Laboratorio di Reti di Calcolatori

Funzioni utili, server ricorsivi, echo client ed echo server.

Paolo D'Arco

Abstract

Scopo della lezione è presentare alcune funzioni di utilità generale (e.g., funzioni per le conversioni degli indirizzi IP da formato presentazione a formato network e viceversa, per le conversioni di numeri da rappresentazione host a rappresentazione network e viceversa, per la lettura e la scrittura su socket, per la manipolazione di byte ...), introdurre i server ricorsivi, discutendone i vantaggi rispetto ai server iterativi, e descrivere un nuovo esempio di comunicazione client-server.

1 Funzioni utili

La scrittura di applicazioni client-server può essere semplificata usando alcune funzioni che risolvono problemi tipici, quali per esempio i passaggi da rappresentazione host a rappresentazione network e viceversa, l'azzeramento di un'area di memoria, la comparazione di due aree di memoria bit a bit, la lettura di stringhe da socket etc ... Alcune di queste funzioni sono già apparse negli esempi precedenti.

1.1 Funzioni per la conversione di formato

#include <sys/socket.h> int *inet_pton*(int *af*, const char* *src*, void* *dest*); valore di ritorno: ≤ 0 se errore 1 altrimenti

Come accennavamo la volta scorsa, gli indirizzi IP possono essere rappresentati sia in decimale che in binario. Quando rappresentati in decimale, cioè come sequenze di quattro byte espressi in decimale, separati da un punto, tipo "192.41.218.1", vengono anche detti in *formato presentazione*, mentre, quando rappresentati come stringhe binarie, tipo 11000000001010011100110100000001, vengono detti in *formato network*. La prima forma è più confortevole per gli esseri umani. La rete utilizza la seconda. La funzione *inet_pton()* trasforma un indirizzo IP in formato presentazione in formato network. Un'invocazione tipica, usata nell'esempio della scorsa lezione, è

inet_pton(AF_INET, argv[1], &servaddr.sin_addr)

Il primo parametro specifica la famiglia di protocolli (per noi sempre AF_INET), il secondo l'indirizzo IP in formato presentazione (in questo caso passato dall'utente da linea di comando), il terzo una

struttura di tipo *in_addr*, in cui memorizzare l'indirizzo in formato network. Si ricordi che l'indirizzo IP è rappresentato tramite una struttura di tipo in_addr. In questo caso abbiamo in precedenza definito *servaddr* come un'istanza di struttura di tipo sockaddr_in, e stiamo indicando il campo sin_addr, che è appunto una struttura di tipo in_addr.

La conversione inversa, viene invece realizzata tramite:

```
#include <sys/socket.h>

const char*
inet_ntop(int af, const void *src, char * dst, socklen_t size);

valore di ritorno: NULL se errore
                   l'indirizzo IP in formato presentazione, altrimenti
```

Un'invocazione tipica è

```
inet_ntop(AF_INET, &cliaddr.sin_addr, strptr, INET_ADDRSTRLEN)
```

Il primo parametro specifica la famiglia di protocolli (per noi sempre AF_INET), il secondo l'indirizzo IP in formato rete (in questo caso abbiamo in precedenza definito *cliaddr* come un'istanza di struttura di tipo sockaddr_in, e stiamo indicando il campo sin_addr, che è appunto una struttura di tipo in_addr), il terzo *strptr* è un vettore di caratteri di lunghezza INET_ADDRSTRLEN, in grado di contenere un indirizzo IP in formato presentazione. La costante INET_ADDRSTRLEN è definita nel file header *sys/socket.h* e rappresenta la lunghezza di un indirizzo IP in formato presentazione. Altre funzioni utili per le conversioni di formato sono

```
#include <arpa/inet.h>

uint32_t htonl(uint32_t hostlong);

uint16_t htons(uint16_t hostshort);

uint32_t ntohl(uint32_t netlong);

uint16_t ntohs(uint16_t netshort);

valore di ritorno: < 0 se errore
                   valore in formato host/network, altrimenti
```

Queste funzioni convertono rispettivamente, un *long* da formato host a network, uno *short* da formato host a network, un *long* da formato network a host, e uno *short* da formato network a host. Le due funzioni che trattano il tipo short sono utili per convertire in numeri di porta da una rappresentazione all'altra.

1.2 Funzioni per leggere e scrivere sui socket

Tre funzioni utili a leggere e scrivere dati su un socket, definite nel nostro file di funzioni *fun-corso-reti.c*, sono:

```
#include "basic.h"

ssize_t reti_readline(int fd, void *vptr, size_t maxlen);

valore di ritorno: < 0 se errore
                   il numero di byte letti, altrimenti
```

Un' invocazione tipica di questa funzione è

reti_readline(*sockfd*, *recvline*, MAXLINE);

La funzione prende in input un socket descriptor *sockfd* e un vettore di caratteri *recvline* di lunghezza MAXLINE (che conterrà la riga letta). Nota che la funzione legge *fino a quando non incontra il carattere di termine riga* \n. Memorizza la riga in *recvline* come stringa, i.e. termina con \n \0.

```
#include "basic.h"

ssize_t reti_writen(int fd, const void *vptr, size_t n);

valore di ritorno: < 0 se errore
                   il numero di byte scritti, altrimenti
```

Analogamente, la funzione *reti_writen*(), scrive n byte sul descrittore *fd*. Riprova fino a quando i dati non vengono effettivamenti scritti. Un' invocazione tipica di questa funzione è

reti_writen(sockfd, sndline, strlen(sndline));

In questo esempio, la funzione prende in input un socket descriptor *sockfd*, una stringa e la sua lunghezza. In modo simile, la funzione *reti_readn* legge esattamente n byte dal descrittore *fd*.

```
#include "basic.h"

ssize_t reti_readn(int fd, void *vptr, size_t n);

valore di ritorno: < 0 se errore
                   il numero di byte letti, altrimenti
```

Infine, funzioni utili per manipolare byte sono

```
#include <strings.h>

void bzero(void *dest, size_t nbytes);

void bcopy(const void *src, void *dest, size_t nbytes);

int bcmp(const void *ptr1, const void *ptr2, size_t nbytes);
valore di ritorno: 0 se uguali
                   diverso da zero se le stringhe sono diverse
```

La funzione *bzero*(), azzera l'area di memoria **dest* di taglia *nbytes*. La funzione *bcopy*() copia *nbytes* da **src* in **dest*. Le due stringhe possono anche sovrapporsi. La funzione *bcmp*() compara due stringhe di lunghezza *nbytes* e dà zero se le due stringhe sono uguali.

1.3 Server Ricorsivi

Nell'esempio di comunicazione client-server visto nella lezione precedente, il server opera essenzialmente come segue

- Si predispone ad offrire i propri servizi invocando le funzioni (*socket(), bind(), listen()).*
- Si pone in attesa di richieste di servizio (connessione) invocando la funzione *accept*().
- All'arrivo di una richiesta, riprende le computazioni, comunica (attraverso il *nuovo socket* che la *accept*() gli ha restituito) con il client, e al termine dell'interazione torna in attesa di nuove richeste di servizio.

Un server organizzato in questo modo si dice *iterativo*. Il server serve *una sola* richiesta alla volta. E' facile intuire che per un semplice server quale *daytimesrv*, visto che le computazioni sono immediate e l'interazione brevissima (uno scambio di messaggi) una organizzazione del genere può anche esser ammissibile.

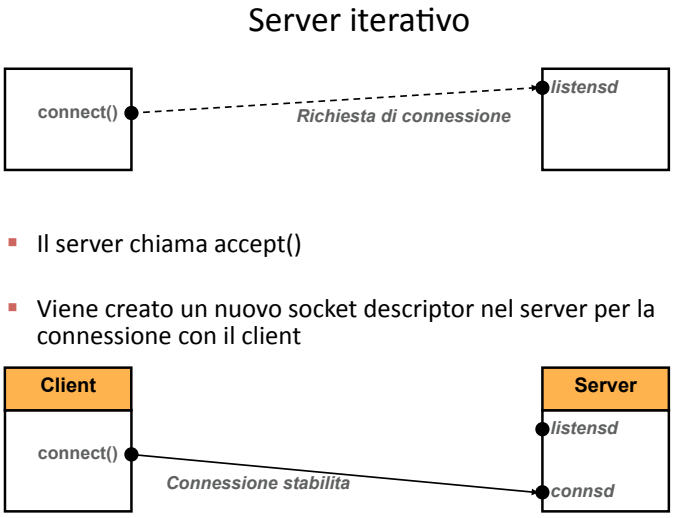


Figure 1: Server Iterativo

Se, invece, le richieste sono frequenti e molteplici, le computazioni più lunghe, e le interazioni più complesse, occorre una diversa strutturazione del server.

Un server *ricorsivo* genera un processo figlio per ogni nuovo client. Il processo padre accetta semplicemente le richieste di servizio. Ad ogni nuova richiesta, crea un processo figlio che gestisce concretamente la richiesta.

In termini di codice, il nostro server verrà modificato come segue

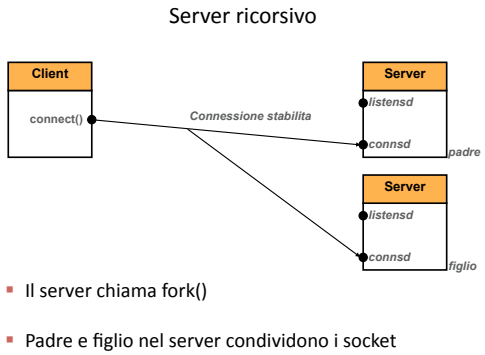


Figure 2: Il padre invoca fork().

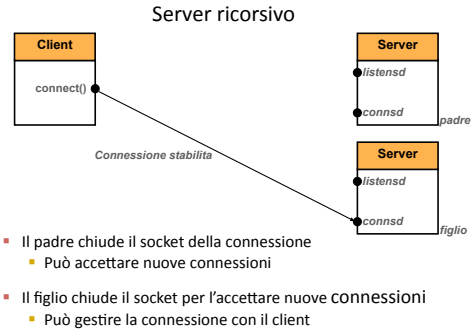


Figure 3: Il figlio gestisce il client.

```
1. #include        "basic.h"

3. int main(int argc, char **argv) {
4.         int                      listenfd, connfd, n;

...       /* TUTTO COME PRIMA */

27.     for ( ; ; ) {
28.       if( (connfd = accept(listenfd, (struct sockaddr *) NULL, NULL)) < 0){
                  printf("accept error \n");
                  exit(1);
             }
29.       if( (pid = fork()) == 0 ) {
30.           close(listenfd);
31.           ticks = time(NULL);
32.           snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
33.           while ( (n=write(connfd, buff, strlen(buff)) ) < 0 );
34.           close(connfd);
35.           exit(0);
36.                 }
37.                 close(connfd);
38.      }
39.}
```

Il processo figlio chiude il socket di ascolto (linea 30.), svolge il proprio compito (linee 31. - 33.), chiude il socket di connessione (linea 34.) e termina (linea 35.). Il padre, invece, chiude il socket di connessione (linea 37.) e torna ad attendere nuove richieste (linea 28.).

1.4 Esempi: echo client ed echo server.

```
1. #include        "basic.h"

2. void client_echo(FILE *fp, int sockfd);

3. int main(int argc, char **argv) {
4.         int                     sockfd, n;
5.         struct sockaddr_in      servaddr;

6.         if (argc != 3)
7.                         { printf("usage: echocli <IPaddress> <PORT>\n"); exit(1); }

8.         if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
9.                 { printf("socket error\n"); exit(1); }

10.         bzero(&servaddr, sizeof(servaddr));
11.         servaddr.sin_family = AF_INET;
12.         servaddr.sin_port   = htons(atoi(argv[2]));     /* echo server port */
13.         if (inet_pton(AF_INET, argv[1], &servaddr.sin_addr) <= 0)
14.                 { printf("inet_pton error for %s", argv[1]); exit(1);}

15.         if (connect(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr)) < 0)
16.                 { printf("connect error\n"); exit(1);}

17.         client_echo(stdin, sockfd);     /* svolge tutto il lavoro del client */
18.         exit(0);
19. }


20. void client_echo(FILE *fp, int sockfd) {
21.         char    sendline[MAXLINE], recvline[MAXLINE];
22.         while (fgets(sendline, MAXLINE, fp) != NULL) {
23.             reti_writen(sockfd, sendline, strlen(sendline));
24.              if (reti_readline(sockfd, recvline, MAXLINE) == 0)
25.              { printf("%s: server terminated prematurely",__FILE__); exit(1); }
26.              fputs(recvline, stdout);
27.         }
28. }
```

La struttura di echoclient è molto simile a quella del client daytime. Il client controlla che l'utente abbia invocato il programma in modo corretto (linee 6. - 7.), si predispone alla connessione con il server (linee 8. - 14.), effettua richiesta di connessione (linee 15. - 16.) e, se non si verificano errori, invoca la funzione *client_echo* che svolge il lavoro vero e proprio del client. Infatti, la funzione *client_echo*(), fino a quando non riceve CTRL D da tastiera (fine input da parte dell'utente), legge una stringa da tastiera (linea 22.) e la invia al server, scrivendola sul socket (linea 23.). Successivamente, legge dal socket la risposta (eco) del server (linea 24.) e, se non si verificano errori, stampa a video la stringa ricevuta.

```
1. #include        "basic.h"

2. void server_echo(int sockfd);

3. int main(int argc, char **argv) {
4.     pid_t                   childpid;
5.     int                     listenfd, connfd;
6.     struct sockaddr_in      servaddr, cliaddr;
7.     socklen_t               cliaddr_len;

8.     if( argc != 2){
9.       printf("Usage: echosrv <PORT> \n"); exit(1);
10.   }

11.    if( (listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
12.      { printf("socket error\n"); exit(1); }

13.    bzero(&servaddr, sizeof(servaddr));
14.    servaddr.sin_family      = AF_INET;
15.    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);  /* wildcard address */
16.    servaddr.sin_port        = htons(atoi(argv[1]));        /* echo server */

17.    if( (bind(listenfd, (struct sockaddr *) &servaddr, sizeof(servaddr))) < 0)
18.      {printf("bind error\n"); exit(1); }

19.    if( listen(listenfd, BACKLOG) < 0 )
            {printf("listen error\n"); exit(1);}

20.    for ( ; ; ) {
21.      cliaddr_len = sizeof(cliaddr);
22.      if( (connfd = accept(listenfd, (struct sockaddr *)&cliaddr, &cliaddr_len)) < 0)
23.      { printf("accept error\n"); exit(1); }

24.      if( (childpid = fork()) == 0 ) {
25.        close(listenfd);
26.        server_echo(connfd);      /* svolge tutto il lavoro del server */
27.        exit(0);
28.     }

29.         close(connfd);
30.   }
31. }
```

Il server, invece, è ricorsivo. Ad ogni nuova richiesta, crea un nuovo processo figlio per gestire la connessione. La struttura di echoserv è la stessa di daytimesrv. Controlla la correttezza dell'invocazione da parte dell'utente (linee 8. - 10.) e si predispone ad accettare richieste di servizio invocando le funzioni *socket(), bind(), listen()* (linee 11. - 19.). Quindi, si prepara a gestire nuove connessioni (linee 20. - 23.). A ogni nuova richiesta, crea un nuovo processo figlio invocando la funzione *fork()*

(linea 24.). Il processo figlio (che riceve 0 come valore di ritorno dalla funzione *fork()*) chiude il socket di ascolto (linea 25.), invoca la funzione *server_echo*, che svolge il lavoro effettivo del server (descritta di seguito), e termina l'esecuzione (linee 26. - 28.). Il processo padre, invece, (che riceve dalla funzione *fork()* il *pid* del processo figlio), chiude il socket di connessione (linea 29.) e torna in ascolto.

```
void server_echo(int sockfd) {

1.        ssize_t         n;
2.        char            line[MAXLINE];

3.        for ( ; ; ) {
4.                if ( (n = reti_readline(sockfd, line, MAXLINE)) == 0)
5.                        return;         /* connection closed by other end */
6.               reti_writen(sockfd, line, n);
7.        }
8. }
```

La funzione server_echo specifica le azioni del server. Legge una linea dal socket (linea 4.) Nota che, per come è stata implementata, *reti_readline()*, legge dal socket fino a quando incontra il carattere di fine linea \ n e restituisce il numero di caratteri letti dal socket. Se l'altra estremità chiude ad un certo punto la connessione, all'invocazione successiva *reti_readline()* restituisce zero. Una volta letta una linea, il server ne fa l'eco, riscrivendola sul socket (linea 6.).

Una nota: è importante che il processo padre chiuda il socket di connessione e che il processo figlio chiuda il socket di ascolto. Perchè? Abbiamo detto che, a meno delle differenze evidenziate, i socket sono molto simili ai file. Infatti, il sistema operativo internamente usa le *stesse* strutture per rappresentare file e socket. Pertanto, così come ad un file è associato un contatore di riferimenti che viene incrementato ad ogni apertura del file da parte di un processo, e il file viene realmente chiuso soltanto quando il contatore è uguale a zero, allo stesso modo ad un socket è associato un contatore e il socket viene realmente chiuso solo quando il contatore è a zero. Nel nostro caso, se per esempio il processo padre non chiudesse il socket di connessione, la terminazione del figlio *non* implicherebbe la chiusura del socket, ma semplicemente il decremento del contatore da 2 a 1.

Come risulterà chiaro dalle lezioni teoriche, il protocollo di trasporto TCP (che, ricordo, i nostri client e server stanno usando creando socket con l'opzione SOCK_STREAM), crea l'astrazione di un canale di comunicazione affidabile tra le due parti, sul quale scorre *un flusso di bit*. L'*interpretazione dei dati* che le parti si scambiano e la *fine del flusso* sono convenzioni che stabiliscono le due applicazioni. Nel nostro esempio, client e server leggono e scrivono linee dal e sul socket. Il carattere di fine linea rappresenta quindi la terminazione del "record" di informazione che una parte invia all'altra. Spesso la terminazione del flusso è rappresentata da una delle parti che chiude la connessione. Esistono però anche altre convenzioni utilizzate dalle applicazioni.

Due funzioni utili e non ancora discusse sono:

```
#include <sys/socket.h>

int getsockname(int sockfd, struct sockaddr * localaddr, socklen_t *addr_len);
int getpeername(int sockfd, struct sockaddr * peeraddr, socklen_t *addr_len);

valore di ritorno: < 0 se errore
                   0, altrimenti
```

La funzione *getsockname*() restituisce l'indirizzo locale associato al socket. Si ricordi, infatti, che quando il client crea un socket (e non invoca *bind*(), operazione lecita ma inusuale per un client), riceve un descrittore, ma *non conosce* il numero di porta che il kernel ha generato (e se l'host ha più interfacce di rete, neanche quale indirizzo IP è stato usato). Con questa funzione, il client riesce a recuperare questi valori. Il sistema operativo copia i valori nella struttura *localaddr*. Il parametro *addr_len* in chiamata contiene la taglia della struttura *localaddr*. Nota che anche il server può aver necessità di usare *getsockname*(): se invoca *bind*() con numero di porta pari a zero, il kernel sceglie una porta effimera. Parimenti, se invoca *bind*() con il valore INADDR_ANY per l'indirizzo IP, per conoscere successivamente l'indirizzo IP del socket di connessione creato dal kernel a seguito di una *accept*() andata a buon fine, il server deve usare *getsockname*() (con il socket descriptor del socket di connessione, naturalmente).

In modo simile, la funzione *getpeername*() restituisce l'indirizzo "dell'altro lato" della comunicazione. Una situazione tipica in cui risulta utile è la seguente. Abbiamo visto che un server ricorsivo crea un figlio per ogni client. Per interazioni particolarmente complesse, il processo figlio potrebbe invocare una chiamata di funzione *exec*, tramite la quale la propria immagine di memoria viene totalmente sostituita (file descriptor, socket descriptor ... vengono mantenuti). In questo caso, il figlio, usando la funzione *getpeername*() riesce a conoscere l'identità del client.

1.5 Conclusioni.

Nella lezione di oggi abbiamo:

- Acquisito diverse funzioni che semplificheranno l'implementazione di applicazioni client-server.
- Discusso di server iterativi e ricorsivi, evidenziando i vantaggi dei server ricorsivi rispetto ai server iterativi.
- Analizzato un secondo esempio concreto di interazione client-server.

1.6 Esercizi: uso delle funzioni e variazioni dell'interazione client-server.

A questo punto disponiamo di un numero sufficiente di elementi per poter iniziare a scrivere semplici applicazioni client-server. Cominciamo a prender confidenza con le struttura del client e del server, apportando modifiche semplici a echocli.c ed echosrv.c.

Esercizio 1. Si modifichi echosrv.c in modo tale che, non appena la funzione *accept*() ritorna con successo, il server stampa a video *l'indirizzo IP* ed il *numero di porta* del client con cui è connesso.

Esercizio 2. Si modifichi echocli.c in modo tale che, non appena la funzione *connect*() ritorna con successo, il client stampa a video *l'indirizzo IP* ed il *numero di porta* che il kernel ha scelto per il socket del client.

Esercizio 3. Si modifichi echosvr.c in modo tale che invii al client *sempre lo stesso messaggio* (i.e., una sorta di messaggio di acknowledgement) in risposta alla riga ricevuta dal client.

Esercizio 4. Si modifichino echocli.c ed echosrv.c come segue:

- il client riceve da tastiera *due righe* e le invia al server
- il server conta il numero di caratteri presenti nelle due righe, stampa a video il risultato e lo invia al client.
- il client legge tale valore, lo stampa a video e ritorna ad attendere righe da tastiera.

Esercizio 5. Si modifichino echocli.c ed echosrv.c dell'esercizio precedente facendo si che, invece di inviare due righe al server, il client *invii k righe*, dove il parametro k viene scelto dall'utente e passato al client da linea di comando (tramite il vettore argv[]) all'avvio.