



**Università degli Studi di Bologna
Scuola di Ingegneria**

Corso di Reti di Calcolatori T

**Esercitazione 3 (svolta)
Socket C senza e con connessione**

Luca Foschini

Anno accademico 2020/2021

SPECIFICA: SOCKET SENZA CONNESSIONE

Sviluppare un'applicazione C/S che consente di effettuare le **quattro operazioni tra interi ottenendo il risultato in remoto**

Il **Client** è un filtro: invia al server pacchetti contenenti il **tipo di operazione richiesta** (somma, sottrazione, moltiplicazione o divisione) e gli **operandi** su cui effettuarla (**due interi**) forniti dall'utente via l'input da console.

Al ricevimento del risultato dell'operazione, si stampa la risposta e si esegue il ciclo fino alla terminazione

Il **Server** estrae i dati della richiesta, esegue l'operazione, e invia al client un datagramma contenente il risultato

Si noti inoltre che client e server devono gestire opportunamente eventuali **fallimenti** delle operazioni invocate, si veda a tale proposito (anche consultando il **man** di Linux) a cosa serve la funzione **perror**

SCHEMA DI SOLUZIONE: IL CLIENT

1. Inizializzazione indirizzo del client (vedere meglio anche il codice più avanti) e del server, utilizzando gli argomenti di invocazione

(**NB:** indirizzo restituito da *gethostbyname*, dentro *host*, è già in *formato di rete*, mentre **formato porta deve essere convertito** da formato locale a formato di rete):

```
memset((char *)&servaddr, 0, sizeof(struct sockaddr_in));
servaddr.sin_family = AF_INET;
host = gethostbyname (argv[1]);
servaddr.sin_addr.s_addr =
    ((struct in_addr *) (host->h_addr))->s_addr;
servaddr.sin_port = htons(atoi(argv[2]));
```

2. Creazione e binding socket datagram:

```
sd=socket(AF_INET, SOCK_DGRAM, 0);
bind(sd,(struct sockaddr *) &clientaddr,
    sizeof(clientaddr));
```

SCHEMA DI SOLUZIONE: IL CLIENT (ANCORA)

3. Lettura da console dei dati della richiesta (non riportato qui) e invio richiesta operazione al server:

```
sendto(sd, &req, sizeof(Request), 0,  
      (struct sockaddr *)&servaddr, len);
```

4. Ricezione risposta contenente il risultato dal server:

```
recvfrom(sd, &ris, sizeof(ris), 0,  
        (struct sockaddr *)&servaddr, &len);
```

5. Chiusura socket e terminazione:

```
close(sd);
```

Si noti: un **unico datagramma** per l'invio e **non**
datagrammi **molteplici**.

Perché?

SCHEMA DI SOLUZIONE: IL SERVER

1. Inizializzazione indirizzo:

```
memset ((char *)&servaddr, 0, sizeof(servaddr));  
servaddr.sin_family = AF_INET;  
servaddr.sin_addr.s_addr = INADDR_ANY;  
servaddr.sin_port = htons(port);
```

2. Creazione, bind, e settaggio opzioni della socket:

```
sd=socket(AF_INET, SOCK_DGRAM, 0);  
setsockopt(sd, SOL_SOCKET, SO_REUSEADDR,  
           &on, sizeof(on));  
bind(sd, (struct sockaddr *)&servaddr,  
      sizeof(servaddr));
```

SCHEMA DI SOLUZIONE: IL SERVER (ANCORA)

3. Ricezione della richiesta dal client:

```
recvfrom(sd, req, sizeof(Request), 0,  
         (struct sockaddr *)&cliaddr, &len);
```

4. Calcolo del risultato (non riportato qui) e invio della risposta al client:

```
sendto(sd, &ris, sizeof(ris), 0,  
       (struct sockaddr *)&cliaddr, len);
```



Ovviamente, si devono sempre fare `close()` di tutte le socket e i file non più necessari

OPDATAGRAMCLIENT 1/5

```
#include <stdio.h>
...
#define LINE_LENGTH 256
/*****/
typedef struct
{ int op1; int op2; char tipoOp; }
Request;
/*****/
int main(int argc, char **argv)
{
    struct hostent *host;
    struct sockaddr_in clientaddr, servaddr;
    int port, sd, num1, num2, len, ris;
    char okstr[LINE_LENGTH];
    char c; int ok;
    Request req;

    if(argc!=3) // Controllo argomenti
    { printf("Error:%s serverAddress serverPort\n", argv[0]);
      exit(1);
    }
}
```

OPDATAGRAMCLIENT 2/5

```
// Inizializzazione indirizzo client e server e fine controllo argomenti
memset((char *)&clientaddr, 0, sizeof(struct sockaddr_in));
clientaddr.sin_family = AF_INET;
clientaddr.sin_addr.s_addr = INADDR_ANY;
clientaddr.sin_port = 0;
memset((char *)&servaddr, 0, sizeof(struct sockaddr_in));
servaddr.sin_family = AF_INET;
host = gethostbyname (argv[1]);
num1=0; // Verifica correttezza porta e host: farla al meglio
while( argv[2][num1] != '\0' )
{ if( (argv[2][num1] < '0') || (argv[2][num1] > '9') )
    { printf("Secondo argomento non intero\n"); exit(2); }
    num1++;
}
port = atoi(argv[2])
if (port < 1024 || port > 65535)
    {printf("Port scorretta..."); exit(2); }
if (host == NULL){ printf("Host not found ..."); exit(2); }
else // valori corretti
    { servaddr.sin_addr.s_addr=
        ((struct in_addr *) (host->h_addr))->s_addr;
        servaddr.sin_port = htons(port);
}
```


OPDATAGRAMCLIENT 3/5

```
sd=socket(AF_INET, SOCK_DGRAM, 0);           // Creazione socket
if(sd<0) {perror("apertura socket"); exit(1);}
// Bind a una porta scelta dal sistema
if (bind(sd, (struct sockaddr *) &clientaddr,
          sizeof(clientaddr))<0)
    {perror("bind socket "); exit(1);}
printf("Inserisci il primo operando (int), EOF per terminare:");

while ((ok = scanf("%i", &num1)) != EOF /* finefile */) // Filtro
{
    if (ok != 1) // errore di formato
    {
        /* Problema nell'implementazione della scanf. Se l'input contiene PRIMA dell'intero
        * altri caratteri la testina di lettura si blocca sul primo carattere (non intero) letto.
        * Ad esempio: |ab1292\n|
        *               ^   La testina si blocca qui
        * Bisogna quindi consumare tutto il buffer in modo da sbloccare la testina. */

        do
        {
            c=getchar(); printf("%c ", c);
        } while (c!= '\n');
        printf("Inserisci il primo operando (int), EOF per terminare: ");
        continue;
    }
}
```

OPDATAGRAMCLIENT 4/5

```
req.op1=htonl(num1);
gets(okstr); // Consumo il resto della linea
printf("Secondo operando (intero): ");
while (scanf("%i", &num2) != 1)
{
    do
    {c=getchar(); printf("%c ", c);}
    while (c!= '\n');
    printf("Secondo operando (intero): ");
}
req.op2=htonl(num2);
gets(okstr);
printf("Stringa letta: %s\n", okstr);
do
{
    printf("Operazione (+ = addizione, - = sottrazione, ... ");
    c = getchar();
} while (c!='+' && c != '-' && c != '*' && c != '/');
req.tipoOp=c;
gets(okstr); // Consumo il resto della linea
printf("Operazione richiesta: %ld %c %ld \n", ntohl(req.op1),
        req.tipoOp, ntohl(req.op2));
```

OPDATAGRAMCLIENT 5/5

```
len=sizeof(servaddr); // Richiesta operazione
    if (sendto(sd, &req, sizeof(Request), 0,
               (struct sockaddr *)&servaddr, len)<0)
    { perror("sendto"); continue; }
    /* ricezione del risultato */
    printf("Attesa del risultato...\n");
    if (recvfrom(sd, &ris, sizeof(ris), 0,
                 (struct sockaddr *)&servaddr, &len)<0)
    {perror("recvfrom"); continue;}

    printf("Esito dell'operazione: %i\n", ntohl(ris));

    printf("Inserisci il primo operando (int), EOF per terminare:");
}

close(sd); // Libero le risorse: chiusura socket
printf("\nClient: termino...\n");
exit(0);
}
```

OPDATAGRAMSERVER 1/4

```
#include <stdio.h>

...
/*****/
typedef struct
{
    int op1;
    int op2;
    char tipoOp;
}
Request;
/*****/

int main(int argc, char **argv)
{
    int sd, port, len, num1, num2, ris;
    const int on = 1;
    struct sockaddr_in cliaddr, servaddr;
    struct hostent *clienthost;
    Request* req = (Request*)malloc(sizeof(Request));
    if(argc!=2) // Contollo argomenti
    { printf("Error: %s port\n", argv[0]); exit(1); }
}
```

OPDATAGRAMSERVER 2/4

```
else{ // Verifica intero
    ...
    port = atoi(argv[1]);
    if (port < 1024 || port > 65535) // Porta nel range porte disponibili
    { printf("Error: %s port\n", argv[0]); exit(2); }
}

memset((char *)&servaddr, 0, sizeof(servaddr)); // Inizializzazione indirizzo
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = INADDR_ANY;
servaddr.sin_port = htons(port);
sd=socket(AF_INET, SOCK_DGRAM, 0); // Creazione, bind e settaggio socket
if(sd <0)
{perror("creazione socket "); exit(1);}
printf("Server: creata la socket, sd=%d\n", sd);
if(setsockopt(sd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on))<0)
{perror("set opzioni socket "); exit(1);}
if(bind(sd, (struct sockaddr *) &servaddr,
    sizeof(servaddr))<0){perror("bind socket "); exit(1);}
printf("Server: bind socket ok\n");
```

OPDATAGRAMSERVER 3/4

```
for (;;) // Ciclo infinito di ricezione e servizio
{
    len=sizeof(struct sockaddr_in);
    if (recvfrom(sd, req, sizeof(Request), 0,
                (struct sockaddr *)&cliaddr, &len)<0)
    {perror("recvfrom "); continue;}
    num1=ntohl(req->op1); // Trattiamo conversioni possibili
    num2=ntohl(req->op2);
    printf("Operazione richiesta: %i %c %i\n",
           num1, req->tipoOp, num2);
    clienthost=gethostbyaddr( (char *)
                              &cliaddr.sin_addr, sizeof(cliaddr.sin_addr), AF_INET);
    if (clienthost == NULL) printf("client host not found\n");
    else printf("Operazione richiesta da: %s %i\n",
               clienthost->h_name,
               (unsigned)ntohs(cliaddr.sin_port));
}
```

OPDATAGRAMSERVER 4/4

```
if(req->tipoOp=='+')
    ris=num1+num2;
else if(req->tipoOp=='-')
    ris=num1-num2;
else if(req->tipoOp=='*')
    ris=num1*num2;
else if(req->tipoOp=='/')
    if (num2!=0) ris=num1/num2;
/* Risultato di default, in caso di errore.
   Sarebbe piu' corretto avere messaggi di errore, farlo per esercizio */
else ris=0;
    ris=htonl(ris);
if (sendto(sd, &ris, sizeof(ris), 0,
           (struct sockaddr *)&cliaddr, len)<0)
{perror("sendto "); continue;}

} // while

} // main
```

SPECIFICA: SOCKET CON CONNESSIONE

Sviluppare un'applicazione C/S che effettui **l'ordinamento remoto di un file testo** inviato dal client al server e restituito **ordinato** al client stesso, che lo scrive nel proprio file system

Il **Client** chiede all'utente il nome del file da ordinare e del file ordinato

se il file da ordinare è presente, viene stampato a video, inviato via uno stream di output al server e ricevuto ordinato dal server stesso, per poi essere salvato con il nome del file ordinato e stampato a video

Il **Server** attende una richiesta di connessione da parte dei client

Usa la connessione per ricevere il file, e inviare, se disponibile, il file ordinato. L'ordinamento locale viene fatto **invocando il comando "sort"** e **usando la ridirezione**: l'input e l'output del comando, anziché lo standard input e lo standard output, vengono ridiretti sulla socket.

Il server deve gestire le richieste in maniera concorrente

SCHEMA DI SOLUZIONE: IL CLIENT

1. Inizializzazione indirizzo del server dall'argomento di invocazione:

```
memset((char *)&servaddr, 0,  
        sizeof(struct sockaddr_in));  
servaddr.sin_family = AF_INET;  
host = gethostbyname (argv[1]);  
servaddr.sin_addr.s_addr =  
    ((struct in_addr *) (host->h_addr)) ->s_addr;  
servaddr.sin_port = htons(port);
```

2. Creazione socket stream e connessione:

```
socket(AF_INET, SOCK_STREAM, 0);  
connect(sd, (struct sockaddr *) &servaddr,  
        sizeof(struct sockaddr));
```

SCHEMA DI SOLUZIONE: IL CLIENT (ANCORA)

3. Lettura da console dei dati della richiesta: nome file da ordinare e nome file ordinato

4. Lettura e invio del file da ordinare al server:

```
while ( (nread=read(fd_sorg, buff, DIM_BUFF)) > 0 )  
    { write(sd, buff, nread) ; }
```

5. Ricezione e scrittura del file ordinato dal server:

```
while ( (nread=read(sd, buff, DIM_BUFF)) > 0 ) {  
    write(fd_dest, buff, nread) ; }
```

6. Chiusura socket e terminazione:

```
close(sd) ;
```

SCHEMA DI SOLUZIONE: IL SERVER

1. Inizializzazione indirizzo:

```
memset ((char *)&servaddr, 0, sizeof(servaddr));  
servaddr.sin_family = AF_INET;  
servaddr.sin_addr.s_addr = INADDR_ANY;  
servaddr.sin_port = htons(port);
```

2. Creazione, bind, settaggio opzioni, e creazione coda d'ascolto della socket:

```
socket(AF_INET, SOCK_STREAM, 0);  
setsockopt(listen_sd, SOL_SOCKET, SO_REUSEADDR,  
           &on, sizeof(on));  
bind(listen_sd, (struct sockaddr *)&servaddr,  
     sizeof(servaddr));  
...  
listen(listen_sd, 5);
```

SCHEMA DI SOLUZIONE: IL SERVER (ANCORA)

3. Ricezione della richiesta dal client:

```
conn_sd = accept(listen_sd,  
    (struct sockaddr *)&cliaddr,&len);
```

4. Nel processo figlio: ridirezione dei file descriptor di standard input e standard output, ed esecuzione comando di ordinamento (sort, vedere il man):

```
close(1); close(0);  
dup(conn_sd); dup(conn_sd); close(conn_sd);  
execl("/bin/sort", "sort", (char *)0);
```

5. Nel processo padre: chiusura della socket di servizio (non di ascolto!)

```
close(conn_sd);
```

REMOTESORTCLIENT 1/4

```
#include <stdio.h>

...

#define DIM_BUFF 256

int main(int argc, char *argv[])
{ int sd, fd_sorg, fd_dest, nread;
  char buff[DIM_BUFF];
  // FILENAME_MAX: lunghezza massima nome file. Costante di sistema.
  char nome_sorg[FILENAME_MAX+1], nome_dest[FILENAME_MAX+1];
  struct hostent *host;
  struct sockaddr_in servaddr;

  // Controllo argomenti
  if(argc!=3)
  { printf("Error:%s serverAddress serverPort\n", argv[0]);
    exit(1);
  }
```

REMOTESORTCLIENT 2/4

```
// Inizializzazione indirizzo server
memset((char *)&servaddr, 0, sizeof(struct sockaddr_in));
servaddr.sin_family = AF_INET;
host = gethostbyname(argv[1]);
if (host == NULL)
{ printf("%s not found in /etc/hosts\n", argv[1]); exit(1); }
// Verifica intero ...
servaddr.sin_addr.s_addr =
    ((struct in_addr*) (host->h_addr))->s_addr;
servaddr.sin_port = htons(atoi(argv[2]));
// Corpo del client
printf("Ciclo di richieste di ordinamento fino a EOF\n");
printf("Nome del file da ordinare, EOF per terminare: ");
/* ATTENZIONE!! Cosa accade se la riga e' piu' lunga di FILENAME_MAX?
 * Stesso problema per ogni gets. Come si potrebbe risolvere il problema? */
while (gets(nome_sorg))
{ printf("File da aprire: __%s__\n", nome_sorg);
```

REMOTESORTCLIENT 3/4

```
if((fd_sorg=open(nome_sorg, O_RDONLY))<0)
{perror("open"); // in caso che il file da ordinare non esista
 printf("Qualsiasi tasto per procedere, EOF per fine: ");
 continue;
}
printf("Nome del file ordinato: ");
if (gets(nome_dest)==0) break; // Creazione file ordinato
if((fd_dest=open(nome_dest, O_WRONLY|O_CREAT, 0644))<0)
{ perror("open");
 printf("Qualsiasi tasto per procedere, EOF per fine:" ");
 continue;
}
sd=socket(AF_INET, SOCK_STREAM, 0); // Creazione socket
if(sd<0) {perror("apertura socket"); exit(1);}
printf("Client: creata la socket sd=%d\n", sd);
if(connect(sd, (struct sockaddr *) &servaddr,
 sizeof(struct sockaddr))<0)
{ perror("connect"); exit(1);}
// BIND implicita e controllo di errore possibile per ogni primitiva
```

REMOTESORTCLIENT 4/4

// Invio e ricezione file

```
while ( (nread=read(fd_sorg, buff, DIM_BUFF)) > 0)
{
    write(1, buff, nread);           // Stampa su console
    write(sd, buff, nread);          // Invio
}
```

```
shutdown(sd, 1);
```

```
while ( (nread=read(sd, buff, DIM_BUFF)) > 0)
```

```
{
    write(fd_dest, buff, nread);
    write(1, buff, nread);
}
```

```
shutdown(sd, 0);
```

```
close(fd_sorg); close(fd_dest); close(sd);
```

```
printf("Nome del file da ordinare, EOF per  
terminare:");
```

```
}
```

```
exit(0);
```

```
}
```


REMOTESORTSERVER 1/4

```
#include <stdio.h>

...
/*****/
void gestore(int signo)
{
    int stato;
    printf("esecuzione gestore di SIGCHLD\n");
    wait(&stato);
}
/*****/

int main(int argc, char **argv)
{ int  listen_sd, conn_sd;
  int port, len; const int on = 1;
  struct sockaddr_in cliaddr, servaddr;
  struct hostent *host;
  if(argc!=2) // Controllo argomenti
  { printf("Error: %s port\n", argv[0]); exit(1); }
  else port = atoi(argv[1]); // Verifica intero ... Controllo porta????
```

REMOTESORTSERVER 2/4

// Inizializzazione indirizzo server

```
memset ((char *)&servaddr, 0, sizeof(servaddr));  
servaddr.sin_family = AF_INET;  
servaddr.sin_addr.s_addr = INADDR_ANY;  
servaddr.sin_port = htons(port);
```

// Creazione, bind e settaggio opzioni socket ascolto

```
listen_sd=socket(AF_INET, SOCK_STREAM, 0);  
if(listen_sd <0)  
{perror("creazione socket "); exit(1);}  
if(setsockopt(listen_sd, SOL_SOCKET, SO_REUSEADDR,  
              &on, sizeof(on))<0) {perror("..."); exit(1);}  
printf("Server: set ok\n");  
if(bind(listen_sd, (struct sockaddr *) &servaddr,  
        sizeof(servaddr))<0)  
{perror("bind socket d'ascolto"); exit(1);}  
printf("Server: bind socket d'ascolto ok\n");  
if (listen(listen_sd, 5)<0) // Creazione coda d'ascolto  
{perror("listen"); exit(1);}
```

REMOTESORTSERVER 3/4

/* Aggancio gestore per evitare figli zombie. Quali altre primitive potrei usare?

* E' portabile su tutti i sistemi? Pregi/Difetti? */

signal(SIGCHLD, gestore);

for(;;) // Ciclo di ricezione richieste

{ **if((conn_sd = accept(listen_sd,**
(struct sockaddr *)&cliaddr,&len)) < 0)

{

/* La accept puo' essere interrotta dai segnali inviati dai figli alla loro teminazione.

* Tale situazione va gestita opportunamente.

Vedere nel man a cosa corrisponde la costante EINTR! */

if (errno==EINTR)

{ **perror("Forzo la continuazione della accept");**
continue;

}

else exit(1);

}

REMOTESORTSERVER 4/4

```
if (fork()==0)    // Figlio
{
    // Chiusura file descriptor non utilizzati e ridirezione di stdin e stdout
    close (listen_sd);
    close(1); close(0);
    dup(conn_sd); dup(conn_sd); close(conn_sd);
    // Esecuzione ordinamento
    execl("/bin/sort", "sort", (char *)0);
}
// come se ci fosse un: else
// PADRE: chiusura socket di connessione (NON di ascolto)
close(conn_sd);
}
}
```