy rozumnější používat funkci `close()`.

5 Aplikace klienta

Ukažme si na jednoduché aplikaci, jak realizovat program, který bude využívat některou z nabízených služeb jako klient. Jako velmi jednoduchá aplikace je vybrána služba *echo*.

Tato služba dovolí poslat dotaz k serveru, který vrací jako odpověď zprávu shodnou s dotazem. Někomu se může zdát, že se jedná o velmi neužitečnou službu, která je však velmi často používána při údržbě počítačové sítě a pro účely ladění.

Pro jednotný přístup k programování klientů dospěje časem každý programátor k tvorbě vlastních funkcí, které mohou být při častém používání zařazeny i do knihovny. Jako základ v našem příkladě byly vzaty s mírnou modifikací funkce, které jsou publikovány v [CS93].

Základní funkce, kterou musí provést klienti spojovaných i nespojovaných služeb, je zřízení socketu a navázání spojení pomocí funkce `connect()`. Protože se jedná o funkce velmi podobné, přímo se nabízí vytvořit funkci, která zřídí socket a na základě parametru vytvoří spojení pro spojované resp. nespojované služby. Funkce se v našem příkladě nazývá `connectsock()` a má tři parametry – adresu serveru, službu a protokol. Všechny parametry lze zadávat pomocí znakového řetězce nebo přímo číselným ekvivalentem pro adresu resp. službu (číslo rezervovaného portu). Pro samotná *TCP* a *UDP* spojení jsou pak vytvořeny funkce `connectTCP()` a `connectUDP()`. Obě dvě funkce volají funkci `connectsock()` pro vytvoření socketu a spojení. Pro *UDP* je vytvořen spojený socket (viz kap. 4.1.2), modifikace pro nespojený socket je snadná a ponechána na čtenáři.

```
/*
 *   Basic subroutines
 *   From "Internetworking with TCP/IP, volume III."
 */
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#ifndef INADDR_NONE
#define INADDR_NONE    0xffffffff
#endif

extern       int      errno;
extern      char      *sys_errlist[];
u_long      inet_addr();
/*
 *       connectsock - allocate & connect a socket using TCP/IP or UDP
 */
int connectsock(host, service, protocol)
/**************************************/
char      *host;      /* name of host to which connection is desired */
char       *service;  /* service associated with the desired port    */
char      *protocol;  /* name of protocol to use ("tcp" or "udp")    */
{
      struct hostent     *phe; /* pointer to host information entry   */
```

```
    struct servent      *pse; /* pointer to service information entry*/
    struct protoent     *ppe; /* pointer to protocol inform. entry   */
    struct sockaddr_in sin;   /* an Internet endpoint address        */
    int      s, type;         /* socket descriptor and type          */

    bzero((char *)&sin, sizeof(sin));
    sin.sin_family = AF_INET;

    /* Map service name to port number */
    if (pse = getservbyname(service, protocol) )
          sin.sin_port = pse -> s_port;
    else if ((sin.sin_port = htons((u_short)atoi(service))) == 0)
          errexit ("Can't get \"%s\" service entry\n", service);

    /* Map hostname to IP address, allowing for dotted decimal */
    if (phe = gethostbyname(host) )
          bcopy(phe -> h_addr, (char *)&sin.sin_addr, phe -> h_length);
    else if ((sin.sin_addr.s_addr = inet_addr(host)) == INADDR_NONE )
          errexit("CAn't get \"%s\" host entry\n", host);

    /* Map protocol name to protocol number */
    if ((ppe = getprotobyname(protocol)) == 0)
          errexit("Can't get \"%s\" protocol entry\n", protocol);

    /* Use protocol to choose a socket type */
    if (strcmp(protocol, "udp") == 0)
          type = SOCK_DGRAM;
    else
          type = SOCK_STREAM;

    /* Allocate a socket */
    s = socket(PF_INET, type, ppe->p_proto);
    if( s < 0)
          errexit("Can't create socket: %s\n", sys_errlist[errno]);

    /* Connect the socket */
    if (connect (s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
          errexit("Can't connect to %s.%s: %s\n", host, service,
                sys_errlist[errno]);
    return s;
}
/*
 * connectTCP - connect to a specified TCP service on a specified host
```

```
 */
int connectTCP (host, service)
/****************************/
char  *host;     /* name of host to which connection is desired */
char  *service;  /* service associated with the desired port    */
{
      return connectsock( host, service, "tcp");
}
/*
 * connectUDP - connect to a specified UDP service on a specified host
 */
int connectUDP (host, service)
/****************************/
char  *host;    /* name of host to which connection is desired */
char  *service; /* service associated with the desired port    */
{
      return connectsock(host, service, "udp");
}
```

V programu je použita kromě standardních funkcí funkce `errexit()`, která slouží pouze pro výpis případných chybových zpráv. Pro pochopení programového rozhraní nemá velký smysl, uvádíme ji pouze pro úplnost.

```
/*
 *  errexit.c
 */

#include <varargs.h>
#include <stdio.h>

int errexit(format, va_alist)
char      *format;
va_dcl
{
      va_list args;
      va_start(args);
      vfprintf(stderr, format, args);
      va_end(args);
      exit(1);
}
```

Tím jsme si připravili základní funkce, které mají poměrně obecné použití a které lze využít pro tvorbu klient programu, ať už založeného na *UDP* službách nebo na *TCP* službách.

Samotný hlavní program pro použití spojované služby *echo* je jednoduchý. Klient vyšle zprávu na zřízený socket a očekává odpověď, která musí být shodná s odeslanou zprávou. Program zároveň měří dobu odezvy. Jako výzva je posílána konstantní zpráva `string`. Všimněte si čtení zprávy, kde vlastně očekáváme předem známou délku zprávy. Při čtení není ošetřena případná chyba, čtenář si snadno může kód modifikovat. Další modifikace, která se nabízí, je interaktivnost zadávání zprávy pro *echo*.

Hlavní program umožňuje spustit program bez parametrů, kdy program pracuje s tzv. *localhost*, což je standardní lokální IP adresa používána především pro ladění programů. V takovém případě echuje počítač, na kterém je klient spuštěn. Je-li zadán parametr, znamená identifikaci vzdáleného počítače buď doménovým jménem nebo IP adresou.

```
/*
 *   Main - TCP client for ECHO service
 */
#include <stdio.h>
#include <string.h>
#include <time.h>

#define LINELEN      128

int main(argc, argv)
int       argc;
char      *argv[];
{
      char *host = "localhost"; /* host to use if none supplied */
      char *service = "echo";   /* default service port         */

      switch (argc) {
      case 1:
            break;
      case 2:
            host = argv[1];
            break;
      default:
            fprintf(stderr, "Usage: TCPecho [host]\n");
            exit(1);
      }
      TCPecho(host, service);
      exit(0);
}

/* TCPecho - invoke echo on specified host & print echo
*/
```

```
TCPecho(host, service)
char      *host;
char      *service;
{
      char      *string = "Echo, ech, ec, e, ..";
      char      buf[LINELEN+1];      /* buffer for online text */
      int      s, n, m, lout;        /* socket, n, read count     */
      struct timeval  first, second;
      struct timezone tzp;

      s = connectTCP(host, service);
      lout = strlen(string);
      write(s, string, lout);
      gettimeofday(&first, &tzp);
      m = 0;
      while((m += (n = read(s, &buf[m], LINELEN))) != lout) ;

      gettimeofday(&second, &tzp);
      if(first.tv_usec > second.tv_usec) {
          second.tv_usec += 1000000;
          second.tv_sec--;
      }

      printf("Echo from %s -: %d.%06d sec\n%s\n",
            host, second.tv_sec - first.tv_sec,
            second.tv_usec - first.tv_usec, buf);
      close(s);
}
```

6 Aplikace serverů

6.1 Iterativní a konkurentní servery

Iterativní server je takový, který zpracovává pouze jeden požadavek v čase, zatímco *konkurentní* server zpracovává více požadavků najednou. U konkurentních serverů se nemusí jednat vždy o zpracování požadavků paralelními procesy, důležité je, že se zpracovává více požadavků současně, tj. že nemusíme čekat se zpracováním požadavku na dokončení předchozího. Je zřejmé, že iterativní server je mnohem snadnější vytvořit než konkurentní. Odměnou za komplikovanější tvorbu konkurentních serverů je pak lepší výkonnost serveru.

6.2 Základní typy serverů

Používané servery se dají klasifikovat pomocí čtyř tříd, které jsou znázorněny na obrázku 5. Nejjednodušší z nich je iterativní server, který se většinou využívá ve spojení s nespojovanými službami, tedy s protokolem *UDP*.

nespojované iterativní	spojované iterativní
nespojované konkurentní	spojované konkurentní

Obr. 5: Základní typy serverů

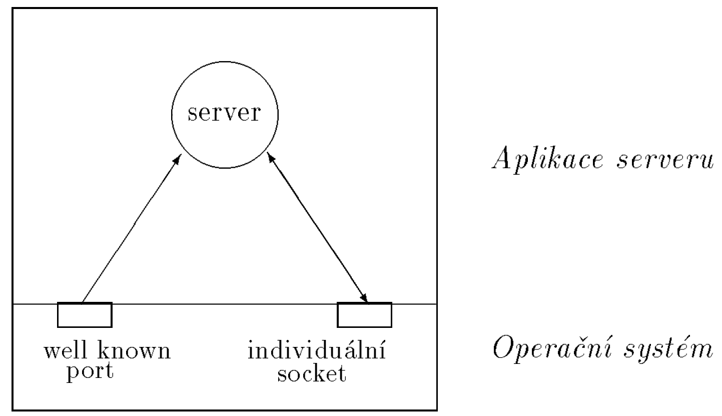
6.3 Iterativní server se spojovanými službami

Specifikujme základní algoritmus na vytvoření takovéhoto serveru.

⇒

1. Otevři socket a spoj jej s požadovaným rezervovaným portem.
2. Přepni socket do pasivního módu.
3. Akceptuj nový požadavek a vytvoř nový socket pro spojení s klientem.
4. Vyměňuj si zprávy s klientem.
5. Ukonči spojení a vrať se na krok 3 a čekej na další požadavky.

Server musí otevřít socket a svázat jej s rezervovaným portem, na kterém bude nabízet své služby. Funkce `bind()`, která specifikuje lokální koncový komunikační bod, vyžaduje však kromě čísla portu i IP adresu. Zde nastává již jednou zmíněný problém. Má-li server více komunikačních interface, který z nich má uvést jako svou IP adresu? Opět by to měla být ta adresa, přes kterou bude vedena komunikace s klientem. To je však v okamžiku volání funkce `bind()` neřešitelná úloha, neboť nemůžeme ani tušit, z kterého komunikačního interface přijde klientův požadavek. Řešení existuje pouze tehdy, má-li server jeden komunikační interface a tedy i jednu IP adresu. Aby bylo možné řešit i server s více komunikačními interface, je definována tzv. *wildcard address*, která se uvádí symbolickou konstantou `INADDR_ANY`. Takto specifikovaná lokální IP adresa pokrývá libovolnou IP adresu lokálního počítače, který tvoří *gateway*. Požadavek klienta na patřičný port je vždy předán serveru nezávisle na IP adrese adresáta. Adresát samozřejmě musí být jedna z IP adres serveru.



Obr. 6: Iterativní *TCP* server.

Pasivní mód serveru se dosáhne pomocí funkce `listen()`. Při iterativním serveru se využívá vnitřní fronta požadavků, ve které se shromažďují příchozí požadavky na spojení a postupně jsou vyřizovány serverem. Zdůrazněme, že jsou to požadavky na spojení, nikoliv jednotlivé zprávy, které přicházejí na již akceptované spojení. Funkce `listen()` dovolí současně i určit velikost fronty požadavků.

Akceptování spojení znamená volání funkce `accept()`, která vytvoří pro spojení nový socket, na kterém pak probíhá veškerá komunikace. Komunikace se děje prostřednictvím funkcí `read()` a `write()`. Po ukončení relace se volá funkce `close()`, která uzavře tento pracovní socket a server se vrací opět na původní socket s rezervovaným portem a opět pomocí funkce `accept()` zpracuje požadavky stojící ve frontě nebo čeká na případné další.

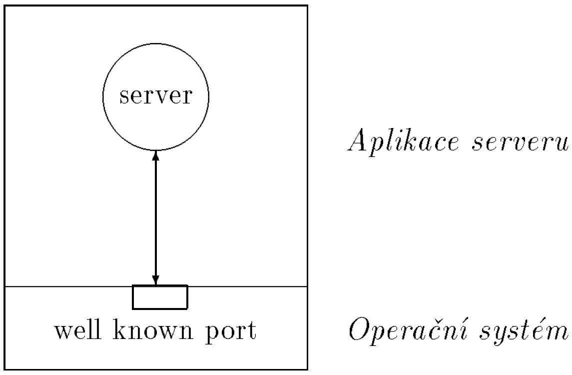
Server svou práci vlastně nikdy nekončí. Jeho ukončení můžeme zařídit vhodným signálem, který je možné iniciovat pomocí klávesnice nebo příkazem `kill`.

6.4 Iterativní server s nespojovanými službami

Iterativní servery pracují dobře, pokud zpracování dotazu probíhá rychle a je časově nenáročné. Tomuto požadavku dobře vyhovují právě servery pracující s protokolem *UDP*, který má podstatně nižší nároky na zpracování nižšími vrstvami. Algoritmus pro iterativní *UDP* servery je následující.

⇒

1. Otevři socket a spoj jej s požadovaným rezervovaným portem.
2. Čti ve smyčce požadavky klientů, zformuluj odpověď a odešli na adresu klienta.



Obr. 7: Iterativní *UDP* server.

Iterativní *UDP* server nemůže používat spojený socket, tj. volat funkci `connect()`. Tím by se omezila jeho činnost pouze na jedno spojení s klientem, což by pravděpodobobně ve většině případů nevyhovovalo. Server tohoto typu musí tedy používat nespojený socket. V každém volání pro odeslání zprávy musí specifikovat nejen odesílanou zprávu, ale i komu je tato zpráva určena. Většinou k tomu používá funkci `sendto()`, kde specifikuje ukazatel na strukturu `sockaddr_in`, v níž je uložena identifikace koncového bodu.

Programové rozhraní *sockets* umožňuje snadné získání adresy vzdáleného komunikačního bodu. Server má k dispozici funkci `recvfrom()`, která slouží pro příjem zprávy. Kromě přijaté zprávy automaticky zaznamenává adresu odesilatele, kterou uloží do parametrem specifikované oblasti. Takto získaná adresa se dá přímo použít funkcí `sendto()` k odeslání odpovědi.

6.5 Konkurentní server se spojovanými službami

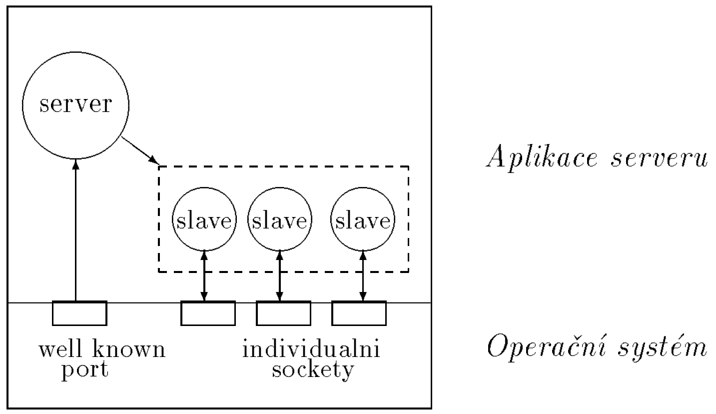
Tento typ serveru je velmi často používán a je dobře znám uživatelům Internetu, neboť servery pro `ftp` případně `telnet` služby jsou právě psány jako konkurentní *TCP* servery. Vytvořené spojení mezi klientem a serverem typicky obsluhuje více požadavků – mezi klientem a serverem probíhá výměna více dotazů a odpovědí. Konkurentnost je dosažena pomocí funkce `fork()`, i když se většinou jedná o zdánlivou konkurentnost, která je dosažena sdílením procesoru v čase. Základní algoritmus pro tento typ serverů je následující.

⇒

1. Otevři socket a spoj jej s požadovaným rezervovaným portem.
2. Specifikuj socket jako pasivní.
3. Pomocí funkce `accept()` přijímej požadavky a vytvoř pro každý požadavek nový proces.
4. V novém procesu vytvoř nové spojení na socketu založeném funkcí `accept()`.

5. Vyměňuj si zprávy s klientem na nově zřízeném spojení.
6. Uzavři socket a ukonči proces.

Server typicky vytváří více procesů. V algoritmu je nutné důsledně oddělit rodičovský proces a proces potomka. Uvedený algoritmus není samozřejmě jediným univerzálním řešením. Další řešení je možné pomocí jediného procesu, kdy využíváme více socketů, které jsou obsluhovány pomocí funkce `select()`. Schematicky lze vyjádřit konkurentní *TCP* servery obrázkem 8.



Obr. 8: Konkurentní *TCP* server.

6.6 Konkurentní server s nespojovanými službami

Tento server je nejméně používaný, neboť se jedná většinou o služby, které pracují na principu individuálního dotazu a odpovědi a pro takovou jednoduchou službu je poměrně "drahá" záležitost tvorba nového procesu. Obrázek 8 vyjadřuje i konkurentní *UDP* servery. Algoritmus pro tyto servery je uveden jen pro úplnost, tento server je velmi vzácný.

⇒

1. Otevři socket a spoj jej s požadovaným rezervovaným portem.
2. Pomocí funkce `recvfrom()` vyčkej na požadavek klienta.
3. Založ nový proces a speciální socket pro relaci.
4. V novém procesu odešli odpověď pomocí funkce `sendto()`.
5. Po odeslání odpovědi uzavři socket a ukonči proces potomka.

V algoritmu opět rozlišujme procesy rodičovský, který vlastně nikdy nekončí a procesy potomků, které naopak mají velmi krátkou dobu života.

Literatura

[Com91] Douglas E. Comer. *Internetworking With TCP/IP*, volume I: Principles, Protocols, and Architecture of *Prentice-Hall Software Series*. Prentice-Hall International, Inc., Englewood Cliffs, N. J. 07632, second edition, 1991.

[Cor90] INTERACTIVE Systems Corporation. *INTERACTIVE TCP/IP Guide*. Santa Monica, California 90404, 1990.

[CS91] Douglas E. Comer and David L. Stevens. *Internetworking With TCP/IP*, volume II: Design, Implementation, and Internals of *Prentice-Hall Software Series*. Prentice-Hall International, Inc., Englewood Cliffs, N. J. 07632, 1991.

[CS93] Douglas E. Comer and David L. Stevens. *Internetworking With TCP/IP*, volume III: Client-Server Programming, and Applications (BSD Socket Version) of *Prentice-Hall Software Series*. Prentice-Hall International, Inc., Englewood Cliffs, N. J. 07632, 1993.

[Hal92] Fred Halsall. *Data Communications, Computer Networks and Open Systems*. Electronic Systems Engineering Series. Addison-Wesley Publishing Company, third edition, 1992.

[Tan81] Andrew S. Tanenbaum. *Computer Networks*. Prentice-Hall Software Series. Prentice-Hall International, Inc., Englewood Cliffs, N. J. 07632, 1981.