Comunicazione fra Processi
l'interfaccia socket

Università di Parma - Appunti del Corso di Telematica
A. Lazzari, Giugno 2001

SOMMARIO

Introduzione

L'interfaccia con cui un processo applicativo accede ai servizi di comunicazione (ossia, in ambiente TCP/IP, l'interfaccia con lo strato di trasporto) è detta genericamente *Application Program Interface* (API).

In teoria ogni sistema elaborativo potrebbe inventarsi le proprie interfacce. In pratica l'interfaccia più diffusa è quella universalmente nota come *socket*. L'interfaccia *socket* è apparsa la prima volta in UNIX con la versione 4.2 del Berkeley System Distribution (BSD), è stata poi portata sulle versioni commerciali da essa derivate (SunOS, Linux, ecc.) e, senza sostanziali modifiche, anche in altri Sistemi Operativi (ad esempio Windows). Un meccanismo alternativo denominato TLI (Transport Layer Interface) è stato realizzato nelle versioni di UNIX sviluppate da AT&T (System V). Di esso, che non ha avuto la stessa diffusione, non ci occuperemo in questa sede.

L'interfaccia *socket* consiste in una serie di chiamate di sistema rese disponibili al programmatore. Il linguaggio di programmazione di riferimento è il C, tuttavia l'interfaccia *socket* è stata resa disponibile in molti altri linguaggi (in pratica tutti). Gli esempi che seguono suppongono un sistema operativo UNIX e il linguaggio C. In ogni caso si deve tenere presente che possono esistere leggere differenze fra diverse versioni di sistema, quindi è bene sempre verificare come effettuare le chiamate usando il comando **`man`** del sistema su cui si lavora.

La comunicazione fra processi avviene sempre tramite il servizio di trasporto supportato dalla pila di protocolli di Internet. A livello trasporto sono offerte due scelte: servizio "*connection oriented*" fornito tramite il protocollo TCP (*Transmission Control Protocol*) e servizio "*connectionless*" (o di tipo datagramma) fornito mediante il protocollo UDP (*User Datagram Protocol*).

Entrambi i protocolli poggiano sul sottostante IP (*Internet Protocol*) che è di tipo datagramma. Ma mentre TCP ne arricchisce le prestazioni in modo da garantire la sequenzialità dei pacchetti e da evitare la loro perdita o duplicazione, UDP si limita a incapsulare ogni datagramma utente in un datagramma IP, fornendo così un servizio intrinsecamente inaffidabile. La scelta di UDP deve pertanto prevedere, a livello applicativo, un insieme di provvedimenti atti a sopperire agli inconvenienti tipici della comunicazione a datagramma. Questo rende in genere l'uso di UDP più difficoltoso.

Servizi Ausiliari

Alcune funzioni di sistema permettono di accedere a servizi di rete senza dover gestire in modo esplicito la comunicazione con altri host, in particolare senza dover manipolare i *socket*.

`gethostbyname ( name )`

Questa funzione fornisce un'interfaccia col *resolver* (ossia quel codice di libreria che manda delle query al Name Server) e viene usata per effettuare l'operazione di *name resolution*, vale a dire ricavare l'indirizzo (o gli indirizzi) IP di un host noto il nome. Una chiamata a `gethostbyname()` provoca in genere l'intervento del *name server* locale ed eventualmente di *name server* remoti. L'unico argomento è:

```
char  *name;
```

Se la chiamata ha avuto successo, la funzione fornisce un puntatore a una struttura **`hostent`** (host entry) definita come segue:

```
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 from name server */
 #define h_addr h_addr_list[0]  /* address, for backward compatibility */
 };
```

La dichiarazione di questa struttura e molte altre dichiarazioni relative alla comunicazione fra processi (ad esempio la dichiarazione per `AF_INET`), sono contenute in una serie di *file* di *heading* che occorre includere nel codice sorgente. La seguente lista, anche se eccessiva in molti casi, contiene i file da includere per risolvere la maggior parte delle situazioni.

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

Se la chiamata è fallita (ad esempio perché l'indirizzo fornito non corrisponde ad alcun host, oppure l'host esiste ma è irraggiungibile) `gethostbyaddr` fornisce un *NULL pointer* (ossia un valore `0`). Occorre sempre, dopo la chiamata, prevedere un *test* per risolvere questa possibilità.

Come `gethostbyaddr` anche `gethostbyname` fornisce un puntatore a una struttura **`hostent`**, oppure un puntatore `NULL` se la chiamata fallisce.

gethostbyaddr (addr, len, type)

Questa funzione viene usata per effettuare una risoluzione inversa (*address resolution*), ossia ricavare il nome di un host noto un indirizzo IP. Gli argomenti sono:

```
char  *addr;
int   len, type;
```

Anche se dichiarato come puntatore a carattere, **`addr`** è un puntatore a un indirizzo in formato binario di lunghezza **`len`**, quindi a un campo di 4 byte contenente l'indirizzo IP. **type** identifica il tipo di indirizzamento e per Internet vale **`AF_INET`** (Address Family Internet). Come si vede la funzione è stata prevista anche per tipi di indirizzamento diversi da quelli di Internet, ma nel caso che interessa sarà sempre:

```
len = 4; type = AF_INET;
```

gethostname (name, namelen)

Questa funzione fornisce il nome ufficiale dell'host locale, ossia quello su cui il programma viene eseguito. Una chiamata a `gethostname()` non provoca quindi l'intervento di name server né alcuno scambio di pacchetti in rete. Gli argomenti sono:

```
char  *name;
int   namelen;
```

In *name* viene depositato il nome dell'host, la lunghezza di questo campo dev'essere di almeno 65 caratteri.

getprotobyname (name)

Ogni protocollo è identificato con un numero intero oppure con un nome simbolico. Questa funzione viene usata per ricavare il numero ufficiale di un protocollo dal suo nome.

`char  *name;` è l'unico argomento che contiene il nome simbolico.

Se la chiamata ha avuto successo, `getprotobyname` fornisce un puntatore a una struttura **protoent** (protocol entry) definita come segue:

```
struct  protoent {
char *p_name;      /* official name of protocol */
char **h_aliases;  /* aliases for the protocol */
int  p_proto;      /* official protocol number */
};
```

Si nota che, similmente ai *domain name* degli host, anche qui si possono avere uno o più *alias*. Se la chiamata fallisce viene fornito il *NULL pointer*.

getservbyname (name, proto)

Esistono in Internet alcuni servizi ufficiali designati con uno o più nomi simbolici *(well known services*"). Questa funzione viene usata per accedere a un servizio di cui si conosce il nome. Gli argomenti sono:

```
char  *name;
char  *proto;
```

Il primo argomento contiene il nome simbolico del servizio, il secondo quello del protocollo con cui si vuole accedere al servizio (ad esempio "tcp"). Se la chiamata ha avuto successo, getprotobyname fornisce un puntatore a una struttura **servent** (service entry) definita come segue:

```
struct  servent {
          char  *s_name;     /* official service name          */
          char **s_aliases;  /* aliases for the service        */
          int    s_port;     /* port used for this service     */
          char  *s_proto;    /* protocol used for this service */
 };
```

Esempio di uso di un servizio ausiliario

Il programma *findname* illustra l'accesso al servizio di *name resolution*.

```
/*---- File: findname.c (Aprile 1996)  --------------------*/
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <memory.h>
#include <netdb.h>
#include <stdio.h>
/*-----------------------------------------------------------------------
 * findname - Dato il nome di un host, ne visualizza:
 *            il nome ufficiale e gli eventuali alias
 *            tutti gli indirizzi in formato "decimale" (a.b.c.d)
 *            uso:  findname <nomehost>
 *-----------------------------------------------------------------------
*/
main(argc, argv)
int argc; char *argv[];
{
      char *buf="123.456.789.ABC"; /* deve contenere indirizzo IP */
      int i;
      struct hostent  *HostPtr;
      struct in_addr dummy;
      HostPtr=gethostbyname(argv[1]);
      if (HostPtr) {
          printf("Nome ufficiale: %s\n", HostPtr->h_name);
          /* Ricerca di eventuali alias */
          i = 0;
          while (HostPtr->h_aliases[i] != NULL) {
              printf( "%s%s\n", i == 0 ? "Alias     " : "\t\t",
                   HostPtr->h_aliases[i]);
              i+=1;
          }
          /* Ricerca di tutti gli indirizzi */
          memcpy(&dummy,HostPtr->h_addr_list[0],4);
          buf=inet_ntoa(dummy);
          printf("%s\n", buf);
          i = 1;
          while (HostPtr->h_addr_list[i] != NULL) {
             memcpy(&dummy,HostPtr->h_addr_list[i++],4);
             buf=inet_ntoa(dummy);
             printf(" %s\n\n", buf);
          }
      }
      else
          printf("Host sconosciuto o irraggiungibile\n\n");
}
```

`findname` effettua una chiamata alla funzione `gethostbyname`. Il risultato di questa chiamata viene depositato in una struttura di tipo hostent puntata da `HostPtr` (*Host Pointer*). Se il puntatore è nullo viene emesso un messaggio e si esce, altrimenti si effettuano le seguenti operazioni:

- stampa del nome principale (non è detto che esso coincida col nome dato)
- ricerca e stampa di eventuali *alias*

- stampa del primo indirizzo
- ricerca e stampa di eventuali altri indirizzi.

Si noti che in `buf` c'è una stringa, che è un indirizzo in formato decimale, che viene generata a partire dall'indirizzo in formato binario mediante la routine di conversione `inet_ntoa()` (`inet` sta per Internet, `ntoa` significa `network to ascii`). La funzione simmetrica di questa routine è la `inet_addr`, che prende come argomento una stringa contenente l'indirizzo decimale e restituisce l'indirizzo binario (in formato `unsigned long`).

L'Interfaccia *Socket*

Un *socket* è un punto terminale di una comunicazione. Esso può essere di due tipi:

stream: usa un servizio di trasporto affidabile (*connection oriented*) e fornisce all'utente un flusso di dati bidirezionale e continuo, cioè non strutturato in record.

datagram: usa un trasporto inaffidabile (*connectionless*) e fornisce un flusso bidirezionale in cui è mantenuta la distinzione in record.

Il meccanismo dei *socket* può essere considerato come un'estensione delle operazioni di I/O. Un applicativo accede alle funzioni di I/O specificando un numero intero, il *file descriptor*, che gli viene assegnato da UNIX mediante una chiamata alla funzione **`open`()**. UNIX usa il file descriptor per puntare a una struttura di dati interna che contiene la descrizione del file.

Similmente UNIX gestisce, per ogni *socket* attivo, una struttura di dati interna a cui punta mediante un numero intero detto ***socket descriptor***. La tabella dei file descriptor e dei *socket descriptor* è comune, quindi uno stesso numero non può essere assegnato a un file e a un *socket*. Per creare un *socket* e farsi assegnare un *socket descriptor* si usa la chiamata di sistema ***socket*()**:

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

L'argomento **`domain`** indica l'ambiente di comunicazione che si vuole usare. In ambiente TCP/IP esso vale sempre `AF_INET` (Address Family Internet).

`type` indica il tipo di *socket* voluto (`SOCK_STREAM` per *socket* di tipo stream; `SOCK_DGRAM` per *socket* di tipo datagram).

`protocol` indica il `protocollo` che si vuole usare. Normalmente si usa il protocollo di default, che viene indicato con `0`.

La chiamata restituisce un *socket descriptor*. In caso di errore restituisce -1 (se ad esempio non si ha il permesso di creare il *socket*, oppure si è specificato un protocollo incompatibile col tipo di *socket*). Quindi per creare un *socket* di tipo stream si usa la sequenza:

```
int sock;  /* socket descriptor */
sock = socket(AF_INET, SOCK_STREAM, 0);
```

Come detto, un *socket* costituisce il punto terminale di una comunicazione. Tale punto è identificato da un indirizzo IP e da un numero di porta. A un *socket* possono far capo più processi, il numero di porta identifica il processo. L'insieme di queste informazioni (indirizzo IP più numero di porta) costituisce l'indirizzo del punto terminale. Un punto terminale è identificato dalla seguente struttura:

```
struct sockaddr
    {
      unsigned short  sa_family; /* address family, AF_xxx */
      char          sa_data[14]; /* 14 bytes of address */
    };
```

Questa struttura è un semplice contenitore che definisce un'area di memoria di 16 byte, dove ai due byte iniziali in cui è messa l'`address family` seguono 14 byte non meglio specificati.

Oltre alla struttura generica `sockaddr`, ne esiste una specifica di Internet chiamata `sockaddr_in` che suddivide i 14 byte in <indirizzo IP più numero di porta>:

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

Il codice sorgente destinato ad operare in ambiente TCP/IP deve far riferimento a una struttura `sockaddr_in` tuttavia, per ragioni di universalità, le chiamate alle funzioni di SO richiedono che ci si riferisca a una struttura tipo `sockaddr`. Questo fatto richiede in genere l'uso di un *cast operator* nella chiamata.

Processi Client TCP

La comunicazione fra processi in Internet è di solito basata sul modello di interazione Clkient/Server. Un processo **Cliente** per accedere a un determinato servizio effettua una serie di richieste al processo **Servente**. Il servente per soddisfare ogni richiesta effettua alcune elaborazioni e fornisce una risposta. Quando il cliente ha inoltrato l'ultima richiesta e ricevuta l'ultima risposta l'interazione finisce. Nel caso più semplice si ha una sola richiesta seguita da una sola risposta.

Come si vede, ciò che distingue il cliente dal servente è il fatto che il cliente prende l'iniziativa dell'interazione. Quindi se i due processi usano un protocollo di comunicazione di tipo *connection oriented* (come TCP) il cliente sarà sempre il chiamante e il servente il chiamato.

Un processo servente è sempre attivo, in attesa di una richiesta da parte di un cliente. Soddisfatta la richiesta si pone in attesa della prossima richiesta, ponendosi così in un *loop* infinito.

Un cliente TCP, ossia che vuole accedere a un servente tramite TCP, segue tipicamente i seguenti passi:

- Specifica del punto terminale remoto
- Allocazione di un *socket*
- Formazione della connessione
- Comunicazione col servente
- Chiusura del *socket*

Specifica del punto terminale remoto

Una comunicazione ha due punti terminali, uno locale e uno remoto. Il punto locale corrisponde al cliente stesso, quello remoto al servente. Specificare il punto terminale del servente significa costruirne l'indirizzo, che comprende indirizzo IP e numero di porta del servente. Supponendo di conoscerli entrambi in formato binario, l'operazione è effettuata con le seguenti istruzioni:

```
u_long  Indserv;    /* Indirizzo dell'host del servente */
u_short Portserv;   /* Numero di porta del servente */
struct sockaddr_in  server;
server.sin_addr.s_addr = htonl(Indserv)
server.sin_port = htons(Portserv)
```

Si tratta di riservare un'area di memoria per la struttura server e di riempirne gli appositi campi con l'indirizzo IP e il numero di porta. A questo proposito si noti l'uso della routine di "*byte swapping*" **`hton`** (host to network). Questa routine viene usata perché UNIX si aspetta i byte ordinati secondo le convenzioni di rete.

Dato che l'ordinamento usato negli elaboratori può essere invertito rispetto a quello di rete (come ad esempio nei VAX), la routine inverte l'ordinamento dei byte. In altre macchine la routine può essere definita come una *null macro*, in ogni caso deve essere usata per garantire la portabilità del software. Di essa esistono due versioni: **`htons`** (hton short) agisce su 2 byte; **htonl** (hton long) agisce su 4 byte. Per passare dall'ordinamento di rete a quello tipico della macchina esistono le routine inverse **`ntohs`** e **`ntohl`**.

Allocazione di un socket

Queta operazione, già vista, serve a definire un descrittore di *socket* e viene effettuata con le istruzioni

```
int sock;
sock = socket(PF_INET, SOCK_STREAM, 0);
```

Formazione della connessione

A questo punto il cliente può richiedere la connessione al servente usando la chiamata di sistema **`connect`()**:

```
int connect(socket, addr, addrlen)
int socket;
```

```
struct sockaddr_in *addr;
int addrlen;
```

L'argomento ***socket*** è il descrittore del *socket*; **addr** è l'indirizzo della struttura che specifica il punto terminale remoto; **addrlen** è la lunghezza dell'argomento precedente.

La chiamata restituisce 0 in caso di successo, -1 in caso di errore. Dato che il caso di insuccesso è abbastanza probabile (l'host può essere irraggiungibile, il servizio non disponibile, ecc.) è particolarmente importante (come del resto sempre in queste chiamate) esaminare il risultato della chiamata. Le istruzioni per effettuare la connessione potranno quindi essere simili alle seguenti:

```
if (connect(sock, &server, sizeof(server)) < 0) {
  perror("Formazione della connessione");
  close(sock);
  exit(1);
}
```

Comunicazione col servente

Dopo che la connessione è stata effettuata, inizia la fase di interazione fra cliente e servente. Essa dipende fortemente dal tipo di servizio e dal protocollo applicativo, per cui è impossibile definire uno schema di carattere generale.

È definito invece il meccanismo di scambio di dati attraverso il *socket* di tipo stream. Esso avviene, come per un file, utilizzando le funzioni di I/O **write()** e **read().** Precisamente, il cliente allocherà ad esempio un'area di memoria `buf_to`, di dimensione utile `SIZETO` byte, in cui formare le richieste da mandare al servente, e un'area `buf_from` di `SIZEFROM` byte in cui ricevere le risposte.

Le istruzioni per lo scambio di dati potranno allora essere del tipo:

```
char buf_to[SIZETO+1];
char buf_from[SIZEFROM+1];
int n;
/* invio richiesta al servente */
write (sock, buf_to, sizeof(buf_to))
/* lettura risposta del servente */
while(n = read(sock, buf_from, SIZEFROM) > 0) {
buf_from[n] = '\0';             /* metti fine stringa */

(void) fputs(buf_from, stdout); /* visualizza */
  ....  ( oppure fai altre cose )  ...
}
```

L'operazione di invio della richiesta, effettuata mediante la funzione `write()`, non presenta particolarità. Invece la lettura della risposta è fatta in modo da premunirsi dalla possibilità di frammentazione provocata dal protocollo di trasporto. Può capitare infatti che il servente generi ad esempio una risposta con una singola operazione di `write()`, ma che TCP la trasporti in due o più segmenti. All'arrivo ogni chiamata a `read()`.provoca l'acquisizione dei soli byte presenti nel buffer di ricezione; per acquisire il resto del messaggio occorrono ulteriori chiamate a `read()`.

La soluzione dell'esempio prevede un ciclo di letture ognuna delle quali acquisisce un numero n di byte, variabile di volta in volta. Dal ciclo si esce con una condizione di *end of file*. Nell'esempio si suppone

che il messaggio sia costituito da una stringa stampabile (NULL-terminated) che viene gradualmente visualizzata su `stdout` (che di *default* coincide col monitor di un terminale video). In caso diverso le istruzioni andranno cambiate opportunamente.

Chiusura del socket

Finita l'interazione cliente/servente il *socket* può essere rilasciato dal cliente mediante la chiamata

```
close(sock);
```

Questa chiamata è già stata utilizzata, nel caso di connessione non riuscita, nelle istruzioni relative alla fase di connessione.

Esempio di Cliente TCP

Il programma ***`getdayti`*** riportato nel prospetto seguente illustra l'accesso al servizio DAYTIME.

Si noti che questo programma è particolarmente semplice, infatti non invia alcun dato al server ma si limita a ricevere una stringa dal server e a visualizzarla su `stdout`. Dato poi che il server svincola immediatamente dopo avere inviato la stringa, il programma non deve nemmeno preoccuparsi di chiudere la connessione.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <stdio.h>
#define  SIZEFROM  128
/*------------------------------------------------------------------
 * getdayti - Cliente TCP per servizio DAYTIME.  Visualizza data e ora

 *      fornite da un host dato per nome (default host locale).
 *      uso: getdayti [<nomehost>]
 *------------------------------------------------------------------
 */
main(argc, argv)
int argc;
char *argv[];
{
      char *host   = "localhost";   /* nome di default */
      char buf_from[SIZEFROM+1];    /* buffer di ricezione stringa */
      struct hostent         *phe;    /* descrizione host servente */
      struct servent         *pse;    /* descrizione servizio */
      struct sockaddr_in server;    /* punto terminale lato servente
*/
      int  sock, n;
      if (argc == 2)
            host = argv[1];
      bzero ((char *)&server, sizeof(server));
      server.sin_family = AF_INET;
      /* Ricava numero porta per servizio daytime */
      pse = getservbyname("daytime", "tcp");
      server.sin_port= pse->s_port;
      /* Ricava indirizzo host */
      if  ( phe = gethostbyname(host) )
          bcopy(phe->h_addr, (char *)&server.sin_addr, phe->h_length);
      else {
          perror("Host sconosciuto");
          close(sock); exit(0);
      }
      /* Alloca e connette il socket      */
      sock = socket(PF_INET, SOCK_STREAM, 0);
      if(sock < 0) {
          perror("Impossibile creare socket");
          exit(0);
      }
      if (connect(sock, &server, sizeof(server))  <0)
          perror("Connessione fallita");
          close(sock); exit(0);     }
      /* lettura risposta del servente */
      while((n = read(sock, buf_from, SIZEFROM)) > 0) {
          buf_from[n] = '\0';             /* metti fine stringa */
          (void) fputs(buf_from, stdout); /* visualizza */
      }
}
```

Server TCP

Un server tcp usa sempre due tipi di socket:

- **`Master`** lo assegna l'applicativo;
- **`Slave`** lo assegna il SO[1].

Il primo serve a ricevere le connessioni in arrivo; il secondo a effettuare lo scambio di dati col *client*. Entrambi vanno dichiarati nel programma.

La struttura del server è più agevolmente illustrata con un esempio. Con riferimento al listato che segue, evidenziamo solo le istruzioni tipiche di un server:

- Linee 19, 20: vengono dichiarati i due *socket*;
- Linea 22: si dichiara la struttura `myself` che contiene i dati del *server* stesso (in un programma *client* questa operazione è inessenziale);
- Linea 35: dopo avere inizializzato la struttura `myself` si chiama la `bind`, così facendo il SO associa la porta voluta al *socket* (i nostri *client* dovranno chiamarci su questa porta, la chiamata sarà ricevuta da `Master_sock`);
- Linea 41: `Master_sock` è messo in modo *passivo* tramite la chiamata `listen`, quando il *socket* si trova in questa modalità la porta associata può solo ricevere chiamate. Il secondo parametro, che nell'esempio vale 1, è il *backlog*, ossia il numero massimo di chiamate che il sistema può mettere in attesa[2], superato questo limite una nuova chiamata andrà persa;
- Linea 47: inizia il *loop* del server, la cui fine si trova alla linea 60. A ogni chiamata ricevuta corrisponderà una esecuzione del *loop*;
- Linea 48: il server si blocca, in attesa della prossima chiamata su `Master_sock`. Quando la chiamata arriva la procedura `accept` restituisce lo `Slave_sock` attraverso cui avviene il dialogo col *client*. I due parametri oltre a `Master_sock` sono rispettivamente il puntatore a una struttura di tipo `sockaddr` in cui verranno messi i dati del chiamante e la lunghezza di tale struttura. Avendo messo a zero il puntatore, rinunciamo a sapere chi ci ha chiamato;
- Linee 50-58: queste istruzioni effettuano l'eleborazione vera e propria gestendo il dialogo col cliente e sono tipiche del servizio;

[1] La terminologia Master Slave non è ufficiale e non fa parte del linguaggio C né delle definizioni relative ai *socket*; viene solo usata qui per comodità, ma ai due *socket* si può assegnare qualunque altro nome.

[2] Il numero minimo è zero. Il massimo dipende dal sistema e in genere non supera qualche unità (ad es. 3 o 5), anche mettendo un valore molto elevato il valore effettivo gestito è comunque il massimo di sistema.

```c
/* Server tcp sequenziale */

#include <stdio.h>
#include <signal.h>
#include <netdb.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <time.h>

/*-------------------------------------------------------------------------
 *  Server tcp per servizio di tipo dimostrativo
 *  uso:  server <porta>
 *-------------------------------------------------------------------------
*/
#define BUFLEN 1024     /* lunghezza buffer */

int main(int argc, char **argv){
  int stato;                      /* valore di ritorno delle system call */
  int Master_sock;                /* socket di ricezione delle chiamate */
  int Slave_sock;                 /* socket di comunicazione col chiamante */
  int port;                       /* porta sulla quale il server attende */
  struct sockaddr_in myself={0};  /* informazioni sul server */
  char secondi[10];               /* contiene il tempo dal 01.01.1970 */
  char buf[BUFLEN];
  int i, rval;

  if (argc<2) {fprintf(stderr,"Uso: %s <port>\n",argv[0]); exit(0);}
  port= atoi(argv[1]);
  Master_sock= socket(AF_INET, SOCK_STREAM, 0);
  if (Master_sock== -1)
      {perror("Master_socket"); exit(-1);}
  myself.sin_family=AF_INET;
  myself.sin_addr.s_addr = INADDR_ANY;
  myself.sin_port=htons(port);
  if (stato=bind(Master_sock, (struct sockaddr *)&myself, sizeof(myself))==-1)

      {
       perror("Errore - provare con un'altra porta");
       exit(1);
      }
  printf(" ----- C Server Daytime -------\n");
  stato = listen(Master_sock,1);
  if (stato == -1)
      { perror("listen()"); exit(-1);}
  for(i = 0; i<BUFLEN; i++) buf[i] = '\0';

  /* ----- LOOP DI SERVIZIO ------ */
  while(buf[0]!='0') {
    Slave_sock=accept(Master_sock,0,0);
    if (Slave_sock== -1){perror("accept()");exit(-1);}
    if ((rval = read(Slave_sock, buf, BUFLEN)) <0)
            perror("Reading stream message");
    printf ("Da client:");
    if(buf[0] == '0')
       printf("richiesta di shutdown da client");
    else
       printf("%s\n", buf);
    time((time_t *)secondi);
    write(Slave_sock, ctime((time_t *)secondi),25);
    close(Slave_sock);
  }
}
```

- Linea 59: finito di servire il cliente, il server deve chiudere il *socket* concludendo così il *loop*. Siamo pronti per il cliente successivo.

Server concorrente

Un server come il precedente, che può servire un solo cliente per volta, è detto *sequenziale*. Un server in grado di servire contemporaneamente più utenti è detto *concorrente*.

Esistono fondamentalmente due tecniche alternative per realizzare un server concorrente:

1. Un unico processo che usa delle chiamate a `listen` non bloccanti, dedica tempo a uno dei clienti attivi quando questo effettua una richiesta e quindi passa ad esaminare il prossimo cliente attivo, in modo *round robin*;

2. Un processo dedicato a ogni cliente. Ogni nuova chiamata genera un nuovo processo.

Illustriamo solo la seconda opzione, che si discosta assai poco come tecnica di programmazione dal server sequenziale. Per la prima opzione, preferibile in caso di *server* con una forte dinamica (ossia con un gran numero di chiamate brevi), si rimanda alla letteratura specializzata.

Supponendo di aver programmato un server sequenziale, per ottenere da esso un server concorrente basta introdurre una chiamata alla funzione `fork`, che genera un nuovo processo (processo figlio) che è una copia del processo iniziale (processo padre). Dopo la `fork` avremo quindi un nuovo processo, caratterizzato dal proprio identificatore (process identifier - pid); per sapere di quale processo si tratta si deve interrogare il valore restituito dalla funzione `fork`, essa assegna zero al figlio, mentre al padre consegna il pid del figlio appena generato.

Lo schema del server concorrente è allora il seguente: il processo padre chiude immediatamente il *socket slave* e torna in *loop* pronto a ricevere una nuova chiamata; il figlio chiude il *socket master* e dialoga col cliente attraverso il *socket slave*; quando ha finito chiude il socket e muore (istruzione `exit`).

```
listen(Master_sock,3);
while(1) {   (inizio loop)
     Slave_sock=accept(Master_sock....);
switch (fork()) {
     case 0:  /* processo figlio */
      close(Master_sock);
        .  .  .
       close(Slave_sock);
  exit;
     default:  /* processo padre */
              close(Slave_sock);
      }
```