Corsi di Laurea in Ingegneria Informatica
Ingegneria delle Telecomunicazioni
Ingegneria dell'Automazione

Corso di Reti di Calcolatori

Simon Pietro Romano (spromano@unina.it)
Antonio Pescapè (pescape@unina.it)
Giorgio Ventre (giorgio@unina.it)

Le socket di Berkeley

Le socket di Berkeley

a cura di Marcello Esposito (mesposit@unina.it)

Nota di Copyright

SOCKET: cosa sono? (1)

- Le socket rappresentano un'astrazione di canale di comunicazione tra processi (locali o remoti).
- Attraverso di esse un'applicazione può ricevere o trasmettere dati.
- I meccanismi restano (quasi) indipendenti dal supporto fisico su cui le informazioni viaggiano.
- Inizialmente nascono in ambiente UNIX
  - Negli anni 80 la Advanced Research Project Agency finanziò l'Università di Berkeley per implementare la suite TCP/IP nel sistema operativo Unix.
  - I ricercatori di Berkeley svilupparono il set originario di funzioni che fu chiamato *interfaccia socket*.
  - Esse originariamente apparvero nella versione 4.1cBSD di Unix.

4

SOCKET: cosa sono? (2)

- Si presentano sotto la forma di un'API (Application Programming Interface), cioè un insieme di funzioni scritte in C, che le applicazioni possono invocare per ricevere il servizio desiderato.
- Rappresentano una estensione delle API di UNIX per la gestione dell'I/O su periferica standard (files su disco, stampanti, etc).
- Questa API è poi divenuta uno standard *de facto*, ed oggi è diffusa nell'ambito di tutti i maggiori sistemi operativi (Linux, FreeBSD, Solaris, Windows... etc.).

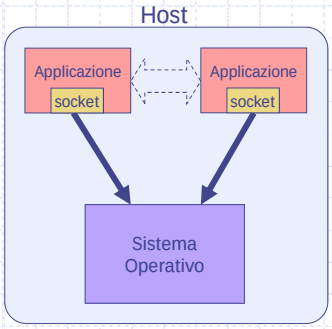
5

Interazione tra Applicazione e SO

- L'applicazione chiede al sistema operativo di utilizzare i servizi di rete
- Il sistema operativo crea una socket e la restituisce all'applicazione
  - restituito un socket descriptor
- L'applicazione utilizza la socket
  - Open, Read,Write,Close.
- L'applicazione chiude la socket e la restituisce al sistema operativo

6

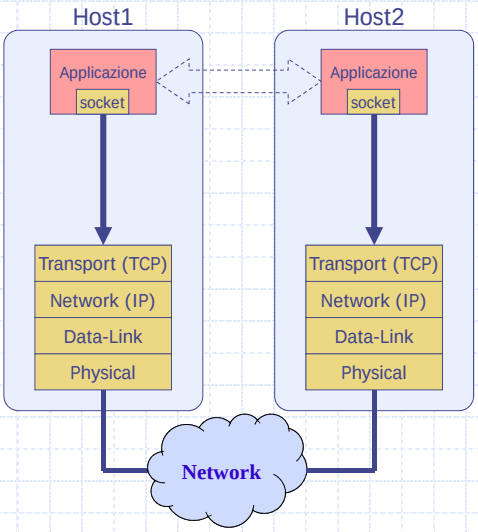
Caso d'uso: comunicazione locale



- Due applicazioni, localizzate sulla stessa macchina, scambiano dati tra di loro utilizzando l'interfaccia delle socket.
- Le socket utilizzate a questo scopo vengono comunemente definite Unix-domain socket.

7

Caso d'uso: comunicazione remota via TCP/IP

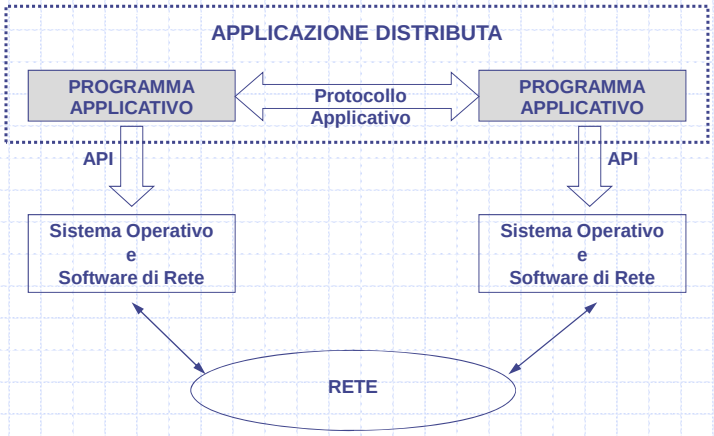


- Anche due applicazioni situate su macchine distinte possono scambiare informazioni secondo gli stessi meccanismi.
- Così funzionano telnet, ftp, ICQ, Napster.

8

Interfacce e protocolli

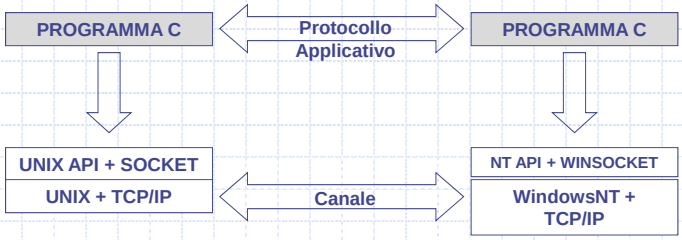
- Esempio di applicazione distribuita:



9

Interfacce e protocolli

- La configurazione di riferimento di una applicazione distribuita basata su TCP/IP e socket è il seguente:



10

Il problema della connessione

- Nel momento in cui una entità decide di instaurare una comunicazione con un'entità paritaria, come assicurarsi che quest'ultima sia disponibile?
  - La chiamata telefonica: chi desidera instaurare la comunicazione compone il numero del destinatario e attende durante il segnale di chiamata. Dall'altro lato uno squillo avverte di una chiamata in arrivo. Se si è disponibili alla comunicazione (si è in casa, si può ascoltare lo squillo e non si è sotto la doccia) si alza la cornetta. Lo squillo dal lato del chiamante termina. Da questo momento in poi la chiamata è instaurata e diviene simmetrica: chiunque può parlare quando vuole.
- E' necessario che il chiamante conosca l'indirizzo del chiamato e che il chiamato sia in attesa di eventuali comunicazioni.

11

Il paradigma Client-Server (C/S)

- Il chiamato è il server:
  - deve aver divulgato il proprio indirizzo
  - resta in attesa di chiamate
  - in genere viene contattato per fornire un servizio
- Il chiamante è il client:
  - conosce l'indirizzo del server
  - prende l'iniziativa di comunicare
  - usufruisce dei servizi messi a disposizione dal server

12

Il concetto di indirizzo

• Una comunicazione può essere identificata attraverso la quintupla:

{protocol, local-addr, local-process, foreign-addr, foreign-process}

• Una coppia {addr, process} identifica univocamente un terminale di comunicazione (end-point).

• Nel mondo IP, ad esempio:

  • local-addr e foreign-addr rappresentano indirizzi IP

  • local-process e foreign-process rappresentano numeri di porto

13

Server concorrente e iterativo

•Un server può ricevere chiamate anche da più client diversi.

•Ogni comunicazione richiederà un certo tempo prima di potersi considerare conclusa.

E se una chiamata arriva mentre il server è già impegnato in una comunicazione?

•Un server che accetti più comunicazioni contemporaneamente si definisce concorrente.

•Un server che accetti una sola comunicazione alla volta è detto iterativo.

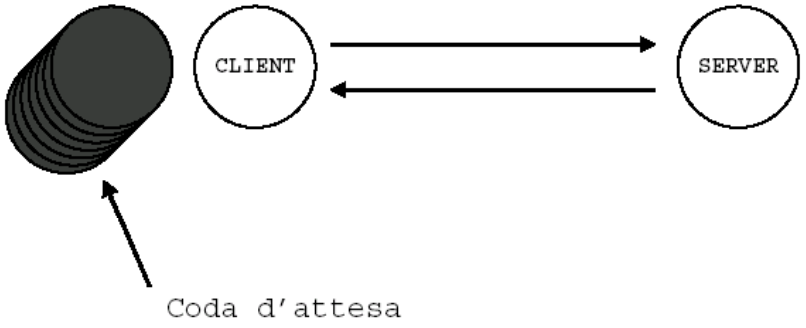
  •In questo ultimo caso una richiesta può essere servita solo quando la precedente si è già conclusa.

  •Questo è il paradigma applicato nel modello di comunicazione telefonica di base.

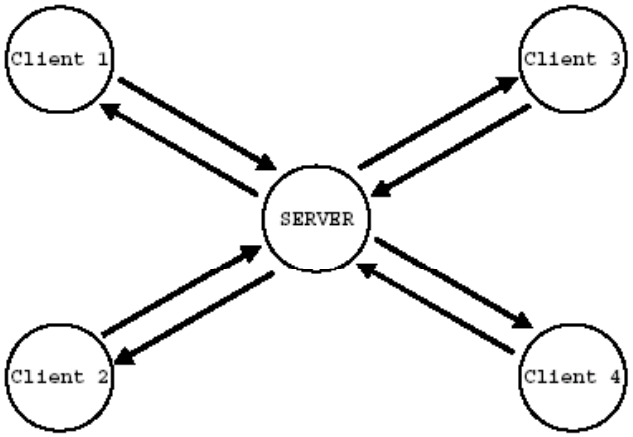
  •E l'avviso di chiamata?...

14

Server Iterativo



Server Concorrente



Il paradigma di comunicazione “Connection-Oriented”

- In una comunicazione dati Connection-Oriented, i due end-point dispongono di un canale di comunicazione che:
  - trasporta flussi
  - è affidabile
  - è dedicato
  - preserva l’ordine delle informazioni
- Il canale si comporta cioè come una sorta di “tubo”: tutto quello che viene inserito al suo interno, arriverà inalterato dall’altro lato e nello stesso ordine con cui è stato immesso.
- Non è detto che vengano però mantenuti i limiti dei messaggi.
- La comunicazione telefonica è più simile ad una comunicazione connection-oriented.

17

Il paradigma di comunicazione “Datagram”

- In una comunicazione Datagram (anche detta connectionless), il canale
  - trasporta messaggi
  - non è affidabile
  - è condiviso
  - non preserva l’ordine delle informazioni
- Se si inviano dieci messaggi dall’altro lato essi possono anche arrivare mescolati tra di loro e tra i messaggi appartenenti ad altre comunicazioni. I limiti dei messaggi vengono comunque preservati.
- La posta ordinaria è un esempio di comunicazione a datagramma.

18

Connection-Oriented vs Datagram

- Connection oriented.
  - Principali vantaggi:
    - Affidabilità
    - Controllo di flusso
  - Principali svantaggi:
    - Overhead per instaurare la connessione
- Datagram.
  - Principali Vantaggi
    - Basso overhead
  - Principali svantaggi
    - Nessun controllo sulla consegna
- Le socket che utilizzano i protocolli Internet sfruttano rispettivamente TCP (Transmission Control Protocol) e UDP (User Datagram Protocol) per implementare le due tipologie di comunicazione. In entrambi i casi il protocollo di livello inferiore è IP (che è un protocollo datagram).

19

Naming / Binding

- È l'operazione in cui associamo un indirizzo transport ad una socket già creata
- In questo modo l'indirizzo diventa noto al sistema operativo ed altre socket sono in grado di stabilire una connessione

20

Byte Stream e Datagram

- È possibile impostare il protocollo che verrà utilizzato per il trasferimento dei dati
- Nel caso delle socket abbiamo due opzioni:
  - **byte-stream**: I dati vengono trasferiti come una sequenza ordinata ed affidabile di byte (`SOCK_STREAM`)
  - **Datagram**: I dati vengono inviati come messaggi indipendenti ed inaffidabili (`SOCK_DGRAM`)

21

Progettazione di un Server TCP

- Creazione di un endpoint
  - Richiesta al sitema operativo
- Collegamento dell'endpoint ad una porta
  - Ascolto sulla porta
    - Processo sospeso in attesa
- Accettazione della richiesta di un client
- Letture e scritture sulla connessione
- Chiusura della connessione

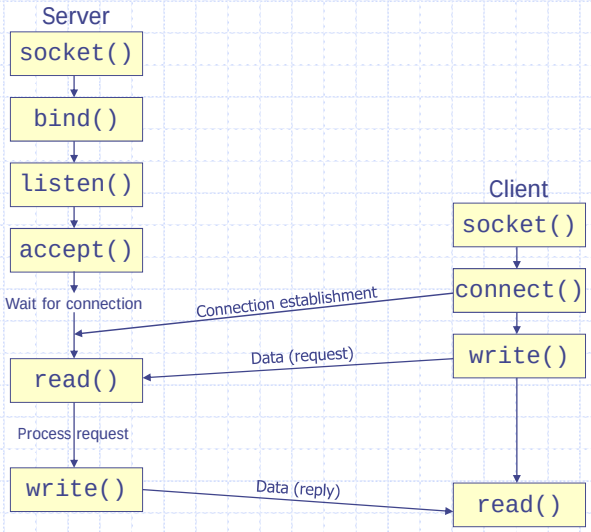
22

Progettazione di un Client TCP

- Creazione di un endpoint
  - Richiesta al sistema operativo
- Creazione della connessione
  - Implementa open di TCP (3-way handshake)
- Lettura e scrittura sulla connessione
  - Analogo a operazioni su file in Unix
- Chiusura della connessione
  - Implementa close di TCP (4-way handshake)

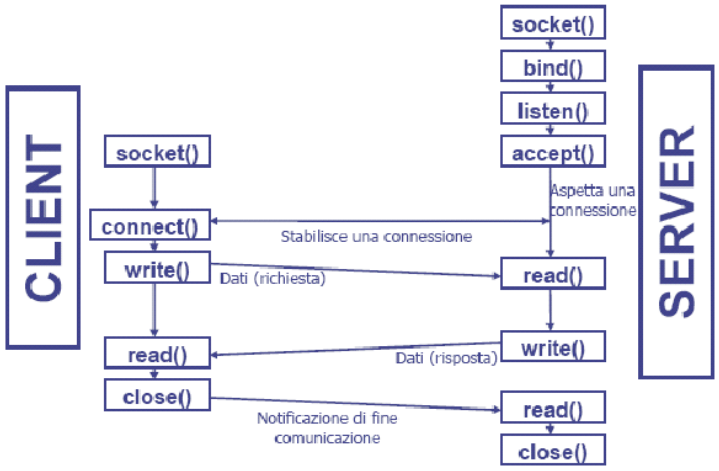
23

Le chiamate per una comunicazione Connection-Oriented



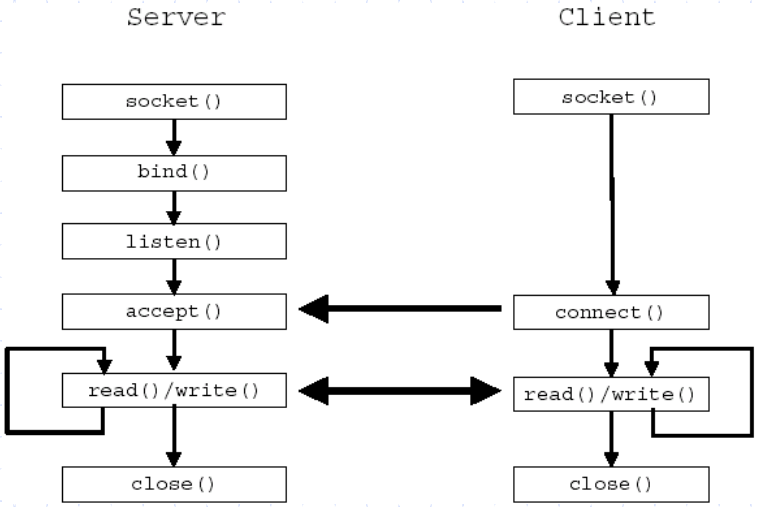
24

Le chiamate per una comunicazione Connection-Oriented



25

Interazione Client/Server Iterativo



Progettazione di un Server UDP

- Creazione di un endpoint
  - Richiesta al sistema operativo
- Collegamento dell'endpoint ad una porta
  - open passiva in attesa di ricevere datagram
- Ricezione ed invio di datagram
- Chiusura dell'endpoint

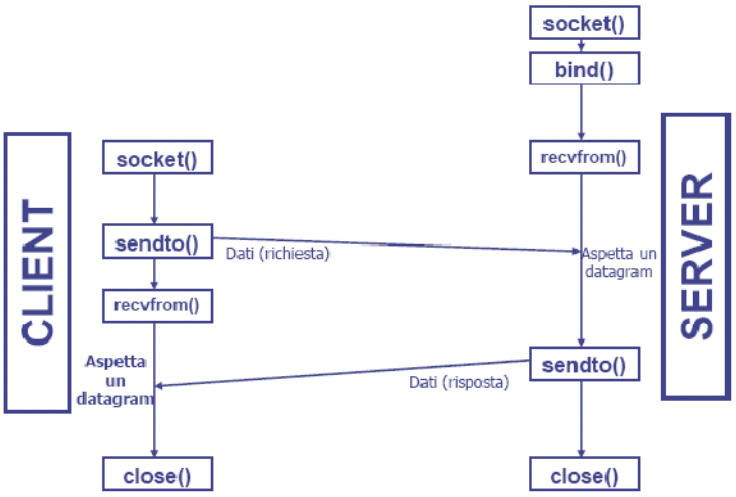
27

Progettazione di un Client UDP

- Creazione di un endpoint
  - Richiesta al sistema operativo
- Invio e ricezione di datagram
- Chiusura dell'endpoint

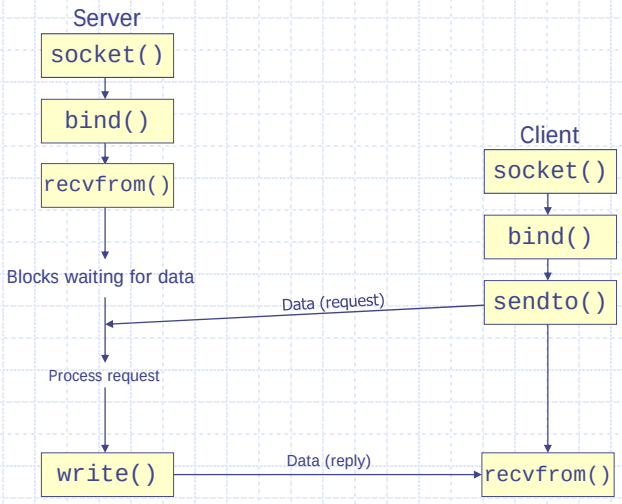
28

Le chiamate per una comunicazione Datagram



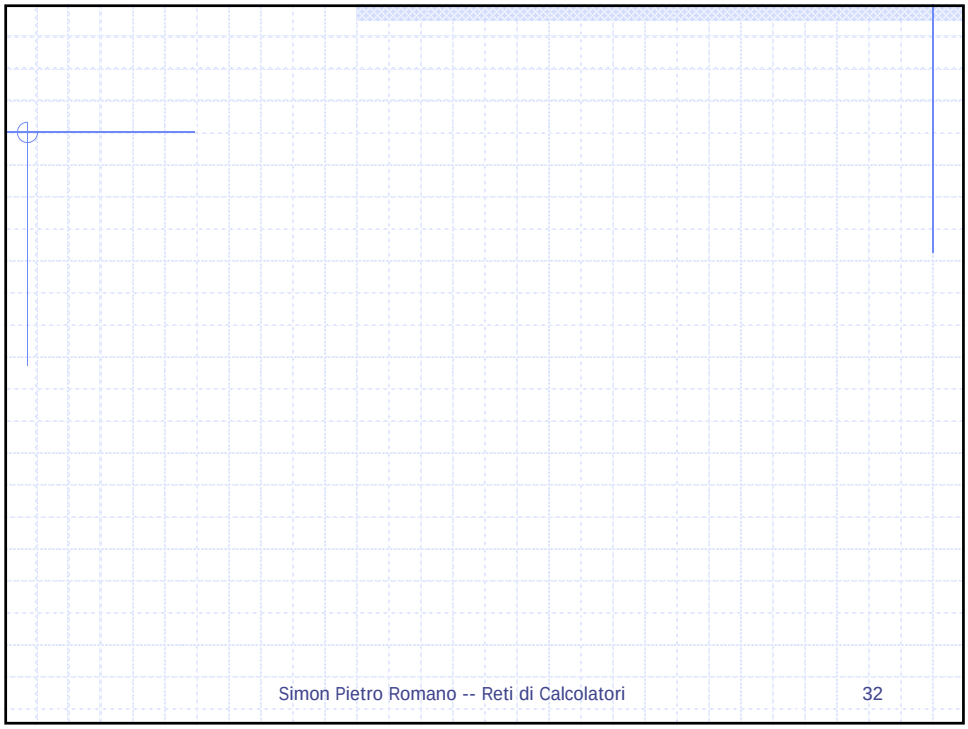
29

Le chiamate per una comunicazione Datagram



30

La programmazione delle socket

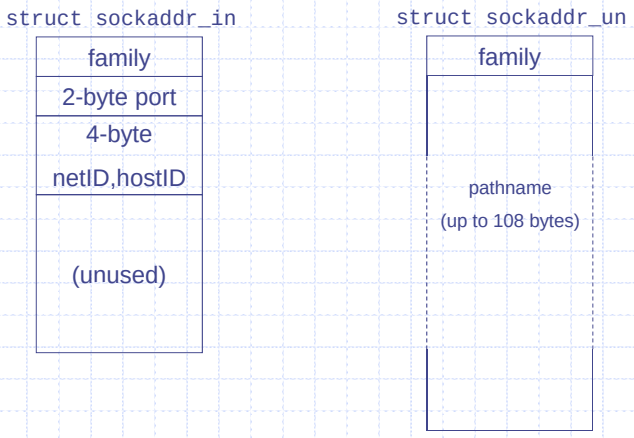


Le strutture dati per le socket: il trattamento degli indirizzi (1)

```
<sys/socket.h>
struct sockaddr {
  u_short sa_family;        /* address family: AF_xxx value */
  char    sa_data[14];      /* up to 14 bytes of protocol-specific address */
};

<netinet/in.h>
struct in_addr {
  u_long s_addr;            /* 32-bit netid/hostid network byte ordered */
};

struct sockaddr_in {
  short           sin_family;       /* AF_INET */
  u_short         sin_port;         /16-bit port number network byte ordered */
  struct in_addr  sin_addr;
  char            sin_zero[8];      /* unused */
};

<sys/un.h>
struct sockaddr_un {
  short   sun_family;       /* AF_UNIX */
  char    sun_path[108];    /* pathname */
};
```

33

Le strutture dati per le socket: il trattamento degli indirizzi (2)



34

La system-call `socket()`

- Questa system-call serve ad instanziare un nuovo descrittore di socket.
- Esso verrà utilizzato in tutte le successive chiamate

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

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

- `family` può essere: `AF_UNIX`, `AF_INET`...
- type può essere: `SOCK_STREAM`, `SOCK_DGRAM`, `SOCK_RAW`, `SOCK_SEQPACKET`, `SOCK_RDM`.
- `protocol` indica il protocollo utilizzato.
- il valore restituito è un descrittore di socket (di tipo `int` secondo lo stile Unix).
- Della quintupla, dopo la chiamata `socket()`, resta specificato solo il primo campo:

  {protocol, local-addr, local-process, foreign-addr, foreign-process}

35

La system-call `bind()` (1)

- Questa system-call serve ad assegnare un indirizzo locale (name) ad una socket.

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

int bind(int sockfd, struct sockaddr *myaddr, int addrlen);
```

- `sockfd` è il descrittore di socket restituito da `socket()`.
- `myaddr` punta ad un generico indirizzo, mentre `addrlen` è la lunghezza di quest'ultimo.
- il valore restituito è indicativo del successo dell'operazione.
- Della quintupla, dopo la chiamata `bind()`, restano specificati il secondo ed il terzo campo, cioè gli estremi locali della comunicazione:

  {protocol, local-addr, local-process, foreign-addr, foreign-process}

36

La system-call `bind()` (2)

- Essa può essere invocata in una molteplicità di casi:
  - un **server** vuole registrare i suoi estremi (già divulgati precedentemente) presso il sistema sul quale si trova. In questo modo è come se dicesse:- "*Questo è il mio indirizzo e tutti i messaggi inviati ad esso devono essere consegnati a me".*
    - Ciò accade sia per i server connection-oriented che per quelli connectionless.
  - un client vuole registrare uno specifico indirizzo per se stesso.
  - un client connectionless vuole assicurarsi uno specifico indirizzo poiché è solo attraverso di esso che può essere raggiunto da un server al quale aveva in precedenza inoltrato una richiesta.
- Può restituire una condizione di errore, per esempio, se l'indirizzo al quale si desidera "legarsi" risulta già occupato.

37

La system-call `connect()` (1)

- Attraverso la chiamata `connect()`, subito dopo una chiamata `socket()`, un processo **client** stabilisce una connessione con un server.

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

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

- `sockfd` è il descrittore di socket restituito da `socket()`.
- `servaddr` punta all'indirizzo (generico) del server, e `addrlen` è sempre la lunghezza di quest'ultimo.
- il valore restituito è indicativo del successo dell'operazione.
- Della quintupla, dopo la chiamata `connect()`, restano specificati tutti i campi relativi agli indirizzi:

  {protocol, local-addr, local-process, foreign-addr, foreign-process}

38

La system-call `connect()` (2)

- Per le comunicazioni connection-oriented, la chiamata `connect()` scatena una fase di segnalazione tra il client ed il server intesa a stabilire realmente la connessione, ed a concordare una serie di parametri che la caratterizzano. In questo caso la `connect()` è bloccante e non ritorna se non dopo aver instaurato la connessione (o, eventualmente, con una condizione di errore).
- Un client non deve necessariamente realizzare una chiamata a `bind()` prima della `connect()`, poiché è il sistema che assegna automaticamente un indirizzo valido tra quelli disponibili. Questo è il motivo per cui tutta la quintupla risulterà valorizzata dopo una chiamata a `connect()`.
- Per un client connectionless c'è ancora la possibilità di invocare la chiamata `connect()`. In questo caso, però, il server non viene realmente contattato (potrebbe anche non essere attivo), ma si produce semplicemente la memorizzazione locale dell'indirizzo del server con la conseguenza che:
  - ogni successivo messaggio scritto sulla socket sarà diretto a quel server;
  - ogni messaggio ricevuto sulla socket verrà accettato solo se proveniente da quel server.

39

La system-call `listen()`

- Attraverso la chiamata `listen()`, un **server** manifesta la sua volontà di apprestarsi a ricevere connessioni.
- Generalmente si esegue dopo `socket()` e `bind()` e prima di `accept()`.

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

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

- `sockfd` è il descrittore di socket restituito da `socket()`.
- `qlen` indica quante richieste di connessione possono essere accodate dal sistema in attesa di essere servite.
- il valore restituito è indicativo del successo dell'operazione.

40

La system-call `accept()` (1)

- Dopo la chiamata `listen()`, un **server** si mette realmente in attesa di connessioni attraverso una chiamata ad `accept()`.

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

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

- `sockfd` è il descrittore di socket restituito da `socket()`.
- `peer` e `addrlen` sono parametri di ingresso-uscita:
  - prima dell' `accept()`, devono essere impostati coerentemente con le caratteristiche delle strutture dati allocate.
  - dopo l'`accept()`, contengono l'indirizzo del client di cui si è accettata la connessione, con la sua lunghezza (minore o uguale a quella preimpostata).

41

La system-call `accept()` (2)

- `accept()` preleva la prima richiesta di connessione dalla coda e crea una nuova socket avente le stesse proprietà di `sockfd`, supponendo implicitamente che il servente sia concorrente.
- Se la coda è vuota la chiamata è invece bloccante.
- `accept()` restituisce fino a tre valori:
  - se è andata a buon fine restituisce
    - il nuovo descrittore di socket;
    - l'indirizzo del client di cui si è accettata la connessione;
    - la sua lunghezza dell'indirizzo.
  - se non è andata a buon fine
    - il codice relativo all'errore verificatosi.

42

La system-call `accept()` (3)

- Il server quindi utilizza due socket diversi per ogni connessione con un client
  - il socket di ascolto (listening socket) è quello creato dalla funzione **socket()**
    - utilizzato per tutta la vita del processo
    - in genere usato solo per accettare richieste di connessione
  - il socket connesso (connected socket) è quello creato dalla funzione **accept()**
    - usato solo per la connessione con un certo client
    - usato per lo scambio dei dati con il client
- I due socket identificano due connessioni distinte

Esempio di `accept()` in un server concorrente.

```
int sockfd, newsockfd;

if ( (sockfd = socket( … ) ) < 0 )
    err_sys("socket error");
if ( bind(sockfd, … ) < 0 )
    err_sys("bind error");
if ( listen(sockfd, 5) < 0 )
    err_sys("listen error");

for ( ; ; ) {
    newsockfd = accept(sockfd, … );     /* blocks */
    if (newsockfd < 0)
          err_sys("accept errror");

    if (fork() == 0) {
          close(sockfd);                /* child */
          doit(newsockfd);              /* process the request */
          close(newsockfd);
          exit(0);
    }

    close(newsockfd);                   /* parent */
}
```

- Dopo l'`accept()` il descrittore `newsockfd` ha la quintupla tutta impostata, ed è pronto ad essere utilizzato.
- `sockfd`, invece, continua ad avere impostati solo i primi tre campi e può essere usato per accettare le altre connessioni, senza la necessità di istanziare, per questo, una nuova socket.

Esempio di `accept()` in un server iterativo.

```
int sockfd, newsockfd;

if ( (sockfd = socket( … ) ) < 0 )
   err_sys("socket error");
if ( bind(sockfd, … ) < 0 )
   err_sys("bind error");
if ( listen(sockfd, 5) < 0 )
   err_sys("listen error");

for ( ; ; ) {
   newsockfd = accept(sockfd, … );        /* blocks */
   if (newsockfd < 0)
        err_sys("accept errror");

   doit(newsockfd);                /* process the request */
   close(newsockfd);
}
```

45

Ricevere ed inviare i dati

- Una volta utilizzate le precedenti chiamate, la "connessione" è stata predisposta.
- La quintupla risulta completamente impostata.
- A questo punto chiunque può inviare o ricevere dati.
- Per questo, è necessario aver concordato un protocollo comune.
  - Per esempio, nella comunicazione telefonica, il chiamato parla per primo e risponde:- "Pronto, chi è?", e quindi il chiamante fornisce la propria identità.

46

Le system-call `send()` e `sendto()`

- `send()` e `sendto()` si utilizzano per inviare dati verso l'altro terminale di comunicazione.

```
int send(int sockfd, char *buff, int nbytes, int flags);
int sendto(int sockfd, char *buff, int nbytes, int flags,
                struct sockaddr *to, int addrlen);
```

- `sockfd` è il descrittore restituito dalla chiamata `socket()`.
- `buff` punta all'inizio dell'area di memoria contenente i dati da inviare.
- `nbytes` indica la lunghezza in bytes del buffer, e quindi, il numero di bytes da inviare.
- `to` e `addrlen` indicano l'indirizzo del destinatario, con la sua lunghezza.
- `flags` abilita particolari opzioni. In generale è pari a `0`.
- entrambe restituiscono il numero di bytes effettivamente inviati.

47

Le system-call `recv()` e `recvfrom()`

- `recv()` e `recvfrom()` si utilizzano per ricevere dati dall'altro terminale di comunicazione.

```
int recv(int sockfd, char *buff, int nbytes, int flags);
int recvfrom(int sockfd, char *buff, int nbytes, int flags,
                struct sockaddr *from, int *addrlen);
```

- `sockfd` è il descrittore restituito dalla chiamata `socket()`.
- `buff` punta all'inizio dell'area di memoria in cui devono essere ricopiati i dati ricevuti.
- `nbytes` è un parametro di ingresso che indica la lunghezza del buffer.
- `from` e `addrlen` contengono, dopo la chiamata, l'indirizzo del mittente con la sua lunghezza.
- `flags` abilita particolari opzioni. In generale è pari a `0`.
- entrambe restituiscono il numero di bytes ricevuti (minore o uguale a `nbytes`).
- in assenza di dati da leggere, la chiamata è bloccante.

48

La system-call `close()`

- Chiude una socket e rilascia le risorse ad essa associate.

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

- in seguito a questa chiamata, eventuali dati pendenti, vengono inviati al destinatario prima che la socket venga chiusa.

49

Il marshalling dei parametri (1)

- Quando si invia un dato sulla rete, in generale, nulla si può ipotizzare sulla architettura dell'host ricevente.
- E' necessario quindi che i dati in transito siano codificati secondo una convenzione standard.
- Con il termine *marshalling* si intende la traduzione di un'informazione in un formato prefissato e comprensibile universalmente.
- L'operazione inversa è detta *un-marshalling*.
- E' un'operazione tipica del sesto livello della pila OSI (presentazione) il cui intento è di assicurare portabilità ad un programma.

- Se, ad esempio, l'host trasmittente è dotato di un processore Intel (little-endian) e l'host ricevente è dotato invece di un processore Motorola (big-endian), lo scambio tra questi non può avvenire senza uniformare le convenzioni sulla codifica dei dati.

50

Il marshalling dei parametri (2)

- Una serie di funzioni è stata prevista, nell'ambito della comunicazione su Internet, proprio a questo scopo .
- Vanno sotto il nome di *Byte Ordering Routines:*
  - da host byte order a TCP/IP byte order (big-endian) e viceversa
- Sui sistemi che adottano la stessa convenzione fissata per Internet, queste routines sono implementate come "null-macros" (funzioni 'vuote').

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

u_long htonl(u_long hostlong);          /* host to net long */
u_short htons(u_short hostshort);       /* host to net short */
u_long ntohl(u_long netlong);           /* net to host long */
u_short ntohs(u_short netshort);        /* net to host short */
```

- E' implicito in queste funzioni che uno short occupi 16 bit e un long ne occupi 32.

51

Operazioni sui buffer (1)

- Le classiche funzioni standard del C per operare sulle stringhe (`strcpy()`, `strcmp()`, etc.) non sono adatte per operare sui buffer di trasmissione e ricezione.
- Esse infatti ipotizzano che ogni stringa sia terminata dal carattere `null` e, di conseguenza, non contenga caratteri `null`.
- Ciò non può essere considerato vero per i dati che si trasmettono e che si ricevono sulle socket.
- E' stato necessario quindi prevedere altre funzioni per le operazioni sui buffer.

52

Operazioni sui buffer (2)

```
bcopy(char *src, char *dest, int nbytes);
bzero(char *dest, int nbytes);
int bcmp(char *ptr1, char *ptr2, int nbytes);
```

- `bcopy()` copia `nbytes` bytes dalla locazione `src` alla locazione `dest`.
- `bzero()` imposta a zero `nbytes` a partire dalla locazione `dest`.
- `bcmp()` confronta `nbytes` bytes a partire dalle locazioni `ptr1` e `ptr2`, restituendo `0` se essi sono identici, altrimenti un valore diverso da `0`.

53

Conversione di indirizzi

- Poiché spesso nel mondo Internet gli indirizzi vengono espressi in notazione *dotted-decimal* (p.es. 192.168.1.1), sono state previste due routine per la conversione tra questo formato e il formato `in_addr`.

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

unsigned long inet_addr(char *ptr);
char *inet_ntoa(struct in_addr inaddr);
```

- `inet_addr()` converte una stringa (C-style) dalla notazione dotted-decimal alla notazione `in_addr` (che è un intero a 32 bit).
- `inet_ntoa()` effettua la conversione opposta.

54

Esercizio

- Scrivere un programma server ed un programma client che funzionino nel seguente modo:
  - il client chiede sullo standard input una stringa di caratteri e la invia al server;
  - il server riceve la stringa, la visualizza e la invia nuovamente al client;
  - anche il client visualizza la stringa appena ricevuta.
- Questo è un esempio di programma *echo*.

55