Laboratorio di Reti di Calcolatori

Comunicazione tra processi in una interrete, Socket API.

Paolo D'Arco

Abstract

Scopo della lezione è spiegare concisamente come possono comunicare due processi in esecuzione su host diversi in una interrete, e presentare le funzioni fondamentali che la Socket API fornisce al programmatore.

1 Comunicazione tra processi su una interrete

I protocolli di comunicazione su reti sono organizzati in maniera stratificata. Ogni strato *utilizza* funzioni e strutture dati dello strato sottostante ed *offre* funzioni e strutture dati al livello superiore. Lo strato Z dell'host A comunica con lo strato Z dell'host B.

1.1 Suite TCP/IP

Nella suite TCP/IP gli strati sono 5.

- Lo *strato fisico* si occupa della trasformazione dell'informazione in segnali che possono essere inviati sui mezzi trasmissivi.
- Lo *strato di collegamento* dei dati rende possibile lo scambio di dati tra *due host che si trovano sulla stessa rete*, e.g., una LAN. Ogni macchina ha un *indirizzo fisico* diverso con cui è univocamente identificabile sulla rete (e.g., MAC address di una scheda di rete per una Ethernet).
- Lo *strato di rete* rende possibile lo scambio di dati tra *due host della interrete*. Ogni macchina ha un *indirizzo logico* con cui è univocamente identificabile in una interrete. Nella suite TCP/IP questo indirizzo è *l'indirizzo IP*, una stringa di 32 bit.
- Lo *strato di trasporto* rende possibile la comunicazione tra un processo X sulla macchina A ed un processo Y sulla macchina B in una interrete. Ogni processo è univocamente identificabile in un host attraverso un *indirizzo logico*. Nella suite TCP/IP questo indirizzo è detto *numero di porta*, ed è una stringa di 16 bit.
- Lo *strato delle applicazioni* dialoga con i livelli sottostanti (in genere con lo strato di trasporto) per realizzare precise funzionalità utili agli utenti del sistema.

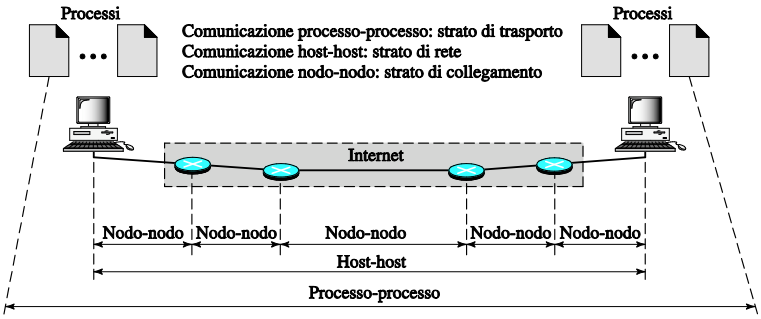


Figure 1: Comunicazione tra processi in una interrete.

1.2 Indirizzi IP e numeri di porta

Un indirizzo IP, oltre che in binario, può essere rappresentato come una sequenza di 4 numeri decimali, separati da punti. Precisamente, la sequenza di 32 bit viene divisa in 4 byte. Ogni numero é il valore decimale del corrispondente byte.

I numeri di porta vanno da 0 a 65535. Le porte da 0 a 1023 sono dette porte *ben note* e vengono usate dai sistemi operativi per gestire i servizi più comuni. Il file */etc/services* elenca questi servizi. Andate a curiosare! Le porte da 1024 a 49151 possono essere registrate, presso l'autorità IANA, per offrire specifici servizi. Le porte da 49152 a 65535 sono le porte dinamiche, non possono essere registrate e possono essere usate liberamente.

La concatenazione di un *indirizzo IP e di una porta* (e.g., 192.15.18.234 : 9877) viene detta *socket* (che, letteralmente, significa "presa"). Un socket individua univocamente un processo in esecuzione su un dato host. Una coppia di processi, per comunicare, ha bisogno di una coppia di socket. La coppia rappresenta le due estremità di un canale di comunicazione, e può essere usata dai due processi per scambiarsi dati.

1.3 Comunicazione Client-Server.

La Socket API è una *collezione di strutture di dati e di funzioni* che permettono al programmatore di scrivere in modo semplice programmi in grado di comunicare sulla interrete. In particolare, permette di creare socket, di leggere dati dal canale associato ad una coppia di socket e di scrivere dati su di esso. In prima approssimazione il canale può essere visto come *una sorta di file* in cui i processi possono leggere e scrivere. Pertanto, così come un file viene prima creato per poi esser letto e scritto, e viene chiuso quando non serve più, così due processi che intendono comunicare su

una interrete, debbono prima creare una coppia di socket, dopodichè possono leggere e scrivere sul canale che la coppia definisce, e chiuderlo quando non serve più.

Sebbene l'interazione tra due (o più) processi possa avere forme diverse, nel prosieguo ci concentreremo su una particolare forma di interazione, detta comunicazione *client-server*. Considereremo, cioè, un modello in cui un processo in esecuzione su una macchina B, il server, offre un servizio in una interrete. Un altro processo dell'interrete, il client, in esecuzione sulla macchina A, ha necessità del servizio e chiede al server di soddisfare le proprie richieste. Più precisamente (ma con diverse approssimazioni che saranno rimosse nel prosieguo), i passi salienti della comunicazione, usando il protocollo dello strato di trasporto TCP (che offre trasmissioni affidabili), saranno i seguenti:

1. Il server, in esecuzione sulla macchina B, per offrire un servizio, crea un socket, informa il sistema operativo del fatto che offre un servizio e si mette *in attesa* che qualcuno ne abbia bisogno. Il socket creato dal server può esser visto come una sorta di *posto in cui inviare richieste di servizio.*

2. Il client, in esecuzione sulla macchina A, crea a sua volta un socket e chiede al sistema operativo della macchina B, di connettere il proprio socket al server in attesa. Nella richiesta il client fa riferimento al socket che il server ha creato per *ricevere richieste di servizio.* Se questa richiesta può essere accolta, il sistema operativo dell' host B, crea un *nuovo socket* per il server e connette questo nuovo socket al socket del client in esecuzione sull'host A. A questo punto, il canale tra i due processi é stabilito.

3. Il client ed il server si scambiano dati, leggendo e scrivendo sul canale associato alla coppia di socket. In particolare, ognuno dei due processi usa la propria estremità del canale (i.e., il proprio socket) per leggere e scrivere.

4. Al termine della comunicazione, il canale viene chiuso. Tale operazione, solitamente viene iniziata dal client, che una volta ricevuto il servizio offerto dal server non ha più motivi per utilizzare il canale e chiude la propria estemità, ma può avvenire anche per iniziativa del server.

5. D'altra parte, il server, una volta chiusa l'estremità del canale che ha usato per comunicare con il client (i.e., il socket che il sistema operativo ha creato su richiesta di connessione del client), torna ad attendere che un client abbia bisogno del proprio servizio ed inoltri una nuova richiesta. A quel punto, la comunicazione avviene secondo le stesse modalità, i.e., i passi da 2. a 5. appena descritti si ripetono.

1.4 Tipica interazione usando TCP

La figura che segue mostra i passi tipici di una interazione client - server, usando le funzioni della Socket API che permettono di stabilire una connessione tra i due processi usando il protocollo dello strato di trasporto TCP.

Brevemente:

- Comincia il Server. Invocando le funzioni *socket(), bind()* e *listen()*, informa il sistema operativo che è pronto ad offrire un servizio. Invocando poi la funzione *accept()*, si predispone ad accettare richieste da parte di eventuali client.

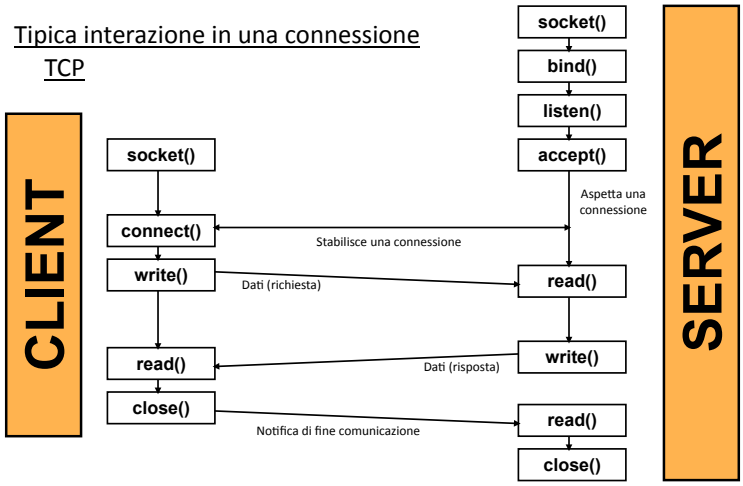


Figure 2: Interazione Client-Server attraverso TCP

- Il client, invocando la funzione *socket()*, crea la propria estremità del canale. Quindi, invocando *connect()* (con parametri che il server ha usato per comunicare al proprio sistema operativo dove è raggiungibile dai client per richieste di servizio), chiede al sistema operativo dell'host del server, di essere connesso al server. Se la richiesta viene accolta, il canale di comunicazione è stabilito.
- Successivamente, attraverso le funzioni *read()* e *write*, client e server comunicano tra di loro.
- Infine, quando la connessione non serve più, il client ed in server chiudono, tramite *close()*, le proprie estremità del canale di comunicazione.

Cerchiamo a questo punto di capire quali sono le *strutture dati* coinvolte in questo processo, e come sono definite le *funzioni* della Socket API invocate dal client e dal server (Tabella 1).

La struttura in_addr definisce un indirizzo IP. E' una struttura con un unico campo s_addr, di tipo in_addr_t (...perchè una struttura e non un intero? ...ragioni storiche). La struttura sockaddr_in definisce invece la *rappresentazione di un socket*. Come mostra la Figura 3, esistono diversi *tipi di socket*. I campi della struttura identificano:

- sin_len la grandezza della struttura espressa in byte (esistono strutture con grandezze diverse)
- sin_family il tipo di protocollo che userà il socket, per noi sarà sempre AF_INET
- sin_port il numero di porta, memorizzato in formato big-endian (network byte order)
- sin_addr la struttura che contiene l'indirizzo IP

```
struct in_addr {
      in_addr_t s_addr; /* 32-bit, network byte ordered */
}
struct sockaddr_in {
     uint8_t sin_len;
     sa_family_t sin_family; /* tipo di protocollo, AF_INET */
     in_port_t sin_port; /* 16-bit, network byte ordered */
     struct in_addr sin_addr; /* struttura indirizzo IP */
     char sin_zero[8];
}
struct sockaddr {
     uint8_t sin_len;
     sa_family_t sin_family; /* tipo di protocollo: AF_XXX  */
     char sa_data[14]; /* indirizzo specifico del protocollo */
}
```

Table 1: Strutture in_addr, sockaddr_in e sockaddr.

- sin_zero[8] usata per far si che la struttura abbia taglia minima di 16 byte.

Infine, la struttura sockaddr, identifica un *socket generico*. Non ci interessano i dettagli. Non lavoreremo mai direttamente con i suoi campi. Come vedremo tra breve, esistono diversi tipi di socket (vd, Figura 3). Le funzioni che operano sui socket prendono in input una struttura di tipo socket generico (per poter gestire tutti i tipi). Questo significherà che ogni volta che sarà necessario, utilizzeremo l'operatore di cast () del linguaggio C, per convertire una struttura di tipo sockaddr_in in una di tipo sockaddr. (Le strutture vengono passate alle funzioni sempre *per riferimento*.)

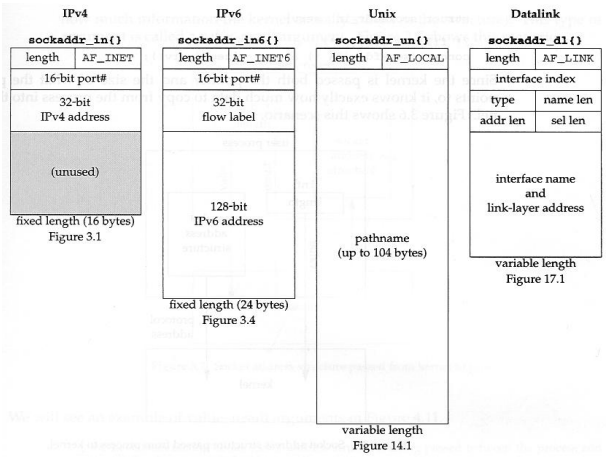


Figure 3: Tipi di socket

Le uniche strutture che ci interesseranno in questo corso sono del primo tipo (e marginalmente del secondo). Le altre che sono definite anche per altri usi da parte del sistema operativo, non saranno trattate. (Di recente è stata aggiunto un nuovo tipo, sockaddr_storage).

1.5 Funzioni principali.

Vediamo a questo punto come sono definite le funzioni principali della socket API, invocate da client e server. Per ogni funzione riporteremo gli *header file* che debbono essere inclusi nel file che contiene l'invocazione, la *sintassi* della funzione, ed i valori di ritorno.

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

int socket(int family, int type, int protocol);

valore di ritorno: -1 se errore
                   un socket descriptor, altrimenti
```

La funzione *socket*() chiede al sistema operativo di creare una struttura socket. Se l'invocazione ha successo, il processo invocante riceve un valore intero, detto *socket descriptor*. Rappresenta l'estremità del canale di comunicazione da cui il processo può leggere ed in cui può scrivere. E' il proprio *punto di accesso*. Un socket descriptor è simile ad un file descriptor (in realtà sono presi dal sistema operativo dallo stesso insieme, per cui se un intero è usato come file descriptor, non può essere usato come socket descriptor, e viceversa ...). Socket e file sono utilizzabili più o meno allo stesso modo e, come vedremo nel seguito, le funzioni *read()*, *write()* e *close()* sono le stesse funzioni.

La funzione *socket*() prende in input tre parametri:

- int *family*: un intero che specifica quale famiglia di protocolli si intende usare
  - AF_INET per il protocollo IPv4
  - AF_INET6 per il protocollo IPv6
  - AF_LOCAL protocollo locale (client e server sullo stesso host)
  - AF_ROUTE socket usati per il routing
  - ...
- int *type*: un intero che specifica il tipo di socket
  - SOCK_STREAM per un flusso di dati (TCP)
  - SOCK_DGRAM per datagrammi (UDP)
  - SOCK_RAW per applicazioni dirette su IP
- int *protocol*: un intero che vale sempre 0, a parte quando usato per SOCK_RAW

Una invocazione tipica della funzione (nei programmi di esempio che studieremo ed in quelli che scriveremo) sarà:

sd= *socket*(AF_INET, SOCK_STREAM,0);

Con essa chiediamo al sistema operativo di creare un socket per lavorare con il protocollo di trasporto TCP ed il protocollo di rete sottostante IP, versione 4 (i.e., indirizzi IP a 32 bit). Quando l'invocazione ha successo, la variabile intera conterrà il socket descriptor con cui successivamente potremo leggere e scrivere sul canale di comunicazione.

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

int connect(int sd, struct sockaddr *servaddr, socklen_t addrlen);

valore di ritorno: -1 se errore
                    0, altrimenti
```

L'invocazione della funzione *connect*() fa si che il sistema operativo apra un canale di comunicazione con il server, cioè stabilisca una *connessione* con il server. Il kernel del sistema operativo crea, per il processo invocante, un socket, con l'indirizzo IP dell'host su cui il processo invocante è in esecuzione, e con una porta *effimera* (prende il primo valore disponibile, solitamente a partire da 49152 fino 65535), così detta perchè "dura" esattamente quanto dura la connessione. I parametri della funzione sono un socket descriptor, ottenuto da una precedente chiamata a *socket*(), l'indirizzo di una struttura dati di tipo sockaddr (che come vedremo tra breve conterrà l'indirizzo IP ed il numero di porta su cui il server riceve richieste di servizio), e la lunghezza in byte di tale struttura.

Quando la funzione *connect()* viene invocata, nel caso di una connessione TCP, il sistema operativo avvia la fase di *handshake* in tre passi di tale protocollo, che permette al client ed al server di stabilire la connessione.

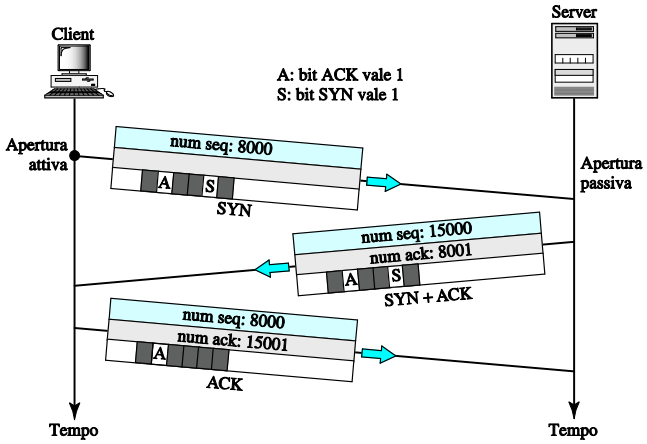


Figure 4: Three-way handshake

In caso di insuccesso, nella variabile di sistema *errno*, viene specificata la ragione del fallimento. Tipicamente, *errno* vale:

- ETIMEDOUT
- ECONNREFUSED

- EHOSTUNREACH

Nelle tre costanti, la E sta per errore. Significano, rispettivamente, che è scaduto il tempo massimo per stabilire una connessione e non c'è stato successo, la connessione è stata rifiutata dal server, il server è irraggiungibile.

Le due funzioni che seguono vengono invocate solitamente dal server.

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

int bind(int sd, struct sockaddr *myaddr, socklen_t addrlen);

valore di ritorno: -1 se errore
                    0, altrimenti
```

La funzione *bind()* permette al server di dare un "indirizzo" ad un socket. Precisamente, il server riempie una struttura di tipo sockaddr, specificando un indirizzo IP, un numero di porta, entrambi o nessuno, e passa per riferimento questa struttura al kernel del sistema operativo tramite la funzione *bind()*. In questo modo, il server chiede al sistema operativo di associare al socket anonimo (precedentemente creato a seguito di invocazione della funzione *socket()* dal sistema operativo) i parametri in sockaddr. Se questi parametri sono "pubblicamente" noti, i client che necessitano dei servizi che il server offre, possono fare richiesta di connessione usandoli nell'invocazione della funzione *connect*().

Nota che, se nella struttura *myaddr* che il server passa al kernel tramite la funzione *bind()*, il numero di porta non è specificato, allora il kernel ne sceglie una effimera. Inoltre, un server (detto *multihomed*) potrebbe essere in esecuzione su un host connesso alla rete attraverso *molteplici interfacce* (e.g., schede di rete). Specificando nella struttura *myaddr* per l'indirizzo IP il valore INADDR_ANY (che vale 0) il cosiddetto indirizzo *wildcard*, il server può ricevere richieste di connessione inviate su qualsiasi interfaccia di rete dell'host. Successivamente, all'arrivo di una richiesta di connessione, il sistema operativo creerà un socket di connessione per il server in cui l'indirizzo IP è quello dell' *interfaccia di rete* sulla quale il server ha ricevuto il primo pacchetto dal client[1].
Dopo aver invocato *socket()* e *bind()*, il server invoca una *listen()*.

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

int listen(int sd, int backlog);

valore di ritorno: -1 se errore
                    0, altrimenti
```

Questa funzione, invocata *soltanto* da un server TCP, serve essenzialmente a convertire il socket da "attivo" a "passivo", per far si che il kernel accetti connessioni da parte di client su quel socket. Questo socket viene anche detto socket di ascolto (*listening socket*). Infatti, per default, quando il sistema operativo crea un socket, si aspetta che il socket sia di un client[2].

Il parametro *backlog*, invece, specifica quante connessioni accettare e mettere in attesa per essere servite. La figura che segue aiuta a capire.

[1]Quando, nel corso di teoria acquisirete familiarità con il diagramma di stato del protocollo TCP, sarà chiaro che questo indirizzo IP si trova nel pacchetto che contiene il SYN, inviato durante in protocollo di handshake in tre passi.

[2]Nel diagramma di stato di TCP, il processo passa dallo stato CLOSED a LISTEN.

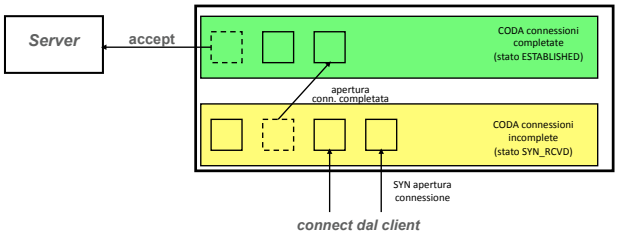


Figure 5: Backlog

Per un *listening socket* il kernel crea due code:

- una coda per le *connessioni incomplete*, che contiene una entry per ogni client che *ha iniziato* la procedura di handshake ma che non l'ha ancora conclusa
- una coda per le *connessioni completate*, che contiene una entry per ogni client che *ha concluso* la procedura

Ogni volta che la procedura di handshake con un client viene completata, la corrispondente entry nella coda delle connessioni incomplete, viene spostata nella coda delle connessioni completate. Il valore del parametro *backlog* stabilisce *quanti elementi possono contenere al più le due code messe assieme*. Cioè, la somma delle entry delle due code può essere al più *backlog*.

Infine, il server invoca la funzione *accept()*, che recupera la *prima connessione completata* dalla coda delle connessioni completate. Se non ce ne sono, il server si blocca, in attesa che una connessione completata sia disponibile.

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

int accept(int sd, struct sockaddr *cliaddr, socklen_t *addrlen);

valore di ritorno: -1 se errore
                   un socket descriptor, altrimenti
```

La funzione *accept* prende in input un socket descriptor, *sd* (il socket restituito al server dalla precedente invocazione della funzione *socket()* e usato nella *bind()* e nella *listen()*), il riferimento ad una struttura sockaddr, *cliaddr*, e il riferimento ad una variabile che contiene la lunghezza in byte della struttura, *addrlen*. I parametri *cliaddr* e *addrlen* servono per far ricevere al server l'identità del client che ha richiesto la connessione. Nelle applicazioni in cui il server non è interessato a

conoscere questa informazione, la funzione *accept* può essere invocata usando il puntatore NULL per questi due parametri.

L'argomento *addrlen* è un esempio di argomenti *valore-risultato*. Precisamente, quando *accept()* viene invocata, il valore intero referenziato da *addrlen corrisponde alla taglia della struttura *cli-addr*, passata dal server al kernel, e che il kernel riempirà con indirizzo IP e il numero di porta effimera del client. Quando *accept()* ritorna, il valore intero referenziato da ∗addrlen corrisponde alla lunghezza in byte della struttura riempita dal kernel del sistema operativo. In parole povere, la stessa area di memoria viene usata *per passare* il valore del parametro e *per ricevere* un risultato dell'esecuzione della funzione.

1.6 Esempi d'uso: server e client daytime

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

2. int main(int argc, char **argv) {
3.        int                     sockfd, n;
4.       char                    recvline[MAXLINE + 1];
5.        struct sockaddr_in      servaddr;

6.        if (argc != 3){
7.                printf("usage: daytimecli <indirizzoIP> <porta> \n");
8.                exit(1);
9. }
10.        if( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0 ){
11.                        printf("socket error \n");
12.                         exit(1);
13. }
14.       bzero(&servaddr, sizeof(servaddr));
15.       servaddr.sin_family = AF_INET;
16.        servaddr.sin_port = htons(atoi(argv[2])); /* server port */

17.        if (inet_pton(AF_INET, argv[1], &servaddr.sin_addr) <= 0){
18.                printf("inet_pton error for %s", argv[1]);
19.                exit(1);
20. }
21.       if (connect(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr)) < 0){
22.                printf("connect error \n");
23.                exit(1);
24. }
25.        while ( (n = read(sockfd, recvline, MAXLINE)) > 0) {
26.                recvline[n] = 0;        /* null terminate */
27.                fputs(recvline, stdout);
28.        }
29.                exit(0);
30.}
```

I due esempi implementano un client che interagisce con un server per ricevere giorno e ora. Il client include tutti prototipi di cui ha bisogno attraverso il file di intestazione *basic.h*. Dichiara variabili di cui avrà bisogno (linee 3.- 5.). Controlla che l'utente abbia invocato il programma in modo opportuno (linee 6.- 9.). Crea il proprio socket (linee 10.-13.). Prepara la richiesta di connessione al server. A tal fine, (linee 14. - 20.) azzera tramite la funzione *bzero()* lo spazio di memoria allocato per la struttura *servaddr* (potrebbe esser sporca ed inficiare le computazioni successive) e, riempie i campi, _family, _port, e sin_addr. In particolare si noti che il numero di porta viene convertito da carattere a intero (*atoi()*) e successivamente nel formato di rappresentazione di rete (*htons()*, host-to-network-short). Parimenti, l'indirizzo IP, che l'utente digita nella rappresentazione decimale, viene convertito nella rappresentazione di rete (*inet_pton()*, internetaddress-presentation-to-network). Descriveremo in dettaglio queste funzioni di conversione tra breve. Successivamente chiede la connessione al server, passando nella richiesta la struttura *servaddr* (linee 21. - 23.). Infine, a connessione stabilita, legge dal socket e memorizza il contenuto nel vettore *recvline[]*, aggiunge "0", per far si che il contenuto sia conforme alla rappresentazione del linguaggio C delle stringhe, e invoca la funzione *fputs()* (che lavora su stringhe) per stamparne il contenuto a video.

Un commento a parte merita la linea 25. Perchè invocare la funzione *read()* in un loop? Quando leggiamo da un file, non lo facciamo. Un socket è simile ad un file, ma *non è* un file. Per capire le differenza, si noti che i dati, quando scritti su un socket, vengono deposti dall'applicazione in un *buffer* e, successivamente spediti dal sistema operativo lungo la rete. Analogamente, i dati vengono letti dall'applicazione da un buffer, che viene riempito dal sistema operativo al sopraggiungere di dati da rete. In parole povere, *read* e *write* leggono e scrivono da/in buffer. Per vari motivi l'applicazione potrebbe non riuscire a leggere perchè i dati non sono ancora nel buffer, o a scrivere perchè non c'è spazio nel buffer che sarà liberato dal sistema operativo a breve. Per risolvere il problema legato a queste *indisponibilità* temporanee, codificando le operazioni in un loop, ci si assicura che l'operazione venga portata a termine. Il loop va terminato o quando la *read()* restituisce 0 (l'altra estremità ha chiuso la connessione) o un valore < 0 (si è verificato un errore).

Analizziamo la struttura del server, riportato nel riquadro della pagina seguente. Vengono inclusi i file di intestazione necessari (linea 1.). Vengono dichiarate le variabili necessarie nel seguito (linee 4. - 7.). Il server controlla che l'utente abbia invocato il programma correttamente (linee 8.-11.). Chiede al sistema operativo di creare un socket per ricevere richieste di servizio (linee 12. - 15.) Riempie una struttura dati *servaddr* con l'indirizzo IP e la porta a cui il client farà riferimento nelle proprie richieste (linee 16. - 19.) e chiede al sistema operativo di associare questi parametri al socket anonimo precedentemente creato (linee 20. - 22.) L'indirizzo IP specificato in questo caso è il cosiddetto valore *wildcard address*. Con questa direttiva, ricordo, il server essenzialmente dice al sistema operativo di voler accettare richieste su qualsiasi interfaccia arrivino. Successivamente, (linee 23. - 26.) il server chiede al sistema operativo di mettere il socket in *ascolto*, trasformandolo da attivo a passivo, e specificando che il server non vuole gestire più di 5 connessioni alla volta (valore backlog). Infine, entra in un loop infinito, e attende richieste di servizio (linee 27. - 29). Ad ogni nuova richiesta, il sistema operativo crea un nuovo socket per il server (un socket detto *di connessione*). Il server invoca la funzione *time()* per ricevere data e ora, costruisce una stringa contente la risposta, la scrive sul socket, e chiude il socket di connessione (linee 30. - 34.) Torna, quindi, ad attendere eventuali nuovi client.

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

3. int main(int argc, char **argv) {
4.         int                     listenfd, connfd, n;
5.         struct sockaddr_in      servaddr;
6.         char                    buff[MAXLINE];
7.         time_t                  ticks;

8.         if (argc != 2 ){
9.                 printf("usage: daytimesrv <porta>\n");
10.                exit(1);
11.}
12.        if( (listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0){
13.                printf("socket error\n");
14.                exit(1);
15.}
16.         bzero(&servaddr, sizeof(servaddr));
17.         servaddr.sin_family      = AF_INET;
18.         servaddr.sin_addr.s_addr = htonl(INADDR_ANY);   /* wildcard address */
19.         servaddr.sin_port        = htons(atoi(argv[1])); /* server port */

20.         if( (bind(listenfd, (struct sockaddr *) &servaddr, sizeof(servaddr))) < 0){
21.                 printf("bind error \n"); exit(1);
22.}
23.         if( listen(listenfd, 5) < 0 )
24.                 { printf("listen error \n");
25.                   exit(1);
26.}
27.         for ( ; ; ) {
28.                 if( (connfd = accept(listenfd, (struct sockaddr *) NULL, NULL)) < 0)
29.                 { printf("accept error \n"); exit(1);}
30.                 ticks = time(NULL);
31.                 snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
32.                 while ( (n=write(connfd, buff, strlen(buff))) < 0 ) ;
33.                 close(connfd);
34.         }
35.}
```

1.7 Protocollo IPv6

La trattazione precedente fa riferimento al protocollo IP originario, IPv4. Come scoprirete andando avanti con le lezioni teoriche, la crescita esponenziale di Internet ha reso necessario un nuovo protocollo che offrisse uno spazio di indirizzamento più grande di quello offerto da IPv4 e caratteristiche di efficienza e sicurezza non presenti in IPv4. In IPv6 un indirizzo IP è lungo 16 byte, cioè 128 bit. Nei prossimi anni IPv6 dovrebbe gradualmente sostituire IPv4.

Client e server appena presentati possono essere facilmente modificati per utilizzare il protocollo IPv6 invece di IPv4. In particolare, riscriviamo il nostro daytime client:

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

2. int main(int argc, char **argv) {
3.         int                     sockfd, n;
4.        char                    recvline[MAXLINE + 1];
5.        struct sockaddr_in6      servaddr;

6.         if (argc != 3){
7.                 printf("usage: daytimecli <indirizzoIP> <porta> \n");
8.                 exit(1);
9. }
10.         if( (sockfd = socket(AF_INET6, SOCK_STREAM, 0)) < 0 ){
11.                         printf("socket error \n");
12.                          exit(1);
13. }
14.        bzero(&servaddr, sizeof(servaddr));
15.        servaddr.sin6_family = AF_INET6;
16.         servaddr.sin6_port = htons(atoi(argv[2])); /* server port */

17.         if (inet_pton(AF_INET6, argv[1], &servaddr.sin6_addr) <= 0){
18.                 printf("inet_pton error for %s", argv[1]);
19.                 exit(1);
20. }
21.        if (connect(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr)) < 0){
22.                 printf("connect error \n");
23.                 exit(1);
24. }
25.         while ( (n = read(sockfd, recvline, MAXLINE)) > 0) {
26.                 recvline[n] = 0;        /* null terminate */
27.                 fputs(recvline, stdout);
28.         }
29.                 exit(0);
30.}
```

Abbiamo modificato le linee $5, 10, 15, 16$ e 17. Alcune funzioni tipo *inet_pton*() dell'API funzionano correttamente sia con IPv4 che IPv6. Due osservazioni: il codice prima dipendeva dal protocollo IPv4, ora dipende dal protocollo IPv6. In generale sarebbe auspicabile aver codice *indipendente* dal protocollo. Inoltre, in entrambi i casi, l'utente è costretto a conoscere l'indirizzo IP del server. Ricordare e maneggiare indirizzi IPv4 è tedioso ma fattibile. Per indirizzi IPv6, improponibile. Gli utenti preferiscono nomi ai numeri (e.g., www.vacanzealmare.it è più facile da ricordare di 193.18.210.19). Fortunatamente, come vedremo nel seguito, esistono funzioni che convertono nomi di host in indirizzi IP e nomi di servizi in numeri di porte.

1.8 Terminologia: una nota

In questa lezione e nelle prossime useremo talvolta espressioni del tipo: "il server crea un socket", "il client si connette al server" e simili. Ovviamente si tratta di una *semplificazione* utile ai fini

espositivi. Client e server sono semplici processi. Le azioni passano *attraverso i sistemi operativi sottostanti* dell'host su cui è in esecuzione il client e dell'host su cui è in esecuzione il server. Per capirci:

- i processi, durante la propria esecuzione, eseguono *codice proprio* (i.e., scritto dal programmatore) ed invocano sia *funzioni* appartenenti a librerie di sistema sia direttamente (ma meno frequentemente) *chiamate di sistema*
- le funzioni di libreria sono spesso, a loro volta, semplicemente degli *involucri* che semplificano all'utente le invocazioni di chiamate di sistema (e.g., la funzione *printf*() invoca la chiamata di sistema *write*(), gestisce i parametri ed i valori di ritorno della *write()*, e restituisce al processo un valore intero, in accordo alla specifica della *printf*(), che dà informazioni sull'operazione effettuata)
- le chiamate di sistema definiscono *i modi* con cui possono essere richiesti servizi al sottostante sistema operativo
- tra i servizi che il sistema operativo offre ci sono anche i *servizi di comunicazione su reti*, realizzati attraverso l'implementazione di protocolli per la comunicazione, e.g., la suite TCP/IP
- l'utente usa le funzioni dell'API per i socket per chiedere al sistema operativo servizi di comunicazione attraverso alcuni dei protocolli implementati nel sistema, per esempio i protocolli TCP o UDP dello strato di trasporto

Pertanto, per esemplificare, supponendo che il processo abbia necessità di comunicazioni affidabili e decida di lavorare con socket TCP, il flusso di dati tipo è:

- Invocazione della funzione dell'API da parte del processo
- Chiamata di sistema da parte della funzione dell'API
- Esecuzione di codice del kernel. Questo passo può richiede la semplice allocazione ed inizializzazione di strutture di dati, come nel caso di richiesta di creazione di un socket, o implicare una comunicazione tra lo strato TCP dell'host locale e lo strato TCP dell'host remoto, per esempio nel caso di letture e scritture
- Ritorno dalla chiamata di sistema e gestione dei valori di ritorno della chiamata da parte della funzione dell'API
- Restituzione al processo da parte della funzione dell'API dei valori stabiliti dalla specifica della funzione.

In conclusione, tutto ciò che l'utente deve capire è come *invocare correttamente* le funzioni, che *significato* hanno i parametri, e *quali informazioni* forniscono i *valori di ritorno*.

Se risulta chiaro che le espressioni menzionate vanno sempre intese come richieste di servizio che i processi rivolgono al sistema operativo, senza perdita di generalità, continueremo ad usarle.

1.9 Conclusioni

Nella lezione di oggi abbiamo capito che:

- La Socket API offre al programmatore strutture dati e funzioni che facilitano la scrittura di programmi comunicanti su una interrete. La API si interfaccia essenzialmente con lo strato di trasporto e vedremo esempi che usano i protocolli TCP (comunicazioni affidabili) e UDP (nessuna garanzia di affidabilità) e, tempo permettendo, anche SCTP (comunicazioni affidabili).
- Concentreremo la nostra attenzione sul paradigma di programmazione client-server, essendo uno dei metodi più diffusi di organizzazione delle computazioni su reti.
- Abbiamo analizzato l'interazione tipica client-server e le funzioni principali della Socket API per realizzare l'interazione. Nelle prossime lezioni la complessità e la qualità dei nostri client e server cresceranno, ma i passi di base dei semplici esempi visti saranno sempre presenti.

1.10 Esercizi: warm up file e directory

In diverse applicazioni il server interagisce (anche pesantemente) con il sistema operativo dell'host su cui è in esecuzione. Per esempio, il server può reperire informazioni di vario tipo dal file system da inviare poi, eventualmente, al client. L'esercizio che segue modella una elaborazione di questo tipo che il server dovrebbe effettuare.

Esercizio 1. Si scriva un programma C che elenca il contenuto della directory corrente e, per ogni file, stampa le seguenti informazioni aggiuntive

- nome del file
- numero dell'inode contenente le informazioni sul file
- lunghezza in byte della corrispondente entry della directory
- se si tratta di un file ordinario o di altro (e.g., directory etc ...)
- tempo dell'ultima modifica
- tempo dell'ultimo accesso
- taglia in byte del file.

L'output atteso è qualcosa del tipo (riporto solo alcune entry):

```
macbook-di-paolo-darco:ESERCIZI pd$ ./edir
Nome .
Inode number 3113242
Record lenght 12
. non e' un file ordinario
Modificato: Mar  8 15:48
Ultimo accesso: Mar  8 16:55
Taglia del file:   340 bytes
```

```
Nome ..
Inode number 1368315
Record lenght 12
.. non e' un file ordinario
Modificato: Mar  3 10:38
Ultimo accesso: Mar  8 15:32
Taglia del file:   2346 bytes

Nome basic.h
Inode number 3113258
Record lenght 16
basic.h e' un file ordinario
Modificato: Mar  3 10:38
Ultimo accesso: Mar  8 15:47
Taglia del file:   840 bytes


Nome edir.c
Inode number 3320817
Record lenght 16
edir.c e' un file ordinario
Modificato: Mar  8 15:47
Ultimo accesso: Mar  8 15:47
Taglia del file:   1970 bytes
```

Funzioni utili alla risoluzione dell'esercizio sono: *opendir, readdir, closedir, stat, localtime, strftime*. Le prime tre servono per aprire, leggere e chiudere una directory. La quarta per leggere le informazioni statistiche di un file. Le ultime due per leggere data e ora in una struttura opportuna e per convertire data e ora in una stringa. In particolare, le entrate di una directory sono definite dalla struttura

```
struct dirent
{
          long            d_ino;         /* numero inode */
          off_t           d_off;
          unsigned short  d_reclen;     /* lunghezza entry */
          char            d_name[NAME_MAX+1];
};
```

La struttura che prende in input (e riempie) la funzione *stat* è definita invece da

```
struct stat
{
              dev_t         st_dev;      /* device */
              ino_t         st_ino;      /* inode */
              umode_t       st_mode;     /* protection */
              nlink_t       st_nlink;    /* number of hard links */
              uid_t         st_uid;      /* user ID of owner */
              gid_t         st_gid;      /* group ID of owner */
              dev_t         st_rdev;     /* device type (if inode device) */
              off_t         st_size;     /* total size, in bytes */
```

```
            unsigned long st_blksize;  /* blocksize for filesystem I/O */
            unsigned long st_blocks;   /* number of blocks allocated */
            time_t        st_atime;    /* time of last access */
            time_t        st_mtime;    /* time of last modification */
            time_t        st_ctime;    /* time of last change */
};
```

Inoltre, la macro S_ISREG(st_mode) permette di capire se si tratta di un file regolare oppure di altro.
La funzione *localtime* (che prende in input un elemento di tipo time_t) restituisce un puntatore ad una struttura tm definita come segue

```
struct tm {
        int     tm_sec;         /* seconds after the minute [0-60] */
        int     tm_min;         /* minutes after the hour [0-59] */
        int     tm_hour;        /* hours since midnight [0-23] */
        int     tm_mday;        /* day of the month [1-31] */
        int     tm_mon;         /* months since January [0-11] */
        int     tm_year;        /* years since 1900 */
        int     tm_wday;        /* days since Sunday [0-6] */
        int     tm_yday;        /* days since January 1 [0-365] */
        int     tm_isdst;       /* Daylight Savings Time flag */
        long    tm_gmtoff;      /* offset from CUT in seconds */
        char    *tm_zone;       /* timezone abbreviation */
};
```

Infine la funzione *strftime* scrive in una stringa, in accordo al formato specificato dal terzo parametro, il tempo in una struttura di tipo tm, passata tramite il quarto parametro. Per esempio, se time_struct è un puntatore ad una struttura di tipo tm, la chiamata

```
strftime(string, sizeof(string), "%h %e %H:%M\n", time_struct);
```

scrive nella stringa data e ora nel formato, "mese, giorno, ore:minuti" e.g., Mar 7 18:55.