

## Programmazione in rete: i socket

**Antonio Lioy**  
< [lioy@polito.it](mailto:lioy@polito.it) >

**Politecnico di Torino**  
**Dip. Automatica e Informatica**

### Avviso ai programmatori

- **la programmazione di rete è pericolosamente vicina al kernel del S.O. e quindi:**
  - può facilmente bloccare il S.O.
    - sforzarsi di controllare l'esito di tutte le operazioni, senza dare niente per scontato
  - le API possono variare in dettagli minimi ma importanti
    - sforzarsi di prevedere tutti i casi per creare programmi "portabili"
    - cercheremo di usare Posix 1.g



## Esercizio - copia dati

- copiare il contenuto del file F1 (primo parametro sulla riga di comando) nel file F2 (secondo parametro sulla linea di comando)

copyfile.c

## Messaggi di errore

- **devono contenere almeno:**
  - [ PROG ] nome del programma
  - [ LEVEL ] livello di errore (info, warning, error, bug)
  - [ TEXT ] segnalazione di errore nel modo più specifico possibile (es. nome di file e riga di input su cui si è verificato il problema)
  - [ ERRNO ] numero e/o nome dell'errore di sistema (se applicabile)
- **forma suggerita:**

( PROG ) LEVEL - TEXT : ERRNO

## Funzioni di errore

- conviene definire funzioni di errore standard che accettino:
  - una stringa di formato per l'errore
  - una lista di parametri da stampare
- UNP, appendice D.4

	errno?	terminazione?	log level
err_msg	no	no	LOG_INFO
err_quit	no	exit(1)	LOG_ERR
err_ret	si	no	LOG_INFO
err_sys	si	exit(1)	LOG_ERR
err_dump	si	abort( )	LOG_ERR

errlib.h

errlib.c

## stdarg.h

- elenco variabile di argomenti (ANSI C)
- dichiarato con ellissi (. . .) come ultimo argomento di una funzione
- argomenti usabili tutti insieme (ap) in apposite funzioni (vprintf, vfprintf, vsprintf, vasprintf, vsnprintf) oppure uno alla volta (va\_arg), ma in questo caso bisogna sapere in altro modo quanti sono

```
#include <stdarg.h>

void va_start (va_list ap, RIGHTMOST);
TYPE va_arg (va_list ap, TYPE);
void va_end (va_list ap);
```

## Esempio uso stdarg.h

- creo una funzione `my_printf`
- ... che si comporti come la `printf` (e quindi accetti un numero variabile di parametri)
- ... ma premetta ad ogni stampa `"(MY_PRINTF)"`

`va_test.c`

## Da 32 a 64 bit: problemi

- il passaggio dalle architetture a 32 bit a quelle a 64 bit ha modificato la dimensione dei dati
- in particolare non si può più assumere che `| int | = | puntatore |`
- attenzione quindi ad usare correttamente i tipi definiti per superare questi problemi (es. `size_t`)

	ILP32	LP64
<code>char</code>	8	8
<code>short</code>	16	16
<code>int</code>	32	32
<code>long</code>	32	64
<code>puntatore</code>	32	64

`datasize.c`

## Scambio dati tra nodi eterogenei

### ■ problema:

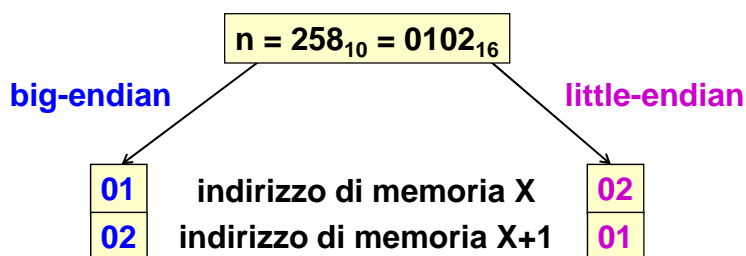
- quando si scambiano dati complessi (ossia non singoli caratteri ASCII) non si può essere certi che siano codificati allo stesso modo sui diversi nodi
- codifica dei dati dipende da HW + S.O.

### ■ soluzione:

- usare un formato neutro (di rete)
- a volte la conversione è fatta automaticamente dalle funzioni ...
- ... ma spesso è compito esplicito del programmatore

## Sorgenti di incompatibilità dei dati

- formati floating-point diversi (IEEE-754 / non-IEEE)
- allineamento dei campi di una struct sui confini delle word
- ordine dei byte negli interi (little-endian o big-endian)



byteorder.c

## Funzioni "host to network"

- per il passaggio dei parametri alle funzioni di rete
- non pensate per il passaggio dei dati applicativi

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

uint32_t htonl (uint32_t hostlong);
uint16_t htons (uint16_t hostshort);
uint32_t ntohl (uint32_t netlong);
uint16_t ntohs (uint16_t netshort);
```

## Note sui tipi interi

- talvolta si trovano ancora i vecchi tipi:
  - u\_long (= uint32\_t)
  - u\_short (= uint16\_t)
- in vecchie versioni di cygwin erano definiti:
  - u\_int32\_t (= uint32\_t)
  - u\_int16\_t (= uint16\_t)
- consiglio:
  - scrivere i programmi coi tipi uint32\_t e uint16\_t
  - se necessario, mapparli con una #define condizionata

## Output di tipi interi "speciali"

- **ISO C99 introduce modificatori per:**
  - evitare conversioni / cast non necessari
  - fare output nativo di tipi interi "speciali"
- **nuovi modificatori per tipi interi:**
  - "j" per indicare conversione di intNN\_t / uintNN\_t
  - "z" per indicare conversione di size\_t
- **il supporto di questi nuovi modificatori dipende dalla libc installata insieme al compilatore C**

## Indirizzi di rete

- le reti IPv4 usano direttamente gli indirizzi su 32 bit
- ... non i nomi (es. www.polito.it), che sono tradotti in indirizzi dal DNS
- ... e neanche gli indirizzi puntati (es. 130.192.11.51)
- ... ma il loro valore numerico su 32 bit
  
- **esempio: 130.192.11.51**
  - =  $130 \cdot 2^{24} + 192 \cdot 2^{16} + 11 \cdot 2^8 + 51$
  - =  $((130 \cdot 256) + 192) \cdot 256 + 11 \cdot 256 + 51$
  - =  $130 \ll 24 + 192 \ll 16 + 11 \ll 8 + 51$
  - = 2,193,623,859

## Conversione di indirizzi di rete

- per generalità, le funzioni standard richiedono che l'indirizzo numerico sia espresso in forma di struct
- per convertire indirizzi numerici da/a stringhe:
  - [IPv4] `inet_ntoa( )` e `inet_aton( )`
  - [IPv4 / v6] `inet_ntop( )` e `inet_pton( )`
- l'uso di altre funzioni (es. `inet_addr`) è deprecato

```
#include <arpa/inet.h>

struct in_addr { in_addr_t s_addr };

struct in6_addr { uint8_t s6_addr[16]; };
```

## inet\_aton ( )

- converte un indirizzo IPv4 ...
- ... da stringa in notazione puntata ("dotted notation")
- ... a forma numerica (di rete)
- restituisce 0 in caso di indirizzo non valido
- la stringa può essere composta da numeri decimali (default), ottali (inizio con 0) o esadecimali (inizio con 0x) quindi 226.000.000.037 è uguale a 226.0.0.31

```
#include <arpa/inet.h>

int inet_aton (
    const char *strptr,
    struct in_addr *addrptr
);
```



## inet\_ntoa ( )

- converte un indirizzo IPv4 ...
- ... da forma numerica (di rete)
- ... a stringa in notazione puntata (“dotted notation”)
- restituisce il puntatore all’indirizzo in forma di stringa oppure NULL in caso di indirizzo non valido
- attenzione! il puntatore restituito punta ad un'area di memoria statica interna alla funzione

```
#include <arpa/inet.h>

char *inet_ntoa (
    struct in_addr addr
);
```

## Esempio: validazione di un indirizzo

Scrivere un programma che:

- accetti sulla riga di comando un indirizzo IPv4 in notazione puntata
- restituisca il suo valore numerico (di rete)
- segnali errore in caso di formato errato o indirizzo illegale

Verifica:

- si ricordi che  $A.B.C.D = A \ll 24 + B \ll 16 + C \ll 8 + D$

avrfy.c

avrfy2.c

## inet\_pton()

- converte un indirizzo IPv4 / IPv6 ...
- ... da stringa in notazione puntata (“dotted notation”)
- ... a forma numerica (di rete)
- restituisce 0 in caso di indirizzo non valido, -1 se la address family è sconosciuta
- address family = AF\_INET o AF\_INET6

```
#include <arpa/inet.h>

int inet_pton (
    int family,
    const char *strptr,
    void *addrptr
);
```

## inet\_ntop()

- converte un indirizzo IPv4 / IPv6 ...
- ... da forma numerica (di rete)
- ... a stringa in notazione puntata (“dotted notation”)
- restituisce il puntatore alla stringa oppure NULL in caso di indirizzo non valido

```
#include <arpa/inet.h>

char *inet_ntop (
    int family,
    const void *addrptr,
    char *strptr, size_t length
);
```

## Esempio: validazione di un indirizzo

Scrivere un programma che:

- accetti sulla riga di comando un indirizzo IPv4 o IPv6
- indichi se è un indirizzo valido in una delle due versioni di IP

avrfy46.c

## Dimensioni degli indirizzi

- per dimensionare la rappresentazione come stringa degli indirizzi v4/v6 usare le macro definite in `<netinet/in.h>`

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

#define INET_ADDRSTRLEN 16
#define INET6_ADDRSTRLEN 46
```

## Indirizzi IPv6

- 128 bit = 8 gruppi da 16 bit, separati da “ : ”
- forma completa = 1080:0:0:0:8:800:200C:417A
- compressione degli zeri = 1080::8:800:200C:417A

## Inizializzazione di strutture (ANSI)

- per trattare strutture dati come sequenze di byte
- ... che possono includere NUL e quindi non essere trattabili con le funzioni “str...” (strcpy, strcmp, ...)

```
#include <string.h>

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

## Inizializzazione di strutture (BSD)

- funzioni pre-ANSI, definite nello Unix BSD
- ancora molto usate

```
#include <strings.h>

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

## Gestione dei segnali

- i segnali sono eventi asincroni
- ad ogni segnale ricevuto corrisponde un comportamento di default implicito
- per cambiare la risposta ad un segnale:
  - modificare il comportamento di default
  - intercettarlo registrando un **signal handler**

## Timeout

- talvolta occorre attivare dei timeout per:
  - attendere inattivi per una durata di tempo prefissata (sleep)
  - sapere quando è trascorso un certo tempo (alarm)

sveglia.c

## sleep ( )

- attiva un timer e sospende il processo per la durata indicata
- se termina perché è trascorsa la durata fissata
  - restituisce zero
- se termina perché interrotta da un segnale
  - restituisce il tempo mancante al termine fissato

```
#include <unistd.h>
unsigned int sleep (
    unsigned int seconds
);
```

## alarm ( )

- attiva un timer al cui scadere viene generato automaticamente il segnale SIGALRM
- il processo non viene sospeso
- nota: la risposta di default a SIGALRM è la terminazione del processo che lo riceve
- una nuova chiamata rimpiazza il timer corrente (es. usare `alarm(0)` per cancellare timer corrente)
- (!) unico timer per tutti i processi del gruppo

```
#include <unistd.h>
unsigned int alarm (
    unsigned int seconds
);
```

## Gestione dei segnali

- attenzione alle differenze di semantica nei vari S.O.
- usare per `signum` i nomi logici dei segnali (SIGCHLD, SIGSTOP, ...)
- handler può essere:
  - una funzione definita dall'utente
  - SIG\_IGN (ignorare il segnale)
  - SIG\_DFL (comportamento di default)

```
#include <signal.h>
typedef void Sigfunc(int);
Sigfunc *signal (int signum, Sigfunc *handler);
```

## Note sui segnali

- **SIGKILL e SIGSTOP non possono essere né intercettati né ignorati**
- **SIGCHLD e SIGURG sono ignorati di default**
- **in Unix usare `kill -l` per elencare tutti i segnali**
- **in Unix i segnali si possono anche generare manualmente col comando**

```
kill -segnale pid
```
- **ad esempio per inviare SIGHUP al processo 1234:**

```
kill -HUP 1234
```

## Segnali standard (POSIX.1)

nome	valore	azione	note
HUP	1	term	hangup of controlling terminal or death of controlling process
INT	2	term	interrupt from keyboard
QUIT	3	core	quit from keyboard
ILL	4	core	illegal instruction
ABRT	6	core	abort signal from abort( )
FPE	8	core	floating-point exception
KILL	9	term	kill
SEGV	11	core	invalid memory reference
PIPE	13	term	broken pipe (write to pipe w/o readers)
ALRM	14	term	timer signal from alarm( )
TERM	15	term	termination signal



## Segnali standard (POSIX.1) – cont.

<i>nome</i>	<i>valore</i>	<i>azione</i>	<i>note</i>
USR1	16/10/30	term	user-defined signal 1
USR2	17/12/31	term	user-defined signal 2
CHLD	18/17/20	ignore	child stopped / terminated
CONT	25/18/19	cont	continue if stopped
STOP	23/19/17	stop	stop process
TSTP	24/20/18	stop	stop from tty
TTIN	26/21/21	stop	tty input for background process
TTOU	27/22/22	stop	tty output from background process

## kill()

- per inviare un segnale da processo a processo
- restituisce 0 se OK, -1 in caso di errore
- (pid=0) invia segnale a tutti i processi del gruppo
  - spesso usato per inviare segnale a tutti i propri figli
- (pid<0) invia segnale ai processi del gruppo “-pid”
- (signum=0) non invia segnale ma effettua controlli di errore (processo o gruppo inesistente)

```
#include <sys/types.h>
#include <signal.h>
int kill ( pid_t pid, int signum );
```

## Il socket

- è la primitiva base delle comunicazioni in TCP/IP
- è il punto estremo di una comunicazione
- adatto per comunicazioni orientate al canale:
  - socket connessi (una coppia di socket connessi fornisce un'interfaccia bidirezionale tipo *pipe*)
  - modello uno-a-uno
- adatto per comunicazioni orientate ai messaggi:
  - socket non connessi
  - modello multi-a-molti

## Tipi di socket

- tre tipi fondamentali:
  - STREAM socket
  - DATAGRAM socket
  - RAW socket
- tipicamente:
  - stream e datagram usati a livello applicativo
  - raw usato nello sviluppo di protocolli (accesso a tutti i campi del pacchetto IP, incluso l'header)

## Stream socket

- **octet stream**
- **bidirezionali**
- **affidabili**
- **flusso sequenziale**
- **flusso non duplicato**
- **messaggi di dimensione illimitata**
- **interfaccia di tipo file sequenziale**
- **solitamente usato per canali TCP**

## Datagram socket

- **message-oriented**
- **messaggio = insieme di ottetti non strutturato (blob binario)**
- **bidirezionale**
- **non sequenziale**
- **non affidabile**
- **eventualmente duplicato**
- **messaggi limitati a 8 KB**
- **interfaccia dedicata (a messaggi)**
- **solitamente usato per pacchetti UDP o IP**

## Raw socket

- fornisce accesso al protocollo di comunicazione sottostante
- tipicamente di tipo datagram
- interfaccia di programmazione complessa (e spesso dipendente dal S.O.): non pensata per utenti di applicativi distribuiti
- usato per sviluppo di protocolli

## Dominio di comunicazione

- un socket è definito nell'ambito di un dominio di comunicazione
- un dominio è un'astrazione che implica:
  - una struttura di indirizzamento "Address Family" (AF)
  - un insieme di protocolli che implementano i socket nel dominio stesso "Protocol Family" (PF)
- per consuetudine si usano quasi sempre solo AF (perché esiste corrispondenza biunivoca con PF)

## Binding

- un socket viene creato senza un identificativo
- nessun processo può referenziare o accedere ad un socket privo di identificativo
- prima di poter usare un socket è necessario associargli un identificativo
  - = indirizzo di rete (per socket di rete)
  - = nome logico (per socket di S.O.)
- il binding stabilisce l'indirizzo del socket e lo rende accessibile in rete
- il binding è legato al protocollo usato
- binding esplicito solitamente fatto dai server (per il client indirizzo determinato dal SO in base al routing)

## Associazione (dominio INET)

- affinché due processi possano comunicare in rete deve esistere tra loro un'associazione
- nel dominio AF\_INET un'associazione è una quintupla
- tutte le quintuple devono essere uniche

protocollo (TCP, UDP, ...)
indirizzo IP (locale)
porta (locale)
indirizzo IP (remoto)
porta (remota)

## Associazione (dominio Unix)

- due processi sullo stesso nodo Unix possono comunicare tra loro un'associazione locale
- nel dominio AF\_UNIX un'associazione è una terna
- tutte le terne devono essere uniche

protocollo (channel, msg)
pathname (locale)
pathname (remoto)

## Associazioni

- si creano con la system call `bind( )` che crea una metà dell'associazione
- l'associazione viene completata:
  - dal server con `accept( )`, che elimina la richiesta dalla coda e crea un socket dedicato per la connessione; è una chiamata bloccante
  - dal client con `connect( )`, che assegna anche la porta locale; è una chiamata bloccante

## Socket connessi (stream)

- la creazione di una connessione è tipicamente un'operazione asimmetrica
- ogni processo crea il proprio endpoint con `socket( )`
- il server:
  - assegna un identificativo al socket tramite `bind( )`
  - si mette in ascolto sul suo socket tramite `listen( )`
  - quando arriva una richiesta di collegamento la accetta con `accept( )`, che elimina la richiesta dalla coda e crea un socket dedicato per la connessione
- il client si collega al server con `connect( )`, che effettua implicitamente il binding assegnando anche la porta locale

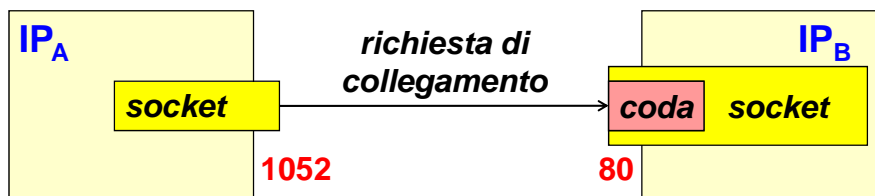
## Stream socket: pre-connessione



**Il client:**  
1. crea un socket

**Il server:**  
1. crea un socket  
2. associa una porta al socket  
3. si mette in ascolto sulla coda

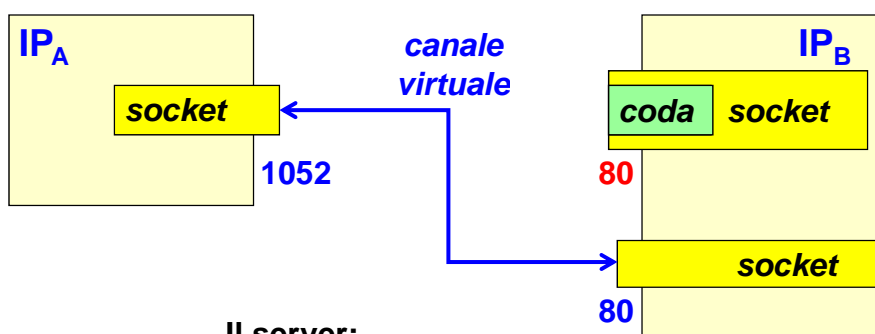
## Stream socket: richiesta di connessione



**Il client:**  
1. richiede collegamento alla porta del server

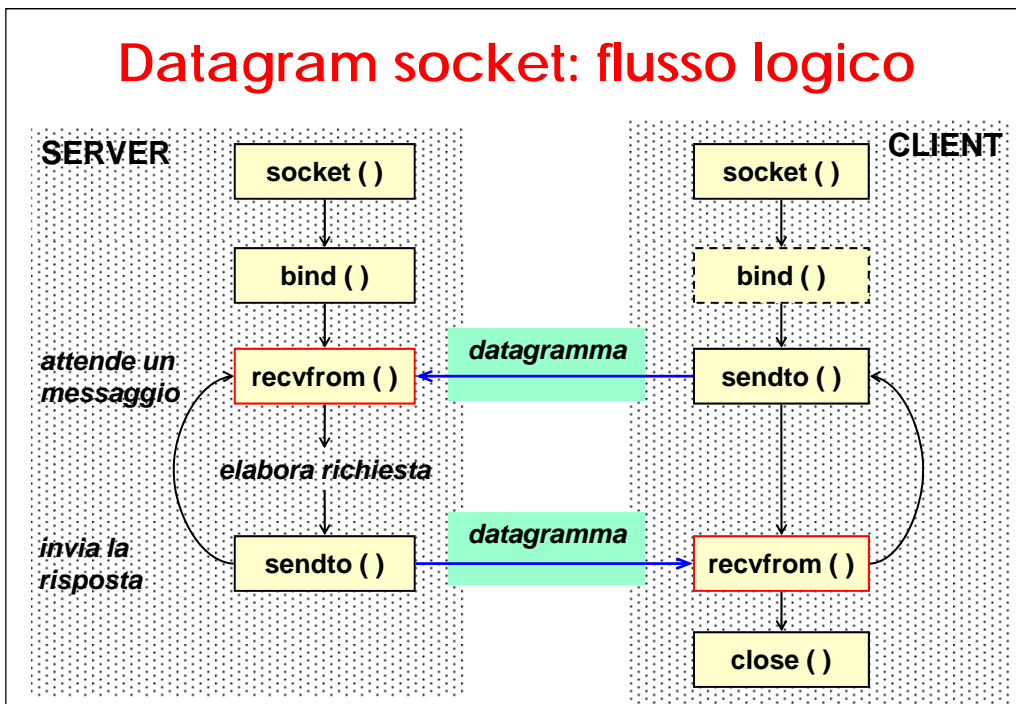
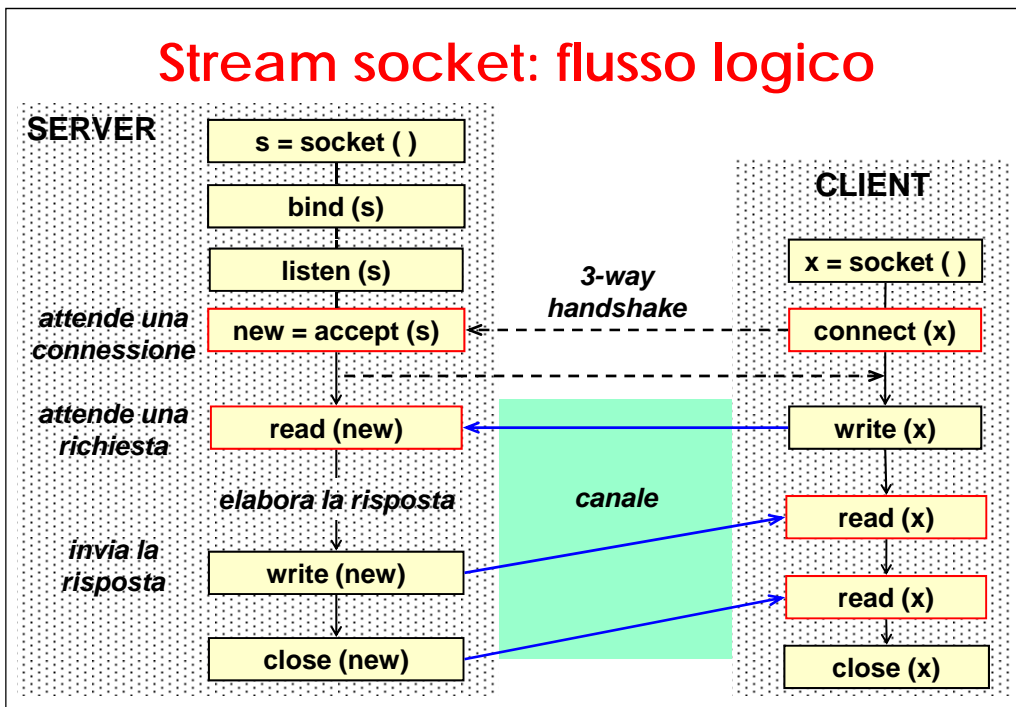
**Il server:**  
1. riceve una richiesta di collegamento

## Stream socket: connessione stabilita



**Il server:**  
1. accetta la richiesta  
2. completa il canale con un nuovo socket  
3. si rimette in attesa sulla coda del socket originale





## Datagram socket - differenze

- **permettono di:**
  - scambiare dati senza connessione (perché i messaggi contengono l'indirizzo di destinazione e provenienza)
  - inviare da un socket a più destinazioni
  - ricevere su un socket da più sorgenti
- **quindi, in generale, il modello è “molti a molti”**
- **i termini client e server si usano solo nel senso della applicazione**
- **non esistono differenze tra le chiamate effettuate dai vari processi coinvolti nella comunicazione**

## I socket in C



## Il compilatore gcc

- uno dei migliori compilatori
- usare flag "-Wall -Werror" per essere certi di non trascurare potenziali sorgenti di errore
- attenzione: il warning sull'uso di variabili non inicializzate viene emesso solo se si ottimizza il programma (almeno con -O1)

## Note per la compilazione

- in Solaris > 7 occorre agganciare varie librerie:
  - libsocket
    - funzioni base dei socket
  - libnsl (name services library)
    - inet\_addr (e altro)
  - libresolv (name resolver library)
    - h\_errno, hstrerror (e altro)

```
gcc -lsocket -lnsl -lresolv ...
```

## make e makefile

- regole in file di nome "Makefile"
  - target: dipendenze ...
  - TAB comando\_per\_creare\_il\_target
- **\$ make [ target ]**
  - il primo target è quello di default
- **esempio:**

```
prova.exe: prova.o mylib.o
    gcc -o prova.exe prova.o lib.o
prova.o: prova.c mylib.h
    gcc -o prova.o -c prova.c
mylib.o: mylib.c mylib.h
    gcc -o lib.o -c lib.c
```

## Dipendenze

- **gcc -MM file**
  - analizza il file e ne crea l'elenco delle dipendenze nel formato di make, ignorando gli header di sistema
  - molto utile per creare ed includere automaticamente le dipendenze (col comando "make depend")

```
depend:
    rm -f depend.txt
    for f in $(SRC); do gcc -MM $$f>>depend.txt; done

# dependencies
-include depend.txt
```

## Dati applicativi: codifiche

- **codifica ASCII:**
  - sscanf / sprintf per leggere / scrivere dati ASCII (solo se sono stringhe C, ossia terminati da \0)
  - memcpy (se non sono stringhe ma è nota lunghezza)
- **codifica binaria:**
  - hton / ntoh / XDR per leggere / scrivere dati binari
- **esempi (n<sub>10</sub>=2252):**

<pre> ASCII (4 byte) '2'  '2'  '5'  '2' 0x32 0x32 0x35 0x32                 </pre>	<pre> bin (16 bit) 0x08 0xCC                 </pre>
--	---

## Dati applicativi: formati

- **formato fisso:**
  - leggere / scrivere il numero di byte specificato
  - convertire i byte secondo la loro codifica
- **formato variabile (con separatori e terminatore):**
  - leggere sino al terminatore, quindi fare la conversione
  - attenzione all'overflow se manca il terminatore
- **formato variabile (TLV = tag-length-value):**
  - leggere il n. di byte dato da length
  - fare la conversione secondo il formato dato dal tag

## Unix socket descriptor

- è un normale descrittore di file riferito ad un socket anziché ad un file
- può essere usato normalmente per la lettura o la scrittura con le funzioni di direct I/O
- possibile usare le system call operanti su file
  - close, read, write
  - eccezione: seek
- disponibili altre system call per funzioni peculiari dei socket (ossia non applicabili a file)
  - send, recv, ...
  - sendto, recvfrom, ...

## socket()

- crea un socket
- restituisce il socket descriptor in caso di successo, -1 in caso di errore
- family = costante di tipo AF\_x
- type = costante di tipo SOCK\_x
- protocol = 0 (ad eccezione nei socket raw)

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

## socket(): parametri

### ■ family:

- AF\_INET
- AF\_INET6
- AF\_LOCAL (AF\_UNIX)
- AF\_ROUTE
- AF\_KEY

### ■ type:

- SOCK\_STREAM
- SOCK\_DGRAM
- SOCK\_RAW
- SOCK\_PACKET (Linux, accesso al layer 2)

## Combinazioni possibili

	AF_INET	AF_INET6	AF_LOCAL	AF_ROUTE	AF_KEY
SOCK_STREAM	TCP	TCP	yes		
SOCK_DGRAM	UDP	UDP	yes		
SOCK_RAW	IPv4	IPv6		yes	yes

## socket() : wrapper

- conviene scrivere una volta per tutte i test invece di ripeterli ogni volta

```
int Socket (int family, int type, int protocol)
{
    int n;
    if ( (n = socket(family,type,protocol)) < 0)
        err_sys ("%s) error - socket() failed", prog);
    return n;
}
```

## Il concetto di “wrapper”

- data una funzione da “wrappare”, creare una funzione:
  - omonima ma iniziante con lettera maiuscola
  - con gli stessi identici parametri
  - di tipo void (a meno che il valore di ritorno sia la risposta della funzione), perché i controlli sono fatti all'interno della nuova funzione
- non occorre inventare controlli “esotici” ma solo leggere attentamente la descrizione della funzione ed attuare i controlli sul valore di ritorno (o su altri meccanismi di segnalazione d'errore)



## Esempio di wrapper

- al posto di `strcpy( )` si preferisca sempre `strncpy( )`
- ... ma anche questa funzione può generare errori:

```
char *strncpy (char *DST, const char *SRC, size_t LENGTH);
```

### DESCRIPTION

'strncpy' copies not more than LENGTH characters from the string pointed to by SRC (including the terminating null character) to the array pointed to by DST.

### RETURNS

This function returns the initial value of DST.

## Wrapper per strncpy

```
void Strncpy (
    char *DST, const char *SRC, size_t LENGTH)
{
    char *ret = strncpy(DST, SRC, LENGTH);
    if (ret != DST)
        err_quit(
            "(%s) library bug - strncpy() failed", prog);
}
```

## Uso di funzioni "wrappate"

- se si verifica un errore, tipicamente il wrapper lo segnala e poi termina il processo chiamante
- se il chiamante è un figlio in un server concorrente, non c'è solitamente problema
- se il chiamante è il processo padre in un server concorrente o è un client che deve continuare ad interagire con l'utente, allora terminare il chiamante potrebbe non essere la cosa corretta da fare ...

## Indirizzo di un socket

- si usa la struttura sockaddr che è l'indirizzo di un generico socket (Internet, Unix, ...)
- in realtà è solo un overlay per i casi specifici (sockaddr\_in, sockaddr\_un, ...)
- definita in `<sys/socket.h>`

```
struct sockaddr {
    uint8_t      sa_len; // non obbligatorio
    sa_family_t  *sa_family, // AF_XXX
    char        sa_data[14] // identificativo
}
```

## Indirizzo di un socket Internet

- indirizzo di rete di livello 3 (in formato di rete)
- porta di livello 4 (in formato di rete)
- il protocollo di livello 4 è definito implicitamente in base al tipo di socket (STREAM, DGRAM)

```
struct sockaddr_in
{
    uint8_t          sin_len; // non obbligatorio
    sa_family_t     *sin_family; // AF_INET
    in_port_t       sin_port; // porta TCP o UDP
    struct in_addr  sin_addr; // indirizzo IP
    char            sin_zero[8] // non usato
}
```

## connect()

- crea un collegamento tra un socket “locale” ed un socket “remoto”, specificato tramite il suo identificativo (=indirizzo e porta)
- in pratica avvia il TCP 3-way handshake
- il sistema operativo assegna automaticamente un identificativo appropriato al socket locale (indirizzo e porta)
- restituisce 0 se OK, -1 in caso di errore

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

## bind()

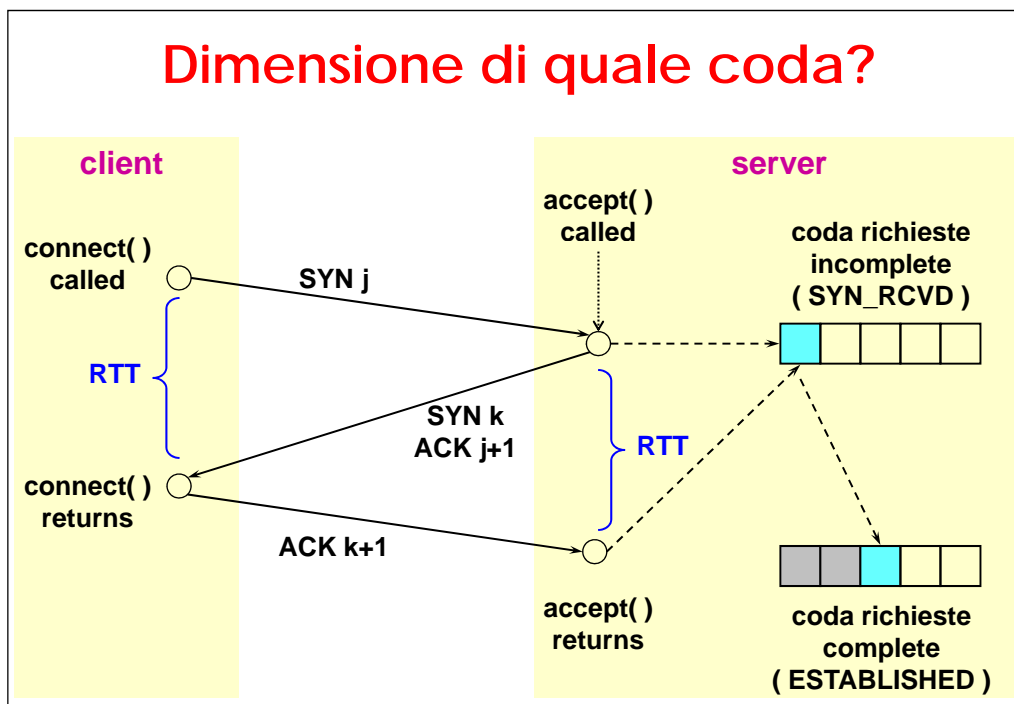
- assegna un indirizzo ad un socket
- restituisce 0 in caso di successo, -1 in caso di errore
- se non si specifica l'indirizzo IP, viene assegnato dal kernel in base al pacchetto SYN ricevuto
- INADDR\_ANY per specificare un indirizzo qualunque

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

## listen()

- trasforma un socket da attivo in passivo
- specifica la dimensione della coda delle richieste pendenti (= somma delle due code, più talvolta un fattore di aggiustamento 1.5)
- fattore critico per:
  - i server ad alto carico
  - resistere agli attacchi "SYN flooding"
- restituisce 0 in caso di successo, -1 altrimenti

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



## Note sul parametro "backlog"

- in Linux (a partire dal kernel 2.2) ed in Solaris (dalla 2.9) si usano i SYN\_COOKIE e quindi il parametro specifica solo la lunghezza della coda Established
- la lunghezza della coda SYN\_RCVD è:
  - dimensionata automaticamente (256-1024 entry) in funzione della RAM disponibile
  - modificabile tramite `sysconf` o `.../ip/...`
  - infinita se si attivano i SYN\_COOKIE (default da kernel 2.4)

## listen() : wrapper

- conviene non fissare la dimensione della coda nel codice ma renderla modificabile (via argomenti o variabili di ambiente)
- ad esempio con un wrapper:

```
#include <stdlib.h> // getenv()

void Listen (int sockfd, int backlog)
{
    char *ptr;
    if ( (ptr = getenv("LISTENQ")) != NULL)
        backlog = atoi(ptr);
    if ( listen(sockfd,backlog) < 0 )
        err_sys ("%s) error - listen failed", prog);
}
```

## Lettura variabili di ambiente in C

- usare la funzione `getenv( )`
- riceve in input il nome della variabile di environment come stringa
- restituisce il puntatore alla stringa relativa al valore
- ... oppure NULL se la variabile non è definita

```
#include <stdlib.h>
char *getenv (const char *varname);
```

## Elenco variabili di ambiente in C

- parametro “envp” passato alla funzione main
  - supportato da MS-VC++ e gcc ... ma non è ANSI
- envp è un array di puntatori a stringhe
  - ogni stringa contiene la coppia:
    - NOME\_VARIABILE=VALORE\_VARIABILE
  - ultimo elemento dell'array ha il valore NULL
    - necessario, dato che non c'è un parametro che riporti il numero delle stringhe

```
int main (int argc, char *argv[], char *envp[]);
```

## Impostare var. d'ambiente (in BASH)

- export VARNAME=VARVALUE
  - aggiunge la variabile all'ambiente di esecuzione
- export -n VARNAME
  - rimuove la variabile dall'ambiente di esecuzione
- env VARNAME=VARVALUE COMANDO
  - esegue il comando inserendo temporaneamente nel suo ambiente la variabile
- printenv [ VARNAME ]
  - elenca tutte le variabili di ambiente o quella indicata
- nota: “export” è built-in in bash mentre “env” e “printenv” sono comandi esterni (in /usr/bin)

## Esempi lettura variabili d'ambiente

- `printevar.c` stampa il valore della variabile d'ambiente il cui nome è fornito sulla riga di comando
- `prallenv.c` stampa nomi e valori di tutte le variabili d'ambiente definite

`printevar.c`

`prallenv.c`

## accept()

- preleva il primo collegamento disponibile nella coda delle richieste pendenti
- si blocca se non ci sono richieste pendenti (eccezione: se il socket è non bloccante)
- restituisce un **nuovo socket descriptor**, connesso con quello del client
- side effect: restituisce l'identificativo del client che si è collegato (a meno che si fornisca NULL)

```
#include <sys/socket.h>
int accept (int listen_sockfd,
            struct sockaddr *cliaddr, socklen_t *addrlenp)
```



## close()

- chiude immediatamente il socket
- il socket non è più accessibile al processo ma il kernel cercherà di inviare eventuali dati rimasti e poi eseguire la chiusura del canale TCP
- comportamento cambiabile con SO\_LINGER
- restituisce 0 se termina con successo, -1 in caso di errore

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

## Comunicazione su stream

- usare funzioni di I/O non bufferizzato per evitare
  - di restare indefinitamente in attesa (input)
  - di terminare precocemente in caso di NUL (output)
- mai la libreria di standard I/O
- in particolare usare le system call read( ) e write( ):
  - operano su file descriptor
  - restituiscono il numero di byte letti o scritti, -1 in caso di errore

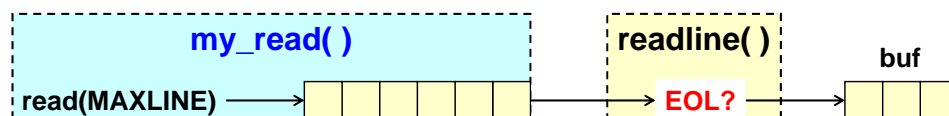
```
#include <unistd.h>
ssize_t read (int fd, void *buf, size_t nbyte )
ssize_t write(int fd, const void *buf, size_t nbyte)
```

## Risultato di una read su un socket

- maggiore di zero: numero di byte ricevuto
- uguale a zero : socket chiuso (EOF)
- minore di zero : errore
  
- **attenzione:**
  - a causa della frammentazione e dei buffer ...
  - ... il numero di dati letti può essere inferiore al numero atteso
  - conviene quindi scrivere funzioni che trattino automaticamente questo problema

## Funzioni di lettura / scrittura "sicure"

- **readn( )** e **writen( )** leggono e scrivono esattamente N byte, a meno che incontrino errore o EOF
- **readline( )** legge fino ad incontrare LF oppure a riempire il buffer, a meno di incontrare errore o EOF
- riferimento: UNP, figg. 3.14, 3.15, 3.16
- nota: **readline( )** deve necessariamente leggere un byte alla volta ma usa un'altra funzione (privata) per caricare un buffer in modo più efficiente



## readn, writen, readline

- le funzioni con iniziale maiuscola controllano automaticamente gli errori

```

ssize_t readn (int fd, void *buf, size_t nbyte)
ssize_t Readn (int fd, void *buf, size_t nbyte)

ssize_t writen(int fd, const void *buf, size_t nbyte)
void Writen (int fd, const void *buf, size_t nbyte)

ssize_t readline (int fd, void *buf, size_t nbyte)
ssize_t Readline (int fd, void *buf, size_t nbyte)
    
```

sockwrap.h

sockwrap.c

## Uso di errlib e sockwrap

- sockwrap usa errlib per segnalare gli errori
- errlib richiede che esista la variabile "prog" e sia inizializzata al nome del programma in esecuzione

```

#include "errlib.h"
#include "sockwrap.h"
char *prog;

int main (int argc, char *argv[])
{
    . . .
    prog = argv[0];
    . . .
}
    
```

## Attenzione!

- **poiché my\_read usa un buffer locale, le funzioni readline non sono rientranti:**
  - non possono essere usate in ambiente multiprocesso o multithread
  - per questi casi occorre sviluppare funzioni rientranti allocando esternamente il buffer per my\_read
  
- **le chiamate a readline non possono essere mischiate con quelle a normali read perché fanno "read ahead" e perciò "rubano dati" che dovrebbero invece essere consumati da una read diversa**

## Esempio: TCP daytime client e server

- **il servizio daytime (tcp/13):**
  - fornisce data e ora correnti in formato comprensibile da un essere umano
  - le risposte sono terminate da CR+LF
- **sviluppare un client che si colleghi al servizio daytime del server specificato sulla riga di comando**
- **sviluppare un server (iterativo) che attende le richieste di servizio e fornisca data e ora, identificando il client che si è collegato**

daytimetcp.c

daytimetcps.c

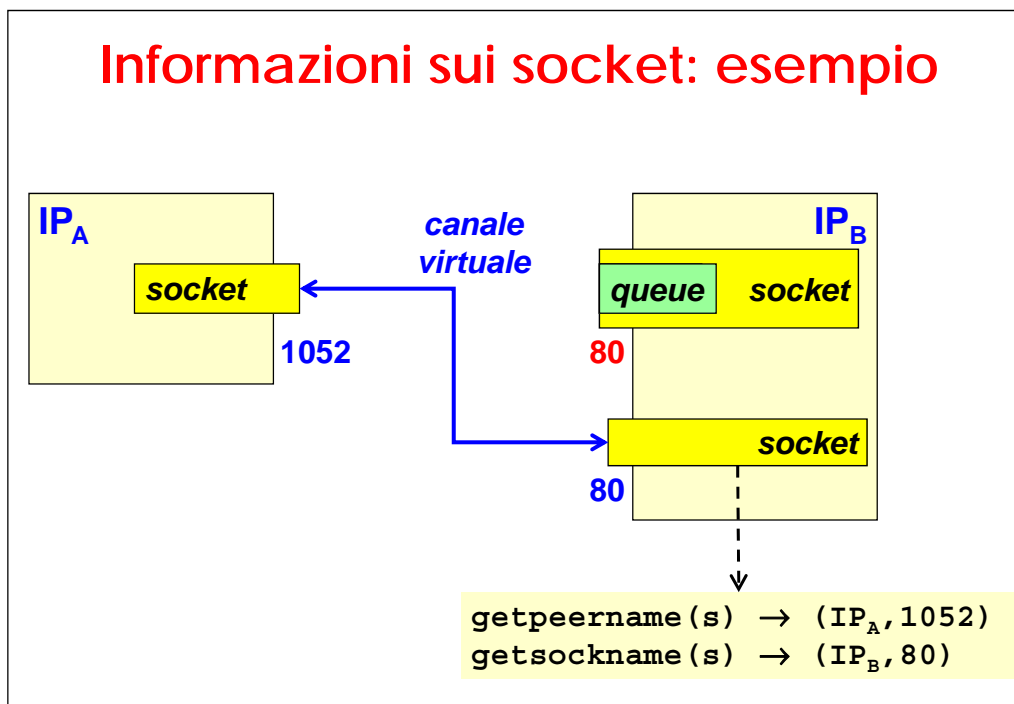
## Esercizi

- **togliere la listen( ) al server:**
  - cosa capita? perché?
- **lasciare la listen( ) ma togliere la bind( ):**
  - cosa capita? perché?

## Informazioni sui socket

- **per conoscere indirizzo e porta**
  - della parte locale: getsockname( )
  - della parte remota: getpeername( )
- **restituiscono 0 se tutto OK, -1 in caso di errore**

```
#include <sys/socket.h>
int getsockname ( int sockfd,
    struct sockaddr *localaddr, socklen_t *addrp )
int getpeername ( int sockfd,
    struct sockaddr *peeraddr, socklen_t *addrp )
```

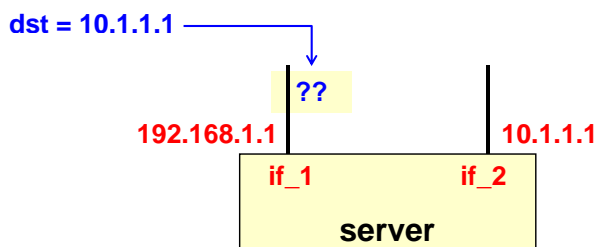


## Binding implicito o esplicito?

- **consideriamo il seguente caso:**
  - server web multihomed (N indirizzi di rete)
  - un sito web diverso per ogni indirizzo
  
- **se il server fa binding a INADDR\_ANY:**
  - un solo processo in ascolto
  - demultiplexing fatto dal server
  
- **se il server fa binding ai singoli N indirizzi:**
  - N processi in ascolto
  - demultiplexing fatto dallo stack TCP/IP

## weak-end / strong-end model

- nel caso di server multihomed ...
- “strong-end model” = quei kernel che accettano pacchetti su un’interfaccia solo se DST\_IP è uguale all’IP dell’interfaccia
- “weak-end model” = quei kernel che accettano pacchetti solo se DST\_IP è uguale all’IP di una qualunque interfaccia del server



## Porte dedicate a server e client

- IANA
  - 1-1023 = well-known ports
  - 1024-49151 = registered ports
  - 49152-65535 = dynamic / ephemeral ports
- UNIX
  - 1-1023 = riservate solo a processi con EUID=0
  - 513-1023 = riservate a client privilegiati (r-cmds)
  - 1024-5000 = BSD ephemeral ports (poche!)
  - 5001-65535 = BSD servers (non-privileged)
  - 32768-65535 = Solaris ephemeral ports
- Windows non fa invece nessun controllo

## La porta zero

- la porta 0 è riservata e non può essere usata per nessun collegamento TCP o UDP
- nel caso di uso dell'interfaccia socket sotto Unix il valore zero può essere usato per richiedere al SO una porta dinamica a caso:
  - solitamente utile solo in UDP
  - il comportamento è diverso con l'interfaccia socket di Windows

## recvfrom() e sendto()

- usate per i socket datagram, pur potendosi usare anche con stream
- restituiscono il numero di byte letti o scritti, -1 in caso di errore
- il parametro "flags" è di solito zero (maggiori dettagli in seguito)

```
#include <sys/socket.h>
int recvfrom ( int sockfd,
               void *buf, size_t nbytes, int flags,
               struct sockaddr *from, socklen_t *addrlenp )
int sendto ( int sockfd,
             const void *buf, size_t nbytes, int flags,
             const struct sockaddr *to, socklen_t addrlen )
```



## Ricezione dati con `recvfrom()`

- ricevere zero dati è OK (=payload UDP vuoto) e non segnala EOF (non esiste in socket datagram!)
- usando NULL come valore di "from" si accettano dati da chiunque ... ma poi non si conosce l'indirizzo per rispondere (!), a meno di riceverlo a livello applicativo

## Binding con socket datagram

- di solito il client non fa `bind()` ma al socket viene assegnato automaticamente un indirizzo e porta la prima volta che viene usato
- in alternativa il client può fare `bind()` alla porta 0 che indica al kernel di assegnargli una porta effimera a caso

## Esempio: UDP daytime client e server

- il servizio daytime (udp/13) fornisce data e ora correnti in formato comprensibile da un essere umano
- il server invia data e ora in un pacchetto UDP ad ogni client che gli invia un qualsiasi pacchetto UDP, anche vuoto
- sviluppare un client che si colleghi al servizio daytime del server specificato sulla riga di comando
- sviluppare un server (iterativo) che attende le richieste di servizio e fornisca data e ora, identificando il client che si è collegato

daytimeudpc.c

daytimeudps.c

## cygwin: problemi noti

- sendto( ) con buffer di dimensione zero non spedisce niente (dovrebbe spedire UDP con payload di lunghezza zero)

## Problemi dei socket datagram

- **problema 1: poiché UDP non è affidabile, il client rischia di rimanere bloccato all'infinito in ricezione**
- **usare un timeout risolve il problema solo talvolta:**
  - OK se rispedita la richiesta non crea problemi
  - inaccettabile in caso contrario (es. transazione di debito o credito)
- **problema 2: come si verifica che la risposta arrivi proprio dal server a cui abbiamo fatto la richiesta?**
- **si devono filtrare le risposte a livello utente oppure a livello kernel**

## Problema 1

```
enum { normale, la_sveglia_ha_suonato } state;

sveglia (...) {
    state = la_sveglia_ha_suonato;
}

signal (SIGALRM, sveglia);
do {
    sendto (...);
    state = normale; alarm (timeout);
    recvfrom (...);
    alarm (0);
} while (state == la_sveglia_ha_suonato);
```

## Problema 1: note

- la soluzione con alarm è soggetta ad una corsa critica (arrivo dei dati simultaneo con lo scadere del timeout)
- con timeout “lungo” è poco probabile che capitino ma non si può escludere
- meglio quindi impostare un timeout direttamente sul socket oppure tramite select( )

## Verifica delle risposte datagram

- confronto binario dell'indirizzo del rispondente con quello del destinatario originale
- possibili problemi coi server multihomed
- soluzione 1: fare la verifica non sull'indirizzo ma sul nome DNS (funziona solo con server registrati)
- soluzione 2: il server fa binding esplicito a tutti i suoi indirizzi e poi aspetta su tutti (select)

```
n = recvfrom(sfd, *buf, nbyte, 0, (SA*)&from, &fromlen);
if ( (fromlen == serverlen) &&
    (memcmp(&servaddr, &from, serverlen) == 0) )
    // risposta accettata
```

## Errori asincroni

- quando si verifica un errore in una trasmissione UDP (es. port unreachable) viene generato un pacchetto ICMP di errore ...
- ... ma la funzione `sendto( )` è già terminata con stato OK
- ... e quindi il kernel non sa a quale applicazione fornire l'errore (ed in quale modo!)
- **soluzioni possibili:**
  - usare socket datagram connessi (!)
  - intercettare gli errori ICMP con un proprio demone

## Socket datagram connessi

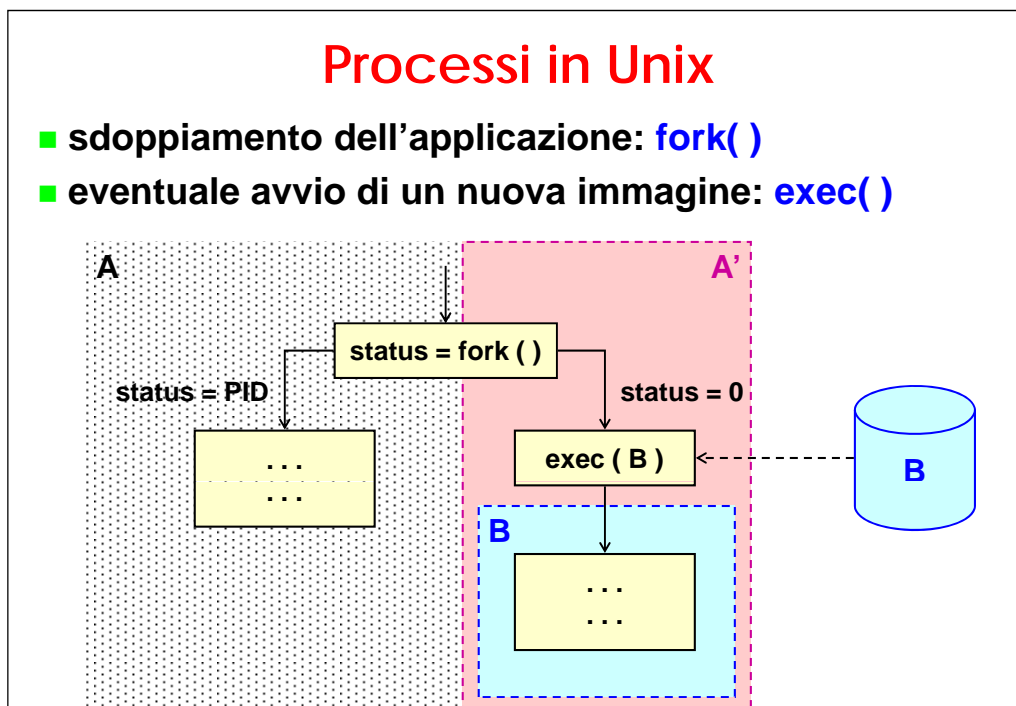
- è possibile chiamare `connect( )` su un socket datagram per specificare una volta per tutte il peer con cui si intende comunicare
- **conseguenze:**
  - non si usa più `sendto( )` ma `write( )` o `send( )`
  - non si usa più `recvfrom( )` ma `read( )` o `recv( )`
  - gli errori asincroni che si verificano vengono restituiti al processo che controlla il socket
  - il kernel effettua automaticamente il filtraggio delle risposte accettando solo pacchetti dal peer
- **in caso di comunicazione ripetuta con lo stesso peer c'è un buon miglioramento delle prestazioni**

## Disconnettere un socket datagram

- si può fare una seconda connect( ) verso un altro indirizzo per cambiare la connessione
- vietato per socket stream
  
- per disconnettere completamente (ossia tornare a socket non connesso) fare connect( ) verso un indirizzo non specificato mettendo AF\_UNSPEC
- in questo caso si potrebbe ricevere un errore EAFNOSUPPORT che si può però trascurare

## Server concorrenti





## fork()

- genera **un nuovo processo** ...
- ... che **condivide immagine ed ambiente di esecuzione col processo che ha fatto fork()**
- **return status:**
  - -1 in caso di errore
  - 0 per il nuovo processo ("figlio" o "child")
  - PID del nuovo processo al "padre" (o "parent")

```
#include <unistd.h>
pid_t fork (void);
```

## getpid(), getppid()

- il figlio può conoscere il PID del padre tramite `getppid()`
- nota: il padre di ogni processo è unico
- per conoscere il proprio PID usare `getpid()`
- return status:
  - -1 in caso di errore
  - il PID desiderato

```
#include <unistd.h>
pid_t getpid (void);
pid_t getppid (void);
```

## Funzioni exec

- sostituiscono l'immagine in esecuzione con una nuova immagine
- restituiscono -1 in caso di errore
- le funzioni L passano gli argomenti come lista di variabili, terminata da NULL
- le funzioni V passano gli argomenti come vettore di puntatori, con ultimo elemento NULL
- le funzioni P localizzano l'immagine tramite PATH, le altre vogliono il pathname completo
- le funzioni con E ricevono le variabili di ambiente come vettore di puntatori, con ultimo elemento NULL; le altre usano la var. esterna "environ"

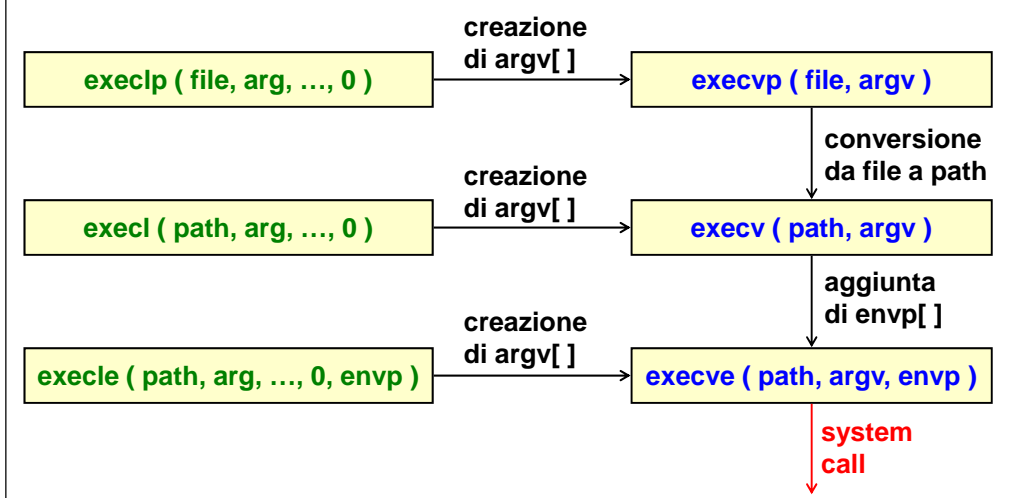


## Funzioni execv() e execl()

```
#include <unistd.h>
int execv ( const char *filename,
            char *const argv[] );
int execve ( const char *filename,
            char *const argv[], char *const envp[] );
int execvp ( const char *pathname,
            char *const argv[], char *const envp[] );
int execl ( const char *filename,
            const char *arg0, ..., (char *)NULL );
int execlp ( const char *pathname,
            const char *arg0, ..., (char *)NULL,
            char *const envp[] );
int execlp ( const char *pathname,
            const char *arg0, ..., (char *)NULL,
            char *const envp[] );
```

## Funzioni exec

- solitamente solo execve( ) è una system call



## Comunicazione attraverso la exec()

- **passaggio dei parametri esplicito:**
  - attraverso gli argomenti
  - attraverso le variabili di ambiente
- **i descrittori di file (file e socket) restano aperti, a meno che il chiamante usi fcntl( ) per settare il flag FD\_CLOEXEC che li fa chiudere automaticamente quando si esegue una exec**

## Scheletro di server concorrente (I)

```
pid_t pid; // PID del figlio
int listenfd; // socket di ascolto
int connfd; // socket di comunicazione

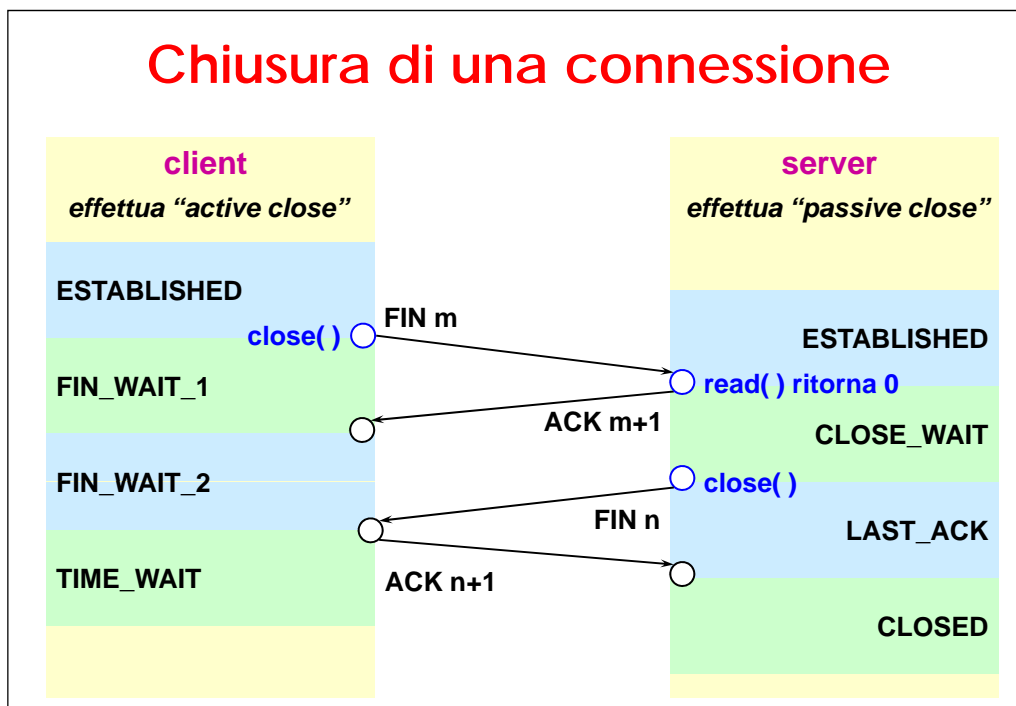
// creazione del socket di ascolto
listenfd = Socket( ... );
servaddr = ...
Bind (listenfd, (SA*)&servaddr, sizeof(servaddr));
Listen (listenfd, LISTENQ);
```

## Scheletro di server concorrente (II)

```
// loop di esecuzione del server
while (1)
{
    connfd = Accept (listenfd, ...);
    if ( (pid = Fork()) == 0 )
    {
        Close(listenfd);
        doit(connfd); // il figlio svolge il lavoro
        Close(connfd);
        exit(0);
    }
    Close (connfd);
}
```

## Importanza della close( )

- **se il padre dimentica di chiudere il socket di connessione ...**
  - esaurisce in fretta i descrittori
  - il canale col client resta aperto anche quando il figlio ha terminato ed ha chiuso il socket di connessione
- **note:**
  - la funzione close( ) non chiude il socket ma semplicemente ne decrementa il reference count
  - solo quando REFCNT diventa zero, il socket viene chiuso (ossia si invia FIN se è un socket TCP)



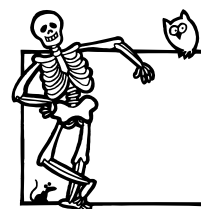
## Lo stato TIME\_WAIT

- **dallo stato TIME\_WAIT si esce solo per timeout:**
  - durata pari a 2 x MSL (Max Segment Lifetime)
  - MSL = 2 minuti (RFC-1122), 30 secondi (BSD)
  - quindi timeout 1...4 minuti
- **esiste per risolvere due problemi:**
  - implementare la chiusura TCP full-duplex
    - l'ultimo ACK potrebbe venir perso ed il client ricevere un nuovo FIN
  - permettere a pacchetti duplicati di "spirare"
    - potrebbero essere interpretati come parte di una nuova incarnazione della stessa connessione

## Terminazione dei figli

- quando un processo figlio termina, viene inviato il segnale SIGCHLD al padre
- reazione di default:
  - ignorato
  - ... il che genera un processo "zombie"
- zombie ereditati ed eliminati dal processo init solo quando il padre termina (ma di solito i server non terminano mai ...)

If we want to avoid zombies,  
 we have to wait for our children.  
 -- W.R.Stevens



## wait() e waitpid()

- ritornano:
  - il PID del figlio terminato; 0 o -1 in caso di errore
  - lo status di terminazione del figlio
- wait() è bloccante
- waitpid():
  - non si blocca se si usa l'opzione WNOHANG
  - permette di specificare il PID di un figlio specifico (-1 per attendere il primo che termina)

```
#include <sys/wait.h>
pid_t wait (int *status);
pid_t waitpid (pid_t pid, int *status, int options);
```

## Intercettare SIGCHLD

- se più figli terminano “simultaneamente” si genera un solo SIGCHLD, quindi bisogna attenderli tutti

```
#include <sys/wait.h>
void sigchld_h (int signum)
{
    pid_t pid;
    int status;

    while ( (pid = waitpid(-1,&status,WNOHANG)) > 0)
#ifdef TRACE
        err_msg (
            "(%s) info - figlio %d terminato con status %d\n",
            prog, pid, status)
#endif
        ; // occhio a questo punto e virgola
}
```

## system call interrotte

- quando un processo esegue una system call “lenta” (ossia una che può bloccare il chiamante)
- ... può sbloccarsi non perché la system call è terminata
- ... ma perché è arrivato un segnale; questo caso è segnalato da `errno == EINTR` (o `ECHILD`)
- occorre quindi prevedere questo caso e ripetere la system call (azione già svolta in `sockwrap`)
- caso molto rilevante per la `accept( )` nei server
- **ATTENZIONE:** si possono ripetere tutte le system call tranne `connect( )`; in questo caso occorre usare la `select( )`

## Server concorrente: esempio

- sviluppare un server concorrente in ascolto sulla porta tcp/9999 che riceva righe di testo contenenti due numeri interi e ne restituisca la somma
- sviluppare il client che:
  - legga righe di testo da standard input
  - le invii alla porta tcp/9999 del server specificato sulla riga di comando
  - riceva righe di risposta e le visualizzi su standard output

addtcps.c

addtcp.c

## Robustezza delle applicazioni



## Server lento

- se un server è lento o sovraccarico ...
- ... il client può completare il 3-way handshake e poi terminare la connessione (RST)
- ... nel tempo che il server impiega tra listen e accept
- questo problema
  - può essere trattato direttamente dal kernel o può generare EPROTO / ECONNABORTED in accept
  - caso frequente nei server web sovraccarichi

## Terminazione del server-figlio

- quando il server-figlio che comunica col client termina propriamente (exit) viene chiuso il socket e quindi:
  - si genera un FIN, accettato dal kernel client ma non trasmesso all'applicazione fino alla prossima read
  - se il client esegue una write riceverà un RST
  - a seconda della tempistica, il client riceverà errore sulla write oppure EOF o ECONNRESET sulla prossima read
  - se il client non ha ancora letto i dati inviati dal server quando questi chiude il canale, questi dati potrebbero essere persi (meglio quindi una chiusura parziale lato server con shutdown\_write + timeout + exit)



## SIGPIPE

- **chi esegue write su un socket che ha ricevuto RST, riceve il segnale SIGPIPE**
  - default: terminare il processo
  - se viene intercettato o ignorato, la prossima write genera l'errore EPIPE
- **attenzione:**
  - se ci sono tanti socket aperti in scrittura ...
  - ... SIGPIPE non segnala quale ha generato errore
  - meglio quindi ignorare il segnale e ricevere EPIPE sulla write

## Crash del server

- **copre anche il caso di server irraggiungibile**
- **le write del client funzionano (non c'è nessuno a rispondere con un errore!)**
- **le read del client andranno in timeout (a volte dopo molto tempo: in BSD 9 m!) generando ETIMEDOUT**
- **read e write potrebbero ricevere EHOSTUNREACH o ENETUNREACH se un router intermedio si accorge del problema e lo segnala tramite ICMP**
- **soluzione: impostare un timeout**
  - direttamente sul socket con le opzioni
  - tramite la `select( )`
  - tramite la `alarm( )` – sconsigliata

## Crash e reboot del server

- **sequenza:**
  - crash (= server irraggiungibile)
  - boot (=server raggiungibile ma ha perso conoscenza delle connessioni esistenti: RST)
- **come conseguenza read e write falliscono con ECONNRESET**

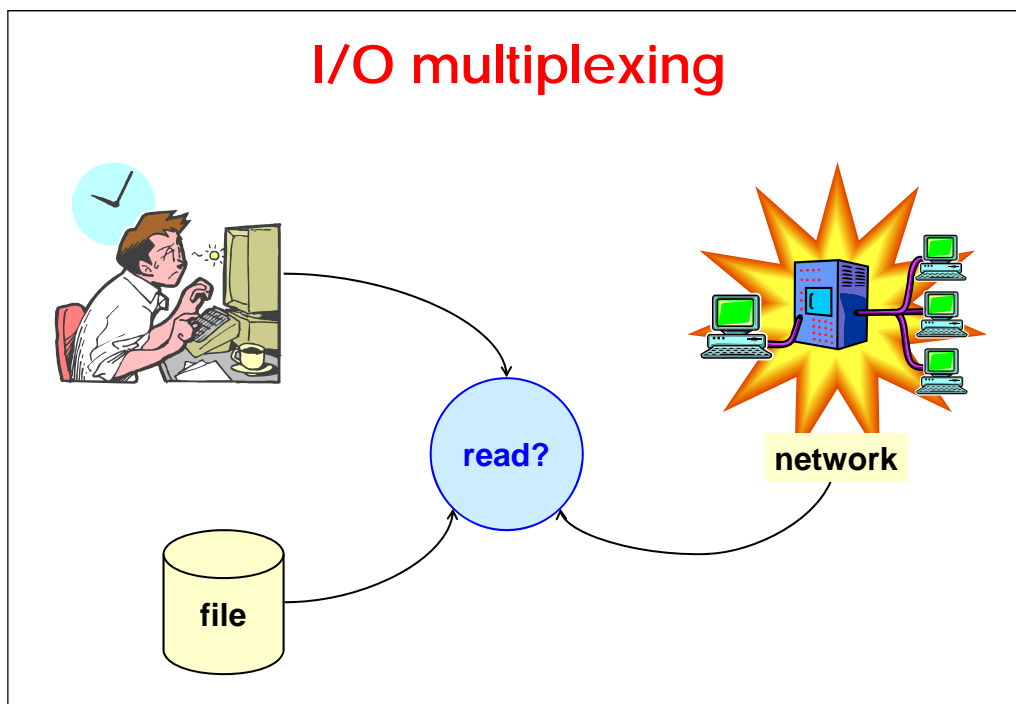
## Shutdown del server

- **allo shutdown di un nodo Unix, il processo init:**
  - invia SIGTERM a tutti processi attivi
  - dopo 5...20 secondi invia SIGKILL
- **SIGTERM può essere intercettato e quindi i server possono tentare di chiudere tutti socket**
- **SIGKILL non può essere intercettato, termina tutti i processi chiudendo tutti i socket aperti**

## Heartbeating

- se si vuole sapere al più presto possibile se il peer è irraggiungibile o guasto occorre attivare un meccanismo di “heartbeating”
- due possibili implementazioni:
  - mediante l'opzione SO\_KEEPALIVE
  - mediante protocollo applicativo di heartbeating

## I/O multiplexing

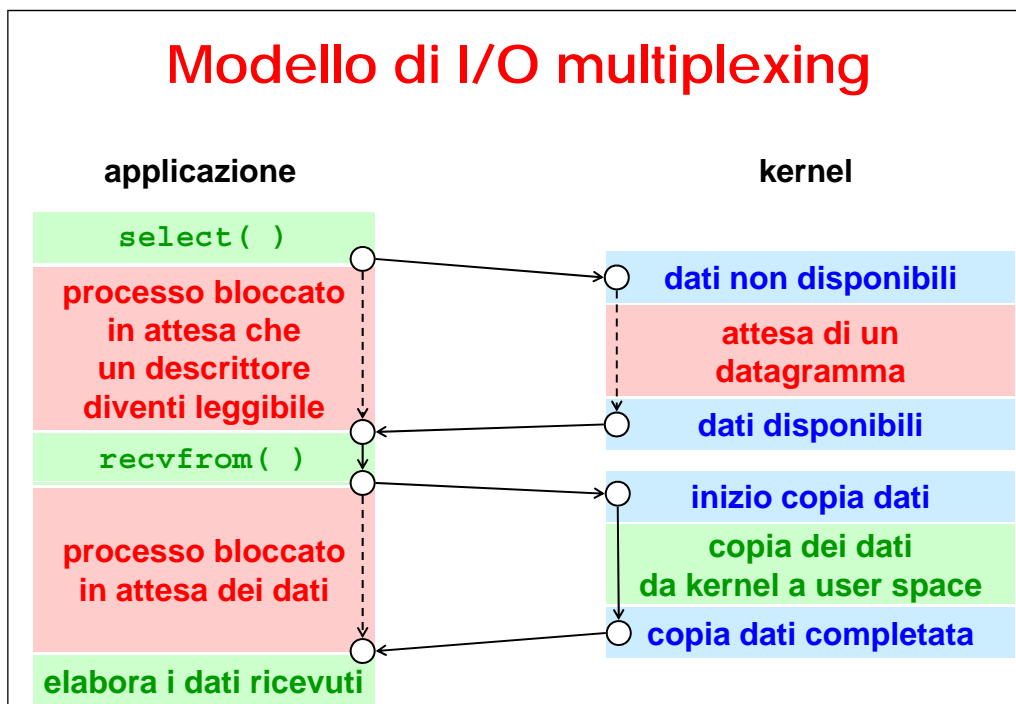


## Applicazioni del multiplexing dell'I/O

- un client che gestisce input da più sorgenti (tipicamente utente da tastiera e server da socket)
- un client che gestisce più socket (raro, ma tipico nei browser web)
- un server TCP che gestisce sia il socket di ascolto sia quelli di connessione (senza attivare processi separati; es. sistemi embedded)
- un server che gestisce sia UDP sia TCP
- un server che gestisce più servizi e/o protocolli (raro, ma tipico del processo inetd)

## Modelli di I/O

- bloccante (read su socket normali)
- non bloccante (read su socket non bloccanti)
- multiplexing (select, poll)
- signal-driven (SIGIO)
- asincrono (funzioni aio\_\*)
  
- tutti attraversano due fasi:
  - attesa che i dati siano pronti
  - copia dei dati da kernel space a user space
- il problema è sempre in lettura, quasi mai in scrittura



## select( )

- **blocca il processo chiamante finché ...**
  - uno dei descrittori selezionati diventa "attivo"
  - oppure scade il timeout (se impostato)
- **maxfdp1 (= MAX FD Plus 1) indica il numero del maggiore descrittore di file da osservare, più uno**
- **restituisce il numero di descrittori attivi, 0 se è terminata per timeout, -1 in caso di errore**

```
#include <sys/select.h>
#include <sys/time.h>
int select (int maxfdp1, fd_set *readset,
            fd_set *writeset, fd_set *exceptset,
            struct timeval *timeout);
```

## Timeout

- attesa infinita: `timeout == NULL`
- tempo massimo di attesa: `timeout != NULL`
- nessuna attesa (cioè polling): `timeout == {0, 0}`
- alcuni sistemi modificano il valore del `timeout`, quindi è meglio re-inizializzarlo ad ogni chiamata

```
#include <sys/time.h>
struct timeval
{
    long tv_sec; // seconds
    long tv_usec; // microseconds
};
```

## fd\_set

- insiemi di flag per selezionare descrittori di file (ossia un “maschera di bit”)
- si opera su di essi con le macro `FD_xxx`
- si usa `FD_ISSET` per sapere su quali descrittori c'è stata attività
- attenzione!!! sono da re-inizializzare ad ogni chiamata

```
#include <sys/select.h>
void FD_ZERO (fd_set *fdset); // azzera la maschera
void FD_SET (int fd, fd_set *fdset); // set(fd)
void FD_CLR (int fd, fd_set *fdset); // reset(fd)
int FD_ISSET (int fd, fd_set *fdset); // test(fd)
```

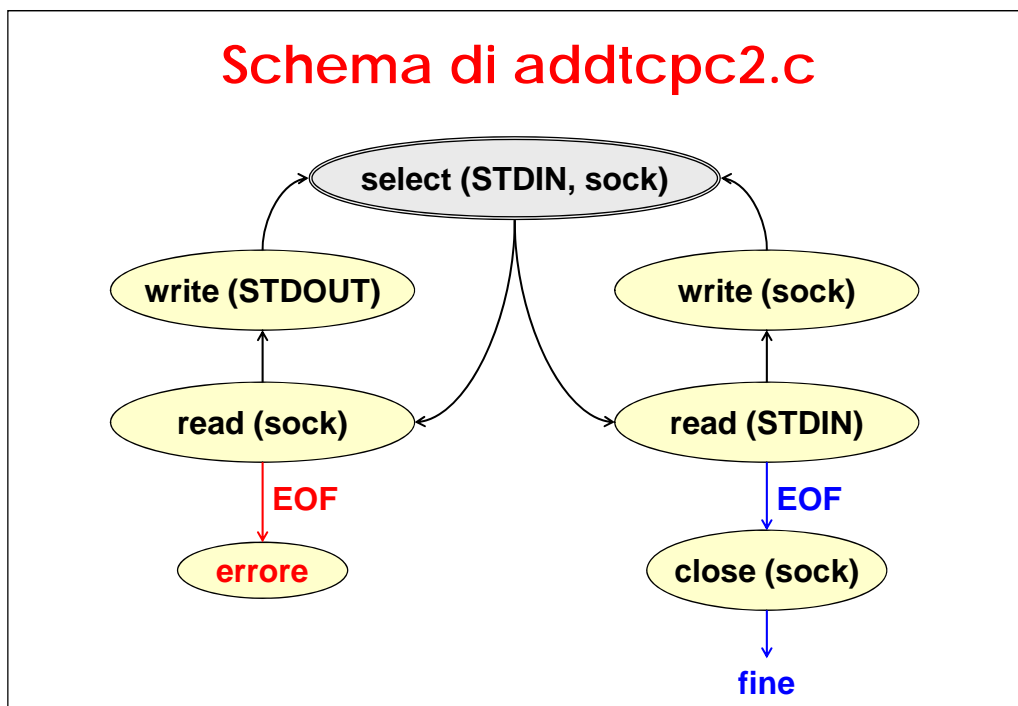
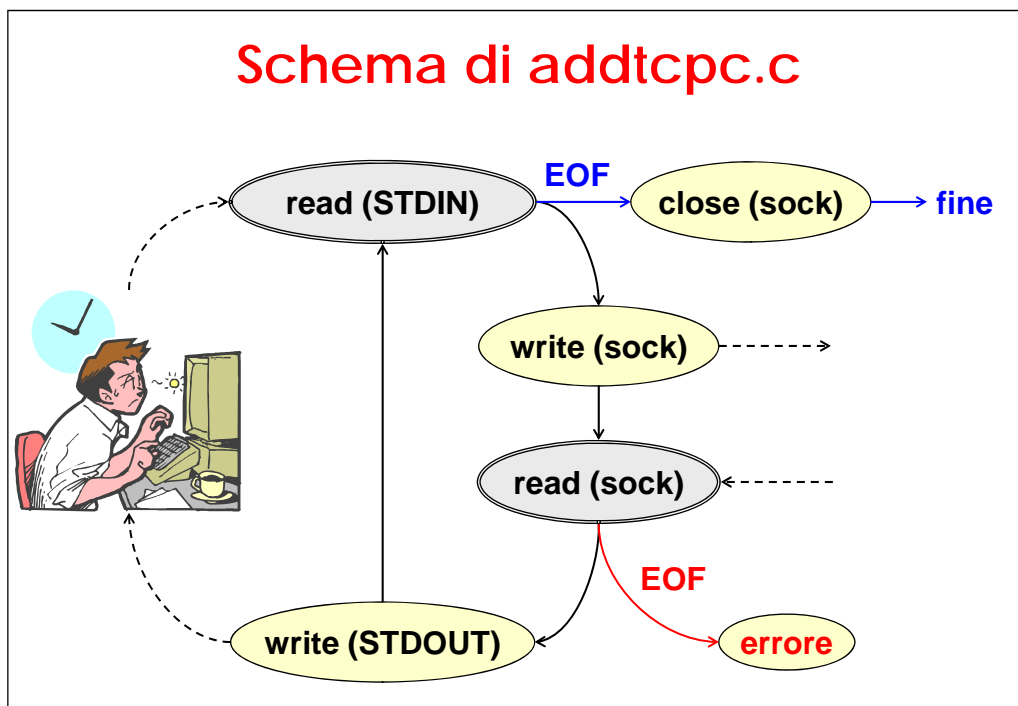
## Quando un descrittore è "pronto"?

- **readset:**
  - ci sono dati da leggere
  - il peer ha chiuso il canale di lettura (ossia EOF)
  - si è verificato errore sul descrittore
  - c'è un nuovo collegamento ad un socket in ascolto
- **writeset:**
  - c'è spazio per scrivere
  - il peer ha chiuso il canale di scrittura (SIGPIPE/EPIPE)
  - si è verificato errore sul descrittore
- **exceptset:**
  - ci sono dati OOB disponibili

## I/O multiplexing: esempio

- **modificare il client addtccp.c in modo da massimizzare il throughput**
- **soluzione:**
  - non alternarsi tra lettura dell'operazione da standard input e lettura della risposta dal socket ma gestire entrambi gli input tramite select( )
  - vedere i diagrammi nelle prossime slide
- **nota: fare il test sia battendo l'input da tastiera, sia ridirigendolo da un file ... si noterà un errore!**

addtccp2.c



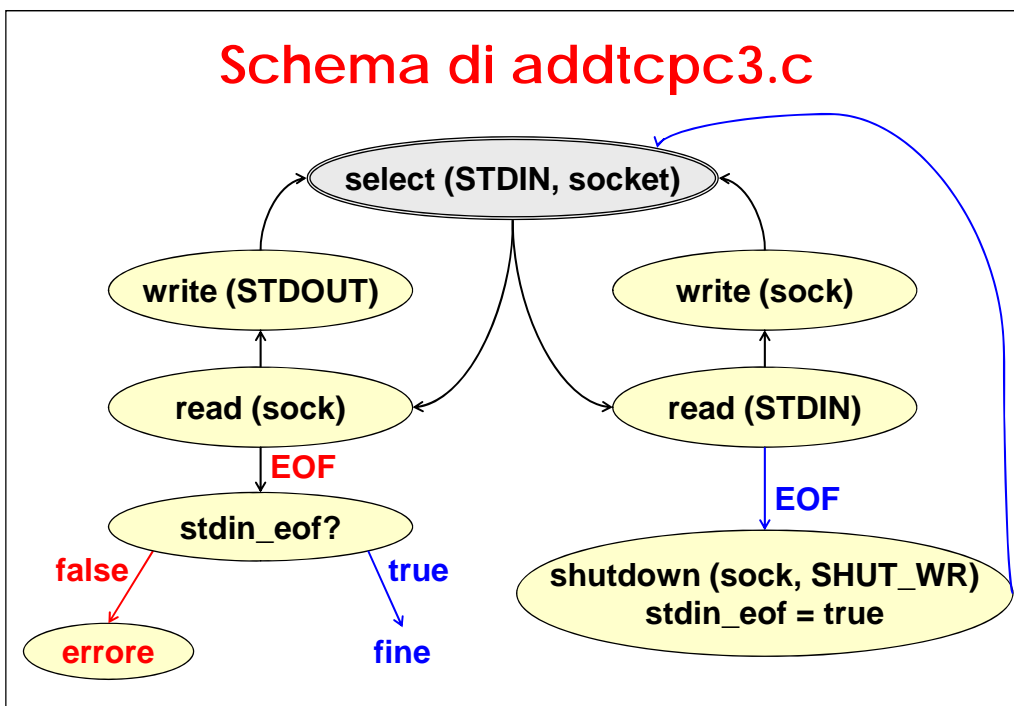


## Batch input

- quando si fornisce input a raffica (come nel caso di lettura da file con un buffer ampio) si rischia di terminare tutto l'input e chiudere il socket senza attendere tutte le risposte
- soluzione: non chiudere il socket completamente (close) ma chiudere solo la parte di scrittura (shutdown), aspettando a chiudere la parte in lettura quando si riceverà EOF

addtcpc3.c

## Schema di addtcpc3.c



## shutdown( )

- chiude uno dei due canali associati ad un socket
- notare che close( ):
  - chiude entrambi i canali (... ma solo se il reference count del descrittore diventa 0)
  - funzionamento esatto dipende dall'opzione LINGER
- valori possibili per howto:
  - SHUT\_RD (oppure 0)
  - SHUT\_WR (oppure 1)
  - SHUT\_RDWR (oppure 2)

```
#include <sys/socket.h>
int shutdown (int sockfd, int howto);
```

## Funzionamento di shutdown( )

- shutdown ( sd, SHUT\_RD)
  - read impossibile sul socket
  - contenuto del buffer di read viene eliminato
  - altri dati ricevuti in futuro vengono scartati direttamente dallo stack
- shutdown ( sd, SHUT\_WR)
  - write impossibile sul socket
  - contenuto del buffer di write spedito al destinatario, seguito da FIN se socket stream

## Opzioni dei socket



## getsockopt( ) e setsockopt( )

- applicabili solo a socket aperti
- “level” indica il livello dello stack di rete:  
SOL\_SOCKET, IPPROTO\_IP, IPPROTO\_TCP, ...
- valori mnemonici per “optname”

```
#include <sys/socket.h>
#include <netinet/tcp.h>
int getsockopt (int sockfd, int level, int optname,
               void *optval, socklen_t *optlen);
int setsockopt (int sockfd, int level, int optname,
               const void *optval, socklen_t optlen);
```

## Alcune opzioni a livello SOCKET

<i>level</i>	<i>optname</i>	<i>get</i>	<i>set</i>	<i>tipo</i>
SOL_SOCKET	SO_BROADCAST	X	X	int (boolean)
	SO_DEBUG	X	X	int (boolean)
	SO_DONTROUTE	X	X	int (boolean)
	SO_ERROR	X		int
	SO_KEEPAIVE	X	X	int (boolean)
	SO_LINGER	X	X	struct linger
	SO_OOINLINE	X	X	int (boolean)
	SO_RCVBUF	X	X	int
	SO_SNDBUF	X	X	int
	SO_RCVTIMEO	X	X	struct timeval
	SO_SNDTIMEO	X	X	struct timeval
	SO_REUSEADDR	X	X	int (boolean)
	SO_REUSEPORT	X	X	int (boolean)
	SO_TYPE	X		int

## Alcune opzioni a livello IP e TCP

<i>level</i>	<i>optname</i>	<i>get</i>	<i>set</i>	<i>tipo</i>
IPPROTO_IP	IP_OPTIONS	X	X	
	IP_TOS	X	X	int
	IP_TTL	X	X	int
	IP_RECVDSTADDR	X	X	int
IPPROTO_TCP	TCP_MAXSEG	X	X	int
	TCP_NODELAY	X	X	int
	TCP_KEEPAIVE	X	X	int

## Esempio di lettura opzioni dei socket

- da UNP, sezione 7.3
- note: in CYGWIN i timeout sono interi

checkopts.c

## Broadcast, Keepalive, buffer

- **SO\_BROADCAST**
  - applicabile solo a datagram socket
  - abilita l'uso di indirizzi broadcast
- **SO\_KEEPALIVE**
  - scambio di un pacchetto di "probe" ogni 2 ore (!)
  - intervento a livello kernel per cambiarne il valore
- **SO\_SNDBUF, SO\_RCVBUF**
  - dimensioni dei buffer locali; impostare prima di connect (per il client) e di listen (per il server)
  - valore  $\geq 3 \times \text{MSS}$
  - valore  $\geq \text{banda} \times \text{RTT}$

## SO\_SNDTIMEO, SO\_RCVTIMEO

- Posix specifica i timeout con una struct timeval
- in precedenza erano solo un intero
- si applicano solo a:
  - read, readv, recv, recvfrom, recvmsg
  - write, writev, send, sendto, sendmsg
- usare altre tecniche per accept, connect, ...

```
#include <sys/time.h>
struct timeval
{
    long tv_sec;
    long tv_usec;
};
```

## SO\_REUSEADDR ( SO\_REUSEPORT )

- SO\_REUSEADDR permette:
  - di fare bind a porta locale occupata da un processo (un socket di connessione ed uno di ascolto)
  - di avere più server sulla stessa porta (es. IP<sub>A</sub>, IP<sub>B</sub>, INADDR\_ANY)
  - di avere più socket di un singolo processo sulla stessa porta ma con indirizzi locali diversi (utile per UDP senza IP\_RECVDSTADDR)
  - di avere più socket multicast sulla stessa porta (su taluni sistemi si usa SO\_REUSEPORT)
- opzione altamente consigliata per tutti i server TCP

## SO\_LINGER

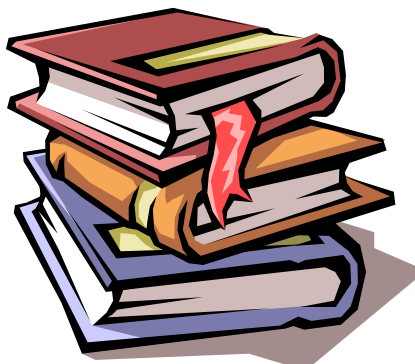
- cambia il comportamento della close( )
- OFF: close non bloccante (ma tenta di inviare i restanti dati)
- ON + l\_linger == 0: close non bloccante e abort della connessione (=RST, non FIN + TIME\_WAIT)
- ON + l\_linger > 0: close bloccante (tenta di inviare i restanti dati sino al timeout; se non ci riesce, ritorna EWOULDBLOCK)

```
#include <sys/socket.h>
struct linger
{
    int l_onoff; // 0=off, non-zero=on
    int l_linger; // linger timeout (seconds in Posix)
};
```

## IP\_TOS, IP\_TTL

- leggono o impostano il valore di TOS e TTL per i pacchetti in uscita
- valori per TOS:
  - IPTOS\_LOWDELAY
  - IPTOS\_THROUGHPUT
  - IPTOS\_RELIABILITY
  - IPTOS\_LOWCOST ( IPTOS\_MINCOST )
- valori che dovrebbero essere i default per TTL:
  - 64 per UDP e TCP (RFC-1700)
  - 255 per socket RAW

## Librerie di supporto



## Network libraries

- **libpcap**
  - cattura di pacchetti
  - <http://www.tcpdump.org>
  - solo per Unix; per Windows vedere winpcap
- **libdnet**
  - raw IP, raw Ethernet, arp, route, fw, if, indirizzi
  - <http://libdnet.sourceforge.net>
  - per Unix e Windows
- **libnet**
  - <http://www.packetfactory.net/projects/libnet/>



## Event libraries

- **libevent**
  - <http://monkey.org/~provos/libevent/>
- **liboop**
  - <http://liboop.org/>

## Gestione di nomi e indirizzi

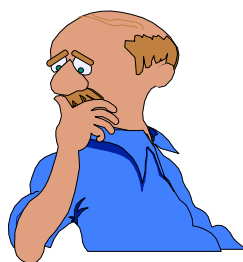
www.polito.it = ???

mail.x.net : smtp = ???

5f1b:df00:ce3e:e200:20:800:2078:e3e3 = ???

tcp / 43 = ???

192.168.1.1 = ???



## Formati mnemonici e numerici

- **conversione tra nomi mnemonici e valori numerici (nodi, servizi, reti, protocolli, indirizzi di rete)**
- **attenzione! dipendono dall'impostazione locale del sistema**
  - file locali (es. /etc/hosts, /etc/services)
  - lookup service di LAN (es. NIS, LDAP)
  - lookup service globale (DNS)
- **solo nel caso del DNS si può (con qualche sforzo) puntare esplicitamente ad questo servizio**

## gethostbyname()

- **restituisce una struttura dati con la descrizione del nodo il cui nome è specificato come argomento**
- **in caso di errore restituisce NULL e setta la variabile h\_errno per specificare l'errore, di cui si può avere una rappresentazione testuale tramite hstrerror(h\_errno)**

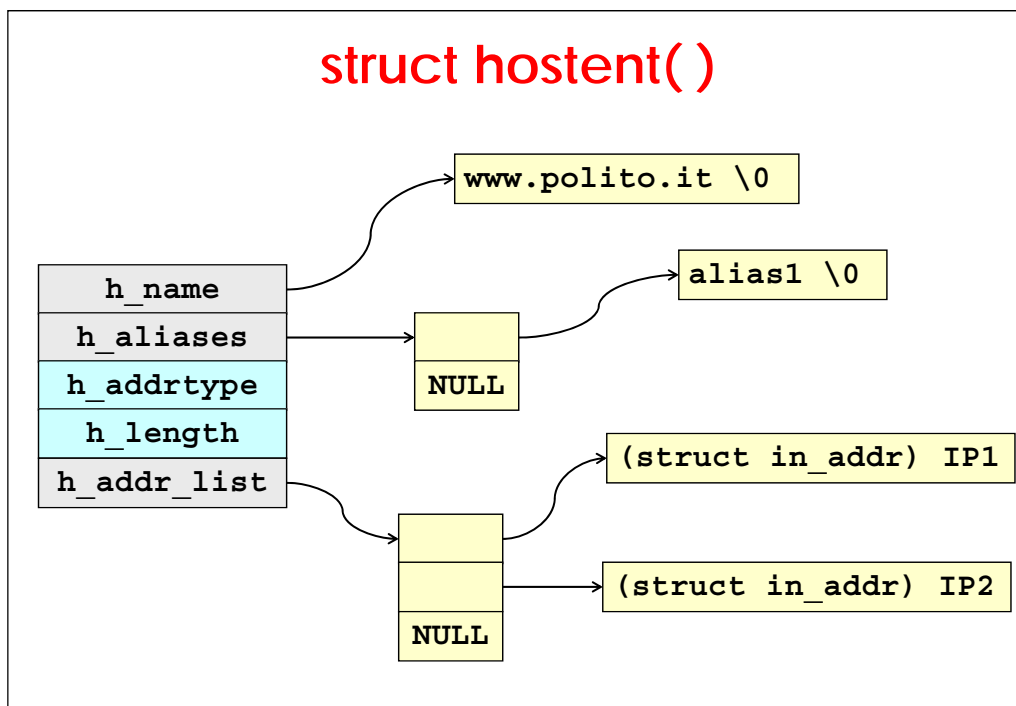
```
#include <netdb.h>
struct hostent *gethostbyname (
    const char *hostname );
extern int h_errno;
char *hstrerror (int h_errno);
```

## struct hostent()

- nota: indirizzi rappresentati come 1 char = 1 byte (meglio: 1 unsigned char)

```
#include <netdb.h>
struct hostent {
    char *h_name; // canonical name
    char **h_aliases; // array of alias names
    int h_addr_type; // AF_INET or AF_INET6
    int h_length; // address length (4 or 16 bytes)
    char **h_addr_list; // array of addresses
};
#define h_addr h_addr_list[0] // BSD compatibility
```

## struct hostent()



## gethostbyaddr()

- restituisce una struttura dati con la descrizione del nodo il cui indirizzo è specificato come argomento
- in caso di errore restituisce NULL e setta la variabile `h_errno` per specificare l'errore, di cui si può avere una rappresentazione testuale tramite `hstrerror(h_errno)`
- nota: l'argomento `addr` in realtà è un puntatore alla struct `in_addr` o `in_addr6`

```
#include <netdb.h>
struct hostent *gethostbyaddr (
    const char *addr, size_t len, int family );
```

## uname()

- identifica il nodo su cui il programma è in esecuzione
- dimensioni e contenuto delle stringhe dipendono dal sistema e dalla sua configurazione sistemistica
- restituisce intero negativo in caso di errore

```
#include <sys/utsname.h>
struct utsname {
    char sysname[...]; // OS name
    char nodename[...]; // network node name
    char release[...]; // OS release
    char version[...]; // OS version
    char machine[...]; // CPU type
};
int uname (struct utsname *nameptr);
```

## Esempi

- (info\_from\_n.c) programma per fornire tutte le informazioni disponibili su un nodo dato il suo nome
- (info\_from\_a.c) programma per fornire tutte le informazioni disponibili su un nodo dato un suo indirizzo
- (myself.c) programma per fornire tutte le informazioni disponibili sul nodo dove il programma sta eseguendo

info\_from\_n.c

info\_from\_a.c

myself.c

## getservbyname(), getservbyport()

- restituiscono informazioni sul servizio di cui è passato come argomento il nome o la porta
- restituiscono NULL in caso di errore
- attenzione: i numeri di porta sono in network order (OK per programmare, non per visualizzarli)

```
#include <netdb.h>
struct servent *getservbyname (
    const char *servname, const char *protoname );
struct servent *getservbyport (
    int port, const char *protoname );
```

## struct servent( )

- nota: errore su UNP (p.251) che dichiara int s\_port

```
#include <netdb.h>
struct servent {
    char *s_name; // official service name
    char **s_aliases; // array of aliases
    short s_port; // port number (network order)
    char *s_proto; // transport protocol
};
```

service.c

## I "demoni"



## Che cos'è un demone?

- un processo autonomo
- svolge tipicamente il ruolo di un server
- attivato automaticamente al boot
- staccato da qualunque terminale
- scrive informazioni in un file di log
- si configura tramite le informazioni passate sulla riga di comando (sconsigliato!) o contenute in un file di configurazione
- esegue coi diritti di un certo utente e gruppo
- lavora all'interno di un certo direttorio

## syslog()

- non è standard Posix ma è obbligatoria per Unix98
- permette di generare dati nel log di sistema
- destinazione effettiva dei dati dipende dalla configurazione di syslogd (es. /etc/syslog.conf)
- comunicazione aperta in modo implicito o esplicito (openlog + closelog)

```
#include <syslog.h>
void syslog (
    int priority, const char *message , ...);
void openlog (
    const char *ident, int options, int facility);
void closelog (void);
```

## syslog priority = level | facility

<b>LEVEL</b>	LOG_EMERG	(priorità massima)
	LOG_ALERT	
	LOG_CRIT	
	LOG_ERR	
	LOG_WARNING	
	LOG_NOTICE	(priorità di default)
	LOG_INFO	
	LOG_DEBUG	(priorità minima)

<b>FACILITY</b>	LOG_AUTH	LOG_AUTHPRIV	LOG_CRON
	LOG_DAEMON	LOG_FTP	LOG_KERN
	LOG_LOCAL0	. . .	LOG_LOCAL7
	LOG_LPR	LOG_MAIL	LOG_NEWS
	LOG_SYSLOG	LOG_USER	LOG_UUCP

## syslog options

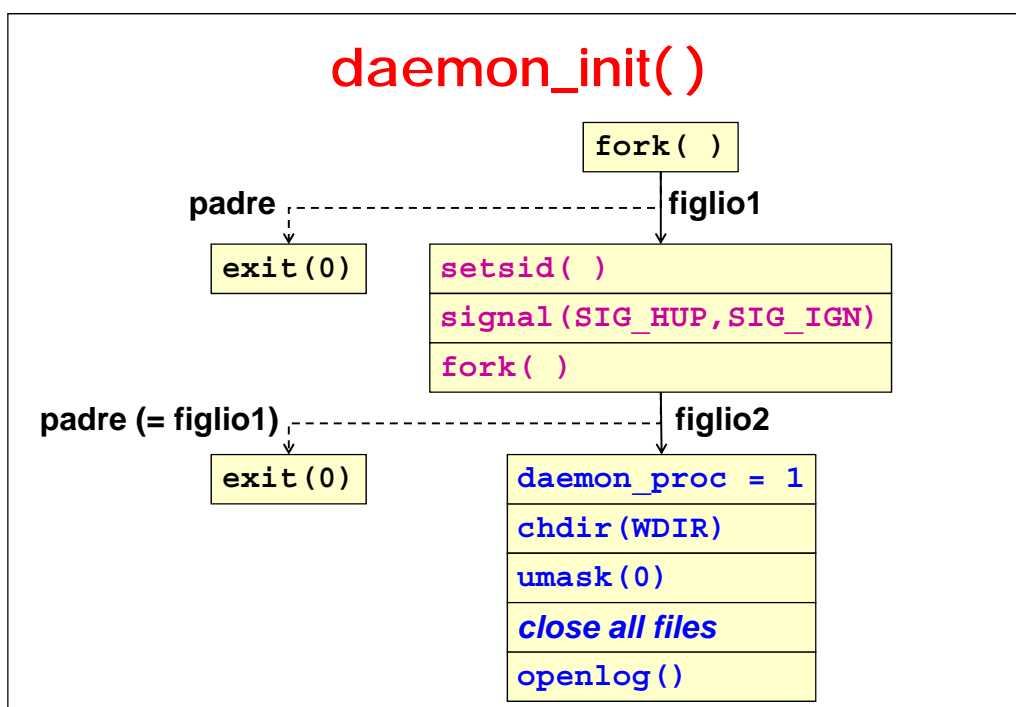
<b>OPTIONS</b>	LOG_CONS	log su console se la comunicazione con syslog fallisce
	LOG_NDELAY	apertura immediata del socket, senza attendere la prima chiamata di syslog( )
	LOG_PERROR	log anche su stderr
	LOG_PID	inserire nel log anche il PID



## Inizializzazione di un demone

- disassociarsi dal terminale di controllo (diventa immune a HUP, INT, WINCH che possono essere usati per altre segnalazioni)
- spostarsi nel direttorio di lavoro
- chiudere tutti i file ereditati
- opzionale, per precauzione verso librerie aliene:
  - aprire /dev/null ed associarlo a stdin, stdout, stderr
  - aprire un file di log ed associarlo a stdout e stderr
- aprire syslog
  
- per non sbagliare l'inizializzazione di un demone, usare la funzione `daemon_init( )` (UNP, p.336)

## daemon\_init( )



## Segnali verso un demone

- **convenzioni molto usate:**
  - SIGHUP per far rileggere il file di configurazione
  - SIGINT per far terminare il demone

## inetd

- **se un nodo di rete offre tanti servizi, deve avere tanti demoni in attesa, ognuno dei quali è un processo ed ha del codice associato**
- **per semplificare tutto questo, in Unix si usa spesso il super-server “inetd”**
  - conosce i servizi specificati in /etc/inetd.conf
  - si mette in ascolto su tutti i socket corrispondenti
  - quando riceve una richiesta di collegamento, attiva il server corrispondente

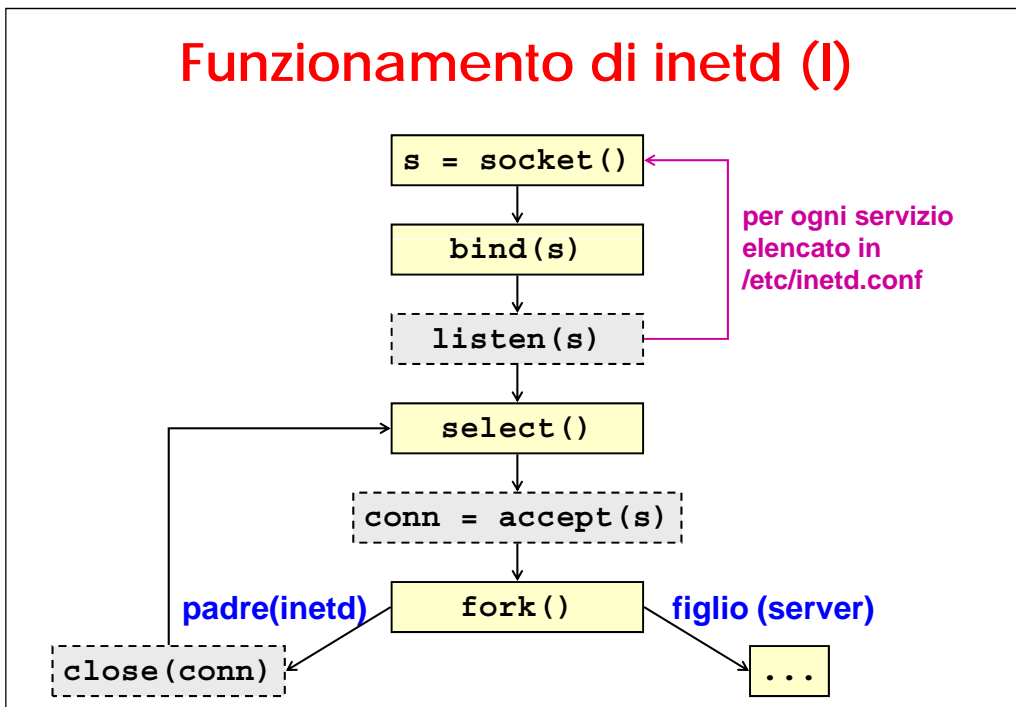
## Formato di /etc/inetd.conf

- ogni riga contiene da 7 a 11 campi:
  - servizio - nome del servizio in /etc/services
  - tipo di socket (`stream`, `dgram`)
  - protocollo (`tcp`, `udp`)
  - flag (`wait`, `nowait`) - server iterativo o concorrente
  - login name (entry in /etc/passwd) - nome utente
  - server program (pathname, `internal`)
  - argomenti - massimo 5, incluso `argv[0]`

## Esempio di /etc/inetd.conf

```
. . .
echo    stream  tcp  nowait  root    internal
echo    dgram  udp  wait    root    internal
ftp     stream  tcp  nowait  root    /usr/etc/ftpd ftpd
telnet  stream  tcp  nowait  root    /usr/etc/telnetd telnetd
login   stream  tcp  nowait  root    /etc/rlogind rlogind
tftp    dgram  udp  wait    nobody  /usr/etc/tftpd tftpd
talk    dgram  udp  wait    root    /etc/talkd talkd
spawn   stream  tcp  nowait  lioy    /usr/local/spawner spawner
. . .
```

## Funzionamento di inetd (I)

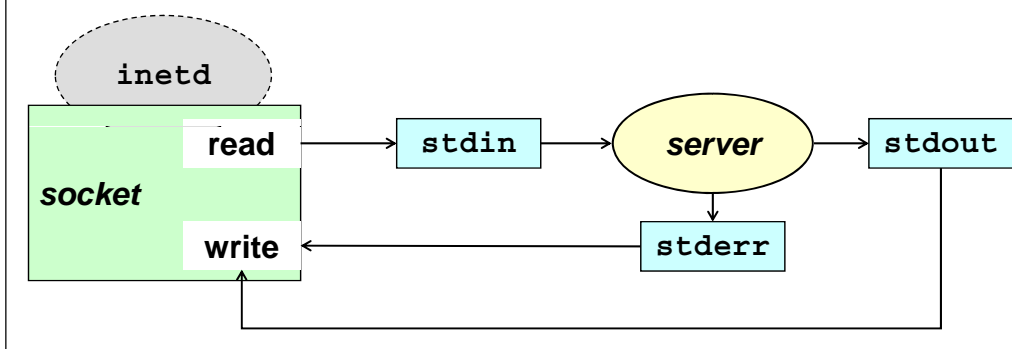


## Funzionamento di inetd (II)



## Server attivati da inetd

- **basta un comune “filtro” che:**
  - legga i dati da stdin (descrittore 0)
  - generi i risultati su stdout / stderr (descrittori 1 / 2)
- **problema con dati non ASCII**



## Altre soluzioni per il super-server

- **tcpd è un sistema per attivare in modo sicuro un server da inetd**
- **xinetd è una versione migliorata di inetd che incorpora già le funzionalità di tcpd**
- **tcpserver è un rimpiazzo completo di inetd+tcpd (ma non è un super-server)**

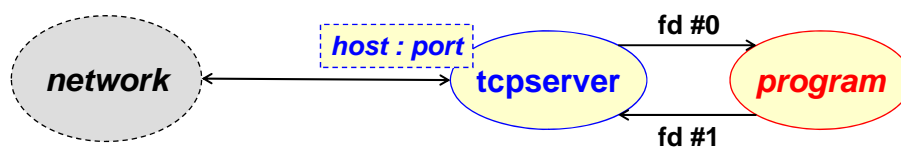
## tcpserver

- opera di D.J.Bernstein (<http://cr.yp.to>)
- è un filtro, in ascolto su una porta TCP
- una copia per ogni servizio da proteggere
  - copie indipendenti = efficienza
  - piccolo = più facile da verificare
- flessibilità, sicurezza e modularità
- meticolosa attenzione ai permessi e alle restrizioni
- controllo sul numero di processi concorrenti
- controllo d'accesso (indirizzi IP, nomi DNS)
- UID e GID impostabili

## tcpserver: attivazione

- tcpserver rimane in ascolto su una porta (port) di un'interfaccia (host) ed esegue un'applicazione (program) per ogni richiesta ricevuta ...
- ... se sono superati i controlli eventualmente specificati (-x)

```
tcpserver [ -qQvdDoOpPhHrRl ] [ -climit ]
[ -bbacklog ] [ -xrules.cdb ] [ -ggid ]
[ -uuid ] [ -llocalname ] [ -ttimeout ]
host port program [ arg ... ]
```



## tcpserver: principali opzioni

- **-cn** (massimo  $n$  processi simultanei)
- **-xr.cdb** (controlla le regole d'accesso in *r.cdb*)
- **-ggid** (imposta il group ID)
- **-userid** (imposta lo user ID)
- **-bn** (consente un backlog di  $n$  TCP SYN)

## tcpserver: regole di accesso

- **definire le regole secondo il seguente formato:**
  - **utente@indirizzo:lista\_di\_istruzioni**
  - **lista\_di\_istruzioni ::= deny | allow , var. di ambiente**
- **compilare con tcprules le regole passate, creando un file .CDB (struttura dati hash)**

```
tcprules regole.cdb regole.tmp < regole.txt
```

```
(regole.txt)
130.192.:allow
192.168:allow,CLIENT="private"
:deny
```