Sistemi di Elaborazione

Introduzione alla Programmazione distribuita

Obiettivi

- Introdurre i concetti di base su programmazione distribuita
  - Modello Client-Server
  - Socket Application Programming Interface (API)
- Progettare e realizzare una semplice applicazione distribuita

Cooperazione fra processi

- Processi indipendenti
  - Il risultato dell'esecuzione di un processo non dipende dagli altri processi nel sistema
- Processi cooperanti
  - Sincronizzazione
  - Scambio di messaggi

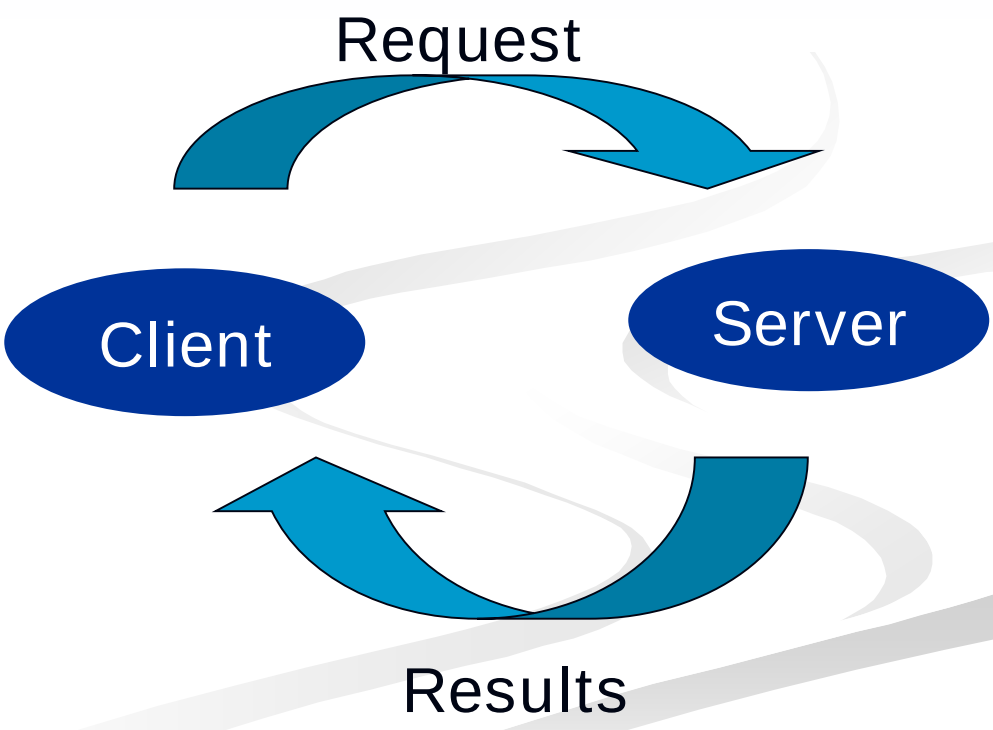
Comunicazione fra processi

- Se i processi sono eseguiti dalla stessa macchina (*host*), ossia dallo stesso kernel. La comunicazione può avvenire tramite
  - Memoria condivisa
  - Scambio di messaggi (IPC)

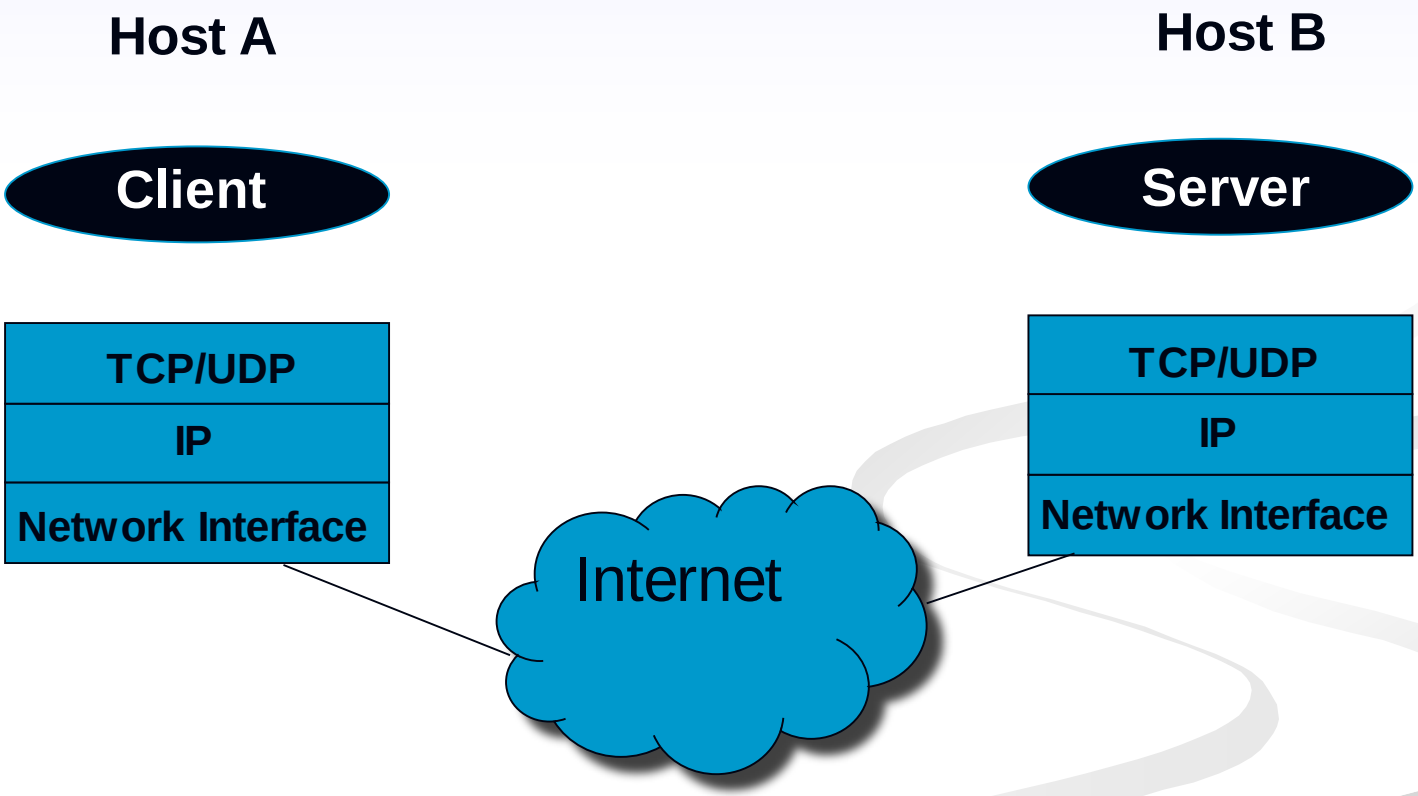
- Se i processi sono eseguiti da macchine diverse il sistema si dice *distribuito*. Possibili paradigmi:
  - Client-Server
  - Remote Procedure Call
  - Remote Method Invocation
  - …

Client-Server

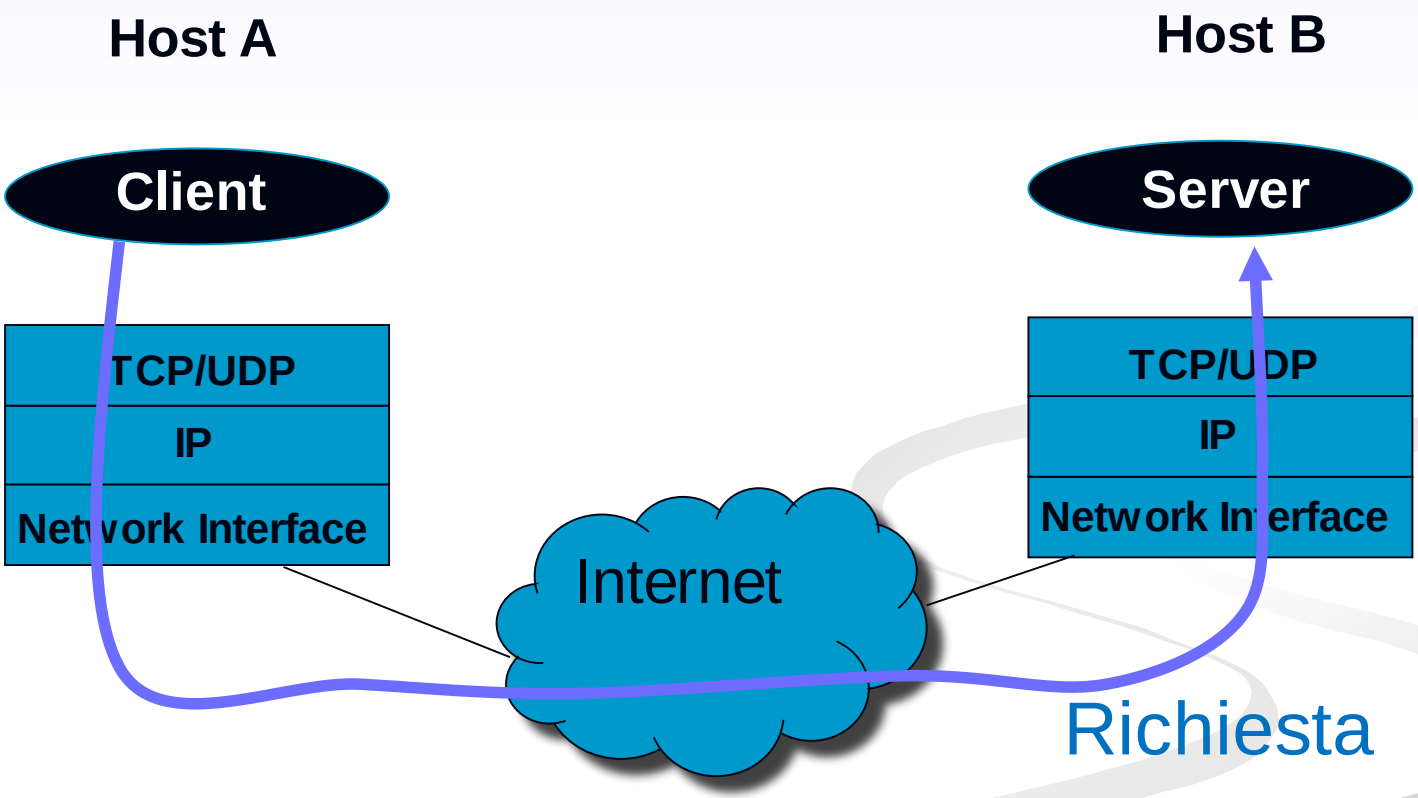
- Paradigma basato su scambio di messaggi
- Scambio di messaggi per
  - Richiesta di servizio
  - Invio dei risultati
- Paradigma generale
  - Usato principalmente in sistemi distribuiti



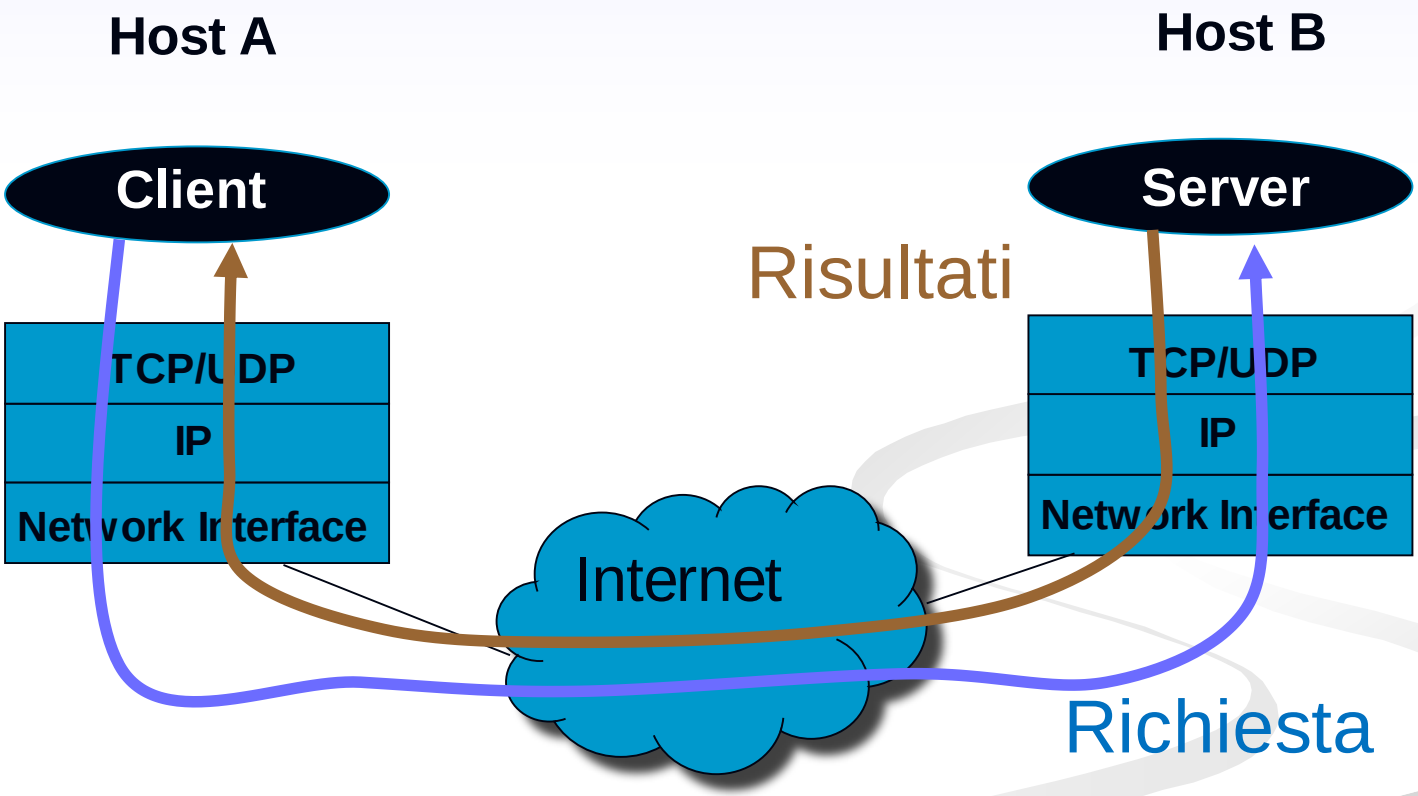
Client-Server in Sistemi Distribuiti



Client-Server in Sistemi Distribuiti



Client-Server in Sistemi Distribuiti



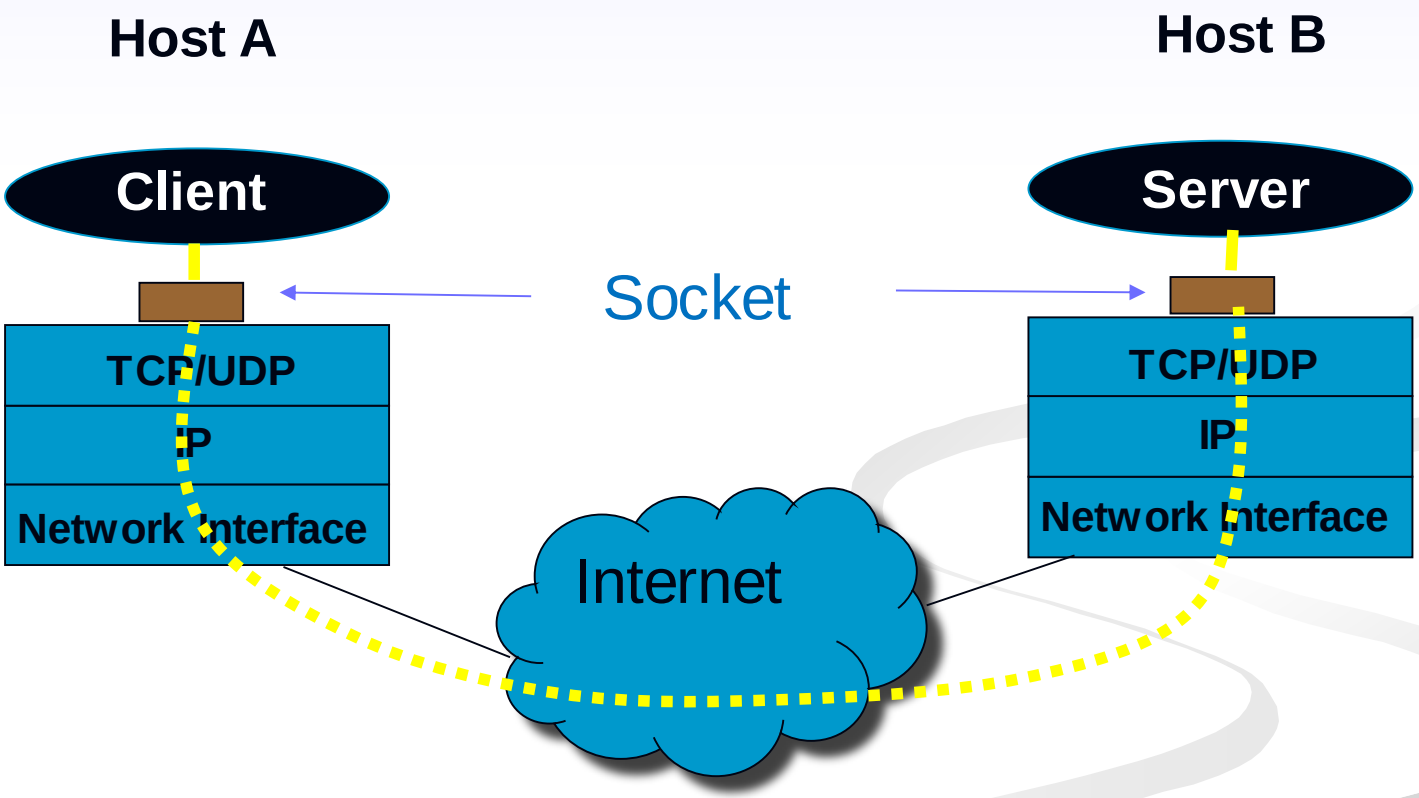
Socket API

- Meccanismo generico di comunicazione tra processi (IPC), che possono o meno eseguire sulla stessa macchina
- I socket forniscono una API unificata per operare con i vari protocolli di rete supportati dal kernel
- L'API nasconde tutti i meccanismi di comunicazione a livello di trasporto (L4), rete (L3) e link (L2).

Socket

- Un socket rappresenta l'estremità di un canale di comunicazione
  - E' identificato da un nome locale (indirizzo)
- Per i socket TCP e UDP, il nome locale ha due componenti:
  - Un Indirizzo IP per identificare l'host
  - Un Numero di porta, per identificare un processo eseguito dall'host
- Il processo all'altro capo del canale utilizza un altro socket

Comunicazione mediante socket



Supporto del SO

- Il SO implementa l'astrazione di socket
- Fornisce system call per
  - Creare un socket
  - Associare un nome locale (indirizzo IP e porta) al socket
  - Mettere in ascolto un processo su un socket (server)
  - Accettare una richiesta di servizio su un socket (server)
  - Aprire una connessione verso un socket remoto, in modo da instaurare il canale (client)
  - Inviare un messaggio tramite il socket
  - Ricevere un messaggio da un socket
  - ....

Creazione di una connessione TCP

Primitiva socket()

- Crea un socket
  - Restituisce un file descriptor (valore intero non negativo)
  - In caso di errore restituisce -1 (settando la variabile *errno*)

int socket(int family, int type, int protocol) [man 2 socket]

- *family*: famiglia di protocolli da utilizzare
  - AF_INET: protocolli internet IPv4 [man 7 ip]
  - AF_UNIX: Unix domain protocol [man 7 unix]
- *type*: contratto di comunicazione che si vuole utilizzare
  - SOCK_STREAM: socket di tipo stream (TCP)
  - SOCK_DGRAM: socket di tipo datagram (UDP)
- protocol: settato a 0 nel caso di AF_INET

```
sk = socket(AF_INET, SOCK_STREAM, 0);
```

Primitiva setsockopt()

- Manipola le opzioni associate con un socket

  int setsockopt(int sd, int level, int optname, const void* optval, socklen_t optlen);

  - level: stabilisce il livello a cui manipolare le opzioni
    - SOL_SOCKET: opzioni di livello socket
  - optname: opzione da settare (man 7 socket per le opzioni di livello socket)
    - SO_REUSEADDR: permette di chiamare bind() su una porta anche se esistono delle connessioni established che usano quella porta. Questa situazione e' molto comune all'atto di rilanciare il server
  - optval e optlen: servono per accedere al valore della opzione

- Restituisce 0 in caso di successo, -1 in caso di errore (setta *errno*)
- Si mette tra la socket() e la bind()
- Es:

```
int optval = 1;
setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof(optval));
```

Strutture Dati per nomi di socket

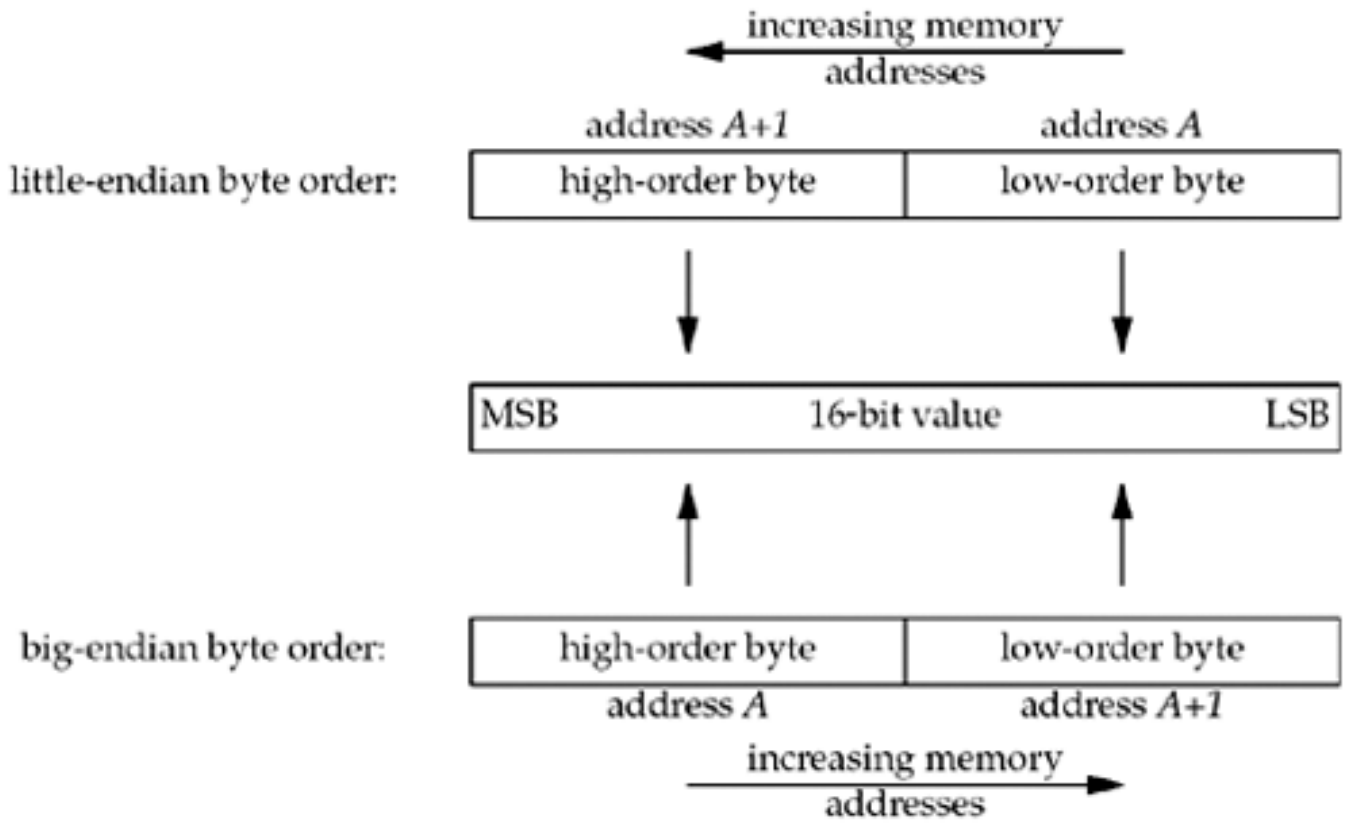
- ```
  struct sockaddr {                    /* man 7 ip */
    sa_family_t sa_family;    /* AF_INET */
    char sa_data[14]          /* address (protocol specific) */
  };
  ```

- ```
  struct sockaddr_in {                    /* man 7 ip */
    sa_family_t sin_family;          /* AF_INET */
    u_int16_t sin_port;              /* porta, 16 bit */
    struct in_addr sin_addr;         /* indirizzo IP 32 bit */
  };
  ```

- ```
  struct in_addr {
    u_int32_t  s_addr;               /* indirizzo IP 32 bit */
  };
  ```
  ```

Endianness

- *Instruction set* diversi possono usare convenzioni diverse per ordinare i byte di una parola di memoria

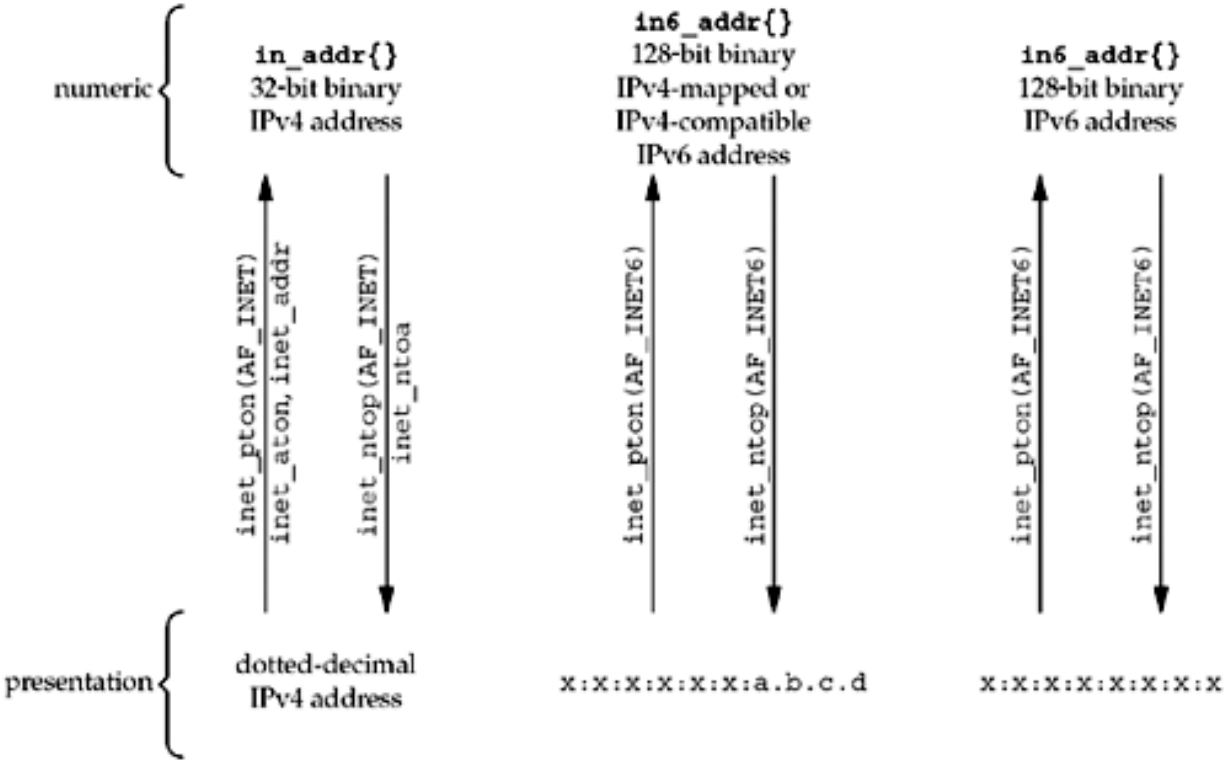


Network order

- L'indirizzo IP ed il numero di porta devono essere specificati nel formato di rete (*network order*, big endian) in modo da essere indipendenti dal formato usato dal calcolatore (*host order*).
- Opportune funzioni di conversione sono disponibili per convertire parole di memoria tra i due formati
  - uint32_t htonl(uint32_t hostlong);
  - uint16_t htons(uint16_t hostshort);
  - uint32_t ntohl(uint32_t netlong);
  - uint16_t ntohs(uint16_t netlong);

Formato di indirizzi IP

- Alcune funzioni consentono di passare dal formato *numeric* al formato *presentation* di un indirizzo IP



Formato di indirizzi IP

- Formato *numeric* : valore binario nella struttura socket
  - int inet_pton(int af, const char* src, void* addr_ptr);
    - Restituisce 0 in caso di insuccesso

- Formato *presentation* : stringa
  - char* inet_ntop(int af, const void* addr_ptr, char* dest, size_t len);
    - len: deve valere almeno INET_ADDRSTRLEN
    - Restituisce un puntatore NULL in caso di errore

Inizializzazione nomi socket IPv4

```
struct sockaddr_in addr_a;

memset(&addr_a, 0, sizeof(addr_a)); /* azzera la struttura*/

addr_a.sin_family = AF_INET;      /* IPv4 address */

addr_a.sin_port = htons(1234);    /* network ordered */

inet_pton(AF_INET, “192.168.1.1”, &addr_a.sin_addr.s_addr);
```

Primitiva bind()

- Assegna un nome *locale* al socket creato con la socket()
- Usata dal server per specificare l'indirizzo su cui il accetta le richieste
  - Indirizzo IP di una interfaccia di rete oppure *any*
  - Numero di Porta
- Il client non esegue la bind()
  - la porta viene assegnata *al volo* dal kernel

Primitiva bind()

int bind(int sd, struct sockaddr* myaddr, int addrlen);

- sd: file descriptor del socket
- myaddr: indirizzo della struttura dati che contiene il nome da associare al socket
  - A seconda della famiglia di protocolli usata dal socket, il formato della struttura dati varia. Occorre eseguire un *cast* del puntatore.
- addrlen: dimensione della struttura myaddr
- Restituisce 0 in caso di successo, -1 in caso di errore (setta la variabile *errno*)

Primitiva bind()

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

man 2 bind per ulteriori dettagli

Primitiva listen()

- Prepara il socket per attendere eventuali connessioni in ingresso.
- Usata dal server per segnalare al kernel la sua disponibilità ad accettare richieste di connessione destinate al nome locale specificato dalla bind() precedente

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

  - sd: file descriptor del socket in oggetto
  - backlog: dimensione massima per la coda di connessioni pendenti (connessioni established in attesa di essere accettate)
  - Restituisce 0 in caso di successo; -1 in caso di errore (setta *errno*)

Primitiva accept()

- Usata dal server per accettare richieste di connessione
- Estrae la prima richiesta di connessione dalla coda delle connessioni pendenti relativa al (listening) socket
- Crea un nuovo socket (connected socket) per gestire la nuova connessione.
- Il listening socket è usato solamente per accettare le richieste
- Il connected socket è usato per la comunicazione vera e propria con il client
  - Scambio di messaggi
- In un server c'è solitamente un solo socket in ascolto
- Le connessioni vengono gestite dai socket creati dalla accept()

Primitiva accept()

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

- sd: file descriptor del listening socket
- addr: argomento di output, punta ad una struttura allocata dal client che viene riempita dal kernel con il nome del socket remoto (indirizzo IP e porta)
- addrlen: argomento di output, punta ad una variabile (allocata dal client) che viene riempita dal kernel con la dimensione della struttura addr

- Restituisce il descrittore del connected socket; -1 in caso di errore (e setta errno)
- Se non ci sono connessioni completate la funzione è di default bloccante

Setup lato server

```
#define SA struct sockaddr
struct sockaddr_in my_addr, cl_addr;
int ret, len, sk, cn_sk;

sk = socket(AF_INET, SOCK_STREAM, 0);
memset(&my_addr, 0, sizeof(my_addr));
my_addr.sin_family = AF_INET;
my_addr.sin_addr.s_addr = htonl(INADDR_ANY);
my_addr.sin_port = htons(1234);

ret = bind(sk, (SA *) &my_addr, sizeof(my_addr));
ret = listen(sk, 10);

len = sizeof(cl_addr);
cn_sk = accept(sk, (SA *) &cl_addr, &len);
```

- Con INADDR_ANY il server si mette in ascolto su una qualsiasi delle sue interfacce di rete

Creazione di una connessione TCP

Primitiva connect()

- Usata dal client per creare una connessione con il server
  - Il canale risultante viene associato al socket specificato come argomento

int connect(int sd, const struct sockaddr* serv_addr, socklen_t addrlen);

  - sd: socket da usare per la connessione (creato con socket())
  - serv_addr: struttura contenente il nome del socket remoto a cui connettersi
    - indirizzo IP ed numero di porta del server da contattare
  - addrlen: dimensione della struttura serv_addr

- Restituisce 0 in caso di connessione; -1 in caso di errore (e setta errno)

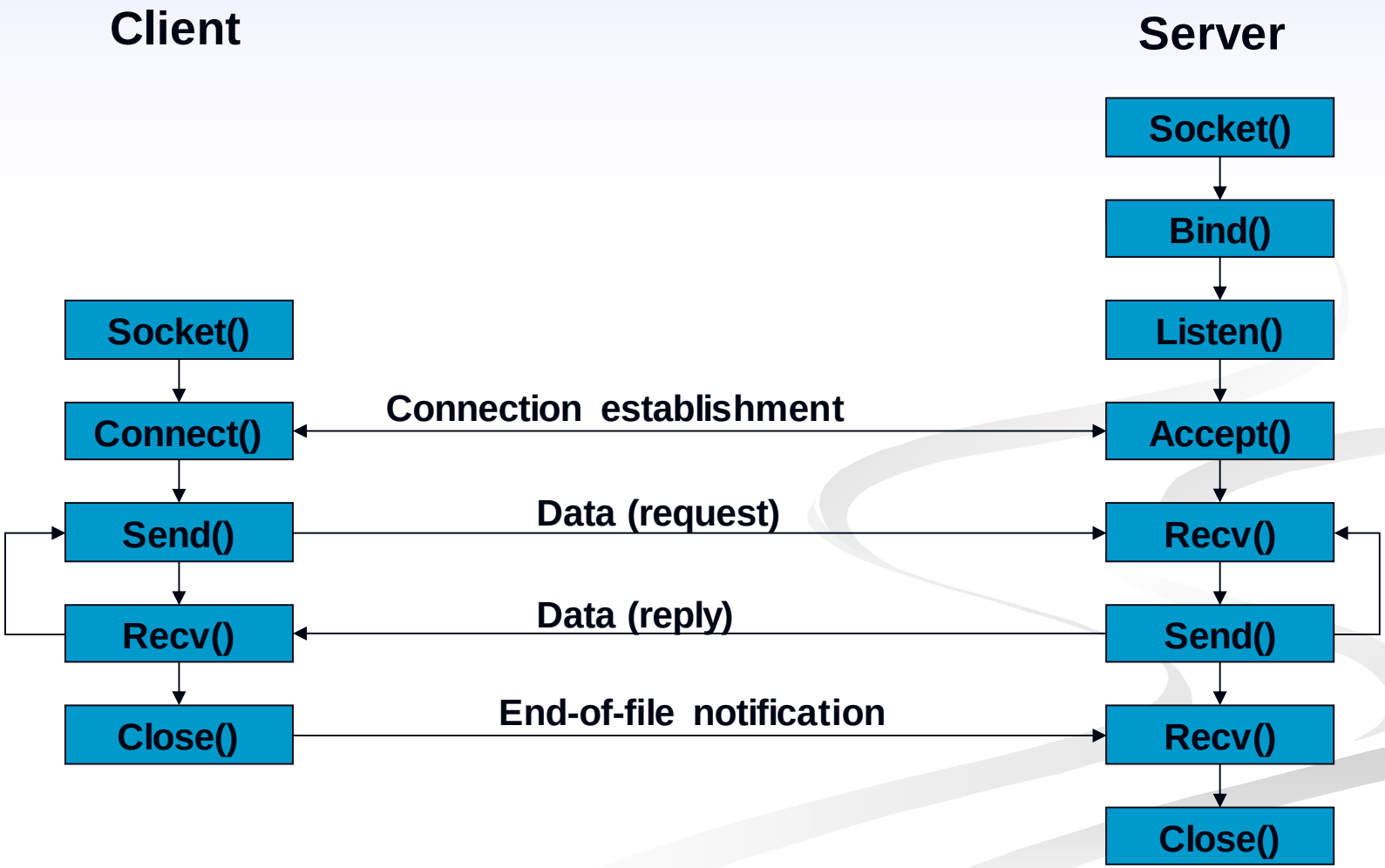
Setup lato client

```
#define SA struct sockaddr
struct sockaddr_in srv_addr;
int ret, sk;

sk = socket(AF_INET, SOCK_STREAM, 0);
memset(&srv_addr, 0, sizeof(srv_addr));
srv_addr.sin_family = AF_INET;
srv_addr.sin_port = htons(1234);
ret = inet_pton(AF_INET, “192.168.1.1”, &srv_addr.sin_addr);

ret = connect(sk, (SA *) &srv_addr, sizeof(srv_addr));
```

Esempio interazioni client-server



Primitiva send()

- Usata per inviare dati attraverso il socket

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

  - sd: descrittore del socket usato per la comunicazione
  - buf: buffer contenente il messaggio da spedire
  - len: lunghezza del messaggio
  - flags: definisce il comportamento della send
- Restituisce il numero di caratteri spediti; -1 in caso di errore

Invio dati

```
int ret, sk_a;
char msg[1024];
...
strncpy(msg, “something to send”, sizeof(msg));
ret = send(sk_a, (void *) msg, strlen(msg), 0);
if(ret == -1 || ret < strlen(msg)){  /* error */
   ...
}
```

Problema: Come fa il ricevitore a conoscere la lunghezza del messaggio inviato dal mittente?

Invio lunghezza e dati

```
int ret, sk_a, dim;
char msg[1024];
...
strncpy(msg, “something to send”, sizeof(msg));
dim = htonl(strlen(msg));
ret = send(sk_a, (void *) &dim, sizeof(dim), 0);
if(ret == -1 || ret < sizeof(dim)){  /* error */
...
}
ret = send(sk_a, (void *) msg, strlen(msg), 0);
if(ret == -1 || ret < strlen(msg)){  /* error */
...
}
```

Primitiva Receive()

- Usata per ricevere dati da un socket

  ssize_t recv(int sd, void* buf, size_t len, int flags);
  - sd: socket dal quale ricevere i dati
  - buf: buffer dove mettere i dati ricevuti
  - len: dimensione del buffer
  - flags: definisce il comportamento della recv
- Restituisce il numero di byte ricevuti; -1 in caso di errore
- È bloccante di default

Ricezione dati

```
int ret, len, sk_a;
char msg[1024];
...
ret = recv(sk_a, (void *) msg, len, MSG_WAITALL);
/* non ritorna finchè non ha letto l'intera lungh. del msg */

ret = recv(sk_a, (void *) msg, len, 0);
/* len is the size of the incoming message (<= sizeof(msg)) */

if( (ret == -1) || (ret<len) ) { /* error */
  ...
}
```

Numero di caratteri che si vogliono leggere

Ricezione lunghezza e dati

```
int ret, sk_a, dim;
char msg[1024];
...

ret = recv(sk_a, (void *) &dim, sizeof(dim), MSG_WAITALL);
if( (ret == -1) || (ret<sizeof(dim)) ) { /* error */
  ...
}
dim = ntohl(dim);

ret = recv(sk_a, (void *) msg, dim, MSG_WAITALL);
if( (ret == -1) || (ret<dim) ) { /* error */
  ...
}
```

Primitiva close()

- Marca il socket come *closed*
- Il socket non può più essere usato per inviare o ricevere dati

int close(int sd)

  - sd è il file descriptor che si vuole chiudere

- Restituisce 0 se tutto è andato bene; -1 altrimenti

Includes

- Header file da includere per usare la socket API
  - #include <unistd.h>
  - #include <sys/types.h>
  - #include <sys/socket.h>
  - #include <arpa/inet.h>
- Per il server multi-threaded utilizzare almeno
  - #include <pthread.h>