lunedì 29 ottobre 2018

Prima parte – Appunti e riassunti da lezioni/slides

Berkeley Socket

Sono un'API che definisce una libreria C per comunicazione inter-processo anche su rete. Standard per la realizzazione di applicazioni di rete.

```
int socket(int famiglia, int tipo, int protocollo);
```

Famiglia da usare: PF_INET

Serve ad utilizzare l'IPV4 Internet protocols.

Tipo da usare: SOCK_STREAM

E' un canale bidirezionale, sequenziale affidabile che opera su connessione. I dati vengono ricevuti e trasmessi come un flusso continuo.

Conversioni endianess: La suite di protocolli internet utilizza il big endian. Funzioni per convertire unsigned:

```
uint32_t htonl (uint32_t x)
uint16_t htons (uint16_t x)
uint32_t ntohl (uint32_t x)
uint16_t ntohs (uint16_t x)
```

h= Host

n= Network (Big endian)

l= Long

s= Short

Porta associata al servizio: L'istruzione memorizza nel campo sin_port della struct serveraddr l'intero 13 scritto in network order - 13 è la porta su cui risponde il server:

```
servaddr.sin_port = htons(13)
```

Conversione dell'ip del server: Questa funzione converte la stringa passata come 23secondo argomento in un indirizzo di rete scritto in network order e lo memorizza nella locazione di memoria puntata dal terzo argomento. Il nostro programma, quindi, dovrà specificare come argomento, l'indirizzo ip del server a cui fare la richiesta. La funzione restituisce un numero <=0 in caso di errore ed un numero positivo in caso di successo:

```
inet_pton(PF_INET, argv[1], &serveaddr.sin_addr)
```

Converte la stringa dell'indirizzo *dotted decimal in* nel numero IP in *network order:*

in_addr_t inet_addr(const char *strptr)

Converte la stringa dell'indirizzo *dotted decimal* in un indirizzo IP:

int inet_aton(const char *src, strict in_addr *dest)

Converte un indirizzo IP in una stringa *dotted decimal:*

char *inet_ntoa(struct in_addr addrptr)

Connect: Connette il socket *sockfd* all'indirizzo *serv_addr*. Il terzo argomento è la dimensione in byte della struttura. Il cast è necessario in quanto la funzione può essere utilizzata con diversi tipi di socket e strutture. Restituisce: 0 Successo -1 Errore.

connect(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr))

Riassunto struttura server TCP/IP

- Creo la struct: *struct sockaddr_in { sa_family_t famiglia, u_int16_t porta, struct in_addr sin_addr)*

  Famiglia: Ad esempio AF_INET;

  Porta: Porta in network order, 16bit;

  sin_addr: struct contenente l'indirizzo ip in network order. Volendo si può assegnare come indirizzo ip la macro INADDR_ANY: in questo caso l'applicazione accetterà connessioni da qualsiasi indirizzo associato al server.

- Dal momento che, a seconda dell'architettura, i dati possono essere memorizzati con codifica Big Endian o Little Endian, mentre la suite di protocolli internet utilizza solo **Big Endian**, sono **necessarie funzioni di conversione**.

  htonl(); Host to network long (indirizzi ip)

  htons(); Host to network short (port)

  Funzioni inverse

  ntohl; Network to host long

  ntohs; Network to host short

  inet_pton(PF_INET, "indirizzo ip", &servaddr.sin_addr)

  inet_pton: Converte l'indirizzo ip, preso in input come stringa in formato dotted, in un indirizzo di rete in network order. Restituisce <=0 in caso di errore. >0 caso suc.

- Creo il socket: *socket (int famiglia, int tipo, int protocollo)*

Famiglia: identifica la categoria di comunicazione. PF_INET si identificano le comunicazioni che seguono il protocollo IPV4 (PF_INET6 – IPV6)

Tipo: Ad esempio, con SOCK_STREAM si crea un canale bidirezionale, sicuro e affidabile in cui i dati sono trasmessi come un flusso di dati continuo (TCP), con SOCK_DGRAM si inviano pacchetti di dimensione fissata e senza necessità di stabilire una connessione

Protocollo: Future implementazioni, ad oggi è sempre posto a zero.

- Gli assegno un indirizzo: *bind(sockfd, (struct sockaddr*)&servaddr, sizeof(servaddr))*

Associa l'indirizzo specificato nel secondo parametro al descrittore del socket specificato dal primo parametro. Il terzo parametro è la dimensione in byte della struttura passata come secondo argomento. In TCP fallisce se la porta è già occupata e restituisce 0 in caso di successo e -1 in caso di insuccesso.

- Mi metto in ascolto: *listen(sockfd, lunghezza_coda)*

Il processo che utilizza questa funzione si mette in ascolto di nuove connessioni sul descrittore sockfd. Il secondo argomento specifica la lunghezza della coda di attesa per le connessioni

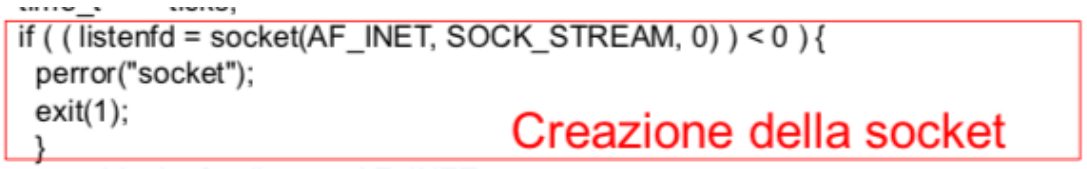
- Accetto una nuova connessione: *accept(sockfd, (struct sockaddr*)&clientaddr, len)*

Accetta le connessioni arrivate sul descrittore che si era messo in ascolto. Il secondo e il terzo parametro sono utilizzati per memorizzare l'indirizzo del client che ha effettuato la connessione e possono essere posti a NULL. In caso di errore restituisce -1, in caso di successo restituisce un nuovo descrittore che sarà utilizzato per la comunicazione, mentre quello vecchio (sockfd) resta ancora in ascolto.

- Chiudo il socket: *close(…)*

ESEMPIO DI APPLICAZIONE SERVER

Creazione della socket:



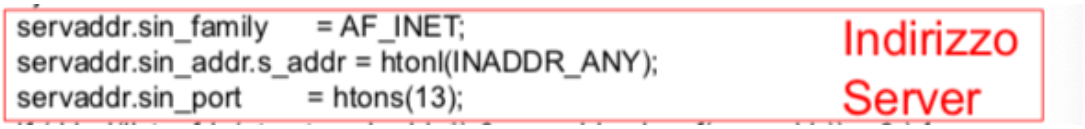
perror: Produce un messaggio sullo standard error che descrive l'ultimo errore avvenuto durante una System call o una funzione di libreria. Se l'argomento passato non è NULL, viene stampato prima del messaggio d'errore seguito da ':'.

void perror(const char *s)

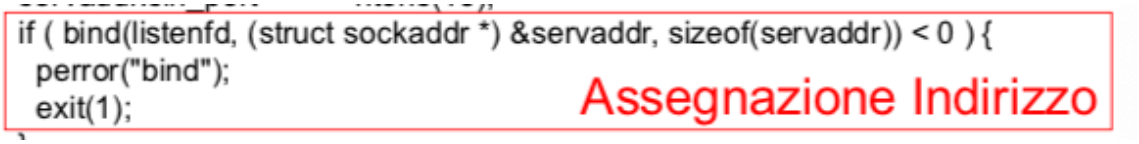
INADDR_ANY: Viene utilizzato come indirizzo del server, l'applicazione accetterà connessioni da qualsiasi indirizzo associato al server.

servaddr.sin_addr.s_addr= htonl(INADDR_ANY);

Indirizzo server:



Assegnazione Indirizzo:



Bind: Assegna l'indirizzo addr al socket sockfd, addr è un sockaddr di tipo generico. Nei socket TCP fallisce se la porta è in uso. Addrlen è sizeof del secondo argomento. Restituisce 0 o -1.

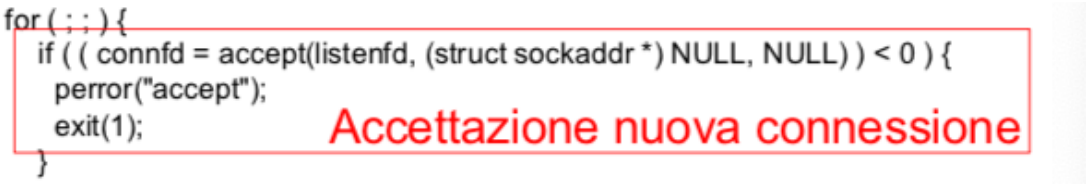
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen)

Messa in ascolto:

Listen: Mette il socket in modalità di ascolto in attesa di nuove connessioni. Il secondo argomento specifica quante connessioni possono essere in attesa di essere accettate. Restituisce 0 o -1.

int listen(int sockfd, int lunghezza_coda)

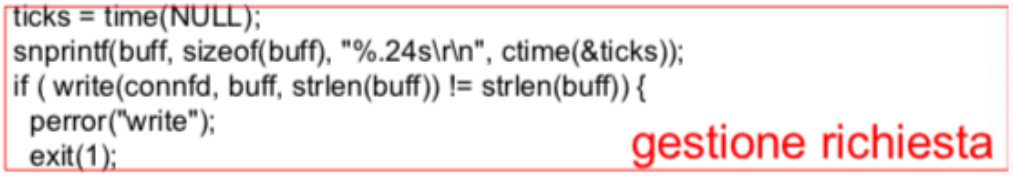
Accettazione nuova connessione:



Accept: Il secondo e terzo argomento servono ad identificare il client, possono essere NULL. Restituisce un nuovo descrittore o -1. Il nuovo Socket è associato alla nuova connessione e il vecchio resta in ascolto.

int accept(int sockfd, struct sockaddr *clientaddr, socklen_t *addr_dim)

Gestione richiesta:



Close: Per terminare una connessione TCP si utilizza la System call *close*. Una volta invocato il descrittore del socket non è più utilizzabile dal processo per operazioni di lettura o scrittura.

DIFFERENZA CON APPLICAZIONE CLIENT & SERVER

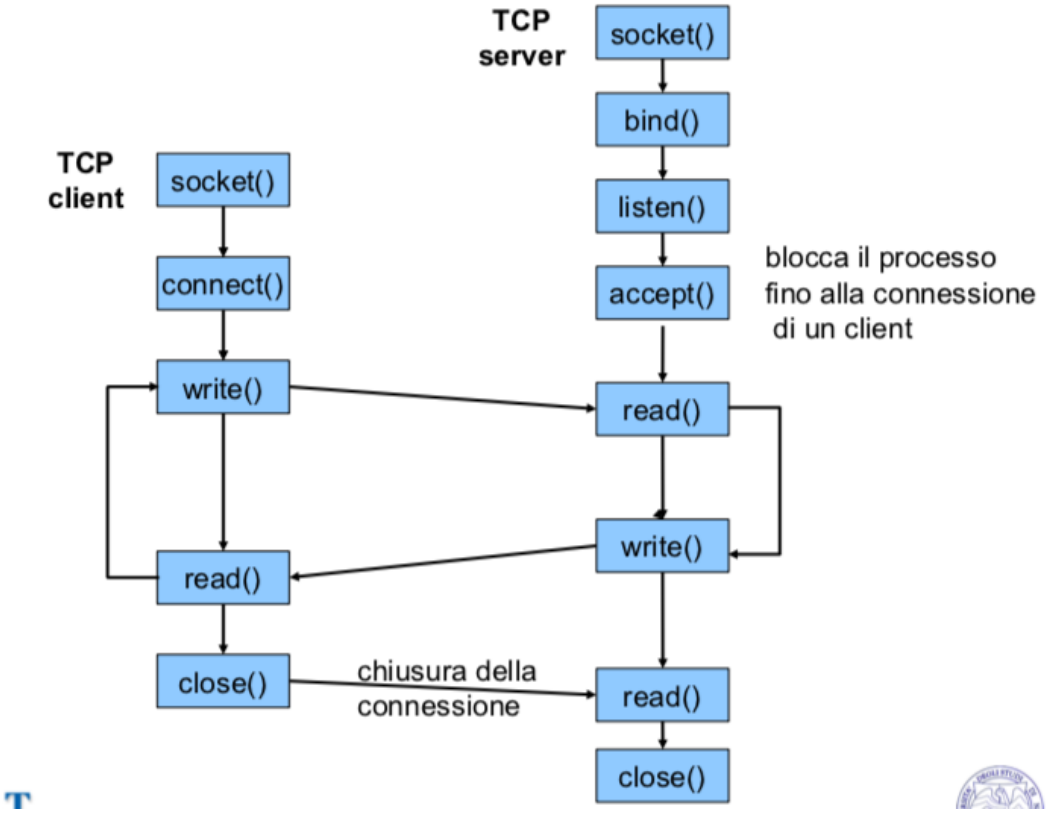
Client

1. Creo la struct;
2. *Socket(…);*
3. *Bind(…);*
4. *Connect(…);*
5. *Close(…);*

Server

1. Creo la struct;
2. *Socket(…);*
3. *Bind(…);*
4. *Listen(…);*

5. *Accept(…);*
6. *Close(…);*



Schema TCP Client-Server

Funzioni wrapper: Nei programmi reali è necessario verificare la condizione di uscita di ogni chiamata a funzione. Spesso gli errori determinano la necessità di terminare l'esecuzione. Per migliorare la leggibilità del codice si possono definire funzioni wrapper che chiamano la funzione, ne verificano l'uscita e terminano l'esecuzione in caso di errore.

Write: Si usa per scrivere su un socket. Write restituisce il numero di byte scritti. Può accadere che si scrivano meno bytes di quelli richiesti. Sono necessarie chiamate successive. Fullwrite scrive esattamente count byte s iterando opportunamente lo scritture

```
ssize_t write(int fd, const void *buf, size_t count)
```

Read: Si usa per leggere da un socket. La read blocca l'esecuzione qualora non ci siano dati da leggere ed il processo resta in attesa di dati. E' normale ottenere meno bytes di quelli richiesti. Ottenere 0 bytes significa che il socket è vuoto ed è stato chiuso.

```
ssize_t read(int fd, void *buf, size_t count)
```

Full Write: Scrive esattamente count byte s iterando opportunamente le scritture. Scrive anche se viene interrotta da una System Call.

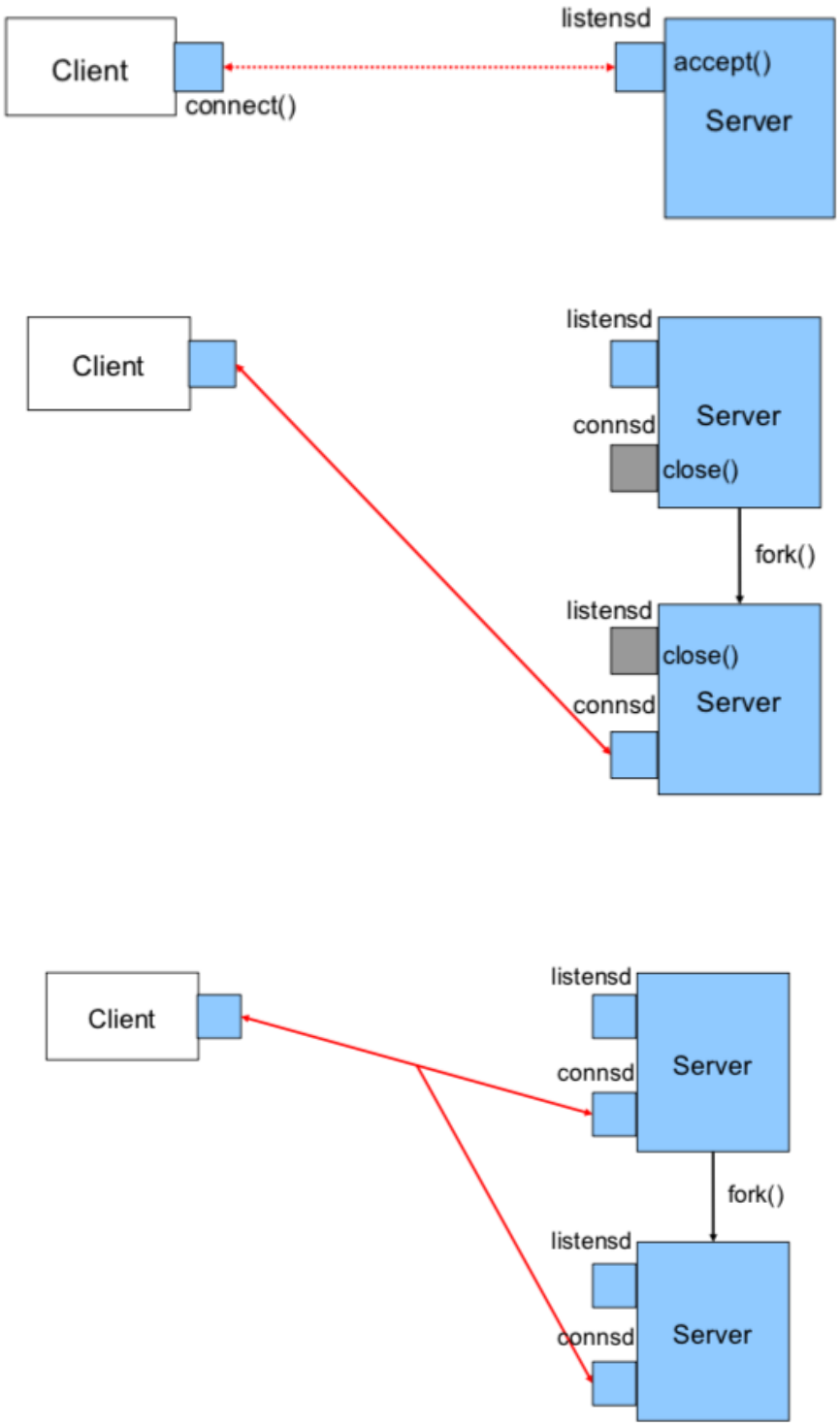
```
ssize_t FullWrite(int fd, const void *buf, size_t count)
{
    size_t nleft;
    ssize_t nwritten;
    nleft = count;
    while (nleft > 0) {             /* repeat finche non ci sono left */
        if((nwritten = write(fd,buf,nleft))<0) {
            if(errno == EINTR){ /* Se si verifica una System Call che interrompe */
                continue; //ripeti il ciclo
            }
            else{ /* Se non è una System Call, esci con un errore */
                exit(nwritten);
            }

            nleft-= nwritten;
            buf += nwritten;

        }
        return (nleft);
    }
```

Full Read: Identico alla Full Write, ma legge.

```
ssize_t FullRead(int fd, void *buf, size_t count)
{
    size_t nleft;
    ssize_t nread;
    nleft = count;
    while (nleft > 0) {             /* repeat finche non ci sono left */
        if((nread = read(fd,buf,nleft))<0){
            if(errno == EINTR){ /* Se si verifica una System Call che interrompe */
                continue; /* Ripeti il ciclo */

            }else{ /* Se non è una System Call, esci con un errore */
                exit(nread);
            }

        }else if(nread == 0){ /* Se sono finiti */
            break; /* esci */

        }
        nleft-= nread;
        buf +=  nread;

    }
    buf = 0;
    return (nleft);
}
```

Server concorrenti: Gestiscono più connessioni contemporaneamente. Utilizzano una seconda istanza di se stessi per gestire le connessioni client. Si utilizza la System call fork() per generare un processo figlio. I processi server padre e figli vengono eseguiti contemporaneamente.

Il processo figlio gestisce la connessione con un dato client mentre il processo padre può accettare nuove connessioni. Ogni nuova connessione, genera un nuovo processo figlio che gestisce le richieste del client.



fork: Crea un nuovo processo figlio copia esatta del processo chiamante (padre). Eredita i descrittori del processo padre. Restituisce un diverso valore al padre e al figlio:

- Al padre restituisce il pid del figlio;
- Al figlio restituisce 0.

pid_t fork(void);

Terminazione processo figlio: Quando un processo figli termina:

- Viene inviato il segnale SIGCHLD al padre;

```
Socket(...); Bind(...); Listen(...);
while (1) {
connsd = Accept(listensd, NULL, NULL);
 if ( (pid = fork() ) == 0 ) { /* processo figlio */
   close(listensd); /* chiude listensd interagisce con il client
                       tramite la connessione con connsd */
   ...
   exit(0); /* Terminazione del figlio */
   }/* il processo padre chiude connsd e ripete il ciclo */
close(connsd);
}
```

connsd – e' il socket della connessione con il client
listensd – e' il socket in attesa di connessioni

```
27      /* fork to handle connection */
28      if ( (pid = fork ()) < 0 ){
29          perror (" fork error ");
30          exit ( -1);
31      }
32      if (pid == 0) { /* child */
33          close ( list_fd );
34          timeval = time ( NULL );
35          snprintf (buffer , sizeof ( buffer ), " %.24 s\r\n", ctime (&
timeval ));
36          if ( ( write ( conn_fd , buffer , strlen ( buffer ))) < 0 ) {
37              perror (" write error ");
38              exit ( -1);
39          }
```

```
40          if ( logging ) {
41                  inet_ntop ( AF_INET , & client . sin_addr , buffer , sizeof (
buffer ));
42                  printf (" Request from host %s, port %d\n", buffer ,
43                  ntohs ( client . sin_port ));
44          }
45      close ( conn_fd );
46      exit (0);
47 } else { /* parent */
48      close ( conn_fd );
49      }
50 }
51 /* normal exit , never reached */
52 exit (0);
53 }
```

- Il processo diventa “zombie”.

Gli zombie sono processi che hanno terminato l’esecuzione ma restano presenti nella tabella dei processi. In genere possono essere identificati dall’output del comando ps per la presenza di una Z nella colonna di stato.

Segnali: Comunicazione asincrona tra processi. Insieme fissato di segnali a cui corrispondono delle azioni di default (man 7 signal). E’ possibile fare in modo che quando il destinatario riceve un segnale venga eseguita una procedura specifica (handler).

Handler: Un handler (gestore) è una funzione del tipo:

```
void funzione(int num_segnale){
    printf(“%d”, num_segnale);
}
```

Una volta che l’handler termina, l’esecuzione del processo riprende dal punto in cui era stato interrotto.

Catturare un segnale: Imposta la funzione handit come handler del segnale SIGINT.

```
signal(SIGINT, handit)
```

E’ possibile ignorare un segnale. In questo modo i processi figli non restino nella condizione di zombie una volta terminati. Non è conforme allo standard POSIX.

```
signal(SIGINT, SIG_IGN)
```

Oppure ritornare alla reazione di default

```
signal(SIGINT, SIG_DFL)
```

Opzioni del socket: Ogni socket aperto ha delle proprietà che ne determinano alcuni comportamenti. Le opzioni del socket consentono di modificare tali proprietà. Ogni opzione ha un valore di default: Alcune opzioni sono binarie (0 - 1), altre hanno un valore (int o strutture).

Funzioni gestsockopt e setsockopt: Sono funzioni per la gestione delle opzioni di un socket. Da utilizzare dopo socket() e prima di bind().

```
Int getsockopt(int sd, int level, int optname, void*optval, socklen_t optlen);

Int setsock opt(int sd, int level, int optname, const void* optval, socklen_t* option)
```

Opzioni di Livello Socket

- SO_BROADCAST permette il broadcast
- SO_DEBUG abilita le informazioni di debug
- SO_DONTROUT Esalta il lookup nella tavola di routing
- SO_ERROR legge l’errore corrente
- SO_KEEPALIVE controlla che la connessione sia attiva
- SO_LINGER controlla la chiusura della connessione
- SO_RCVBUF grandezza del buffer in ricezione
- SO_SNDBUF grandezza buffer in spedizione
- SO_RCVLOWAT soglia per il buffer in ricezione
- SO_SNDLOWAT soglia per il buffer in spedizione
- SO_RCVTIMEO timeout per la ricezione
- SO_SNDTIMEO timeout per la spedizione
- **SO_REUSEADDR permette riutilizzo indirizzi locali**
- SO_REUSEPORT permette riutilizzo porte locali
- SO_TYPE il tipo di socket
- SO_USELOOPBACK per i socket di routing (copia i pacchetti)

Il primo parametro identifica il socket da cui leggere\scrivere su cui settare l’opzione.

Il secondo parametro identifica il livello (ad esempio livello socket, livello ip e così ..).

Il terzo parametro indica il nome dell'opzione da prendere in considerazione (es: SO_REUSEADDR).

Il quarto parametro indica il valore da settare o cui inserire in lettura dell'opzione. Può essere binario o meno.

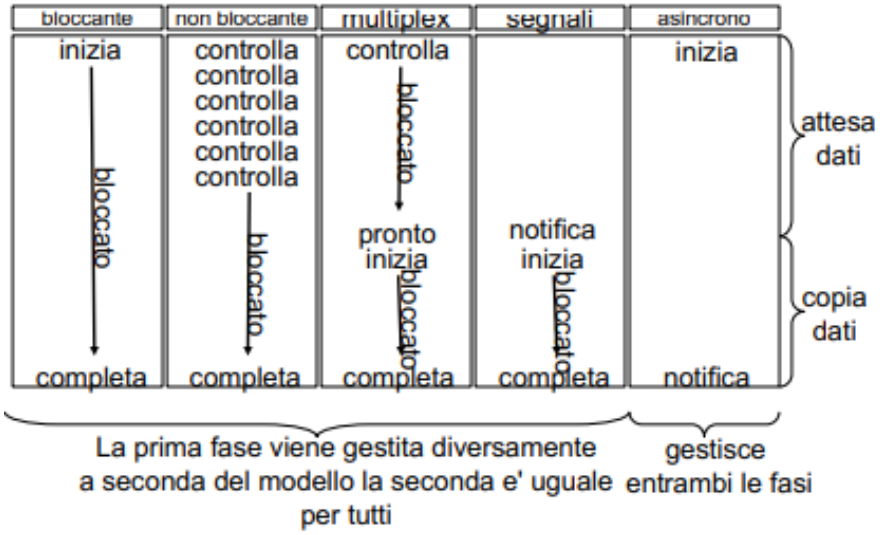
Il quinto parametro indica la dimensione del valore dell'opzione.

```
Int enable=1;

setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &enable, sizeof(int))
```

SO_REUSEADDR permette il riutilizzo di indirizzi locali.

Gestione dell' input: Mentre l'applicazione è bloccata in operazione di lettura non si accorge di cosa accade.

Lotta all'ultimo sangue tra i diversi tipi di server



I/O Bloccante: Modello predominante, il processo attende durante entrambi le fasi. L'esecuzione del processo utente si blocca nella chiamata a read e riprende l'esecuzione solo quando questa viene soddisfatta o si verifica un errore. Lo stato del processo in attesa è ***waiting***. I socket di **default sono bloccanti.**

I/O Non bloccante: Un socket non bloccante restituisce un errore ogni volta che si richiede un'operazione di I/O non ancora effettuabile. Solitamente la richiesta viene reiterata fino a quando non si ottiene una risposta positiva. Questa pratica viene **chiamata polling**. Il polling è considerato uno spreco di tempo per la CPU. Solitamente viene utilizzato nei sistemi dedicati.

Multiplex: Il processo rimane in attesa di eventi su uno o più descrittori: l'esecuzione si blocca fino a quando uno dei descrittori diventa pronto. Il vantaggio nell'uso di questo modello è che si possono **monitorare più canali di comunicazione.**

I/O Controllato da segnali: Il kernel notifica al processo in esecuzione che il canale di comunicazione **è pronto inviando il segnale SIGIO**. Il processo deve impostare un handler per SIGIO (ovvero una funzione che viene chiamata quando viene ricevuto il segnale SIGIO).

I/O Asincrono: E’ definito dallo standard POSIX. Il kernel si prende carico di entrambi le fasi dell’operazione I/O. Una volta che l’operazione è stata completata viene inviato un segnale al processo. Il processo avvia l’operazione comunicando al kernel il descrittore, il buffer e il segnale con cui notificare il completamento dell’opzione.

Seconda parte – Riassunti dal GAPIL

Three-way handshake

E' il processo che porta a creare e stabilire una connessione TCP:

1. Il server deve essere preparato per accettare le connessioni in arrivo; il procedimento si chiama apertura passiva del socket (*passive open*). Questo viene fatto chiamando: *socket, bind* e *listen*. Completato il *passive open*, il server chiama la funzione ***accept*** e il processo si blocca in attesa di connessioni.

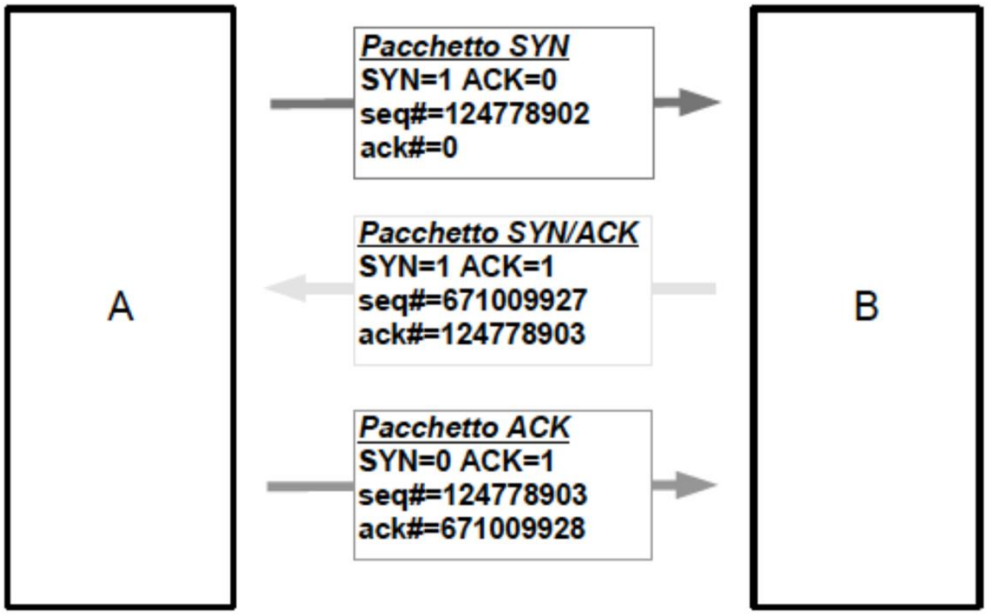
2. Il client richiede l'inizio della connessione usando la funzione ***connect****,* attraverso un procedimento chiamato apertura attiva (*active open*). La chiamata di *connect* blocca il processo a causa l'invio da parte del client di un segmento **SYN**, in sostanza viene inviato al server un pacchetto IP che contiene solo gli header IP e TCP (con il numero di sequenza iniziale e il flag SYN).

3. Il server deve dare ricevuto (*acknowledge)* del SYN del client, inoltre anche il server deve inviare il suo SYN al client (e trasmettere il suo numero di sequenza iniziale) questo viene fatto ritrasmettendo un singolo segmento in cui sono impostati entrambi i flag **SYN** e **ACK.**

4. Una volta che il client ha ricevuto l'acknowledge dal server la funzione *connect* ritorna, l'ultimo passo è dare il ricevuto del SYN del server inviando un **ACK**. Alla ricezione di quest'ultimo, la funzione *accept* del server ritorna e la connessione è stabilita.

I numeri di sequenza si utilizzano per la connessione affidabile, infatti, il protocollo TCP, prevede nell'header la presenza di un numero a 32bit (*sequence number*) che identifica a quale byte nella sequenza del flusso corrisponde il primo byte della sezione dati contenuta nel segmento.

Il numero di sequenza di ciascun segmento viene calcolato a partire da un numero di sequenza iniziale generato in maniera casuale dal kernel all'inizio della connessione e trasmesso con il SYN; l'acknowledgement di ciascun segmento viene effettuato dall'altro capo della connessione impostando il flag ACK e restituendo nell'apposito campo dell'header un acknowledge number pari al numero di sequenza che il ricevente si aspetta di ricevere con il pacchetto successivo; dato che il primo pacchetto SYN consuma un byte, nel three way handshake il numero di acknowledge è sempre pari al numero di sequenza iniziale incrementato di uno.



Le strutture degli indirizzi dei socket

Nella creazione di un socket non si specifica nulla oltre al tipo di famiglia di protocolli che si vuole utilizzare, in particolare nessun indirizzo che identifichi i due capi della comunicazione. La funzione si limita ad allocare nel kernel quanto necessario per poter realizzare la comunicazione. Gli indirizzi infatti vengono specificati attraverso apposite strutture che vengono utilizzate dalle altre funzioni della interfaccia dei socket, quando la comunicazione viene effettivamente realizzata.

La struttura generica

Le strutture degli indirizzi vengono sempre passate alle varie funzioni attraverso puntatori (reference), ma le funzioni devono poter maneggiare puntatori a strutture relative a tutti gli indirizzi possibili nelle varie famiglie di protocolli; questo pone il problema di come passare questi puntatori, il C moderno risolve questo problema con i puntatori generici (void *), ma l'interfaccia dei socket è antecedente alla definizione dello standard ANSI C, e per questo nel '82 fu scelto di definire una struttura generica per gli indirizzi dei socket, *sockaddr*.

La struttura degli indirizzi IPv4

I Socket di tipo *PF_INET* vengono usati per la comunicazione attraverso internet. L'indirizzo di un socket internet comprende l'indirizzo internet di un'interfaccia più un numero di porta. Il protocollo IP non prevede numeri di porta, che sono utilizzati solo dai protocolli di livello superiore (TCP/UDP).

```
struct sockaddr_in {
    sa_family_t     sin_family; /* address family: AF_INET */
    in_port_t       sin_port;   /* port in network byte order */
    struct in_addr  sin_addr;   /* internet address */
};
/* Internet address. */
struct in_addr {
    in_addr_t       s_addr;     /* address in network byte order */
};
```

- *Il membro sin_family* deve essere sempre impostato a *AF_INET,* altrimenti si avrà un errore di *EINVAL;*
- Il membro *sin_port* specifica il numero di porta. I numeri di porta sotto il 1024 sono chiamati *riservati* in quanto utilizzati da servizi standard e soltanto processi con i privilegi di amministratori possono utilizzare la funzione *bind*;
- Il membro *sin_addr* contiene un indirizzo internet, e viene acceduto sia come struttura che direttamente come un intero.

Infine occorre sottolineare che sia gli indirizzi che i numeri di porta **devono essere specificati in network order**, cioè con i bit in formato *big endian*.

Le funzioni di conversione degli indirizzi

Si utilizzano funzioni di conversione che servono a tener conto, automaticamente, della possibile differenza fra l'ordinamento usato sul computer e quello che viene usato nelle trasmissione sulla rete; queste funzioni sono *htonl, htons, ntohl* e *ntohs*. I rispettivi prototipi sono:

```
#include <netinet/in.h>
unsigned long int htonl(unsigned long int hostlong)
    Converte l'intero a 32 bit hostlong dal formato della macchina a quello della rete.
unsigned short int htons(unsigned short int hostshort)
    Converte l'intero a 16 bit hostshort dal formato della macchina a quello della rete.
unsigned long int ntohl(unsigned long int netlong)
    Converte l'intero a 32 bit netlong dal formato della rete a quello della macchina.
unsigned sort int ntohs(unsigned short int netshort)
    Converte l'intero a 16 bit netshort dal formato della rete a quello della macchina.
Tutte le funzioni restituiscono il valore convertito, e non prevedono errori.
```

La lettera **n** si utilizza per indicare l'ordinamento usato sulla rete (*network order*) e la lettera **h** per l'ordinamento usato nella macchina locale (*host order*), mentre le lettere **s** e **l** stanno ad indicare i tipi di dato (*long* o *short*).

Usando queste funzioni si ha la conversione automatica: nel caso la macchina che si sta usando abbia un architettura *big endian* queste funzioni sono definite **macro** che non fanno nulla. Per questo motivo vanno sempre utilizzate, in modo da assicurare la portabilità del codice su tutte le architetture.

Le funzioni inet_aton, inet_addr e inet_ntoa

Un secondo insieme di funzioni di manipolazione serve per passare dal formato binario usato nelle strutture degli indirizzi alla rappresentazione simbola dei numeri IP che si usa normalmente. Le prime tre funzioni di manipolazione riguardano la conversione degli indirizzi IPv4 da una stringa in cui il numero di IP è espresso in notazione *dotted-decimal* (192.168…) al formato binario (*network order*) e viceversa; Si usa la lettera **a** per indicare le stringhe:

```
#include <arpa/inet.h>
in_addr_t inet_addr(const char *strptr)
     Converte la stringa dell'indirizzo dotted decimal in nel numero IP in network order.
int inet_aton(const char *src, struct in_addr *dest)
     Converte la stringa dell'indirizzo dotted decimal in un indirizzo IP.
char *inet_ntoa(struct in_addr addrptr)
     Converte un indirizzo IP in una stringa dotted decimal.
```

Tutte queste le funzioni non generano codice di errore.

- ***inet_addr*** restituisce l'indirizzo a 32 bit in network order a partire dalla stringa passata nell'argomento *strptr*. In caso di erorre restituisce il valore *INADDR_NONE* che tipicamente sono tradue bit. Questo però comorta che la stringa è 255.255.255.255, che pure è un indirizzo valido, non può essere usata con questa funzione; per questo motivo è generalmente deprecata a favore di *inet_aton*
- **inet_aton** converte la stringa puntata da *src* nell'indirizzo binario che viene memorizzato nell'opportuna struttura ***in_addr*** situata all'indirizzo dato dall'argomento ***dest***. La funzione restituisce **0 in caso di successo e 1 in caso di fallimento.**
- **inet_ntoa** converte il valore a 32 bit dell'indirizzo (in *network order*) restituendo il puntatore alla stringa che contiene l'espressione in formato *dotted decimal*.

Le funzioni inet_pton e inet_ntop

Le tre funzioni precedenti sono limitate solo ad indirizzi IPv4, per questo motivo è preferibile usare le due funzioni ***inet_pton*** e ***inet_ntop*** che possono convertire anche gli indirizzi IPv6. Le lettere **n** e **p** sono per: ***numeric*** e ***presentation***.

La prima funzione, *inet_pton*, serve a convertire una stringa in un indirizzo:

```
#include <sys/socket.h>
int inet_pton(int af, const char *src, void *addr_ptr)
     Converte l'indirizzo espresso tramite una stringa nel valore numerico.
```

La funzione restituisce un valore negativo se `af` specifica una famiglia di indirizzi non valida, con `errno` che assume il valore `EAFNOSUPPORT`, un valore nullo se `src` non rappresenta un indirizzo valido, ed un valore positivo in caso di successo.

La funzione converte la stringa tramite ***src*** nel valore numerico dell'indirizzo IP del tipo specificato ***AF*** che viene memorizzato all'indirizzo puntato da ***addr_ptr,*** la funzione restituisce un **valore positivo in caso di successo, nullo se la stringa non rappresenta un indirizzo valido, e negativo se AF specifica una famiglia non valida.**

La seconda funzione di conversione è *inet_ntop* che converte un indirizzo in una stringa:

```
#include <sys/socket.h>
char *inet_ntop(int af, const void *addr_ptr, char *dest, size_t len)
```

Converte l'indirizzo dalla relativa struttura in una stringa simbolica.

La funzione restituisce un puntatore non nullo alla stringa convertita in caso di successo e `NULL` in caso di fallimento, nel qual caso `errno` assume i valori:

`ENOSPC` le dimensioni della stringa con la conversione dell'indirizzo eccedono la lunghezza specificata da `len`.

`ENOAFSUPPORT` la famiglia di indirizzi `af` non è una valida.

La funzione converte la struttura dell'indirizzo puntata da ***addr_ptr*** in una stringa che viene copiata nel buffer puntato dall'indirizzo ***dest*** ; questo deve essere preallocato dall'utente e la lunghezza deve essere almeno *INET_ADDRSTRLEN* in caso di indirizzi IPv4; la lunghezza del buffer deve essere specifiata attraverso il parametro ***len****.*

Gli indirizzi vengono convertiti da/alle rispettive strutture di indirizzi, che devono essere precedentemente allocate e passate attraverso il puntatore *addr_ptr*; l'argomento dest di *inet_ntop* non può essere nullo e deve essere allocato precedentemente.

Socket Pair

Data una connessione TCP si suole chiamare *socket pair* la combinazione dei quattro numeri che definiscono i due capi della connessione cioè l'indirizzo IP locale e la porta TCP locale, e l'indirizzo IP remoto e la porta TCP remota. Questa combinazione, che scriveremo usando una notazione del tipo (1.1.1.1:2.3.3.3), identifica univocamente una connessione su internet.

Le funzioni di base per la gestione dei socket

Funzione *socket*

La creazione di un socket avviene attraverso l'uso della funzione *socket*; essa restituisce un ***file descriptor*** che serve come riferimento al socket; Si noti che la creazione del socket si limita ad allocare opportune strutture nel kernel e non comporta nulla riguardo all'indicazione degli indirizzi remoti o locali attraverso i quali si vuole effettuare la comunicazione.

Domain

Dati i tanti e diversi protocolli di comunicazione disponibili, esistono vari tipi di socket, che vengono classificati raggruppandoli in quelli che si chiamano *domini*. La scelta di un dominio equivale in sostanza alla scelta di una famiglia di protocolli, e viene effettuata attraverso l'argomento domain della funzione socket. Ciascun dominio ha un suo nome simbolico che convenzionalmente è indicato da una costante che inizia per PF (*protocol family)*. A ciascun tipo di dominio corrisponde un analogo nome simbolico, anch'esso associato ad una costante, che inizia invece per AF (*andress family*) che identifica il formato degli indirizzi usati in quel dominio. L'idea alla base dalla distinzione fra questi due insiemi di costanti che era una famiglia di protocolli potesse supportare vari tipi di indirizzi, per cui il prefisso PF si sarebbe dovuto usare nella creazione dei socket e il prefisso AF in quello delle strutture degli indirizzi. Non tutte le famiglie di protocolli sono utilizzabili dall'utente generico, ad esempio tutti i socket di tipo *SOCK_RAW* possono essere creati solo da processi che hanno privilegi di amministratore.

Tipo

La scelta di un dominio non comporta però la scelta dello stile di comunicazione, questo infatti viene a dipendere dal protocollo che si andrà ad utilizzare fra quelli disponibili nella famiglia scelta. L'interfaccia dei socket premette di scegliere lo stile di comunicazione indicando il tipo di socket con l'argomento type di socket.

Elenco costanti (tipi) utilizzati in rete:

- *SOCK_STREAM*: Provvede un canale di trasmissione dati bidirezionale, sequenziale e affidabile. Opera su una connessione con un altro socket. I dati vengono ricevuti e trasmessi come un flusso continuo di byte (*stream*) e possono essere letti in blocchi di dimensioni qualunque.
- *SOCK_DGRAM*: Viene usato per trasmettere pacchetti di dati (*datagram*) di lunghezza massima prefissata, indirizzati singolarmente. Non esiste una connessione e la trasmissione è effettuata in maniera non affidabile (UDP).

Funzione *bind*

La funzione bind assegna un indirizzo locale ad un socket. E' usata cioè per specificare la prima parte della *socket pair*. Viene usata sul lato server per specificare la porta su cui poi ci si porrà in ascolto.

```
#include <sys/socket.h>
int bind(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen)
```

Assegna un indirizzo ad un socket.

La funzione restituisce 0 in caso di successo e -1 per un errore; in caso di errore la variabile `errno` viene impostata secondo i seguenti codici di errore:

`EBADF`	il file descriptor non è valido.
`EINVAL`	il socket ha già un indirizzo assegnato.
`ENOTSOCK`	il file descriptor non è associato ad un socket.
`EACCES`	si è cercato di usare una porta riservata senza sufficienti privilegi.
`EADDRNOTAVAIL`	il tipo di indirizzo specificato non è disponibile.
`EADDRINUSE`	qualche altro socket sta già usando l'indirizzo.

ed anche `EFAULT` e per i socket di tipo `AF_UNIX`, `ENOTDIR`, `ENOENT`, `ENOMEM`, `ELOOP`, `ENOSR` e `EROFS`.

Il primo argomento è un file descriptor ottenuto da una precedente chiamata a socket, mentre il secondo e terzo argomento sono l'indirizzo (locale) del socket e la dimensione della struttura che lo contiene.

La chiamata *bind* permette di specificare l'indirizzo, la porta, entrambi o nessuno dei due. In genere i server utilizzano una porta nota che assegnano all'avvio, se questo non viene fatto è il kernel a scegliere una porta effimera quando vengono eseguite le funzioni *connect* e *listen* ma se questo è normale per il client, non lo è per il server che in genere viene identificato dalla porta sui cui risponde.

Con *bind* si può assegnare un indirizzo IP specifico ad un socket, purché questo appartenga ad una interfaccia della macchina. Per un client TCP questo diventerà l'indirizzo sorgente usato per i tutti pacchetti inviati sul socket, mentre per un server TCP questo restringerà l'accesso al socket solo alle connessioni che arrivano verso tale indirizzo.

Per specificare un indirizzo generico, con IPv4 si usa il valore ***INADDR_ANY*** (Indirizzo generico 0.0.0.0). Esistono anche altre costanti: *INADDR_BROADCAST, INADDR_LOOPBACK.*

Funzione *connect*

La funzione connect è usata da un client TCP per stabilire la connessione con un server TCP, il prototipo della funzione è il seguente:

```
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen)
```

Stabilisce una connessione fra due socket.

La funzione restituisce zero in caso di successo e -1 per un errore, nel qual caso `errno` assumerà i valori:

`ECONNREFUSED` non c'è nessuno in ascolto sull'indirizzo remoto.

`ETIMEDOUT` si è avuto timeout durante il tentativo di connessione.

`ENETUNREACH` la rete non è raggiungibile.

`EINPROGRESS` il socket è non bloccante (vedi sez. 12.2.1) e la connessione non può essere conclusa immediatamente.

`EALREADY` il socket è non bloccante (vedi sez. 12.2.1) e un tentativo precedente di connessione non si è ancora concluso.

`EAGAIN` non ci sono più porte locali libere.

`EAFNOSUPPORT` l'indirizzo non ha una famiglia di indirizzi corretta nel relativo campo.

`EACCES`, `EPERM` si è tentato di eseguire una connessione ad un indirizzo *broadcast* senza che il socket fosse stato abilitato per il *broadcast*.

altri errori possibili sono: `EFAULT`, `EBADF`, `ENOTSOCK`, `EISCONN` e `EADDRINUSE`.

Il primo argomento è un file descriptor ottenuto da una precedente chiamata a *socket*, mentre il secondo e il terzo argomento sono rispettivamente l'indirizzo e la dimensione della struttura che contiene l'indirizzo del socket.

Nel caso di socket TCP la funzione ***connect*** avvia il **three way handshake**, e ritorna solo quando la connessione è stabilita o si è verificato un errore. Se si fa riferimento al diagramma degli stati del TCP, la funzione connect, porta un socket dallo stato *CLOSED* prima allo stato *SYN_SEND* e poi, al ricevimento del *ACK,* nello stato ESTABLISHED. Se invece la connessione fallisce il socket non è più utilizzato e va chiuso.

Si noti infine che con la funzione connect si è specificato solo indirizzo e porta del server, quindi solo una metà della socket pair; essendo questa funzione usata nel client l'altra metà contente indirizzo e porta locale viene lasciata all'assegnazione automatica del kernel, e non è necessario effettuare una bind.

Funzione *listen*

La funzione *listen* serve ad usare un socket in modalità passiva, cioè, come dice il nome, per metterlo in ascolto di eventuali connnessioni; in sostanza l'effetto della funzione è di portare il socket dallo stato *CLOSED* a quello *LISTEN.* In genere si chiama la funzione in un server dopo le chiamate a *socket* e *bind* e prima della chiamata ad *accept.*

```
#include <sys/socket.h>
int listen(int sockfd, int backlog)
```

Pone un socket in attesa di una connessione.

La funzione restituisce 0 in caso di successo e -1 in caso di errore. I codici di errore restituiti in `errno` sono i seguenti:

`EBADF`	l'argomento `sockfd` non è un file descriptor valido.
`ENOTSOCK`	l'argomento `sockfd` non è un socket.
`EOPNOTSUPP`	il socket è di un tipo che non supporta questa operazione.

La funzione pone il socket specificato da *sockfd* in modalità passiva e predispone una coda per le connessioni in arrivo di lunghezza pari a *backlog* . La funzione si può applicare solo a socket di tipo *SOCK_STREAM* o *SOCK_SEQPACKET.*

L'argomento backlog indica il numero massimo di connessioni pendenti accetate; se esso viene ecceduto il client al momento della richiesta della connessione riceverà un errore di tipo *ECONNREFUSED*, o se il protocollo, come accade nel caso del TCP, supporta la ritrasmissione, la richiesta sarà ignorata in modo che la connessione possa venire ritentata.

Per ogni socket in ascolto vengono mantenute due code:

1. La **coda delle connessioni incomplete**: contiene un riferimento per ciascun socket per il quale è arrivato un *SYN* ma il *three way handshake* non si è ancora concluso. Questi socket sono nello stato *SYN_RECV*
2. La **coda delle connessioni complete**: contiene un ingresso per ciascun socket per il quale *three way handshake* è stato completato ma ancora *accept* non è ritornata. Questi socket sono nello stato *ESTABLISHED*

Storicamente il valore dell'argomento ***backlog*** era corrispondente al massimo valore della somma del numero di voci possibili per ciascuna delle due code. In Linux (da v2.2) per prevenire l'attacco chiamato SYN flood (basato sull'emissione da parte dell'attaccante di un grande numero di pacchetti SYN indirizzati verso una porta, così che i SYN+ACK vanno perduti e la code delle connessioni incomplete viene saturata, impedendo di fatto ulteriori connessioni). Per ovviare a questo, il significato del backlog, è stato cambiato a indicare la lunghezza della coda delle connessioni complete. La lunghezza della coda delle connessioni incomplete può essere ancora controllata usando la funzione ***sysctl***. La scelta storica per il valore di questo parametro era di 5, e alcuni vecchi kernel non supportavano valori superiori, ma la situazione è cambiata per via della presenza di server web che devono gestire un gran numero di connessioni per cui un tale valore non è più adeguato. Non esiste comunque una risposta univoca per la scelta del valore, per questo non conviene specificarlo con una costante.

Funzione *accept*

La funzione *accept* è chiamata da un server per gestire la connessione una volta che sia stato completato il *three way handshake*. La funzione restituisce un nuovo socket descriptor su cui si potrà operare per effettuare la comunicazione. Se non ci sono connessioni completate, il processo, viene messo in attesa. Il prototipo è:

```
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen)
```

Accetta una connessione sul socket specificato.

La funzione restituisce un numero di socket descriptor positivo in caso di successo e -1 in caso di errore, nel qual caso `errno` viene impostata ai seguenti valori:

`EBADF` l'argomento `sockfd` non è un file descriptor valido.

`ENOTSOCK` l'argomento `sockfd` non è un socket.

`EOPNOTSUPP` il socket è di un tipo che non supporta questa operazione.

`EAGAIN` o `EWOULDBLOCK` il socket è stato impostato come non bloccante (vedi sez. 12.2.1), e non ci sono connessioni in attesa di essere accettate.

`EPERM` le regole del firewall non consentono la connessione.

`ENOBUFS, ENOMEM` questo spesso significa che l'allocazione della memoria è limitata dai limiti sui buffer dei socket, non dalla memoria di sistema.

`EINTR` la funzione è stata interrotta da un segnale.

Inoltre possono essere restituiti gli errori di rete relativi al nuovo socket, diversi a secondo del protocollo, come: `EMFILE`, `EINVAL`, `ENOSR`, `ENOBUFS`, `EFAULT`, `EPERM`, `ECONNABORTED`, `ESOCKTNOSUPPORT`, `EPROTONOSUPPORT`, `ETIMEDOUT`, `ERESTARTSYS`.

La funzione estrae la prima connessione relativa al socket *sockfd* in attesa sulla coda delle connessioni complete, che associa ad un nuovo socket con le stesse caratteristiche di *sockfd*. Nella struttura *addr* e nella variabile *addrlen* vengono restituiti indirizzo e relativa lunghezza del client che si è connesso. I due argomenti addr e addrlen (passato per indirizzo per avere indietro il valore) sono usati per ottenere l'indirizzo del client da cui proviene la connessione.

Prima della chiamata addrlen deve essere inizializzato alle dimensioni della struttura il cui indirizzo è passato come argomento in addr; al ritono della funzione addrlen conterrà il numero di byte scritti dentro addr. Se questa informazione non interessa basterà inizializzare a NULL detti puntatori.

Se la funzione ha successo **restituisce il descrittore di un nuovo socket creato dal kernel** (*connected socket*) a cui viene associata la prima connessione completa che il client ha effettuato verso il socket *sockfd.* Quest'ultimo (*listening socket*) è quello creato all'inizio e messo in ascolto con *listen*, e non viene toccato dalla funzione. **Se non ci sono connessioni pendenti da accettare la funzione mette in attesa il processo finche non ne arriva uno.**

In generale c'è **sempre un solo socket in ascolto**, detto per questo *listening socket*, che resta per tutto il tempo nello stato *LISTEN*, mentre le connessioni **vengono gestite dai nuovi socket**, detti connected socket, ritornati da accept, che

si trovano automaticamente nello stato *ESTABLISHED*, e vengono utilizzati per lo scambio dei dati, che avviene su di essi, fino alla chiusura della connessione.

Funzioni *getsockname* e *getpeername*

Sono funzioni ausiliari che possono essere usate per recuperare alcune informazioni relative ai socket ed alle connessioni ad essi associati.

Getsocketname serve ad ottenere l'indirizzo locale associato ad un socket:

```
#include <sys/socket.h>
int getsockname(int sockfd, struct sockaddr *name, socklen_t *namelen)
```

Legge l'indirizzo locale di un socket.

La funzione restituisce 0 in caso di successo e -1 in caso di errore. I codici di errore restituiti in `errno` sono i seguenti:

`EBADF`	l'argomento `sockfd` non è un file descriptor valido.
`ENOTSOCK`	l'argomento `sockfd` non è un socket.
`ENOBUFS`	non ci sono risorse sufficienti nel sistema per eseguire l'operazione.
`EFAULT`	l'indirizzo `name` non è valido.

La funzione restituisce la struttura degli indirizzi del socket *sockfd* nella struttura indicata dal puntatore *name* la cui lunghezza è specificata tramite l'argomento *namlen*.

La funzione si usa tutte le volte che si vuole avere l'indirizzo locale di un socket: ad esempio può essere usata da un client (che usualmente non chiama bind) per ottenere numero IP e porta locale associati al socket restituito da una connect, o da un server che ha chiamato bind su un socket usando 0 come porta locale per ottenere il numero di porta effimera assegnato dal kernel.

Tutte le volte che si vuole avere l'indirizzo remoto di un socket si usa la funzione getpeername:

```
#include <sys/socket.h>
int getpeername(int sockfd, struct sockaddr * name, socklen_t * namelen)
```

Legge l'indirizzo remoto di un socket.

La funzione restituisce 0 in caso di successo e -1 in caso di errore. I codici di errore restituiti in `errno` sono i seguenti:

`EBADF`	l'argomento `sockfd` non è un file descriptor valido.
`ENOTSOCK`	l'argomento `sockfd` non è un socket.
`ENOTCONN`	il socket non è connesso.
`ENOBUFS`	non ci sono risorse sufficienti nel sistema per eseguire l'operazione.
`EFAULT`	l'argomento `name` punta al di fuori dello spazio di indirizzi del processo.

La funzione è identica a getsockname ed usa la stessa sintassi, ma restituisce l'indirizzo remoto del socket, cioè quello associato all'altro capo della connessione.

Funzione close

E’ la stessa funzione di Unix per chiudere i file. Quando si utilizza questa funzione sui socket “si marca” come chiuso e si ritorna immediatamente al processo.

Una volta chiamata il socket descriptor non è più utilizzabile dal processo e non può essere usato come argomento per una *read* o *write.*

Funzioni *FullRead FullWrite*

Con i socket è comune che funzioni come *read* o *write* possano restituire in input o scrivere in output un numero di byte inferiore di quello richiesto. Le due funzioni *Fullread* e *Fullwrite* eseguono lettura e scrittura tenendo conto di questa caratteristica e sono in grado di ritornare solo dopo aver letto o scritto esattamente il numero di byte specificato.

```
ssize_t FullRead(int fd, void *buf, size_t count)
{
    size_t nleft;
    ssize_t nread;

    nleft = count;
    while (nleft > 0) {             /* repeat until no left */
        if ( (nread = read(fd, buf, nleft)) < 0) {
            if (errno == EINTR) {   /* if interrupted by system call */
                continue;           /* repeat the loop */
            } else {
                return(nread);      /* otherwise exit */
            }
        } else if (nread == 0) {    /* EOF */
            break;                  /* break loop here */
        }
        nleft -= nread;             /* set left to read */
        buf +=nread;                /* set pointer */
    }
    return (nleft);
}
```

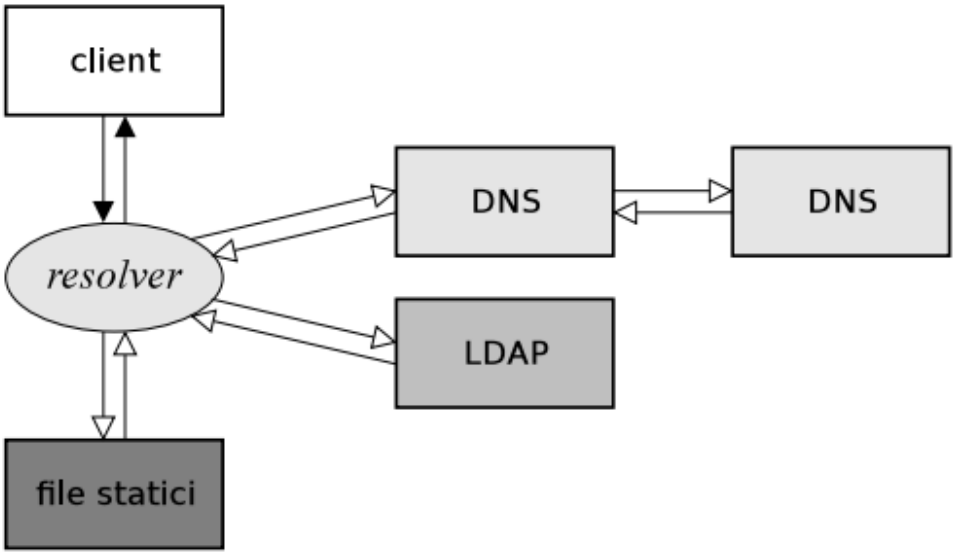
Come si può notare le due funzioni ripetono la lettura/scrittura in un ciclo fino all’esaurimento del numero di byte richiesti, in caso di errore viene controllato se questo è *EINTR* (Interruzione dovuta da una System call da un segnale), nel qual caso l’accesso viene ripetuto, altrimenti l’errore viene ritornato al programma chiamante, interrompendo il ciclo.

Nel caso della lettura, se il numero di byte letti è zero, significa che si è arrivato alla fine del file (nei socket significa che l’altro capo è stato chiuso, e quindi non sarà più possibile leggere niente) e pertanto si ritorna senza aver concluso la lettura di tutti i byte richiesti. Entrambe le funzioni **restituiscono 0 in caso di successo**, ed un valore negativo in caso di errore. ***FullRead* restituisce il numero di byte non letti in caso di EOF prematuro.**

Resolver DNS

La risoluzione dei nome è associata al servizio Domain Name Service che permette di identificare le macchine su internet con un nome a dominio piuttosto che un indirizzo IP.

Il linguaggio C fornisce un insieme di routine dette resolver che consentono di effettuare query al DNS delle applicazioni. In sostanza i programmi hanno a disposizione un insieme di funzioni di libreria (chiamate resolver) per eseguire le operazioni necessarie, che possono essere la lettura di informazioni mantenute nei relativi file statici presenti sulla macchina (file host), una interrogazione ad un DNS, o la richiesta altri server per i quali sia fornito il supporto, come LDAP.



Risoluzione dei nomi a dominio

La principale funzionalità del resolver resta quella di risolvere i nomi a dominio in indirizzi IP. La prima funzione è ***gethostbyname*** il cui scopo è ottenere l'indirizzo di una stazione noto il suo nome a dominio:

```
#include <netdb.h>
struct hostent *gethostbyname(const char *name)
```

Determina l'indirizzo associato al nome a dominio **name**.

La funzione restituisce in caso di successo il puntatore ad una struttura di tipo **hostent** contenente i dati associati al nome a dominio, o un puntatore nullo in caso di errore.

La funzione prende come argomento una stringa name contenente il nome a dominio che si vuole risolvere, in caso di successo i dati ad esso relativi vengono **memorizzati in una struttura *hostent***:

```
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 */
}
#define h_addr  h_addr_list[0]  /* for backward compatibility */
```

Quando un programma chiama *gethostbyname* e questa usa il DNS per effettuare la risoluzione del nome, è con i valori contenuti nei relativi record che vengono riempite le varie parti della struttura hostent.

Oltre ai normali nomi a dominio la funzione accetta come argomento *name* anche indirizzi numerici, in formato dotted decimal per IPv4 o con la notazione per IPv6. In questo caso, la funzione, non eseguirà nessuna interrogazione remota, ma si limiterà a copiare la stringa nel campo h_name ed a creare la corrispondente struttura in_addr da indirizzare con h_addr_list[0].

Con l'uso di *gethostbyname* normalmente si ottengono solo gli indirizzi IPv4, se si vogliono ottenere indirizzi IPv6 occorrerà prima impostare l'opzione RES_USE_INET6 nel campo *_res.options* e poi chiamare la *res_init* per modificare le impostazioni del resolver. Se vogliamo risparmiare tempo, si può utilizzare la funzione ***gethostbyname2*** che prende come secondo argomento *af* che indica la famiglia di indirizzi che dovrà essere utilizzata nei risultati restituiti dalla funzione. Per tutto il resto la funzione è identica a *gethostbyname*, ed identici sono i risultati.

```
#include <netdb.h>
#include <sys/socket.h>
struct hostent *gethostbyname2(const char *name, int af)
```

Determina l'indirizzo di tipo **af** associato al nome a dominio **name**.

La funzione restituisce in caso di successo il puntatore ad una struttura di tipo **hostent** contenente i dati associati al nome a dominio, o un puntatore nullo in caso di errore.

Le opzioni dei socket – Funzioni *setsockopt* e *getsockopt*

Benché dal punto di vista del loro uso come canali di trasmissione di dati **i socket siano trattati allo stesso modo dei file**, ed acceduti tramite i file descriptor, la normale interfaccia usata per **la gestione dei file non è sufficiente a poterne controllare tutte le caratteristiche**, che variano tra l'altro a seconda del loro tipo.

Le due funzioni *setsockopt* e *getsockopt* permettono di impostare le cosidette *socket options* (caratteristiche specifiche dei socket piuttosto dei file).

```
#include <sys/socket.h>
#include <sys/types.h>
int setsockopt(int sock, int level, int optname, const void *optval, socklen_t
    optlen)
```

Imposta le opzioni di un socket.

La funzione restituisce 0 in caso di successo e -1 in caso di errore, nel qual caso `errno` assumerà i valori:

`EBADF`	il file descriptor `sock` non è valido.
`EFAULT`	l'indirizzo `optval` non è valido.
`EINVAL`	il valore di `optlen` non è valido.
`ENOPROTOOPT`	l'opzione scelta non esiste per il livello indicato.
`ENOTSOCK`	il file descriptor `sock` non corrisponde ad un socket.

Il primo argomento della funzione, *sock*, indica il socket su cui si intende operare; per indicare l'opzione da impostare si devono usare i due argomenti successivi, ***level*** e ***optname***. I protocolli di rete sono strutturati su vari livelli, ed l'interfaccia dei socket può usarne più di uno. Si avranno allora funzionalità e caratteristiche diverse per ciascun protocollo usato da un socket, e quindi saranno anche diverse le opzioni che si potranno impostare per ciascun socket, a seconda del livello (trasporto, rete,ecc) su cui si vuole andare ad operare.

Il valore di *level* **seleziona allora il protocollo su cui vuole intervenire,** mentre *optname* **permette di scegliere su quale delle opzioni che sono definite per quel protocollo si vuole operare.** In sostanza la selezione di una specifica opzione viene fatta attraverso una coppia di valori level – optname e chiaramente la funzione avrà successo soltanto se il protocollo in questione prevede quella opzione ed è utilizzata dal socket.

Il quarto argomento, *optval* è un puntatore ad una zona di memoria che contiene i dati che specificano il valore dell'opzione che si vuole passare al socket, mentre l'ultimo argomento *optlen* è la dimensione in byte dei dati presenti all'indirizzo indicato da *optval.* La gran parte delle opzioni utilizzano optval per un valore intero, se poi l'opzione esprime una condizione logica, il valore è sempre un intero, ma si dovrà usare un valore non nullo per abilitarla ed un valore nullo per disabilitarla.

La seconda funzione usata per controllare la proprietà dei socket è ***getsockopt***, che serve a leggere i valori delle opzioni dei socket ed a farsi restituire i dati relativi al loro funzionamento:

```
#include <sys/socket.h>
#include <sys/types.h>
int getsockopt(int s, int level, int optname, void *optval, socklen_t *optlen)
```

Legge le opzioni di un socket.

La funzione restituisce 0 in caso di successo e -1 in caso di errore, nel qual caso `errno` assumerà i valori:

`EBADF`	il file descriptor `sock` non è valido.
`EFAULT`	l'indirizzo `optval` o quello di `optlen` non è valido.
`ENOPROTOOPT`	l'opzione scelta non esiste per il livello indicato.
`ENOTSOCK`	il file descriptor `sock` non corrisponde ad un socket.

I primi tre argomenti sono identici ed hanno lo stesso significato di quelli di *setsockopt*, anche se non è detto che tutte le opzioni siano definite per entrambe le funzioni. In questo caso *optval* viene usato per ricevere le informazioni ed indica l'indirizzo a cui andranno scritti i dati letti dal socket, infine *optlen* diventa un puntatore ad una variabile che viene usata come *value result argument* per indicare, prima della chiamata della funzione, la lunghezza del buffer allocato per *optval* e per ricevere indietro, dopo la chiamata della funzione, la dimensione effettiva dei dati scritti su di essi.

Nella "prima parte – riassunti delle slide" ci sono degli esempi con il SO_REUSEADDRESS

I/O Multiplexing

L'I/O multiplexing è una modalità di operazioni che consente di tenere sotto controllo più file descriptor in contemporanea, permettendo di bloccare un processo quando le operazioni volute non sono possibili, e di riprenderne l'esecuzione una volta che almeno una di quelle richieste sia effettuabile, in modo da poterla eseguire con la sicurezza di non stare bloccati.

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
int select(int ndfs, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct
    timeval *timeout)
```

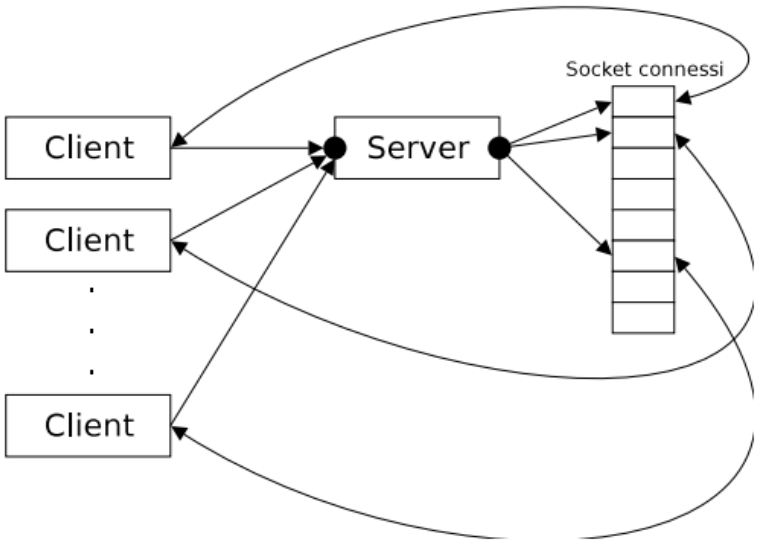
Attende che uno dei file descriptor degli insiemi specificati diventi attivo.

La funzione in caso di successo restituisce il numero di file descriptor (anche nullo) che sono attivi, e -1 in caso di errore, nel qual caso `errno` assumerà uno dei valori:

`EBADF`	si è specificato un file descriptor sbagliato in uno degli insiemi.
`EINTR`	la funzione è stata interrotta da un segnale.
`EINVAL`	si è specificato per `ndfs` un valore negativo o un valore non valido per `timeout`.

ed inoltre `ENOMEM`.

La funzione mette il processo in stato di sleep fintanto che almeno uno dei file descriptor degli insiemi specificati (*readfs, writefds, exceptfs*), non diventa attivo, per un tempo massimo specificato da *timeout.* Per specificare quali file descriptor si intende selezionare, la funzione, usa un particolare oggetto, il *file descriptor set* identificato dal tipo *fd_set* che serve ad identificare un insieme di file descriptor.



In questo caso avremo un solo processo che ad ogni nuova connessione da parte di un client sul socket in ascolto si limiterà a registrare l'entrata in uso di un nuovo file descriptor ed utilizzerà *select* per rilevare la presenza di dati in arrivo su tutti i filedescriptor attivi, operando direttamente su ciascuno di essi.

Restituisce -1 in caso di errore, 0 in caso di timeout e un intero >0 indicante il numero di descrittori pronti.
Il terzo parametro, che indica il tempo massimo che il processo deve restare bloccato nella system call, prevede tre casi:

1. NULL -> Nessun timeout
2. 0 -> Polling
3. Tempo specifico -> Tempo di timeout impostato

I parametri 2, 3 e 4 specificano gli insiemi di descrittori pronti. Sono di tipo *fd_set* *, array di bit in cui ogni bit identifica un descrittore:

- *readset* = Descrittore pronti in lettura
- *writeset* = Descrittore pronto in scrittura
- *exceptionset* = Dati urgenti

Un descrittore è pronto in lettura quando:

1. Il numero di byte nel buffer di ricezione del socket e' uguale o maggiore di un valore massimo chiamato low-watermark "LWM" per il buffer di ricezione (ci sono byte da leggere)
2. La connessione è stata chiusa.

3. Il socket è in *listen mode* e ci sono nuove connessioni (Il client esegue *connect*()).
4. Si è verificato un errore

Un descrittore è pronto in scrittura quando:
1. Il numero di byte liberi nel buffer di spedizione del socket è maggiore del LWM per il buffer di spedizione (c'è spazio per scrivere nuovi byte)
2. La connessione è stata chiusa
3. Un socket che ha utilizzato una *connect* non bloccante ha terminato la connessione oppure ha riscontrato un errore
4. In caso di errore.

Socket UDP

UDP è protocollo molto semplice che non supporta le connessioni e non è affidabile: esso si appoggia direttamente sopra IP. I dati vengono inviati in forma di pacchetti, e non è assicurata né la effettiva ricezione ne l'arrivo nell'ordine in cui vengono inviati. Il vantaggio del protocollo è la velocità, non è necessario trasmettere le informazioni di controllo ed il risultato è una trasmissione di dati più veloce e immediata.

Applicazioni che utilizzano questo protocollo: DHCP, DNS.

Questo significa che a differenza dei socket TCP, i socket UDP non supportano una comunicazione di tipo *stream* in cui si ha disposizione un flusso continuo di dati che può essere letto un po' alla volta, ma piuttosto una comunicazione di tipo *datagram*, in cui i dati arrivano in singoli blocchi che devono essere letti integralmente.

Non esiste il meccanismo di *three way handshake* né quello degli stati del protocollo, tutto quello che avviene nella comunicazione attraverso dei socket UDP è la trasmissione di un pacchetto da un client ad un server o viceversa.

La struttura generica di un server UDP prevede, una volta creato il socket, la chiamata a *bind per mettersi in ascolto dei dati.* Questa è l'unica parte comune con un server TCP: non essendovi il concetto di connessione, le funzioni *listen* e *accept* non sono mai utilizzate nel caso di server UDP. La ricezione dei dati dal client avviene attraverso la funzione ***recvfrom***, mentre un eventuale risposta sarà inviata con la funzione ***sendto***.

Da parte del client, invece, una volta creato il socket non sarà necessario connettersi con *connect,* ma si potrà effettuare direttamente una richiesta inviando un pacchetto con la funzione ***sendto*** e si potrà leggere una eventuale risposta con la funzione ***recvfrom***.

Anche se UDP è completamente diverso rispetto a TCP, resta identica la possibilità di gestire più canali di comunicazione fra le due macchine utilizzando le porte. In

questo caso il server dovrà usare comunque usare la funzione *bind* per scegliere la porta su cui ricevere i dati.

I socket UDP **non hanno uno stato**.

Le funzioni *sendto* e *recvfrom*

La necessità di usare queste funzioni è dovuta dal fatto che non esistendo con UDP il concetto di connessione, non si ha neanche a disposizione un *socket* connesso su cui sia possibile usare direttamente *read* e *write* avendo già stabilito quali sono sorgente e destinazione dei dati. *Sendto* e *recvfrom* sono due funzioni comunque utilizzabili in generale per la trasmissione di dati attraverso qualunque tipo di socket.

```
#include <sys/types.h>
#include <sys/socket.h>
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags, const struct
    sockaddr *to, socklen_t tolen)
    Trasmette un messaggio ad un altro socket.
```

La funzione restituisce il numero di caratteri inviati in caso di successo e -1 per un errore; nel qual caso `errno` viene impostata al rispettivo codice di errore:

`EAGAIN`	il socket è in modalità non bloccante, ma l'operazione richiede che la funzione si blocchi.
`ECONNRESET`	l'altro capo della comunicazione ha resettato la connessione.
`EDESTADDRREQ`	il socket non è di tipo connesso, e non si è specificato un indirizzo di destinazione.
`EISCONN`	il socket è già connesso, ma si è specificato un destinatario.
`EMSGSIZE`	il tipo di socket richiede l'invio dei dati in un blocco unico, ma la dimensione del messaggio lo rende impossibile.
`ENOBUFS`	la coda di uscita dell'interfaccia è già piena (di norma Linux non usa questo messaggio ma scarta silenziosamente i pacchetti).
`ENOTCONN`	il socket non è connesso e non si è specificata una destinazione.
`EOPNOTSUPP`	il valore di `flag` non è appropriato per il tipo di socket usato.
`EPIPE`	il capo locale della connessione è stato chiuso, si riceverà anche un segnale di `SIGPIPE`, a meno di non aver impostato `MSG_NOSIGNAL` in `flags`.

ed anche `EFAULT`, `EBADF`, `EINVAL`, `EINTR`, `ENOMEM`, `ENOTSOCK` più gli eventuali altri errori relativi ai protocolli utilizzati.

I primi tre argomenti sono identici a quelli della funzione write e specificano il socket sockfd a cui si fa riferimento, il buffer buf che contiene i dati da inviare e la relativa lunghezza *len.* Come per *write* **ritorna il numero di byte inviati**; nel caso di UDP però questo deve sempre corrispondere alla dimensione totale specificata da *len* in quanto i dati vengono sempre inviati in forma di pacchetto e non possono essere spezzati in invii successivi. Qualora non ci sia spazio nel buffer di uscita la funzione si blocca, se invece non è possibile inviare il messaggio all'interno di un unico pacchetto, lancia l'errore *EMSGSIZE*.

I due argomenti *to* e *tolen* servono a specificare la destinazione del messaggio da inviare, e indicano rispettivamente la struttura contenente l'indirizzo di quest'ultima e la sua dimensione; questi argomenti vanno specificati nella stessa forma in cui li si

sarebbero usati con *connect.* Nel nostro caso *to* dovrà puntare alla struttura contenente l'indirizzo IP e la porta di destinazione verso cui si vogliono inviare i dati.

Flags è un intero usato come maschera binaria che permette di impostare una serie di modalità di funzionamento della comunicazione attraverso il socket.

La seconda funzione utilizzata nella comunicazione fra socket UDP è ***recvfrom,*** che serve a ricevere i dati inviati da un altro socket:

```
#include <sys/types.h>
#include <sys/socket.h>
ssize_t recvfrom(int sockfd, const void *buf, size_t len, int flags, const struct
    sockaddr *from, socklen_t *fromlen)
    Riceve un messaggio ad un socket.
```

La funzione restituisce il numero di byte ricevuti in caso di successo e -1 in caso di errore; nel qual caso `errno` assumerà il valore:

`EAGAIN`	il socket è in modalità non bloccante, ma l'operazione richiede che la funzione si blocchi, oppure si è impostato un timeout in ricezione e questo è scaduto.
`ECONNREFUSED`	l'altro capo della comunicazione ha rifiutato la connessione (in genere perché il relativo servizio non è disponibile).
`ENOTCONN`	il socket è di tipo connesso, ma non si è eseguita la connessione.

ed anche `EFAULT`, `EBADF`, `EINVAL`, `EINTR`, `ENOMEM`, `ENOTSOCK` più gli eventuali altri errori relativi ai protocolli utilizzati.

Come per *sendto* i primi tre argomenti sono identici agli analoghi di read: dal socket vengono letti *len* byte che vengono salvati nel buffer *buf*. A seconda del tipo di socket i byte in eccesso che non sono stati letti possono rispettivamente andare persi o restare disponibili per una lettura successiva. Se non sono disponibili dati la funzione si blocca, a me no di non aver aperto il socket in modalità non bloccante (in questo caso si avrà il solito errore di *EAGAIN*.

I due argomenti *from* e *fromlen* sono utilizzati per ottenere l'indirizzo del mittente del pacchetto che è stato ricevuto, e devono essere opportunamente inializzati; il primo deve contenere il puntatore alla struttura che conterrà l'indirizzo e il secondo puntatore alla variabile con la dimensione di detta struttura. Se non si è interessati a questa informazione, entrambi gli argomenti devono essere inizializzati NULL.

Una differenza fondamentale di queste funzioni rispetto alle usuali *read* e *write* che abbiamo usato con i socket TCP è che in questo caso è perfettamente legale inviare con *sendto* **un pacchetto vuoto** (conterrà solo le intestazioni di IP e UDP) specificando un valore nullo per len. Allo stesso modo è possibile ricevere con *recvfrom* un valore di ritorno 0 byte, **senza che questo configurarsi come una chiusura della connessione** o come una **cessazione delle comunicazioni.**

Terza parte – Appunti generali

Applicazioni di rete sono applicazioni che operano in modo indipendente su macchine diverse scambiando informazioni tramite una rete, gestita da un sottosistema di comunicazione che gestisce lo scambio attraverso propri servizi.

Protocollo: insieme di regole e convenzioni utilizzate dai partecipanti alla comunicazione (ISO/OSI e TCP/IP)

Indirizzo IP: un numero che identifica univocamente un dispositivo connesso ad una rete informatica utilizzando lo standard IP (Internet Protocol)

Port: unsigned short (da 0 a 65k circa). Da 0 a 1023 riservate ai processi di root. Da 1024 si distinguono porte registrate e porte effimere:

- Le porte registrate sono usate per servizi o server privati.
- Le porte effimere sono assegnate dal kernel (range e politica di assegnazione dipendono dal kernel).

Endpoint = coppia di indirizzo IP e Port. Una comunicazione avviene tra due endpoint.

DNS = Domain Name Service -> associa un nome (simbolico) ad un indirizzo IP. È strutturato ad albero, la cui radice è il dominio principale, rappresentato da un punto e solitamente omesso. Ogni nodo dell'albero è un dominio, mentre le foglie rappresentano degli host di rete.
Il nome di un dominio è ottenuto unendo i nomi dei nodi che compongono un percorso foglia/radice (e separandoli mediante un .)

È un servizio distribuito (sono presenti più server, ciascuno con una zona di competenza). Una richiesta di traduzione è inviata al server di zona: se l'informazione non è disponibile invia la richiesta ad un altro server. Ogni zona organizza le proprie informazioni di competenza in record di risorsa (contengono varie informazioni).

La risoluzione di un dominio può avvenire: iterativamente, ricorsivamente.
Fornisce anche servizi per la risoluzione inversa: a partire da un indirizzo IP restituisce il nome associato.

FTP = permette il trasferimento di file di testo e binari tra computer su una rete. Utilizza due porte: una per il controllo e una per i trasferimenti dati. Quella di controllo è sempre aperta, l'altra viene aperta in fase di trasferimento (effettivo). Ad ogni comando, il server risponde con un codice numerico memorizzato in 3 caratteri "xyz" in cui, da sinistra verso destra, ciascun numero fornisce informazioni sempre più dettagliate (x dà un'informazione, y aggiunge dettaglio a x e z aggiunge dettaglio a y).

Quando due endpoint vogliono comunicare si devono identificare univocamente. Ciò avviene tramite due livelli di indirizzamento:

1) si identifica l'host su cui è attivo il processo (ogni host ha un indirizzo IP)
2) si determina il processo con cui comunicare (ad ogni processo è associato un numero di porta con cui comunicare)

I datagrammi UDP sono formati da un header che contiene una serie di informazioni, come la porta di sorgente e destinazione o un campo per il check.

I segmenti TCP contengono molte più informazioni, come il numero di sequenza del pacchetto o un valore di acknowledgement.

Sia i datagrammi UDP, sia i segmenti TCP, sono "incapsulati" in un header IP,

Three way handshake

1. Il Client, che vuole comunicare con il Server, gli invia un segmento TCP in cui #seq=N e #ack=0 e il flag SYN=1 e ACK=0

2. Server genera un numero iniziale di sequenza M e risponde al Client con un segment TCP i cui flag SYN=ACK=1, mentre il #seq=M e #ack = N+1

3. Client invia a Server un segmento in cui i flag SYN=0 e ACK=1 e #seq=N+1 e #ack=M+1

Comandi UNIX:

ifconfig mostra/modifica la configurazione delle interfacce di rete disponibili sul dispositivo.

route modifica la tabella di routing (è possibile impostare così l'indirizzo di gateway).

whois fornisce informazioni sul dominio specificato.

nslookup in Unix/Windows permette di interrogare il DNS.

dig/host preferibili su Linux rispetto a nslookup.

ICMP protocollo di livello network per segnalare errori all'interno della rete. Esistono diversi tipi di ICMP un messaggio ICMP viene incapsulato in un datagram IP (campo dati). Comandi come **"ping" e "traceroute" usano messaggi ICMP.**

I servizi di rete sono inizializzati durante il boot del sistema. Tramite processi daemon, tali servizi restano in funzione.
La gestione dei daemon può essere autonoma (standalone) o affidata ad un processo supervisore.

- **Standalone** -> programmi avviati al boot del sistema, sempre in esecuzione, ascoltano su una determinata porta, controllano gli accessi al servizio di rete (NFS)

- **Supervisore**-> sono avviati dal supervisore in caso di richieste (FTP)

Quarta parte – Teoria in Java

Java è un linguaggio **orientato alla rete**.

La classe *java.net.**InetAdress*** astrae gli indirizzi di rete. Ha tre scopi principali:

- Incapsula un indirizzo
- Esegue il name lookup (Nome host -> indirizzo IP)
- Esegue il reverse lookup (Indirizzo IP -> nome host)

Quando si istanzia un oggetto di tipo InetAdress, l'oggetto diventa immutabile (Si crea con l'indirizzo necessario e si "butta" quando non serve più).

Sebbene a livello di Sistema Operativo esista un unico tipo di socket, in Java esistono oggetti diversi per socket associate a protocolli diversi. In particolare per le socket TCP/IP vi sono due oggetti:

- **Socket** Per le socket client
- **ServerSocket** Per le socket server

I metodi delle Socket Java sono realizzati in modo da astrarre e virtualizzare il più possibile le corrispondenti funzioni C. Ad esempio, la funzione connect() esegue da sola, se necessario, la DNS query per risolvere il FQDN in IP andress.

In Java le funzioni read/write sono accessibili mediante oggetti **InputStream** o **OutputStream**. InputStream e OutputStream trattano byte. Affinche possono trattare char devono essere manipolati con **InputStreamReader** e **OutputStreamReader** (anche stringhe). Con la classe **BufferedReader**, applicata a una InputStreamReader consente la lettura di linee intere della socket.

DIFFERENZA CON APPLICAZIONE CLIENT & SERVER

Client

1. Creazione del socket;
2. Richiesta di connessione;
3. Lettura – scrittura dei dati;
4. Chiusura

Server

1. Creazione del socket;
2. Bind ad una porta;
3. Listen, predisposizione a ricevere sulla porta;
4. Accept, blocca il server in attesa di una connessione;
5. Lettura – scrittura dei dati;
6. Chiusura

SKU – STAT CHIN E UAI -w