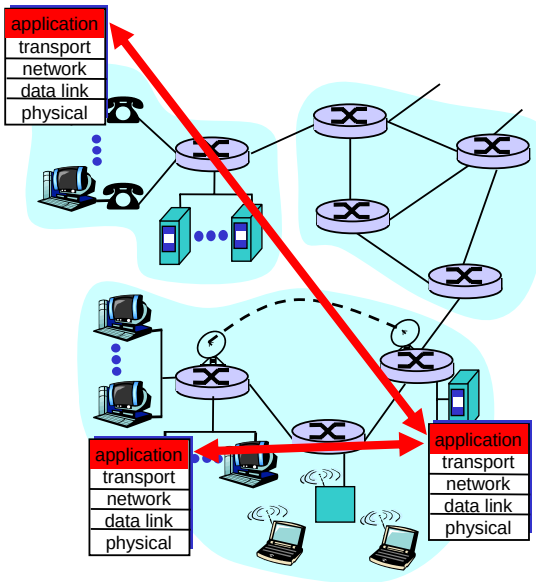
Programmazione di applicazioni di rete

Valeria Cardellini
Università di Roma Tor Vergata

Applicazioni di rete

- Applicazioni di rete
  - forniscono i servizi di alto livello utilizzati dagli utenti
  - determinano la percezione di qualità del servizio (QoS) che gli utenti hanno della rete sottostante

• Applicazioni: **processi comunicanti, distribuiti**
  - ✓ in esecuzione sugli *host* (sistemi terminali) della rete, tipicamente nello "spazio utente"
  - ✓ la comunicazione avviene utilizzando i servizi offerti dal sottosistema di comunicazione
  - ✓ comunicazione a scambio di messaggi
  - ✓ la cooperazione può essere implementata secondo vari modelli (più diffuso: **client-server**)



Modello client/server

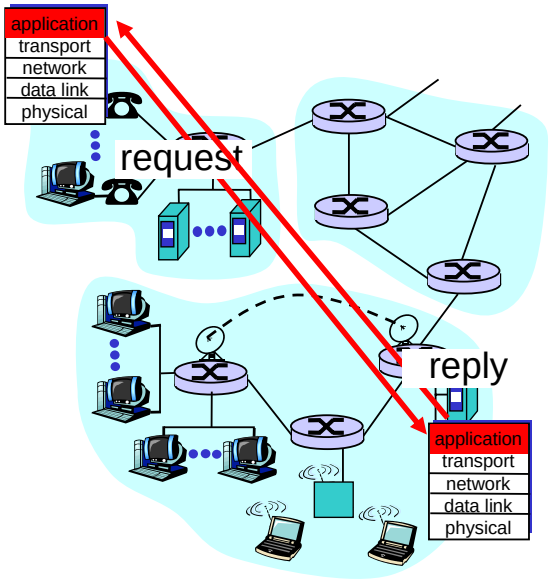
Applicazioni di rete composte da due parti: *client* e *server*

Client:

- È l'applicazione che richiede il servizio (inizia il contatto con il server)
- Es.: richiede una risorsa Web, invia una e-mail

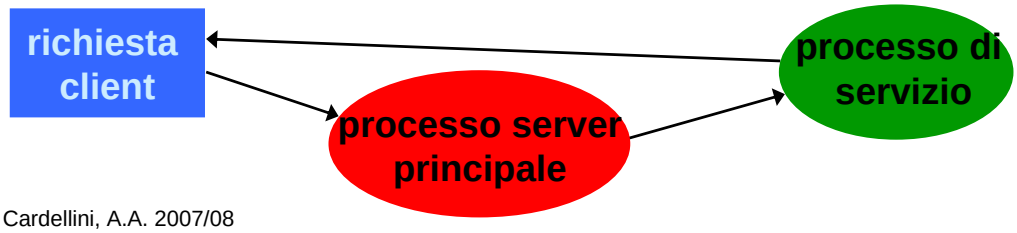
Server:

- È l'applicazione che fornisce il servizio richiesto
- Attende di essere contattato dal client
- Es.: invia la risorsa Web richiesta, riceve/memorizza l'e-mail ricevuta



Modello client/server (2)

- Come fa un'applicazione ad identificare in rete l'altra applicazione con la quale vuole comunicare?
  - **indirizzo IP** dell'host su cui è in esecuzione l'altra applicazione
  - **numero di porta** (consente all'host ricevente di determinare a quale applicazione locale deve essere consegnato il messaggio)
- Server e concorrenza
  - Server **iterativo** (o sequenziale): gestisce una richiesta client per volta
  - Server **ricorsivo** (o concorrente): in grado di gestire più richieste client contemporaneamente
    - In un server ricorsivo, viene creato un nuovo processo/thread di servizio per gestire ciascuna richiesta client



Modello peer-to-peer

- Non vi è un server sempre attivo, ma coppie arbitrarie di host, chiamati **peer** (ossia pari) che comunicano direttamente tra di loro
- Nessuno degli host che prende parte all'architettura peer-to-peer (P2P) deve necessariamente essere sempre attivo
- Ciascun peer può ricevere ed inviare richieste e può ricevere ed inviare risposte
- Punto di forza: scalabilità delle applicazioni P2P
- Svantaggio: le applicazioni P2P possono essere difficili da gestire a causa della loro natura altamente distribuita e decentralizzata

- Alcune applicazioni di rete sono organizzate come un *ibrido* delle architetture client-server e P2P

Un problema apparente…

- Una porta viene assegnata ad un servizio, ma nel caso del multitasking/multithreading vi potrebbero essere più processi/thread server attivi per lo stesso servizio
- D'altro canto, le richieste di un client devono essere consegnate al processo/thread server corretto

- Soluzione:

  Usare sia le informazioni del server, sia le informazioni del client per indirizzare i pacchetti
- I protocolli di trasporto TCP e UDP usano 4 informazioni per identificare una connessione:
  - Indirizzo IP del server
  - Numero di porta del servizio lato server
  - Indirizzo IP del client
  - Numero di porta del servizio lato client

Interazione protocollo di trasporto con applicazioni

- Il client ed il server utilizzano un protocollo di trasporto (TCP o UDP) per comunicare
- Il software di gestione del protocollo di trasporto si trova all'interno del sistema operativo
- Il software dell'applicazione si trova all'esterno del sistema operativo
- Per poter comunicare due applicazioni devono interagire con i rispettivi sistemi operativi: come?

Si utilizza un meccanismo che svolge il ruolo di ponte (interfaccia) tra sistema operativo ed applicazione di rete:
Application Programming Interface (API)

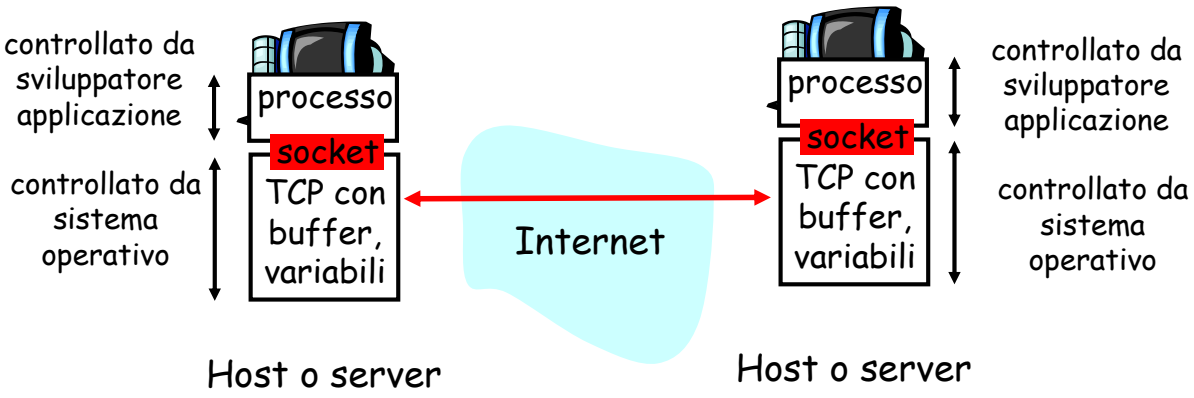
Application Programming Interface

- Standardizza l'interazione con il sistema operativo (SO)
- Consente ai processi applicativi di utilizzare i protocolli di rete, definendo:
  - le funzioni consentite
  - gli argomenti per ciascuna funzione

- **Socket**: Internet API
  - Definito inizialmente per sistemi Unix BSD per utilizzare i protocolli TCP/IP
    - Utilizza un'estensione del paradigma di I/O di questi sistemi
  - Divenuto uno standard di riferimento per tutta la programmazione su reti, disponibile su vari sistemi operativi (ad es. WinSock)
  - Due tipi di servizio di trasporto
    - Datagram non affidabile (UDP)
    - Flusso di byte orientato alla connessione ed affidabile (TCP)

Socket

- E' un'interfaccia (o "porta") tra il processo applicativo e il protocollo di trasporto end-to-end (UCP o TCP)
- E' un'interfaccia **locale** all'host, controllata dal sistema operativo, creata/posseduta dall'applicazione tramite la quale il processo applicativo può inviare/ricevere messaggi a/da un altro processo applicativo (remoto o locale)



Socket (2)

- I socket sono delle API che consentono ai programmatori di gestire le comunicazioni tra processi
  - E' un meccanismo di Interprocess Communication (IPC)
- A differenza degli altri meccanismi di IPC (pipe, code di messaggi, FIFO e memoria condivisa), i socket consentono la comunicazione tra processi che possono risiedere su macchine **diverse** e sono collegati tramite una rete
  - Costituiscono lo strumento di base per realizzare servizi di rete
- Astrazione dei processi sottostanti
  - In pratica, i socket consentono ad un programmatore di effettuare trasmissioni TCP e UDP senza curarsi dei dettagli "di più basso livello" che sono uguali per ogni comunicazione (*three-way handshaking*, *finestre*, …)

Socket (3)

- Socket
  - astrazione del SO
  - creato dinamicamente dal SO su richiesta
  - persiste soltanto durante l'esecuzione dell'applicazione
  - il suo ciclo di vita è simile a quello di un file (apertura, collegamento ad un endpoint, lettura/scrittura, chiusura)
- Descrittore del socket
  - un intero
  - uno per ogni socket attivo
  - significativo soltanto per l'applicazione che possiede il socket
- Endpoint del socket
  - ogni associazione socket è una quintupla di valori **{protocollo, indirizzo locale, porta locale, indirizzo remoto, porta remota}**
  - l'associazione deve essere completamente specificata affinché la comunicazione abbia luogo

Dichiarazione al sistema operativo

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

- E' la prima funzione eseguita sia dal client sia dal server
- Definisce un socket: crea le strutture e riserva le risorse necessarie per la gestione di connessioni
- Restituisce un intero che va interpretato come un descrittore di file: un identificatore dell'entità appena creata
- Il client utilizzerà il socket direttamente, specificando il descrittore in tutte le funzioni che chiamerà
- Il server utilizzerà il socket indirettamente, come se fosse un modello o un prototipo per creare altri socket che saranno quelli effettivamente usati
- Se la creazione del socket fallisce, viene restituito il valore -1

Parametri della funzione socket()

int domain — Dominio del socket (famiglia di protocolli); alcuni valori:

- valore usuale: AF_INET (IPv4), ossia InterNET Protocol Family (comunicazione tra processi in Internet);
- AF_INET6 per IPv6
- AF_UNIX per comunicazioni tra processi sullo stesso host Unix

int type — Tipo di comunicazione

- SOCK_STREAM: orientato alla connessione (flusso continuo)
- SOCK_DGRAM: senza connessione (pacchetti)
- SOCK_RAW: per applicazioni dirette su IP (riservato all'uso di sistema)

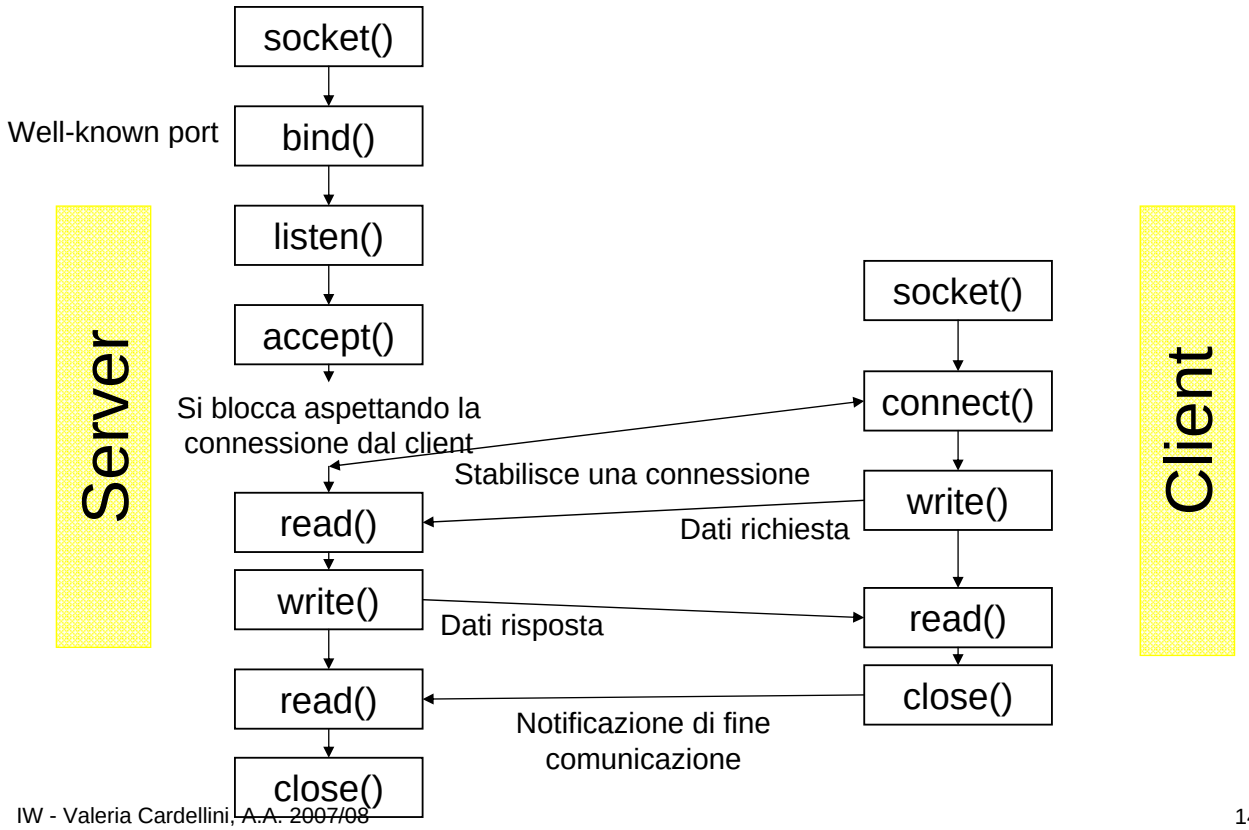
int protocol — Lo specifico protocollo richiesto: solitamente, si pone a 0 per selezionare il protocollo di default indotto dalla coppia domain e type (tranne che per SOCK_RAW):

- AF_INET + SOCK_STREAM determinano una trasmissione TCP (protocol = IPPROTO_TCP)
- AF_INET + SOCK_DGRAM determinano una trasmissione UDP (protocol = IPPROTO_UDP)

Gestione degli errori

- In caso di successo le funzioni dell'API socket
  - restituiscono un valore positivo
- In caso di fallimento le funzioni dell'API socket
  - restituiscono un valore negativo (-1)
  - assegnano alla variabile globale errno un valore positivo
    - Ogni valore identifica un tipo di errore ed il significato dei nomi simbolici che identificano i vari errori è specificato in sys/errno.h
    - Le funzioni strerror() e perror() permettono di riportare in opportuni messaggi la condizione di errore verificatasi
- Il valore contenuto in errno si riferisce all'ultima chiamata di sistema effettuata
- Dopo aver invocato una funzione dell'API si deve
  - verificare se il codice di ritorno è negativo (errore)
  - in caso di errore, leggere immediatamente il valore di errno

Comunicazione orientata alla connessione



Per inizializzare indirizzo locale e processo locale (*server*)

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

- Serve a far sapere al SO a quale processo vanno inviati i dati ricevuti dalla rete
  - sockfd: descrittore del socket
  - addr: puntatore ad una struct contenente l'indirizzo locale
  - len: dimensione *in byte* della struct contenente l'indirizzo locale

Struttura degli indirizzi

- Struttura generica

```
struct sockaddr {
    sa_family_t sa_family;
    char sa_data[14]; }
```

- Struttura degli indirizzi IPv4

```
struct sockaddr_in {
    sa_family_t syn_family;  /* AF_INET */
    in_port_t syn_port;      /* numero di porta (2 byte) */
    struct in_addr sin_addr; /* struttura indirizzo IP (4 byte) */
    char sin_zero[8]; }      /* non usato */
```

Equivale a sa_data[14] in struct sockaddr

- Struttura dati in_addr

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


Inizializzare indirizzo IP e numero di porta

- L'indirizzo IP ed il numero di porta in sockaddr_in devono essere nel **network byte order**
  - Prima il byte più significativo, ovvero ordinamento *big endian*
- Funzioni di conversione da rappresentazione testuale/binaria dell'indirizzo/numero di porta a valore binario da inserire nella struttura sockaddr_in
- Per inizializzare i campi di sockaddr_in

```
unsigned short htons(int hostshort);    /* numero di porta */
unsigned long htonl(int hostlong);      /* indirizzo IP */
```

  - htons: host-to-network byte order short (16 bit)
  - htonl: host-to-network byte order long (32 bit)
- Per il riordinamento (da rete a macchina locale):
  - ntohs(), ntohl()

Inizializzare indirizzo IP e numero di porta (2)

- Per convertire gli indirizzi IP dalla notazione puntata (*dotted decimal*) in formato ASCII al network byte order in formato binario

```
int inet_aton(const char *src, struct in_addr *dest);
```

  - Es: inet_aton("160.80.85.38", &(sad.sin_addr));
- Per la conversione inversa

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

- Esistono anche le funzioni inet_pton() e inet_ntop()

```
int inet_pton(int af, const char *src, void *addr_ptr);
char *inet_ntop(int af, const void *addr_ptr, char *dest, size_t len);
```

  - A differenza di inet_aton() e inet_ntoa(), possono convertire anche gli indirizzi IPv6

Indirizzo IP in bind()

- Si può assegnare un indirizzo IP specifico
  - L'indirizzo deve appartenere ad un'interfaccia di rete della macchina
- Per indicare un indirizzo IP generico (0.0.0.0)
  - Si usa la costante INADDR_ANY
- Per indicare l'interfaccia di loopback (127.0.0.1)
  - Si usa la costante INADDR_LOOPBACK

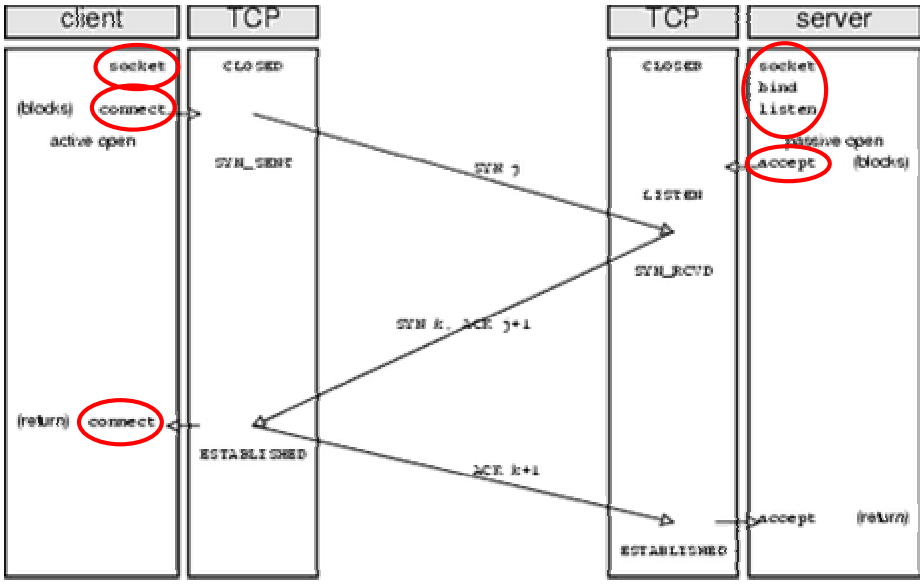
Header file

- sys/socket.h definisce i simboli che iniziano con PF_ e AF_ ed i prototipi delle funzioni (ad es., bind())
- netinet/in.h definisce i tipi di dato per rappresentare gli indirizzi Internet (ad es., la definizione di struct sockaddr_in)
- arpa/inet.h definisce i prototipi delle funzioni per manipolare gli indirizzi IP (ad es., inet_aton())
- In più, sys/types.h, netdb.h e unistd.h (per close())

Esempio funzione bind()

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
…
  struct sockaddr_in sad;
  int sd;
…
  if ((sd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    perror(“errore in socket");
    exit(-1);
  }
  memset((void *)&sad, 0, sizeof(sad));
  sad.sin_family = AF_INET;
  sad.sin_port = htons(1234);
  sad.sin_addr.s_addr = htonl(INADDR_ANY); /* il server accetta
     richieste su ogni interfaccia di rete */
  if (bind(sd, (struct sockaddr *)&sad, sizeof(sad)) < 0) {
…
```

Three way handshake del TCP



Funzione connect() (*client*)

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

- Permette al client TCP di aprire la connessione con un server TCP
  - Bloccante: termina solo quando la connessione è stata creata
  - Restituisce 0 in caso di successo, -1 in caso di errore
- Parametri della funzione
  - sockfd: descrittore del socket
  - servaddr: indirizzo (indirizzo IP + numero di porta) del server cui ci si vuole connettere
  - addrlen: dimensione in byte della struttura con l'indirizzo remoto del server
- In caso di errore, la variabile errno può assumere i valori:
  - ETIMEDOUT: è scaduto il timeout del SYN
  - ECONNREFUSED: nessuno in ascolto (RST in risposta a SYN)
  - ENETUNREACH: errore di indirizzamento (messaggio ICMP di destinazione non raggiungibile)

Gestione di errori transitori in connect()

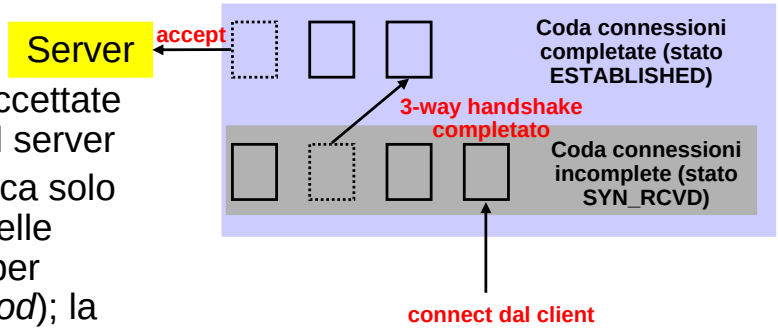
- Algoritmo di backoff esponenziale
  - Se la chiamata di connect fallisce, il processo attende per un breve periodo e poi tenta di nuovo la connessione, incrementando progressivamente il ritardo, fino ad un massimo di circa 2 minuti

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

# define MAXSLEEP 128

int connect_retry(int sockfd, const struct sockaddr *addr, socklen_t alen)
{
    int nsec;

    for (nsec=1; nsec <= MAXSLEEP; nsec <<=1) {
        if (connect (sockfd, addr, alen) == 0)
            return(0);          /*connessione accettata */
    /* Ritardo prima di un nuovo tentativo di connessione */
    if (nsec <= MAXSLEEP/2)
        sleep(nsec);
    }
    return(-1);
}
```


Funzione listen() (*server*)

int listen(int sockfd, int backlog);

- Mette il socket in ascolto di eventuali connessioni (dallo stato CLOSED allo stato LISTEN)
- Specifica quante connessioni possono essere accettate dal server e messe in attesa di essere servite nella coda di richieste di connessione (*backlog*)
  - Le connessioni possono essere accettate o rifiutate dal SO senza interrogare il server
- Storicamente, nel backlog c'erano sia le richieste di connessione in corso di accettazione, sia quelle accettate ma non ancora passate al server
- In Linux, il backlog identifica solo la lunghezza della coda delle connessioni completate (per prevenire l'attacco *syn flood*); la lunghezza massima è pari a SOMAXCONN



Funzione accept() (*server*)

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

- Permette ad un server di prendere dal backlog la prima connessione completata sul socket specificato
  - Se il backlog è vuoto, il server rimane bloccato sulla chiamata finché non viene accettata una connessione
- Restituisce
  - -1 in caso di errore
  - Un nuovo descrittore di socket creato e assegnato automaticamente dal SO e l'indirizzo del client connesso
- Parametri della funzione
  - sockfd: descrittore del socket originale
  - addr: viene riempito con l'indirizzo del client (indirizzo IP + porta)
  - addrlen: viene riempito con la dimensione dell'indirizzo del client; prima della chiamata inizializzato con la lunghezza della struttura il cui indirizzo è passato in addr

Socket d'ascolto e socket connesso

- Il server utilizza due socket diversi per ogni connessione con un client
- Il **socket d'ascolto** (listening socket) è quello creato dalla funzione socket
  - Utilizzato per tutta la vita del processo server
  - In genere usato solo per accettare richieste di connessione
  - Resta per tutto il tempo nello stato LISTEN
- Il **socket connesso** (connected socket) è quello creato dalla funzione accept
  - Usato solo per la connessione con un dato client
  - Usato per lo scambio di dati con il client
  - Si trova automaticamente nello stato ESTABLISHED
- I due socket identificano due connessioni distinte

Associazione orientata alla connessione

- In un flusso di dati orientato alla connessione (type=SOCK_STREAM), l'impostazione delle varie componenti dell'associazione è effettuata dalle seguenti chiamate di sistema:

Endpoint	Protocollo	Indirizzo locale Processo locale	Indirizzo remoto Processo remoto
Server	socket()	bind()	listen()+accept()
Client	socket()	connect()	

- un socket sulla macchina del server

  {tcp, server-address, porta, *, *}

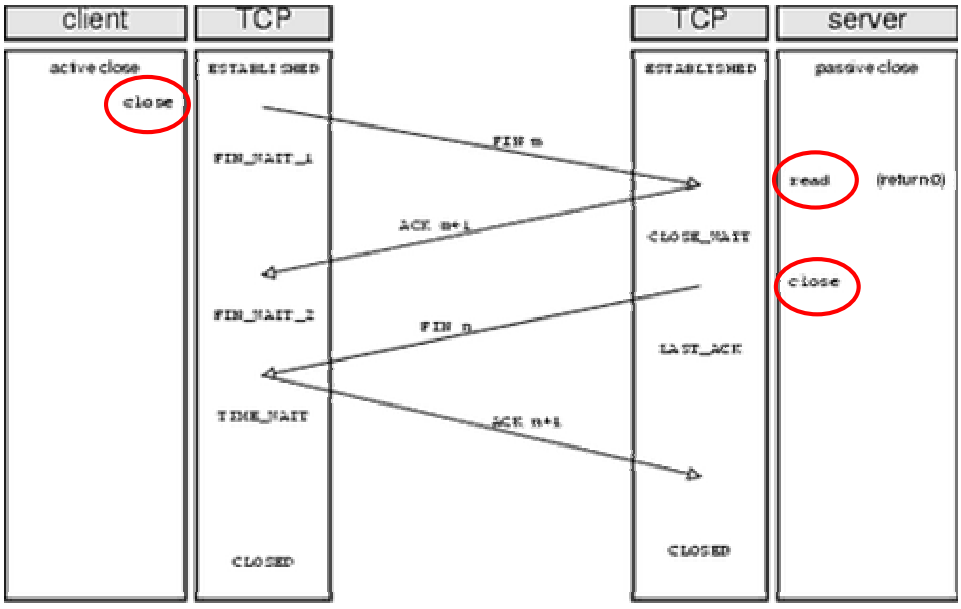
- per ogni client, un socket sulla macchina del server

  {tcp, server-address, porta, client-address, nuovaporta}

- per ogni client, un socket sulla macchina del client

  {tcp, client-address, nuovaporta, server-address, porta}

Four way handshake del TCP



La chiusura di una connessione TCP

int close(int sockfd);

- Il processo che invoca per primo close() avvia la chiusura attiva della connessione
- Il descrittore del socket è marcato come chiuso
  - Il processo non può più utilizzare il descrittore ma la connessione non viene chiusa subito, perché il TCP continua ad utilizzare il socket trasmettendo i dati che sono eventualmente nel buffer
- Restituisce 0 in caso di successo, -1 in caso di errore
- All'altro capo della connessione, dopo che è stato ricevuto ogni dato eventualmente rimasto in coda, la ricezione del FIN viene segnalata al processo come un EOF in lettura
  - Rilevato l'EOF, il secondo processo invoca close() sul proprio socket, causando l'emissione del secondo FIN

La chiusura di una connessione TCP (2)

- Per chiudere una connessione TCP, si può usare anche la funzione shutdown()

int shutdown(int sockfd, int how);

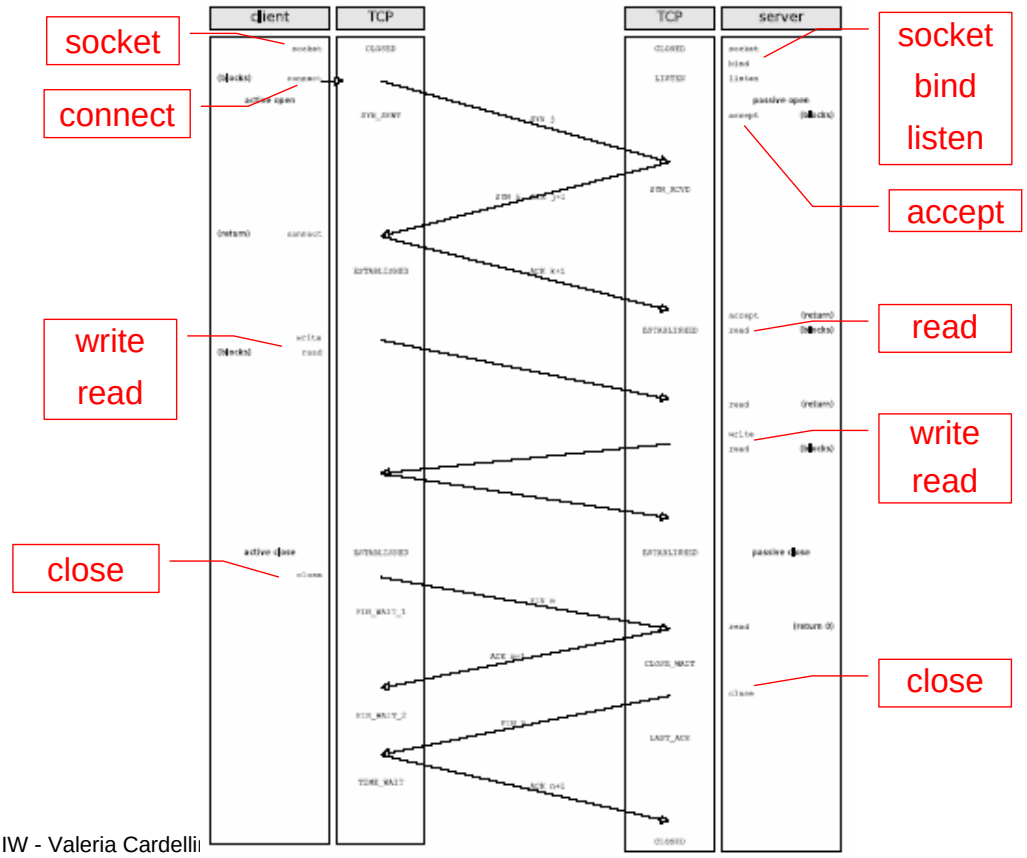
  - shutdown() permette la chiusura asimmetrica
  - Il parametro how può essere uguale a:

    SHUT_RD (0): ulteriori receive sul socket sono disabilitate (viene chiuso il lato in lettura del socket)

    SHUT_WR (1): ulteriori send sul socket sono disabilitate (viene chiuso il lato in scrittura del socket)

    SHUT_RDWR (2): ulteriori send e receive sulla socket sono disabilitate
    - Non è uguale a close(): la sequenza di chiusura del TCP viene effettuata immediatamente, indipendentemente dalla presenza di altri riferimenti al socket (ad es. descrittore ereditato dai processi figli oppure duplicato tramite dup())
    - Con close(), la sequenza di chiusura del TCP viene innescata solo quando il numero di riferimenti del socket si annulla

Scambio di pacchetti in una connessione



Per leggere/scrivere dati

- Per leggere e scrivere su un socket si usano le funzioni read() e write()

```
int read(int sockfd, void *buf, size_t count);
int write(int sockfd, const void *buf, size_t count);
```

- Attenzione: non si può assumere che write() o read() terminino avendo sempre letto o scritto il numero di caratteri richiesto
  - read() o write() restituiscono il numero di caratteri *effettivamente* letti o scritti, oppure -1 in caso di errore
- Come determinare la condizione EOF: read() restituisce 0 (la parte remota dell'associazione è stata chiusa)
- Se il processo prova a scrivere su un socket la cui parte remota è stata chiusa, il SO invia il segnale SIGPIPE e la chiamata di write() restituisce -1 e definisce l'errore EPIPE

Per leggere/scrivere dati (2)

- In alternativa a write() e read(), si possono usare

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

  - Se flags=0 allora send() e recv() equivalgono a write() e read()

Funzione readn

```
#include <errno.h>
#include <unistd.h>
int readn(int fd, void *buf, size_t n)
{
  size_t nleft;
  ssize_t nread;
  char *ptr;

  ptr = buf;
  nleft = n;
  while (nleft > 0) {
    if ( (nread = read(fd, ptr, nleft)) < 0) {
      if (errno == EINTR)     /* funzione interrotta da un segnale prima di aver
                                 potuto leggere qualsiasi dato. */
        nread = 0;
      else
        return(-1);           /*errore */
    }
```

Legge *n* byte da un socket, eseguendo un ciclo di letture fino a leggere *n* byte o incontrare EOF o riscontrare un errore

Restituisce 0 in caso di successo, -1 in caso di errore, il numero di byte non letti in caso di EOF

Funzione readn (2)

```
    else if (nread == 0) break;            /* EOF: si interrompe il ciclo */
    nleft -= nread;
    ptr += nread;
  }  /* end while */
  return(nleft);       /* return >= 0 */
}
```


Funzione writen

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

ssize_t writen(int fd, const void *buf, size_t n)
{
  size_t nleft;
  ssize_t nwritten;
  const char *ptr;

  ptr = buf;
  nleft = n;
  while (nleft > 0) {
    if ( (nwritten = write(fd, ptr, nleft)) <= 0) {
       if ((nwritten < 0) && (errno == EINTR)) nwritten = 0;
       else return(-1);            /* errore */
     }
     nleft -= nwritten;
     ptr += nwritten;
  } /* end while */
  return(nleft);
}
```

Scrive *n* byte in un socket, eseguendo un ciclo di scritture fino a scrivere *n* byte o riscontrare un errore

Restituisce 0 in caso di successo, -1 in caso di errore