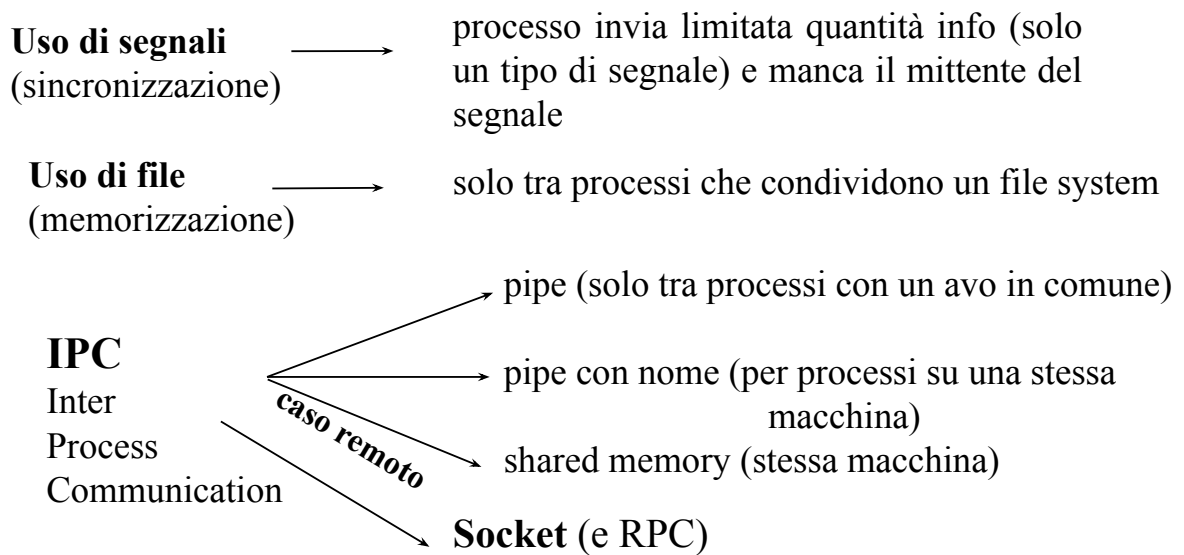


Unix: strumenti di sincronizzazione, memorizzazione, comunicazione

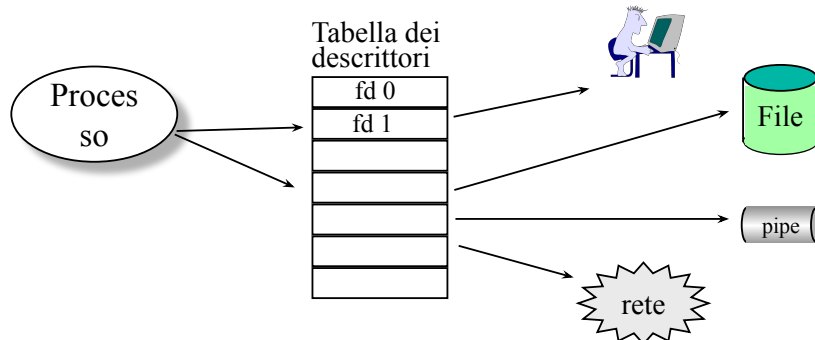


Socket in UNIX - 1

I Processi e l'I/O

I processi interagiscono con l'I/O secondo il paradigma *open-read-write-close*

Un processo vede il mondo esterno come un insieme di descrittori



Flessibilità (possibilità di pipe e ridirezione)

Anche le **Socket** sono identificate da un **descrittore** (socket descriptor).
Stessa semantica delle pipe (es. read blocca in attesa dati)

Socket in UNIX - 2

Programmazione di rete e paradigma *open-read-write-close*

La programmazione di rete richiede delle funzionalità non gestibili in modo completamente omogeneo alle pipe (e ai file):

- un collegamento via rete, cioè l'associazione delle due parti comunicanti può essere
 - **con connessione** (simile o-r-w-c)
 - **senza connessione**
- i descrittori: trasparenza dai nomi va bene nel caso file (flessibilità), ma nel caso di network programming può non essere sufficientemente espressivo
- in programmazione di rete bisogna specificare più parametri per definire un collegamento con connessione
<protocollo; indirizzo locale; processo locale; indirizzo remoto; processo remoto>

Socket in UNIX - 3

Programmazione di rete e paradigma *open-read-write-close*

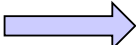
Altre problematiche:

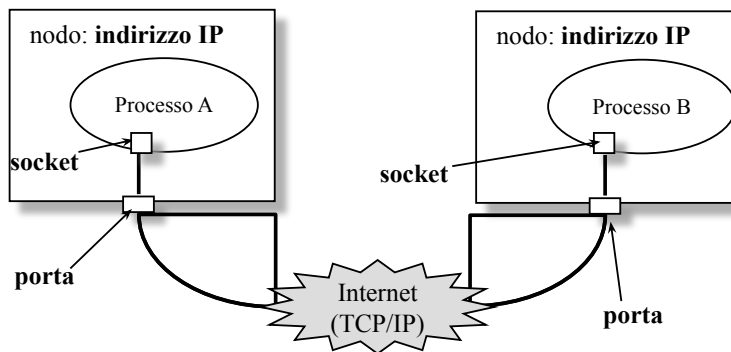
- UNIX I/O è orientato allo stream, non a messaggi di dimensioni prefissate → Alcuni protocolli di rete fanno uso di messaggi di dimensioni prefissate
- è necessario che l'interfaccia socket possa gestire più protocolli
- schemi Client/Server sono asimmetrici (si ricordi che le pipe sono simmetriche)

Socket in UNIX - 4

L'interfaccia Socket (Obiettivi)

- Comunicazione fra processi su nodi diversi
- Interfaccia indipendente da architettura di rete
- Supportare diversi protocolli, diversi hardware, diversi tipi nomi, etc.

UNIX + TCP-UDP/IP  BSD UNIX (Progetto Università Berkeley finanziato da ARPA, ma ora Socket disponibili su tutte versioni Unix, Windows, etc)



Il **canale di comunicazione** tra il processo A e il processo B è definito da:
<protocollo; indirizzo IP locale; porta locale; indirizzo IP remoto; porta remota>

Socket in UNIX - 5

Socket

Una socket è un'**astrazione** che definisce il terminale di un **canale di comunicazione bidirezionale**.

Uso di descrittori per le socket, ma **diverse operazioni per avere diverse semantiche**

Tabella dei
descrittori

| |
|----------------------|
| fd 0 |
| fd 1 |
| |
| socket descriptor |
| |
| |

struttura dati socket

| |
|-----------------|
| family: AF_INET |
| service: |
| local IP: |
| remote IP: |
| local port: |
| remote port: |
| pgid: |
| |

In Linux la struttura dati di riferimento per la gestione di socket è `struct socket`:
<https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/include/net/sock.h>

Socket in UNIX - 6

Strutture dati Socket

Una socket è creata all'interno di un dominio di comunicazione

Dominio di comunicazione:

semantica di comunicazione + standard di denominazione

Esempi domini: **AF_INET** (IPv4), **AF_INET6** (IPv6 con compatibilità IPv4 ove possibile), **AF_UNSPEC** (IPv4 e IPv6), **AF_UNIX** (Unix socket), etc.



Socket in UNIX - 7

Formato indirizzi e struttura socket

DOMINIO AF_INET : comunicazioni Internet IPv4-only.

Indirizzo composto da indirizzo IPv4 dell'host (32 bit) e da numero di porta (16 bit).

DOMINIO AF_INET6 : comunicazioni Internet IPv6-enabled.

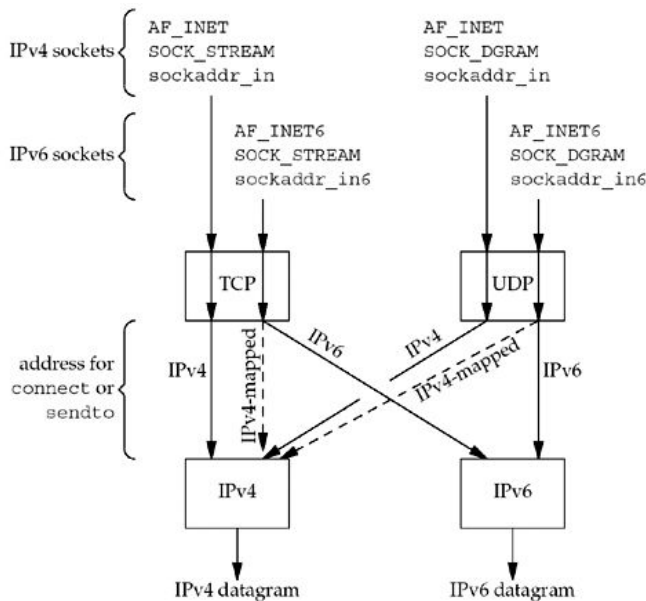
Indirizzo Internet composto da indirizzo IPv6 dell'host (128 bit) e da numero di porta (16 bit). API retrocompatibile con IPv4 attraverso uso di indirizzi IPv6 speciali denominati V4-mapped (es. ::127.0.0.1, ::192.168.0.1, ecc.).

DOMINIO AF_UNIX : comunicazioni (à la FIFO) tra processi UNIX. L'indirizzo ha stesso formato di nome di file.

In ciascun dominio abbiamo bisogno di una struttura dati specifica per rappresentare gli indirizzi delle socket.

Socket in UNIX - 8

Retrocompatibilità



Le socket AF_INET6 sono state progettate per proporre una API retrocompatibile con IPv4 attraverso uso di indirizzi IPv6 speciali denominati V4-mapped (es. ::127.0.0.1, ::192.168.0.1, ecc.).

Socket in UNIX - 9

Retrocompatibilità

Alcuni sistemi non offrono retrocompatibilità (es. in OpenBSD per motivi di sicurezza gli stack IPv4 e IPv6 sono completamente separati, in versioni vecchie di Windows lo stack IPv6 era separato perché - pare - preso da FreeBSD e quindi difficile da integrare con lo stack IPv4 esistente). **In tali sistemi non è quindi possibile catturare richieste di connessioni e traffico IPv4 da una socket AF_INET6.**

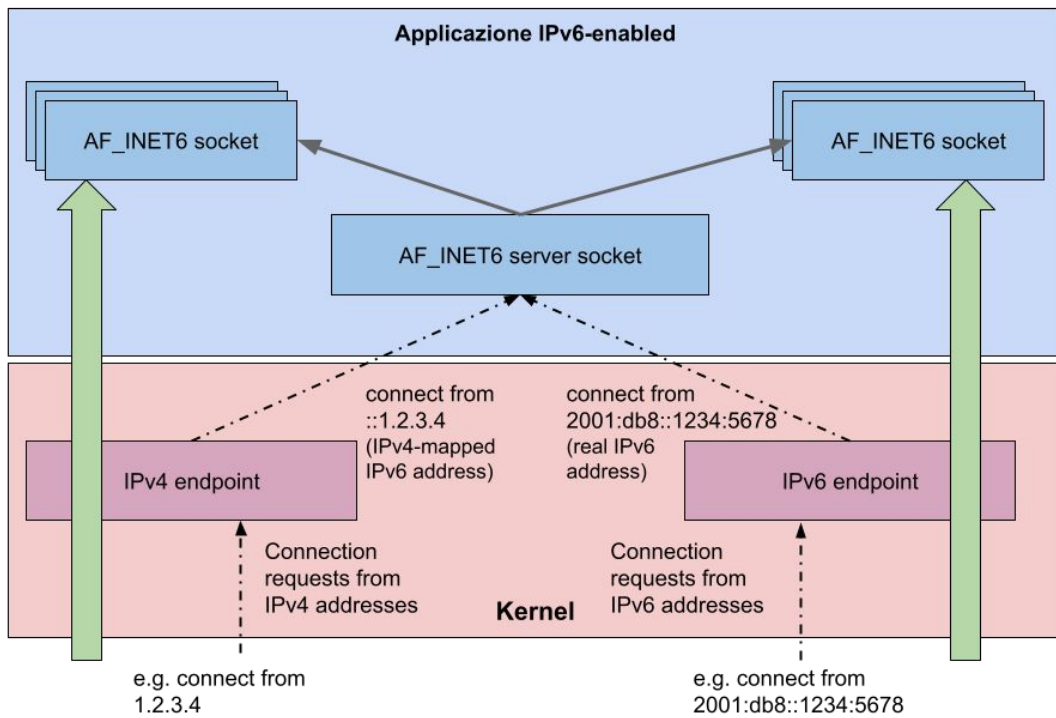
Problema: come gestire richieste di connessione contemporaneamente su più socket? Necessità adottare un'architettura significativamente più complicata (due socket passive, una per IPv4 e una per IPv6 e uso di select o poll prima di accept per attendere simultaneamente connessioni su entrambe le socket).

Soluzione complessa che non considereremo nel corso; si veda RFC 4038.

È anche possibile disabilitare esplicitamente il supporto alla retrocompatibilità tramite l'opzione IPV6_V6ONLY di una socket AF_INET6.

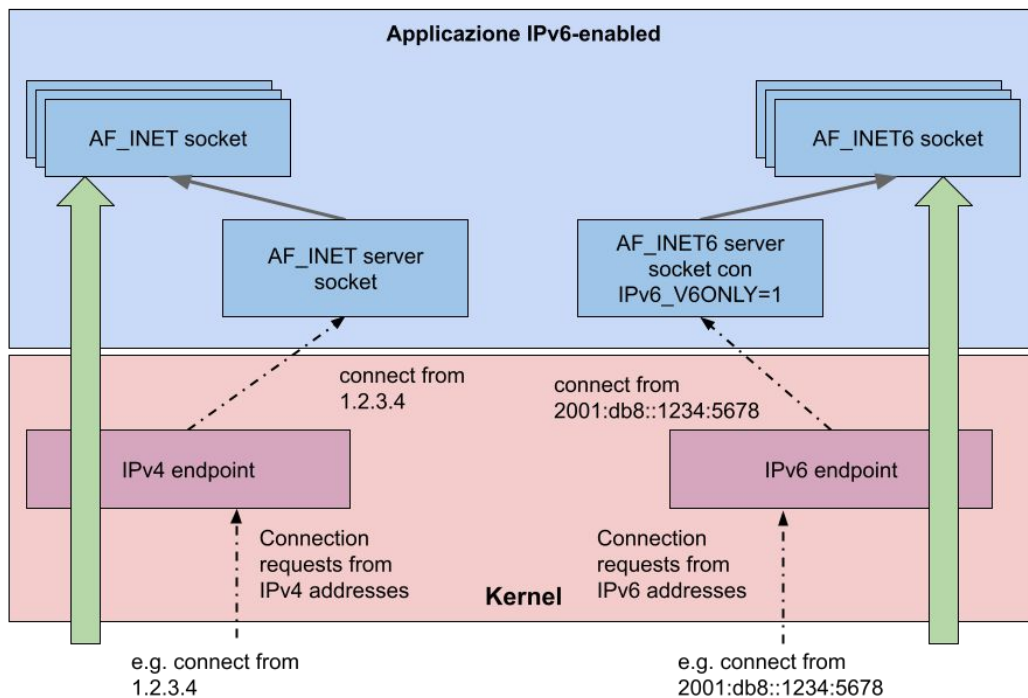
Socket in UNIX - 10

Comportamento con retrocompatibilità (default)



Socket in UNIX - 11

Comportamento con retrocompatibilità disabilitata



Socket in UNIX - 12

Formato indirizzi nel Dominio AF_INET

struttura in_addr struct in_addr {
 in_addr_t s_addr;
};

struttura sockaddr_in struct sockaddr_in {
 sa_family_t sin_family;
 in_port_t sin_port;
 struct in_addr sin_addr;
 char sin_zero [8] ;
};

Formato indirizzi nel Dominio AF_INET6

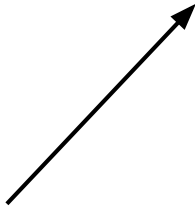
struttura in6_addr struct in6_addr {
 uint8_t s6_addr[16];
};

struttura sockaddr_in6 struct sockaddr_in6 {
 sa_family_t sin6_family;
 in_port_t sin6_port;
 uint32_t sin6_flowinfo;
 struct in6_addr sin6_addr;
 uint32_t sin6_scope_id;
};

Formato indirizzi nel Dominio AF_UNIX

struttura sockaddr_un

```
struct sockaddr_un {  
    sa_family_t sun_family;  
    char sun_path[108];  
};
```



Nome di file nel filesystem
corrispondente alla socket

Socket in UNIX - 15

Formato indirizzi e struttura socket

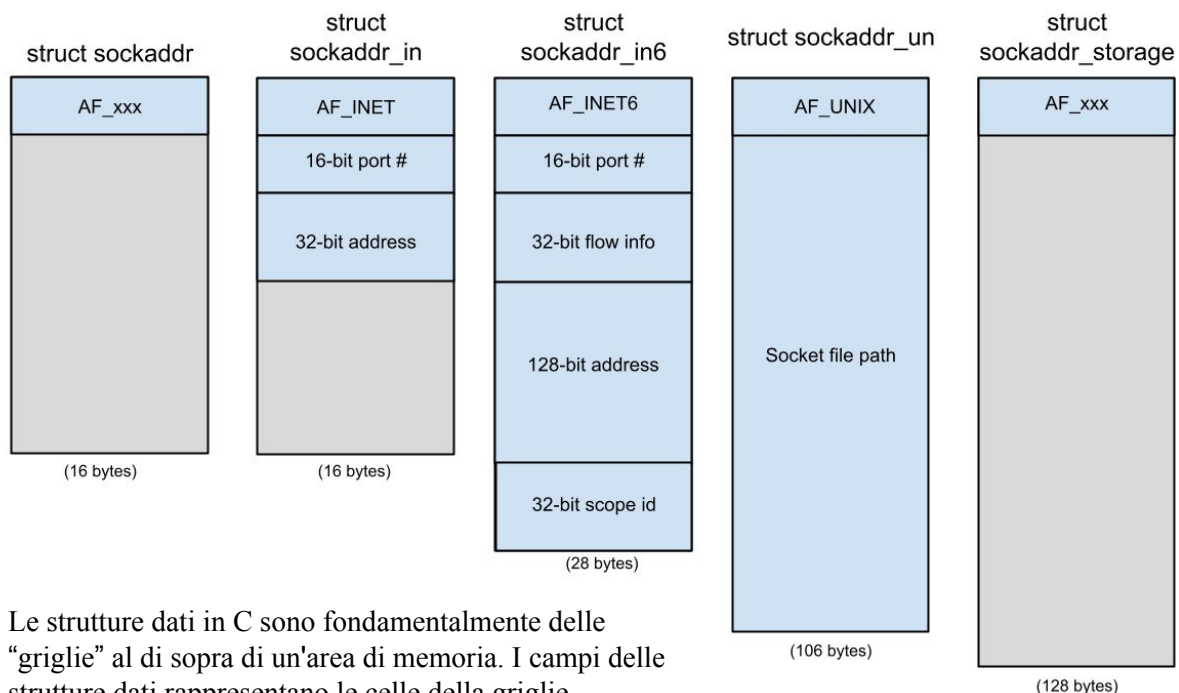
Le funzioni per la programmazione di rete nella Socket API (es. `bind`, `connect`) possono ricevere diversi tipi di indirizzo a seconda della famiglia di protocolli scelta (es. `AF_UNIX`, `AF_INET`, o `AF_INET6`)

Uso di due strutture dati generiche: `struct sockaddr` e `struct sockaddr_storage`, e poi cast al tipo specifico (a seconda della famiglia):

- `sockaddr` come interfaccia comune per tutti i diversi possibili tipi di indirizzi (una sorta di interface in Java)
- `sockaddr_storage` per contenere i diversi possibili tipi di indirizzi

```
struct sockaddr {  
    sa_family_t sa_family;  
    char sa_data[14];  
}  
  
struct sockaddr_storage {  
    sa_family_t ss_family;  
    char ss_padding[SIZE];  
}
```

Socket in UNIX - 16



Le strutture dati in C sono fondamentalmente delle “griglie” al di sopra di un'area di memoria. I campi delle strutture dati rappresentano le celle della griglia.

La `struct sockaddr` definisce la parte di griglia che tutte le varie strutture `sockaddr_*` hanno in comune.

Socket in UNIX - 17

Per ulteriori informazioni sulle struct in C

A chi di voi fosse un po' «arrugginito» sull'uso delle *struct* nel linguaggio C, o semplicemente volesse approfondire l'argomento, consiglio caldamente la lettura di:

- R. Reese, «Understanding and Using C Pointers», O'Reilly, 2013 (disponibile in biblioteca);
- E. Raymond, «The Lost Art of C Structure Packing», available at: <http://www.catb.org/esr/structure-packing/>.
- R. Seacord, «Effective C», No Starch, 2020 (disponibile in biblioteca);

Il libro di Reese è anche un ottimo riferimento per ripassare e/o approfondire l'uso dei puntatori e delle stringhe in C, che sono argomenti di importanza essenziale ai fini delle esercitazioni e (soprattutto) dell'esame scritto.

Socket in UNIX - 18

Implementazione primitive di comunicazione nel kernel (esempio connect)

```
int connect(int sd, const struct sockaddr *sa, socklen_t len)
{
    ...
    if (len < sizeof(struct sockaddr)) return -1;
    switch (sa->sa_family) {
    case AF_INET:
        /* chiamo versione IPv4 della primitiva */
        if (len < sizeof(struct sockaddr_in)) return -2;
        connect_ipv4(sd, (struct sockaddr_in *)sa);
    case AF_INET6:
        /* chiamo versione IPv6 della primitiva */
        if (len < sizeof(struct sockaddr_in6)) return -3;
        connect_ipv6(sd, (struct sockaddr_in6 *)sa);
    case AF_UNIX: /* chiamo versione Unix della primitiva */
        if (len < sizeof(struct sockaddr_un)) return -4;
        connect_unix(sd, (struct sockaddr_un *)sa);
    ...
}
```

Tramite l'uso della "griglia" di memoria comune `struct sockaddr` e il meccanismo del downcasting si riesce a presentare una API comune a tutte le famiglie di indirizzi.

Socket in UNIX - 19

The devil is in the details...

In realtà, sebbene funzionalmente equivalente, l'architettura di gestione dei diversi domini di comunicazione nel kernel di Linux è leggermente più complessa di quella presentata nella slide precedente.

La system call `connect` è implementata dalla funzione `__sys_connect` nel file <https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/net/socket.c>, che fa riferimento alla `struct proto_ops` contenuta nella `struct socket` di interesse. La `struct proto_ops` contiene i puntatori alle specifiche versioni delle system call per il dominio di comunicazione di riferimento (l'equivalente di `connect_ipv4` e `connect_ipv6` nella slide precedente).

`AF_INET` usa `struct proto_ops inet_stream_ops` e `inet_dgram_ops`:
https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/net/ipv4/af_inet.c

`AF_INET6` invece usa `inet6_stream_ops` e `inet6_dgram_ops`:
https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/net/ipv6/af_inet6.c

Socket in UNIX - 20

Tipi di Socket

| TIPO | Descrizione |
|--------------------|--|
| SOCK_STREAM | terminale di un canale di comunicazione <u>con</u> connessione e affidabile (socket stream o TCP) |
| SOCK_SEQPACKET | trasferimento affidabile di sequenze di pacchetti (protocollo SCTP) |
| SOCK_DGRAM | terminale di un canale di comunicazione <u>senza</u> connessione non affidabile (socket datagram o UDP) |
| SOCK_RAW | accesso diretto al livello di rete (IP) |
| SOCK_DCCP | protocollo DCCP (~ UDP + congestion control) |

Socket in UNIX - 21

DOMINI Internet (AF_INET e AF_INET6)

MODELLO OSI UNIX BSD ESEMPIO

| | | |
|---------------|---------------------|-----------------------------|
| APPLICAZIONE | PROGRAMMI | TELNET |
| PRESENTAZIONE | | |
| SESSIONE | | |
| TRASPORTO | SOCKET TRASPORTO | SOCKET STREAM <i>TCP</i> |
| RETE | RETE | <i>IP</i> |
| DATA LINK | DRIVER DI RETE | DRIVER DI RETE |
| FISICO | HARDWARE | ETHERNET |

Socket in UNIX - 22

Tipi di Socket (e quindi di interazione C/S)

Una **socket STREAM** stabilisce una **connessione** (socket TCP, SOCK_STREAM). La comunicazione è **affidabile** (garanzia di consegna dei messaggi), **bidirezionale** e i byte sono consegnati **in ordine**.

Non sono mantenuti i **confini dei messaggi**. (presenza di meccanismi out-of-band) Queste caratteristiche sono forzate dalla scelta di TCP come protocollo di livello di trasporto (livello 4 OSI). (semantica at most once)

Una **socket DATAGRAM** non stabilisce alcuna connessione (**connectionless**) (socket UDP, SOCK_DGRAM).

NON c'è garanzia di consegna del messaggio.

I **confini dei messaggi** sono mantenuti.

Non è garantito l'**ordine** di consegna dei messaggi.

Queste caratteristiche sono forzate dalla scelta di UDP come protocollo di livello di trasporto (livello 4 OSI). (semantica may be)

Socket in UNIX - 23

Creazione di una socket

```
sd = socket (dominio, tipo, protocollo);  
int sd, dominio, tipo, protocollo;
```

Crea una SOCKET e ne restituisce il **descrittore** sd (socket descriptor).

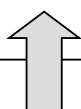
dominio denota il particolare dominio di comunicazione (es. AF_INET o AF_INET6)

tipo indica il tipo di comunicazione (es. SOCK_STREAM o SOCK_DGRAM)

protocollo specifica uno dei protocolli supportati dal dominio (se si indica zero viene scelto il protocollo di default)

Definisce il protocollo usato dalla socket. In una connessione definisce:

<protocollo;indirizzo IP locale;porta locale;indirizzo IP remoto;porta remota>



Socket in UNIX - 24

Associazione socket - indirizzo locale

```
error = bind (sd, ind, lun);  
int error, sd;  
const struct sockaddr * ind;  
socklen_t lun;
```

Associa alla socket di descrittore sd l'indirizzo codificato nella struttura puntata da ind e di lunghezza lun (la lunghezza è necessaria, poiché la funzione bind può essere impiegata con indirizzi di lunghezza diversa)

Collega la socket a un indirizzo locale. In una connessione definisce :

*<protocollo;**indirizzo IP locale**;porta locale;indirizzo IP remoto;porta remota>*



Socket in UNIX - 25

Comunicazione connection-oriented

(socket STREAM o TCP)

Collegamento (**asimmetrico**) tra processo Client e processo Server:

- 1) il server e il client devono **creare ciascuno una propria socket** e definirne l'indirizzo (primitive socket e bind)
- 2) deve essere creata la **connessione** tra le due socket
- 3) fase di **comunicazione**
- 4) **chiusura** delle socket

Socket in UNIX - 26

Comunicazione connection-oriented

2) Creazione della connessione tra il client e il server (schema **asimmetrico**):

LATO CLIENT

Il client richiede una connessione al server (primitiva **connect**)

Uso di una **socket attiva**

LATO SERVER

Il server definisce una coda di richieste di connessione (primitiva **listen**) e attende le richieste (primitiva **accept**)

Uso di una **socket passiva** (listening socket)

Quando arriva una richiesta, la richiesta viene accettata, stabilendo così la connessione. La comunicazione può quindi iniziare.

Socket in UNIX - 27

Comunicazione connection oriented (lato **Client**)

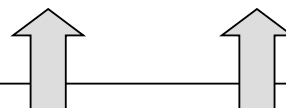
```
error = connect (sd, ind, lun);  
int error, sd;  
struct sockaddr * ind;  
socklen_t lun;
```

Richiede la connessione fra la socket locale il cui descrittore è sd e la socket remota il cui indirizzo è codificato nella struttura puntata da ind e la cui lunghezza è lun.

La connect() può determinare la sospensione del processo?

Definisce l'indirizzo remoto a cui si collega la socket:

*<protocollo;indirizzo IP locale;porta locale;**indirizzo IP remoto;porta remota**>*



Socket in UNIX - 28

Comunicazione connection oriented (lato Server)

```
error = listen (sd, dim);  
int error, sd, dim;
```

In applicazioni real-life è solitamente consigliabile usare SOMAXCONN (ovverosia il massimo valore disponibile) come dimensione della coda di richieste di connessione.

Trasforma la socket sd in **passiva** (listening), pronta per ricevere una richiesta di connessione.

Crea una **coda**, associata alla socket sd in cui vengono inserite le richieste di connessione dei client.

La coda può contenere al più dim elementi.

Le richieste di connessione vengono estratte dalla coda quando il server esegue la accept().

Comunicazione connection oriented (lato Server)

```
nuovo_sd = accept (sd, ind, lun);  
int nuovo_sd, sd;  
struct sockaddr * ind;  
socklen_t (* lun);
```

Indirizzo è parametro in/out

Estrae una richiesta di connessione dalla coda predisposta dalla listen().

Se **non** ci sono richieste di connessione in coda, **sospende il server** finché non arriva una richiesta alla socket sd.

Quando la richiesta arriva, **crea una nuova socket** di lavoro nuovo_sd e restituisce l'indirizzo della socket del client tramite ind e la sua lunghezza tramite lun.

La comunicazione (read/write) si svolge sulla nuova socket nuovo_sd

Comunicazione connection oriented

The diagram illustrates a connection-oriented communication setup between two nodes connected via a network (rete TCP/IP).

Client Node (nodo: indirizzo IP):

- Contains a **Processo Client**.
- A **socket client** is shown as a small square box connected to the client process.
- The **porta client** (client port) is represented by another small square box below the socket client.

Server Node (nodo: indirizzo IP):

- Contains a **Processo Server**.
- A **socket server (sd)** is shown as a small square box connected to the server process.
- A **socket server (nuovo_sd)** is shown as another small square box, connected to the server process and the main server socket.
- The **porta server** (server port) is represented by a small square box below the socket server.

Network (rete TCP/IP):

- Represented by a star-shaped cloud in the center.
- Curved arrows indicate the flow of communication between the client's port and the network, and between the network and the server's port.

Socket in UNIX - 31

Comunicazione connection oriented

Una volta completata la connessione si possono utilizzare le normali primitive read e write

```
nread = read (sock_desc, buf, lun);  
nwrite = write (sock_desc, buf, lun);
```

buf è il puntatore a un buffer di lunghezza lun dal quale prelevare o in cui inserire il messaggio.

sock_desc è il socket descriptor (di una socket attiva!).

uso di send e recv per dati out-of-band.

Una volta completata la connessione si possono utilizzare le normali primitive read e write

```
nread = read (sock_desc, buf, lun);
nwrite = write (sock_desc, buf, lun);
```

buf è il puntatore a un buffer di lunghezza lun dal quale prelevare o in cui inserire il messaggio.

sock_desc è il socket descriptor (di una socket attiva!).

uso di send e recv per dati out-of-band.

Socket in UNIX - 32

Comunicazione connection oriented (termine connessione)

Al termine della comunicazione, la connessione viene interrotta mediante la primitiva `close` che chiude la socket

```
error = close (sd);  
int sd;
```

Problemi `close()`:

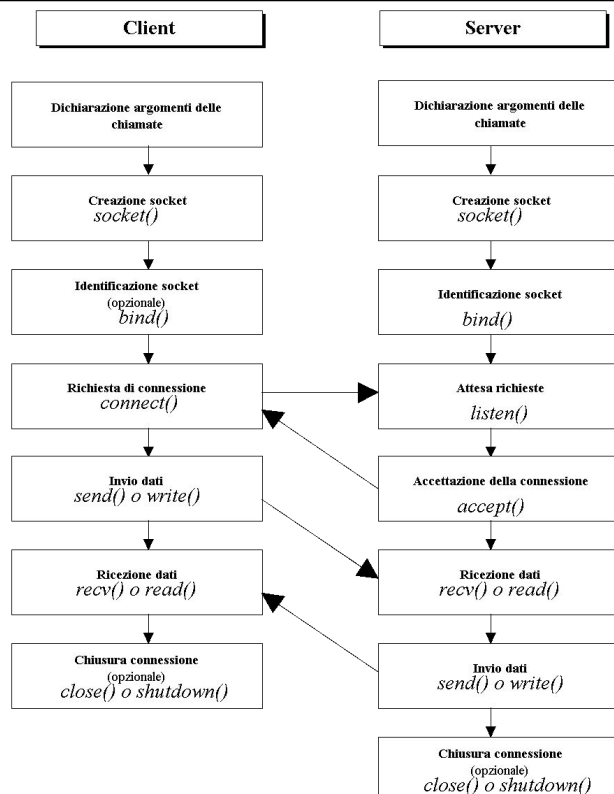
- chiusura connessione avviene solo se `sd` è l'**ultimo** descrittore aperto della socket
- chiusura della connessione in entrambi i versi (sia ricezione sia trasmissione)

```
error = shutdown (sd, how);  
int sd, how;
```

Per terminare una connessione si può usare anche la `shutdown`, che fornisce maggiore flessibilità:

- chiusura solo dell'estremo di ricezione (`how=0, SHUT_RD`)
- chiusura solo dell'estremo di trasmissione (invio EOF) (`how=1, SHUT_WR`)
- chiusura dei due estremi (`how=2, SHUT_RDWR`)

Comunicazione connection-oriented



Esempio: Dominio **AF_INET**, socket stream (con connessione) (**Client**)

```
int main (int argc, char **argv) { // argv[1] e argv[2] indirizzo IP e porta server
    int sd; struct sockaddr_in rem_indirizzo;
    ...
    sd = socket (AF_INET, SOCK_STREAM, 0);
    .....
    /* preparazione in struttura sockaddr_in rem_indirizzo dell'indirizzo del server */
    memset(&rem_indirizzo, 0, sizeof(rem_indirizzo));
    rem_indirizzo.sin_family = AF_INET;
    rem_indirizzo.sin_addr.s_addr = inet_addr(argv[1]);
    rem_indirizzo.sin_port = htons(atoi(argv[2]));
    .....
    connect (sd, (struct sockaddr *)&rem_indirizzo, sizeof(rem_indirizzo));
    .....
    write (sd, buf, dim); read (sd, buf, dim);
    .....
    // chiusura, uso di close o shutdown
    close (sd);
    ...
}
```

Attenzione! Specificando esplicitamente **AF_INET** stiamo limitando severamente la funzionalità dell'applicazione! Infatti, essa non supporterà comunicazioni IPv6!

Socket in UNIX - 35

Esempio: Dominio **AF_INET**, server

```
int main (int argc, char **argv) { // argv[1] contiene porta server
    int ss; struct sockaddr_in mio_indirizzo;
    ...
    ss = socket(AF_INET, SOCK_STREAM, 0);
    ...
    /* preparazione nella struttura mio_indirizzo dell'indirizzo del server */
    memset(&mio_indirizzo, 0, sizeof(mio_indirizzo));
    mio_indirizzo.sin_family = AF_INET;
    mio_indirizzo.sin_addr.s_addr = htonl(INADDR_ANY);
    mio_indirizzo.sin_port = htons(atoi(argv[1]));
    ...
    bind (ss, (struct sockaddr *)&mio_indirizzo, sizeof(mio_indirizzo));
    .....
    listen (ss, SOMAXCONN);
    .....
}
```

Indirizzo IPv4 + porta

Socket in UNIX - 36

Esempio: Server

```
...
for ( ; ; ) {
    ns = accept (ss, NULL, NULL);
    .....
    nread = read (ns, buf, dim);
    .....
    close (ns);
}
...
close (ss);
return 0;
}
```

Si noti che questo è un **server iterativo** sequenziale.

Socket in UNIX - 37

Esempio: Dominio **AF_INET6**, socket stream (con connessione) (**Client**)

```
int main (int argc, char **argv) { // argv[1] e argv[2] indirizzo IP e porta server
    struct sockaddr_in6 rem_indirizzo;
    int sd = socket (AF_INET6, SOCK_STREAM, 0);
    .....
    /* preparazione in struttura sockaddr_in rem_indirizzo dell'indirizzo del server */
    memset(&rem_indirizzo, 0, sizeof(rem_indirizzo));
    rem_indirizzo.sin6_family = AF_INET6;
    inet_pton(AF_INET6, argv[1], &rem_indirizzo.sin6_addr);
    rem_indirizzo.sin6_port = htons(atoi(argv[2]));
    .....
    connect (sd, (struct sockaddr *)&rem_indirizzo, sizeof(rem_indirizzo));
    ....
    write (sd, buf, dim); read (sd, buf, dim);
    .....
    // chiusura, uso di close o shutdown
    close (sd);
    return 0;
}
```

Attenzione! Specificando esplicitamente **AF_INET6** stiamo limitando severamente la portabilità del codice! Questa applicazione infatti non compilerà su sistemi che non supportano IPv6!

Socket in UNIX - 38

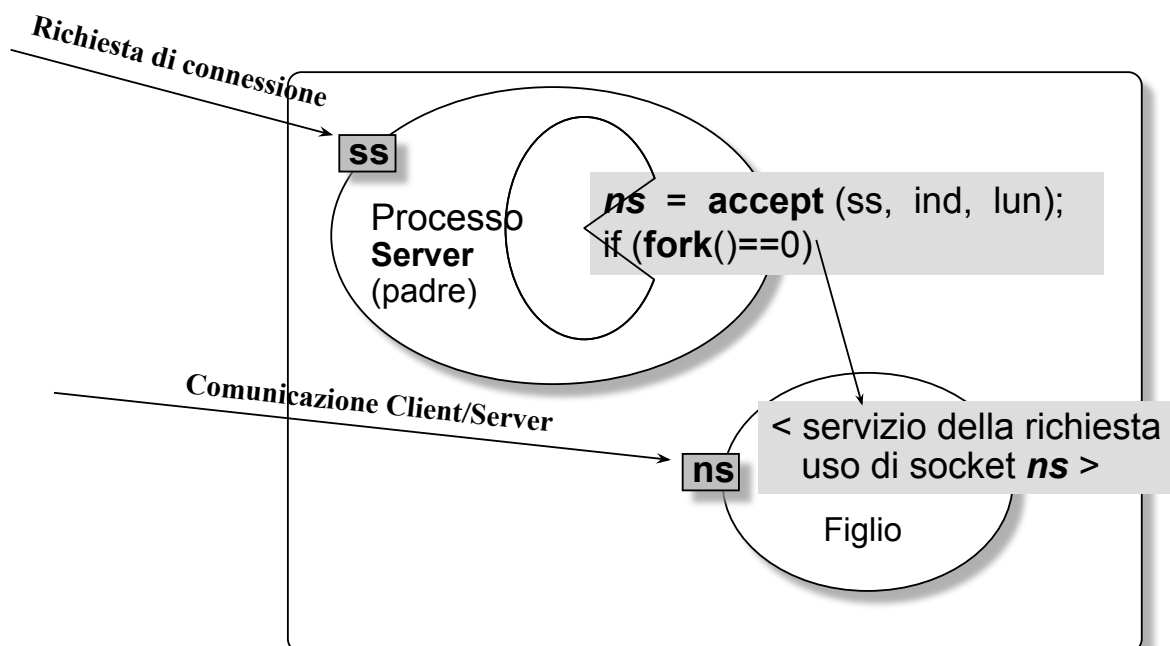
Esempio: Dominio AF_INET6, server

```
int main (int argc, char **argv) { // argv[1] contiene porta server
    struct sockaddr_in6 mio_indirizzo;
    int ss = socket(AF_INET6, SOCK_STREAM, 0);
    ...
    /* preparazione nella struttura mio_indirizzo dell'indirizzo del server */
    memset(&mio_indirizzo, 0, sizeof(mio_indirizzo));
    mio_indirizzo.sin6_family = AF_INET6;
    mio_indirizzo.sin6_addr = in6addr_any;
    mio_indirizzo.sin6_port = htons(atoi(argv[1]));
    ...
    bind (ss, (struct sockaddr *)&mio_indirizzo, sizeof(mio_indirizzo));
    .....
    listen (ss, SOMAXCONN);
    .....
```

Indirizzo IPv6 + porta

Socket in UNIX - 39

Server *concorrente* multi-processo connection-oriented



Socket in UNIX - 40

Trasformazione da nome logico a indirizzi fisici

```
int getaddrinfo(const char *host_name, const char *service_name,
               struct addrinfo *hints, struct addrinfo **res);
```

getaddrinfo serve per effettuare la risoluzione dei nomi. Riceve in ingresso: nome logico di host Internet; servizio a cui connettersi (numero di porta); varie preferenze (in *hints*); restituisce una lista di strutture **addrinfo** in cui ciascun nodo rappresenta uno degli indirizzi fisici del servizio

```
struct addrinfo {
    int ai_flags;           /* flag di controllo */
    int ai_family;         /* famiglia di indirizzi (es. AF_INET, AF_INET6, ...) */
    int ai_socktype;       /* tipo socket (es. SOCK_STREAM) */
    int ai_protocol;       /* tipo di protocollo (di solito 0) */
    socklen_t ai_addrlen;   /* dimensione struttura puntata da ai_addr */
    char *ai_canonname;     /* nome canonico dell'host remoto */
    struct sockaddr *ai_addr; /* indirizzo endpoint remoto */
    struct addrinfo *ai_next; /* puntatore al prossimo nodo della lista */
};
```

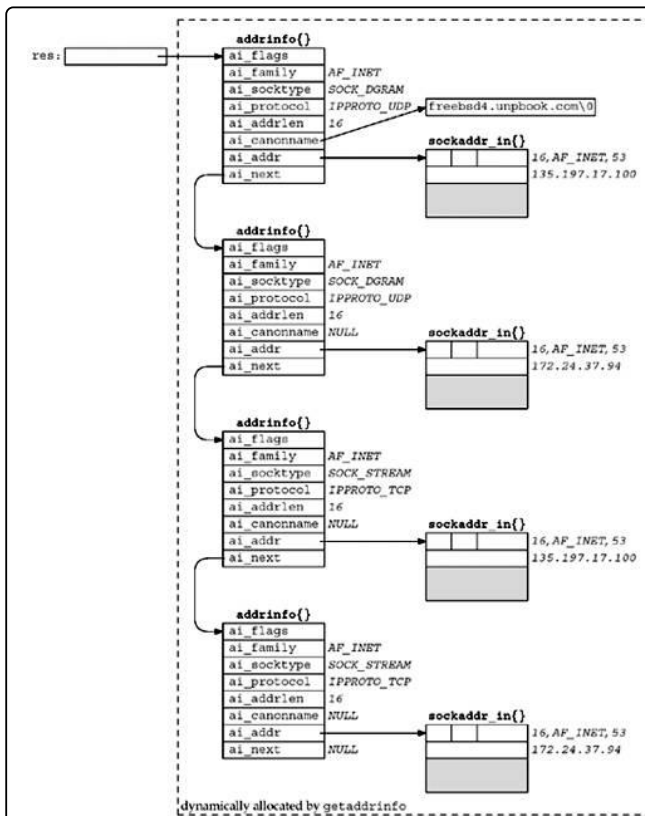
Socket in UNIX - 41

Preparazione di un indirizzo remoto

```
int err;                               /* controllo errori */
struct addrinfo hints;                 /* direttive per getaddrinfo */
struct addrinfo *res;                 /* lista indirizzi processo remoto */
char *host_remoto = "www.unife.it";   /* nome host remoto */
char *servizio_remoto = "http";       /* nome (o numero porta) servizio remoto */
.....
/* preparazione delle direttive per getaddrinfo */
memset((char *)&hints, 0, sizeof(hints));
hints.ai_family = AF_INET; /* voglio solo indirizzi di famiglia AF_INET */
hints.ai_socktype = SOCK_STREAM; /* voglio usare una socket di tipo stream */

err = getaddrinfo(host_remoto, servizio_remoto, &hints, &res);
if (err != 0) { /* controllo errori */
    /*gai_strerror restituisce un messaggio che descrive il tipo di errore */
    fprintf(stderr, "Errore risoluzione nome: %s\n", gai_strerror(err)); exit(1);
}
```

Socket in UNIX - 42



Esempio output getaddrinfo

Nella figura a sinistra è rappresentato il risultato che si ottiene con la chiamata:

```
memset((char *)&hints, 0, sizeof(hints));
hints.ai_family = AF_INET;
/* AF_INET per com. con server IPv4,
AF_INET6 per com. con server IPv6,
AF_UNSPEC per com. sia con server
IPv4 che IPv6 (dove disponibile) */
getaddrinfo("freebsd4.unpbook.com",
"53", &hints, &res);
```

Si noti che, non avendo specificato alcuna restrizione per il tipo di socket, compaiono risultati sia per `SOCK_STREAM` che per `SOCK_DGRAM`.

Quale tipo di famiglia AF usare per la risoluzione?

Specificando opportunamente il valore di `hints.ai_family` è possibile controllare il comportamento di `getaddrinfo`.

Scegliendo `AF_INET` effettuiamo solo la risoluzione da nome a indirizzo IPv4 (DNS record type A).

Scegliendo `AF_INET6` effettuiamo solo la risoluzione da nome a indirizzo IPv6 (DNS record type AAAA).

Scegliendo `AF_UNSPEC` effettuiamo sia la risoluzione da nome a indirizzo IPv4 (DNS record type A) che quella da nome a indirizzo IPv6 (DNS record type AAAA) se il sistema operativo supporta quest'ultimo protocollo.

(ATTENZIONE: anche se il sistema su cui state sviluppando supporta IPv6 non è detto che esso abbia anche connettività a Internet anche attraverso IPv6!!!)

Procedura di connessione “naive”

Uso risultato di `getaddrinfo` come argomento per le system call `socket` e `connect`

```
/* mi connetto al primo degli indirizzi restituiti da getaddrinfo */
if ((sd = socket(res->ai_family, res->ai_socktype, res->ai_protocol)) < 0) {
    fprintf(stderr, "Errore creazione socket!"); exit(2);
}

if (connect(sd, res->ai_addr, res->ai_addrlen) < 0) {
    fprintf(stderr, "Errore di connessione!"); exit(3);
}

/* una volta terminato l'utilizzo, la memoria allocata da getaddrinfo
   va esplicitamente liberata tramite una chiamata a freeaddrinfo */
freeaddrinfo(res);
...
```

Attenzione! Questa procedura di connessione è fragile! Meglio usare la versione con fallback!

Socket in UNIX - 45

Best practice per codice IPv6-enabled 1/2

Per lo sviluppo di applicazioni portabili IPv6-enabled, è bene specificare sempre:

```
hints.ai_family = AF_UNSPEC
```

In questo modo, le nostre applicazioni useranno automaticamente IPv6 dove presente, e continueranno comunque a funzionare in IPv4 laddove IPv6 non sia ancora supportato.

Tuttavia, questa pratica può portare a problemi di connessione in macchine che supportano IPv6 ma che non hanno connettività IPv6 (ovverosia gran parte dei sistemi attualmente connessi a Internet).

Infatti, `getaddrinfo`, ordina la lista di indirizzi restituiti secondo la procedura specificata nello RFC 6724.

Socket in UNIX - 46

Best practice per codice IPv6-enabled 2/2

Quando chiamata con `AF_UNSPEC` su un sistema con supporto a IPv6, `getaddrinfo` elenca per primi gli indirizzi IPv6. Questo significa che tipicamente l'applicazione proverà prima a connettersi al server tramite IPv6. Se non c'è connettività IPv6, il tentativo di connessione ovviamente fallirà.

Per evitare questo tipo di problemi è bene progettare la fase di connessione delle nostre applicazioni in modo da consentire il fallback a indirizzi IPv4 laddove la connettività IPv6 non fosse presente.

(NOTA: su una macchina Unix è possibile verificare - e cambiare - le politiche di ordinamento degli indirizzi restituiti da `getaddrinfo` accedendo al file `/etc/gai.conf`)

Procedura di connessione con fallback

```
struct addrinfo hints, *res, *ptr;
...
for (ptr = res; ptr != NULL; ptr = ptr->ai_next) {
    /* se socket fallisce salto direttamente alla prossima iterazione */
    if ((sd = socket(ptr->ai_family, ptr->ai_socktype, ptr->ai_protocol)) < 0)
        continue;
    /* se connect funziona esco dal ciclo */
    if (connect(sd, ptr->ai_addr, ptr->ai_addrlen) == 0)
        break;
    close(sd);
}
/* se ptr vale NULL vuol dire che nessuno degli indirizzi restituiti da
   getaddrinfo è raggiungibile */
if (ptr == NULL) {
    fprintf(stderr, "Errore di connessione!\n"); exit(3);
}
...
freeaddrinfo(res); /* non dimentichiamo mai di chiamare freeaddrinfo! */
```


Happy eyeballs

Se si implementa una procedura di connessione con fallback, è possibile aver tempi di connessione al server molto lunghi - soprattutto nel caso non vi sia connettività IPv6.

Per risolvere questo problema, IETF ha pubblicato lo RFC 6555, che raccomanda l'adozione di un nuovo algoritmo di connessione denominato "happy eyeballs" (aprile 2012).

Con happy eyeballs, il client prova a connettersi contemporaneamente al server sia via IPv4 che via IPv6, e usa la prima delle due connessioni che viene stabilita.

Happy eyeballs è un meccanismo pensato in ottica di breve-medio termine, per facilitare la migrazione a IPv6, e verrà deprecato quando IPv6 sarà la versione più diffusa del protocollo IP.

Happy eyeballs

Browser moderni come Firefox e Chrome implementano già happy eyeballs, che è recentemente diventato anche il comportamento standard di OS X, a partire dalla versione 10.11 «El Capitan».

Per ulteriori informazioni:

<http://tools.ietf.org/html/rfc6555>

<http://www.potaroo.net/ispcol/2012-05/notquite.html>

<http://happy.vaibhavbajpai.com>

<http://daniel.haxx.se/blog/2012/01/03/getaddrinfo-with-round-robin-dns-and-happy-eyeballs/>

Server e getaddrinfo

In realtà, `getaddrinfo` è così comoda che ci conviene usarla (con la flag `AI_PASSIVE`) anche sul server per preparare l'indirizzo da passare a `bind`:

```
memset((char *)&hints, 0, sizeof(hints));
hints.ai_flags = AI_PASSIVE;
hints.ai_family = AF_INET; /* AF_INET6 per comunicare con server IPv6,
                             AF_UNSPEC per comunicare sia con server IPv4
                             che con server IPv6 (dove disponibile) */
hints.ai_socktype = SOCK_STREAM;

err = getaddrinfo(NULL, "50000", &hints, &res);

sd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

if (bind(sd, res->ai_addr, res->ai_addrlen) < 0) { perror("bind"); exit(1); }

freeaddrinfo(res); /* non dimentichiamoci mai di liberare la memoria!*/
...
```

Socket in UNIX - 51

Caveat

Nonostante il codice nella slide precedente sia più che adeguato per scopi didattici, e assolutamente sufficiente per quanto riguarda questo corso, esso presenta un problema nel caso `hints.ai_family = AF_UNSPEC` e la configurazione di default del file `/etc/gai.conf` assegni (erroneamente) assegni una priorità più elevata a indirizzi IPv4 rispetto a i

Questo problema si può facilmente risolvere su Linux e OS X modificando la configurazione di default del file `/etc/gai.conf` (modifica già applicata sui PC del laboratorio di informatica grande).

Per ulteriori informazioni si veda il post:

<http://www.tortonesi.com/blog/2015/03/13/fixing-getaddrinfo/>

Socket in UNIX - 52

Perché non usare il fallback anche lato server?

Attenzione: il libro di testo W. Stevens et al., «Unix Network Programming», Vol. 1, III ed., propone l'adozione del meccanismo di fallback anche per la creazione di socket lato server (paragrafi 11.13 e 11.16). Tale approccio è *****FONDAMENTALMENTE ERRATO*****.

Infatti, lato server vogliamo usare solo la prima struttura sockaddr restituita da getaddrinfo. Se questa non funziona, molto meglio terminare il processo con un errore piuttosto che tentare di usare le strutture sockaddr successive, che limiterebbero significativamente la connettività del nostro server.

Se proprio non vi piace l'idea di agganciarvi a una singola socket lato server, considerate l'alternativa (significativamente più complicata ma corretta) proposta dallo RFC 4038, paragrafo 6.3:

<https://www.ietf.org/rfc/rfc4038.txt>

Socket in UNIX - 53

Strumenti test_gai e testga

Si possono usare gli strumenti `test_gai` (disponibile sul sito del corso) e `testga` (in allegato al testo “Unix Network Programming” di Stevens et al. e liberamente disponibile al sito web <http://unpbook.com>) per interrogare getaddrinfo con diversi parametri e stampare a video gli indirizzi restituiti.

Esempi d'uso di `test_gai`:

```
./test_gai -s github 443 # AF_UNSPEC e SOCK_STREAM  
./test_gai -6 -d dns.unife.it dns # AF_INET6 e SOCK_DGRAM
```

Esempio d'uso di `testga`:

```
./testga -f inet -s http -h www.google.com # AF_INET
```

Socket in UNIX - 54

Esercizio: copia remota di un file (rcp)

Si scriva un'applicazione distribuita Client/Server che presenti l'interfaccia:

rcp nodoserver nomefile

dove ***nodoserver*** specifica l'indirizzo logico del nodo contenente il processo Server e ***nomefile*** rappresenta il nome assoluto di un file nella macchina Server.

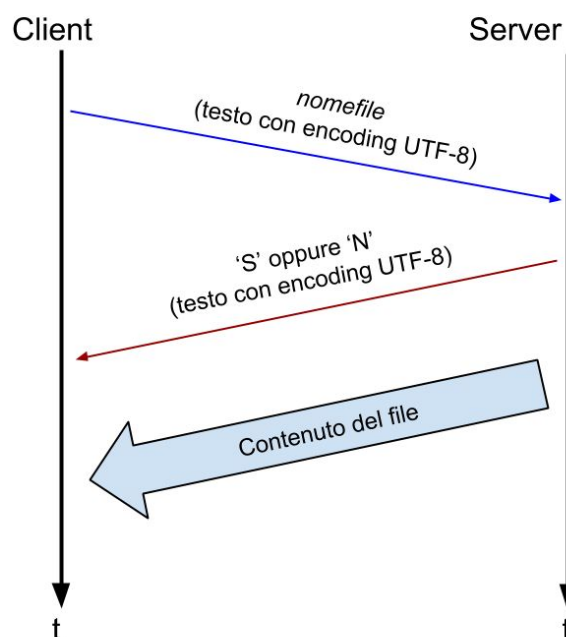
Per prima cosa, il processo Server deve controllare se il file ***nomefile*** effettivamente esiste. In caso affermativo, il Server dovrà prima trasmettere al Client il carattere 'S' e poi procedere con l'invio del contenuto del file. Altrimenti, il server dovrà trasmettere il carattere 'N' e chiudere immediatamente la connessione.

A sua volta, il processo Client deve copiare il file ***nomefile*** nel direttorio corrente (si supponga di avere tutti i diritti necessari per eseguire tale operazione), ma **solo se** in tale direttorio non è già presente un file con lo stesso nome, per evitare di sovrascriverlo.

Si supponga inoltre che il Server si leghi alla porta 50001.

Socket in UNIX - 55

Protocollo Applicativo



Socket in UNIX - 56

Esercizio: rcv - lato Client (TCP)

```
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_UNSPEC; hints.ai_socktype = SOCK_STREAM;

if ((err = getaddrinfo(argv[1], "50001", &hints, &res)) != 0) {
    fprintf(stderr, "Errore ris nome: %s\n", gai_strerror(err)); exit(1);
}
if ((sd=socket(res->ai_family, res->ai_socktype, res->ai_protocol))<0){
    fprintf(stderr, "Errore in connect"); exit(1);
}
if(connect(sd,res->ai_addr, res->ai_addrlen)<0) {
    perror("Errore in connect"); exit(1);
}
freeaddrinfo(res);

if (write(sd, argv[2], strlen(argv[2]))<0) { perror("write"); exit(1);}
if ((nread = read(sd, buff, 1))<0) { perror("read"); exit(1);}

if(buff[0]=='S') {
    if((fd=open(argv[2], O_WRONLY|O_CREAT|O_EXCL))<0) {
        printf("File locale esiste già, termino\n"); exit(1);
    }
    while((nread=read(sd, buff, DIM_BUFF))>0) write(fd,buff,nread);
} else printf("File remoto non esiste, termino\n");
close(sd); return 0;
```

Socket in UNIX - 57

Esercizio: rcv - lato Server (TCP)

```
...
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_UNSPEC; hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;

if ((err = getaddrinfo(NULL, "50001", &hints, &res)) != 0) {
    fprintf(stderr, "Errore setup indirizzo bind: %s\n",
        gai_strerror(err));
    exit(1);
}
if ((sd = socket(res->ai_family, res->ai_socktype, res->ai_protocol)) < 0){
    perror("Errore in socket"); exit(2);
}
if (bind(sd, res->ai_addr, res->ai_addrlen) < 0) {
    perror("Errore in bind"); exit(3);
}
freeaddrinfo(res);

listen(sd, 100); /* trasforma in socket passiva d'ascolto */

chdir("/var/local/files");
...
```

Socket in UNIX - 58

Esercizio: rcp - lato Server - versione iterativa

```
...
for(;;) {
    ns=accept(sd,NULL,NULL);
    memset(buff, 0, sizeof(buff));
    read(ns, buff, sizeof(buff)-1);
    printf("il server ha letto %s \n", buff);
    if((fd = open(buff, O_RDONLY)) < 0) {
        printf("File non esiste\n");
        write(ns, "N", 1);
    } else {
        printf("File esiste, lo mando al client\n");
        write(ns, "S", 1);
        while((nread = read(fd, buff, DIM_BUFF)) > 0) {
            write(ns, buff, nread);
            cont+=nread;
        }
        printf("Copia eseguita di %d byte\n", cont);
        close(fd);
    }
    close(ns);
}
```

Socket in UNIX - 59

Esercizio: rcp - lato Server - versione concorrente

```
for(;;) {
    ns=accept(sd,NULL,NULL);
    if (fork()==0) { /* figlio */
        close(sd);
        memset(buff, 0, sizeof(buff));
        read(ns, buff, sizeof(buff)-1);
        printf("il server ha letto %s \n", buff);
        if((fd = open(buff, O_RDONLY)) < 0) {
            printf("File non esiste\n");
            write(ns, "N", 1);
        } else {
            printf("File esiste, lo mando al client\n");
            write(ns, "S", 1);
            while((nread = read(fd, buff, DIM_BUFF)) > 0) {
                write(ns,buff,nread); cont+=nread;
            }
            printf("Copia eseguita di %d byte\n", cont);
            close(fd);
        }
        close(ns); exit(0);
    }
    close(ns);
    wait(&status); } /* attenzione: sequenzializza ... Cosa fare? */
```

Socket in UNIX - 60

Assunzioni di lavoro – dati contenuti in singoli messaggi TCP

Attenzione! Nell'implementazione dell'esercizio rcp il client ha inviato il dato *nomefilesorgente* come una stringa senza alcun terminatore in un'unica write e il server ha letto l'intero dato dalla socket con una singola read.

C'è un'implicita assunzione che il dato verrà comunicato in un singolo messaggio TCP, il cui contenuto verrà catturato interamente dalla read. Questo approccio consente di mantenere il codice delle nostre applicazioni ragionevolmente semplice, ma non è adeguato per sviluppare applicazioni che abbiano la velleità di essere qualcosa in più di un semplice «giocattolo».

Se volessimo sviluppare applicazioni che comunicano stringhe di dimensioni maggiori e/o applicazioni che realizzano servizi Internet, saremmo costretti ad adottare protocolli più robusti. Ad esempio, **dovremmo definire e usare un codice di terminazione delle stringhe, ad es. '\n', e realizzare e usare una funzione di lettura di stringhe di complessa implementazione (si veda *readline* in «Unix Network Programming, Vol. 1», paragrafi 3.9 e 26.5).**

Assunzioni di lavoro – gestione stringhe UTF-8

Attenzione! Nell'implementazione dell'esercizio rcp non abbiamo effettuato la validazione delle stringhe UTF-8 prima di utilizzarle, né alcuna conversione da codifica UTF-8 a codifica locale.

C'è un'implicita assunzione che il processo con cui comunichiamo ci fornirà stringhe UTF-8 valide e che client e server girino su piattaforme che usano UTF-8 come codifica locale del testo. Questo approccio ci consente di mantenere il codice delle nostre applicazioni ragionevolmente semplice, ma è adeguato solo per applicazioni «giocattolo».

Infatti, qualsiasi applicazione seria **dovrebbe come minimo controllare la validità delle stringhe UTF-8, e molto probabilmente anche convertire le stringhe da UTF-8 alla codifica locale, prima di poterle utilizzare.** Inoltre, è necessario considerare attentamente quali politiche di gestione delle stringhe non valide (es.: sanitizzazione o rifiuto?) adottare e implementare consistentemente.

Comunicazione connectionless (socket UDP o DATAGRAM)

Non viene creata una connessione tra processo client e processo server.

- 1) la primitiva `socket()` crea la socket
- 2) la primitiva `bind()` lega la socket a un numero di porta (opzionale per client)
- 3) i processi inviano/ricevono msg su socket. Per ogni msg specifica indirizzo destinatario (IP e porta)

Si noti che un processo può usare la stessa socket per spedire/ricevere messaggi a/da diversi altri processi.

Attenzione che le socket DATAGRAM permettono comunicazione:

- Senza connessione (connectionless)
- Non affidabile (unreliable)
- Datagram

(caratteristiche che derivano dalla semantica del protocollo sottostante UDP)

Socket in UNIX - 63

Comunicazione connectionless (socket DATAGRAM)

int **sendto**(int sd, const void *msg, size_t len, int flags,
const struct sockaddr *to, socklen_t tolen);

int **recvfrom**(int sd, void *buf, size_t len, int flags,
struct sockaddr *from, socklen_t *fromlen);

Le primitive `recvfrom` e `sendto`, oltre agli stessi parametri delle `read` e `write`, hanno anche due parametri aggiuntivi che denotano indirizzo e lunghezza di una struttura socket:

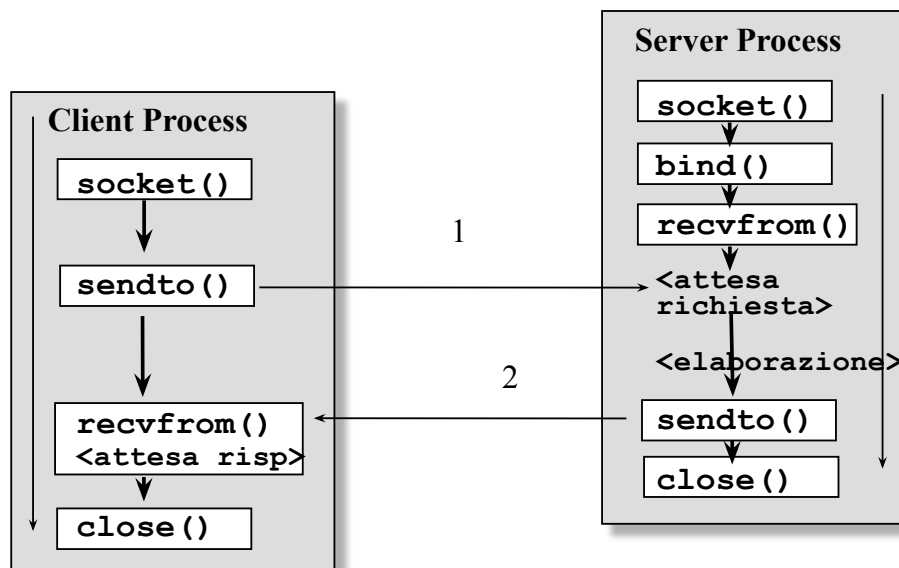
- nella `sendto` servono per specificare l'indirizzo del destinatario;
- nella `recvfrom` servono per restituire l'indirizzo del mittente.

La `recvfrom` sospende il processo in attesa di ricevere il messaggio (coda di messaggi associata alla socket).

La `recvfrom` può ritornare zero, se ha ricevuto un messaggio (un datagramma) di dimensione zero.

Socket in UNIX - 64

Comunicazione senza connessione (socket DATAGRAM)



Socket in UNIX - 65

Esercizio: echo UDP - Lato Server

```
int sockfd, cc; socklen_t len; uint8_t mesg[MAXLINE];
struct sockaddr_storage client_address;

memset(&hints, 0, sizeof(hints));
hints.ai_flags = AI_PASSIVE;
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_DGRAM;

err = getaddrinfo(NULL, "7", &hints, &res);
sd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
bind(sd, res->ai_addr, res->ai_addrlen) < 0);
freeaddrinfo(res);

for (;;) { /* ciclo infinito, non termina mai */
    len = sizeof(client_address);
    cc = recvfrom(sd, mesg, sizeof(mesg), 0,
                  (struct sockaddr *)&client_address, &len);
    sendto(sd, mesg, cc, 0,
            (struct sockaddr *)&client_address, len);
}
```

Socket in UNIX - 66

Note conclusive sulle socket UDP

Ovviamente si possono fare le stesse considerazioni svolte nella parte delle socket Java:

- **UDP non è affidabile**, perdita messaggi può bloccare Cliente (utilizzo di timeout?)
- **Blocco del Client** anche per perdita mesg verso Server inattivo (errori nel collegamento al server notificati solo sulle socket connesse)
- **UDP non ha flow control**, Server lento perde messaggi (opzione SO_RCVBUF per modificare la lunghezza coda)

Quale tipo di Socket utilizzare?

Per la **scelta del tipo di socket** stesse considerazioni viste in Java:

Servizi che richiedono una connessione \longleftrightarrow servizi connectionless

Socket STREAM sono **affidabili**, DATAGRAM no.

Prestazioni STREAM inferiori alle DATAGRAM (costo di mantenere una connessione logica).

Socket STREAM hanno **semantica** at-most-once, socket DATAGRAM hanno semantica may be (servizio idempotente?)

Ordinamento messaggi (preservato in STREAM, non in DATAGRAM)

Per fare del **broadcast/multicast** più indicate le DATAGRAM (altrimenti richiesto apertura di molte connessioni contemporaneamente).

ATTENZIONE: queste differenze tra le socket derivano dalle differenze dei protocolli sottostanti (**STREAM su TCP, DATAGRAM su UDP**)

problema di multicast (socket DATAGRAM)

Si progetti un'applicazione distribuita Client/Server utilizzando le chiamate di sistema UNIX. Il Client deve offrire la seguente interfaccia:

inviomessaggio nodo1 nodo2 ... nodoN stringa

dove ***nodo1***, .. ***nodoN*** sono nomi logici di nodi sulla rete Internet e ***stringa*** è una stringa di caratteri qualsiasi.

Il Client deve inviare la stringa a tutti i processi Server in esecuzione sui nodi passati come parametro. Ogni Server deve visualizzare la stringa ricevuta sulla "console".

Il Server si deve legare alla porta 54321.

problema di multicast - Lato Client

```
...
if ((sd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) { ... }

memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_DGRAM;

for (i=1; i < argc-1; i++) {

    if ((err = getaddrinfo(argv[i], "54321", &hints, &res)) != 0) { ... }

    sendto(sd, argv[argc-1], (strlen(argv[argc-1])), 0,
           res->ai_addr, res->ai_addrlen);

    freeaddrinfo(res);

} /*end for*/

close(sd);
return 0;
```

problema di multicast - Lato Server (iterativo)

```
hints.ai_flags = AI_PASSIVE;
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_DGRAM;

if ((err = getaddrinfo(NULL, "54321", &hints, &res)) != 0) { ... }
if ((sd = socket(res->ai_family, res->ai_socktype,
                res->ai_protocol)) < 0) {...}
if (bind(sd, res->ai_addr, res->ai_addrlen) < 0) { ... }

fd = open("/dev/console", O_WRONLY); /* NB: scrive su console*/

for (;;) {
    struct sockaddr_storage clnt_addr; socklen_t len;
    len = sizeof(clnt_addr); /* va re-inizializzato a ogni iterazione */
    memset(buff, 0, sizeof(buff));
    cc = recvfrom(sd, buff, sizeof(buff), 0,
                  (struct sockaddr *)&clnt_addr, &len);
    write(fd, buff, cc);
}
close(sock); return 0;
```

Socket in UNIX - 71

Opzioni per le Socket

Le opzioni permettono di modificare il comportamento delle socket.

Configurazione attraverso le primitive:

getsockopt() *setsockopt()*

leggere e fissare le modalità di utilizzo delle socket (tipicamente il valore del campo vale 0 o 1)

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

```
#include <sys/socket.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);
```

sockfd = socket descriptor

level= livello (semantico) del protocollo (es. SOL_SOCKET, IPPROTO_TCP, ecc.)

optname = nome dell'opzione

optval = puntatore a un'area di memoria per valore

optlen = lunghezza (o puntatore) quarto argomento

Socket in UNIX - 72

Opzioni per le Socket

| Opzioni | Descrizione |
|----------------|---|
| SO_DEBUG | abilita il debugging (valore diverso da zero) |
| SO_REUSEADDR | riuso dell'indirizzo locale |
| SO_DONTROUTE | abilita il routing dei messaggi uscenti |
| SO_LINGER | ritarda la chiusura per messaggi pendenti |
| SO_BROADCAST | abilita la trasmissione broadcast |
| SO_OOBINLINE | messaggi prioritari pari a quelli ordinari |
| SO_SNDBUF | setta dimensioni dell'output buffer |
| SO_RCVBUF | setta dimensioni dell'input buffer |
| SO_SNDLOWAT | setta limite inferiore di controllo di flusso out |
| SO_RCVLOWAT | limite inferiore di controllo di flusso in ingresso |
| SO_SNDTIMEO | setta il timeout dell'output |
| SO_RCVTIMEO | setta il timeout dell'input |
| SO_USELOOPBACK | abilita network bypass |
| SO_KEEPAIVE | Controllo periodico connessione |
| SO_PROTOCOL | setta tipo di protocollo |

Socket in UNIX - 73

Opzioni per le Socket: Riutilizzo del socket address (STREAM)

L'opzione **SO_REUSEADDR** modifica il comportamento della syscall `bind()`.

Il sistema tende a non ammettere più di un utilizzo di un indirizzo locale. (Problema di attesa stato `TIME_WAIT` di TCP che, a seconda del sistema in considerazione, può impedire il re-bind sullo stesso indirizzo per un intervallo di tempo da 1 a 4 minuti.) Con l'opzione **SO_REUSEADDR**, si forza l'uso dell'indirizzo di una socket **senza controllare l'unicità di associazione**.

Se il demone termina, il riavvio necessita **SO_REUSEADDR**, altrimenti la chiamata a `bind()` sulla stessa porta causerebbe errore poiché *risulta già presente una connessione legata allo stesso socket address*.

Attenzione a non confondere
SO_REUSEADDR con
SO_REUSEPORT!!!

```
int optval=1;
setsockopt(sd, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof(optval));
bind(sd, res->ai_addr, res->ai_addrlen);
```

Socket in UNIX - 74

Opzioni per le Socket: TCP_NODELAY, TCP_QUICKACK e TCP_CORK

Le opzioni **TCP_NODELAY** (standard), **TCP_QUICKACK** e **TCP_CORK** (presenti solo in Linux) modificano il comportamento delle socket stream:

- **TCP_NODELAY** disabilita l'algoritmo di Nagle che, per migliorare performance, cerca di consolidare numerose chiamate a write con buffer di piccole dimensioni in un numero minore di messaggi TCP di maggiori dimensioni. In pratica, l'algoritmo di Nagle forza un limite minimo di 200ms tra la trasmissione di due messaggi TCP consecutivi qualora essi non raggiungessero la dimensione massima consentita dalla rete (Maximum Segment Size, MSS).
- **TCP_QUICKACK** forza l'immediata trasmissione di ACK dopo la ricezione di un pacchetto (per default si implementa una politica delayed ACK di 40 msec per minimizzare messaggi ACK senza payload).
- **TCP_CORK** disabilita la trasmissione di messaggi TCP che non raggiungono la dimensione massima consentita dalla rete (MSS).

Per ulteriori informazioni si vedano il post http://baus.net/on-tcp_cork e il paragrafo 61.4 del libro M. Kerrisk, «The Linux Programming Interface»

Socket in UNIX - 75

Altre Opzioni per le Socket

Controllo periodico della connessione

Il protocollo di trasporto può inviare messaggi di controllo periodici per analizzare lo stato di una connessione (**SO_KEEPALIVE**)

Se problemi ==> connessione è considerata abbattuta
i processi avvertiti da un **SIGPIPE** *chi invia dati*
da **end-of-file** *chi li riceve*

Di solito, verifica ogni 45 secondi e dopo 6 minuti di tentativi
opzione in dominio Internet tipo boolean

Dimensioni buffer di trasmissione/ricezione

Opzioni **SO_SNDBUF** e **SO_RCVBUF**

Aumento della dimensione buffer di trasmissione ==> invio messaggi più grandi senza attesa
massima dimensione possibile 65535 byte

```
int result; int buffersize=10000;  
result=setsockopt (s, SOL_SOCKET, SO_RCVBUF, &buffersize, sizeof(buffersize));
```

Attenzione! Usare **SO_SNDBUF** può essere controproducente perché disabilita auto-tuning effettivamente diminuendo la dimensione massima del buffer di ricezione anziché aumentarla:
<https://twitter.com/majek04/status/1534442192441319424>

Socket in UNIX - 76

Gestione SIGPIPE

Il comportamento di default delle socket per la gestione degli errori purtroppo è mal definito. Se un processo effettua una `write()` su una socket chiusa, anziché ricevere un errore -1, viene lanciato un SIGPIPE che di default termina il processo!

Ricordarsi sempre di disabilitare SIGPIPE in tutti i propri programmi:

```
signal(SIGPIPE, SIG_IGN);
```