Reti Programmazione

a. a. 2003/04

Introduzione all'API socket di Berkeley

Prof. Vincenzo Auletta
auletta@dia.unisa.it
http://www.dia.unisa.it/professori/auletta/

Università degli studi di Salerno
Laurea e Diploma in Informatica

Sviluppo di Applicazioni su Rete

- Le applicazioni che operano su una rete consistono di varie componenti che operano indipendentemente e che si scambiano informazioni
  - ogni componente è un processo che gira localmente su un host
  - la comunicazione è ottenuta utilizzando i servizi offerti dal sottosistema di comunicazione
- La cooperazione tra le componenti può essere implementata secondo vari modelli
  - il modello più diffuso è il client-server

Modello Client-Server

- Un'applicazione opera localmente ma può richiedere informazioni anche ad altre applicazioni
  - invia la richiesta, attende il risultato e poi riprende il suo normale svolgimento
- Questo modello di interazione può essere considerato a vari livelli
  - a livello di programma (una funzione invoca un'altra funzione)
  - a livello di processi locali (comunicazione interprocesso)
  - a livello di processi remoti (comunicazione tra processi in esecuzione su host distinti)
- L'applicazione che richiede l'informazione si chiama client
- L'applicazione che fornisce la risposta si chiama server
  - in genere il server ha la sola funzione di rispondere alle richieste dei client

Interazione tra Processi

- Le applicazioni sono sviluppate come processi utente
- La comunicazione tra le applicazioni è basata sui servizi forniti dal sottosistema di comunicazione
  - basata sull'implementazione di una suite di protocolli di comunicazione integrata nel sistema operativo
- Per poter comunicare due applicazioni devono interagire con i rispettivi sistemi operativi
  - ogni applicazione deve chiedere al suo S.O. di inviare o ricevere dati tramite la rete
  - l'interazione con il S.O. è fatta tramite l'invocazione di chiamate di sistema (system call)

Application Programming Interface

- L'interazione con il sistema operativo non è standard
  - un'applicazione che funziona su una certa piattaforma non è portabile su altre piattaforme
- Si usa un'API (Application Programming Interface) per standardizzare l'interazione con il sistema operativo
  - insieme di funzioni che possono essere invocate per effettuare chiamate di sistema
    - le interfacce delle funzioni sono indipendenti dalla piattaforma
  - La più diffusa è Berkeley Sockets

Socket

- Letteralmente significa "presa"
  - implementa l'astrazione di un canale di comunicazione fra due host connessi da una rete
- Sviluppato per sistemi UNIX (BSD e System V)
  - Utilizza una estensione del paradigma di I/O di questi sistemi
    - open-read-write-close
    - accesso alle connessioni tramite descrittori
    - bisogna distinguere le open e close tra attive e passive
- I socket sono divenuti uno standard di riferimento per tutta la programmazione su reti
  - Utilizzate da Windows (winsocket)
  - Linux, SUN Solaris, ecc.

Operazioni su un Socket

- Il ciclo di vita di un socket è simile a quello di un file
  - Apertura
  - Collegamento ad un endpoint (specifica del socket)
  - Lettura e/o scrittura
  - Chiusura

Interazione tra Applicazione e S.O.

- L'applicazione chiede al sistema operativo di utilizzare i servizi di rete
- Il sistema operativo crea un socket e lo restituisce all'applicazione
  - identificato da un socket descriptor
- L'applicazione utilizza il socket
  - Simile all'uso di un file
- L'applicazione chiude il socket e lo restituisce al sistema operativo

Progettazione di un Client UDP

- Creazione di un endpoint
  - Richiesta al sistema operativo
- Invio e ricezione di datagram
  - si può specificare da quale destinatario si accettano datagram
- Chiusura dell'endpoint

Progettazione di un Server UDP

- Creazione di un endpoint
  - Richiesta al sitema operativo
- Collegamento dell'endpoint ad una porta
  - open passiva in attesa di ricevere datagram
- Ricezione ed invio di datagram
- Chiusura dell'endpoint

Progettazione di un Client TCP

- Creazione di un endpoint
  - Richiesta al sistema operativo
- Creazione della connessione
  - Implementa open di TCP (3-way handshake)
- Lettura e scrittura sulla connessione
  - Analogo a operazioni su file
- Chiusura della connessione
  - Implementa close di TCP (4-way handshake)

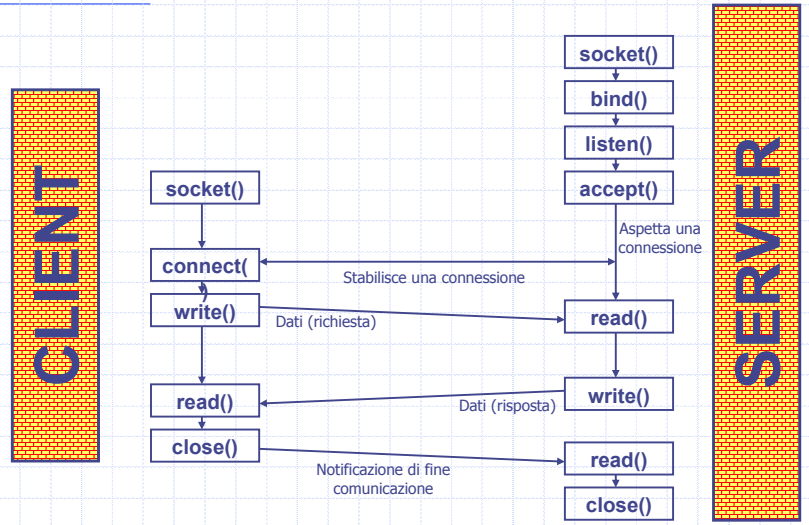
Progettazione di un Server TCP

- Creazione di un endpoint
  - Richiesta al sitema operativo
- Collegamento dell'endpoint ad una porta
- Ascolto sulla porta
  - sleeping
- Accettazione della richiesta di un client
- Letture e scritture sulla connessione
- Chiusura della connessione

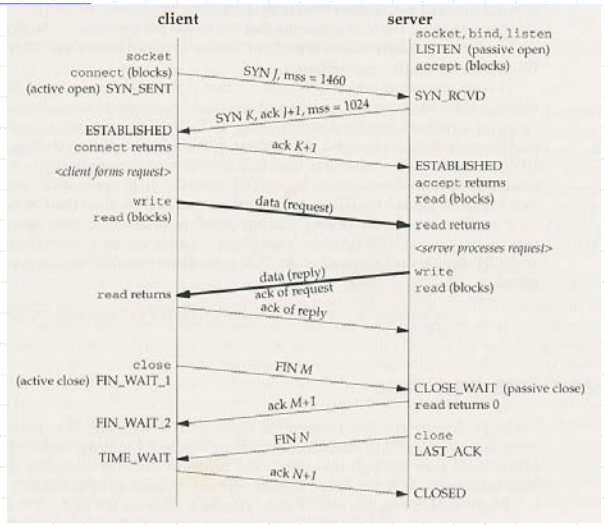
Standard POSIX

- Gli standard POSIX definiscono il kernel dei sistemi UNIX
- Lo standard POSIX.1g ha definito il supporto di UNIX per il networking
  - L'API socket è basata su questo standard
- Linux ha una propria implementazione del networking
  - Non necessariamente corrispondente a POSIX.1g
- Controllare sul manuale in linea i dettagli relativi alle implementazioni delle varie funzioni dell'API

Struttura di un'Applicazione TCP



Scambio di Pacchetti su una Connessione TCP



Gestione degli Errori

- In caso di errore le funzioni dell'API socket operano nella seguente maniera
  - restituiscono un valore negativo (-1)
  - assegnano alla variabile globale ERRNO un valore intero
    - ogni valore identifica un tipo di errore ed i significati sono specificati nel file sys/errno.h
- Il valore contenuto in ERRNO si riferisce all'ultima chiamata a sistema effettuata
- Dopo aver invocato una funzione dell'API si deve
  - verificare se il risultato è negativo
  - in caso di errore leggere il valore di ERRNO

Funzione socket

```
#include <sys/socket.h>
int socket(int family, int type, int protocol);
```

- Crea un endpoint
- Restituisce
  - -1 se la creazione non è riuscita
  - il descrittore del socket se la creazione è riuscita
- Un descrittore di socket è simile ad un descrittore di file
  - Sono presi dallo stesso insieme
    - Se un intero è usato come descrittore di file non può essere usato come descrittore di socket e viceversa
  - Socket e file sono visti più o meno allo stesso modo
    - read, write, close sono le stesse funzioni dei file

Parametri della Funzione socket

- family specifica la famiglia di protocolli da usare:
  - AF_INET IPv4
  - AF_INET6 IPv6
  - AF_LOCAL prot. locale (client e server sullo stesso host)
  - AF_ROUTE Socket per routing
  - altri …
- type identifica il tipo di socket
  - SOCK_STREAM per uno stream di dati (TCP)
  - SOCK_DGRAM per datagrammi (UDP)
  - SOCK_RAW per applicazioni dirette su IP
- protocol
  - 0, tranne che per SOCK_RAW

Funzione connect

```
#include <sys/types.h>
#include <sys/socket.h>
int connect(int sd, const SA *servaddr, socklen_t addrlen);
```

- Permette ad un client di aprire una connessione con il server
  - Il SO sceglie una porta effimera (ed eventualmente l'indirizzo IP) ed effettua una open attiva
  - la funzione termina solo dopo che la connessione è stata creata
- Restituisce
  - -1 in caso di errore
  - 0 se la connessione è stata creata

Parametri della Funzione connect

- sd è il socket descriptor
- servaddr è un puntatore all'indirizzo dell'endpoint cui ci si vuole collegare
  - indirizzo IP + numero di porta
  - puntatore di tipo sockaddr
- addrlen è la lunghezza in byte di servaddr
- In caso di errore restituisce
  - ETIMEDOUT è scaduto il time out del SYN
  - ECONNREFUSED il server ha rifiutato il SYN
  - EHOSTUNREACH errore di indirizzamento

Funzione bind

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sd, const SA*myaddr, socklen_t addrlen);
```

- Permette di assegnare uno specifico indirizzo al socket
  - se non si esegue la bind il S.O. assegnerà al socket una porta effimera ed uno degli indirizzi IP dell'host
    - dipende dall'interfaccia utilizzata
  - in genere eseguito solo dal server per usare una porta prefissata

Parametri della Funzione bind

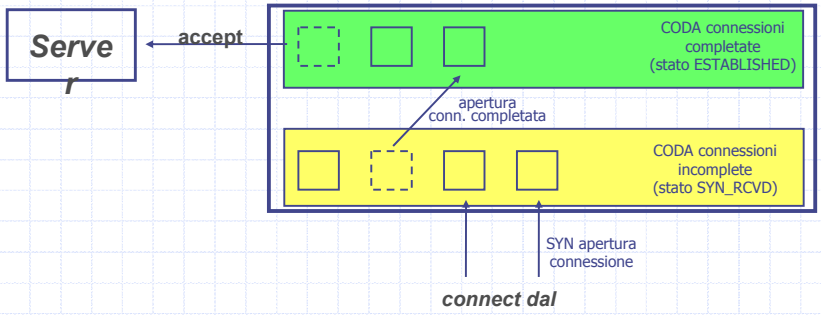
- L'oggetto puntato da myaddr **può** specificare
  - l'indirizzo IP
  - il numero di porta
- Un campo non specificato è messo a 0
  - Se la porta è 0 ne viene scelta una effimera
  - Se l'indirizzo IP è INADDR_ANY (0) il server accetterà richieste su ogni interfaccia
    - quando riceve un segmento SYN utilizza come indirizzo IP quello specificato nel campo destinazione del segmento
- La funzione restituisce un errore se l'indirizzo non è utilizzabile
  - EADDRINUSE

Funzione listen

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

- Utilizzata per rendere un socket passivo
  - il S.O. accetta tutte le richieste inviate al socket ma non prende mai l'iniziativa
  - utilizzata solo dai server TCP
  - Nel diagramma a stati di TCP fa muovere da CLOSED a LISTEN
- Specifica quante connessioni possono essere accettate e messe in attesa di essere servite
  - le connessioni sono accettate o rifiutate dal S.O. senza interrogare il server

Backlog



- Nel backlog ci sono sia le richieste di connessione in corso di accettazione che quelle accettate ma non ancora passate al server
- La somma degli elementi in entrambe le code non può superare il *backlog*

Funzione accept

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int sd, const SA*cliaddr, socklen_t* addrlen);
```

- Permette ad un server di prendere la prima connessione completata dal backlog
  - Se il backlog è vuoto il server rimane bloccato sulla chiamata a funzione fino a quando non viene accettata una connessione
- Restituisce
  - -1 in caso di errore
  - un nuovo descrittore di socket assegnato automaticamente dal S.O. e l'indirizzo del client
    - la porta del nuovo descrittore è effimera

Socket di Ascolto e Socket Connesso

- Il server utilizza due socket diversi per ogni connessione con un client
  - il socket di ascolto (listening socket) è quello creato dalla funzione socket
    - utilizzato per tutta la vita del processo
    - in genere usato solo per accettare richieste di connessione
  - il socket connesso (connected socket) è quello creato dalla funzione accept
    - usato solo per la connessione con un certo client
    - usato per lo scambio dei dati con il client
- I due socket identificano due connessioni distinte
  - sono connessi allo stesso endpoint del client ma usano porte differenti

Parametri Valore-Risultato

- cliaddr e addrlen sono argomenti valore-risultato
  - sono variabili passate per riferimento usate sia per passare argomenti alla funzione che per ricevere dei risultati
- cliaddr contiene
  - prima l'indirizzo di sd (indirizzo IP + porta)
  - dopo l'indirizzo del client (indirizzo IP + porta)
- addrlen contiene
  - prima la lunghezza dell'indirizzo di sd
  - dopo la lunghezza dell'indirizzo del client

Funzione close

```
#include <unistd.h>
int close(int sd);
```

- marca il descrittore come chiuso
  - il processo non può più utilizzare il descrittore ma la connessione non viene chiusa subito
- Restituisce
  - -1 in caso di errore
  - 0 se OK
- più processi possono condividere un descrittore
  - un contatore mantiene il numero di processi associati al descrittore
  - la procedura di close della connessione viene avviata solo quando il contatore arriva a 0

Funzioni di Gestione degli Errori

- Negli esempi controlleremo il valore restituito da ogni chiamata ad una funzione dell'API
  - se il risultato è < 0 sarà invocata una funzione per gestire l'errore
- Le 5 funzioni di gestione operano come segue
  - prendono una stringa in input
  - costruiscono una stringa concatenando il messaggio associato al valore di ERRNO con la stringa input e la stampano
  - differiscono su dove stampano la stringa e cosa fanno dopo
- Le definizioni delle funzioni sono in error.c

Funzioni di error.c

- void err_sys(const char* frmt, …);
  - errore fatale relativo ad una chiamata di sistema
- void err_quit(const char* frmt, …);
  - errore fatale non relativo ad una chiamata di sistema
- void err_dump(const char* frmt, …);
  - errore fatale relativo ad una chiamata di sistema che richiede un dump del core
- void err_ret(const char* frmt, …);
  - errore non fatale relativo ad una chiamata di sistema
- void err_msg(const char* frmt, …);
  - errore non fatale non relativo ad una chiamata di sistema

Struttura degli Indirizzi dei Socket

- Gli indirizzi sono implementati come struct
  - Sempre passate per riferimento
- Ogni famiglia di indirizzi definisce la propria struct sockaddr_xxx
  - sockaddr_in per TCP/IP
  - le funzioni accettano indirizzi di tutte le famiglie
- Le funzioni hanno un parametro di tipo sockaddr
  - una struct generica non riferita a nessuna famiglia di indirizzi specifica
    - Quando è stata progettata l'API non esistevano i puntatori void*
  - nella chiamata serve un cast

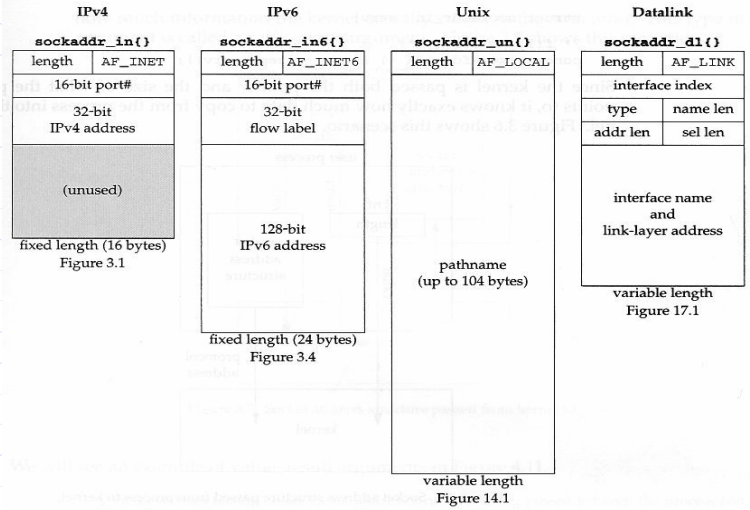
Sockaddr_in

```
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];
};
```

- sin_zero serve per far si che la grandezza della struttura sia almeno 16 byte
- sin_len non è richiesto dallo standard Posix ma è usato per gestire strutture di diverse lugnhezze

Lunghezze delle Strutture Sockaddr_xxx



Tipi di Dati di sockaddr

- Lo standard POSIX1.g prevede i seguenti tipi di dati
  - int8_t, int16_t, int32_t (interi con segno)
  - uint8_t, uint16_t, uint32_t (interi senza segno)
  - sa_family_t (famiglia di indirizzi)
  - socklen_t (in genere uint32_t)
  - in_addr_t (in genere uint32_t)
  - in_port_t (in genere uint16_t)
- Le definizioni sono contenute in sys/types.h, sys/socket.h e netinet/in.h

Rappresentazione degli Interi

- Tutti i protocolli di TCP/IP assumono che gli interi siano trasmessi in network byte order
  - prima il byte più significativo (big endian)
  - Prima di scrivere un intero sulla connessione bisogna convertirlo in network byte order
  - Prima di leggere un intero ricevuto dalla connessione bisogna convertirlo in host byte order
- Funzioni di conversione (in netinet/in.h)
  - **htons** (16 bit), **htonl** (32 bit)
  - **ntohs** (16 bit), **ntohl** (32 bit)
- Le informazioni in sockaddr devono essere in network byte order

Funzioni di Manipolazione dei Byte – 1

- Molte operazioni sui socket richiedono di operare su sequenze di byte
  - Nessuna interpretazione dei dati
  - Non assume che sono stringhe C
- strings.h contiene

```
void bzero(void* dest, size_t  n);
void bcopy(const void* src, void* dest, size_t  n);
int bcmp(const void* ptr1, const void* ptr2, size_t  n);
```

- obsolete
  - per comodità si usa solo bzero() per inizializzare a 0 delle strutture

Funzioni di Manipolazione dei Byte – 2

- string.h contiene funzioni analoghe ANSI C

```
void* memset(void* dest, int c, size_t  n);
void* memcpy(const void* dest, void* src, size_t  n);
int memcmp(const void* ptr1, const void* ptr2, size_t  n);
```


Funzioni di Manipolazione degli Indirizzi

- usate per convertire indirizzi dal formato testuale (ASCII) in interi in network byte order
  - definite in arpa/inet.h

```
int inet_aton(const char* strptr, struct in_addr* addptr);
char* inet_ntoa(const char* strptr);
      convertono solo  indirizzi IP4
in_addr_t inet_addr(struct in_addr* addptr);
      restituisce l'indirizzo contenuto in addptr
int inet_pton(int family, const char* strptr, voidr* addptr);
const  char*  inet_ntop(int  family,  const  void*  addptr,  char*
addptr,  size_t  n);
      convertono ogni tipo di indirizzi
```


Esempio: Applicazione Daytime

- Applicazione daytime
  - Il client richiede l'ora e la data
  - Il server recupera l'informazione dal S.O. e la invia al client
  - Il client stampa la risposta del server su stdout
- Assunzioni
  - il server invia la risposta in un unico blocco come una stringa
  - il client legge una sola stringa ed esce
    - nessuna interazione
  - client e server usano una connessione TCP

Client Daytime – 1

```
#include"../lib/basic.h"
int main(int argc, char **argv)  {
  int                    sd, n;
  char                   recvline[MAXLINE + 1];
  struct sockaddr_in     servaddr;

  if ( argc != 2 )                                          (1)
    err_quit("utilizzo: daytime_tcp_client <IPaddress>");
  if ( (sd = socket(AF_INET, SOCK_STREAM, 0)) < 0 )         (2)
    err_sys("errore in socket");
  bzero(&servaddr, sizeof(servaddr));                       (3)
  servaddr.sin_family = AF_INET;
  servaddr.sin_port   = htons(SERV_PORT);
  if ( inet_pton(AF_INET, argv[1], &servaddr.sin_addr) <= 0
)
        err_quit("errore in inet_pton per %s", argv[1]);
```

1. controllo del numero di argomenti
2. creazione del socket
3. inizializzazione della struct sockaddr_in
   - prima si azzera
   - poi si inizializza ogni campo

Client Daytime – 2

```
  if ( connect(sd, (struct sockaddr *) &servaddr,
sizeof(servaddr)) < 0 )                                  (4)
      err_sys("errore in connect");
  while ( (n = read(sd, recvline, MAXLINE)) > 0 ) {      (5)
     recvline[n] = 0;                                    (6)
     if (fputs(recvline, stdout) == EOF)                 (7)
        err_sys("errore in fputs");
  }
  if (n < 0)                                             (8)
     err_sys("errore in read");
  exit(0);                                               (9)
}
```

4. connessione al server
5. legge fino a quando trova EOF o errore
6. inserisce il carattere terminatore
7. stampa il risultato su stdoout
8. se la read ha dato errore interrompe
9. esce

Server Daytime – 1

```
#include"../lib/basic.h"
int main(int argc, char **argv)  {
  int                    listend, connd;
  struct sockaddr_in     servaddr;
  char                   buff[MAXLINE];
  time_t                 ticks;

  if( (listend = socket(AF_INET, SOCK_STREAM, 0)) < 0) (1)
        err_sys("errore in socket");
  bzero(&servaddr, sizeof(servaddr));
  servaddr.sin_family      = AF_INET;
  servaddr.sin_addr.s_addr = htonl(INADDR_ANY);        (2)
  servaddr.sin_port        = htons(SERV_PORT);
  if( (bind(listend, (SA *) &servaddr, sizeof(servaddr))) < 0)
                                                       (3)
        err_sys("errore in bind");
  if( listen(listend, LISTENQ) < 0 )                   (4)
        err_sys("errore in listen");
```

1. creazione del socket
2. inizializzazione della struct sockaddr_in
   - INADDR_ANY
3. associa il socket all'indirizzo di servaddr
4. mette il socket in ascolto

Server Daytime – 2

```
  for ( ; ; ) {                                          (5)
     if( (connd = accept(listend, (SA *) NULL, NULL)) <
0)                                                       (6)
        err_sys("errore in accept");
     ticks = time(NULL);                                 (7)
     snprintf(buff, sizeof(buff), "%.24s\r\n",
ctime(&ticks));                                          (8)
     write(connd, buff, strlen(buff));                   (9)
     close(connd);                                       (10)
  }
}
```

5. server sempre in ascolto
6. accetta una connessione
   - ottiene il socket connesso
7. legge l'orario
8. formatta la data
9. scrive il messaggio sul socket
10. chiude la connessione sul socket connesso

Funzioni di Input/Output

- Per leggere e scrivere su un socket si utilizzano le funzioni read e write

```
int read(int sd, char* recvline, size_t size);
int write(int sd, const void* sendline, size_t size);
```

- Queste chiamate di sistema su un socket TCP si comportano diversamente che su un file
  - la funzione read può leggere meno dati di quanto richiesto
    - non ci sono dati disponibile nel buffer del socket
  - la funzione write può scrivere meno dati di quanto richiesto
    - non c'è spazio disponibile nel buffer del socket
- le funzioni devono essere chiamate in un ciclo

Funzione writen

```
ssize_t writen(int fd, const void* vptr, size_t n) {
  size_t nleft;
  ssize_t nwritten;
  const char* ptr;
  ptr = vptr;
  nleft = n;
  while( nleft > 0 ) {                                    (1)
     if( (nwritten = write(fd, ptr, nleft)) <= 0 ) { (2)
           if( errno == EINTR )  nwritten = 0;     (3)
           else return(-1);
     }
     nleft -= nwritten;
     ptr += nwritten;
  }
  return(n);
};
```

1. nleft caratteri da scrivere
   - itera finché nleft < 0
2. ad ogni iterazione chiama write e scrive nwritten caratteri
   - quelli disponibili
3. controlla il risultato di write
   - se segnala errore (EINTR) non scrive nulla ma continua
   - se si verifica un altro errore esce

Funzione readn

```
ssize_t readn(int fd, void* vptr, size_t n) {
  size_t nleft;  ssize_t nread;
  char* ptr;
  ptr = vptr;
  nleft = n;
  while( nleft > 0 ) {                                    (1)
     if( (nread = read(fd, ptr, nleft)) <= 0 ) {     (2)
           if( errno == EINTR )  nread = 0;          (3)
           else return(-1);
     }
     else if( nread == 0 )  break;
     nleft -= nread;
     ptr += nread;
  }
  return(n - nleft);
};
```

1. nleft caratteri da leggere
   - itera finché nleft < 0
2. ad ogni iterazione chiama read e legge nread caratteri
   - quelli disponibili
3. controlla il risultato di read
   - se segnala errore (EINTR) non legge nulla ma continua
   - se si verifica un altro errore esce
   - se legge EOF termina

Funzione readline

- la funzione readn itera finché non legge n valori o un EOF
  - l'applicazione non sa quanti caratteri deve leggere
- la funzione readline legge una sequenza di caratteri terminata da un "newline"
  - la funzione resta bloccata se manca il newline
  - ogni applicazione deve inserire un newline alla fine di ogni messaggio scritto sul socket per consentire al suo pari di uscire dal ciclo di lettura
- la funzione readline utilizza una versione modificata di read
  - my_read legge tutti i dati disponibili e li passa a readline uno alla volta

Funzione isfdtype

- la funzione isfdtype verifica di che tipo è un descrittore
  - riconosce se è un descrittore di socket o di file
  - prende in input una costante che identifica un tipo di descrittore e risponde true o false

Organizzazione degli Esempi

- Tutti i codici degli esempi si trovano nella directory PROGRETI/src
  - una directory per ogni esempio
  - una directory lib
  - una directory bin
- le directory degli esempi contengono
  - README
  - l'implementazione di client e server
  - Makefile
- lib contiene funzioni di utilizzo comune
  - per ora funzioni di input/output, funzioni di gestione degli errori, funzione di gestione dei segnali
- bin contiene gli eseguibili dei file contenuti in lib

File di libreria

- basic.h contiene
  - include dei principali header file di sistema necessari per programmi su rete
    - sys/socket.h, sys/errno.h, netinet/in.h, ecc.
    - signal.h, fcntl.h, stdio.h, stdlib.h, string.h, ecc.
  - definizioni di costanti e macro di uso comune
    - LISTENQ, MAXLINE, BUFFSIZE, ecc.
- error.c contiene le definizioni delle funzioni di gestione degli errori
- my_io.c contiene le definizioni delle funzioni di supporto
  - readn, writen, readline, my_read

Utilizzo degli Esempi

- gli esempi devono essere compilati
  - client e server compilati separatamente e linkati alle librerie
  - **gcc** –**g** –**o** client client.c
  - **gcc** –**g** –**L**mylibpath –**l**mylib –**I**myinclpath –**D**DEBUG -**W**error -**O** –**o** server server.c
- è possibile gestire la compilazione dell'intero progetto con il Makefile (gmake)
  - ricompila solo i file che sono stati modificati
  - controlla tutte le dipendenze tra i file del progetto
  - esegue anche altre operazioni oltre la compilazione

Esempio di Makefile

```
PROGS =  client, server
CFLAGS = -g
CC = gcc
CLEANFILES = client.o server.o
BIN = ../bin
all: ${PROGS}
client.o: client.c
        ${CC} ${CFLAGS} -c client.c
server.o: server.c
        ${CC} ${CFLAGS} -c server.c
client:  client.o ${BIN}/error.o
        ${CC} -o $@ $^
server:  server.o ${BIN}/error.o
        ${CC} -o $@ $^
clean:
        rm -f ${CLEANFILES}
```

- definisce variabili
  - si leggono con ${}
- definisce i target
  - ogni target può essere specificato come argomento di gmake
- per ogni target specifica una regola
  - file da cui dipende
  - operazioni da eseguire