

Programmazione di rete (sockets)

Programmazione di rete

- ❑ Consente di implementare un protocollo usando i servizi forniti dai protocolli di livello più basso
 - ❖ Esempio: implementazione di un protocollo di livello applicazione usando i servizi di livello trasporto (forniti dal S.O.)
- ❑ L'interazione avviene attraverso un'interfaccia (API - *Application Program Interface*)
- ❑ Generalmente gli standard di protocollo
 - ❖ specificano le funzionalità dei protocolli (il servizio offerto)
 - ❖ non specificano le interfacce (come il servizio viene fruito)
- ❑ Per facilitare la **portabilità** del software si tende a utilizzare interfacce standardizzate

Esempi di interfacce

❑ NDIS

(Network Driver Interface Specification)
originariamente sviluppato
da Microsoft e 3Com)

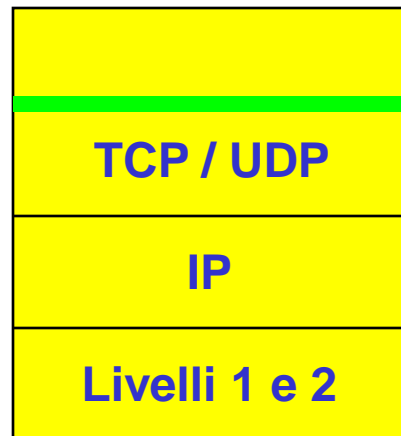


Device driver

Firmware della scheda di rete

Scheda di rete

❑ Socket



applicativo

Sistema
operativo

Tipi di interfacce

❑ **Interfaccia procedurale:**

è costituita da un insieme di procedure che realizzano le funzionalità richieste (per esempio aprire una connessione o inviare un datagram) - interazione tipicamente **sincrona**

❑ **Interfaccia basata su messaggi**

è costituita da un insieme di messaggi che fruitore e fornitore si scambiano secondo un certo protocollo - interazione tipicamente **asincrona**

❑ **Interfaccia basata su meccanismi specifici** (p. es. interrupt software)

Interfaccia di rete «socket»

- ❑ E' diventata in pratica lo standard per accedere ai servizi della rete Internet
- ❑ Nata in ambito BSD Unix, è stata implementata per tutti i principali S.O. (per es. *Windows sockets*)
- ❑ E' basata su un'astrazione chiamata **socket** (da cui il nome)
- ❑ Documentazione:
 - ❖ W. R. Stevens, "Unix Network Programming", Prentice Hall, 1990
 - ❖ D. E. Comer, D. L. Stevens, "Internetworking with TCP/IP", Vol. 3, Prentice Hall, 1997

Funzioni dell'API «socket» per l'accesso al layer 4 (TCP)

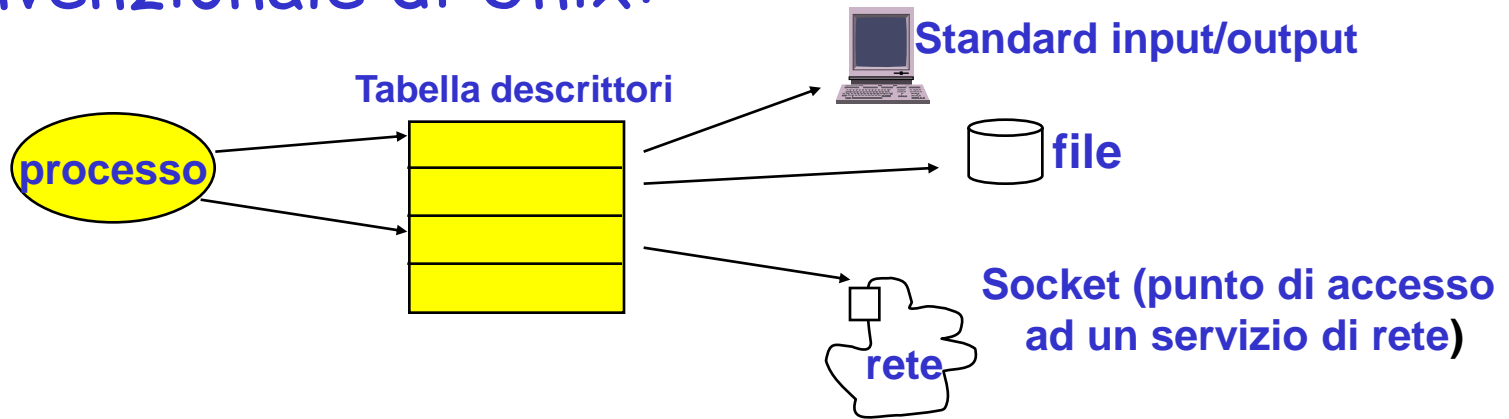
- ❑ Allocare risorse locali per la comunicazione
- ❑ Specificare gli end points (locale e remoto)
- ❑ Iniziare una connessione (lato client)
- ❑ Attendere una connessione (lato server)
- ❑ Spedire/ricevere dati
- ❑ Determinare/essere notificati quando arrivano i dati
- ❑ Generare/trattare dati urgenti
- ❑ Terminare localmente una connessione in modo ordinato
- ❑ Rispondere a richieste di terminazione e trattare le condizioni di errore che causano terminazione
- ❑ Rilasciare le risorse locali quando termina una connessione

Funzioni dell'API «socket» per l'accesso al layer 4 (UDP)

- ❑ Allocare risorse locali per la comunicazione
- ❑ Specificare gli end points (locale e remoto)
- ❑ Spedire un datagram
- ❑ Ricevere un datagram
- ❑ Rilasciare le risorse locali al termine

Caratteristiche generali

- ❑ Interfaccia Procedurale in linguaggio C
- ❑ Nasce con l'idea di estendere il modello di I/O convenzionale di Unix:



- ❑ L'adesione al modello però non può essere totale (il paradigma *open-read-write-close* si adatta bene solo alle comunicazioni connection-oriented)

Caratteristiche generali (cont.)

- Definisce procedure **generali** che prevedono l'utilizzo di diversi stack di protocollo (TCP/IP=caso particolare)

`maketcpconnection(...)`

`makeconnection(tcp,...)`



- In realtà è stata definita riflettendo i meccanismi dei protocolli TCP/IP
 - ❖ modello asimmetrico di connessione
 - ❖ stream senza delimitazione dei messaggi
 - ❖ ...

I socket

- ❑ Un **socket** è l'astrazione di un'interfaccia di comunicazione tra processi (endpoint, SAP - service access point).
- ❑ I socket “vivono” in **domini**, ciascuno con la propria **famiglia di protocolli** e **famiglia di indirizzi**
- ❑ La comunicazione tra domini diversi è **impossibile**.

Esempi di domini

Dominio	Famiglia di protocolli	Famiglia di indirizzi
ARPA Internet	PF_INET	AF_INET
Internet con IPv6	PF_INET6	AF_INET6
ISO/OSI	PF_ISO	AF_ISO
Unix pipes	PF_UNIX	AF_UNIX

Esempio di «protocol families»

PF_LOCAL, PF_UNIX, PF_FILE Local to host (pipes and file-domain)

PF_INET IP protocol family

PF_AX25 Amateur Radio AX.25

PF_IPX Novell Internet Protocol

PF_APPLETALK Appletalk DDP

PF_NETROM Amateur radio NetROM

PF_BRIDGE Multiprotocol bridge

PF_ATMPVC ATM PVCs

PF_X25 Reserved for X.25 project

PF_INET6 IP version 6

PF_ROSE Amateur Radio X.25 PLP

PF_DECnet Reserved for DECnet project

PF_NETBEUI Reserved for 802.2LLC project

PF_SECURITY Security callback pseudo AF

PF_KEY PF_KEY key management API

PF_NETLINK, PF_ROUTE routing API

PF_PACKET Packet family

PF_ASH Ash

PF_ECONET Acorn Econet

PF_ATMSVC ATM SVCs PF_SNA Linux SNA Project

PF_IRDA IRDA sockets

PF_PPPOX PPPoX sockets

PF_WANPIPE Wanpipe API sockets

PF_BLUETOOTH Bluetooth sockets

Caratteristiche dei socket : TIPO

□ Identifica la *tipologia di servizio* accessibile tramite il socket. I tipi supportati normalmente sono:

❖ **SOCK_STREAM**

flusso bidirezionale continuo (senza delimitazioni) di byte, trasmessi in modo affidabile, in sequenza e senza duplicazioni (connection-oriented service offerto a livello 4, per es. **TCP**).

❖ **SOCK_DGRAM**

trasmissione bidirezionale di messaggi (**delimitati**) senza garanzia di affidabilità, ordinamento o assenza di duplicazioni (connectionless service, offerto a livello 4, per es. **UDP**).

Socket Raw

❖ **SOCK_RAW**

accesso diretto ai protocolli di rete di basso livello (per esempio livello 2 o 3)

Di uso **sconsigliato**, a meno di particolari problemi

Caratt. dei socket : PROTOCOLLO

□ In ogni dominio, e per ogni tipo di socket, è possibile selezionare un particolare protocollo da utilizzare. Per esempio, nel dominio PF_INET:

tipo **SOCK_STREAM**

❖ Protocollo TCP (IPPROTO_TCP)

tipo **SOCK_DGRAM**

❖ Protocollo UDP (IPPROTO_UDP)

tipo **SOCK_RAW**

❖ Protocollo ICMP (IPPROTO_ICMP)

❖ Pacchetti IP (IPPROTO_RAW)

Caratter. dei socket: OPZIONI

- Specificano altre caratteristiche varie del socket.

Esempi:

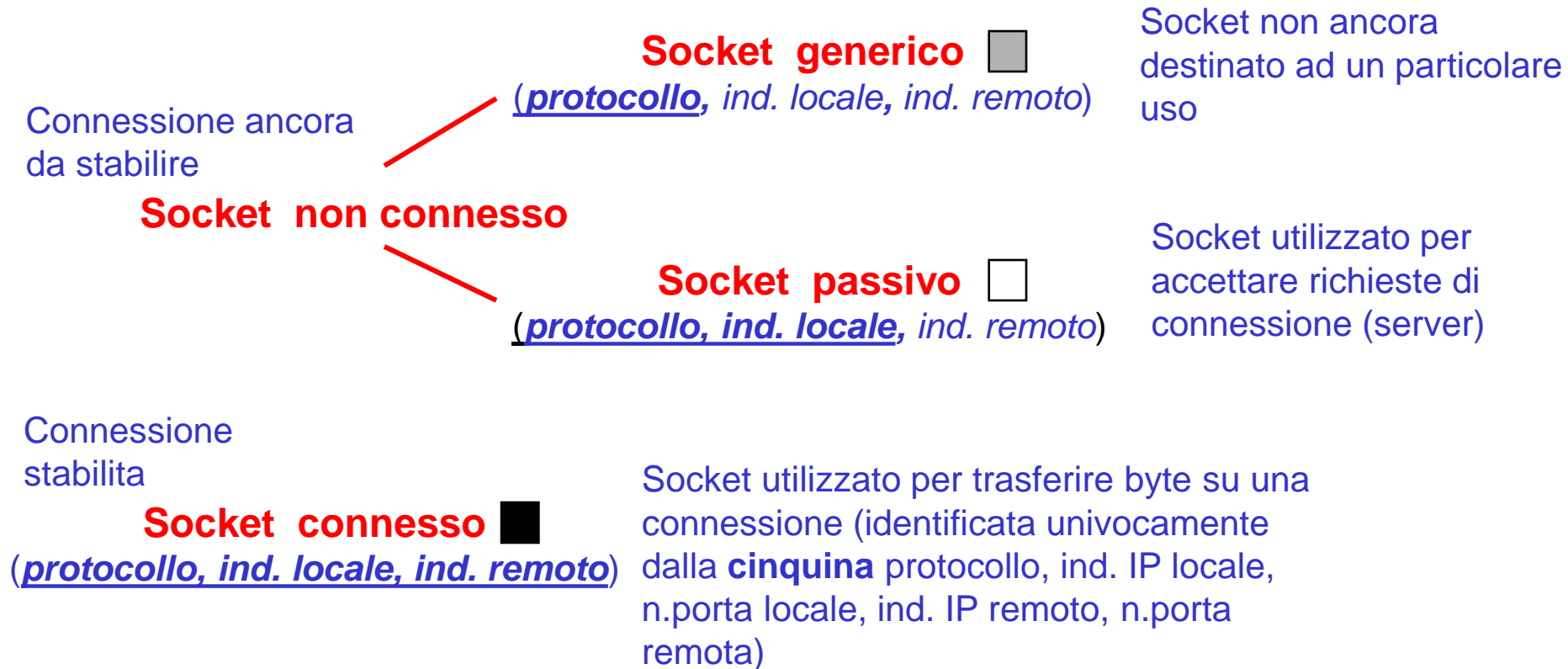
- ❖ **SO_SNDBUF** (dimensione del buffer per la trasmissione)
- ❖ **SO_RCVBUF** (dimensione del buffer per la ricezione)
- ❖ **SO_LINGER** (se abilitata, ritarda la chiusura delle connessioni in presenza di dati nei buffer)
- ❖ **SO_KEEPALIVE** (se abilitata, tra i socket connessi vengono scambiati messaggi periodici. Nel caso in cui non arrivi la risposta, la connessione viene chiusa)

Struttura dati associata ad un socket generico

dominio
tipo
protocollo
indirizzo locale
indirizzo remoto
opzioni

Utilizzo dei socket STREAM

- ❑ Per realizzare i diversi tipi di interazione servono diversi tipi di endpoint:



Primitive dell'API (interfaccia)

- ❑ Sono di natura sincrona (bloccanti)
- ❑ Segnalano condizioni di errore restituendo il valore -1
- ❑ Il codice relativo all'errore verificatosi si può leggere:
 - ❖ in ambiente UNIX leggendo la variabile **errno**

```
void err_fatal(char *mes) {  
    printf("%s, errno=%d\n", mes, errno);  
    perror(""); exit(1); }
```

- ❖ in ambiente Windows Sockets chiamando la funzione

WSAGetLastError()

```
void err_fatal(char *mes) {  
    printf("%s, errno=%d\n", mes, WSAGetLastError());  
    perror(""); exit(1);  
}
```

Creazione di un socket

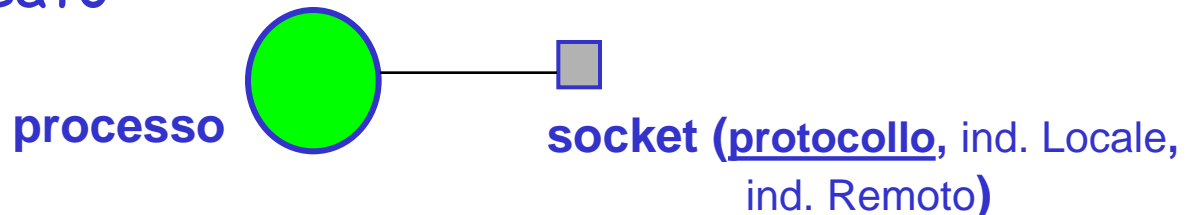
```
int socket (int family, int type, int  
           protocol)
```

- family dominio del socket
- type tipo di socket
- protocol protocollo utilizzato dal socket

Es. `socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)`

Valore di ritorno :

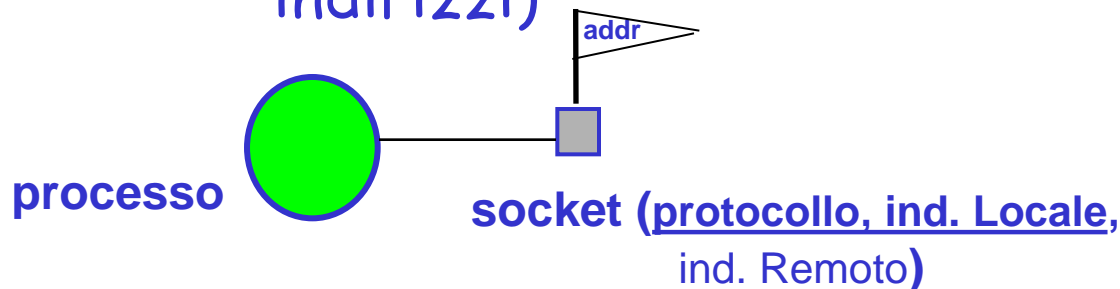
- ❖ l'identificativo locale (in Unix il file descriptor)
del socket creato



Assegnazione di un indirizzo locale di rete ad un socket

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

- **socket** socket cui assegnare l'indirizzo
- **addr** puntatore all'indirizzo da assegnare
- **addrlen** lunghezza della struttura sockaddr (che cambia secondo la famiglia di indirizzi)



Nota: In UNIX è possibile bind a porte <1024 solo da superutente

Come si specifica un endpoint (SAP)

- ❑ Le informazioni sono raccolte in una struttura:

```
struct sockaddr {                /*struct per endpoint*/
    uint8_t  sa_len;              /*  lungh.  totale  */
    uint16_t sa_family;           /*  tipo di indirizzi */
    uint8_t  sa_data[14];        /*  bytes dell'indir.*/
};
```

Struttura usata solo per effettuare corretto casting
, cioè non avere warnings in compilazione, poiché le
funzioni si aspettano struct sockaddr *

End point per IPv4

□ È consigliabile usare una struttura specifica

```
struct sockaddr_in{
    uint8_t  sin_len;          /* lungh. totale*/
    uint8_t  sin_family;      /* addr. family*/
    uint16_t sin_port;         /* n. porta */
    struct in_addr sin_addr; /* ind.IP, unico */
                                /* campo uint32_t */
    char sin_zero[8];
}
```

Nel dominio IPv4 (AF_INET) l'indirizzo dell'endpoint è specificato come composto da due parti:
porta (2 bytes) e IP address (4 bytes)

Esempio di uso di struct sockaddr_in

```
struct in_addr {
    uint32_t s_addr;
}
struct sockaddr_in {
    uint8_t    sin_len;    /* non presente in alcune
                           versioni, per es. su Linux */
    uint8_t    sin_family;
    uint16_t   sin_port;
    struct in_addr sin_addr;
    uint8_t    sin_zero[8];
};
```

```
struct sockaddr_in saddr;
saddr.sin_len      = sizeof(struct sockaddr_in);
saddr.sin_family   = AF_INET;
saddr.sin_port     = 80;
saddr.sin_addr.s_addr = htonl(INADDR_ANY);
                    /* valore 0.0.0.0 */
result = bind(s, (struct sockaddr *)&saddr, sizeof(saddr));
```


Network byte order

- ❑ In Internet gli interi sono rappresentati e trasferiti usando l'ordine dei bytes cosiddetto «big-endian» (= network byte order)



Primo byte immagazzinato (all'indirizzo di memoria minore) o spedito sulla rete

- ❑ Quindi nella struttura `sockaddr_in` essi devono essere inseriti in ordine big-endian

Conversione di byte order

- ❑ Se il programma gira su macchina little-endian è necessaria conversione prima dell'assegnamento nella struct sockaddr_in
- ❑ Per facilitare la portabilità si utilizzano le funzioni fornite dall'API (che non fanno nulla sulle macchine big-endian)

```
#include <netinet/in.h>

uint16_t htons(uint16_t x); /* host-to-network short */
uint32_t htonl(uint32_t x); /* host-to-network long  */
uint16_t ntohs(uint16_t x); /* network-to-host short */
uint32_t ntohl(uint32_t x); /* network-to-host long  */
```

Altre funzioni di conversione

- Dalla notazione «decimale puntata» (dotted decimal, es. 10.0.0.3) a network byte order e viceversa:

```
int inet_aton(const char *str, struct in_addr *addr)
    /* ritorna 1 se successo, 0 se stringa non valida */
    /* es. 10.0.0.3 -> 0x0A 0x00 0x00 0x03 */

uint32_t inet_addr (const char *str)
    /* SCONSIGLIATA:
       ritorna INADDR_NONE ( $2^{32}-1$ ) se errore
       non funziona con l'indirizzo di broadcast */

char *inet_ntoa(struct in_addr addr)
```

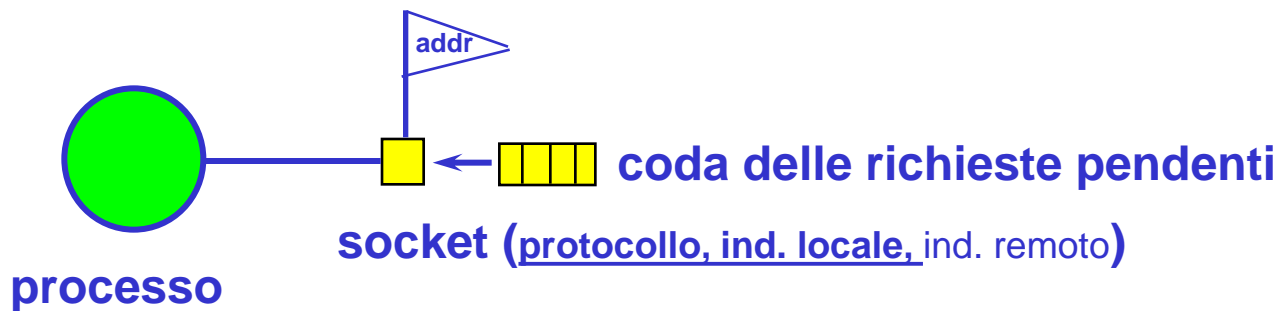
Connessione di socket STREAM

- ❑ Segue il modello Client/Server tipico del TCP:
 - ❖ Open Passiva (lato server), processo server **bloccato**
 - ❖ Open Attiva (lato client), client **bloccato**
 - ❖ Il server accetta la richiesta di connessione
 - ❖ Completato l'handshake, Client e Server si **sbloccano** e possono **entrambi** trasmettere e ricevere dati.
 - ❖ In qualunque istante possono decidere di chiudere la connessione, oppure chiudere soltanto uno dei due flussi di dati (**shutdown**)

Predisposizione del socket per ricevere richieste di connessione (server)

```
int listen (int socket, int backlog)
```

- `socket` socket sul quale mettersi in attesa
- `backlog` lunghezza massima della coda delle richieste pendenti



Richiesta di connessione (client)

```
int connect (int socket, const struct  
sockaddr *destaddr, socklen_t addrlen)
```

- **socket** socket che si vuole connettere
- **destaddr** puntatore all'indirizzo del server remoto cui si vuole indirizzare la richiesta
- **addrlen** lunghezza della struttura



Accettazione di una richiesta di connessione (server)

```
int accept (int socket, struct sockaddr  
            *srcaddr, socklen_t *addrlen)
```

- **socket** socket dove si riceve la richiesta
- **srcaddr** puntatore all'indirizzo del socket
remoto con cui viene stabilita la
connessione
- **addrlen** puntatore alla lunghezza della struttura
(inizializzata dal chiamante)
che conterrà l'indirizzo remoto

Valore di ritorno:

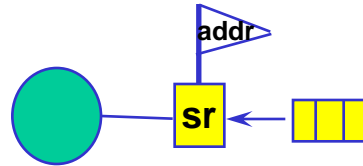
- ❖ l'identificativo locale di **una copia del socket locale**,
connessa al socket remoto.

SERVER

CLIENT

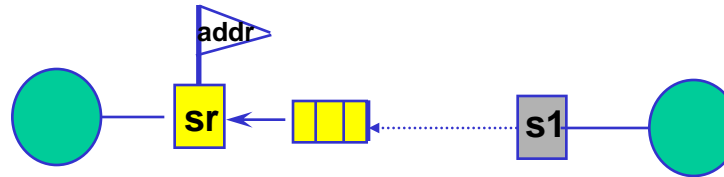


`listen(sr, backlog)`



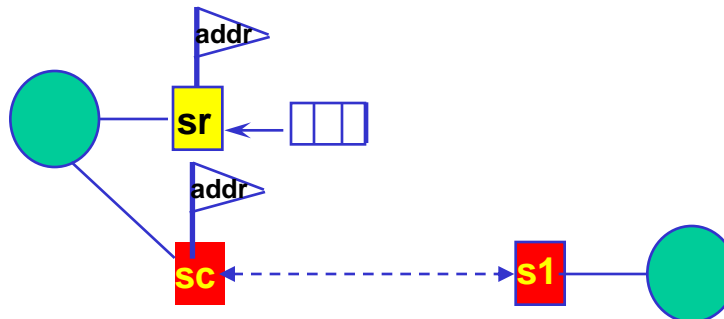
`sc = accept(sr, srcname, sizeof(srcname))`

attesa



`connect(s1, addr, sizeof(addr))`

attesa



Apertura Connessione (Client)

```
uint32_t          taddr_n; /*NB: in network-byte-order*/
uint16_t          tport_n; /*NB: in network-byte-order*/
struct sockaddr_in saddr;
SOCKET            s;
int               result;
. . .
s = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
if (s == INVALID_SOCKET)
    err_quit("socket() failed");

saddr.sin_family      = AF_INET;
saddr.sin_port        = tport_n;
saddr.sin_addr.s_addr = taddr_n;

result = connect(s, (struct sockaddr*)&saddr, sizeof(saddr));
if (result == -1)
    err_quit("connect() failed");
```

Apertura Connessione (Server)

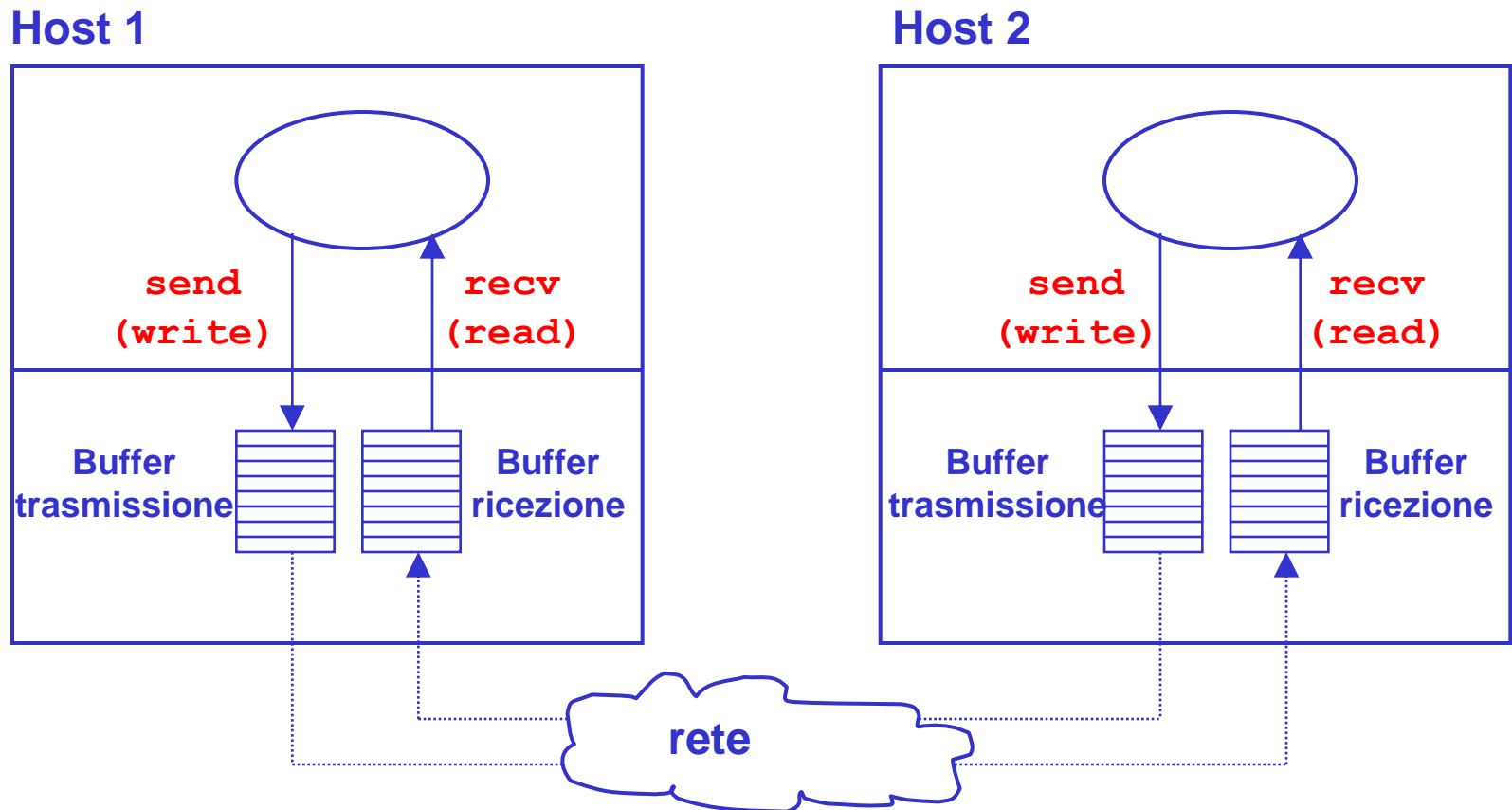
```
uint16_t          lport_n; /* porta server */
struct sockaddr_in saddr, caddr;
socklen_t         addrlen;
SOCKET            s, s1;
int               result;
. . .
s = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);

saddr.sin_family      = AF_INET;
saddr.sin_port        = lport_n;
saddr.sin_addr.s_addr = INADDR_ANY;

result = bind(s, (struct sockaddr *) &saddr, sizeof(saddr));
result = listen(s, backlog);
addrlen = sizeof(struct sockaddr_in);
s1 = accept(s, (struct sockaddr *) &caddr, &addrlen);
```

Trasferimento dati su connessioni

- Avviene secondo il seguente modello:



Invio dati su una connessione

```
ssize_t send (int socket, const void *data,  
              size_t datalen, int flags)
```

- ❑ **socket** socket connesso attraverso cui
 si inviano i dati
- ❑ **data** buffer contenente i dati da inviare
- ❑ **datalen** lunghezza del blocco di dati da inviare
- ❑ **flags** specifica eventuali opzioni
 (es. dati out-of-band)

Valore di ritorno:

- ❖ numero di byte effettivamente inviati, <0 se errore (es. socket non connesso). Si blocca se il buffer locale è pieno.
- ❖ NB: Se connessione chiusa, il programma riceve **SIGPIPE**: intercettare il segnale o usare il flag **MSG_NOSIGNAL**

Ricezione dati su una connessione

`ssize_t recv (int socket, void* buffer, size_t buflen, int flags)`

- ❑ `socket` socket connesso attraverso cui si ricevono i dati
- ❑ `buffer` buffer di ricezione
- ❑ `buflen` lunghezza del buffer di ric.
- ❑ `flags` specifica eventuali opzioni (es. "out-of-band data")

Valore di ritorno:

n. di byte ricevuti, che può essere minore della lunghezza del buffer (0 se è stata chiusa la connessione correttamente, <0 se errore).

Se non ci fossero dati da leggere, bloccherebbe (non ritorna 0)

NB: In particolari condizioni («connection reset by peer», es. l'altro host ha chiuso senza leggere tutti i dati) `recv` può anche ritornare un valore < 0 (errore) - importante saperlo per i server - per evitare exit (molte funzioni dello «Stevens» escono in questo caso)

Esempio: invio di un buffer

```
int writen(SOCKET s, char *ptr, size_t nbytes)
{
    size_t nleft; ssize_t nwritten;

    for (nleft=nbytes; nleft > 0; )
    {
        nwritten = send(s, ptr, nleft, 0);
        if (nwritten <=0) /* error */
            return (nwritten);
        else {
            nleft -= nwritten;
            ptr += nwritten;
        }
    }
    return (nbytes - nleft);
}
```

Esempio: ricezione di n byte

NB: Solo se è NOTO il numero di bytes da leggere!

```
int readn (SOCKET s, char *ptr, size_t len)
{
    ssize_t nread; size_t nleft;

    for (nleft=len; nleft > 0; )
    {
        nread=recv(s, ptr, nleft, 0);
        if (nread > 0)
        {
            nleft -= nread;
            ptr += nread;
        }
        else if (nread == 0) /* conn. closed by
                               party */
            break;
        else /* error */
            return (nread);
    }
    return (len - nleft);
}
```

Esercizio: ricezione fino all' «a capo»

Esercizio:

- Modificare la funzione precedente in modo da:
 - ❖ Leggere fino a quando si trova un «a capo» (LF «line feed», '\n')
 - ❖ Utilizzando un buffer più grande di 1 byte
 - ❖ Conservando i dati in più eventualmente letti per la successiva chiamata (è necessario gestire uno stato, all'interno della funzione, tramite per es. una variabile locale static)

Chiusura

- ❑ Una connessione può essere chiusa in modo corretto tramite:

int close(int socket)

- ❑ La close rilascia tutte le risorse associate al socket
- ❑ Controllare il valore di ritorno
 - ❖ Principalmente per rilevare errori non riportati in precedenza perché non si è avuto necessità di fare send
- ❑ Il sistema operativo chiude i files e socket ancora aperti al termine dell'esecuzione
 - ❖ Non è una buona pratica di programmazione

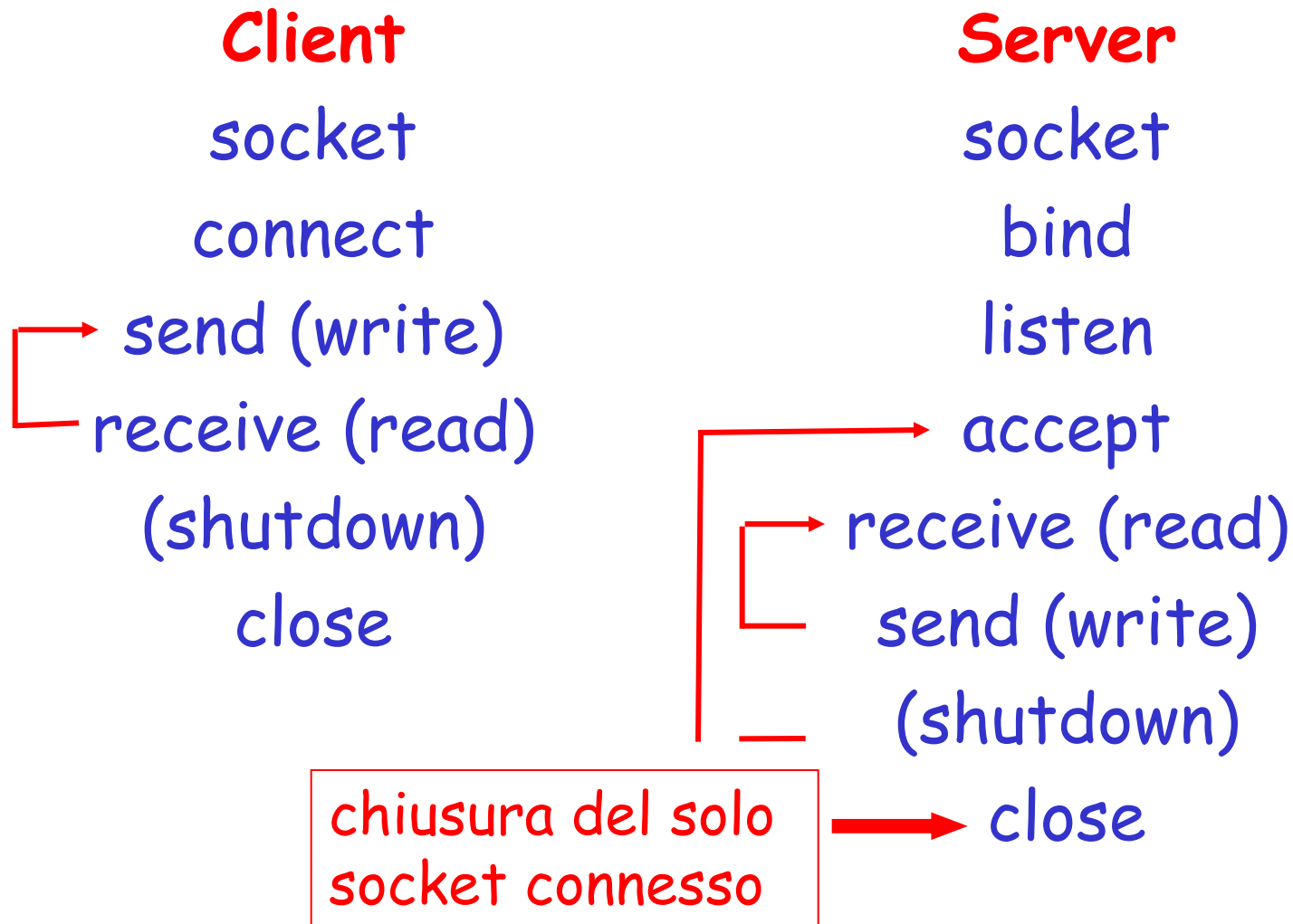
Chiusura parziale

- Una connessione può essere chiusa anche solo limitatamente ad una direzione con la primitiva:

int shutdown (int socket, int how)

- La modalità di chiusura è controllata dal parametro **how**:
 - ❖ SHUT_RD disabilita solo la ricezione di dati dal socket
 - ❖ SHUT_WR disabilita solo l'invio di dati sul socket
 - ❖ SHUT_RDWR disabilita entrambe le operazioni sul socket
- L'operazione di shutdown **non rilascia le risorse del socket**. Solo la close le rilascia.

Sommario operazioni su TCP



Files di header

- Tutte le functions, i tipi, le costanti e le variabili standard per i socket sono definiti nei seguenti headers:

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

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

```
#include <sys/time.h> /*per timeouts*/
```

```
#include <netinet/in.h> /*per Internet*/
```

```
#include <unistd.h> /*per close*/
```

Windows sockets

- ❑ L'API Windows Sockets implementa tutte le funzioni principali della API UNIX, più alcune altre message/event oriented
- ❑ Il trattamento file/socket non è omogeneo
- ❑ I Windows Sockets sono basati su una libreria a link dinamico (DLL) che deve essere inizializzata e linkata, e infine rilasciata tramite:

```
WSAStartup (...);  
WSACleanup (...);
```

Schema di programma C windows

```
...  
#include <winsock.h>  
...  
void main()  
{  
...  
/* Inizializzazione Windows Sockets */  
... WSStartup(...);  
  
/* Uso delle primitive Unix-like */  
  
/* Rilascio risorse Windows Sockets */  
.. .. WSACleanup();  
}
```

Per facilitare la portabilità è possibile racchiudere in un header file tutte le definizioni che cambiano tra UNIX e Windows (vedi mysocket.h)

Consigli

❑ Controllare sempre il valore di ritorno delle funzioni

❖ Utile il file `sockwrap.c` derivato da `wrapsock.c` dello Stevens

- Attenzione a cosa fanno in caso di errori (`exit` può non essere appropriato per un server)
- Attenzione a mischiare funzioni di lettura non bufferizzate e bufferizzate (es. `myread()`)

❖ Stesse funzioni socket ma con iniziale maiuscola

- Controllo e stampa messaggi di errore relativi al valore di ritorno della funzione
- Ri-esecuzione della funzione in caso di interruzione dovuta a segnale UNIX (es. `SIGCHLD`)

Architettura di un client

□ Più semplice di un server

- ❖ Spesso non è necessaria la concorrenza
 - La concorrenza può essere utile (ma non è indispensabile) per avere l'interfaccia utente sempre funzionante, anche con cliente in attesa
- ❖ Ridotte esigenze di sicurezza
 - Eseguito senza speciali privilegi
 - Crash del client generalmente non impatta su altre applicazioni
- ❖ Richiesta di prestazioni non critica

Architettura di un server

❑ Complessa, per molte ragioni

- ❖ Spesso è necessaria la concorrenza altrimenti le prestazioni possono essere molto scarse
- ❖ La sicurezza e robustezza sono aspetti importanti
 - Eseguito generalmente con privilegi super-utente
 - Il server deve girare ininterrottamente, senza interruzioni, per giorni/mesi/anni (daemon)
 - Un crash coinvolge numerosi clienti e li blocca tutti
- ❖ Richiesta di prestazioni critica, importanti gli aspetti di scalabilità
 - Può essere necessario mantenere informazioni di stato per ogni singolo client

Esempio di client e server TCP

□ Applicazione «Echo»

- ❖ Il client legge una linea di testo, la invia al server
- ❖ Il server reinvia indietro in forma identica quanto ricevuto
- ❖ Il client resta in attesa di una risposta dal server. Quando ricevuta, a fronte di uno specifico input (per esempio la stringa speciale «stop»), termina

Socket di tipo SOCK_DGRAM

❑ Servizio connectionless/datagram:

- ❖ non servono operazioni preliminari
- ❖ si trasmette/riceve **un datagram per** volta (non c'è bufferizzazione nel socket)
- ❖ ad ogni trasmissione si deve specificare il destinatario (che può anche cambiare)

❑ `socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP)` ;

❑ Terminate le operazioni, il socket deve essere chiuso (per rilasciare le risorse)

Invio di datagram

```
ssize_t sendto (int socket, const void*  
data, size_t datalen, int flags, const  
struct sockaddr *addr, socklen_t addrlen)
```

- ❑ **socket** socket attraverso cui inviare il datagram
- ❑ **data** buffer contenente il datagram da inviare
- ❑ **datalen** lunghezza datagram da inviare
- ❑ **flags** eventuali opzioni
- ❑ **addr** puntatore all'indirizzo del destinatario
- ❑ **addrlen** lunghezza dell'indirizzo
- ❑ Ritorna: numero di bytes spediti (NB: non bufferizza)

Ricezione di datagram

```
int recvfrom (int socket, void *buffer,  
              size_t datalen, int flags,  
              struct sockaddr *from, int *fromlen)
```

- ❑ **socket** socket attraverso cui ricevere il datagram
- ❑ **buffer** buffer in cui verrà depositato il datagram ricevuto
- ❑ **datalen** lunghezza del buffer
- ❑ **flags** eventuali opzioni
- ❑ **from** puntatore indirizzo del mittente
- ❑ **fromlen** puntatore alla lunghezza dell'indirizzo del mittente
- ❑ Ritorna: numero di bytes letti (\leq datalen)

Altre primitive per socket datagram

□ **bind**

- ❖ si deve usare, sul lato server, per associare un end point al socket. Sul client: solo se necessario usare una specifica porta di uscita, sempre che sia libera

□ **connect**

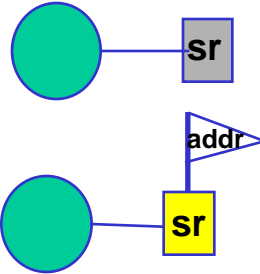
- ❖ si può usare per associare un end point remoto fisso al socket. NON è una connessione: non esiste per DGRAM

□ **send, recv, write, read**

- ❖ si possono usare se si è eseguito connect
- ❖ il destinatario/mittente è implicito, deciso con la connect
- ❖ bisogna comunque leggere/scrivere a **messaggi interi**

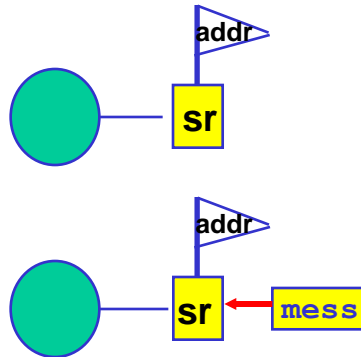
SERVER

```
sr=socket(PF_INET,SOCK_DGRAM,
          IPPROTO_UDP)
```



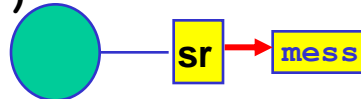
```
bind(sr,.....)
```

```
recvfrom(sr,buf,len,&caddr,
         &caddrlen)
```



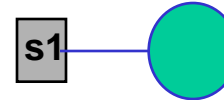
attesa

```
sendto(s1,buffer,length,&caddr,
       sizeof(caddr))
```



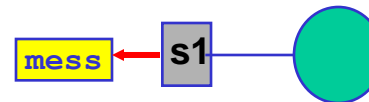
CLIENT

```
s1=socket(PF_INET,SOCK_DGRAM,
          IPPROTO_UDP)
```

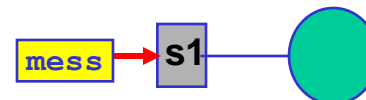


```
connect(s1,&saddr,sizeof(saddr))
```

```
send(s1,buffer,length)
sendto(s1,buffer,length,&saddr,
       sizeof(saddr))
```



```
recv(s1,buffer1,length1)
recvfrom(s1,buffer1,length1,
         &saddr,&saddrlen)
```



attesa

Esempio di client e server UDP

□ Applicazione «Echo»

- ❖ Il client legge una porzione di dati, la invia al server in un unico pacchetto UDP
- ❖ Il server reinvia indietro in forma identica quanto ricevuto in un unico pacchetto UDP
- ❖ Il client resta in attesa di una risposta dal server. Quando ricevuta, a fronte di uno specifico input (per esempio la stringa speciale «stop»), termina

Read e write

- ❑ Sulle connessioni (SOCK_STREAM) è possibile anche utilizzare le primitive di I/O **read** e **write**, perdendo la possibilità di specificare flags

```
char *str="Hello world";  
SOCKET s;
```

```
.....
```

```
write (s,str,strlen(str));
```

Dipende dal protocollo
definito a livello applicativo



- ❑ Si può usare la read/recv anche per i datagram (SOCK_DGRAM), ma se il datagram in arrivo è più lungo del buffer, **i byte in eccesso vengono scartati**

Protocolli applicativi

- ❑ In molti casi le specifiche del sistema "client/server" non includono le modalità di scambio dei dati («protocollo»)
 - ❖ È compito dello sviluppatore scegliere/progettare un protocollo applicativo adatto allo scopo
 - ❖ Usare considerazioni di «buon senso»
- ❑ Protocollo: bilanciamento tra
 - ❖ Complessità
 - ❖ Efficienza
- ❖ Ognuno di questi fattori è influenzato dalle proprietà del protocollo scelto

Protocollo con o senza stati

- ❑ In un protocollo **senza stati** per ogni connessione logica avviene un solo scambio **richiesta/risposta**
- ❑ Un protocollo **con stati** consente invece lo scambio di più coppie **richiesta/risposta**
 - ❖ Nell'ambito della connessione logica, le successive risposte considerano le richieste precedenti (concetto di sessione)
 - ❖ Si deve gestire il caso di terminazione inaspettata della connessione logica
- ❑ NB: Si possono gestire sessioni anche con protocolli privi di stati (es. HTTP)
 - ❖ La sessione deve essere identificata in qualche modo esplicito ed il client deve citare l'identificativo in ogni richiesta successiva

Codifica sul canale di comunicazione

- ❑ Un protocollo cosiddetto «testuale» invia e riceve solo messaggi costituiti da sequenze di caratteri
 - ❖ Più facile da gestire e da debuggare/correggere
 - ❖ Esistono molte codifiche alternative dei caratteri (ASCII, UTF-8, UTF-16, ...): mittente e destinatario devono essere d'accordo sulla codifica adottata
 - ❖ Peraltro esistono modi per codificare dati binari arbitrari sotto forma di sequenze di caratteri stampabili, per es. la codifica base64 (utilizzata nelle emails per gli allegati)
- ❑ Un protocollo «binario» prevede lo scambio di dati arbitrari
 - ❖ Sistemi di elaborazione differenti possono rappresentare i dati elementari in modo differente (low-endian/big-endian), occorre porre attenzione
 - ❖ Debuggare una comunicazione binaria può essere complicato

Codifica dei singoli messaggi

- ❑ Il ricevitore deve essere in grado di segmentare i messaggi ricevuti
 - ❖ L'interfaccia «socket» TCP consente solo di trasferire flussi di bytes, il cui contenuto (struttura) è ignoto
 - ❑ Struttura generica di un messaggio:
 - ❖ **Intestazione:** meta-informazioni riguardanti il messaggio (dimensione del messaggio, versione del protocollo, codifica adottata nel corpo, ...)
 - ❖ **Corpo:** il messaggio vero e proprio
 - ❑ La codifica dell'intestazione deve essere univoca!
 - ❖ Ha una lunghezza fissa (per es. intero rappresentato su 4 bytes in un certo formato)
- oppure
- ❖ C'è una sequenza di caratteri univoca (es.: CR/LF o CR solo) che consente di delimitarla (es. echo su TCP, per ogni «linea»)

Socket full-duplex

- ❑ I socket supportano un canale full-duplex
 - ❖ In altre parole, mittente e destinatario possono inviare dati quando lo ritengono opportuno, senza attendere quelli dell'altra parte
- ❑ Spesso ciò non è necessario a livello applicativo
 - ❖ Per esempio, il pattern «richiesta del client» / «risposta del server» non lo richiede
 - ❖ Se è assolutamente necessario, occorre gestire la concorrenza tramite appositi meccanismi (processi, thread, select, ...)







Protocolli: consigli pratici

- ❑ Usare codifiche testuali quando possibile
- ❑ Progettare protocolli privi di stato
- ❑ Usare una comunicazione a livello applicativo di tipo «half-duplex»
- ❑ Essere tolleranti in ciò che si riceve (lato ricevitore), precisi in ciò che si trasmette
 - ❖ Prevedere e gestire opportunamente gli errori di comunicazione (violazioni di protocollo, ecc.)

Il problema del blocking

- ❑ Un processo bloccato su una primitiva (p. es. **recv** o **accept**) non è in grado di reagire al verificarsi di altri eventi
- ❑ In alcuni casi è necessario un meccanismo per **evitare il blocking indefinito**
 - ❖ p. es. se non so da quale socket arriverà il prossimo input
- ❑ Possibili soluzioni
 - ❖ Usare primitive per l'I/O multiplexing (**select** e **poll**)
 - ❖ Cambiare l'I/O blocking mode dei socket in **nonblocking I/O** tramite `fcntl`, `EWOULDBLOCK` error viene ritornato se l'operazione non può completarsi senza bloccarsi
 - ❖ Usare **signal-driven I/O** o **asynchronous I/O**

Confronto modalità I/O (read)

	Blocking	Non-blocking	Multiplexing	Signal-driven	Asynchronous
Attesa dati	Inizio 	Try-fail Try-fail Try-fail Try-fail Try-fail Try-fail Try	Inizio attesa  Pronto	Settaggio signal handler Notifica	Settaggio signal handler Inizio
Copia dati da spazio kernel	 Fine	 Fine	Inizio  Fine	Inizio  Fine	Notifica

I/O multiplexing: select

- ❑ Nell'interfaccia dei socket un meccanismo per l'I/O multiplexing è fornito dalla primitiva **select**, che consente di:
 - ❖ attendere il verificarsi di un qualsiasi evento (p. es. arrivo di dati) tra quelli cui si è interessati
 - ❖ **attendere al massimo per un certo tempo**
- ❑ Quando la **select** è chiamata, il processo chiamante si blocca finquando almeno uno degli eventi dichiarati ha luogo oppure il tempo di timeout è trascorso
 - ❖ Dichiarando più eventi, il singolo processo può fare I/O multiplexing

select: eventi controllabili

Un evento può corrispondere al verificarsi di

- ❑ una condizione che garantisce di poter eseguire una determinata operazione **senza bloccarsi**:
 - ❖ La condizione di **leggibilità** di un socket è l'OR delle seguenti:
 - vi è almeno una richiesta di connessione pendente (socket passivi)
 - sono disponibili dati nel buffer di ricezione (socket connessi)
 - C'è un datagram ricevuto nel buffer di ricezione (socket datagram)
 - si è verificato errore sul socket - es. chiusura connessione
 - ❖ La condizione di **scrivibilità** di un socket è l'OR delle seguenti:
 - il socket è connesso e il buffer di trasmissione del socket non è pieno
 - si è verificato un errore sul socket - es. chiusura connessione
 - ❖ Si è verificata **un'eccezione**: presenza di dati urgenti

select: sintassi

```
int select (
```

```
    int nfd,    Massimo+1 tra i valori dei file descriptors nei 3 insiemi fd_set
```

```
    fd_set *readfd,
```

```
    fd_set *writefd,
```

```
    fd_set *exceptfd,
```

Insiemi di socket di cui interessa il verificarsi di condizioni di leggibilità, scrivibilità e di eccezioni.

Possono essere NULL se non interessano

```
    struct timeval *timeout
```

Tempo massimo di bloccaggio (se NULL il tempo è infinito)

```
)
```

select: funzionamento

- ❑ Se nessuna delle condizioni specificate è verificata, la select blocca
- ❑ Non appena almeno una delle condizioni si verifica, la select si sblocca e:
 - ❖ rimpiazza i 3 insiemi di socket da selezionare con i corrispondenti sottoinsiemi di socket sui quali si sono verificate le condizioni richieste
 - ❖ restituisce il numero complessivo di socket selezionati
- ❑ Se nessun socket è diventato selezionabile entro il tempo massimo specificato dal parametro timeout, la select si sblocca e restituisce il valore 0.

select: macro per operare sugli insiemi di socket

- ❑ **FD_SET(int s, fd_set *fd)**
(Inserisce il socket s nell'insieme puntato da fd)
- ❑ **FD_CLR(int s, fd_set *fd)**
(Elimina il socket s dall'insieme puntato da fd)
- ❑ **FD_ZERO(fd_set *fd)**
(Azzera l'insieme puntato da fd)
- ❑ **int FD_ISSET(int s, fd_set *fd)**
(Restituisce valore TRUE se il socket s è
settato nell'insieme puntato da fd)

Esempio con due sockets

```
struct timeval tval;
fd_set cset;
FD_ZERO(&cset); FD_SET(s1, &cset); FD_SET(s2, &cset);
int t = 15; tval.tv_sec = t; tval.tv_usec = 0;
if ((n = select(FD_SETSIZE, &cset, NULL, NULL, &tval)) == -1)
    err_fatal("select() failed");
if (n>0) {
    if (FD_ISSET(s1, &cset))
        s = s1;
    else
        s = s2;
    /* NB: Uso if/else perche' voglio gestire comunque un solo socket anche se
       potrebbero essersene sbloccati due */
    fromlen = sizeof(struct sockaddr_in);
    n=recvfrom(s,buf,BUFLLEN,0,(struct sockaddr*)&from,&fromlen); /* o recv */
    if (n != -1)
        printf("Received message from socket %d\n", s);
    else
        printf("Error in receiving response\n");
} else {
    printf("No response after %d seconds\n",t); /* tval is reset by select */
}
```

Altro esempio: Echo client UDP con timeout in caso di
mancanza di risposta dal server

Altre API socket: stato e opzioni

- ❑ `int getsockname (int s, struct sockaddr *
addr, socklen_t * addrlen)`
(Scrive in `addr` l'indirizzo associato al socket `s`)
- ❑ `int getpeername (int s, struct sockaddr *
addr, socklen_t * addrlen)`
(Scrive in `addr` l'indirizzo del socket cui `s` è connesso)
- ❑ `int getsockopt (int s, int level, int
opt_name, void *opt_val, socklen_t *val_len)`
(Scrive in `opt_val` il valore dell'opzione specificata dalla coppia
(`opt_name`, `level`))
- ❑ `int setsockopt (int s, int level, int
opt_name, void *opt_val, socklen_t *val_len)`
(imposta l'opzione del socket `s` specificata dalla coppia (`opt_name`, `level`))

Localizzare il server

- ❑ È una delle prime operazioni da fare
- ❑ In Internet, il server è individuato da
 - ❖ indirizzo IP o nome DNS
 - ❖ n. porta
 - ❖ protocollo

Informazioni sul server

- ❑ Diversi modi di specificare gli host:
 - ❖ mail.polito.it
 - ❖ 130.192.3.44
- ❑ Diversi modi di specificare le porte:
 - ❖ mail.polito.it smtp
 - ❖ mail.polito.it:smtp
 - ❖ mail.polito.it 25
- ❑ Diversi modi di specificare i protocolli:
 - ❖ http://www.polito.it
 - ❖ mailto:pippo@polito.it
- ❑ I «resolver» possono mappare da nomi a numeri e viceversa

Cercare un nome di host

- ❑ Se il server è dato mediante un nome, bisogna ottenere la traduzione in indirizzo (usando la configurazione locale e/o il DNS)
- ❑ Si può usare **gethostbyname** (query per A record) che usa

```
struct hostent{  
    char *h_name;           /*nome canonico*/  
    char **h_aliases;       /*elenco alias*/  
    int  h_addrtype;        /*tipo indir. */  
    int  h_length;          /*lunghezza indir.*/  
    char **h_addr_list;     /*lista linkata indir.,  
                           ultimo elem. = NULL*/  
}
```

- ❑ Risoluzione inversa tramite **gethostbyaddr**: query del PTR record nel dominio in-addr.arpa

Cercare una well-known port

- Si usa la struttura seguente con **getservbyname**

```
#include <netdb.h>
struct servent {
    char *sname;           /* nome ufficiale*/
    char **saliases;       /* elenco alias */
    int  sport;            /* num. porta */
    char *sproto;          /* nome protoc. */
}
```

Risoluzione IPv4 e IPv6, host e service

La **getaddrinfo** è una primitiva che fornisce tutte le funzioni di ricerca per un server ed una porta

```
int getaddrinfo(char * host, char* service,  
struct addrinfo *hints, struct addrinfo **res)
```

host	stringa con il nome o numero IPv4 o 6 dell'host
service	stringa con il nome standard del protocollo o numero di porta
hints	NULL o punt. a struttura con dati desiderati
res	punt. (passato by reference) alla lista di dati in uscita

Valore di ritorno: 0 successo, altrimenti c'è stato un errore

addrinfo structure

```
struct addrinfo {  
    int ai_flags;      /* input flags */  
    int ai_family;     /* protocol family for socket */  
    int ai_socktype;   /* socket type */  
    int ai_protocol;   /* protocol for socket */  
    socklen_t ai_addrlen; /* length of ai_addr.*/  
    struct sockaddr *ai_addr; /*address*/  
    char *ai_canonname;    /* canonical name */  
    struct addrinfo *ai_next; /* next in list */  
};
```

In rosso i campi che possono essere settati nell'hints
In nero quelli settati solo nel result

Esempio

```
SOCKET connectedTCP (char *host, char *serv) {
    struct addrinfo hints, *res, *res0;
    int error, s; char *cause;
    memset(&hints, 0, sizeof(hints));
    hints.ai_family = PF_INET; /* mi va bene solo IPv4 */
    hints.ai_socktype = SOCK_STREAM;
    if (error = getaddrinfo(host, serv, &hints, &res0))
        {   errx(1, "%s", "getaddrinfo"); /*NOTREACHED*/   }
    s = -1;
    for (res = res0; res!=NULL; res = res->ai_next) {
        s = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
        if (s < 0) {cause = "socket"; continue;}
        if (connect(s, res->ai_addr, res->ai_addrlen) < 0) {
            cause = "connect"; close(s);
            s = -1;
            continue;
        }
        break; /* okay we got one */
    }
    freeaddrinfo(res0); /* free list of structures */
    if (s < 0) {errx(1, "%s", cause); /*NOTREACHED*/ }
    return s;
}
```

Indipendenza IPv4 e IPv6

- ❑ L'API socket ha delle dipendenze dalla famiglia di indirizzi (AF) nelle proprie librerie
- ❑ Usando la risoluzione dei nomi si può mascherare:
 - ❖ Tramite nomi mappati su indirizzi: la `getaddrinfo` usando una generica famiglia (`AF_UNSPEC`) ritorna indirizzi sia IPv4 sia IPv6
 - ❖ La struttura generica `sockaddr_storage` viene usata per contenere gli indirizzi
 - è adatta sia a indirizzi IPv4 sia IPv6
 - sempre necessario cast a `sockaddr` quando usata nelle chiamate di funzione)

Il caso IPv6

- ❑ IPv6 di solito non è mai usato senza IPv4 («dual stack»)
- ❑ Il dual stack converte automaticamente gli indirizzi IPv4 in IPv4-mapped IPv6 addresses
 - ❖ Per esempio: un client IPv6 deve risolvere il nome di un server IPv4: il resolver ritorna l'indirizzo IPv4-mapped IPv6
- ❑ Si può provare a rendere il codice valido per entrambi i protocolli (esempi su libro Stevens)

IPv6: nuova sockaddr

IPv6

```
struct sockaddr_in6 {  
    uint8_t sin6_len; /* length*/  
    uint8_t sin6_family; /* AF_INET6 */  
    uint16_t sin6_port; /* port number */  
    uint32_t sin6_flowinfo; /* IP6 flow information */  
    struct in6_addr sin6_addr; /* IPv6 address*/  
    uint32_t sin6_scope_id; /* scope zone index*/  
}
```

Programmazione dei server

- ❑ Servizi TCP vs UDP
- ❑ Servizi stateful vs stateless
- ❑ Server sequenziali vs concorrenti

Problemi dei server

□ Usare TCP o UDP?

❖ Vantaggi TCP

- Trattamento errori (ritrasmissione automatica)
- Consegna in ordine e senza duplicati
- Frammentazione e ri-assemblaggio messaggi
- Rilevamento di periodi eccessivi di inattività (opzionale)

❖ Svantaggi TCP

- Apertura/chiusura delle connessioni fa perdere tempo
- Si usa un socket per connessione, che **deve essere liberato alla fine** (potenziale leakage)

Differenze fra i clienti UDP e TCP

□ Legate alla inaffidabilità dell'UDP

- ❖ Bisogna prevedere meccanismi di ritrasmissione
- ❖ Bisogna prevedere meccanismi per rimettere in sequenza i messaggi arrivati (as es. ID nella richiesta che deve essere replicato nella risposta)
- ❖ Bisogna essere in grado di rivelare duplicati
- ❖ Non si è sicuri che il server sia ancora lì (crash)

Server stateful o stateless?

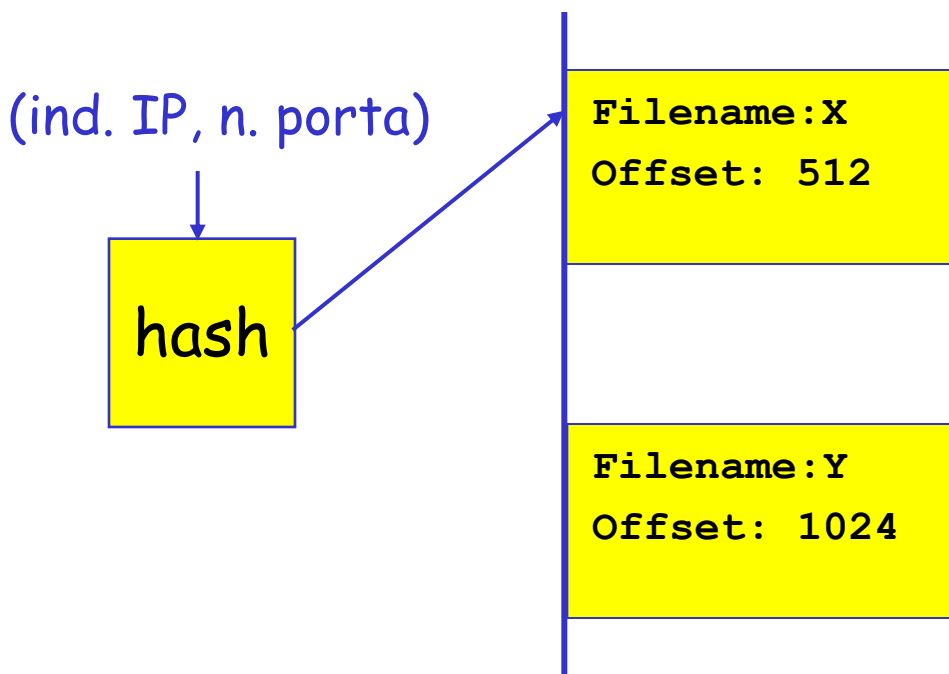
- ❑ **Stateful:** il server conserva informazioni di **stato** relativo ai clienti
 - ❖ Esempio: server FTP
- ❑ **Stateless:** il server **dimentica** le richieste precedenti, dopo la risposta
 - ❖ Esempio: server HTTP (versione originale)
- ❑ In generale, meglio preferire servizi stateless quando possibile
 - ❖ Più semplici da programmare
 - ❖ Nessun problema di memory leakage per lo stato dell'applicazione nei confronti del client

Esempio stateless

- ❑ Server che legge dati da un file richiesti dal cliente
- ❑ Parametri: nome file, offset, lunghezza
- ❑ Per ogni richiesta, bisogna:
 1. Aprire il file
 2. Posizionarsi sul byte opportuno
 3. Leggere i dati
 4. Chiudere il file

Esempio stateful

- Tenere conto delle letture precedenti, es. tramite una tabella di hash che mi dice se il file è già aperto e a quale offset

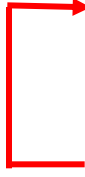


Esempio stateful: problemi

- ❑ Se il cliente va in crash e riparte, occupa una nuova entry (cambia la porta), e l'altra è inutilizzata (**leakage**)
- ❑ Se si usa cancellazione con strategia LRU, crash/ripartenze frequenti dei client possono cancellare entries valide ma usate poco frequentemente
- ❑ Possibile soluzione: **soft state (non permanente)**
 - ❖ Cancellare le entries che sono inattive da troppo tempo
 - ❖ Con crash/ripartenze frequenti, si potrebbe avere riempimento della tabella


Server iterativi o concorrenti?

❑ Iterativi (richieste processate sequenzialmente, FIFO)

- 
1. Estrae una richiesta dalla coda
 2. Effettua il servizio
 3. Invia una risposta

❑ Concorrenti


❖ Padre

- 
1. Estrae una richiesta dalla coda
 2. Trova/crea un task o thread e gli affida la richiesta

❖ Figlio

1. Effettua il servizio
2. Invia una risposta
3. Fine

Server iterativo su TCP

1. Creare un socket e fare **bind** sulla porta di interesse
 2. Mettere il socket in stato passivo (mettersi in attesa di richieste)
 3. Accettare una nuova richiesta di connessione
 4. Leggere una richiesta, fornire una risposta (finché il cliente smette)
 5. Chiudere il socket (connessione)
- 

Problemi server iterativo

- ❑ Ogni richiesta deve aspettare che tutte le precedenti siano state servite
- ❑ Si attende anche quando la CPU non è occupata!
- ❑ Se il numero di richieste è elevato (tempo tra le richieste comparabile con tempo di servizio della richiesta) è probabile che il server sia occupato quando la richiesta arriva
 - ❖ Le richieste saranno scartate o il client va in timeout
- ❑ Soluzione: server concorrenti

Server concorrenti

- ❑ Serve più richieste in modo concorrente
 - ❖ Quando una richiesta arriva il suo servizio viene attivato e procede parallelamente agli altri
 - Il processo finisce quando è generata la risposta
 - ❖ La probabilità che la richiesta arrivi quando il server non la può servire è ridotta
- ❑ Differenti modi di implementazione
 - ❖ Assegnare ogni richiesta ad un processo differente
 - ❖ Assegnare ogni richiesta ad un thread differente
 - ❖ Simulare la concorrenza all'interno di un singolo processo/thread (complesso)
- ❑ L'uso di processi/thread rende la soluzione non facilmente portabile tra S.O. diversi

Algoritmo per server su TCP

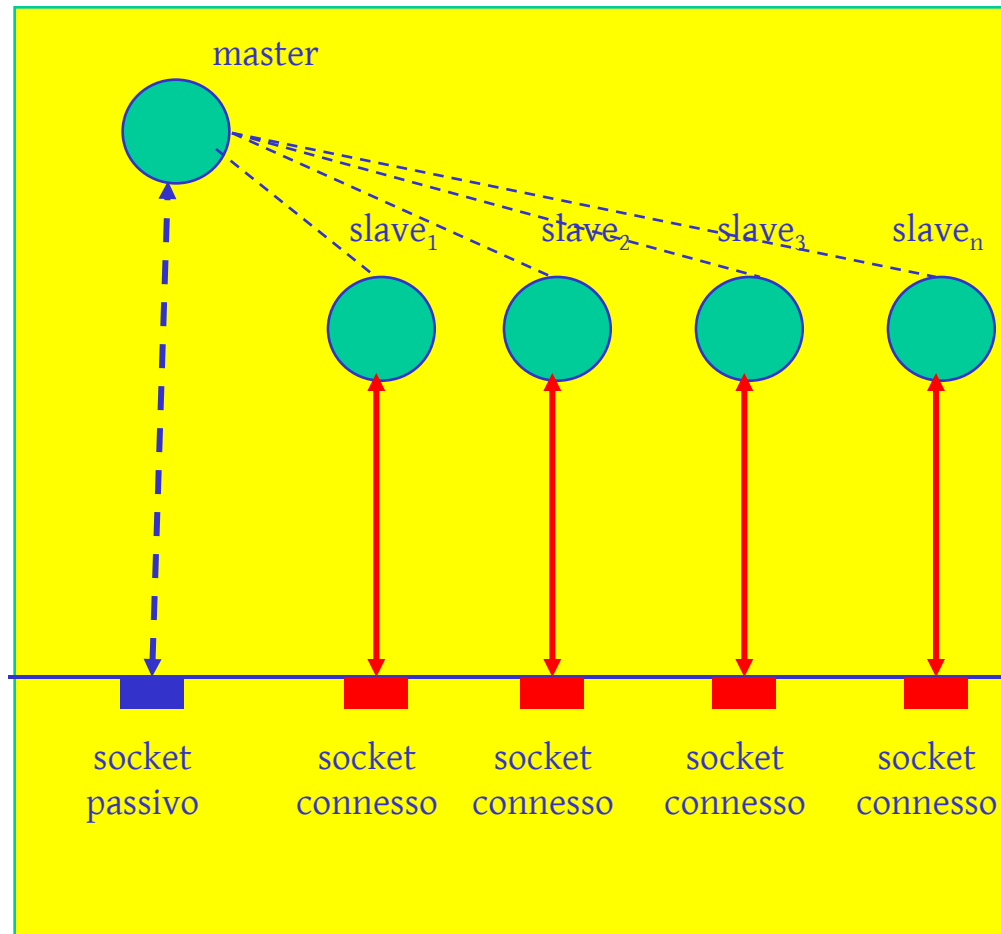
Padre (master)

1. Crea il socket
2. **bind** sulla porta di ascolto
3. **listen** per mettere il socket in modo passivo
4. **accept** per aprire una connessione
5. Crea un figlio per gestire le richieste e passa copia del socket

Figlio (slave)

1. Riceve il socket connesso
2. Interagisce con il cliente secondo il protocollo
3. Chiude il socket connesso
4. Termina il processo

Server TCP concorrente



Processi padre e figlio

- ❑ Possono condividere lo stesso programma
 - ❖ Ad un certo punto si chiama **fork**
 - ❖ Il codice per padre e figlio si possono distinguere tramite il risultato: diverso PID
- ❑ Possono essere anche programmi eseguibili differenti
 - ❖ Nel programma del padre si chiama **fork** seguita da **execve**

Es.: Server Concorrente (UNIX)


```
/* creazione socket s, bind, listen */
...
for (;;)
{
    addrlen = sizeof(struct sockaddr_in);
    new = accept(s, (struct sockaddr *) &c_addr, &addrlen);
    if (new == INVALID_SOCKET)
        err_fatal("accept() failed");

    if((childpid=fork())<0) err_fatal("fork() failed");
    else if (childpid > 0)
    {
        /* processo padre */
        close(new);          /* chiudo nuovo socket */
    }
    else
    {
        /* processo figlio */
        close(s);            /* chiudo socket del padre */
        service(new);        /* servo il client */
        exit();              /* dimenticarsi e' errore comune:
                               si farebbe accept con socket chiuso! */
    }
}
```

Alternativa con 1 solo processo

- ❑ Si possono servire più richieste in modo concorrente con 1 solo processo
- ❑ Perché?
 - ❖ Creare/distruggere processi sono operazioni complesse
 - ❖ Se il tempo di servizio è breve, una gran parte dell'esecuzione è assorbita da queste operazioni
 - ❖ Siamo interessati alla concorrenza del servizio, non a quella dei processi servitori
 - ❖ La frequenza delle richieste è bassa, riesco a servirle tutte con un solo processo

Algoritmo server singolo processo

1. Creare un socket ed associarlo ad una porta di ascolto
 2. Porre il socket in modo passivo
 3. Aggiungerlo ad una lista di socket esistenti
 4. Ricevere una richiesta da uno dei socket (**select** per attendere su più socket)
 5. Se è una nuova connessione aggiungere il socket a quelli da cui è possibile I/O, se è un messaggio dalla connessione attiva, eseguire il servizio
- 

Server con pre-allocazione

❑ Problemi delle versioni concorrenti precedenti:

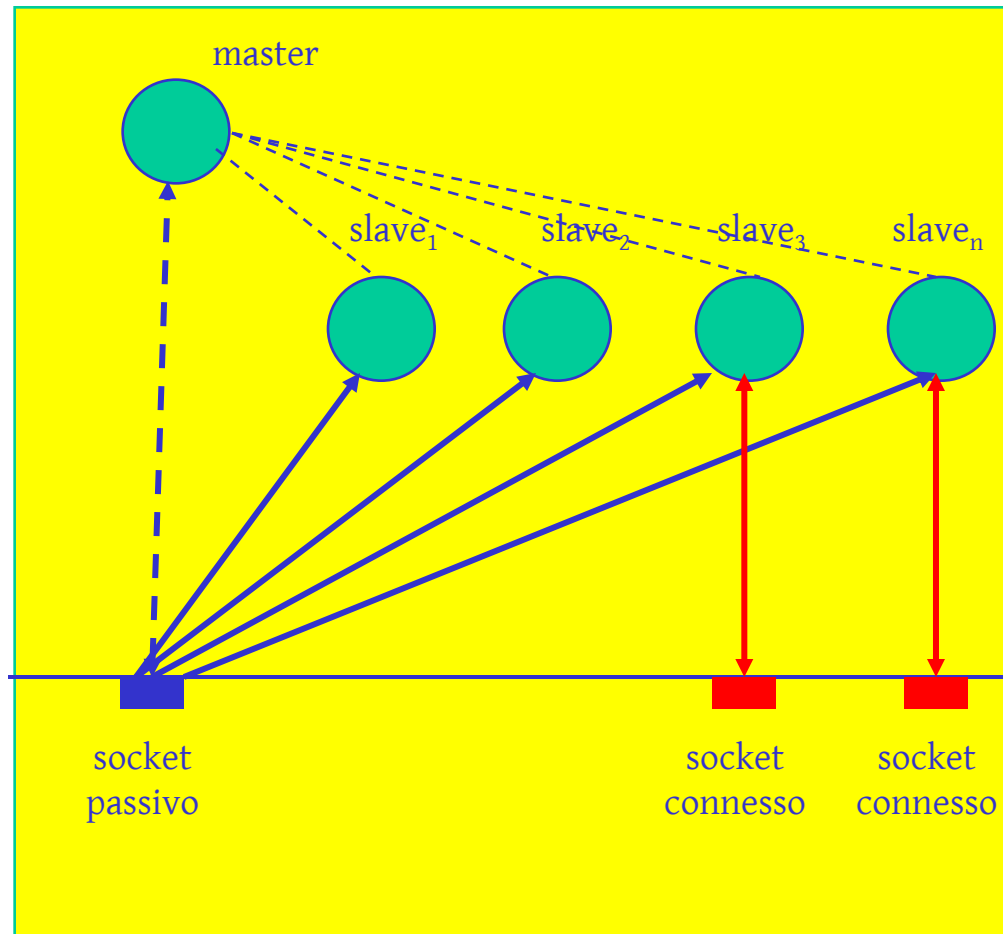
- ❖ Il tempo per creare/distruocere un processo può essere lungo
- ❖ Il numero di processi non è controllato (n. processi = n. richieste in esecuzione)
- ❖ Possibilità di un numero troppo elevato di processi concorrenti

❑ Possibile soluzione: pre-allocazione degli slave (process pooling)

Esempio con TCP

1. Il processo master crea un socket passivo
2. Il master crea un certo numero di slave
3. Ogni slave ottiene una **copia del socket passivo**
4. Ogni slave esegue una **accept** sul socket (in **mutua esclusione** , gestita dal S.O.)

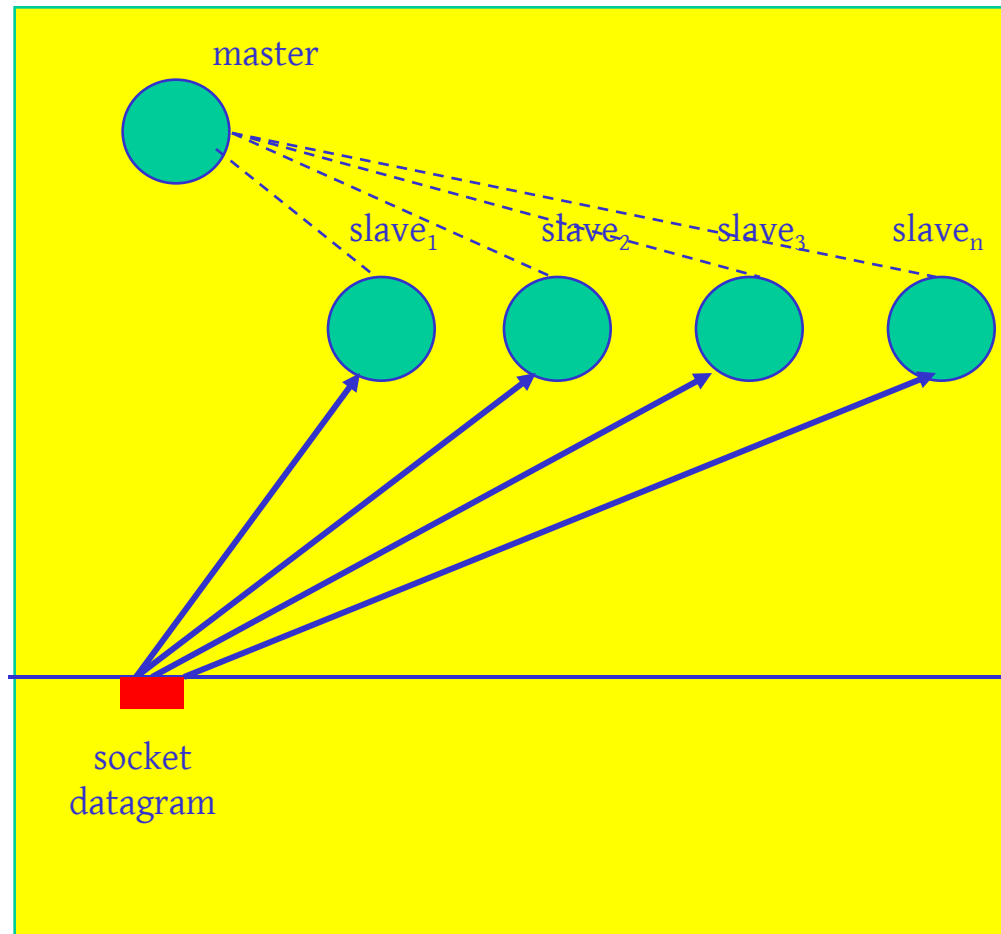
Server con pre-allocazione (TCP)



Esempio con UDP

- ❑ Il master crea il socket datagram
- ❑ Il master crea gli slave
- ❑ Ogni slave si mette in attesa di datagram (**recvfrom**) sullo stesso socket (**in mutua esclusione, gestita dal S.O.**)
- ❑ Lo slave che riceve il datagram lo serve
- ❑ A fine servizio si torna in attesa sul socket

Server con pre-allocazione (UDP)



Altre caratteristiche

- ❑ Il master può :
 - ❖ Funzionare solo per l'inizializzazione e poi terminare
 - ❖ Può diventare lo slave (n+1)-simo
- ❑ In caso di congestione, le richieste restano in coda
- ❑ Se vi sono troppe richieste possono andare perdute (overflow del buffer)

Altri linguaggi

- ❑ Java: l'interfaccia è basata sui socket
- ❑ Maschera le primitive con classi e metodi di più alto livello:
 - ❖ package java.net
 - class Socket
 - Realizza un socket di tipo STREAM connesso
 - class ServerSocket
 - Realizza un socket di tipo STREAM passivo
 - class DatagramSocket
 - Realizza un socket di tipo DATAGRAM