Università di Ferrara

Architettura di Reti

Socket in UNIX

Carlo Giannelli
carlo.giannelli@unife.it
http://www.unife.it/scienze/informatica/insegnamenti/architettura-reti/
http://docente.unife.it/carlo.giannelli

Unix: strumenti di sincronizzazione, memorizzazione, comunicazione

Uso di segnali (sincronizzazione) → processo invia limitata quantità info (solo un tipo di segnale) e manca il mittente del segnale

Uso di file (memorizzazione) → solo tra processi che condividono un file system

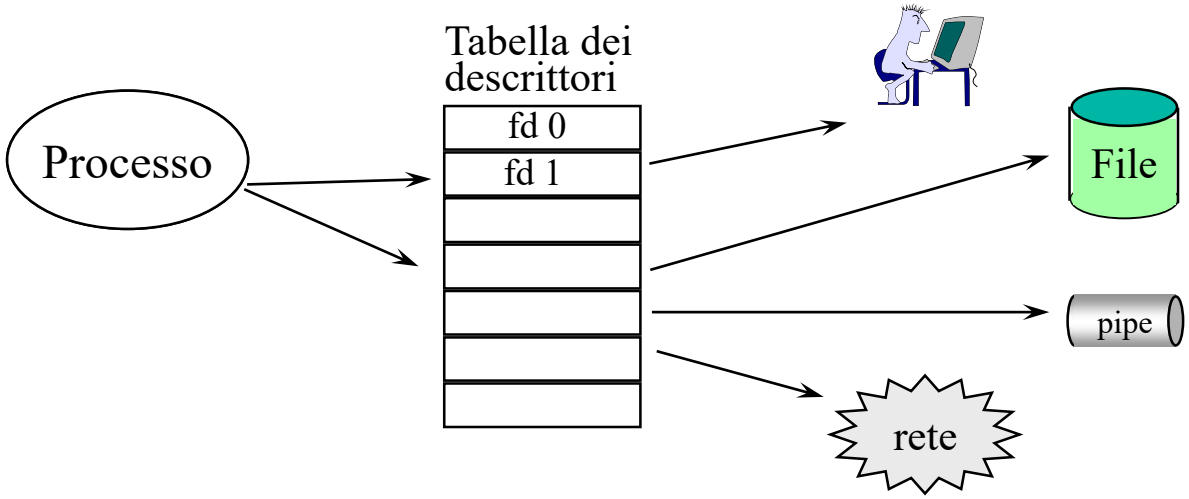
IPC
Inter Process Communication

- pipe (solo tra processi con un avo in comune)
- pipe con nome (per processi su una stessa macchina)
- shared memory (stessa macchina)
- **caso remoto** → **Socket** (e RPC – Remote Procedure Call)

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 (ad esempio, read bloccante in attesa dati)

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 (ad esempio sendto socket datagram necessita di indirizzo/porta di destinazione)

- in programmazione di rete bisogna specificare più parametri per definire un collegamento con connessione

  <protocollo; indirizzo locale; processo locale; indirizzo remoto; processo remoto>

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

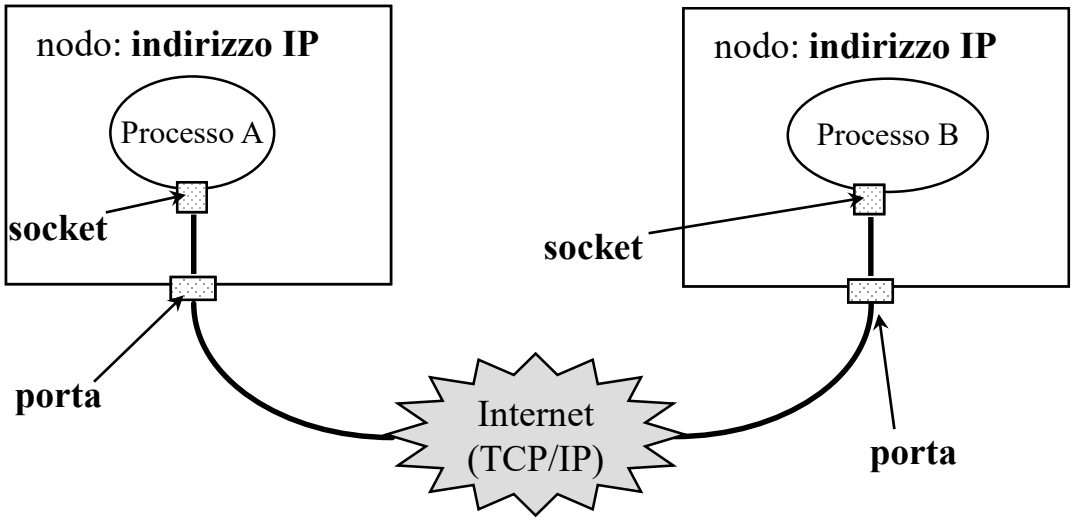
- è necessario che l'interfaccia socket possa gestire più protocolli

- schemi Client/Server sono asimmetrici (si ricordi che le pipe sono simmetriche)

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)

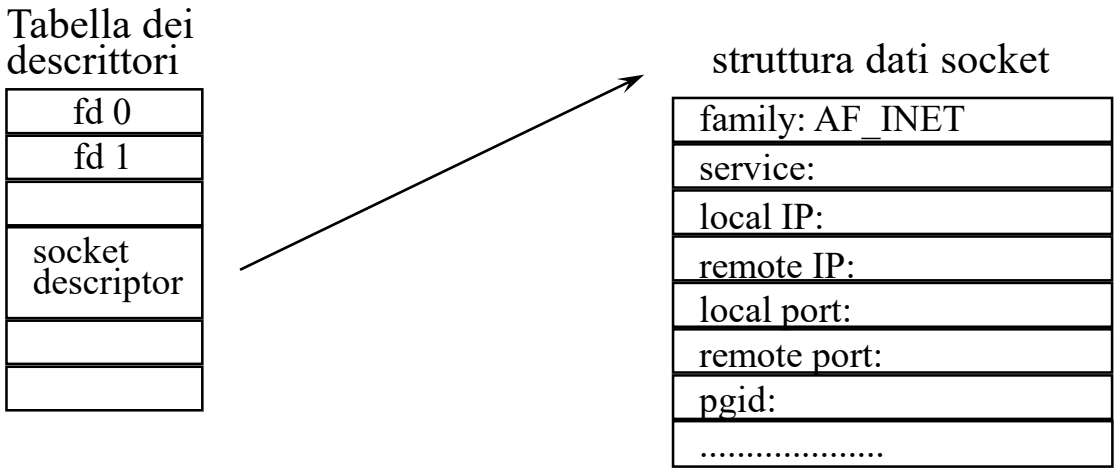


Il **canale di comunicazione** tra il processo A e il processo B è definito da:
<***protocollo***; ***indirizzo IP locale***; ***porta locale***; ***indirizzo IP remoto***; ***porta remota***>

Socket

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**

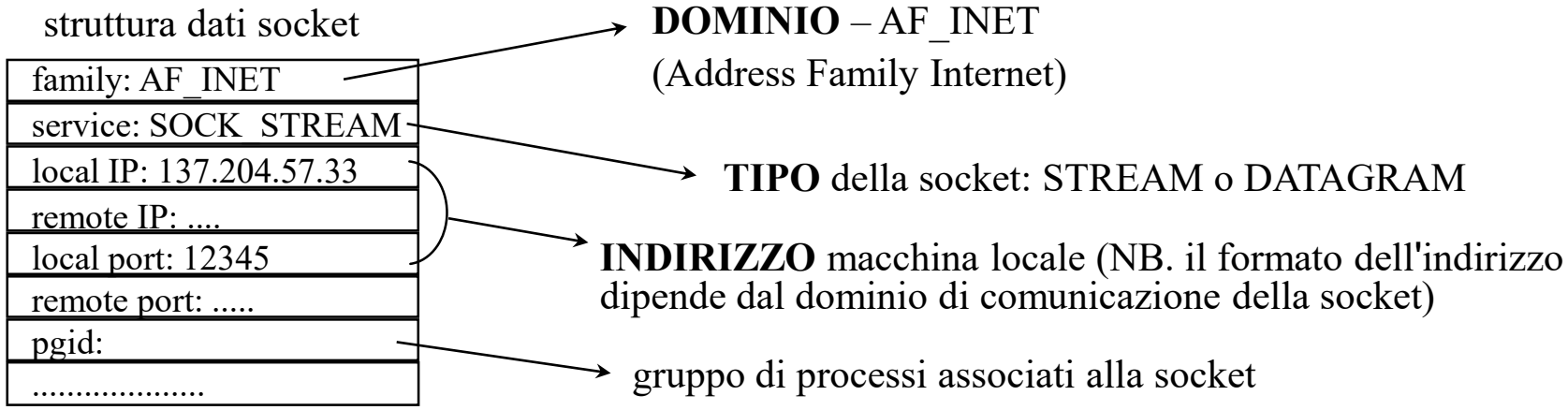


Strutture dati Socket

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

Dominio di comunicazione:
semantica di comunicazione + standard di denominazione

Esempi di domini: AF_UNIX (Unix sockets), **AF_INET (IPv4),** AF_INET6 (IPv6 con retrocompatibilità), AF_UNSPEC (IPv4 e IPv6), etc.



Formato indirizzi e struttura socket

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

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.).

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

Formato indirizzi nel Dominio AF_INET

struttura in_addr

```
struct in_addr {
        in_addr_t s_addr;
};
```

struttura sockaddr_in

```
struct sockaddr_in {
        uint8_t sin_len;
        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 {
        uint8_t sin6_len;
        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 e struttura socket

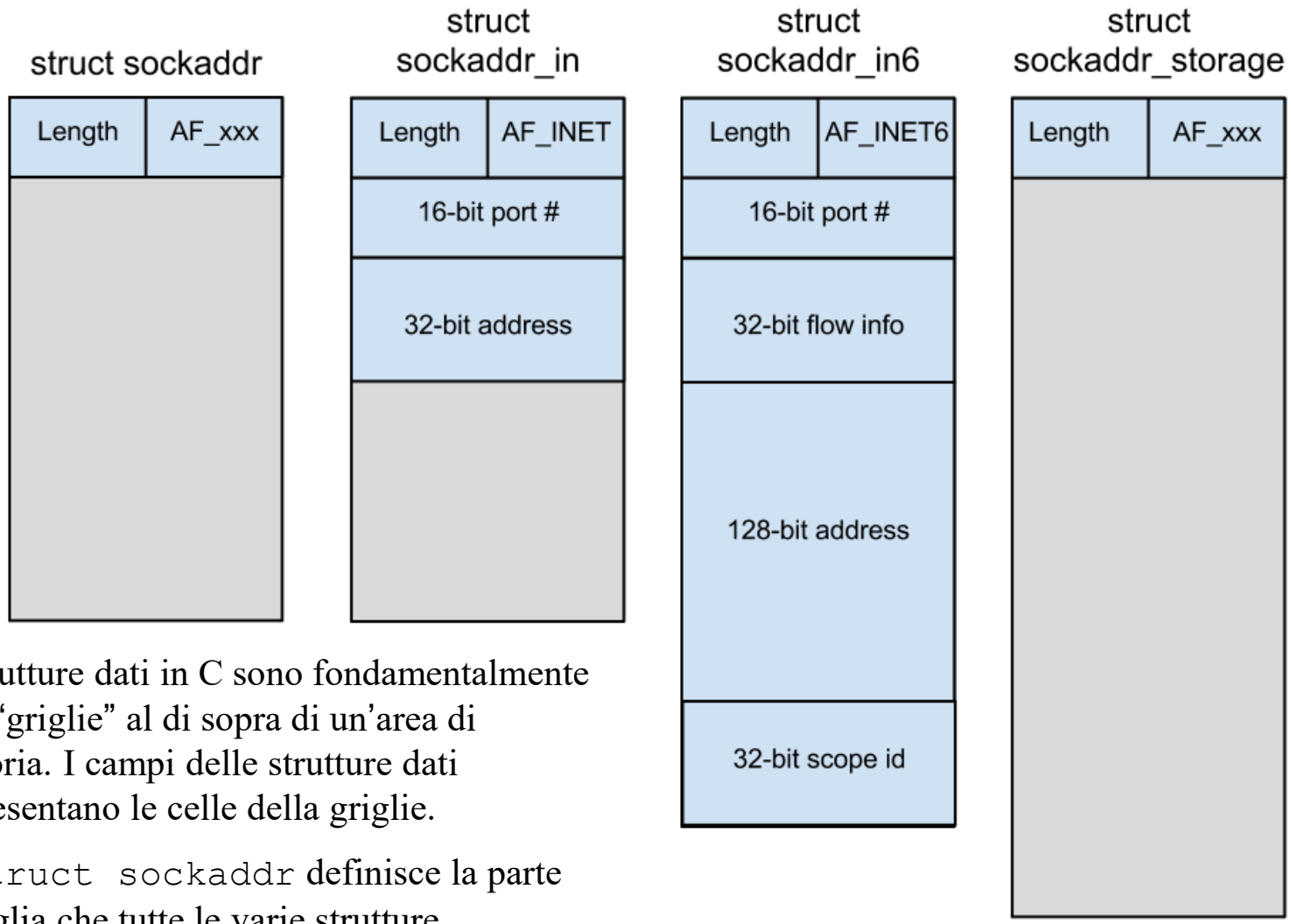
Le funzioni per la programmazione di rete nella Socket API (es. `bind`, `connect`) possono ricevere diversi tipi di indirizzo a seconda della famiglia di protocolli scelta (es. AF_UNIX, AF_INET, o AF_INET6)

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 {
        uint8_t sa_len;
        sa_family_t sa_family;
        char sa_data[14];
}
```

```
struct sockaddr_storage {
        uint8_t ss_len;
        sa_family_t ss_family;
        char ss_padding[SIZE];
}
```



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 griglie.

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

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/.

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

Implementazione primitive di comunicazione nel kernel (esempio connect)

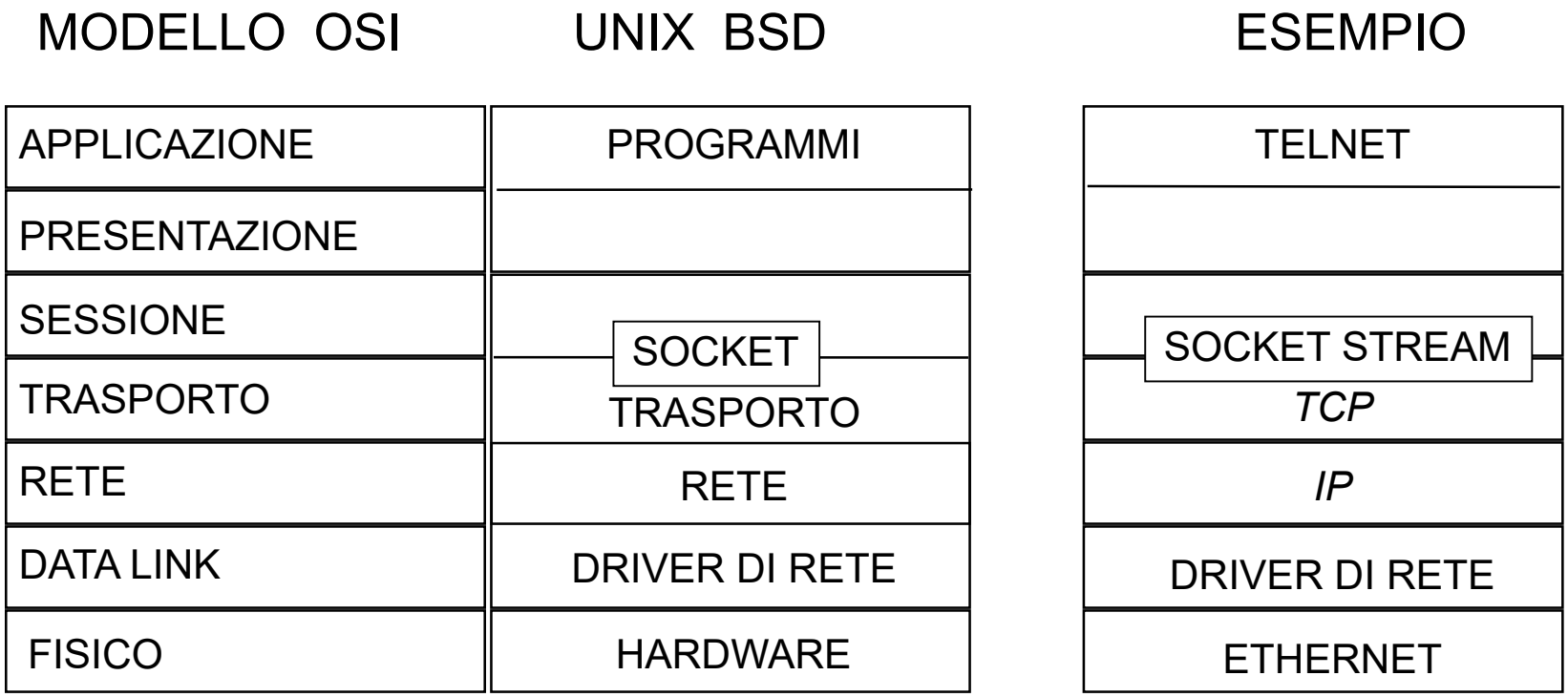
```
int connect(int sd, const struct sockaddr *sa, socklen_t len)
{
        struct sockaddr_in *sin;
        struct sockaddr_in6 *sin6;
        …
        if (len < sizeof(struct sockaddr)) return -1;
        switch (sa->sa_family) {
        case AF_INET:
                if (len < sizeof(struct sockaddr_in)) return -2;
                sin = (struct sockaddr_in *) sa;
                connect_ipv4(sd, sin); /* chiamo versione IPv4 della primitiva */
        case AF_INET6:
                if (len < sizeof(struct sockaddr_in6)) return -3;
                sin6 = (struct sockaddr_in6 *) sa;
                connect_ipv6(sd, sin6); /* chiamo versione IPv6 della primitiva */
        …
}
```

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.

Tipi di Socket

TIPO	descrizione
SOCK_STREAM	terminale di un canale di comunicazione **con 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 **senza connessione non affidabile** (socket datagram o UDP)
SOCK_RAW	accesso diretto al livello di rete (IP)
SOCK_DCCP	protocollo DCCP (~ UDP + congestion control)

DOMINI Internet (AF_INET e AF_INET6)



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), con 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), con semantica **may be**.

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)
- **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), definendo il protocollo usato dalla socket. In una connessione definisce:
- <**protocollo**; indirizzo IP locale; porta locale; indirizzo IP remoto; porta remota>

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>*

Associazione socket – indirizzi e porte

Host possono avere **più indirizzi locali**, ad esempio in caso di molteplici interfacce di rete

- in IPv4, indirizzo locale 0.0.0.0 (**INADDR_ANY**) per effettuare la bind su tutte le interfacce
  - in quali casi meglio non fornire un servizio di rete su tutte le interfacce? IP pubblico OK per Web server, forse no per database
- in IPv4, indirizzo locale 127.0.0.1 (**INADDR_LOOPBACK**) per effettuare la bind sul solo indirizzo di loopback (localhost)
  - socket non raggiungibile da processi di altri host

Porte nel range 0 – 65535 (2^16-1)

- **porte nel range 0 - 1023 sono "well-known"**, riservate e utilizzate per servizi di rete ben noti, ad esempio porta 80 per http.
- su sistemi operative Unix-like è necessario avere privilegi di root per effettuare la bind su una porta locale nel range 0 – 1023.

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

Comunicazione connection-oriented

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

LATO CLIENT
Uso di una **socket attiva**
Il client richiede una connessione al server (primitiva **connect**)

LATO SERVER
Uso di una **socket passiva** (listening socket)
Il server definisce una coda di richieste di connessione (primitiva **listen**) e attende le richieste (primitiva **accept**)
Quando arriva una richiesta, la richiesta viene accettata, stabilendo così la connessione. La comunicazione può quindi iniziare.

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****>*

Comunicazione connection oriented
(lato **Server**)

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

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

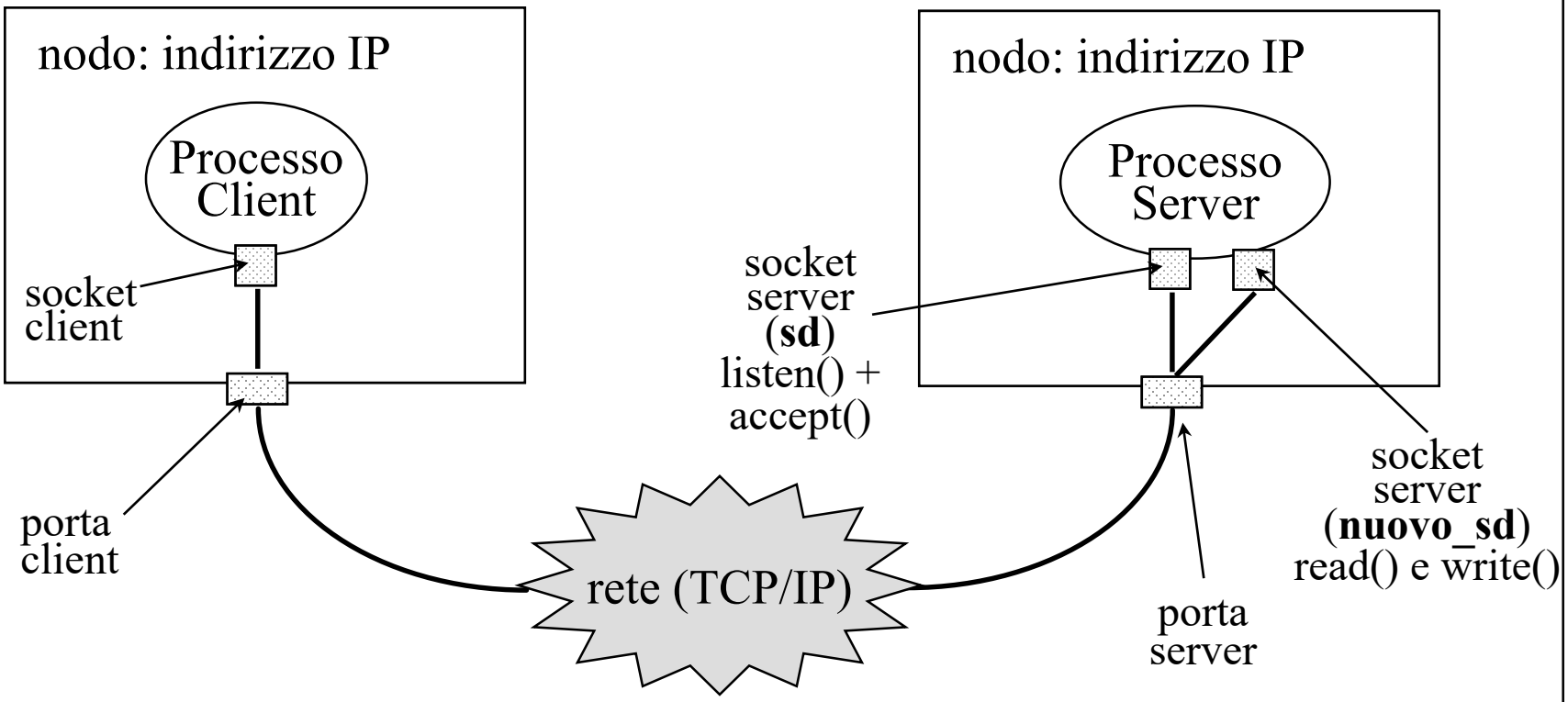
Estrae una richiesta di connessione dalla coda predisposta dalla listen().

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



Comunicazione connection oriented

Una volta completata la connessione si possono utilizzare le normali primitive **read** e **write** (uniformi alle omologhe funzioni su file)

```
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, non passiva!).

uso di primitive send() e recv() per sfruttare opzioni specifiche delle socket, invio di dati out-of-band.

Note su read e write via socket (1)

La scrittura/write consegna al driver i byte.
La lettura/read attende e legge almeno 1 byte disponibile, cercando di trasferire in spazio utente i byte arrivati sulla connessione.

write: i messaggi sono comunicati (ovvero inviati tramite uno specifico segmento TCP) ad ogni invocazione della primitiva?
NO

- i dati sono bufferizzati dal protocollo TCP: **non è detto che siano inviati subito** ma raggruppati e inviati poi alla prima comunicazione 'vera' decisa dal driver TCP
- soluzione per avere certezza dell'invio: messaggi di lunghezza pari al buffer o comandi espliciti di flush del buffer

Note su read e write via socket (2)

read: come preservare i messaggi in ricezione?

- ogni lettura/read restituisce i dati preparati dal driver locale: **TCP a stream di byte non implementa marcatori di fine messaggio**. Alcune possibili soluzioni:
  - messaggi a **lunghezza fissa**
  - messaggi a **lunghezza variabile con terminatori**, ad esempio "\r\n" al termine di ciascuna riga dell'header HTTP
  - messaggi a **lunghezza variabile senza terminatore**, si alternano un messaggio a lunghezza fissa e uno variabile in lunghezza (il primo contiene la lunghezza del secondo), ossia uso di messaggi con indicatori espliciti di lunghezza letti con due letture in successione

Ricezione di un messaggio di lunghezza specificata

```
int ricevi (int sd, char* buf, int n){
   int i, j;
   i = recv (sd, buf, n, 0);
   if ( i != n && i != 0) {
      if (i == -1) {
         fprintf(stderr, "%errore in lettura\n"); exit(1); }
      while (i < n) {
         j = recv (sd, &buf[i], n-i, 0);
         if (j == -1) {
            fprintf(stderr,"errore in lettura\n"); exit(1); }
         i += j;
         if (j == 0) break;
      }
   } /* si assume che tutti i byte arrivino… se si verifica il fine file si esce */
   return i;
}
```

La recv ritorna appena vi sono dati e non attende tutti i dati richiesti.
Il ciclo interno verifica che la recv() non ritorni un messaggio più corto di quello atteso (n byte) e garantisce la ricezione fino al byte richiesto.

Ricezione Messaggi

Nelle applicazioni Internet è molto comune trovare funzioni come quella appena vista di ricezione.
Dobbiamo ricordare che **la ricezione considera un successo un qualunque numero di byte ricevuto (anche 1)** e ne segnala il numero nel risultato.
Per evitare problemi, dobbiamo fare ricezioni/letture ripetute.

In ogni caso, in ricezione dobbiamo sempre verificare la dimensione del messaggio, se nota, o attuare un protocollo per conoscerla durante l'esecuzione, per aspettare l'intero messaggio significativo.

Per messaggi di piccola dimensione la frammentazione di un messaggio applicativo in molteplici segmenti TCP è improbabile, ma con dimensioni superiori (qualche Kbyte), il pacchetto può venire suddiviso dai livelli sottostanti, e una ricezione parziale diventa più probabile.

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)

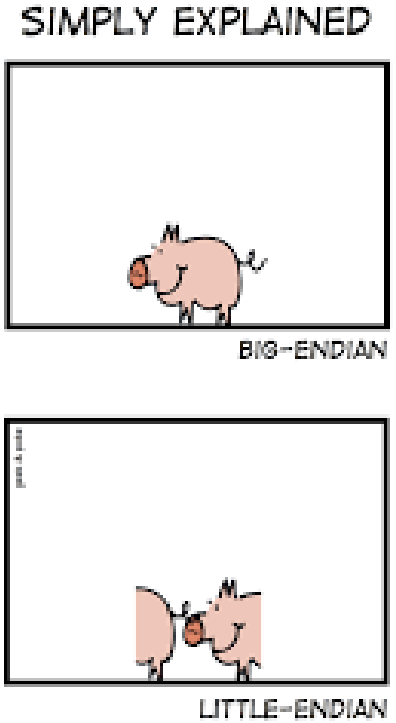
Ordinamento di byte in memoria (1)

I numeri composti da più byte (ad esempio 2 byte per i numeri a 16 bit, 4 byte per quelli 32 bit) possono essere rappresentati in memoria, salvati su storage persistente o trasmessi via rete secondo due modalità di ordinamento.

Big-endian byte order:
byte di ordine più alto nell'indirizzo iniziale

Little-endian byte order:
byte di ordine più basso nell'indirizzo iniziale

Esempio con intero di 2 byte
53482 decimale, 1101000011101010 binario
Big-endian: 11010000 11101010
Little-endian: 11101010 11010000



Ordinamento di byte in memoria (2)

Non c'è standard sull'adozione dell'uno o dell'altro.

Host Byte Order (HBO): modalità di ordinamento di byte usata in un sistema operativo (rilevabile con semplici programmi di testing)
Ad esempio: Linux → Little-endian, Solaris → Big-endian

Network Byte Order (NBO): modalità di ordinamento di byte usata da un protocollo di rete, ad esempio Protocolli Internet → Big-endian

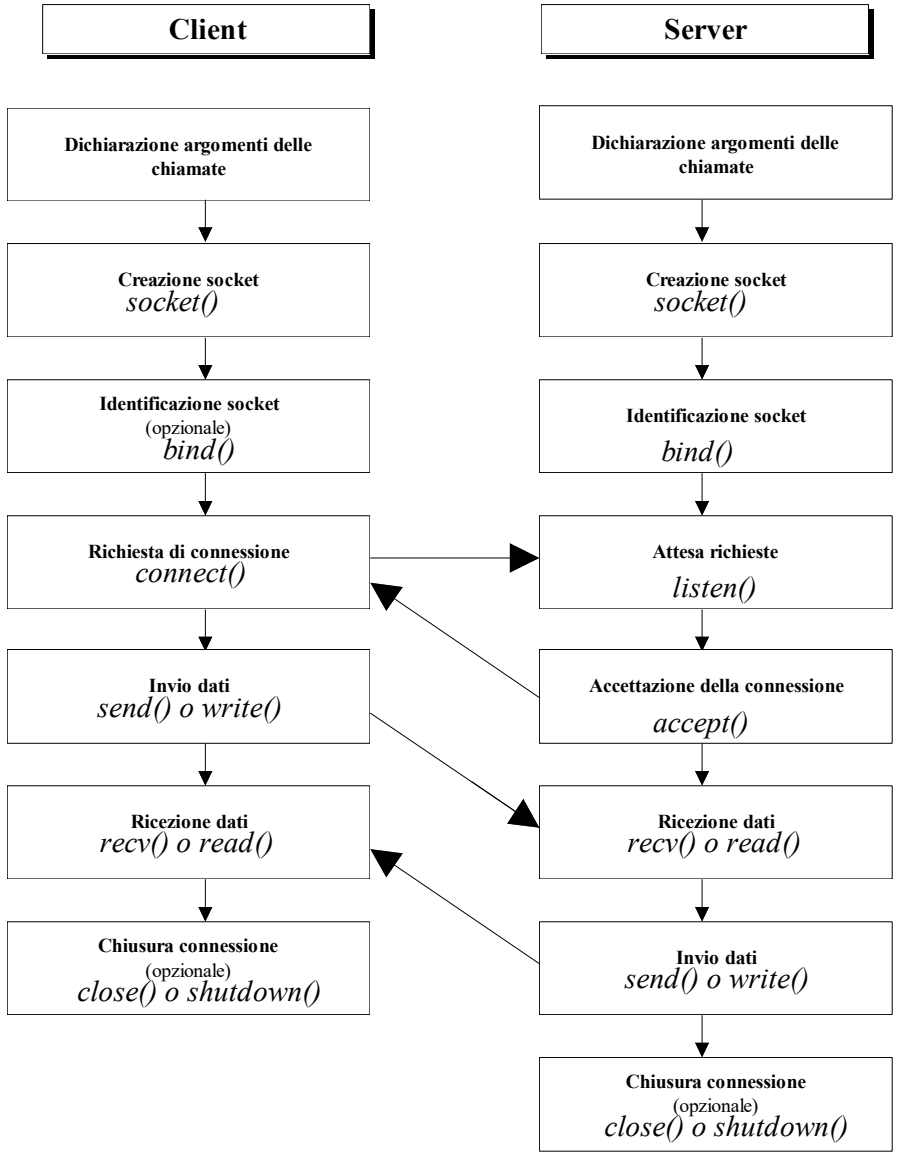
Domanda: HBO e NBO coincidono o sono distinti? Dipende!

Funzioni di manipolazione del byte order

- htons() e htonl() convertono da HBO (qualunque esso sia) a NBO (big-endian) un valore a 16/32 bit
- ntohs() ntohl() convertono da NBO (Big-endian) a HBO (qualunque esso sia) un valore a 16/32 bit

Nel caso in cui HBO e NBO coincidano lasciano inalterato il byte ordering

Comunicazione connection-oriented



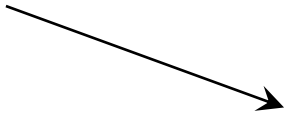
Esempio: Dominio **AF_INET**, socket stream (con connessione) **(Client)**

```
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_len = 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!

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

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.

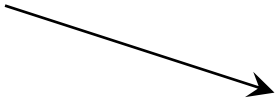
Esempio: Dominio **AF_INET6**, socket stream (con connessione) **(Client)**

```
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_len = 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!

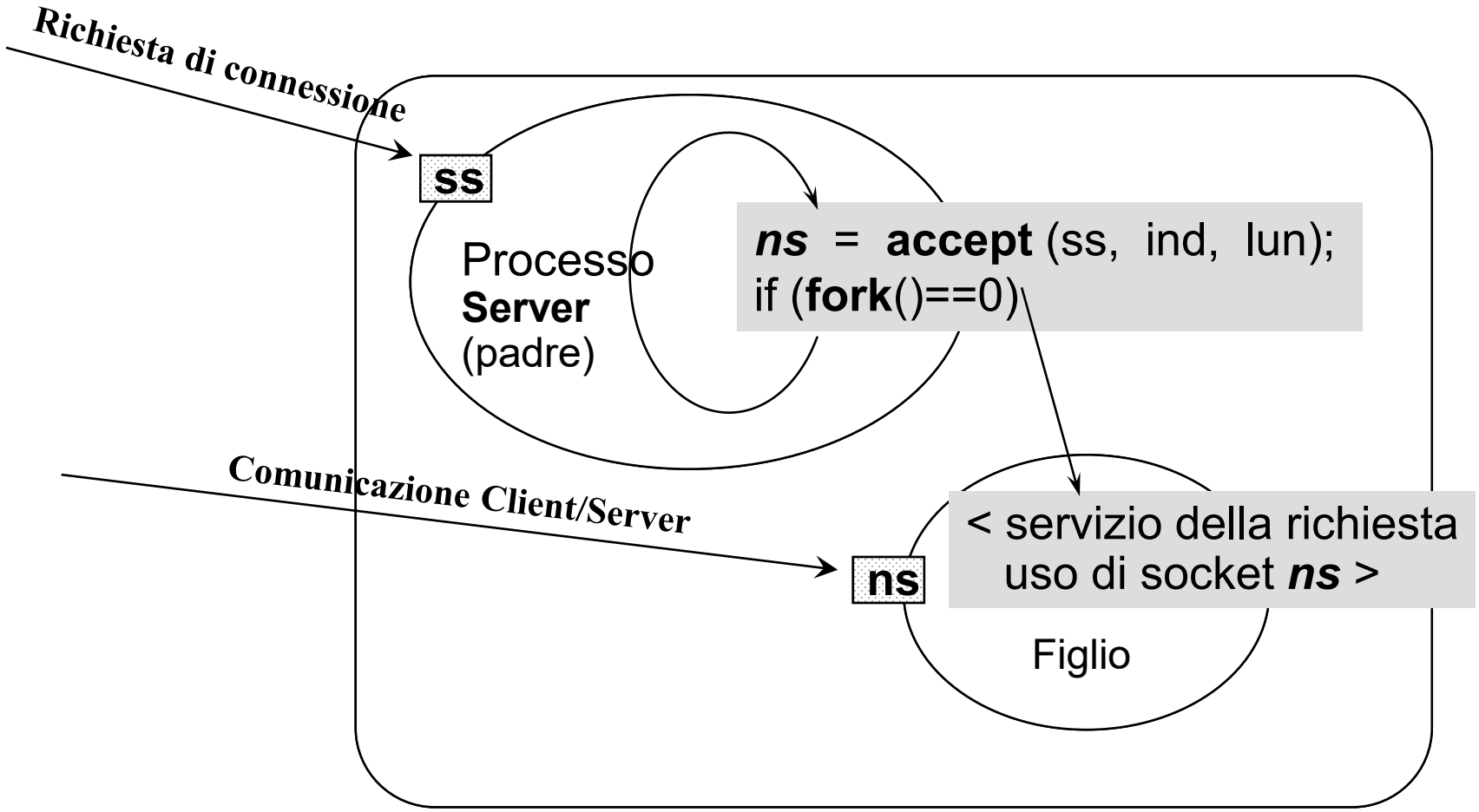
Esempio: Dominio AF_INET6, server

```
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_len = 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

Server *concorrente* multi-processo connection-oriented



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

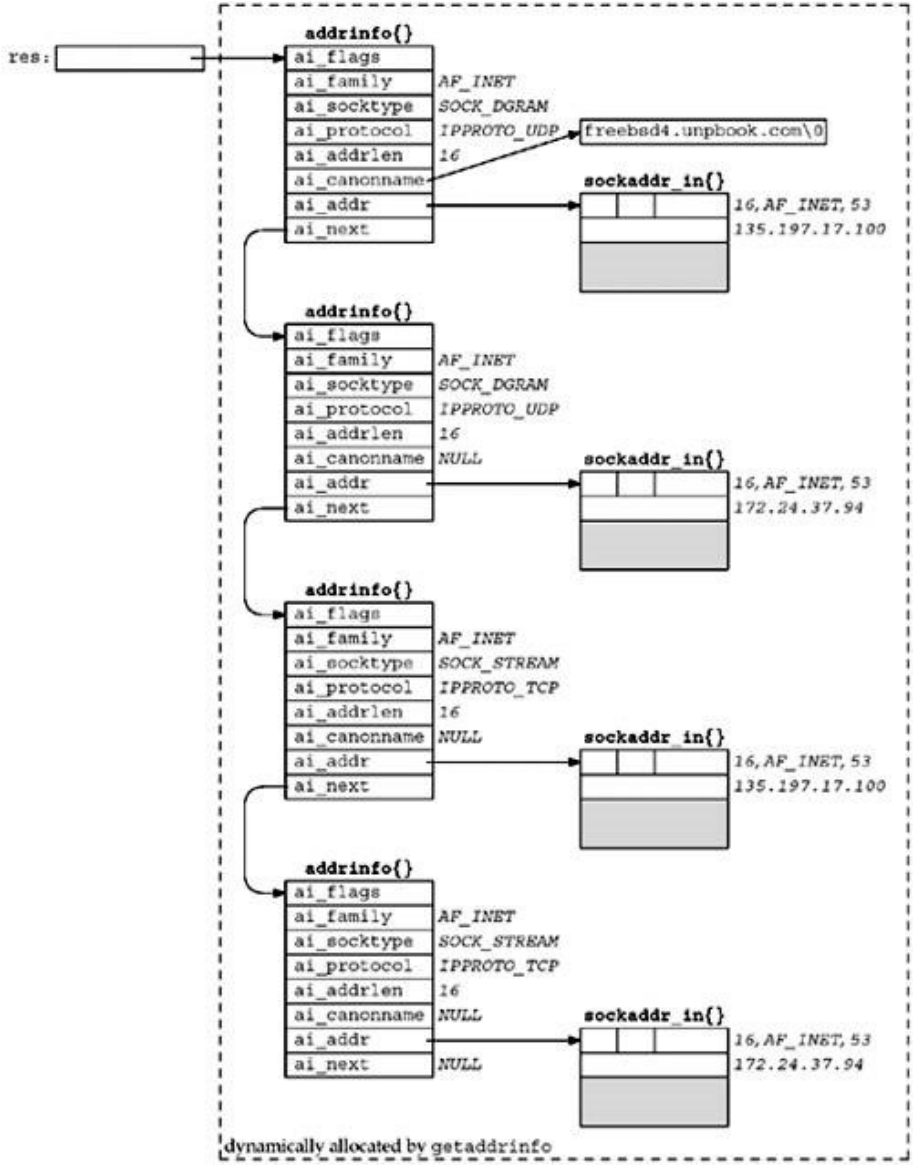
- riceve in ingresso: nome logico di host Internet; servizio a cui connettersi (numero di porta); varie preferenze (in *hints*)
- restituisce in uscita: una lista di strutture **addrinfo** in cui ciascun nodo rappresenta uno degli indirizzi fisici del servizio

```
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 */
};
```

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);
}
```



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, IPv4 address record).

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

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 terminatone 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!

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.

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 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. Quale il timeout di default in fase di connect?

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

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 deprecatao 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 IPv6,
                              AF_UNSPEC per comunicare sia con IPv4
                              che con 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!*/
…
```

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 di portabilità nel caso hints.ai_family = AF_UNSPEC

Questo problema si può facilmente risolvere su Linux e OS X modificando la configurazione di default del file `/etc/gai.conf`. Tuttavia, nel caso volessimo sviluppare server IPv6-enabled portabili anche su sistemi operativi con implementazioni più restrittive dello stack di rete, come OpenBSD e Windows, dovremmo utilizzare un'architettura significativamente più complicata (due socket passive, una per IPv4 e una per IPv6 e uso di **select** prima di **accept** per attendere simultaneamente connessioni su entrambe le socket).

Per ulteriori informazioni si veda il post:
`http://www.tortonesi.com/blog/2015/03/13/fixing-getaddrinfo/`

Esercizio: copia remota di un file (rcp)

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

rcp nodoserver nomefile

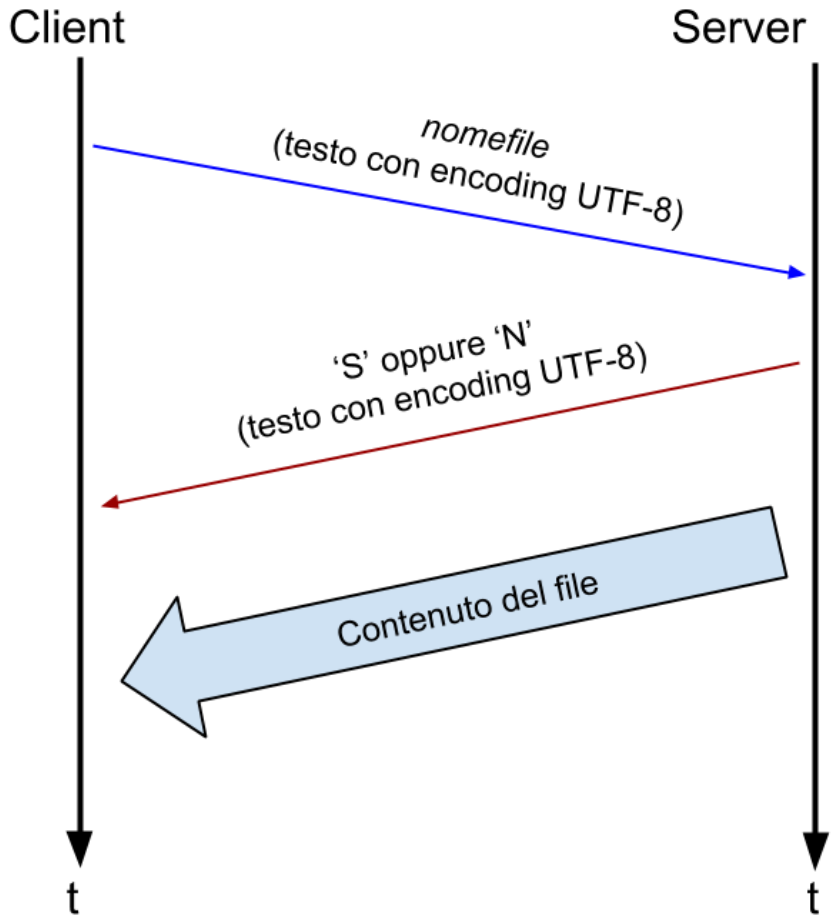
dove ***nodoserver*** specifica l'indirizzo logico del nodo contenente il processo Server e ***nomefile*** rappresenta il nome assoluto di un file nella macchina Server.

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 (bind) alla porta 50001.

Protocollo Applicativo



Esercizio: rcp - 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;
```

Esercizio: rcp - 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");
...
```

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);
}
```

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);
```

Unica read() per leggere il nome del file
solo perché il nome del file è breve,
altrimenti avremmo dovuto (ad esempio)
inviare prima la lunghezza del nome del file (byte? char? quanti?)
poi il nome del file vero e proprio.

```
                close(fd);
        }
         close(ns);
}
```

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? */
```

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-based

(caratteristiche che derivano dalla semantica del sottostante protocollo UDP)

Comunicazione connectionless
(socket DATAGRAM)

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

int **recvfrom**(int sd, char *buf, int len, int flags,
struct sockaddr *from, int *fromlen);

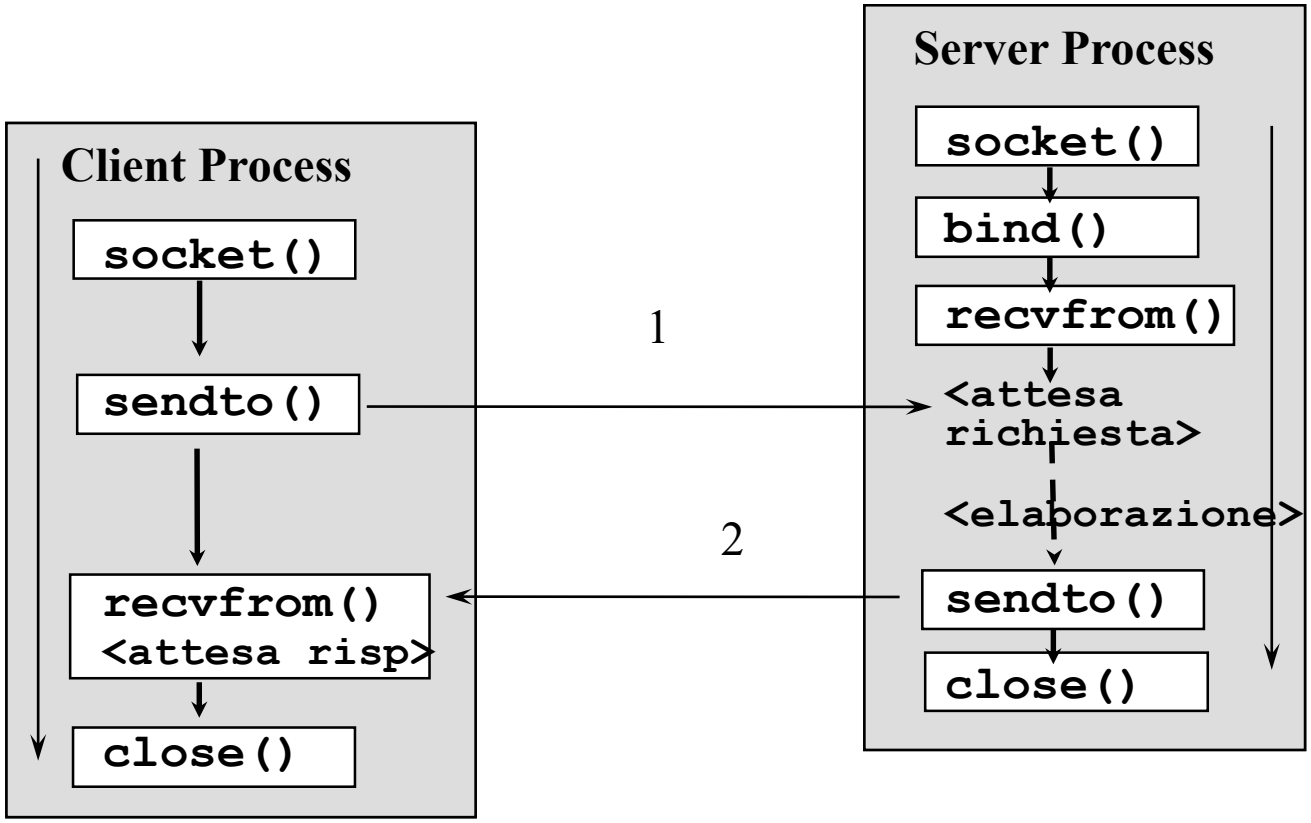
Le primitive recvfrom() e sendto(), oltre agli stessi parametri delle read e write, hanno anche due parametri aggiuntivi che denotano **indirizzo** e **lunghezza** di una struttura socket:

- 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 datagram) di dimensione zero.

Comunicazione senza connessione
(socket DATAGRAM)



Esercizio: echo UDP - Lato Server

```
int sockfd, n; socklen_t len; char 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);
   n = recvfrom(sd, mesg, MAXLINE, 0, &client_address, &len);
   sendto(sd, mesg, n, 0, &client_address, len);
}
```

Per migliorare la leggibilità si possono creare delle semplici funzioni Bind, Socket, Sendto, etc. che incapsulano le rispettive system call bind, socket, sendto, etc. e gestiscono gli errori (Stevens).

Note conclusive sulle socket UDP

Ricordarsi che:

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

Quale tipo di Socket utilizzare?

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, congestion avoidance, slow start

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**.

Esempio invio multiplo (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.

Esempio invio multiplo - Lato Client

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

hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_DGRAM;

for (i=1; i<argc-2; 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;
```

Esempio invio multiplo - 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) { … }
sd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
if (sd < 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, &clnt_addr, &len);
  write(fd, buff, cc);
}
close(sock); return 0;
```

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 o 1 o 0)

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

int getsockopt (int sockfd, int level, int optname, void *optval, sock_len *optlen);
int setsockopt (int sockfd, int level, int optname, const void *optval, sock_len optlen);
```

sockfd = socket descriptor
level = tipo di protocollo (per socket SOL_SOCKET)
optname = nome dell'opzione
optval = puntatore a un'area di memoria per valore
optlen = lunghezza (o puntatore) quarto argomento

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_PCVLOWAT	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_KEEPALIVE	Controllo periodico connessione
SO_PROTOTYPE	setta tipo di protocollo

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.*

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

Altre Opzioni per le Socket

Controllo periodico della connessione
Il protocollo di trasporto può inviare messaggi di controllo periodici (simili ad ACK duplicati) 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));
```