Socketar i ett nötskal

© Tommy Olsson <tao@ida.liu.se>, Institutionen för datavetenskap, Linköpings universitet.

En *socket* är en ändpunkt för kommunikation mellan processer/program i ett nätverk. Socketar kan användas för att implementera dataöverföring mellan datorer i ett nätverk men också mellan processer som körs på samma dator.

Denna översikt är skriven för UNIX och specifikt för Solaris 2.6, vilket innebär att de programexempel som används kan vara beroende av detta.

Olika slags socketar

Det finns olika slags socketar. Man brukar i princip skilja mellan socketar som är *connection-based* och socketar som är *connectionless*.

Connection-based innebär att avsändande och mottagande socketar har gjort en uppkoppling innan meddelanden utbyts. Socketar som kopplats upp på detta vis kommunicerar enligt en klient/server-model. Servern har en för klienterna känd adress och lyssnar ständigt efter att meddelanden från klienter ska anlända. Normalt behöver man inte ge en klientsocket någon adress i denna typ av uppkoppling.

Connectionless innebär att ingen uppkoppling görs mellan kommunicerande processer. I stället medskickas adressen för sändande respektive mottagande socket i varje meddelande. Socketar kommunicerar i detta fall som jämlikar (*peer-to-peer*). Varje socket har givits en adress och processer kan skicka meddelanden till varandra via dessa adresser.

Adressdomäner och adressformat

I ett UNIX-system finns två vanliga adressdomäner, *UNIX* och *Internet*. I UNIX-domänen utgörs en adress av *sökvägen för en fil* (*path name*). I Internetdomänen utgörs adresser av *namnet på värddatorn* och ett *portnummer.*

Den adressdomän som en socket tilldelas bestämmer dess adressformat och även det transportprotokoll som ska användas (t ex TCP eller UDP). Socketar som ska kommunicera med varandra måste tillhöra samma domän.

Sockettyper

En socket tillhör alltid en viss typ och den bestämmer på vilket sätt data överförs. Socketar som ska kommunicera med varandra måste vara av samma typ.

Typen *virtual circuit* innebär tvåvägs, pålitligt, sekvensiell dataöverföring. Man kan likna denna typ av kommunikation med ett vanligt telefonsamtal. Socketar som är *connection-based* är vanligtvis av typen *virtual circuit.*

Typen *datagram* innebär tvåvägs överföring av, vanligtvis korta, meddelanden. En process som läser från en datagramsocket kan komma att få ta emot meddelanden i en annan ordning än de avsänts i. Dessutom kan meddelanden ha blivit duplicerade. Man kan likna datagramkommunikation med att skicka brev. Det är vanligtvis snabbare att kommunicera via datagramsocketar, jämfört med *virtual circuit*, men å andra sidan mindre pålitligt. En socket som är av typen *connectionless* är vanligtvis en datagramsocket.

Varje sockettyp stöder ett eller flera *transportprotokoll*. Det finns alltid ett givet *standardprotokoll* för respektive sockettyp. Standardprotokollet för *virtual circuit* är *TCP* (Transfer Connect Protocol) och för *datagram* är det *UDP* (User Datagram Protocol). Både UNIX- och internetdomänen stöder dessa typer av socketar.

Socketoperationer

För att hantera socketar finns följande grundläggande, socketspecifika nätverksfunktioner:

`socket`(3N) Skapar en socket. *Adressdomän*, *sockettyp* och *transportprotokoll* anges som argument. Transportprotokollet väljs automatiskt om man anger argumentet 0 för transportprotokollparametern, vilket är det vanliga.

`bind`(3N) Tilldelar en socket ett namn. I UNIX-domänen innebär detta att en socket skapas i filsystemet. Filen måste tas bort av den process som anropat `bind`() då socketen inte längre behövs, vilket görs med `unlink(2)`.

`listen`(3N) Används för att anger maximalt antal väntande klientmeddelanden som kan köas för en serversocket. Socketen lyssnar i och med att `listen`() anropas efter inkommande meddelanden från klientprocesser. Används *ej* i samband med datagramsocketar.

`accept`(3N) Används av en serversocket för att ta emot en uppkopplingsbegäran från en klientsocket. Processen blockeras till dess en uppkopplingsbegäran ankommer från en klientprocess. `listen`() avänds *ej* i samband med datagramsocketar.

`connect`(3N) Används av en klientprocess för att skicka en uppkopplingsbegäran till en serversocket.

`send`(3N) Skickar ett meddelande till en fjärrsocket. Endast för *connected-based* socketar.

`sendto`(3N), `sendmsg`(3N)
Skickar ett meddelande till en fjärrsocket. Kan användas både för *connected-based* och *connectionless* socketar.

`recv`(3N) Tar emot ett meddelande från en fjärrsocket. Endast för *connected-based* socketar.

`recvfrom`(3N), `recvmsg`(3N)
Tar emot ett meddelande från en fjärrsocket. Kan användas både för socketar som är *connected-based* och *connectionless*.

`shutdown`(3N) Stänger en socket. Detta kan gälla läsning eller skrivning eller bådadera.

Andra systemanrop för att hantera socketar är `getsockname`(3N), `getpeername`(3N), `getsockopt`(3N) och `setsockopt`(3N).

Program som använder socketfunktioner ska inkludera `<sys/types.h>` och `<sys/socket.h>`.

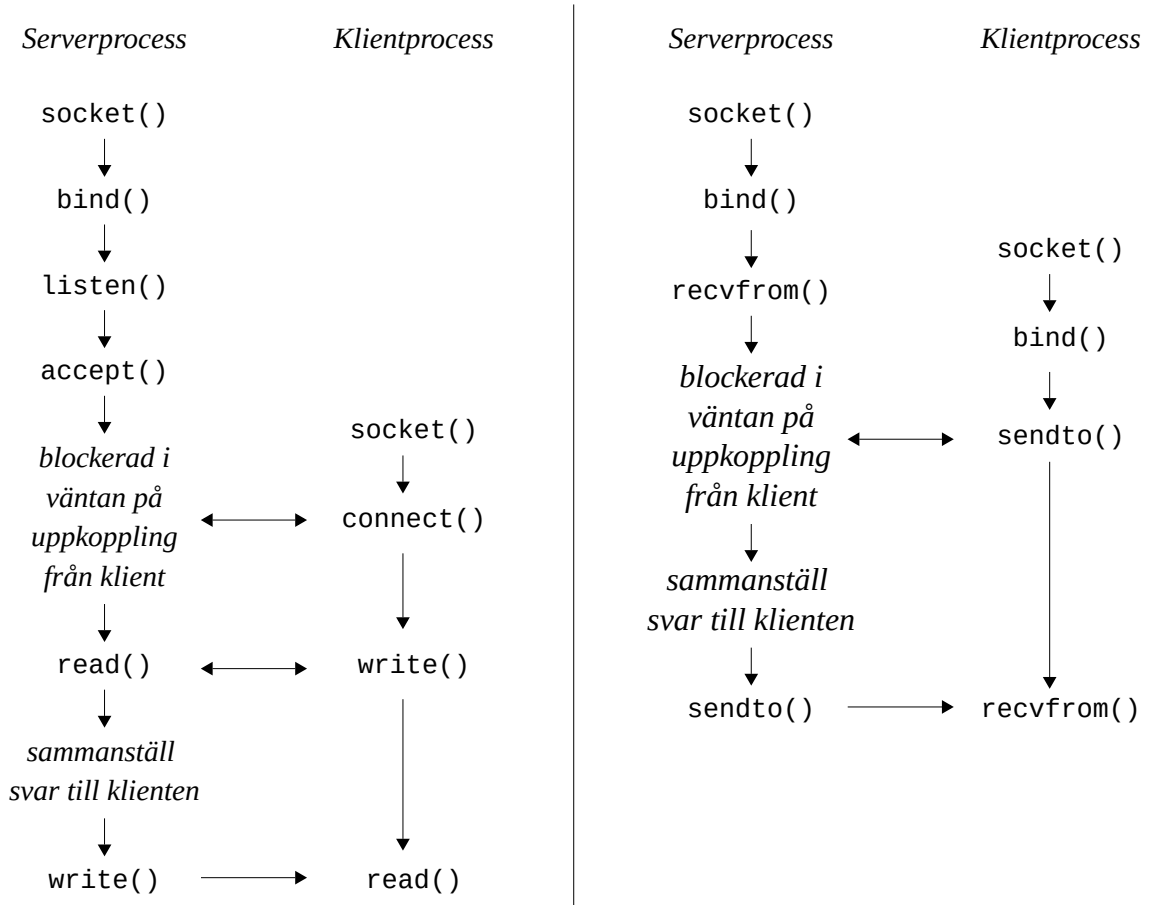
Då man kompilerar/länkar program som använder socketar ska biblioteket `libsocket` ingå i länkningen. Vissa socketfunktioner kräver även bilioteket `libnsl`. Detta anges med länkflaggorna `-lsocket` respektive `-lnsl.`

En socket är egentligen är en *descriptor*. Detta innebär att förutom ovan nämnda socketoperationer kan även många av de systemanrop som finns för filhantering i allmänhet användas, t ex `read`(2V) och `write`(2V).

Det är viktig att man stänger *descriptor*er och för UNIX-socketar även att man tar bort motsvarande filer. För detta kan man använda `close`(2) respektive `unlink`(2).

Grundläggande anropssekvenser för socketoperationer

I nedanstående figur visas i vilken ordning socketoperationer anropas för en *virtual circuit*-baserad klient/server-uppkoppling t.v., och en *connectionless* transaktion, t.h.



Virtual circuit-baserad klient/server-uppkoppling

Connectionless transaktion

Vissa variationer kan förekomma då det gäller val av t.ex. läs/skrivoperationer och adressbindning. För *connectionless* transaktion behöver i vissa fall ej `bind`() anropas.

Att skapa en socket

En socket skapas genom att anropa `socket`(), som returnerar en descriptor (typ **`int`**) eller `-1` om anropet misslyckades. Tre argument ska anges i anropet av `socket`(), domän, sockettyp och protokoll:

```
socket(domän, typ, protokoll)
```

En socket skapas antingen för UNIX-domänen eller för internetdomänen. Följande exempel visar hur en strömsocket (*virtual circuit*) för UNIX-domänen skapas. Den returnerade descriptorn lagras i `s`. Värdet `0` som protokollargument gör att standardprotokollet TCP väljs. `AF_UNIX` och `SOCK_STREAM` är symboler som definieras i inkluderingsfilerna som nämnts ovan.

```
int  s = socket(AF_UNIX, SOCK_STREAM, 0);

if (s == -1) {
   perror(”socket”);
   exit(1);
}
```

Om man vill skapa en strömsocket för internetdomänen ser `socket`()-anropet ut enligt följande.

```
int  s = socket(AF_INET, SOCK_STREAM, 0);
```

Om datagramsocketar för UNIX- respektive internetdomänen ska skapas, byts argumentet `SOCK_STREAM` mot `SOCK_DGRAM`. Standardprotokollet för datagramsocketar är UDP.

Det finns ytterligare sockettyper, nämligen `SOCK_RAW`, `SOCK_SEQPACKET` och `SOCK_RDM`, men dessa är antingen endast tillåtna för priviligierade användare eller ej implementerade.

Att knyta ett namn till en socket

En socket skapas utan något namn. En fjärrprocess har dock ingen möjlighet att referera till en socket innan en adress har knutits till socketen. Processer som kommunicerar är uppkopplade genom adresser. I UNIX-domänen är en uppkoppling vanligtvis sammansatt av en eller två sökvägar, dels en lokal sökväg och dels en fjärrsökväg.

<*lokal sökväg*, *fjärrsökväg*>

I internetdomänen är en uppkoppling sammansatt enligt följande:

<*protokoll*, *lokal-IP-adress*, *lokalt-portnummer*, *fjärr-IP-adress*, *fjärrportnummer*>

Funktionen `bind`() används för att ange den lokala adresshalvan i en uppkoppling, `accept`() och `connect`() komplettera adressen med den andra halvan, fjärradressen.

Ett anrop av `bind`() returnerar `0` om operationen lyckades, `-1` om något fel inträffade. Ett anrop av `bind`() kan typiskt se ut enligt följande:

```
if (bind(socket, namn, namnlängd) == -1) {
   perror(”bind”);
   exit(1);
}
```

Argumentet *socket* är en descriptor som tidigare skapats med `socket`(), t.ex. `s` ovan. Argumentet *namn* är en post vars typ och innehåll beror av om det är en socket i UNIX-domänen eller i internetdomänen. Argumentet *namnlängd* beräknas på olika sätt, beroende på om namn avser UNIX- eller internetdomänen.

En socketadress i UNIX-domänen definieras i `<sys/un.h>` enligt följande:

```
struct sockaddr_un {
   short  sun_family;
   char   sun_path[108];
};
```

Komponenten `sun_family` ska sättas till `AF_UNIX` och komponenten `sun_path` till det namn (filnamn) som önskas. Den namnlängd som ska ges till `bind`() beräknas som storleken av `sun_family` plus längden av namnet i `sun_path`. Inga tomtecken (*null bytes*) ska räknas.

Följande kodavsnitt visar hur man knyter ett namn ”/tmp/SOCK” till en UNIX-domänsocket s. Observera typomvandlingen av adressen/pekaren till addr.

```
#include <sys/un.h>
…
struct sockaddr_un  addr;
int                 addr_size;

addr.family = AF_UNIX;
strcpy(addr.sun_path, ”tmp/SOCK”);
addr_size = sizeof(addr.sun_family) + strlen(addr.sun_path);

if (bind(s, (struct sockaddr*)&addr, addr_size) == -1) {
   perror(”bind”);
   exit(1);
}
```

En internetadress är mer komplicerad och definieras i <netinet/in.h> enligt följande:

```
struct sockaddr_in {
   short           sin_family;
   u_short         sin_port;
   struct in_addr  sin_addr;
   char            sin_zero[8];
};
```

Följande kod visar hur man, givet ett namn host_name och ett portnummer port, knyter en IP-adress till en internetdomänsocket socket_id. I detta fall ska *namnlängd* vara storleken på hela posten. Användning av **struct** hostent och gethostbyname(3N)[1] kräver inkludering av <netdb.h>.

```
#include <netiniet/in.h>
#include <netdb.h>
…
struct sockaddr_in  addr;
int                 addr_size = sizeof addr;
struct hostent*     hp;

addr.family = AF_INET;
hp = gethostbyname(host_name);
memcpy((char*)&addr.sin_addr, (char*)hp->h_addr_list[0], hp->h_length);
addr.sin_port = htons(port);

if (bind(s, (struct sockaddr*)&addr, addr_size) == -1) {
   perror(”bind”);
   exit(1);
}
```

gethostbyname(3N) returnerar en post med information om den värddator vars namn angivits som argument till gethostbyname(). memcpy(2) kopierar adressen från denna post till addr. Om värddatorns och nätverkets byteordning skiljer, översätter htons(3N) portadressen port så att byteordningen i addr.sin_port blir korrekt. Om ordningen är densamma utför htons() inget.

1. Det finns en MT-säker variant gethostbyname_r(3N).

Uppkopplade förbindelser

Upprättning av en uppkopplad förbindelse är ofta asymmetrisk, dvs en process agerar som server och den andra som klient. Serverprocessen binder en socket till en känd adress för den tjänst som erbjuds. Därefter blockeras serverprocessen i väntan på att en begäran om uppkoppling inkommer från en klient.

Serversidan

Följande exempel förutsätter att en internetsocket `srv`, som ska agera server, har skapats med `socket`() och bundits till en adress med `bind`(). För att kunna ta emot en begäran om uppkoppling måste sedan två steg utföras. Det första är att med `listen`() ange hur många väntande uppkopplingar som kan köas (5). Det andra steget är att invänta en uppkoppling. Detta sker med `accept`(), som innebär att processen blockeras i väntan på uppkoppling; `accept`() returnerar alltså först efter att en uppkopplingsbegäran inkommit.

```
int                 clt;
struct sockaddr_in  from;
int                 flen = sizeof from;
…
if (listen(srv, 5) == -1) {
   perror(”listen”);
   exit(1);
}

if ((clt = accept(srv, (struct sockaddr*)&from, &flen)) == -1) {
   perror(”accept”);
   exit(1);
}
```

Adressposten `from` fylls i med med adressen till klienten, och `flen` anger adressens längd.

Funktionen `accept`() returnerar en descriptor för en socket där klienten kopplats upp. Det finns ingen möjlighet för servern att ange att uppkopplingar endast tas emot från specifika adresser. Servern kan dock i efterhand undersöka adressen i `from` och stänga en oönskad uppkoppling.

Klientsidan

En klientprocess som efterfrågar en tjänst initierar en uppkoppling mot serverns socket med ett anrop av `connect`(). En internetuppkoppling kan i detta fall se ut enligt följande:

```
struct sockaddr_in  svr;

svr.sin_family = AF_INET;
…
if (connect(clt, (struct sockaddr*)&svr, sizeof svr) == -1) {
   perror(”connect”);
   exit(1);
}
```

En uppkoppling i UNIX-domänen kan se ut enligt följande:

```
struct sockaddr_un  svr;

svr.sun_family = AF_UNIX;
…
if (connect(clt, (struct sockaddr*)&server,
     sizeof(svr.sun_family) + strlen(svr.sun_path)) == -1) {
   perror(”connect”);
   exit(1);
}
```

Om klientens socket `clt` är obunden då `connect`() anropas, väljer systemet automatiskt en adress och binder denna till socketen. Detta är det vanliga sättet att binda lokala adresser till en socket på klientsidan.

Dataöverföring

Det finns ett flertal funktioner för att sända och ta emot data då man har en *connection-based* uppkoppling. Dels kan man använda de vanliga systemanropen `read`(2) och `write`(2) för att läsa respektive skriva filer, eftersom en socketdescriptor i princip är en fildescriptor.

```
read(fildes, buf, nbyte)
write(fildes, buf, nbyte)
```

Argumentet *fildes* ska vara en socketdescriptor, *buf* ska vara adressen till den minnesbuffert där data ska placeras vid `read`(), respektive hämtas vid `write`(), *nbyte* anger hur många byte som maximalt får läsas respektive som finns att skriva. Om något fel inträffar returnerar `read`() och `write`() `-1`, i annat fall det antal bytes som lästs eller skrivits.

Socketoperationerna `send`(3N) och `recv`(3N) är snarlika filoperationerna `read`() respektive `write`() men har ytterligare en parameter *flags*, som gör det möjligt att styra operationernas utförande.

```
send(fildes, buf, nbyte, flags)
recv(fildes, buf, nbyte, flags)
```

En eller flera av följande flaggor kan anges, i annat fall 0.

`MSG_OOB`	Skicka eller mottag ”out-of-band”-data. Endast för strömsocketar.
`MSG_PEEK`	Data returneras utan att konsumeras, vilket gör det möjligt för en efterföljande läsoperation att se samma data. Endast `recv`().
`MSG_DONTROUTE`	Knappast av intresse för flertalet användare. Endast `send`().

Operationerna `send`() och `recv`() kan endast användas för uppkopplade socketar.

Stängning av socket

En `SOCK_STREAM` kan stängas med systemanropet `close`(2). Om data är köat på en socket som utlovar pålitlig leverans efter stängning, kommer protokollet att försöka överföra detta data. Om datat fortfarande ej har levererats efter en viss tid, kommer det att kasseras.

Systemanropet `shutdown`(3N) kan också användas för att stänga en socket. Båda processerna kan i detta fall tala om att de är klara med sändning. Anropet har formen:

```
shutdown(socket, how)
```

där *how* anger hur stängningen ska ske: `0` medger ingen fortsatt mottagning, `1` medger ingen fortsatt sändning, och `2` medger varken mottagning eller sändning i fortsättningen. `how`() returnerar `0` om anropet lyckades, `-1` om något fel inträffade.

Connectionless dataöverföring

Utbyte av data kan ske symmetriskt. I detta fall finns inget krav på att en förbindelse upprättats innan data överförs, utan varje meddelande bär i stället med sig mottagarens adress.

En socket skapas som tidigare visats med `socket`(), antingen för UNIX- eller internetdomänen. Typen ska vara `SOCK_DGRAM`. Därefter kan ett namn bindas till socketen med `bind`(), men ej nödvändigtvis. Om inte en viss lokal adress krävs av någon anledning, kan man i stället låta systemet automatiskt tilldela en socket ett namn och det sker i så fall första gången som data sänds. `accept`() och `listen`() använd ej för datagramsocketar.

För att sända data på en datagramsocket används systemanropet `sendto`(3N):

```
sendto(socket, buf, buflen, flags, (struct sockaddr*)&to, tolen)
```

Parametrarna *socket*, *buf*, *buflen* och *flags* är desamma som för `send`() och `recv`(), vilka tidigare beskrivits i samband med *connection-oriented* socketar. Parametern *to* anger mottagarens adress, och *tolen* dess längd. `send`() returnerar det antal byte som sänts eller `-1`, om ett lokalt fel inträffat.

För att ta emot ett meddelande på en datagramsocket används `recvfrom`(3N):

```
recvfrom(socket, buf, buflen, flags, (struct sockaddr*)&from, fromlen)
```

Före anrop av `recvfrom`() ska *fromlen* sättas till storleken av *from*. Om anropet lyckas returnerar `recvfrom`() antalet byte som lästs, `-1` om mottagningen misslyckats. Då `recvfrom`() returnerar är *from* ifylld med avsändarens adress och *fromlen* anger dess längd.

Systemanropen `sendmsg`() och `recvmsg`() motsvarar `sendto`() respektive `recvfrom`() men har en parameter *msg* av typen **`struct`** `msghdr` för att minimera antalet parametrarar.

```
sendmsg(socket, msg, flags)
recvmsg(socket, msg, flags)
```

Observera att `sendto`(), `sendmsg`(), `recvfrom`() och `recmsg`() även kan användas för uppkopplade socketar.

Referenser

Bloomer, J.: *Power Programming with RPC*. (1992) O'Reilly.

Chan T.: *Unix System Programming Using C++*. (1997) Prentice-Hall.

Bilaga 1: Serversocket av strömtyp i UNIX-domänen

```
/*
 *  unix_stream_srv.cc
 */
#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
using namespace std;

int main()
{
  // Skapa en strömsocket för UNIX-systemets interna protokoll.

  int  socket_id;

  if ((socket_id = socket(AF_UNIX, SOCK_STREAM, 0)) < 0) {
    perror(”socket”);
  }

  // Bind ett UNIX-filnamn SOCK till serversocketen.

  char             path_name[] = ”SOCK”;
  struct sockaddr  addr;

  addr.sa_family = AF_UNIX;
  strcpy(addr.sa_data, path_name);

  int  len = sizeof(addr.sa_family) + strlen(path_name);
  int  rc;

  if ((rc = bind(socket_id, &addr, len)) < 0) {
    perror(”bind”);
  }

  if (rc != -1 && (rc = listen(socket_id, 5)) < 0) {
    perror(”listen”);
  }

  // Serversocketen lyssnar nu efter uppkopplingar. Om en begäran om
  // uppkoppling från en klientsocket inkommer och uppkopplingen lyckas,
  // returneras en descriptor för klientsocketen, i annat fall -1.

  int  ns_id;

  if ((ns_id = accept(socket_id, 0, 0)) < 0) {
    perror(”accept”);
  }

  // Uppkopplingen lyckades. Socketen kan nu användas för att läsa och
  // skriva data till och från klientsocketen.
```

```
  // Läs ett meddelande från klienten och skriv ut det.

  char       buf[80];
  const int  BUF_SIZE = sizeof buf;

  if ((rc = recv(ns_id, buf, BUF_SIZE, 0)) < 0) {
    perror(”recv”);
  }

  cout << ”Server har mottagit meddelandet: ’” << buf << ”’” << endl;

  // Sänd svar till klienten.

  char*  answer = ”Meddelande 1 mottaget av servern.”;
  int    answer_len = strlen(answer) + 1;

  if (send(ns_id, answer, answer_len, 0) < 0) {
    perror(”send”);
  }

  // Läs ett till meddelande från klienten och skriv ut det.

  if ((rc = recv(ns_id, buf, BUF_SIZE, 0)) < 0) {
    perror(”recv”);
  }

  cout << ”Server har mottagit meddelandet: ’” << buf << ”’” << endl;

  // Sänd svar till klienten.

  answer = ”Meddelande 2 mottaget av servern.”;
  answer_len = strlen(answer) + 1;

  if (send(ns_id, answer, answer_len, 0) < 0) {
    perror(”send”);
  }

  // Stäng socketarna och ta bort filen SOCK.

  shutdown(socket_id, 2);
  close(socket_id);
  shutdown(ns_id, 2);
  close(ns_id);
  unlink(”SOCK”);

  cout << ”Servern avslutad.” << endl;

  return 0;
}
```

Bilaga 2: Klientsocket av strömtyp i UNIX-domänen

```
/*
 *  unix_stream_clt.cc
 */
#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
using namespace std;

int main()
{
  char       buf[80];
  const int  BUF_SIZE = sizeof buf;
  char*      msg;
  int        msg_len;

  // Skapa en klientsocket.

  int  socket_id;

  if ((socket_id = socket(AF_UNIX, SOCK_STREAM, 0)) < 0) {
    perror(”socket”);
  }

  // Gör uppkoppling till en serversocket i UNIX-domänen.

  char             path_name[] = ”SOCK”;
  struct sockaddr  addr;

  addr.sa_family = AF_UNIX;
  strcpy(addr.sa_data, path_name);

  int len = sizeof(addr.sa_family) + strlen(path_name) + 1;
  int rc;

  if ((rc = connect(socket_id, &addr, len)) < 0) {
    perror(”connect”);
    return 1;
  }

  // Sänd ett meddelande till servern.

  msg = ”Meddelande nummer 1 från klienten.”;
  msg_len = strlen(msg) + 1;

  if (send(socket_id, msg, msg_len, 0) < 0) {
    perror(”send”);
  }
```

```
  // Läs svar från servern och skriv ut.

  if (recv(socket_id, buf, BUF_SIZE, 0) < 0) {
    perror(”recv”);
  }

  cout << ”Klienten har mottagit svaret: ’” << buf << ”’” << endl;

  // Sänd ett andra meddelande till servern.

  msg = ”Meddelande nummer 2 från klienten.”;
  msg_len = strlen(msg) + 1;

  if (send(socket_id, msg, msg_len, 0) < 0) {
    perror(”send”);
  }

  // Läs svaret från servern och skriv ut.

  if (recv(socket_id, buf, BUF_SIZE, 0) < 0) {
    perror(”recv”);
  }

  cout << ”Klienten har mottagit svaret: ’” << buf << ”’” << endl;

  // Stäng klientsocketen.

  shutdown(socket_id, 2);
  close(socket_id);

  cout << ”Klienten avlutad.” << endl;

  return 0;
}
```