



# Laboratorio di Reti Informatiche

Corso di Laurea Triennale in Ingegneria Informatica  
A.A. 2017/2018

**Ing. Carlo Vallati**  
**carlo.vallati@unipi.it**



# Esercitazione 4

Programmazione con i socket – Parte 1



# Programma di oggi

- Introduzione al linguaggio C
- Programmazione con i socket



# Introduzione al linguaggio C

Attraverso le principali differenze tra C e C++

# Definizione di variabili

- Le variabili possono essere definite solo all'inizio di un blocco (standard C89):

## Stile C++

```
int main() {  
    int a = 5, i, b;  
    a = func(1000);  
    int b = f(a);  
    // ...  
    for (i = 0; a < 100; ++i) {  
        b = f(a);  
        int c = 0;  
        // ...  
    }  
}
```

## Stile C

```
int main() {  
    int a = 5, i, b;  
    int b = f(a);  
    a = func(1000);  
    // ...  
    for (i = 0; a < 100; ++i) {  
        int c = 0;  
        b = f(a);  
        // ...  
    }  
}
```

# Strutture

- Si deve specificare sempre **struct** quando si crea una variabile di tipo struttura:

## Stile C++

```
struct Complex {  
    double Re;  
    double Im;  
};  
  
int main() {  
    int a = 4;  
    Complex c;  
    // ...  
}
```

## Stile C

```
struct Complex {  
    double Re;  
    double Im;  
};  
  
int main() {  
    int a = 4;  
    struct Complex c;  
    // ...  
}
```



# Memoria dinamica

```
#include <stdlib.h>

int main() {
    int mem_size = 5;
    void *ptr;
    ptr = malloc(mem_size);
    if (ptr == NULL) {
        // Gestione errore
    }
    // ...
    free(ptr);
}
```

man malloc  
man 3 free



# Operazioni di I/O

```
#include <stdio.h>

char *str = "Hello!\n";
printf(str);
printf("str = %s", str);

printf("Hello World!\n");

int i = 5;
printf("i = %d\n", i);

scanf("%d", &i);
printf("i = %d\n", i);
```

```
man 3 printf
man scanf
man stdio
```





# Stringhe

```
#include <string.h>

// Lunghezza
char *str1 = "Hello \n";
int len;
len = strlen(str1);

// Confronto
char *str2 = "World!\n";
int ret;
ret = strcmp(str1, str2);
```

H	e	l	l	o		\n	\0
---	---	---	---	---	--	----	----

8 byte allocati, ma `strlen()` restituisce 7!

man `strlen`  
man `strcmp`

`ret = 0` se le stringhe sono identiche  
`ret < 0` se `str1` è alfabeticamente minore di `str2`  
`ret > 0` se `str1` è alfabeticamente maggiore di `str2`

# Stringhe



```
#include <string.h>
```

```
// Copia
```

```
char str[20];  
n = sizeof(str);  
strncpy(str, "Hello \n", n);
```

```
// Concatenazione
```

```
char *str2 = "World!\n";  
strncat(str1, str2, 8);
```

```
man strncpy  
man strncat
```

- Fare attenzione a copiare anche il terminatore di stringa
- Le funzioni non controllano che ci sia spazio nella stringa di destinazione

H	e	l	l	o		\n	W	o	r	l	d	!	\n	\0
---	---	---	---	---	--	----	---	---	---	---	---	---	----	----



# Gestione dei file

```
#include <stdio.h>
```

```
// Apertura file  
FILE *fd;  
fd = fopen("/tmp/foo.txt", "r" );  
if (fd == NULL) {  
    // Gestione errore  
}
```

**man fopen**

Si specifica un percorso (assoluto o relativo) e una modalità di apertura:

- **r** = sola lettura
- **w** = sola scrittura
- **r+** = lettura e scrittura
- **a** = *append*
- **a+** = lettura e *append*

Se il file non esiste e si specifica scrittura o *append*, il file viene creato



# Gestione dei file

```
#include <stdio.h>

// Lettura file
int ret, n;
FILE *fd1;
fd1 = fopen("/tmp/foo.txt", "r");
ret = fscanf(fd1, "%d", &n);

// Scrittura file
char *str = "Hello!\n";
FILE *fd2;
fd2 = fopen("/tmp/bar.txt", "w");
ret = fprintf(fd, "%s", str);
```

`man fscanf`  
`man fprintf`

Si comportano allo stesso modo di `scanf()` e `printf()`, ma usano il file specificato invece di `stdin` e `stdout`



# Gestione dei file

```
#include <stdio.h>
#include <sys/stat.h>

// Dimensione
int ret, size;
struct stat info;
ret = stat("/tmp/foo.txt", &info);
size = info.st_size;

// Chiusura file
FILE *fd;
fd = fopen("/tmp/bar.txt", "w");
fclose(fd);
```

man 2 stat  
man fprintf

Non serve aprire il file  
per usare `stat()`



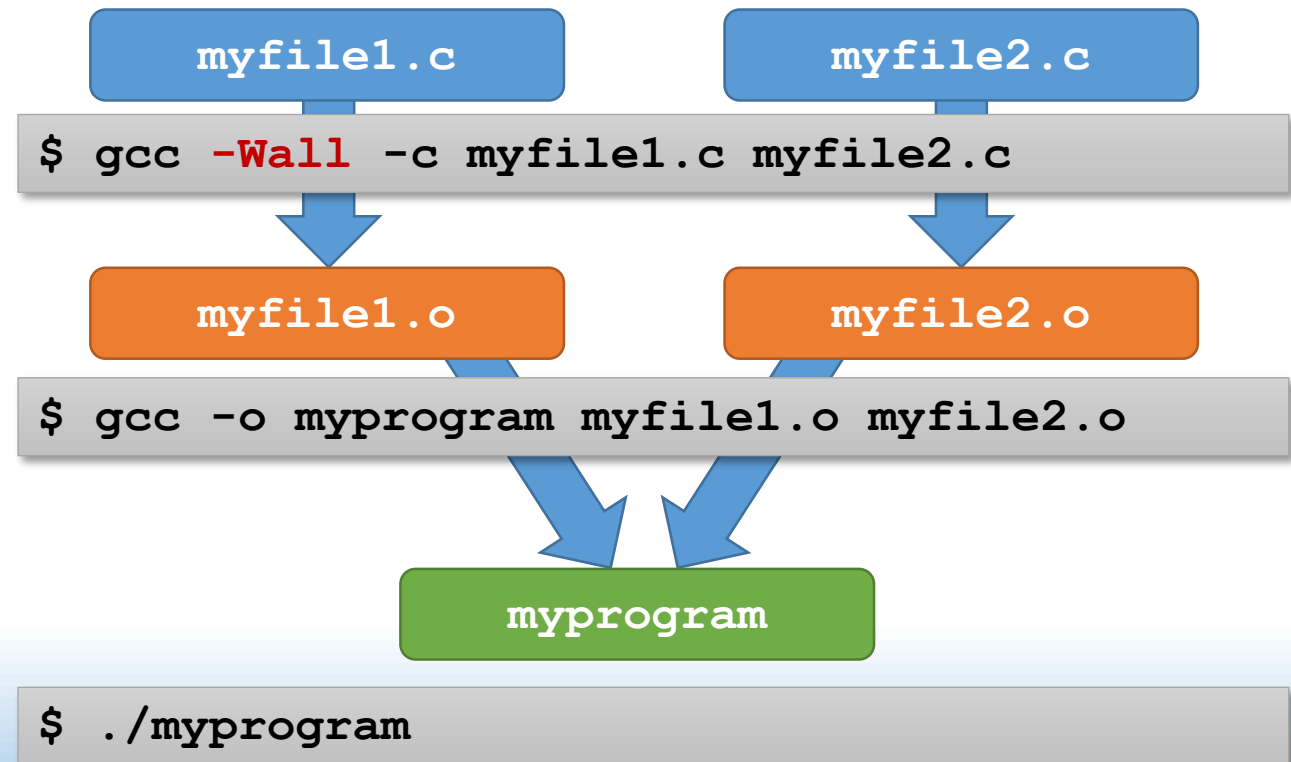
# Compilazione

- Useremo la *GNU Compiler Collection* (GCC)

`man gcc`

Con la **compilazione**, si creano i *file oggetto* a partire dai *file sorgente*

Con il **linking**, si crea un *file eseguibile* a partire dai *file oggetto*





# Programmazione distribuita



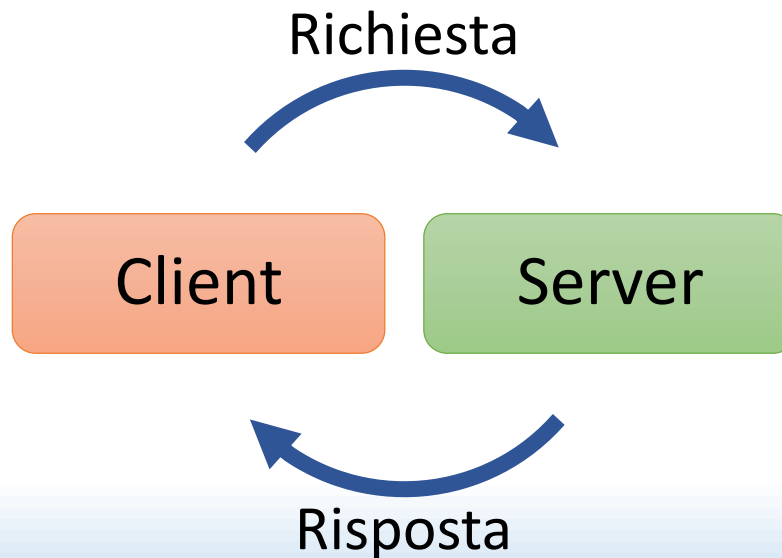
# Cooperazione tra processi

- Due processi possono essere:
  - Indipendenti
  - **Cooperanti**
    - Sulla stessa macchina
    - Su macchine diverse (**sistema distribuito**)
- Due processi possono cooperare attraverso:
  - Sincronizzazione (Es. semafori)
  - **Comunicazione**, cioè scambio di informazioni
    - Memoria condivisa
    - Chiamate a procedura remota
    - **Scambio di messaggi**

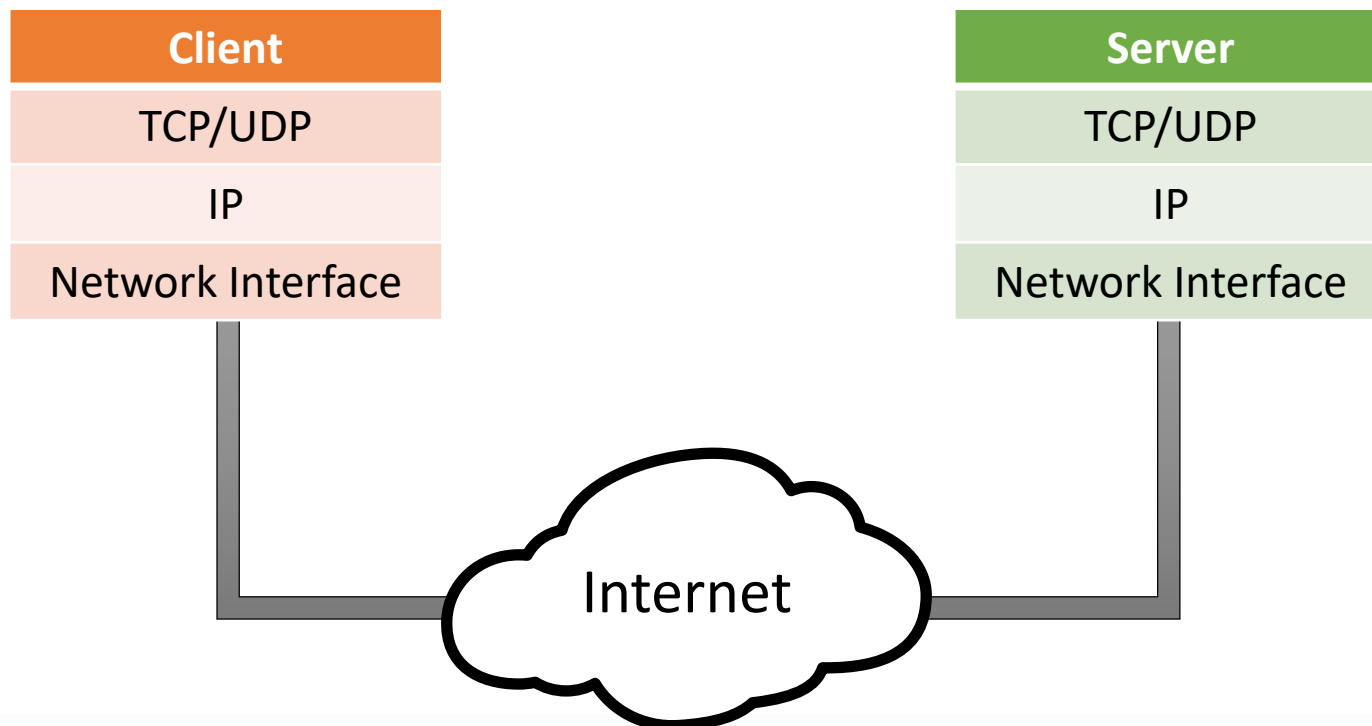


# Client/server

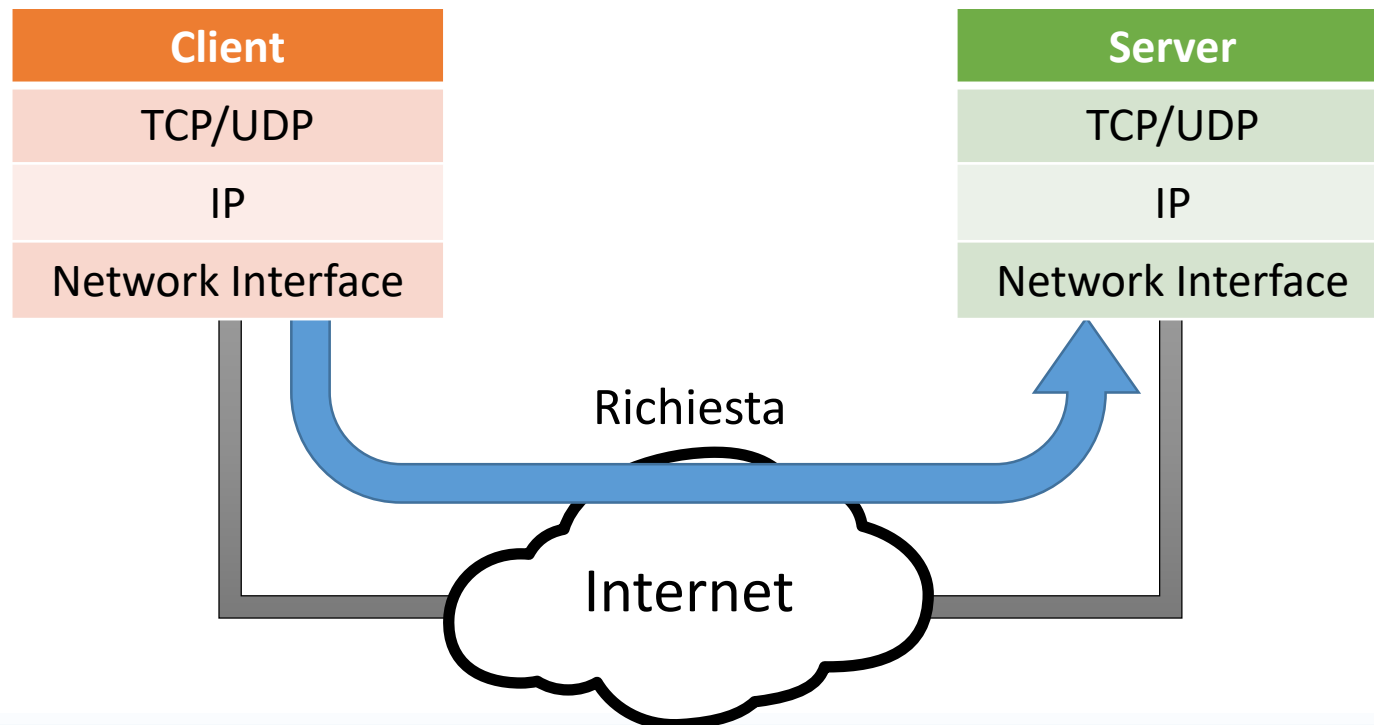
- Paradigma basato su scambio di messaggi
- Viene usato principalmente per sistemi distribuiti



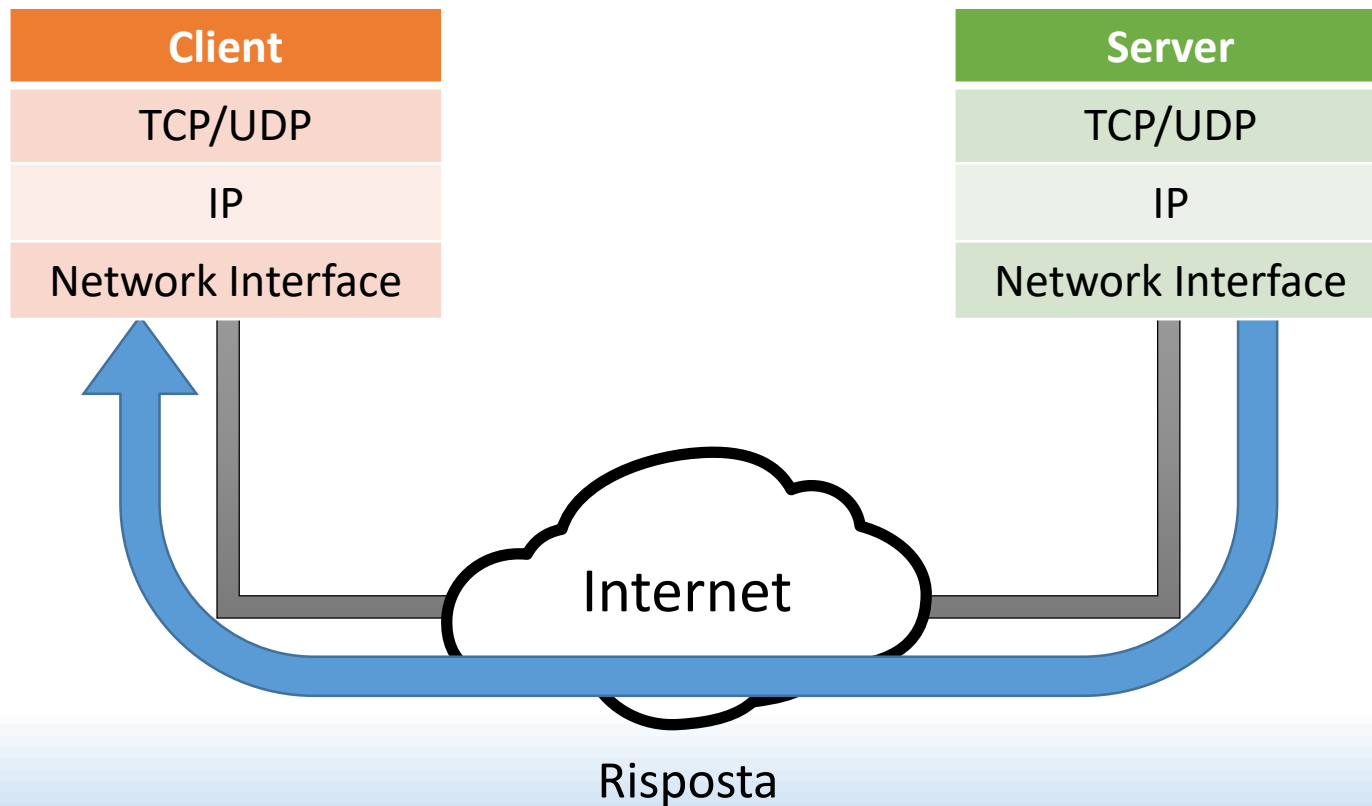
# Client/server



# Client/server



# Client/server



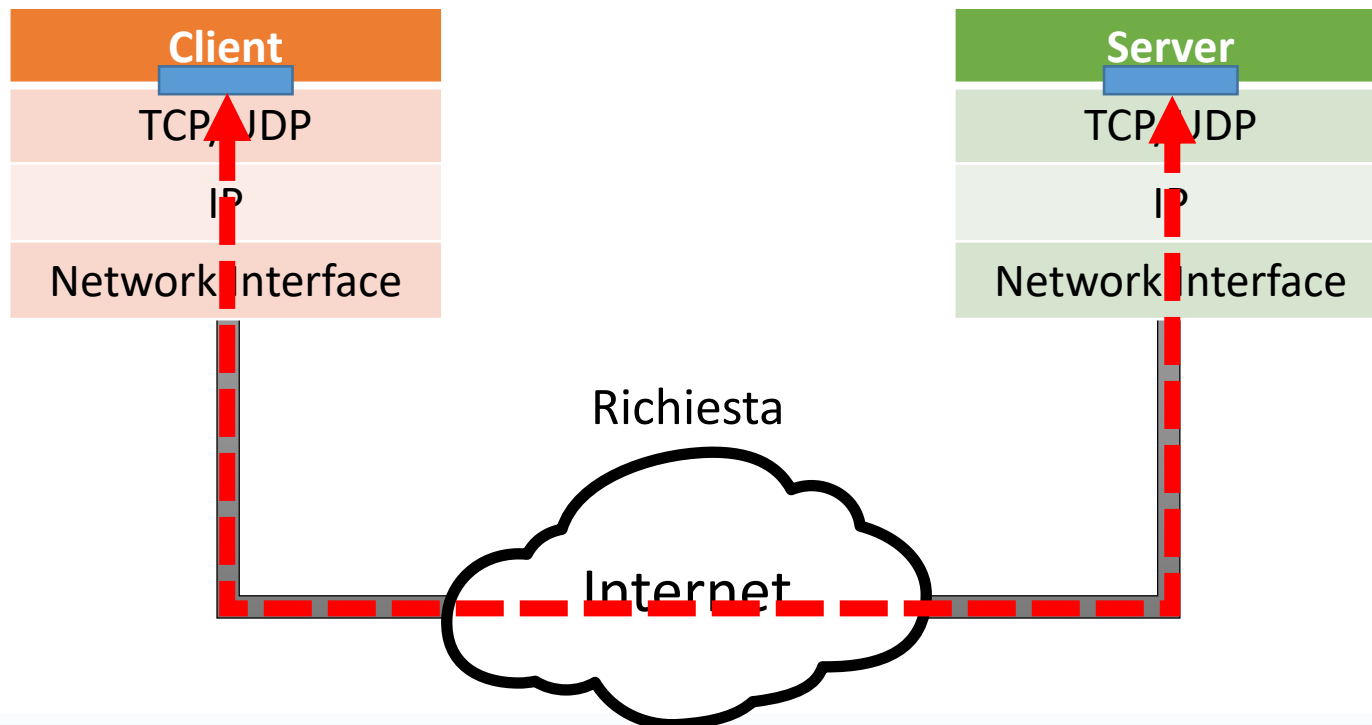


# Socket

- Meccanismo per la comunicazione tra processi
  - Sulla stessa macchina o su macchine differenti
- È un'**astrazione**:
  - Un interfaccia unica per operare con diversi protocolli di rete
  - Nasconde i dettagli dei livelli sottostanti
- Un socket è identificato da un *indirizzo*:
  - Indirizzo host (TCP/IP: **Indirizzo IP**)
  - Indirizzo processo (TCP/IP: **Numero di porta**)

# Socket

Un socket è l'estremità di un canale di comunicazione





# Socket

- L'astrazione di socket è implementata dal SO
- Abbiamo a disposizione delle **primitive** per:
  - Creare un socket
  - Assegnargli un indirizzo
  - Connettersi a un altro socket
  - Accettare una connessione
  - Inviare e ricevere dati attraverso i socket
  - ...



# Primitiva **socket** ()

- **Crea un socket**

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

```
man 7 ip
man 2 socket
man 7 socket
```

```
int socket(int domain, int type, int protocol);
```

- **domain**: famiglia di protocolli da utilizzare
  - **AF\_LOCAL** – Comunicazione locale
  - **AF\_INET** – Protocolli IPv4, TCP e UDP
- **type**: tipologia di socket
  - **SOCK\_STREAM** – Connessione affidabile, bidirezionale (TCP) - Oggi vedremo questo
  - **SOCK\_DGRAM** – Invio di pacchetti senza connessione (UDP)
- **protocol**: sempre a 0
- La funzione restituisce un *descrittore di file* (oppure -1 su errore)
- **Attenzione**: il socket *non* è ancora associato a un indirizzo IP o a una porta





# Strutture per gli indirizzi

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

man 7 ip

```
struct sockaddr_in {
    sa_family_t    sin_family; /* address family: AF_INET */
    in_port_t      sin_port;   /* port in network byte order */
    struct in_addr sin_addr;   /* internet address */
};

/* Internet address. */
struct in_addr {
    uint32_t        s_addr;    /* address in network byte order */
};
```

**Attenzione:** esiste anche la struttura **struct sockaddr** usata da alcune funzioni descritte più avanti ma non la useremo direttamente.



# Ordine dei byte

- Calcolatori diversi possono usare modalità diverse per **ordinare i byte** all'interno di una *word*
- Esempio: numero **422990**, in 32 bit (4 byte)
  - 1100111010001001110

- **Memorizzazione *big-endian*** - Per primo il byte più significativo (MSB)

Indirizzo A	Indirizzo A+1	Indirizzo A+2	Indirizzo A+3
0000 0000	0000 0110	0111 0100	0100 1110

- **Memorizzazione *little-endian*** - Per primo il byte meno significativo (LSB)

Indirizzo A	Indirizzo A+1	Indirizzo A+2	Indirizzo A+3
0100 1110	0111 0100	0000 0110	0000 0000



# Ordine dei byte

- Il formato di rete (*network order*), usato nei socket, è *big-endian*
- Il formato dell'host (*host order*) dipende dal singolo host
- Funzioni di conversione:

**man 3 byteorder**

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

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

**Attenzione:** `uint16_t` e `uint32_t` sono **interi senza segno** che occupano **sempre** 16 e 32 bit a prescindere dal calcolatore usato per compilare. Sono utili quando si vuole avere il controllo completo della dimensione delle variabili, come in questo caso. Sono definiti nell'header `<stdint.h>`.



# Formato degli indirizzi

- Formato *numerico*: 32-bit, usato dal computer
- Formato *presentazione*: stringa in notazione decimale puntata

```
int inet_pton(int af, const char *src, void *dst);
```

- **af**: famiglia (**AF\_INET**)
- **src**: stringa del tipo "ddd.ddd.ddd.ddd"
- **dst**: puntatore a una **struct in\_addr**

```
man inet_pton  
man inet_ntop
```

```
const char *inet_ntop(int af, const void *src,  
                      char *dst, socklen_t size);
```

- **af**: famiglia (**AF\_INET**)
- **src**: puntatore a una **struct in\_addr**
- **dst**: puntatore a un buffer di caratteri lungo **size**
- **size**: deve valere almeno **INET\_ADDRSTRLEN**



# Iniziamo a fare qualcosa

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

int main () {
    /* Creazione socket */
    int sd = socket(AF_INET, SOCK_STREAM, 0);

    /* Creazione indirizzo */
    struct sockaddr_in my_addr;
    memset(&my_addr, 0, sizeof(my_addr)); // Pulizia
    my_addr.sin_family = AF_INET ;
    my_addr.sin_port = htons(4242);
    inet_pton(AF_INET, "192.168.4.5", &my_addr.sin_addr);
```



# Programmazione distribuita

Lato server



# Primitiva **bind()**

- **Assegna** un socket a un indirizzo
  - Viene usata dal *server* per specificare indirizzo e porta sui quali ricevere richieste
  - Di solito, il client non ha bisogno di eseguire la **bind()**

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

**man 2 bind**

- **sockfd**: descrittore del socket
- **addr**: puntatore alla struttura di tipo **struct sockaddr**
  - Visto che usiamo **struct sockaddr\_in** bisogna convertire il puntatore
- **addrlen**: dimensione di **addr**
- La funzione restituisce 0 se ha successo, -1 su errore

```
ret = bind(sd, (struct sockaddr*)&my_addr, sizeof(my_addr));
```



# Primitiva **listen()**

- Specifica che il socket è **passivo**, cioè verrà usato per ricevere richieste di connessione
  - Come vedremo, si possono mettere in attesa solo i socket **SOCK\_STREAM**

```
int listen(int sockfd, int backlog);
```

**man 2 listen**

- **sockfd**: descrittore del socket
- **backlog**: dimensione della coda, cioè quante richieste dai client possono rimanere in attesa di essere gestite
- La funzione restituisce 0 se ha successo, -1 su errore

```
ret = listen(sd, 10);
```





# Primitiva **accept()**

- **Accetta una** richiesta di connessione pervenuta sul socket
  - Anche in questo caso, ha senso solo sui socket **SOCK\_STREAM**

```
int accept(int sockfd, struct sockaddr *addr,  
           socklen_t *addrlen);
```

man 2 accept

- **sockfd**: descrittore del socket
- **addr**: puntatore a una struttura (vuota) di tipo `struct sockaddr`
  - Qui ci viene salvato l'indirizzo del client
- **addrlen**: dimensione di **addr**
- La funzione restituisce il **descrittore di un nuovo socket** che verrà usato per la comunicazione, -1 su errore
- **La funzione è bloccante**: il programma si ferma finché non arriva una richiesta

```
struct sockaddr_in cl_addr;  
int len = sizeof(cl_addr);  
new_sd = accept(sd, (struct sockaddr*)&cl_addr, &len);
```



# Codice del server

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

int main () {
    int ret, sd, new_sd, len;
    struct sockaddr_in my_addr, cl_addr; // Due strutture!
    /* Creazione socket */
    sd = socket(AF_INET, SOCK_STREAM, 0);
    /* Creazione indirizzo */
    memset(&my_addr, 0, sizeof(my_addr)); // Pulizia
    my_addr.sin_family = AF_INET ;
    my_addr.sin_port = htons(4242);
    inet_pton(AF_INET, "192.168.4.5", &my_addr.sin_addr);
    // In alternativa: my_addr.sin_addr.s_addr = INADDR_ANY;
    ret = bind(sd, (struct sockaddr*)&my_addr, sizeof(my_addr));
    ret = listen(sd, 10);
    int len = sizeof(cl_addr);
    new_sd = accept(sd, (struct sockaddr*)&cl_addr, &len);
```



# Programmazione distribuita

Lato client



# Primitiva **connect** ()

- **Connette** il socket a un indirizzo remoto
  - Cioè a un altro socket

```
int connect(int sockfd, const struct sockaddr *addr,  
            socklen_t addrlen);
```

**man 2 connect**

- **sockfd**: descrittore del socket (locale)
- **addr**: puntatore alla struttura contenente l'indirizzo del server
- **addrlen**: dimensione di **addr**
- La funzione restituisce 0 se ha successo, -1 su errore
- **La funzione è bloccante**: il programma si ferma finché la richiesta di connessione non è stata accettata

```
ret = connect(sd, (struct sockaddr*)&sv_addr, sizeof(sv_addr));
```



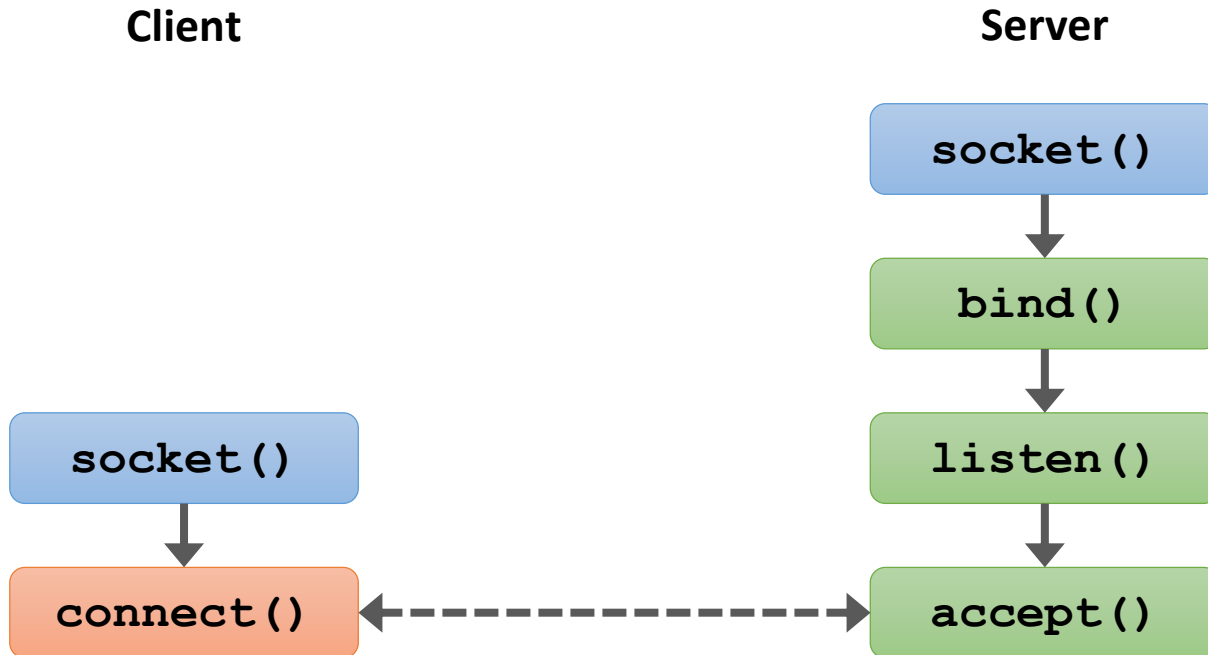
# Codice del client

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

int main () {
    int ret, sd;
    struct sockaddr_in sv_addr; // Struttura per il server
    /* Creazione socket */
    sd = socket(AF_INET, SOCK_STREAM, 0);
    /* Creazione indirizzo del server */
    memset(&sv_addr, 0, sizeof(sv_addr)); // Pulizia
    sv_addr.sin_family = AF_INET ;
    sv_addr.sin_port = htons(4242);
    inet_pton(AF_INET, "192.168.4.5", &sv_addr.sin_addr);

    ret = connect(sd, (struct sockaddr*)&sv_addr, sizeof(sv_addr));
```

# Cosa succede





# Programmazione distribuita

Scambio di dati



# Primitiva **send()**

- Invia un messaggio attraverso un socket **connesso**

```
ssize_t send(int sockfd, const void *buf, size_t len,  
             int flags);
```

man 2 send

- **sockfd**: descrittore del socket
- **buf**: puntatore al buffer contenente il messaggio da inviare
- **len**: dimensione in byte del messaggio
- **flags**: per settare delle opzioni, lasciamolo a 0
- La funzione restituisce il **numero di byte inviati**, -1 su errore
- **La funzione è bloccante**: il programma si ferma finché non ha scritto tutto il messaggio





# Primitiva **send()**

```
int ret, sd, len;
char buffer[1024];

//...

strcpy(buffer, "Hello Server!");
len = strlen(buffer);
ret = send(sd, (void*)buffer, len, 0);
if (ret < len) {
    // Gestione errore
}
```



# Primitiva **recv()**

- Preleva un messaggio da un socket **connesso**

```
ssize_t recv(int sockfd, const void *buf, size_t len,  
             int flags);
```

man 2 recv

- **sockfd**: descrittore del socket
- **buf**: puntatore al buffer in cui salvare il messaggio
- **len**: dimensione in byte del messaggio desiderato
- **flags**: per settare delle opzioni
- La funzione restituisce il **numero di byte ricevuti**, -1 su errore, 0 se il socket remoto si è chiuso (vedi più avanti)
- **La funzione è bloccante**: il programma si ferma finché non ha letto *qualcosa*



# Primitiva **recv()**

```
int ret, sd, bytes_needed;
char buffer[1024];

//...

bytes_needed = 20;
ret = recv(sd, (void*)buffer, bytes_needed, 0);
// Adesso 0 < ret <= bytes_needed
if (ret < bytes_needed) {
    // Gestione errore
}

ret = recv(sd, (void*)buffer, bytes_needed, MSG_WAITALL);
// Adesso ret == bytes_needed
```



# Primitiva **close()**

- **Chiude** un socket
  - Non può più essere usato per inviare o ricevere dati

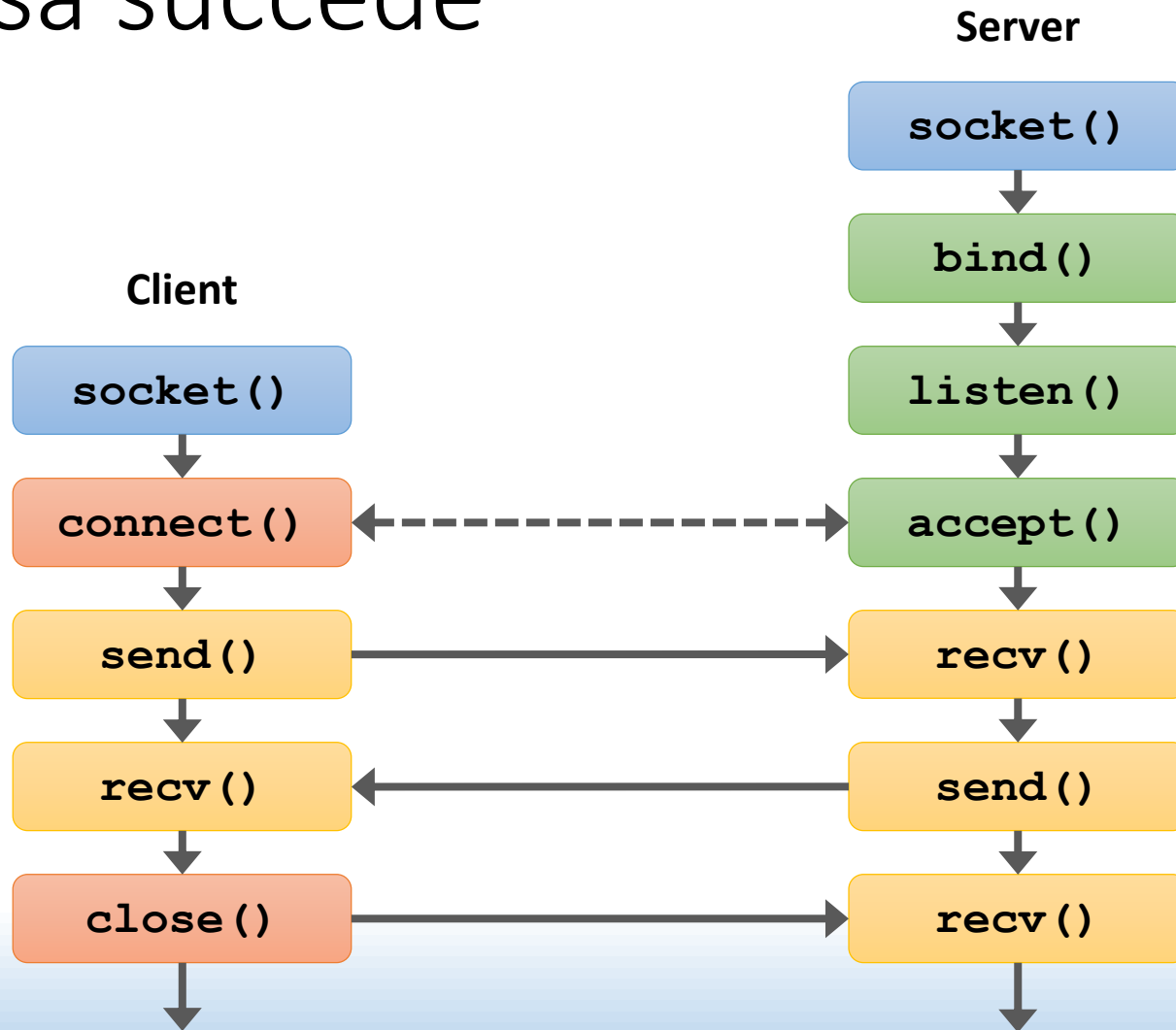
```
#include <unistd.h>
```

```
int close(int fd);
```

**man 2 close**

- **fd**: descrittore del socket
- La funzione restituisce 0 se ha successo, -1 su errore
- L'host remoto riceverà 0 dalla **recv()**

# Cosa succede





# Gestione degli errori



# Gestione degli errori

- Le primitive viste finora restituiscono -1 quando c'è un errore
  - In più settano una variabile, **errno**, che può essere letta per scoprire il motivo dell'errore
  - Nel manuale di ogni funzione c'è l'elenco degli errori possibili

```
#include <errno.h>
//...
ret = bind(sd, (struct sockaddr*)&my_addr, sizeof(my_addr));
if (ret == -1) {
    if (errno == EADDRINUSE) { /* Gestisci errore */}
    if (errno == EINVAL) { /* Gestisci errore */}
    //...
}
```

**man 3 errno**



# Gestione degli errori

- A volte vogliamo solo sapere l'errore e uscire
  - `perror()` legge `errno` e stampa l'errore su schermo in forma leggibile

**man 3 perror**

```
#include <stdio.h>
//...
ret = bind(sd, (struct sockaddr*)&my_addr, sizeof(my_addr));
if (ret == -1) {
    perror("Error: ");
    exit(1);
}
//...
```