Piccolo recap

Nella lezione precedente abbiamo le API di programmazione basata su socket (in C), ed in particolare la comunicazione basata su datagram. Quando usiamo l'architettura TCP/IP il protocollo a livello di trasporto che impieghiamo è il protocollo UDP.

In questa lezione sono contenute delle "esercitazioni" riguardanti gli strumenti di cui abbiamo parlato nelle lezioni precedenti.

E' importante utilizzare una macchina basata sul sistema UNIX, o in alternativa usare cygwin.

Esercitazione 2020-10-6: Cliet-Server

Scrittura del codice

Per questa esercitazione utilizzeremo il codice visto nella lezione 5, quindi ci basterà copiarlo all'interno dei due file `client.c` e `server.c`.

Compilazione del codice

Per compilare il codice possiamo usare un ambiente di sviluppo come **codeblocks o devcpp**, ma in questa esercitazione verrà utilizzata esclusivamente la console MacOs (UNIX like).

Per compilare il codice ci spostiamo nella dir contenente i files:

```
cd /root/path/to/files
```

Successivamente compiliamo con:

```
gcc -o server server.c
gcc -o client client.c
```

Questi comandi ci creeranno dei files chiamati `server` e `client`, privi di estensione; questo perchè sui sistemi unix l'estensione non indica l'eseguibilità di un file, ma possiamo controllarla con il comando:

```
ls -la
```

Ottenendo:

Se notiamo i diritti, le 'x' ci dicono che il file in questione è **eseguibile**; ad esempio server.c **non è eseguibile**, mentre *server* si.

Avviare server e client

Per avviare il server ed il client su UNIX scriviamo nel terminale:

```
./server
./client
```

All'avvio del server il terminale va in attesa (processo padre-figlio visto nell'esame di architettura dei calcolatori):

Nel momento in cui avviamo il client (**Attenzione! dobbiamo passare come argomento l'indirizzo IP del server; siccome il server viene eseguito sulla stessa macchina del client ci basterà passare come indirizzo l'indirizzo di loopback: 127.0.0.1**) otteniamo:

Cattura dei pacchetti con WireShark

Per catturare i pacchetti con wireshark ci basterà selezionare l'interfaccia di **loopback** (siccome stiamo eseguendo client e server sulla stessa macchina) e filtrare i pacchetti ad **udp port 5193**, siccome il nostro servizio utilizza questa specifica porta:

> Attenzione: per far funzionare il filtro bisogna **prima** inserire il filtro, e **poi** selezionare la rete.

Siamo quindi in ascolto; per il momento non viene visualizzato nulla perche ne server ne client sono stati avviati:

Nonappena avviamo il server con `./server` e lanciamo il client con `./client` visualizziamo:

Se guardiamo la colonna delle **info** di wireshark, possiamo notare come i nostri due pacchetti sono stati correttamente inviati:

Richiesta

La richiesta è effettuata dal client, infatti notiamo **57269 -> 5193**. La porta 57269 fa parte del range delle porte utilizzabili per scopi privati, infatti il nostro client viene mappato **dinamicamente** su questa specifica porta; la porta 5193, invece, è la porta che **abbiamo definito** con il server, infatti esso è in ascolto proprio su questa porta.

Risposta

Siccome il server è in ascolto su questa specifica porta, nonappena il client invia un messaggio il server lo riceve ed elabora la risposta. Notiamo infatti **5193 -> 57269**, che ci dice che il server ha risposto.

Il messaggio

Se clicchiamo sul messaggio inviato dal server al client notiamo:

Questa lunga stringa non è altro che la codifica ASCII del nostro messaggio.

Modello di comunicazione orientato ai flussi - Stream

Ci soffermiamo sul modello orientato ai flussi, che vede a livello di trasporto l'impiego di un altro protocollo (rispetto a quello visto finora - UDP): il protocollo **TCP**. Questo modello è più complesso sia a livello di programmazione che a livello di protocollo stesso; quando scenderemo al livello di trasporto, questo protocollo è molto più articolato rispetto all'UDP; questo perchè mette a disposizione molti più meccanismi e servizi.

Anche in questo caso procederemo *by example*: a differenza di quanto visto nella scorsa lezione, dove abbiamo considerato come applicazione l'applicazione DNS (e quindi il protocollo e porta che il servizio utilizza), questa volta consideriamo come applicazione di esempio, il **web**. Immaginiamo quindi di voler creare un server e client per questo tipo di applicazione.

Per quanto riguarda il server ipotizziamo di costruire la parte strutturale di un **server web**, ed il servizio specifico prevede l'utilizzo del numero di porta **80**. Questo numero di porta è quello che viene impiegato di default.
Il protocollo di trasporto utilizzato è il protocollo **TCP**.

socket()

A differenza di quanto visto nella scorsa lezione, l'invocazione della socket() prevede come secondo parametro non più la costante **SOCK_DGRAM**, ma la costante **SOCK_STREAM**; cambia quindi il secondo parametro della funzione socket.

Ricordiamo che la funzione socket() ci consente di richiedere la creazione di un endpoint di comunicazione per il modello di comunicazione ordinato allo stream.

Lo "0" anche in questo caso sta ad indicare che vogliamo usare il protocollo di default, in questo caso TCP.

```
int fd;

if((fd = socket(PF_INET, SOCK_STREAM, 0)) < 0){
  perror("socket");
  exit(1);
}
```

> La creazione della socket è molto simile al protocollo UDP.

bind()

Anche in questo caso è necessario invocare la funzione bind(). Dobbiamo fare in modo che la socket sia contattabile da remoto, ed andiamo ad operare con la stessa struttura (c) che ci consente di gestire gli indirizzi di trasporto dell'architettura TCP/IP.

```
int f;
struct sockaddr_in srv;

// creazione della socket
// inizializzazione  dell'indirizzo socket:
srv.sin_family = AF_INET;
srv.sin_port = htons(80); // porta di default usata in TCP
srv.sin_addr.s_addr = htonl(INADDR_ANY);

if(bind(fd, (struct sockaddr*) &srv, sizeof(srv)) < 0){
  perror("bind");
  exit(1);
}
```

> anche in questo caso la bind è molto simile al protocollo UDP.
>
> C'è da dire che il client, a questo stato, non può ancora contattare il server.

La socket, così inizializzata, pur essendo identificabile da remoto, non è ancora impiegabile dalla comunicazione. Nel caso delle comunicazioni orientate ai datagram, subito dopo la bind() abbiamo l'invocazione della **recvfrom()**, che ci consente di ricevere subito un messaggio.

In questo caso, però, dobbiamo utilizzare la funzione **listen()**:

listen()

Questa funzione è usata solo da lato server, quindi la API che stiamo analizzando è **asimmetrica**, quindi da lato server si programma in modo diverso da come si programma lato client.

Solo a seguito dell'invocazione di questa funzione, il server è pronto a ricevere, **non messaggi, ma richieste di connessione.** Questo eprchè il modello orientato ai flussi prevede che la comunicazione sia anche orientata alle **connessioni**; non è quindi possibile avviare una trasmissione senza aver prima stabilito una connessione.

In particolare, con l'invocazione di questa funzione lato server, la socket deve predisporsi all'ascolto, e la stiamo quindi **caratterizzando come socket di tipo server**, e che questa socke deve prevedere una **coda per gestire le richieste di connessioni** in arrivo dai client.

```
int fd;
struct sockaddr_in srv;
// creazione socket
// bind della socket ad una porta

if(listen(fd,5) < 0){
  perror("listen");
  exit(1);
}
```

> 5 è la dimensione della coda associata alla socket server, per ospitare richieste di connessione; non è possibile mantenere nella coda **più di 5 richieste** non servite.
>
> Per evitare che le richieste di connessione vengano rifiutate, è necessario servire, e quindi svuotare la coda.

accept()

Questa funzione che viene invocata subito dopo la listen(), consente di estrarre una richiesta di connessione (se presente) dalla coda associata alla socket server, configurata con la funzione listen().

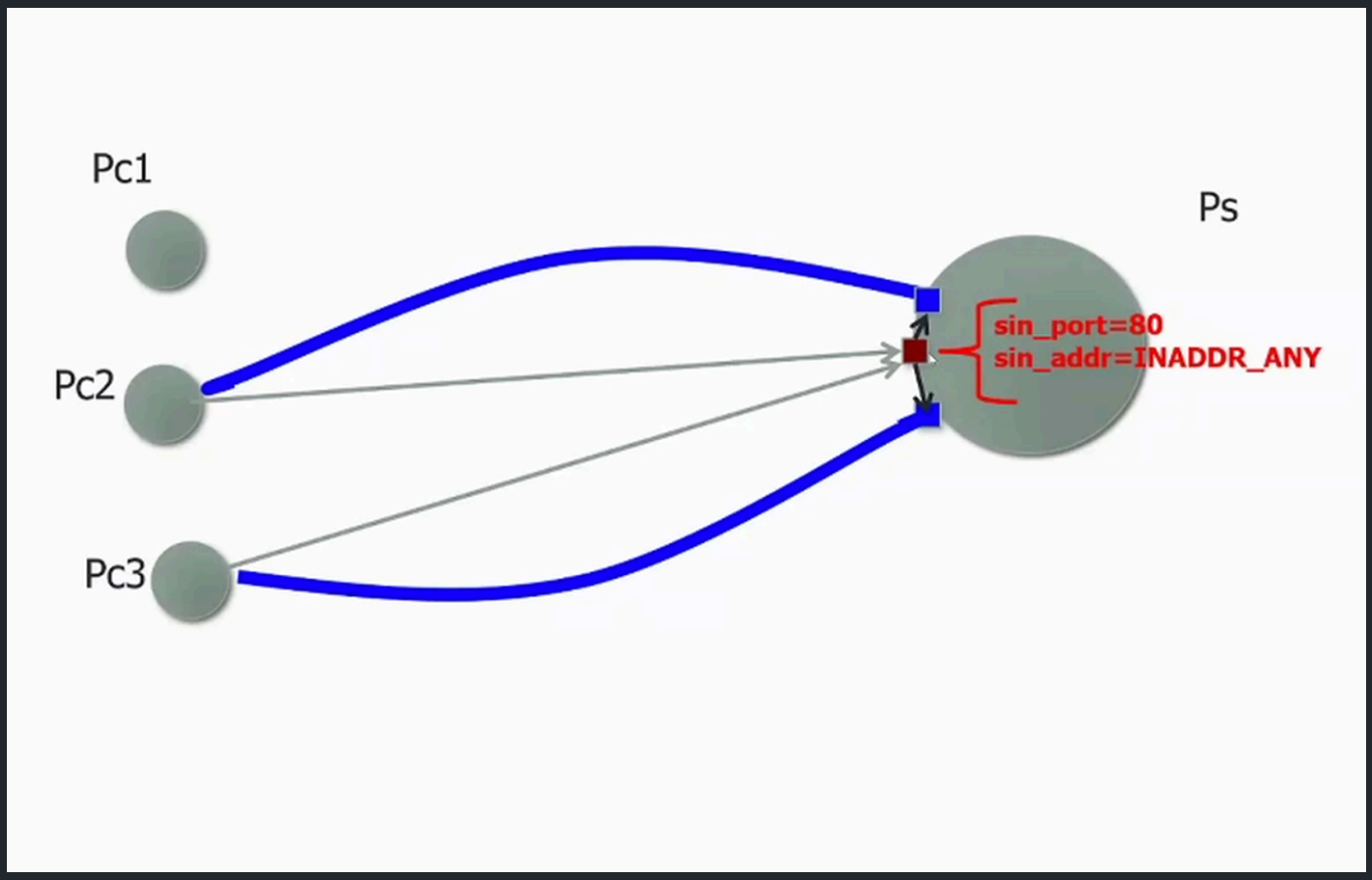
Quindi, accept() è una **funzione bloccante come la recvfrom()** usata nel protocollo UDP; questo vuol dire che se non ci sono richieste all'interno della coda, il processo viene sospeso, visto che non può completare la richiesta di connessione.

Se invece arriva una richiesta di connessione da un client, accept permette di gestire la richiesta:

```
int fd;
struct sockaddr_in srv;

struct sockaddr_in cli;
int cli_len = sizeof(cli);
int newfd;

// creazione socket
// bind della socket ad una porta
// invocazione della listen sulla porta

newfd = accept(fd, (struct sockaddr*) &cli, &cli_len);
if(newfd < 0){
  perror("accept");
  exit(1);
}
```

Accept() restituisce un intero che è il descrittore di una nuova socket; ancora una volta la socket verrà usata per la comunicazione verso **lo specifico client che ha richiesto la connessione**.

A differenza di ciò che accade con l'altro modello, la comunicazione in questo caso, tra socket, è detta **connessa**, proprio perchè le socket possono "parlare" con un solo interlocutore.



L'endpoint in rosso scuro, è la rappresentazione della socket iniziale, a cui abbiaomo associato l'indirizzo di trasporto INADDR_ANY:80. Quando un client effettua una connessione, o chiede di usare quella socket, viene creato un altro endpoint, restituito dalla funzione accept. Questo endpoint è quello che verrà effettivamente utilizzato per la conversazione.

Le socket in blu, sono "figlie" della socket iniziale, quindi avranno, lato server, lo stesso numero di porta.

Quindi

Capiamo quindi che la socket che creiamo inizialmetne è solo utilizzata per stabilire la connessione, mentre quelal che verrà usata effettivamente per la comunicazione sarà un'ulteriore socket creata nel momento in cui il server gestisce una richiesta.

Il secondo parametro della funzione accept(), che in questo caso è `(struct sockaddr*) &cli` è un parametro di uscita che consente di ospitare l'indirizzo di trasporto del **client che ha fatto richiesta di connessione**. Questo indirizzo di trasporto (presente anche nella recvfrom) non è particolarmente importante; questo perchè una volta servita quella richiesta di connessione, il server tornerà ad eseguire funzioni di accept(). Ad ogni invocazione viene restituito un descrittore diverso, che consentirà la comunicazione con un client diverso.

🏁 1:01

read()

Dopo l'accettazione della connessione, operiamo l'operazione di lettura; questa operazione non va fatta su fd, ovvero la socket iniziale, ma su **newfd**, ovvero la nuova socket creata nel momento in cui il server ha accettato la comunicazione.

In questa variabile abbiamo il descrittore di socket che il server usa per comunicare con uno specifico client; non abbiamo bisogno di sapere qual è il suo indirizzo di trasporto:

```
int fd;
struct sockaddr_in srv;
struct sockaddr_in cli;
int cli_len;
int newfd;
```

```
char buf[512];
int nbytes;

// creazione della socket
// binding della socket ad una porta
// ascolto sulla socket (iniziale)
// accettazione della connessione - creazione di una nuova socket

if ((nbytes = read (newfd, buf, sizeof(buf))) < 0){
  perror("read");
  exit(1);
}
```

Il comportamento è ovviamente diverso, siccome il modello utilizzato in questo caso **è orientato ai flussi**, e voglaimo quindi leggere dal canale in maniera **stream oriented**. Una volta costruito il canale potremo estrarre **con continuità** blocchi di byte.

Non estraiamo quindi più un **datagram**, ma un blocco di byte appartenente ad una **sequenza di byte** che sarà terminato da un *fine stringa*; finchè non arriva il delimitatore, è possibile estrarre blocchi di byte dal canale.

La funzione read() ci permette di leggere dalla socket specificata, e quindi portare dall'area di memoria indicata (buf) un blocco **al più pari a** sizeof(buf).
Sul canale potrebbero essere inviati anche un milione di byte, ma verrà estratto sempre solo il blocco di byte specificato (da sizeof(buf)).

Quando invochiamo la funzione read(), restituisce un intero, che rappresenta **il numero di byte effettivamente letti**. Questo intero è particolarmente importante, perchè possiamo fare un controllo per vedere se l'operazione è andata a buon fine. Non è detto quindi che vengano estratti tutti i byte specificati (al più).
Quindi, i byte che dobbiamo ritenere significativi, sono quelli che la funzione ci

"dirà" di aver letto.
Guardando il codice, questo valore viene salvato all'interno della variabile `nbytes`, che viene usata dalla read().

Un client per la comunicazione Stream

Anche lato client troviamo le funzioni che abbiamo già visto, come la creazione iniziale della socket iniziale, invocando la funzione **socket()**. Prepariamo quindi un indirizzo di trasporto, che non è quello che vogliamo assegnare alla socket client, ma prepariamo un indirizzo di trasporto per poter **contattare il server**, infatti il numero di porta che usiamo è il numero 80.

```
int fd;
struct sockaddr_in srv;

// creazione della socket
// connessione: specifichiamo la famiglia dell'indirizzo
srv.sin_family = AF_INET;

// connessione: specifichiamo il numero di porta
srv.sin_port = htons(80);

// connessione: specifichiamo l'indirizzo IP da contattare
srv.sin_addr.s_addr = inet_addr("128.2.35.50");

if(connect(fd, (struct sockaddr*) &srv, sizeof(srv)) < 0){
  perror("connect");
  exit(1);
}
```

Il client, per poter dialogare con un server, deve chiedere di **attivare una connessione**; specifichiamo nella connect l'indirizzo di trasporto del server. Facendo questa operazione, stiamo dicendo che **vogliamo creare una socket fd connessa con la socket remota, il cui indirizzo di trasporto è in srv.** Stiamo completando l'inizializzazione della socket lato client, dicendo che la socket deve essere accoppiata ad una specifica socket.

Quando viene invocata la funzione connect() lato client, viene realizzata questa interazione:

Questa produce un messaeggio di richiesta di connessione, che sarà ospitato nella coda delle richieste lato server. Questo messaggio di richiesta verrà poi accettato con la funzione accept().

Se la connessione va a buon fine, sarà possibile avviare il dialogo; per dialocare il client esegue un'operazione di scrittura:

Write()

```
int fd;
struct sockaddr_in srv;
char buf[512];
int nbytes;

// creazione della socket
// connessione della socket locale con la socket del server

if((nbytes = write(fd, buf, sizeof(buf))) < 0){
  perror("write");
  exit();
}
```

La write è l'operazione **duale all'operazione di lettura lato server (read())**, che ha la stessa struttura; anche in questo caso abbiamo come **valore di ritorno** il numero di bytes effettivamente trasmessi. Anche lato client è utile sapere quanti bytes sono stati trasmessi, proprio per essere sicuri di aver inviato il giusto numero di dati.

Comunicazione point to point

Dire che una socket è connessa, significa dire che tra le informazioni che la caratterizzano, non troviamo solo l'indirizzo di trasporto di **una delle due entità**, ma troviamo quello di **entrambe le entità**.

Nell'immagine abbiamo due tipi di socket:

- in ROSSO abbiamo la socket non connessa, ovvero la socket iniziale usata lato server.
- in BLU troviamo la socket che viene creata restituita dal server, che usiamo effettivamente per la comunicazione; questa è la socket che lato client viene inizializzata a seguito dell'esecuzione della funzione connect().

Lato server, la **socket di ascolto** (iniziale), o **socket di benvenuto**, contiene solo l'indirizzo di trasporto del server; questo perchè ovviamente il server non può (solitametne) conoscere gli indirizzi dei client.

Quando viene invece creata la socket connessa, quella restituita dall'accept(), la socket ha delle informazioni supplementari:

- indirizzo di trasporto del server (locale scelto dal server)
- indirizzo di trasporto del client

Il canale di comunicazione che viene realizzato tra le due socket connesse è **affidabile e FIFO**; è affidabile perchè il protocollo utilizzato, essendo TCP, fornisce affidabilità, ed è FIFO perchè il protocollo TCP, anche su una rete che non prevede ordinamento, attraverso dei meccanismi basati sui **numeri di sequenza** ordina i blocchi di byte che arrivano a destinazione, in modo da preservare l'ordine di presentazione.

Questo è importante nel momento in cui vogliamo trasferire un file, questo perchè l'ordine di ricezione dei blocchi **deve essere lo stesso** dell'ordine di trasmissione.

Riassuntazzo

Riepilogando, le funzioni sono maggiori rispetto a quelle usate con il protocollo UDP, e sono anche impiegate in maniera asimmetrica.

Server

Per scrivere il server (codice) dobbiamo invocare una **socket()**, invocare una **bind()** per associare a quella socket un indirizzo di trasporto che sarà usato dal client per il primo contatto, invochiamo **listen()** per associare alla socket una coda di ascolto di richieste, invochiamo finalmente **accept()**, che ci consente di estrarre le richieste di connessione dalla prima socket, e di creare una nuova socket **connessa**; da questo momento in poi la comunicazione avverrà sulla nuova socket.

La prima socket di ascolto, viene chiusa solo nel momento in cui il processo del server viene terminato.

Client

Lato client abbiamo quindi la prima creazione della socket, successivamente l'invocazione della **connect()**, che va invocata **prima del dialogo** per stabilire la connessione, che interagisce con la funzione **accept()** lato server. Dopo aver stabilito la connessione viene invocata la **write()** che ci permette di scrivere i dati che saranno letti dal server con la **read()**. Il server, dopo aver letto, invia una risposta al client con una write(), che il client leggerà con una read() speculare.

🏁 1:30

Esempio

Vediamo come l'esempio realizzato nella scorsa esercitazione, potrebbe essere scritto mediante le socket orientate ai flussi. Dovremo quindi realizzare un client che contatta un server; questo server risponde al client il numero delle volte che il server è stato contattato.

Programmi

Useremo anche in questo caso due files:

- **server.c** : viene eseguito in background; il numero di porta che il server ascolta è la 5193.
- **client.c** : il client viene eseguito come programma utente, utilizza la medesima porta 5193.

server.c

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/types.h>
```

```c
#include <fcntl.h>

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include <fcntl.h> // for open
#include <unistd.h> // for close

#define PROTOPORT 5193  // numero di porta
#define QLEN 6  // massimo numero di richieste attive

int visits = 0;

int main(int argc, char* argv[]){
  struct sockaddr_in sad;
  struct sockaddr_in cad;

  int sd, sd2;
  int port;
  int cadlen;
  char buf[1000];
```

> Nella parte preliminare del programma abbiamo l'inclusione delle varie librerie che useremo, della dichiarazione dei vari indirizzi di trasporto che useremo e di alcune variabili.

```
memset((char*) &sad, 0, sizeof(sad)); //puliamo sad
sad.sin_family = AF_INET;
sad.sin_addr.s_addr = htonl(INADDR_ANY);

// se non viene specificata la porta come parametro, viene usata la porta hardwired in
PROTOPORT (5193)
if(argc > 1){
  port = atoi(argv[1]);
}else{
  port = PROTOPORT;
}

sad.sin_port = htons((u_short) port);
```

> Anche in questa porzione di codice abbiamo una sorta di fase preliminare, dove viene pulito l'indirizzo di trasporto che useremo per mettere in ascolto la socket.

```
// creiamo la socket
sd = socket(PF_INET, SOCK_STREAM, 0);
if(sd < 0){
  fprintf(stderr, "creazione della socket fallita");
  exit(1);
}
```

> In questa porzione di codice stiamo creando la socket su cui verrà messo in ascolto il server

```
// binding della socket all'indirizzo
if(bind(sd, (struct sockaddr*) &sad, sizeof(sad)) < 0){
  fprintf(stderr, "errore nella bind");
  exit(1);
}
```

> dopo la creazione della socket effettuiamo il binding di essa con l'indirizzo di trasporto.

```
// specifichiamo la dimensione della coda che ospita le richieste di connessione.
if(listen(sd, QLEN) < 0){
  fprintf(stderr, "errore nella listen");
  exit(1);
}
```

> attiviamo finalmente la listen che ci permette di servire le richieste

Terminata la fase preliminare, entriamo nella porzione di codice più interessante del server:

```
while(1){
  alen = sizeof(cad);
  if((sd2 = accept(sd, (struct sockaddr*) &cad, &cadlen)) < 0){
    fprintf(stderr, "errore nell'accept");
    exit(1);
  }

  visits++;

  sprintf(buf, "questo server è stato contattato %d volte\n", visits);
  write(sd2, buf, strlen(buf));
  close(sd2);
 }
}
```

> IMPORTANTE! E' la close() invocata su sd2 a segnalare al client che la trasmissione è completa.

Abbiamo un loop while, che ci indica che il server è sempre in ascolto. Ci aspettiamo di avere la ricezione del messaggio e poi la scrittura del messaggio di risposta al client.

A differenza dell'altro modello, la prima operazione che dobbiamo prevedere in questo caso è proprio **l'accept()**, perchè dobbiamo accettare le richieste di connessione proposte dai client. Subito dopo l'accept() dobbiamo prevedere la lettura, quindi una **write()**, proprio perchè dobbiamo servire le richieste.

In questo server specifico non è stata inserita un'operazione di lettura, proprio perchè il messaggio del client sarà probabilmente vuoto, visto che non c'è bisogno di leggere nulla; il server conta solo le richieste di connessione.

Client.c

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <fcntl.h>

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include <fcntl.h> // for open
#include <unistd.h> // for close

#define PROTOPORT 5193  // numero di porta
```

> Vari Defines ed include

```
char localhost[] = "127.0.0.1";
char *host;

int main(int argc, int *argv[]){
  struct sockaddr_in sad;
  int sd;
  int port;
  int n;
  char buf[1000];
```

> la preparazione del main è la medesima del servere

```
 memset((char*) &sad, 0, sizeof(sad));
 sad.sin_family = AF_INET;

 if(argc > 2){
   port = atoi(argv[2]);
 }else{
   port = PROTOPORT;
 }

 if(port > 0){
   sad.sin_port = htons((u_short) port);
 }else{
   fprintf(stderr, "bad port number");
   exit(1);
 }
```

> leggiamo il numero di porta o come argomento o le assegnamo un valore hardwired

```
// se un arg1 è esistente
if(argc > 1){
  host = argv[1];
}else{
  host = localhost;
}
```

> se è fornito un indirizzo lo assegnamo, altrimenti assegnamo localhost - loopback

```
// creiamo una socket
sd = socket(PF_INET, SOCK_STREAM, 0);
if(sd < 0){
  fprintf(stderr, "creazione della socket fallita");
  exit(1);
}
```

> creiamo la socket di contatto

```
if(connect(sd, (struct sockaddr*) &sad, sizeof(sad)) < 0){
  fprintf(stderr, "connessione fallita");
  exit(1);
}

// IMPORTANTE! il client non invia nessun messaggio, basta inviare un messaggio di richiesta

n = read(sd, buf, sizeof(buf));
while(n > 0){
  write(1, buf, n);
  n = read(sd, buf, sizeof(buf));
}

// chiudiamo la socket
close(sd);
exit(0);
```

Per invocare la connect() dobbiamo aver preparato l'indirizzo di trasporto, che abbiamo fatto precedentemente; troviamo quindi il descrittore della socket che abbiamo appena creato (sd) e l'indirizzo di trasporto del server che vogliamo contattare (sad).

Una volta invocata la connect, attraverso sd possiamo usare la socket connessa e scrivere, ma in questa applicazione non è necessario, visto che al server basta il contatto da parte del client; non a caso il server non implementa una lettura.

Operazione read()

L'operazione di read() viene realizzata in un modo particolare: leggiamo dal canale di comunicazione un blocco di byte di dimesione massima buf. Poiché stiamo leggendo a stream, e siccome non è detto che abbiamo letto tutta la stringa con un'unica operazione di lettura, in questo caso dobbiamo prevedere il ciclo:

```
n = read(sd, buf, sizeof(buf));
while(n > 0){
  write(1, buf, n);
  n = read(sd, buf, sizeof(buf));
}
```

Questo ciclo termina solo quando i byte letti dal canale di comunicazione non sono zero, ovvero legge finchè c'è qualcosa da leggere. Questo accade perchè se chi spedisce utilizza un buffer più grande rispetto a chi legge, non si riuscirebbe a leggere tutto il messaggio in un unico colpo.

Tornando al codice, viene eseguita un'operazione di scrittura `write(1, buf, n)`: stiamo dicendo che il contenuto del buffer sarà riversato nello standard out (indicato da 1); 'n' ci dice il numero di byte da scrivere.

Inoltre, scriviamo subito sullo stdout in modo da liberarci subito dei dati appena letti, in modo da liberare il buffer per un nuovo blocco di dati, senza dover salvare nessun tipo di dato.

La lettura si conclude nel momento in cui viene invocata la close(sd2) da parte di chi invia i dati (in questo caso del server), che ci fa capire che la trasmissione è finita; il valore inviato è **l'End of Stream**.

---

fine lezione 6