Komunikace pomocí Socketu

Vilém Vychodil

7. ledna 2001

Abstrakt

Tento referát se zabývá komunikací pomocí Socketů a je zaměřen na prostředí operačních systémů odvozených od systému UNIX. Socket jako komunikační nástroj poprvé představila implementace Berkeley UNIXu, známá pod zkratkou BSD. Aparát je dostatečně mocný a díky svým doménám i dostatečně variabilní, tato variabilita spočívá především, ale nikoli jen, v možnosti použití stejného API například při komunikaci přes TCP/IP, IPX, AppleTalk či Xerox Network. Zaručuje i platformovou přenositelnost, této problematice je věnována samostatná kapitolka. Tento referát je rovněž doplněn ukázkovými kódy.

Socket a jeho typy

Socket je mechanismus, kterým je možno zprostředkovat lokální či vzdálenou komunikaci dvou uzlů, která má charakter *klient/server*. Druhý fakt je stěžejní, do jisté míry je manipulace se socketem obdobná meziprocesové komunikaci pomocí roury, ta však nijak striktně neodlišuje strany, které komunikují.

Komunikaci zahajuje strana *serveru*. Na počátku vyžádá pomocí systémového volání nový volný socket. Na socket je možné se odkazovat pomocí *deskriptoru*, který je v systému UNIX principiálně podobný například deskriptoru souboru. Novému socketu je přiřazena adresa. Tvar a specifikace adresy je už ale přímo závislá na použité *socketové doméně*, v dalším textu se budu zabývat pouze souborovými sokety a sokety domény internetu. Příchozí „zprávy“ od klienta jsou na serveru řazeny ve frontě, server je obvykle cyklicky obsluhuje. Pokud se server rozhodne přijmout příchozí připojení, vytvoří se pro něj *nový socket*, který je odlišný od původního, tento socket se používá ke komunikaci s daným klientem. Koncepce socketu tedy principiálně umožňuje připojení několika klientů k jednomu serveru.

Činnost klienta je o mnoho průhlednější. Klient vytvoří nový socket, kterému je posléze přiřazena adresa serveru. Potom se klient může pokusit o komunikaci se serverem. Po ukončení komunikace by měly obě strany uzavřít spojení pomocí volání jádra `close`(2). Komunikace klient/server pomocí socketu je obousměrná.

Socketové domény

Socketová doména jednoznačně charakterisuje způsob, kterým mezi sebou uzly komunikují. Některé domény jsou určeny pouze pro lokální komunikaci, jiné přes různé síťové služby, využívající služeb různých síťových vrstev.

Základní domény jsou,

- *souborová doména*, označovaná `AF_UNIX`,
- *doména internetu*, označovaná `AF_INET`.

Souborová doména má lokální charakter, při vytvoření socketu je obvykle v adresáři `/tmp` vytvořen *speciální soubor* — socket, jehož přístupová práva jednoznačně omezují přístup klientů k serveru. Toto typické UNIXově elegantní řešení bohužel nenalézá ekvivalenty u jiných operačních systémů.

Doména internetu využívá protokolu IP, který na síťové vrstvě zaručuje propojení dvou rozhraní. Každá rozhraní v IP síti má unikátní IP adresu, která se skládá ze čtyřbytového čísla, obvykle zapisovaného po bytech oddělených tečkami. Za každým IP rozhraním ale může existovat řada otevřených socketů používajících stejnou socketovou doménu, proto je socket identifikován dvojicí *IP adresa* a *port*, kde port unikátní šestnáctibitové číslo. Operační systém obvykle vyhrazuje jistý počet portů, které je možné obsazovat pouze s privilegii uživatele `root`. Jsou to obvykle porty 0–1023. Specifikace X/Open definuje v header souboru `netdb.h` konstantu `IPPORT_RESERVED`, která udává nejvyšší číslo reservovaného portu.

Typ spojení a protokoly

V síťových technologiích se obvykle uvažují dva kvalitativně odlišné druhy přenosových služeb, *spolehlivé* a *nespolehlivé*. Pomocí spolehlivé služby se obvykle navazuje trvalé spojení, pomocí kterého se transferuje proud dat od odesílatele k příjemci. Podstatné je to, že je zaručeno, že nedojde ke ztrátě dat[1]. Opakem je nespolehlivý transfer, při němž jsou obvykle data kouskována do *datagramů*, které se pošlou příjemci. Některé dojdou, jiné se ztratí, jiné dojdou duplicitně. Příkladem takové komunikace budiž *DNS resolver*.

Jednotlivé socketové domény mohou mít více úrovní služeb, u souborové domény není co řešit, ale právě u internetové domény jsou standardně dvě úrovně — *streamy* a *datagramy*. Streamy jsou spolehlivé, jsou nejčastěji využívány, je pro ně vyhrazena konstanta `SOCK_STREAM`, pro datagramy je vyhrazeno `SOCK_DGRAM`. Streamy jsou implementovány pomocí spojení TCP/IP, datagramová komunikace pomocí UDP/IP. Typ spojení v `AF_INET` doméně

[1] Pokud vyloučíme přestřižení kabelu a podobně.

implikuje i protokol, buďto TCP nebo UDP. Při programování aplikací se za argumenty protokolu dosazuje obvykle nula, které znamená implicitní protokol.

Vytvoření a uzavření socketu

Sockety se v operačním systému UNIX vytvářejí pomocí volání jádra `socket`(2), které bere tři argumenty **int** ◁*domain*▷, **int** ◁*type*▷ a **int** ◁*protocol*▷, vrací deskriptor. Argumenty byly již popsány. Po vytvoření socketu je potřeba přidělit mu jeho adresu, ta má již pro každou doménu jiný tvar.

Pro přiřazení adresy se používá funkce `bind`(2), jejíž argumenty jsou deskriptor, struktura s identifikací adresy a délka této struktury. V případě souborového socketu se adresa předává ve struktuře **sockaddr_un**, které má dvě položky, `sun_family`, nastavuje se na hodnotu `AF_UNIX` a `sun_path`, což je řetězec representující soubor. Soubor je vytvořen s právy podle aktuálního nastavení `umask`(2). V případě domény `AF_INET` se používá struktura **sockaddr_in**, která obsahuje identifikátor `sin_family`, který je roven opět `AF_INET`, 16-bitové číslo `sin_port`, a položku `sin_addr` typu **in_addr**, která obsahuje 32-bitové číslo `s_addr` representující IP adresu uzlu.

Pro konversi IP adresy ve tvaru 32 čísla a řetězce zapsaného klasicky s oddělujícími tečkami existují funkce `inet_addr`(3), která z řetězce vrací IP číslo a `inet_ntoa`(3), která konvertuje naopak. Uzavření socketu je triviální, používá se k němu volání `close`(2).

Platformová nezávislost

Kromě jasně definovaného principu socketu je rovněž nutné respektovat práci v heterogenním síťovém prostředí. Různé platformy mohou používat odlišnou representaci celého čísla, některé používají notaci *little endian*, jiné *big endian*. Jelikož TCP/IP neřeší problémy presentace dat, jako například presentační vrstva RM/OSI, musíme je řešit sami. K tomu existuje celá řada funkcí. Jména funkcí jsou ve tvaru `stodx`, kde `s` a `d` označují zdroj a cíl a mohou nabývat hodnot `h` — host nebo `n` — net. Symbol `x` označuje délku položky je buďto `s` — short, nebo `l` pro long integer.

Vytvoření fronty

Server po vytvoření socketu a přiřazení adresy vytvoří frontu, do které se řadí jednotlivé žádosti o spojení. Frontu lze vytvořit pomocí funkce `listen`(2), jejíž první argument je deskriptor socketu, druhý argument je tak zvaný „backlog“. Argument backlogu definuje maximální délku fronty, obvyklá hodnota je pět až deset. Po vytvoření fronty může server přijímat konexe. V případě domény `AF_INET` má adresa uzlu speciální význam, charakterisuje klienty, kteří se mohou k socketu připojit, pokud chceme povolit připojení klienta z libovolného uzlu, můžeme za hodnotu `sin_addr.s_addr` dosadit konstantu `INADDR_ANY`.

Přijetí spojení

Jakmile program na serveru vytvořil a pojmenoval socket, může pomocí systémového volání `accept`(2) čekat na spojení. Toto systémové volání se vrací, když se klient pokusí připojit na socket určený parametrem socket. Argumenty pro funkci `accept` jsou deskriptor socketu, ukazatel na strukturu **sockaddr** a délka. Funkce vrací nový socket, který je stejného typu jako serverový socket s tím, že je použit pouze ke komunikaci s daným klientem. Adresa a informace o volajícím klientu jsou umístěny rovněž ve struktuře **sockaddr**. Délka, předávaná jako argument pro `accept` určuje délku klientovy struktury, pokud je délka překročena, struktura je ořezána. Před voláním `accept` musí být délka nastavena na konkrétní hodnotu.

Jestliže v socketové frontě nečekají žádná připojení, `accept` zablokuje chod programu, to může být v některých situacích patologické. Toto chování je možné potlačit pomocí volání jádra `fcntl`(2) tak, že deskriptoru socketu nastavíme příznak `O_NONBLOCK`.

```
int flags = fcntl (socket, F_GETFL, 0);
fcntl (socket, F_SETFL, O_NONBLOCK | flags);
```

Nyní vrátí `accept` -1, právě když je fronta prázdná.

Žádost o spojení

Pokud se chce klient spojit se serverem, musí de facto vytvořit propojení mezi nepojmenovaným socketem a socketem listeneru na straně serveru. To lze provést voláním `connect`(2). Argumenty jsou deskriptor soketu, struktura s adresou a její délka. Volání `connect` má stejnou blokovací povahu jako volání `accept`, lze ji ale potlačit stejným způsobem.

Vlastní komunikace

K socketu, ať už na straně serveru či klienta, lze přistupovat pomocí volání jádra `read`(2), `write`(2), která jsou definovány standardně v header souboru `unistd.h`. Rovněž je možné použít například fundamentální knihovní funkci `fdopen`(3), která otevře vstupní/výstupní tok typu **FILE** pro deskriptor souboru a dále můžeme se socketem manipulovat například pomocí příkazů `printf`(3) a podobně.

Obsluha více klientů

Klíčovou záležitostí může být otázka, jak efektivně obsluhovat příchozí procesy. Jejich postupné akceptování může být neefektivní, jednotlivá spojení se ve frontě nesmějí nechat dlouho čekat. Na první pohled jednoduché řešení je klonovat server při každém přijetí spojení pomocí volání jádra `fork`(2). Za předpokladu, že nastavíme ignorování signálu `SIG_CHLD` tak činit jistě lze. Ignorování tohoto signálu je vhodné z důvodu, že nečekáme na dokončení dceřiných procesů a mohlo by se stát, že operační systém bude za jistou dobu zazombovaný[2].

U některých typů aplikací ale tento přístup není příliš vhodný, nezanedbatelná režie při vytváření procesu může udělat svoje, navíc takové řešení pro jednoduché problémy patří spíš ke kultuře programování na jiných platformách.

[2]Sorry, Karle, ale to je prostě realita.

Nám jde přitom jen o primitivní problém, naslouchat z několika deskriptorů zároveň. V operačním systému existuje volání jádra `select`(2), které čeká na tři skupiny deskriptorů, volání čeká, dokud alespoň jeden deskriptor z dané skupiny není připraven pro čtení, zápis nebo nemá chybovou podmínku. Volání `select` lze zadat i časový timeout, pokud je timeout roven 0, doba čekání není nijak limitovaná. Po zaregistrování události na deskriptorech lze postupně projít seznam deskriptorů a požadovanou aktivitu.

Síťové informace

Při reálné práci v prostředí internetu se nelze při komunikaci obejít bez implementace překladu doménových jmen na IP adresy. Systém doménových jmen je hierarchický, kořenové servery, označované `.` obsluhují domény první úrovně `org`, `net`, `cz` a tak dále. Klient, respektive klientský software, který hledá IP adresu podle doménového jména se označuje termínem *resolver* — řešitel. DNS servery se řadí do tří úrovní, *primární*, *sekundární* a *cache* servery. Třetí z nich poskytuje neautoritativní odpověď. Samotný proces prohledávání může být rekursivní či nerekursivní. Pro přehled IP adres zpět na jména je definována reversní DNS, existuje speciální doména `.in-addr.arpa`. Doménová jména se skládají z převrácené IP adresy a `.in-addr.arpa`. Resolver používá protokol UDP/IP.

V operačním systému existují funkce na oba dva typy překladů, respektive funkce pro získávání doménových jmen a IP adres. Fundamentální roli při jejich použití hraje struktura **hostent**, definovaná v header souboru `netdb.h`.

```
struct hostent {
  char *h_name;        /* název hostitele */
  char **h_aliases;    /* seznam aliasů */
  int h_addrtype;      /* typ adresy */
  int h_length;        /* délka adresy */
  char **h_addr_list;  /* seznam adres */
};
```

Název hostitele je representován řetězcem, aliasy hostitele jsou v seznamu řetězců, který je ukončen nulovým pointerem, stejně tak je organisován i seznam adres. Informace o hostiteli je možné získat pomocí volání `gethostbyaddr`(3), `gethostbyname`(3). Myslím, že není co dodat. Definiční header soubor `unistd.h` definuje rovněž volání jádra `gethostname`(2), které vrací název aktuálního hostitele.

Informace o službách je možné získat pomocí volání funkcí `getservbyname`(3) a `gethostbyport`(3), které vracejí ukazatel na strukturu **servent**, která je svým způsobem podobná předcházející.

```
struct servent {
  char *s_name;      /* jméno služby */
  char **s_aliases;  /* seznam aliasů */
  int s_port;        /* číslo IP portu */
  char *s_proto;     /* typ služby */
};
```

Typ služby bývá obvykle `"tcp"` nebo `"udp"`. Pomocí popsaných funkcí se lze připojit na standardní službu, symbolické názvy služeb, příslušné porty a protokoly, jsou dány obsahem souboru `/etc/services`.