Örebro universitet
Institutionen för naturvetenskap och teknik
Försättsblad av JPZ

DATORKOMMUNIKATION OCH NÄT

Laboration 4
Mer om UNIX och IPC

Laborationens förhållande till teorin

Laborationerna **UNIX** och **Mer om UNIX och IPC** har till syfte att bl.a. lära ut hur kompilering görs i operativsystemen UNIX och Linux. Många av kommunikationsprotokollen kommer från UNIX-världen. Därför finns det en nära relation mellan datorkommunikation och UNIX som på senare tid har fått en efterföljare som kallas Linux. För nybörjaren i UNIX/Linux ger dessa laborationer också en introduktion till terminaler och kommandon.

I laborationen **Mer om UNIX och IPC** visas en praktisk protokollstack för kommunikation med protokollen TCP och IP. Det är dessa protokoll som är grunden för kommunikation över Internet. Laborationen visar också en vanligt förekommande teknik med Socket som gränssnitt mellan applikationsskiktet och transportskiktet. Det är det senare skiktet som innehåller TCP och UDP. De program som skrivs under laborationen utgör servrar och klienter. Principen klient/server är mycket viktig att förstå eftersom den i hög grad förekommer på lokala nät och Internet. Ett serverprogram har till uppgift att besvara frågor som sänds från klientprogrammen. En klients begäran (eng. *request*) besvaras av servern med ett svar (eng. *reply*). Servrar kallas också hanterare och exempel på sådana är filserver, applikationsserver, databashanterare, FTP-server, utskriftsserver och webbserver.

Redovisning

Laborationen **Mer om UNIX och IPC** redovisas med en skriftlig rapport. I denna rapport ska du beskriva laborationen, redovisa dina svar och programmen som du har skrivit. Använd kursens rapportmall. Rapporten ska bestå av följande delar:

1. Titelsida
   Fyll i rubrik (laborationens titel) och personuppgifter. Gör en kort sammanfattning av laborationen.
2. Innehållsförteckning
   (Höger-klicka på innehållsförteckningen för att uppdatera.)
3. Bakgrund
   Laborationens koppling till datorkommunikation
4. Resultat
   Uppgift 1: beskrivning av uppgiften och kort om resultat
   Uppgift 2: beskrivning av uppgiften och hänvisning till bilaga 1
   Uppgift 3: beskrivning av uppgiften och hänvisning till bilaga 2
   Uppgift 4: beskrivning av uppgiften , hänvisning till bilaga 3 och hänvisning till bilaga 4
5. Bilagor
   1. Program enligt uppgift 2
   2. Program enligt uppgift 3
   3. Program enligt uppgift 4
   4. Makefile (efter den fjärde uppgiften)

DATAKOMMUNIKATION

Laboration 4

Mer om UNIX och IPC

I Berkeley Unix BSD 4.3 gjorde man ett stort tillägg för kommunikation mellan processer, IPC (Interprocess Communication). Syftet var att göra IPC så lika filhantering som möjligt. Redan tidigare hade man infört I/O-deskriptorer för att hantera filer, utrustningar som terminaler (devices) och rör (pipes). Nyheten i BSD 4.3 var att man införde en ny typ av deskriptor, nämligen ***sockets*** ("socklar"), som möjliggjorde kommunikation mellan processer i olika maskiner på samma enkla sätt som filhantering och användning av rör. Socket-gränssnitt har sedan dess införts i alla UNIX-dialekter och i många andra operativsystem som exempelvis VMS och alla Windows-versioner (WinSock).

Begreppet *sockets* är en abstraktion som kan användas i en rad olika protokollfamiljer med tillhörande adressdomäner. Vi ska titta närmare på DARPA:s protokollfamilj TCP/IP med adressdomänen Internet Domain (AF_INET).

Användningen av deskriptorer består i huvudsak av tre faser, nämligen *skapa - läsa/skriva - förstöra*. Om vi arbetar med ordinära filer använder vi systemanropen *open()/creat() - read()/write() - close()*. För rör (pipes) gäller *pipe() - read()/write() - close()*. När vi använder sockets blir det lite mer komplicerat. Vi har i allmänhet en klient som anropar en server, ofta i en annan dator. För att klienten ska kunna hitta servern krävs att åtminstone serverns socket är bunden till någon form av adress. Anropet kan dessutom ske i form av datagram (som *TFTP* över *UDP*) eller förbindelsebaserat (som *Telnet* eller *FTP* över *TCP*).

En TCP-server använder bland annat systemanropen *socket() - bind() - listen() - (select()) - accept() - read()/write() - close()* och en TCP-klient *socket() - connect() - read()/write() - close()*. Som alternativ till *read()/write()* kan *send()/recv()* användas. Socketadresseringen hanteras av *bind()* resp. *connect()*. Eftersom *UDP* är förbindelsefritt, kan inte *connect()* användas. Serverns socket-adress måste bifogas vid varje skrivning. Följdaktligen krävs här en ny skrivinstruktion som ersätter write nämligen *sendto()*. Om en server vill veta vem som anropat den kan *read()* ersättas med *recvfrom()*.

Vad är då en socket-adress? Som tidigare nämnts kan sockets användas i flera adressdomäner. Vi ska titta närmare på internet-domänen (AF_INET). I headerfilen <netinet/in.h> hittar vi följande struct (något förenklat):

```
struct sockaddr_in {
  short sin_family;             /* protocol family  */
  u_short sin_port;             /* port number      */
  struct in_addr sin_addr;      /* Internet address */
  char sin_zero[8];             /* padding          */
};
```

Struct *sockaddr_in* är en variant av den generiska socketadressen *struct sockaddr*, som vi hittar i <sys/socket.h>. Fältet *sin_family* är gemensamt för alla adressfamiljer och talar om vilken variant som används. Om *struct sockaddr_in* används ska *sin_family* tilldelas värdet AF_INET. Fältet *sin_zero* i slutet av structen är utfyllnad (padding), så att *sockaddr_in* får samma storlek som *sockaddr*.

Resten av structen innehåller adressinformationen. Den består i princip av en IP-adress och ett portnummer. Varje tjänst (service) är kopplad till en port och minst ett protokoll. När man kopplar upp sig mot en tjänst i en server måste man förutom serverns IP-adress även ange tjänstens portnummer. Portnummer under 1024 har reserverats för standardiserade tjänster (well known ports). Kopplingen mellan tjänstens namn, portnummer och protokoll finns i filen *services*, som vanligen ligger i katalogen */etc*. Där ser vi bland annat att *FTP* använder *TCP*-port 21, *Telnet* använder *TCP*-port 23 och *TFTP* använder *UDP*-port 69. Vissa tjänster, som *Time* kan använda både *TCP* och *UDP*. Port 0 används om man bara vill ha en ledig port och inte bryr sig om vilken.

Fältet *sin_port* är alltså portnumret i form av en *unsigned short int* i "**network byte order**". Olika datorer lagrar heltal i olika byte-ordning. Somliga lägger den *minst* siginifikanta byten på den lägsta adressen, andra lägger den *mest* signifikanta byten på lägsta adressen. Oavsett vilken ordning datorn använder kallas denna "**host byte order**". Det finns några standard-funktioner deklarerade i <netinet/in.h> som kastar om byte-ordningen i en long resp. short int vid behov. Dessa är *htonl()* (host-to-network long), *htons()* (host-to-network short) *ntohl()* (network-to-host long) och *ntohs()* (network-to-host short). Använd alltid dessa funktioner om programmet hanterar portnummer på något sätt, för att få generella program som fungerar på alla datorer.

Om man ska använda en standardtjänst, som finns listad i filen *services*, bör man använda någon av databasfunktionerna i <netdb.h>. Dessa läser automatiskt information från *services*. Funktionen *getservbyname()* söker efter en tjänst med hjälp av namnet på tjänsten och funktionen *getservbyport()* söker efter ett portnummer (i network byte order) i stället. Båda returnerar en pekare till *struct servent* eller NULL om posten saknas:

```
struct servent {
  char *s_name;          /* official service name */
  char **s_aliases;      /* alias list            */
  int  s_port;           /* port number           */
  char *s_proto;         /* protocol to use       */
};

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

Känner vi till tjänstens namn anropar vi alltså bara *getservbyname()* med namn (exv "telnet") och protokoll ("tcp" eller "udp") i parameterlistan. Fältet *s_port* i *struct servent* ger sedan portnumret i network byte order.

```
struct sockaddr_in sin;
struct servent *sp;
...
if (sp = getservbyname("telnet", "tcp"))
  sin.sin_addr = sp->s_port;
```

Om vi nu återgår till *struct sockaddr_in*, så återstår bara fältet *sin_addr* av typen *struct in_addr*, som endast innehåller den numeriska 32-bitars IP-adressen i *network byte order*:

```
struct in_addr
{
  unsigned long int s_addr;
};
```

Struct in_addr återfinns i <netinet/in.h>. Problemet här är att vi troligen bara har datorns namn, eventuellt det fullständiga domännamnet i den NIS- eller DNS-domän den tillhör eller IP-adressen i decimal punktnotation.

Vi måste alltså översätta en namn- eller adress-sträng till ett 32-bitars heltal i *network byte order*. En server kan dessutom ha mer än en IP-adress om den har flera nätverkskort och är ansluten till flera subnät. Om det är den egna serverns adress vi söker, och kan tänka oss att acceptera anrop från alla subnät kan vi använda "joker"-adressen (wildcard) INADDR_ANY.

```
sin.sin_addr.s_addr = INADDR_ANY;
```

Vi fortsätter med att titta på översättning från decimal punktnotation. I header-filen <arpa/inet.h> hittar vi funktionen *inet_aton()* (ascii-to-network) som gör precis det.

```
int inet_aton(const char *cp, struct in_addr *inp);
```

Vi behöver en temporär in_addr-struct. Funktionsanropet ser ut som följer:

```
struct in_addr sin_addr;
...
if (inet_aton("130.243.104.42", &sin_addr))
  sin.sin_addr = sin_addr;
```

Vid översättning av namn används funktionen *gethostbyname()* i <netdb.h>. Den returnerar en pekare till *struct hostent*.

```
struct hostent {
  char *h_name;             /* official name of host */
  char **h_aliases;         /* alias list            */
  int  h_addrtype;          /* host address type     */
  int  h_length;            /* length of address     */
  char **h_addr_list;       /* list of addresses     */
}
#define h_addr h_addr_list[0]  /* for backward compatibility */

struct hostent *gethostbyname(const char *name);
```

Här får vi eventuellt en hel lista av adresser. Vi tar helt enkelt och kopierar över den första.

```
struct hostent *hp;
...
if (hp = gethostbyname("delta"))
  bcopy(hp->h_addr, &sin.sin_addr, hp->h_length);
```

Några av systemanropen i <sys/socket.h>

Nu när vi vet vad en socketadress är kan det vara dags att titta lite noggrannare på systemanropen.

Socket() skapar en ändpunkt för kommunikation och returnerar en socket-fildeskriptor eller -1 vid fel.

```
int socket(int domain, int type, int protocol);
```

Första parametern, *domain*, är adressdomänen enligt ovan, och ska i vårt fall vara AF_INET. Med AF_UNIX avses Unix interna protokollfamilj och med AF_ISO avses ISO-protokollen. Den andra parametern, *type*, är sockettypen. Den kan ha något av värdena SOCK_STREAM, SOCK_DGRAM, SOCK_RAW, SOCK_SEQPACKET och SOCK_RDM. Den tredje parametern, *protocol*, kan normalt sättas till 0, varvid sockettypens standardprotokoll inom aktuell protokollfamilj används. Internetprotokollens namn och nummer finns i protokolldefinitionsfilen *protocols*, vanligen i katalogen */etc*.

SOCK_STREAM ger en pålitlig tvåvägs förbindelsebaserad (connection oriented) byteströmsocket, med TCP som standardprotokoll. SOCK_DGRAM ger en förbindelsefri (connection less) datagramsocket med UDP som standardprotokoll.

```
int tcp_sock, udp_sock;
...
tcp_sock = socket(AF_INET, SOCK_STREAM, 0);
udp_sock = socket(AF_INET, SOCK_DGRAM, 0);
```

Bind() binder en socketadress till en socket och används framförallt av servrar på alla typer av sockets. Om anropet lyckades, returneras 0, annars -1.

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

Första parametern, *sockfd*, är en fildeskriptor som exempelvis skapats av *socket()*. Andra parametern, *my_addr*, är en pekare till en adress-struct innehållande socketadressen till den server som vi håller på att implementera, dvs den egna IP-adressen och den port vi tänker lyssna på. Här skickar vi alltså med en pekare till en *sockaddr_in*-struct enligt ovan, eftersom vi använder TCP/IP-protokollen. Tredje parametern, *addrlen*, ska vara adressens storlek i byte räknat, dvs structens storlek.

```
int sock, status;
struct sockaddr_in server;
...
server.sin_family = AF_INET;
server.sin_addr.s_addr = INADDR_ANY;
server.sin_port = htons(12345);

status = bind(sock, &server, sizeof(server));
```

Listen() sätter köstorleken och initierar lyssning efter anrop på en socket. Denna funktion används framför allt av servrar på sockets av typen SOCK_STREAM. Funktionsanropet är inte blockerande, dvs processen blir *inte* hängande här i väntan på socketanrop. Om allt gick bra, returneras 0, annars -1.

```
int listen(int sockfd, int backlog);
```

Första parametern, *sockfd*, är en socket-fildeskriptor som skapats av *socket()* och bundits till en adress av *bind()*. Andra parametern, *backlog*, anger den maximala längd kön av väntande anrop får ha. Denna parameter får inte vara större än SO_MAXCONN (normalt 128).

```
int sock, status;
...
status = listen(sock, 5);
```

Accept() accepterar ett anrop på en socket. Även denna funktion används endast av servrar. Detta funktionsanrop är normalt blockerande, dvs här fastnar vi om kön till serverns socket är tom. *Accept()* returnerar en ny socket-fildeskriptor för det accepterade anropet, eller -1 vid fel. Genom att skapa en dotterprocess (*fork()*) som får ta hand om den nya deskriptorn kan moderprocessen genast göra ett nytt Accept-anrop och vi kan på så sätt hantera flera anrop parallellt.

```
int accept(int sockfd, struct sockaddr *clnt_addr, int *addrlen);
```

Den första parametern, *sockfd*, är en socket-fildeskriptor som skapats med sock(), bundits till en adress av *bind()* och lyssnar efter anrop med hjälp av *listen()*. Den andra parametern, *clnt_addr*, är en utparameter med den anropande klientens socketadress. Den tredje parametern är också en utparameter, där vi får *clnt_addr*-structens storlek.

```
int sock, new_sock;
...
new_sock = accept(sock, NULL, NULL);
```

Connect() initierar en uppkoppling mot en socket och används främst på sockets av typen SOCK_STREAM. En klient använder connect för att koppla sig till en serversocket med TCP-protokollet. Om anropet lyckades, returneras 0, annars -1.

```
int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
```

Första parametern, *sockfd*, är en fildeskriptor som exempelvis skapats av *socket()*. Andra parametern, *serv_addr*, är en pekare till en adress-struct innehållande socketadressen till den server vi vill koppla upp oss mot. Här skickar vi alltså med en pekare till en *sockaddr_in*-struct enligt ovan, eftersom vi använder TCP/IP-protokollen. Tredje parametern, *addrlen*, ska vara adressens storlek i byte räknat, dvs structens storlek.

```
int sock, status;
struct sockaddr_in server;
...
status = connect(sock, &server, sizeof(server));
```

Några av systemanropen i <unistd.h>

Read() försöker läsa upp till *count* bytes från fildeskriptorn *fd* till bufferten *buf*. *Count* får inte vara större än SSIZE_MAX. *Read()* returnerar antalet lästa bytes, som alltså är ett heltal ≤ *count*. Vid filslut, stängd pipe eller socket etc. returneras 0 och vid fel returneras -1.

```
ssize_t read(int fd, void *buf, size_t count);
```

Write() skriver upp till *count* bytes från bufferten *buf* till fildeskriptorn *fd*. Om skrivningen lyckades returneras antalet skrivna bytes, 0 om inget skrivits. Vid fel returneras -1.

```
ssize_t write(int fd, const void *buf, size_t count);
```

Close() stänger en fildeskriptor. Om stängningen gick bra returneras 0, annars -1.

```
int close(int fd);
```

INETutils

För att underlätta programmeringen något har jag skrivit ett enklare socket-gränssnitt på en något högre nivå än systemanropen i <sys/sockets>. En vinst i förenkling ger dock oftast en förlust i flexibilitet. För närvarande finns fem funktioner i filen ***INETutils.c*** med tillhörande header-fil ***INETutils.h***. Syftet med *INETutils* är att ni snabbt ska kunna komma igång och skriva enkla klienter och servrar. Därefter är det tänkt att *INETutils* ska fungera som en mall när ni skriver egna klienter och servrar som direkt använder systemanropen i <sys/sockets.h>.

Samtliga funktioner i *INETutils* (utom *TCPclose()*) skickar relevanta felutskrifter till *stderr* och returnerar en felkod, vanligen -1. Här följer en kortfattad beskrivning av alla funktioner. För mer information, se *INETutils.c* eller *INETutils.h*.

INETget_sockaddr() fyller i en socketadress av typen *struct sockaddr_in* med hjälp av de namn, adresser och/eller portnummer som skickats med i parameterlistan. Funktionen anropas internt i *TCPconnect()* och *TCPlisten()* och behöver normalt inte anropas direkt.

TCPconnect() initierar en uppkoppling mot en socket. Den anropar i tur och ordning *INETget_sockaddr(), socket()* och *connect()*.

TCPlisten() initierar lyssning på en socket. Den anropar *INETget_sockaddr(), socket(), setsockopt(), bind()* och *listen()*.

TCPaccept() accepterar ett inkommande anrop. Den anropar endast *accept()*.

TCPclose() stänger en socket. Den anropar *shutdown()* och *close()*.

En TCP-klient kan implementeras med hjälp av *TCPconnect() - read()/write() - TCPclose()* och en TCP-server använder i stället *TCPlisten() - TCPaccept - read()/write() - TCPclose*.

Laboration

Under denna laboration ska CC-kommandon köras från Makefile, ett slags skript. Körning av Makefile görs i **Terminal**. Detta skript ska redigeras för de olika uppgifterna under laborationen. Val av textredigerare är oväsentligt, men en rekommendation är att använda en som kan visa radnummer för att underlätta felrättning. Därför rekommenderas **SciTE**, **utan att kompilera och länka från redigeraren**.

Kompilering med makefile

Det är brukligt att skapa skript som anger hur kompilering och läkning ska gå till. Redigerare som SciTE och Eclipse använder sådana skript. Namnet kan skrivas med versal begynnelsebokstav, men även gemen förekommer. Oavsett hur begynnelsebokstaven skrivs, ges kommandot för att starta skriptet på följande sätt:

$ make
(**$** står för prompten.)

Exempel på Makefile

```
CC = cc
LD = cc
CFLAGS = -c

all:            INETutils.o bounce bounced

clean:          # Empty rule
                rm INETutils.o bounce.o bounced.o

veryclean:      # Empty rule
                rm *.o *.c~ *.h~ Makefile~

INETutils.o:    INETutils.c INETutils.h Makefile
                $(CC) $(CFLAGS) $(IFLAGS) INETutils.c

bounce:         bounce.o INETutils.o
                $(LD) bounce.o INETutils.o $(LIBS) -o bounce

bounce.o:       bounce.c INETutils.h Makefile
                $(CC) $(CFLAGS) $(IFLAGS) bounce.c

bounced:        bounced.o INETutils.o
                $(LD) bounced.o INETutils.o $(LIBS) -o bounced

bounced.o:      bounced.c INETutils.h Makefile
                $(CC) $(CFLAGS) $(IFLAGS) bounced.c
```

Märken (labels) anger möjliga startpunkter inne i skriptet, exempelvis följande:

$ make bounce

Arkiv

1. Gå till **Labbar**.
2. Dekomprimera (höger-klicka > Extract Here) arkivet **IPC-Lab.tar.gz**.
3. Döp om **IPC-Lab** till **ipc**.
4. Kopiera **ipc** till **datorkom**.

Uppgifter

1. Första uppgiften är att provköra de två demoprogrammen: **demobounced** (server) och **demobounce** (klient). (Bokstaven *d* i bounce**d** står för daemon. Inom UNIX- och Linux-världen har daemon fått betydelsen ”program som körs i bakgrunden”, och det är ofta fråga om just servrar.)

   Starta servern från ett terminalfönster med exempelvis

   ```
   konto:~$ ./demobounced 9000
   ```

   där 9000 är ett (nästan) godtyckligt valt portnummer för kommunikationen.

   Ett alternativ är att starta servern och låta den köras i bakgrunden. I sådant fall kan samma terminal användas för klienten. Om så önskas, starta bakgrundskörning med

   ```
   konto:~$ ./demobounced 9000 &
   ```

   Starta klienten från en annan terminal, eller om servern körs i bakgrunden, från samma terminal. Observera att samma port måste anges.

   ```
   konto:~$ ./demobounce 9000
   ```

   Eftersom vår server ger status över förbindelserna, är separata terminaler utan bakgrundskörning av servern att rekommendera.

   Klienten ger anvisningar om hur den kan användas och stängas av. Däremot gör inte servern detta. För att stänga av servern ger man enklast kommandot Ctrl/C. Fler sätt står i handledningen till Linux.

2. Nu är det dags att titta på hur ett serverprogram ser ut. Servern **bounced** väntar på anrop. När ett anrop kommer, skapar moderprocessen en subprocess (**fork()**) som tar hand om anropet samtidigt som moderprocessen återgår till att ta hand om nya anrop. När en anropande klient stänger sin socket terminerar serverns dotterprocess automatiskt. Din uppgift är att skriva färdigt funktionen **take_call()** i filen bounced.c.

   Skriv en loop där du upprepat läser från socketen och omedelbart skriver tillbaka det du läst igen till samma socket. Loopen avslutas vid *End-of-File* (Ctrl/D skickas från klienten). Placera ut utskriftsatserna på lämpligt ställe i koden.

För att kompilera **bounced.c** och tillhörande **INETutils.c** samt länka ihop dem till en exekverbar fil, kan man skriva

```
konto:~$ cc bounced.c INETutils.c –o bounced
```

Det går också att kompilera och länka separat. Då slipper man kompilera om filer man inte har ändrat på:

```
cc –c bounced.c                  /* Ger bounced.o   */
cc –c ./bounced.c                /* Säkert rätt fil */
cc –c INETutils.c                /* Ger INETutils.o */
cc bounced.o INETutils.o –o bounced    /* Länkning */
```

Det vanligaste är nog att man skriver en make-fil (*Makefile*) som beskriver hur beroende-förhållanden mellan alla filer och hur kompilering och länkning ska gå till. Försök att lista ut hur make fungerar genom att titta i **Makefile** och konsultera manualer på Internet. Kompilera och länka därefter **bounced** med hjälp av **make**:

```
konto:~$ make bounced
```

Använd make-filen i fortsättningen av laborationen. (Ignorera varningar angående funktionen **exit()**.)

Provkör din färdiga **bounced** med hjälp av klienten **demobounce**.

3. Skriv en egen testklient, **bounce**, till "eko-servern" **bounced** i uppgift 2. Använd system-anropen i <sys/socket.h> direkt. Du hittar en version av **bounce** som använder **INETutils** bland filerna du har packat upp.

Byt ut INETutils-anropen, ett efter ett, mot motsvarande sys/socket-anrop. Det gäller att ersätta anropen **TCPconnect(host, NULL, port)** och **TCPclose(sock)** med de satser som finns i **INETutils.c**. Det sist nämnda anropet, TCPclose(), är enkelt att ersätta men TCPconnect() är svårare. Det beror på att TCPconnect() i sin tur anropar **INETget_sockaddr()**. Satserna i denna funktion ska också tas med utan att använda ett liknande funktionsanrop av INETget_sockaddr().

Eftersom funktionen INETget_sockaddr() använder en pekare för **sin** som är av typen **struct sockaddr_in**, måste detta beaktas då satserna ska användas i bounce.c. Detta innebär att **sin** ska ersättas med **&sin** och exempelvis **sin->sin_port** ska ersättas med **sin.sin_port**.

Utgå från att alla kommunikationsfunktioner kommer att fungera; Ta bort alla tester som görs med ***if***-satser. Om en if-sats har ett villkor som ges av ett funktionsanrop, ska endast funktionsanropet finnas kvar. Om villkoret använder pekaren **serv**, blir det enkelt. Denna pekare sätts lika med **NULL** i anropet av TCPconnect() och vidare i anropet av INETget_sockaddr(). Den enda if-sats som är svår att förutbestämma, är selektionen som används för *Get host binary IP address in network byte order*. I denna kan du använda fallet **Try host name**.

För sätta sig in i kommunikationsfunktionerna ska **varje** sats som ersätter anropen TCPconnect(host, NULL, port) och TCPclose(sock) kommenteras. Resultatet för ersättningen av TCPconnect(host, NULL, port) kan se ut som följande:

```
bzero(&sin, sizeof(sin));        /* Clear Internet socket
                                    address structure     */
gethostname(buf, sizeof(buf));   /* Get local host name   */
host = buf;                      /* Default host name =
                                     local host name      */
(Fler satser)

connect(sock, (struct sockaddr *) &sin, sizeof(sin));
                                 /* Connection to server  */
```

Modifiera **Makefile** genom att ta bort länkningen mellan **bounce.o** och **INETutils.o**. Kompilera och länka (**make bounce**).

Provkör din version av klienten **bounce** mot servern **bounced** (från uppgift 2).

4. Skriv med hjälp av INETutils (eller utgå från servern **bounced** i uppgift 2) en egen "chargen-server" (teckengenerator) med namnet ***blah-blahd***. (En sådan kan användas för test av terminaler.) På kommando ska den skicka alla skrivbara (printable) ASCII-tecken (ASCII 32-126) och på ett annat kommando skicka alla siffror (digits, 0-9).

Kommando	Utmatning
\n (= [Return] eller [Enter])	ASCII-tecken
p\n eller P\n	ASCII-tecken
d\n eller D\n	Siffror

Om servern ska stänga socketen automatiskt eller om klienten ska kunna ge fler kommandon innan den initierar nedkopplingen med Ctrl/D, kan du välja själv.

Lägg till alternativet ***blah*** i **Makefile** som kompilerar (och länkar) **blah-blahd.c** och **INETutils.c**.

Använd din **bounce** (från uppgift 3) för att testa servern.