Socket

Protocol Architecture

Protocollo: insieme di regole che gestiscono lo scambio dati tra due entità (thread o processi); è necessario per definire la comunicazione tra due entità in modo che non ci siano "fraintendimenti" e che entrambe le entità comprendano ciò che dice l'altra + entrambe le entità devono essere pronte per la comunicazione (es. un processo non manda un file a un altro se sa che non è in esecuzione)

Comunicazione di rete → 2 parti:

1. *Comunicazione tra computer,* in cui definisco tutti gli aspetti della comunicazione (con chi interagisco, come rappresentare il dato …)
2. *Rete* (come è strutturata, che dispositivi ci sono nel mezzo)

➔ entrambe le parti necessitano di protocolli (es. IP, TCP, …)

Elementi chiave di un protocollo:

- Sintassi → formato dei dati e livello dei segnali
- Semantica → controllo delle informazioni e gestione errori
- Tempo → avere stessa velocità e sincronizzazione

Protocol Architecture → insieme di moduli che implementano le funzioni di comunicazione; la divisione in moduli è necessaria per non dover implementare tutto in un unico modulo

es. trasmissione di un file:

- livello di applicazione → modulo di file transfer con tutta la gestione logica per la trasmissione
- livello di trasporto → modulo di servizio di comunicazione che assicura che i due computer siano attivi e pronti per il trasferimento, e per tener traccia dei dati che stanno per essere scambiati per assicurare la consegna
- livello di rete → modulo di accesso alla rete separato per gestire la parte logica della trasmissione in rete

➔ a livello più alto sembra che comunichino direttamente i due processi, ma in realtà si appoggiano su livelli di comunicazione e di rete

TCP/IP → protocollo sperimentale inventato mentre si definiva l'architettura OSI; nato per gestire i pacchetti della rete ARPNET; ingloba insieme i due layer

Livello fisico → copre la parte fisica; specifica le caratteristiche del mezzo di comunicazione, la natura dei segnali e la velocità di trasmissione dei dati

Livello di accesso di rete → come scambiare le informazioni tra vari dispositivi; dipende dal tipo di rete utilizzata

Livello di rete → come raggiungere dalla sorgente il destinatario tramite un numero di dispositivi nel mezzo che rinviano i dati tra loro finché non si raggiunge la destinazione

Livello di trasporto → contiene i meccanismi per lavorare con entità (identificazione dei processi coinvolti nella trasmissione dei pacchetti)

Livello di applicazione → contiene la logica necessaria per supportare le varie applicazioni per utenti

2 livelli di indirizzamento affinché ogni entità abbia un indirizzo unico:

1. per indirizzare il sistema → IP ⇨ **indirizzo IP**
2. per indirizzare l'entità → TCP ⇨ **porta**

Applicazioni che utilizzano il protocollo TCP:

1. Simple Mail Transfer Protocol → invio e-mail base
2. File Transfer Protocol (FTP) → usato per inviare file da un sistema a un altro con comando user
3. Secure Shell → fornisce capacità di login da remoto in modo da poter permettere a utenti da casa di lavorare su server in remoto

Socket

concetto sviluppato negli anni 80 in ambiente UNIX; permette la comunicazione tra un processo client e un processo server

possono essere sia orientate alla connessione (TCP/IP) che senza connessione (UDP/IP)

due processi che hanno bisogno di comunicare comunicheranno attraverso socket; per far comunicare il client col server deve conoscere la socket su cui è in ascolto

interfacce standard; concatenazione di porta e indirizzo IP

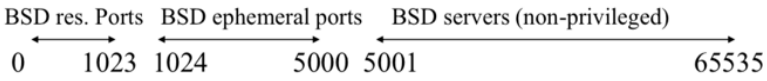
tre tipi di socket:

1. Stream socket → usa TCP; garantisce un trasferimento di dati connection oriented
2. Datagram socket → usa UDP; non sono garantite né consegna né l'ordine
3. Raw socket → gestisce layer più bassi (es. per gestire ping)

La socket è un end-point determinato da un indirizzo IP e una porta

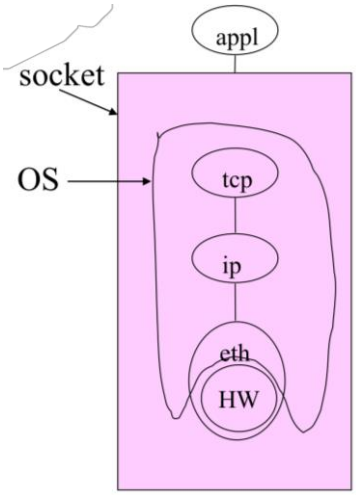
➔ è un end-point per una connessione IP ed è dove lo strato di applicazione si inserisce; due end-points determinano una connessione

Porte



0-1023 → riservate; es:

- telnet 23/tcp
- ftp 21/tcp
- finger 79/tcp
- snmp 161/udp



1024-5000 → effimere = aperte dal sistema operativo per i client

Le socket sono gestite dal sistema operativo (gestisce TCP, IP, …), per lavorare con esse, il sistema operativo genera un file descriptor quando viene chiamata la funzione *socket( )*

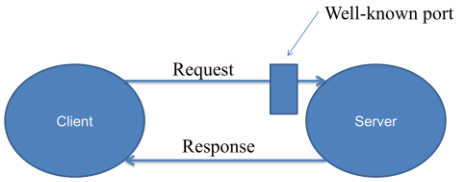
Primitive importanti:

Primitive	Packet sent	Meaning
LISTEN	(none)	Block until some process tries to connect
CONNECT	CONNECTION REQ.	Actively attempt to establish a connection
SEND	DATA	Send information
RECEIVE	(none)	Block until a DATA packet arrives
DISCONNECT	DISCONNECTION REQ.	This side wants to release the connection

si usa una struttura dati apposita per rappresentare l'indirizzo della socket

```
struct in_addr {
  in_addr_t s_addr;          /* 32-bit IPv4 addresses */
};
struct sockaddr_in {
  uint8_t       sin_len;     /* length of structure */
  sa_family_t   sin_family;  /* AF_INET */
  in_port_t     sin_port;    /* TCP/UDP Port num */
  struct in_addr sin_addr;   /* IPv4 address (above) */
  char sin_zero[8];          /* unused */
}
```

Come funziona logicamente una connessione client-server?

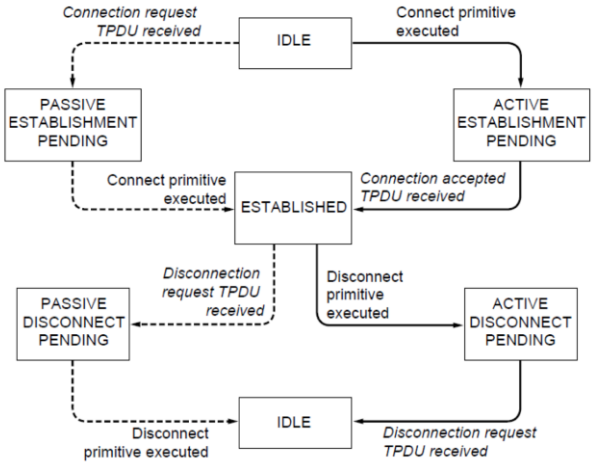


➔ Il server agisce passivamente, il client è invece attivo; tutti e due sono inattivi inizialmente

➔ Il server si mette in attesa di una connessione; il client invece (che vuole attivare una connessione) stabilisce che in questo momento vuole comunicare; il server risponde alla richiesta attiva del client

➔ Dopo aver stabilito la connessione TCP e ognuno manda messaggi/risponde ai messaggi e il server aspetta passivamente che il client invii un segnale di terminazione

➔ Il client chiama una disconnect che verrà ricevuta dal server, per cui si disconnette e lì termina la comunicazione tra di loro



Primitive per socket TCP:

Primitive	Meaning
SOCKET	Create a new communication end point
BIND	Associate a local address with a socket
LISTEN	Announce willingness to accept connections; give queue size
ACCEPT	Passively establish an incoming connection
CONNECT	Actively attempt to establish a connection
SEND	Send some data over the connection
RECEIVE	Receive some data from the connection
CLOSE	Release the connection

Per una Stream Socket (TCP/IP), una volta creata la socket, deve essere settata una connessione con una socket in remoto → il client fa *connect()* specificando sia la propria socket locale (sorgente) che la socket di destinazione

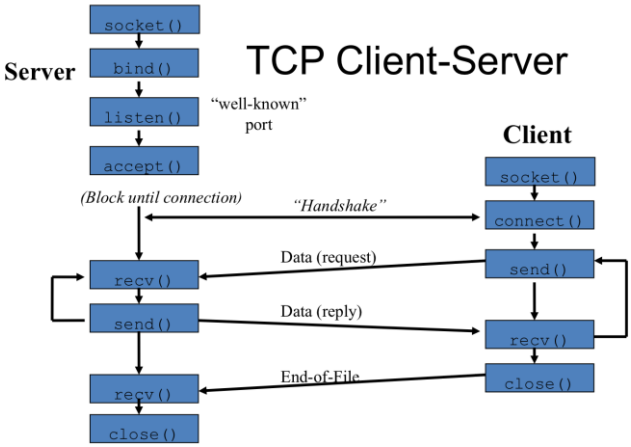
Una volta che la connessione è stata stabilita, possiamo ottenere anche il nome del dispositivo, che può essere utilizzato per identificarsi con *getpeername()*

Dal lato server, il server farà *listen()* (che indica che è pronto a ricevere) e successivamente *accept()*

Comandi per configurare la connessione:

- user → OS
  - *bind()* → specifico qual è la socket
  - *connect()* → specifico la socket di destinazione (TCP)
  - *sendto()* → manda dato a una socket (UDP)
- OS → user
  - *accept()* → riceve la socket di chi ha mandato il dato (TCP)
  - *recvfrom()* → riceve un dato da una socket (UDP)

il server ha solitamente 1 porta per ogni tipo di servizio fornito che richiede connessione; per ognuna delle richieste andrà a rispondere tramite una porta effimera fornita dal sistema operativo che sarà utilizzata per la comunicazione con il client finchè il client non ha terminato
La porta "ben nota" serve solo per instaurare la connessione, tengo poi le altre porte per la comunicazione con il client



Funzioni per connessioni Client-Server

Server TCP

int socket (int family, int type, int protocol) → crea una socket con:

- family: keyword per descrivere il protocollo sottostante (come vanno rappresentati i dati in IP); AF_INET (IPV4)
- type: tipo di socket (Stream socket, Datagram socket o Raw socket)
- protocol: 0

➔ ritorna intero (socket file descriptor) successo, -1 fallimento

int bind (int sockfd, const struct sockadrr *myaddr, socklen_t addrlen) →

- sockfd: intero restituito da *socket()*
- myaddr: puntatore ad una struttura che contiene la porta e l'IP (proprio del server); se non viene specificata la porta, la decide il sistema operativo
- addrlen: lunghezza della struttura myaddr

➔ ritorna 0 successo, -1 fallimento [es. errore EADDRINUSE (indirizzo già in uso)]

➔ nelle chiamate che passano la struttura indirizzo da processo utente a kernel viene passato un puntatore alla struttura ed un intero che rappresenta la grandezza della struttura così che il sistema operativo sappia esattamente quanti byte deve copiare nella sua memoria

➔ nelle chiamate che passano la struttura indirizzo dal kernel al processo utente vengono passati nella system call eseguita dall'utente un puntatore la struttura e un puntatore ad un intero in cui è stata preinserita la dimensione della struttura indirizzo quindi il sistema operativo sa già la dimensione della struttura e non ne oltrepassa i limiti

int listen (int sockfd, int backlog) → attiva la socket per un server TCP

- sockfd: intero restituito da *socket()*
- backlog: numero massimo di connessioni gestite dal server in contemporanea; di solito 5

int accept (int sockfd, struct sockadrr *cliaddr, socklen_t *addrlen) → restituisce un nuovo socket file descriptor con la connessione completata

- sockfd: intero restituito da *socket()*
- cliaddr e addrlen: socket del client

➔ restituisco un nuovo file descriptor perché il vecchio file descriptor deve rispondere alle richieste di connessione; avere un nuovo file descriptor implica avere una nuova socket perché il server restituisce al client una nuova porta

diversa da quella in cui è in ascolto per evitare che la porta utilizzata per l'ascolto possa non essere più raggiungibile da altri client

➔ ritorna -1 in caso di errore e setta errno

➔ quando ricevo una connessione posso anche creare un nuovo processo dedicato tramite *fork()* o un thread dedicato con *pthread_create()*

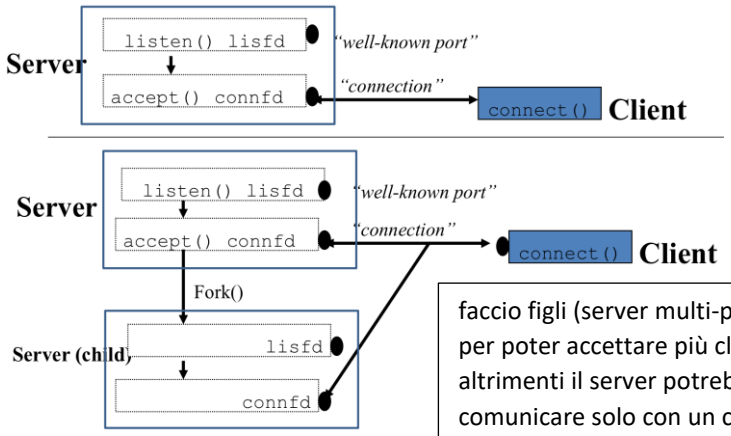
if pid==0 → processo figlio → chiudo la socket della *listen()* "dal mio punto di vista" (nel senso che non è una close che chiude la socket nell'intero sistema operativo)

il padre chiuderà da parte sua la socket del figlio perché il suo lavoro è generare le socket per tutti i processi figli (devo chiudere tutte le connessioni altrimenti le esaurisco tutte)

int close (int sockfd) → chiude la socket in lettura/scrittura

➔ ritorna -1 errore

```
lisfd = socket(....);
bind(lisfd,...);
listen(lisfd, 5);
while(1) {
  connfd = accept(lisfd,.....);
  if (pid = fork() == 0) {
     close(lisfd);
     doit(connfd);
     close(connfd);
     _exit(0);
  }
  close(connfd);
```



faccio figli (server multi-processo) per poter accettare più client, altrimenti il server potrebbe comunicare solo con un client per volta; potrei renderlo server multi-thread se invece che fare figli creo thread con pthread

Client-Server TCP

int recv (int sockfd, void * buff, size_t numBytes, int flags)

int send (int sockfd, void * buff, size_t numBytes, int flags)

➔ ritornano il numero di byte ricevuti/inviati; il sistema operativo potrebbe aver lanciato un'interrupt per cui n° di byte inviati/ricevuti potrebbe essere diverso da quelli da inviare/ricevere

➔ come read e write nelle pipe ma con flags:

- MSG_DONTWAIT (send non-bloccante)
- MSG_OOB (out of band data, 1 byte sent ahead) → aumenta la priorità
- MSG_PEEK (look, but don't remove) → usato in receive; serve per non far scartare il messaggio da chi lo ha ricevuto
- MSG_WAITALL (don't give me less than max)
- MSG_DONTROUTE (bypass routing table)

Client TCP

int connect (int sockfd, const struct sockaddr *servaddr, socklen_t addrlen) →

- sockfd: socket del client
- servaddr: puntatore a una struttura formata da porta e indirizzo IP che **devono** essere specificate
- addrlen: lunghezza dell'indirizzo

→ il client non necessita del *bind()* ⇨ il sistema operativo setterà automaticamente le informazioni dell'indirizzo IP e della porta

➔ ritorna -1 se fallisce, altrimenti il socket descriptor

UDP Client-Server

int recvfrom (int sockfd, void *buff, size_t numBytes, int flags, struct sockaddr *from, socklen_t *addrlen)

int sendto (int sockfd, void *buff, size_t numBytes, int flags, const struct sockaddr *to, socklen_t addrlen)

➔ simili a *recv()* e *send()* ma è riportato l'indirizzo di arrivo/partenza del pacchetto (dato che non è di default perché non c'è un canale di comunicazione unico come per stream socket)

int gethostname(char *hostname, int bufferLength) → usato da un server per conoscere il nome di chi si sta connettendo (evita l'overflow)

- hostname: nome host
- bufferLenght: limita la possibile lunghezza del nome dell'host; se eccede questa lunghezza viene troncato

unsigned long inet_addr (char *address); char* inet_ntoa (struct in_addr address)

→ funzioni per passare da notazione dotted a stringa di bit e viceversa

struct hostent* getbyhostname (const char* hostname) → dato un host name ritorna le informazioni dell'host; funzione usata nei DNS; la struct è composta da:

- char* h_name → nome ufficiale dell'host
- char** h_aliases → lista di alias/nomi alternativi
- short h_addrtype → address family
- short h_length → lunghezza dell'indirizzo
- char** h_addr_list → lista degli indirizzi (versione stringa)

struct hostent* gethostbyaddr (const void *addr, int len, int type) → dato un indirizzo ritorna le informazioni dell'host; simile a quello dei DNS

WinSock

Berkeley	WinSock
bzero()	memset()
close()	closesocket()
read()	not required
write()	not required
ioctl()	ioctlsocket()

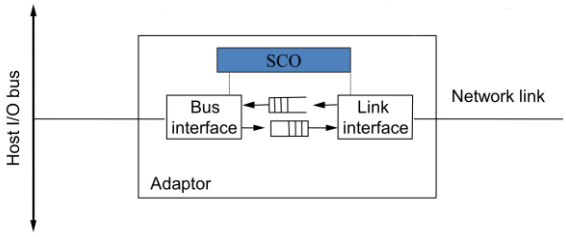
Simili alle Berkley socket; ci sono piccole differenze

3 modi differenti:

1. bloccante
2. non-bloccante
3. asincrono

ci sono 2 librerie per le WinSock (cambiano alcune API)

Network Adaptors



Network Adaptor = scheda di rete → è un'interfaccia tra il bus di I/O e rete

➔ costituita da 2 parti separate che interagiscono tramite FIFO che maschera l'asincronia tra rete e bus interno

- la prima interfaccia interagisce con la CPU della scheda
- la seconda interfaccia interagisce con la rete implementando livello fisico e di collegamento

Il sistema controllato da una SCO (sottosistema di controllo della scheda)

La SCO e la CPU comunicano attraverso dei flag del registro CSR (Control Status Register) della SCO che entrambi leggono; ogni bit del CSR ha un significato

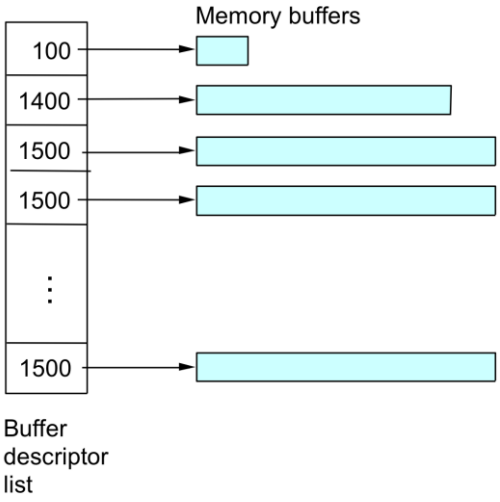
```
/* CSR
* leggenda: RO - read only; RC - Read/Clear (writing 1 clear, writing 0 has no effect);
* W1 write-1-only (writing 1 sets, writing 0 has no effect)
* RW - read/write; RW1 - Read-Write-1-only
*/
#define LE_ERR 0X8000
……..
#define LE_RINT 0X0400  /* RC richiesta di interruzione per ricevere un pacchetto */
#define LE_TINT 0X0200  /* RC pacchetto trasmesso */
……..
#define LE_INEA 0X0040  /* RW abilitazione all'emissione di un interrupt da parte
……..                       dell'adaptor verso la CPU */
#define LE_TDMD 0X0008  /* W1 richiesta di trasmissione di un pacchetto dal device
                           driver verso l'adaptor */
```

L'host controlla cosa accade in CSR in 2 modi:

1. Busy Waiting → CPU esegue un test continuo di CSR finchè non viene modificato indicando l'operazione da eseguire; ragionevole solo per i calcolatori che devono solo attendere e inviare pacchetti (router)
2. Interrupt → adaptor invia interrupt all'host che fa partire un interrupt handler che legge CSR per capire l'operazione da fare

Il trasferimento dei dati può avvenire in 2 modi:

1. Direct Memory Access → una parte della RAM dell'host viene messa a disposizione della scheda di rete ⇨ necessito poca memoria per la scheda di rete stessa; è la più usata
   - creo un array di 64 buffer ognuno da 1500 bytes (Buffer Descriptor)
   - la CPU/processo sa quali posizioni sono libere e dal puntatore di una di esse scrive in memoria i dati e segnala in quale posizione li ha messi
   - posso mettere un frame su più buffer se è più grande di esso
   - metto frame distinti su buffer distinti



2. Programmed I/O → scambio dati tra memoria e adaptor passa per la CPU; bufferizzo i dati e vengono trasmessi un po' per volta; la memoria deve essere dual port perché il processore e l'adaptor possono leggere e scrivere sulla porta (deve essere sincronizzato)

Quando un messaggio viene inviato da un utente in una certa socket:

1. Il sistema operativo copia il messaggio dal buffer della memoria utente in una zona di buffer descriptor (BD)
2. Tale messaggio viene processato da tutti i livelli protocollari (esempio TCP, IP, device driver) che provvedono ad inserire gli opportuni header + checksum e ad aggiornare gli opportuni puntatori presenti nel BD in modo da poter sempre ricostruire il messaggio
3. Quando il messaggio ha completato l'attraversamento del protocol stack (⇨ è all'interno del buffer), viene avvertita la SCO dell'adaptor dal device driver attraverso il set dei bit del CSR (LE_TDMD e LE_INEA). Il primo invita la SCO ad inviare il messaggio sulla linea. Il secondo abilita la SCO ad inviare una interruzione (⇨"1. devo trasmettere qualcosa e 2. ti abilito a interrompermi per segnalarmi che l'hai trasmesso")
4. La SCO dell'adaptor invia il messaggio sulla rete
5. Una volta terminata la trasmissione, la SCO notifica il termine alla CPU attraverso il set del bit (LE_TINT) del CSR e scatena una interruzione
6. Tale interruzione avvia un interrupt handler che prende atto della trasmissione, resetta gli opportuni bit (LE_TINT e LE_INEA) e libera le opportune risorse (operazione semsignal su xmit_queue)

Device driver → collezione di routine di OS che serve per unire il sistema operativo all'hardware sottostante specifico dell'adaptor

Esempio routine di richiesta di invio di un messaggo sul link

```
#define csr ((u_int) 0xffff3579 /*CSR address*/
Transmit(Msg *msg)
{    descriptor *d;
     semwait(xmit_queue);      /* abilita non piu' di 64 accessi al BD*/
     d=next_desc();
     prepare_desc(d,msg);
     semwait(mutex);           /* abilità a non piu' di un processo (dei potenziali 64)
                               alla volta la trasmissione verso l'adaptor */
     disable_interrupts();     /* il processo in trasmissione si protegge da eventuali
                               interruzioni dall'adaptor */
     csr= LE_TDMD | LE_INEA;   /* una volta preparato il messaggio invita la SCO dell'adaptor a
                               trasmetterlo e la abilita la SCO ad emettere una interruzione
                               una volta terminata la trasmissione */
     enable_interrupts();     /* riabilita le interruzioni */
     semsignal(mutex);         /* sblocca il semaforo per abilitare un altro processo a
}                              trasmettere */
```

"next_desc()" ritorna il prossimo buffer descriptor disponibile nel buffer descriptor list
"prepare_desc(d,msg)" il messaggio msg nel buffer d in un formato comprensibile dall'adaptor

Cosa succede quando gli interrupt vengono disabilitati con LE_INEA? La CPU

- disabilita le interruzioni
- legge il CSR per capire cosa fare; 3 possibilità:
  1. c'è stato un errore
  2. trasmissione completata
  3. frame ricevuto

```
lance_interrupt_handler()
{
    disable_interrupts();

    /* some error occurred */
    if (csr & LE_ERR)
    {
        print_error(csr);
        /* clear error bits */
        csr = LE_BABL | LE_CERR | LE_MISS | LE_MERR | LE_INEA;
        enable_interrupts();
        return();
    }

    /* transmit interrupt */
    if (csr & LE_TINT)
    {
        /* clear interrupt */
        csr = LE_TINT | LE_INEA;

        /* signal blocked senders */
        semSignal(xmit_queue);

        enable_interrupts();
        return(0);
    }

    /* receive interrupt */
    if (csr & LE_RINT)
    {
        /* clear interrupt */
        csr = LE_RINT | LE_INEA;

/* process received frame */
lance_receive();
        enable_interrupts();
        return();
    }
}
```

Cosa succede ai pacchetti dati?

1. L'utente fa una *send()* sulla socket e il pacchetto dati viene messo in uno dei 64 buffer

Il sistema operativo elabora il pacchetto mettendo gli header

2. TCP
3. IP
4. Ethernet
5. viene settato il CSR in memoria ⇨ è settato automaticamente nella scheda di rete
6. La SCO automaticamente capisce che è stato settato il CSR e prende il pacchetto dalla posizione corretta del buffer lo modifica e lo rappresenta proprio come un segnale da trasmettere in impulsi
7. La scheda di rete invia il pacchetto sulla rete

8. Appena inviato il network adaptor
    a. se abilitato invia un interrupt alla CPU
    b. altrimenti aspetta
9. Quando la CPU riceve l'interrupt, chiama l'interrupt handler all'interno della socket
10. L'interrupt handler andrà a leggere nuovamente il CSR che è stato modificato dalla scheda di rete per capire cosa è successo (errore, è stato trasmesso il pacchetto, è arrivato un nuovo pacchetto…)