Stefano Russo
Dipartimento di Informatica e Sistemistica
Università di Napoli "Federico II"

Il modello a scambio di messaggi in ambiente UNIX distribuito: l’interfaccia socket BSD

La famiglia di protocolli TCP/IP

- TCP/IP è una famiglia di protocolli di rete (*protocol suite*)

- Essa fu sviluppata inizialmente per interconnettere i sistemi eterogenei degli enti di ricerca statunitensi

- Tale interconnessione richiedeva dei nuovi protocolli non proprietari (*vendor independent*)

- La rete che ne nacque fu ARPAnet, divenuta poi **INTERNET**

- I principali protocolli "standard" della famiglia sono:

- **FTP** File Trasfer Protocol
  E' il protocollo a livello applicativo (livello 7 nel modello OSI) che permette il trasferimento di file tra sistemi eterogenei, curando le conversioni di formato

- **TELNET**
  E' il protocollo a livello applicativo per il servizio di terminale virtuale tra sistemi eterogenei, che permette ad un utente di un sistema di collegarsi interattivamente ad un sistema remoto in rete

- **SMTP** Simple Mail Transfer Protocol
  E' il protocollo a livello applicativo per il servizio di posta elettronica; trasferisce messaggi di tipo testo
  Adotta uno schema di tipo client/server, con uno **User Agent** (UA) che interagisce con l'utente e prepara il messaggio per il **Transfer Agent** (TA), che lo inoltra al TA del destinatario

- **BIND**
  E' il protocollo per il servizio di risoluzione dei nomi simbolici delle stazioni nei loro corrispondenti indirizzi numerici di rete

La famiglia di protocolli TCP/IP (segue)

- **TCP** Transmission Control Protocol
  E' il protocollo a livello di trasporto (livello 4 nel modello OSI) orientato alle connessioni, corrispondente al servizio di circuito virtuale
  Garantisce la consegna ordinata ed affidabile dei messaggi

- **UDP** User Datagram Protocol
  E' il protocollo di trasporto *connectionless* , corrispondente al servizio datagramma
  Non garantisce la consegna ordinata dei messaggi, demandata al livello superiore

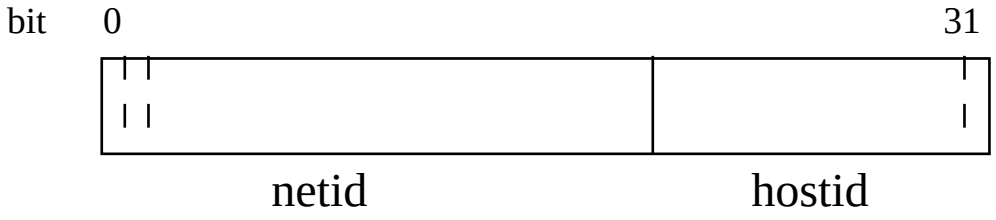
- **IP** Internet Protocol
  E' il protocollo a livello di rete (livello 3 nel modello OSI) per l'instradamento dei messaggi su reti geografiche (*routing*)

- **ICMP** Internet Control Message Protocol
  E' il protocollo su cui sono basati alcuni servizi per il monitoraggio dello stato della rete, il controllo di raggiungibilità dei nodi, ecc.

- **ARP** Address Resolution Protocol
  E' il protocollo per la determinazione dell'indirizzo fisico di un host a partire dal suo indirizzo IP
  La funzione inversa è svolta dal protocollo RARP (Reverse ARP), utilizzato ad esempio dalle stazioni *diskless*
  Nel caso di LAN Ethernet, l'indirizzo fisico è l'indirizzo Ethernet (di 48 bit)

Indirizzi di rete IP

- Nel protocollo IP, l'indirizzo di un nodo di rete (*host*) è costituito da 32 bit, logicamente divisi in due parti
- La prima parte identifica la rete di cui l'host fa parte (*netid*)
- La seconda parte identifica il nodo all'interno della rete (*hostid*)
- L'instradamento dei messaggi da una rete ad un host di un'altra rete avviene sulla base della parte netid
- L'instradamento finale verso l'host viene effettuato all'interno della rete di destinazione



- Nella notazione detta "**dot notation**", i 32 bit di un indirizzo vengono rappresentati come 4 byte, separati da un punto, e rappresentati da interi decimali tra 0 e 255
  Es.: 192.55.101.131
- Per convenzione, l'indirizzo di tutta una rete ha la parte hostid con tutti bit pari a 0
  Es.: 140.164.0.0 rete CNR
- Esistono tre classi di indirizzi, diverse a seconda del numero di byte riservati riservati a netid e hostid

- I *range* di indirizzi sono:

  Classe A: da 1.0.0.0 a 126.0.0.0
  Classe B: da 128.1.0.0 a 191.254.0.0
  Classe C: da 192.1.0.0 a 254.254.0.0

Il sistema dei nomi a domini

- Le applicazioni e gli utenti referenziano di solito le risorse di rete non tramite i loro **indirizzi numerici IP**, bensì tramite **nomi simbolici**
  Es.:
  192.55.101.131 nadis.dis.unina.it

- Ciascuna rete e ciascuna stazione (*host*) devono avere nomi ad essi associati in maniera univoca

- Sulla rete pubblica Internet, lo spazio dei nomi dei nodi è organizzato gerarchicamente, a zone e domini

- La rete Internet è concettualmente divisa in **zone**. Una zona è una ripartizione amministrativa che comprende uno o più **domini**. Ciascuna zona è responsabile dello spazio dei nomi delle stazioni in essa presenti

- Lo **spazio dei nomi a domini** è rappresentabile con un albero, i cui nodi corrispondono ai domini e sottodomini, e le foglie corrispondono agli host veri e propri

- Per ciascuna zona esiste una sola stazione, detta **nameserver primario**, che svolge le funzioni di traduzione dallo spazio dei nomi simbolici a quello degli indirizzi numerici IP

- Il protocollo che implementa le funzionalità di nameserving è **BIND** (Berkeley Internet Name Domain); esso corrisponde al *daemon* named in ambiente UNIX

L’interfaccia socket di Berkeley

- Nei primi anni 80 l'Agenzia statunitense per i Progetti di Ricerca Avanzati (ARPA) finanziò l'Università della California a Berkeley per trasportare TCP/IP sul sistema operativo UNIX

- Come parte del progetto, fu definita una interfaccia per la comunicazione su rete tra le applicazioni, nota come **Berkeley socket interface**

- L'interfaccia introduceva in UNIX poche nuove primitive di sistema (*system calls*), adoperando ove possibile le primitive già esistenti di UNIX

- Ne scaturì la versione di UNIX nota come Berkeley UNIX o BSD UNIX; TCP apparve a partire dalla versione 4.1

- L'interfaccia progettata fornisce funzioni generalizzate per le comunicazioni su rete

- Potenzialmente, essa supporta più famiglie di protocolli; TCP/IP è solo una di esse

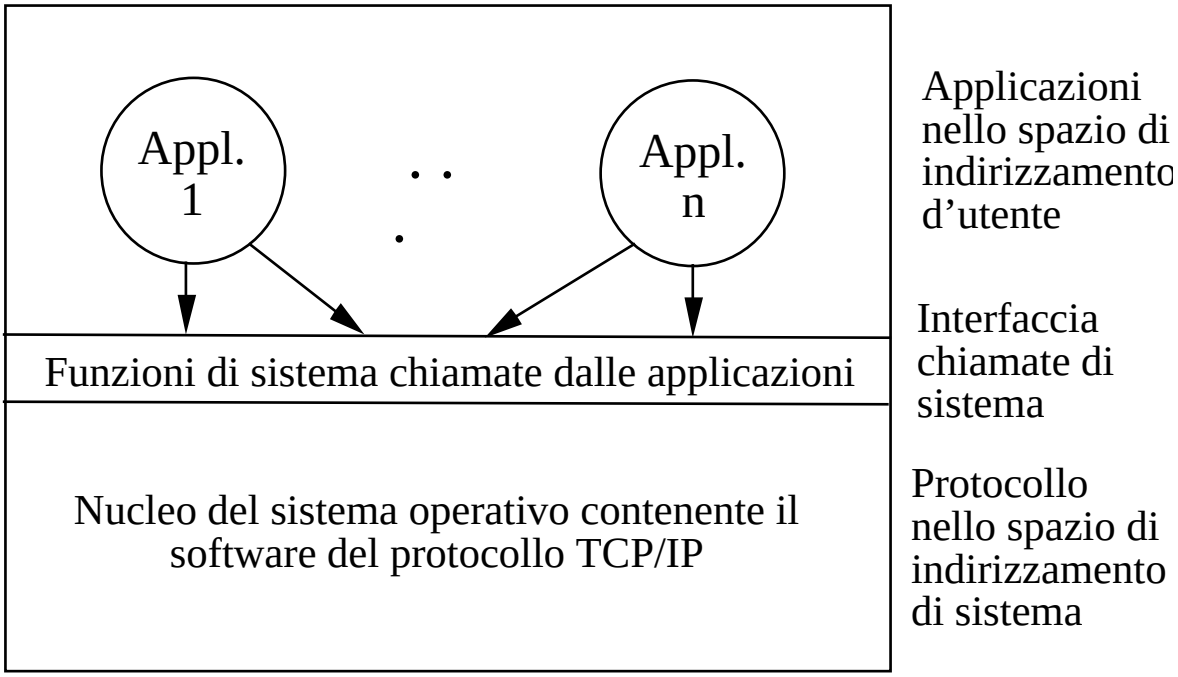
- Le primitive dell'interfaccia prevedono appositi parametri per specificare la famiglia di protocolli, il tipo di servizio richiesto, ed il protocollo all'interno della famiglia

Funzionalità a livello d'interfaccia

- Lo standard TCP/IP non definisce in realtà l'interfaccia socket vera e propria, ma solo le funzionalità richieste
- Esso fornisce inoltre un'interfaccia concettuale, che serve come esempio per chi vuole implementare lo standard
- In pratica, esistono poche implementazioni dell'interfaccia socket
- Le principali sono quella di Berkeley e quella definita dalla AT&T per la versione System V di UNIX
- Quest'ultima è nota con l'acronimo **TLI (*Transport Layer Interface*)**
- Un'implementazione deve fornire le seguenti funzionalità principali:

    - allocare le risorse locali di comunicazione
    - specificare i partner nella comunicazione
    - inizio della connessione (cliente)
    - attesa di una connessione (servente)
    - invio e ricezione di dati
    - segnalare l'arrivo di dati
    - terminazione di una connessione
    - rilascio delle risorse locali
    - gestione delle condizioni d'errore

Modalità di interazione tra applicazioni e TCP/IP

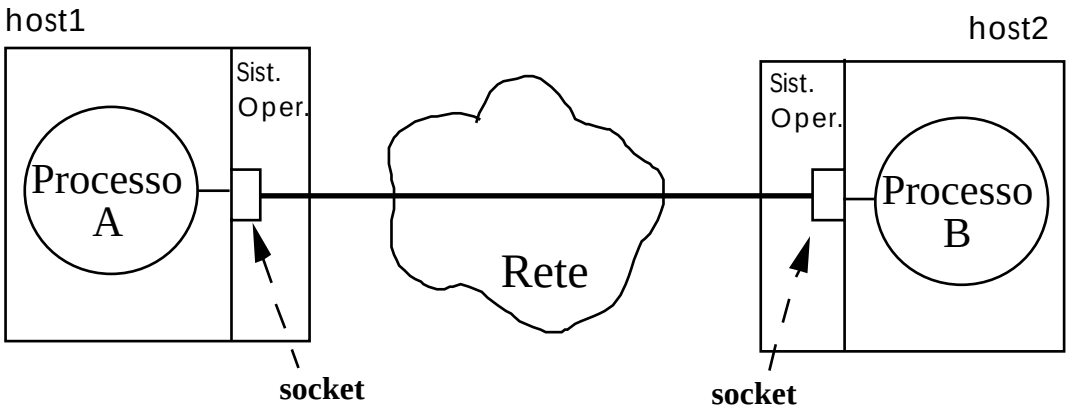


- Le applicazioni interagiscono con il software TCP/IP attraverso chiamate al sistema operativo
- Dal punto di vista dell'applicazione, una system call è come una chiamata di funzione, ma
- trasferisce il controllo all'interno del sistema operativo
- La funzione chiamata viene eseguita in stato supervisore, ed all'interno dello spazio di indirizzamento del sistema operativo

L’astrazione socket

- L'elemento centrale dell'interfaccia è il concetto di **socket** (presa), come astrazione per le comunicazioni di rete

- L'idea base di questo approccio è che una sola primitiva (socket) è sufficiente per creare una qualsiasi astrazione di comunicazione, indipendentemente dalle sue caratteristiche; le restanti chiamate riguardano i dettagli del suo uso



- Una socket si dice connessa quando collega stabilmente due partner

- Una socket TCP (circuito virtuale) deve essere connessa

- Una socket UDP può invece essere usata per inviare datagrammi a diversi partner; perciò tipicamente non è connessa

Strutture dati di sistema per le socket

- Quando un processo invoca la primitiva socket, il sistema operativo alloca una struttura dati atta a memorizzare le informazioni necessarie per la comunicazione, e

- inserisce un puntatore alla struttura dati allocata per la socket nella tabella dei descrittori associati al processo

- Tale tabella è la stessa di quella usata da UNIX per i descrittori dei file aperti dal processo

- In questo modo la gestione delle socket da parte del sistema operativo è consistente (ma non del tutto) con la gestione dei file e dei dispositivi tipica di UNIX (I/O indipendente dalla periferia)

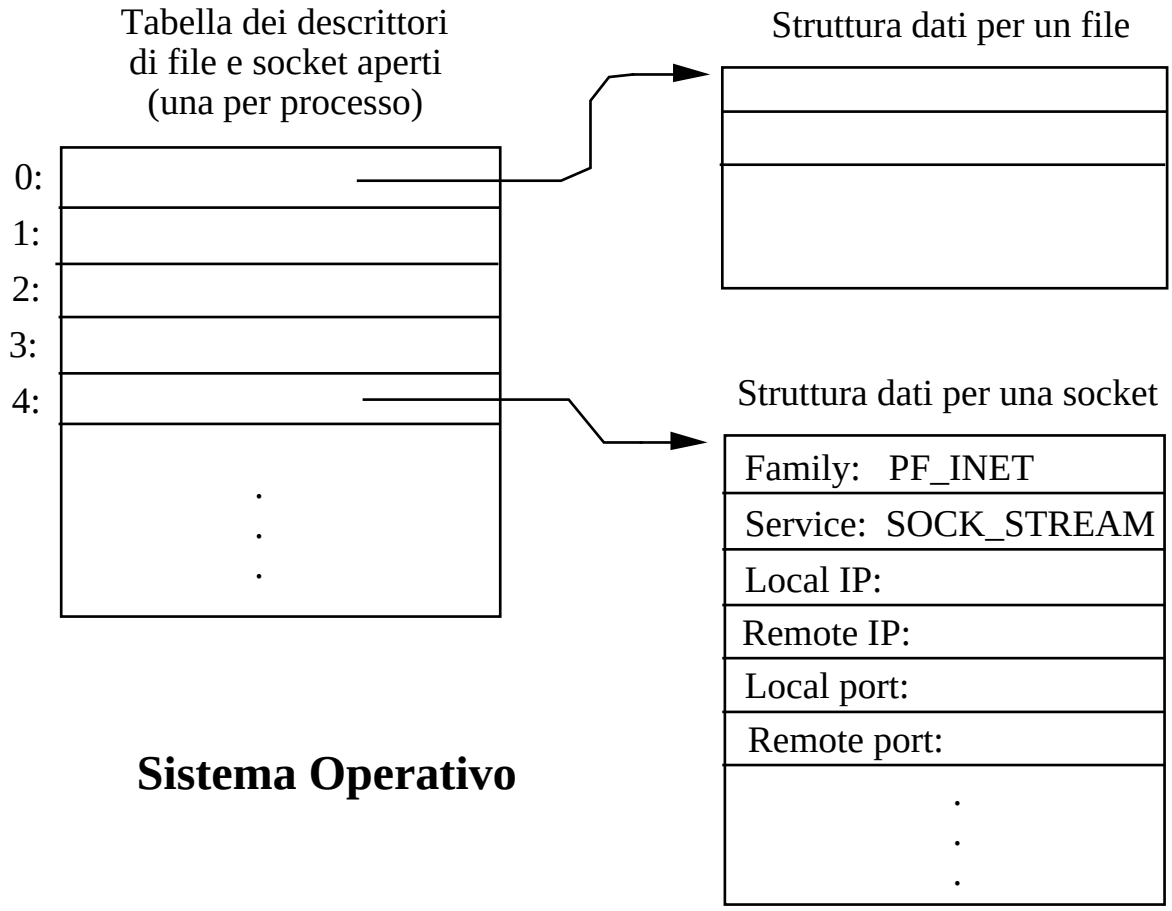


Tabella dei descrittori
di file e socket aperti
(una per processo)
0:
1:
2:
3:
4:
Struttura dati per un file
Struttura dati per una socket
Family: PF_INET
Service: SOCK_STREAM
Local IP:
Remote IP:
Local port:
Remote port:
Sistema Operativo

Strutture dati di sistema per le socket (segue)

- Una volta creata la struttura dati per la socket, occorre invocare altre chiamate di sistema per riempirne i campi, a seconda dei servizi richiesti

- Prima di usarla occorre infatti specificare gli estremi (*endpoints*) della socket, cioè i due partner nella comunicazione (nel caso di socket connesse), e le modalità d'uso

- Dopo di ciò la comunicazione sulla socket può finalmente aver luogo

- Dei due partner in una comunicazione, quello che prende l'iniziativa della comunicazione assume il ruolo di **cliente**, mentre l'altro viene detto **servente**

- Una socket usata da un cliente è detta **attiva**

- Una socket usata da un servente è detta **passiva**, perchè il servente deve essere in attesa passiva di comunicazioni su di essa

La primitiva socket

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

int s;
s = socket(protocol_family, service_type, protocol);
```

s	descrittore della socket
protocol_family	PF_INET per TCP/IP
service	circuito virtuale o datagramma (SOCK_STREAM o SOCK_DGRAM)
protocol	int. identificativo del protocollo (TCP/UDP)

- Crea una socket (alloca le strutture dati di sistema)
- Usata sia dal cliente che dal servente
- Ritorna un descrittore da usare come riferimento nelle restanti primitive
- Gli argomenti specificano:
  - la famiglia di protocolli che si vuole usare (in questo caso, TCP/IP)
  - il tipo di servizio all'interno della famiglia (TCP/IP fornisce attualmente i servizi circuito virtuale e datagramma)
  - il protocollo che fornisce il servizio, nel caso ve ne sia più d'uno (attualmente, TCP è il solo protocollo per il servizio circuito virtuale e UDP il solo per il servizio datagramma)

La primitiva connect

int connect(*s*, *remoteaddr*, *remoteaddrlen*);

s	descrittore della socket (ritornato da socket())
remoteaddr	servente sulla macchina remota (indirizzo IP, numero di porto)
remoteaddrlen	# bytes del secondo param.

- Invocata da un cliente (TCP o UDP) per stabilire una connessione con un servente remoto
- Ritorna un valore negativo in caso di insuccesso (es., irraggiungibilità del servente)
- Se la connect ha successo, il cliente può trasferire (leggere e/o scrivere) dati sulla socket
- Il secondo parametro specifica l'*endpoint* remoto nella connessione
- Esso è un puntatore ad una struttura di tipo sockaddr_in (definito in socket.h), contenente l'indirizzo IP dell'host su cui gira il servente, ed il numero di porto al quale ci si vuole connettere
- Il terzo parametro specifica la dimensione (in bytes) del secondo

La primitiva connect (segue)

- Nel caso di connessioni TCP (connect su una socket di tipo SOCK_STREAM), connect:
  - controlla se il descrittore è valido e non è già connesso
  - sceglie un *endpoint* locale (indirizzo IP, # porto)
  - preleva l'*endpoint* remoto dal secondo parametro e lo inserisce nelle strutture dati di sistema per la socket
  - apre la connessione col servente remoto
  - ritorna un intero che segnala il successo o meno nell'apertura della connessione

- Nel caso di connessioni UDP (connect su una socket di tipo SOCK_DGRAM), connect:
  - non controlla se il descrittore è valido
  - preleva l'*endpoint* remoto dal secondo parametro e lo inserisce nelle strutture dati di sistema per la socket
  - non contatta il servente
  - anche se la chiamata ha successo, non vuol dire che l'indirizzo remoto è valido, né che il servente è raggiungibile

- Esempio d'uso di connect:

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

int s;
struct sockaddr_in *sin;
...         /* invoca socket() */
if (connect(s, sin, sizeof(sin) < 0) {
            printf("Connessione non riuscita\n");
}
```

La primitiva write

int write(*s*, *buf*, *buflen*);

s	descrittore della socket (da socket())
buf	puntatore ai dati da spedire
buflen	# bytes da spedire

- Inoltra un messaggio al partner all'altro capo della socket
- Copia il messaggio in un buffer di I/O sistema
- L'applicazione chiamante può proseguire l'elaborazione mentre il messaggio viene inoltrato da TCP/IP
- Se il buffer di sistema si riempie la chiamata diviene bloccante
- In tal caso la write ritorna quando, per effetto dell'invio dei messaggi precedenti, il buffer si svuota e si crea spazio sufficiente per la copia
- Coincide con l'omonima primitiva di I/O di UNIX
- Esempio d'uso:

```
char    *msg = "Messaggio da inoltrare";

write(s, msg, strlen(msg));
```

La primitiva read

int read(*s*, *buf*, *buflen*);

s	descrittore della socket (da socket())
buf	puntatore all'area dati dove inserire i dati letti dalla socket
buflen	lunghezza di *buf*

- Estrae i byte di dati in arrivo sulla socket e li copia nell'area dati specificata
- Ritorna il numero di byte letti
- E' bloccante: se non sono arrivati dati, attende che arrivino
- Per una socket TCP, se arrivano più dati della dimensione del buffer, estrae solo quelli che entrano nel buffer
- Se arrivano meno dati, li legge tutti e ritorna il numero di byte letti =>
  tipicamente si inserisce una read in un ciclo quando si legge da una socket TCP
- Per una socket UDP, legge un intero messaggio (un datagramma d'utente)
- Se il buffer è più corto del messaggio UDP, riempie il buffer e getta via il resto del messaggio
- Esempio d'uso:

```
/* s è un descr. di socket TCP */
while ((n = read(s, bufptr, buflen) > 0) {
```

```
        bufptr += n;
        buflen -=n;    }
```

- Coincide con l'omonima primitiva di I/O di UNIX

La primitiva close

int close(*s*); *s* descrittore della socket (da socket())

- Se un solo processo sta usando la socket, dealloca la socket e rilascia le risorse associate (strutture dati di sistema)
- Se più processi condividono la socket, decrementa un contatore e dealloca la socket, rilasciando le risorse, solo quando il contatore vale zero
- Coincide con l'omonima primitiva di I/O di UNIX

Chiusura di una connessione TCP

- Poichè con TCP la comunicazione è bidirezionale, nel caso di applicazioni di tipo cliente/servente, la chiusura di una connessione può richiedere un coordinamento tra i due partner

- Il servente non può iniziare la terminazione di una connessione, in quanto non può sapere se il cliente invierà altre richieste

- Il cliente non può chiudere la connessione, finchè non è sicuro di aver ricevuto tutti i dati inviatigli dal servente (questo problema si pone solo quando la risposta ad una richiesta comporta il trasferimento di una quantità di dati variabile)

- Per coordinare cliente e servente, esiste la possibilità di una chiusura parziale della connessione tramite la primitiva shutdown

- Quando un cliente non ha più richieste, chiude parzialmente in uscita la socket, senza deallocarla

- Il servente riceve una segnalazione di *end-of-file* sulla socket

- Il servente invia l'ultima risposta e chiude la connessione

La primitiva shutdown

errcode = shutdown(*s*, *direction*);

s	descrittore della socket (ritornato da socket())
direction	specifica la direzione (chiusura in ingresso, uscita, ingr./uscita)

- Invocando shutdown su una socket TCP, il protocollo sottostante segnala la chiusura parziale della connessione al partner nella comunicazione

- Il parametro direction è un intero:
  - 0 chiusura della socket in ingresso (non sono più ammesse letture)
  - 1 chiusura della socket in uscita (non sono più ammesse scritture)
  - 2 chiusura della socket in entrambe le direzioni (non sono più ammesse letture né scritture)

Chiusura di una connessione UDP

- Per chiudere una socket UDP e rilasciare le risorse ad essa associate si usa close

- Quando una socket è stata chiusa, UDP rigetta ulteriori messaggi indirizzati al porto associato alla socket. Tuttavia UDP non notifica la chiusura all'entità all'altro capo della socket

- Di conseguenza, una applicazione che usa il protocollo di trasporto *connectionless* deve essere progettata in modo che il nodo remoto sappia quando deve chiudere la socket

- Si può usare shutdown() per una socket UDP, ma in tal caso non viene notificato nulla al processo all'altro estremo della socket

La primitiva bind

- Quando una socket viene creata, non ha nessuna nozione circa i suoi estremi locale e remoto
- Una applicazione chiama bind per specificare l'estremo locale
- Nel caso TCP/IP, l'estremo locale viene definito definito dal contenuto di una struttura di tipo sockaddr_in, che comprende un indirizzo IP ed un numero di porto
- E' usata essenzialmente da un servente per specificare il porto locale su cui desidera mettersi in attesa di richieste
- Esempio d'uso:

```
int s;     /* descr. di socket */
struct sockaddr_in *sin;
... /* definisce l'endpoint locale e invoca socket() */
if (bind(s, sin, sizeof(sin)) < 0) {
        printf("Errore nel bind\n");
}
```

La primitiva listen

int listen(*s, qlen*);	
s	descrittore della socket (da socket())
qlen	intero, lunghezza della coda

- Quando una socket viene creata, non è né attiva né passiva
- La primitiva listen viene invocata da un servente TCP per definire passiva la socket e renderla pronta ad accettare comunicazioni in arrivo
- Mentre un servente sta gestendo una richiesta arrivata su una socket passiva, una nuova richiesta può sopraggiungere
- Per questo motivo la primitiva listen prevede come secondo parametro la dimensione della coda da associare alla socket, per accodare le richieste dei clienti
- Esempio d'uso:

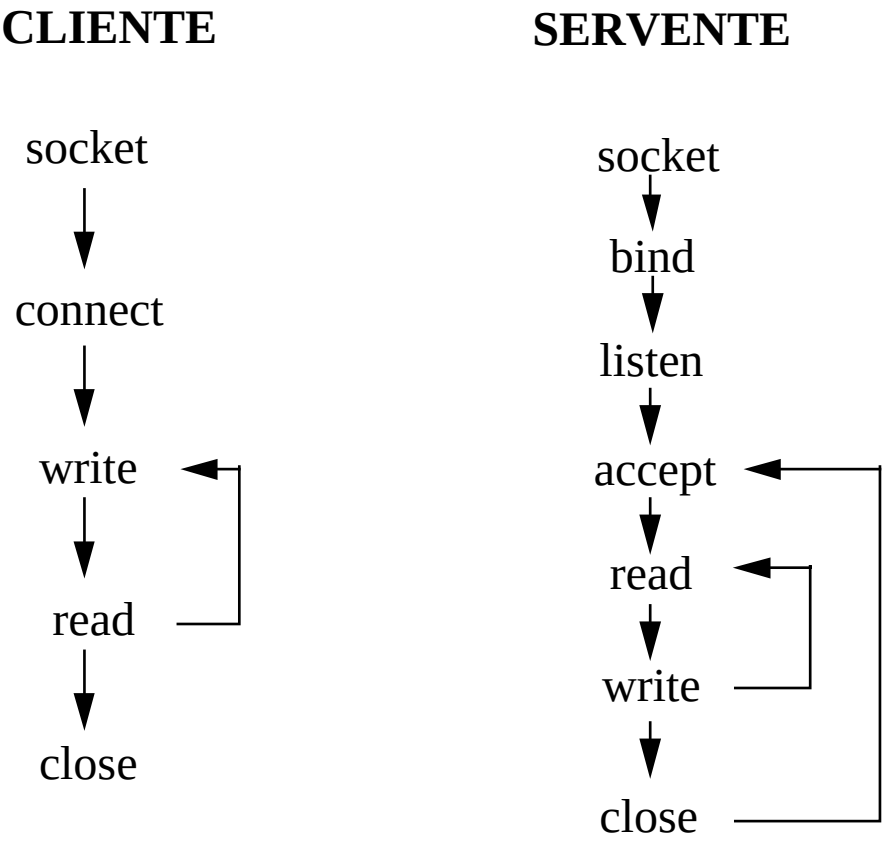
```
int s;              /* descr. di socket TCP */
int len = 5;   /* lunghezza della coda */
...
if (listen(s, len) < 0) {
        printf("Errore nella listen\n");
}
```

La primitiva accept

- Un servente invoca accept per prelevare una richiesta di connessione dalla coda associata ad una socket passiva
- accept ha per effetto quello di clonare la socket, creando una nuova socket di lavoro, di cui ritorna il relativo descrittore
- Il servente usa la nuova socket solo per comunicare col cliente che ha inoltrato la richiesta
- Il servente continua ad usare la socket iniziale per gestire le successive richieste di connessione
- Dope aver accettato una connessione, il servente può trasferire dati per interagire col cliente
- Quando ha finito, il servente chiude la socket

Schema d’uso delle primitive

- Le sequenze di chiamate delle primitive per la gestione dei socket da parte di un cliente e di un servente che usano TCP seguono lo schema seguente



- Il servente esegue un ciclo continuo, attendendo nuove connessioni su un porto prestabilito

- Accettata una connessione, il servente elabora la richiesta del cliente e poi chiude la connessione

- Nel caso UDP, tipicamente si usano le primitive sendto e receivefrom al posto rispettivamente di write e read

Riepilogo delle primitive per la gestione di socket

primitiva	significato
socket	*alloca un descrittore per le comunicazioni*
connect	*crea una connessione con un servente*
bind	*associa un indirizzo IP locale ed un porto alla socket (servente)*
listen	*rende passiva la socket e specifica la lunghezza della coda (servente)*
accept	*accetta una richiesta di connessione (serv.)*
read	*legge dati da una socket*
recv	*legge un datagramma in arrivo*
recvmsg	*legge un datagr. in arrivo (variante di recv)*
recvfrom	*legge un datagramma in arrivo e registra l'indirizzo del mittente*
write	*invia dati su una socket*
send	*invia un datagramma*
sendmsg	*invia un datagramma (variante di send)*
sendto	*invia un datagramma ad un indirizzo prestabilito*
shutdown	*chiude parzialmente una connessione*
close	*termina la comunicazione e rilascia il descrittore*

getsockopt	*ritorna le opzioni correnti di una socket*
setsockopt	*modifica le opzioni di una socket*
getpeername	*ritorna l'indirizzo del partner*

Sottoprogrammi d'utilità per la conversione di interi

- TCP/IP definisce una propria rappresentazione per gli interi usati nelle strutture dati per le socket (es.: il campo *port number* nella struttura sockaddr_in)

- Tale rappresentazione è nota come "**network byte order**", ed in realtà coincide con quella "big endian"

- L'interfaccia socket comprende alcune funzioni di libreria per eseguire le conversioni degli interi dal formato locale a quello di rete:

htons	host to network format - short (16 bit)
ntohs	network to host format - short (16 bit)
htonl	host to network format - long (32 bit)
ntohl	network to host format- long (32 bit)

- Le applicazioni dovrebbero sempre chiamare tali routine di conversione, in modo da garantire la portabilità

Sommario

- TCP/IP è una famiglia di protocolli di rete per l'interconnessione di sistemi eterogenei
- TCP/IP è la famiglia di protocolli attualmente adoperata sulla rete di calcolatori mondiale INTERNET, che conta alcuni milioni di nodi
- TCP/IP offre due protocolli di trasporto: TCP per il servizio circuito virtuale e UDP per il servizio datagramma
- Le applicazioni TCP/IP fanno riferimento ai nodi di rete con indirizzi in formato numerico (*dot notation*) o simbolico (*domain name*)

- UNIX fornisce l'astrazione delle socket per lo sviluppo di applicazioni di rete
- Tale astrazione è implementata in pratica nella famiglia di protocolli TCP/IP
- L'interazione tra applicazioni e TCP/IP avviene a mezzo di un insieme di chiamate al sistema operativo

- La primitiva socket alloca e ritorna il descrittore di una socket non connessa
- Una socket TCP deve essere connessa
- Una socket UDP viene connessa se si comunica sempre con lo stesso partner
- Un cliente (TCP o UDP) rende connessa una socket con connect
- Un servente TCP rende connessa una socket con bind e poi accettando connessioni da parte dei clienti (accept)
- Per poter accettare connessioni su una socket TCP occorre prima renderla passiva e creare una coda con listen
- Un servente UDP non rende connessa una socket; esso deve solo specificare l'*endpoint* locale con bind

- I partner in una comunicazione scambiano messaggi su una socket mediante le primitive read e write, o altre simili del tipo send/receive