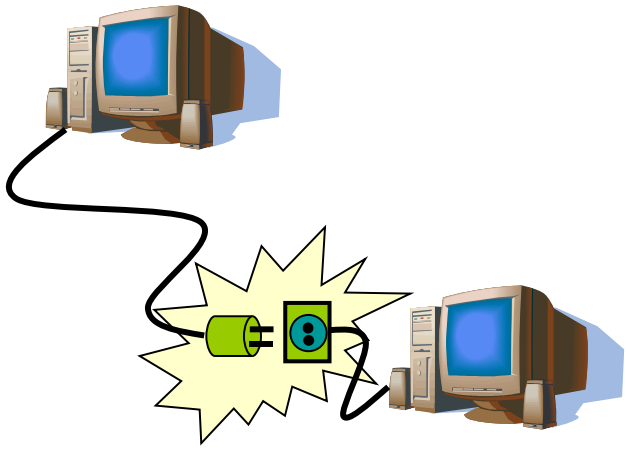
Socket API



- Socket Interface per Unix
- Modello client-server
- API in C
- API in Java (java.net)

Modello client-server

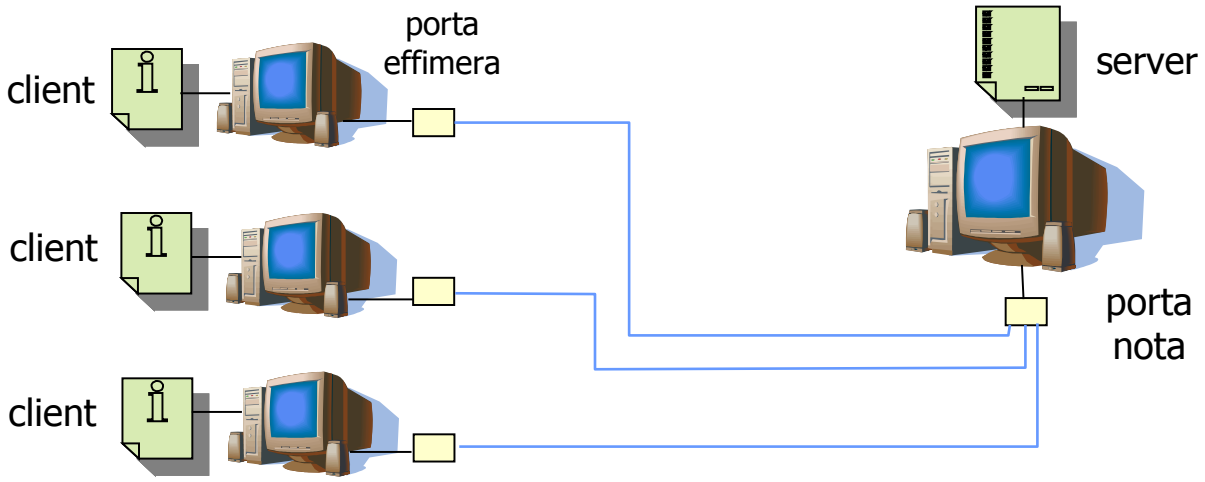
- **Il server fornisce servizi sulla rete**
  - Viene eseguita l'applicazione server su un host
  - L'applicazione attende connessioni dalla rete
- **Il client usufruisce del servizio attraverso la rete**
  - Deve conoscere l'indirizzo del server (host+porta)
  - Deve effettuare la richiesta di connessione
  - Scambia dati con il server

- **Sia il client che il server sono programmi applicativi che utilizzano i servizi di rete**
  - Talvolta client e server si usano per riferirsi all'host su cui viene eseguito l'applicativo

Server

- **Su un host possono essere in esecuzione più applicativi server**
  - I server sono in ascolto (LISTEN) su porte diverse
    - HTTP su porta 80 TCP
    - FTP su porta 21 TCP
    - Telnet su porta 23 TCP
  - Su UNIX **`inetd`** è un super-server capace di gestire le richieste su più porte
    - Il file di configurazione /etc/inetd.conf specifica il programma server specifico da attivare quando arriva una richiesta su una data porta

Client multipli



- **I client aprono connessioni col server (apertura attiva)**
- **Il server apre una connessione su richiesta di un client (apertura passiva)**
  - Un server sequenziale serve le richieste una dopo l'altra (coda delle richieste)
    - Generalmente per server UDP (senza connessione)
  - Un server parallelo divide il tempo fra le varie richieste (multithread)
    - Generalemnte per server TCP (con connessione)

netstat

- **Stampa di informazioni sul sottosistema di rete**
  - Lista dei socket aperti
  - Routing table (come route)
  - Appartenenza a gruppi multicast
  - Caratteristiche delle interfacce di rete (come ifconfig)
  - Statistiche per ogni protocollo

netstat -p -n -a --inet

  - mostra i socket IP (--inet)
  - mostra sia i socket aperti che in ascolto (-a = all)
  - mostra indirizzi IP e di porta numerici (-n)
  - mostra il processo che possiede il socket (-p da root)

netstat -p -n --inet -a

```
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address  Foreign Address  State
    PID/Program name
tcp        0    283 10.6.1.9:23    10.0.0.1:1855    ESTABLISHED
    16907/in.telnetd: f
tcp        0      0 10.6.1.9:3219  10.0.0.1:22      ESTABLISHED 5134/ssh
tcp        0      0 0.0.0.0:6000   0.0.0.0:*        LISTEN      645/X
tcp        0      0 0.0.0.0:3047   0.0.0.0:*        LISTEN      635/kdm
tcp        0      0 0.0.0.0:139    0.0.0.0:*        LISTEN      587/smbd
tcp        0      0 0.0.0.0:22     0.0.0.0:*        LISTEN      550/sshd2
tcp        0      0 0.0.0.0:25     0.0.0.0:*        LISTEN      510/master
tcp        0      0 0.0.0.0:23     0.0.0.0:*        LISTEN      408/inetd
tcp        0      0 0.0.0.0:21     0.0.0.0:*        LISTEN      408/inetd
tcp        0      0 0.0.0.0:111    0.0.0.0:*        LISTEN
    349/portmap
udp        0      0 0.0.0.0:177    0.0.0.0:*                    635/kdm
udp        0      0 0.0.0.0:111    0.0.0.0:*
    349/portmap
raw        0      0 0.0.0.0:1      0.0.0.0:*           7        -
raw        0      0 0.0.0.0:6      0.0.0.0:*           7        -
```

coda di ricezione (byte in coda)

coda di trasmissione (byte senza ack)

socket in C

- **Le API per l'uso di socket in C comprendono**
  - Definizione di tipi di dati (indirizzi IP, indirizzi di socket)
  - Funzioni di utilità (conversione dati, risoluzione dei nomi DNS)
  - Funzioni per la gestione dei socket (creazione, connessione)

- **I socket non sono necessariamente relativi al protocollo TCP/IP o UDP**
  - Si possono specificare la famiglia e il tipo di protocollo
  - Sono possibili modalità con connessione (es. TCP) che senza connessione (es. UDP)

Indirizzo di socket

```
struct sockaddr_in {
  u_char          sin_len;
  u_short         sin_family;
  u_short         sin_port;
  struct in_addr  sin_addr;
  char            sin_zero[8];
}
```

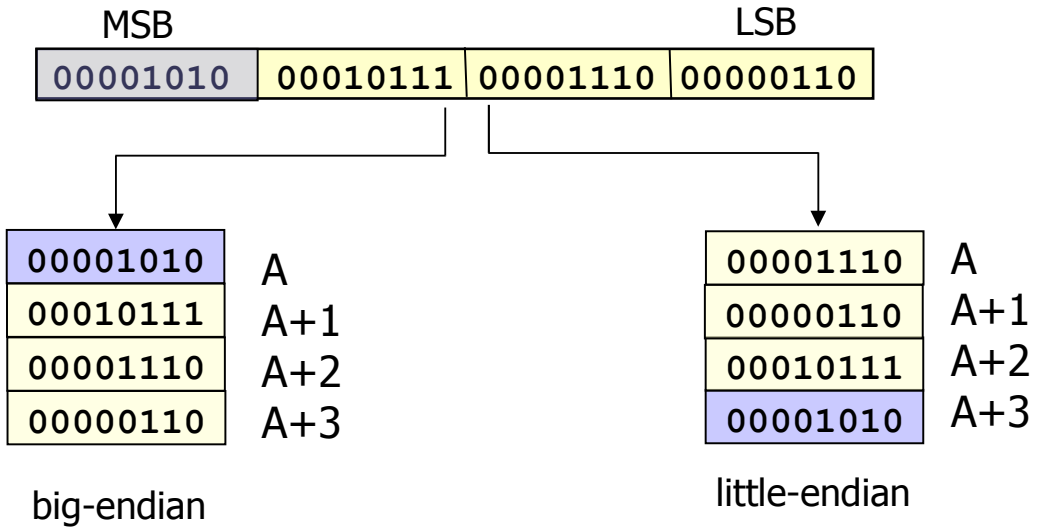
- **Permette di definire un indirizzo di un socket**
  - **`sin_family`** individua la famiglia del protocollo (IPv4 – AF_INET -, IPv6, UNIX,..)
  - **`sin_port`** individua la porta locale (16 bit)
  - **`in_addr`** individua l'indirizzo di rete (**`sin_addr.s_addr`** è un **`u_long`**)

Tipi di socket

- **Per creare un socket occorre specificare**
  - Famiglia
    - PF_INET utilizza IPv4
    - PF_UNIX, PF_LOCAL per connessioni locali
  - Tipo
    - SOCK_STREAM con connessione (es. TCP)
    - SOCK_DGRAM senza connessione (es. UDP)
    - SOCK_RAW utilizza direttamente il livello di rete
  - Protocollo
    - E' necessario specificarlo solo se Famiglia e Tipo non lo individuano univocamente (es. PF_INET e SOCKS_STREAM $\rightarrow$ tcp)
    - E' un numero (le associazioni nome,numero sono in /etc/protocol)

Ordine dei byte

- **Organizzazione dei byte in memoria/nei pacchetti**



- I protocolli TCP/IP usano il sistema big-endian (header del pacchetto)
- Se si scambiano dati binari bisogna prestare attenzione...

Conversione

- **Sono disponibili funzioni per la conversione dal formato dell'host a quello di rete (big-endian)**
  - `u_short htons(u_short host_short)`
  - `u_short ntohs(u_short network_short)`
  - `u_long htonl(u_long host_long)`
  - `u_long ntohl(u_long network_long)`

Esempio: conversione dell'indirizzo di porta (u_short) e IP (u_long)

```
addr.sin_port = htons((u_short)4321);
addr.sin_addr.s_addr = htonl(INADDR_ANY);
```

Funzioni per indirizzi IP

- **Conversione da rappresentazione con formato ASCII decimale e binario a 32 bit**
  - `int inet_aton(const char *strptr, struct in_addr *addrptr)`
  - `char *inet_ntoa(struct in_addr inaddr)`
- **Funzione di conversione hostname (nome simbolico) in indirizzo IP**
  - `struct hostent *gesthostbyname(const char *hostname)`
  - E' una chiamata al DNS
  - La struttura prodotta contiene numerose informazioni sull'host

Creazione di un socket

- **Prima di utilizzare un socket occorre crearlo**

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

  - **`family`**
    Individua la famiglia di protocolli da usare per il socket (es. PF_INET)
  - **`type`**
    Individua il tipo di protocollo (con/senza connessione)
  - **`protocol`**
    Individua il protocollo specifico (0 se la scelta è univoca)
- **La chiamata produce un socket descriptor che è utilizzato per riferirsi al socket nel sistema (-1 in caso di errore)**

Esempio di creazione

```
int sd;    /* socket descriptor */
struct  protoent *ptrp;  /* pointer to a protocol
  table entry */

  /* Map TCP transport protocol name to protocol number
   */
  if (((int)(ptrp = getprotobyname("tcp"))) == 0) {
    fprintf(stderr, "cannot map \"tcp\" to protocol
   number");
    exit(1);
  }

  /* Create a socket */
  if((sd = socket(PF_INET, SOCK_STREAM, ptrp-
   >p_proto))<0) {
    fprintf(stderr, "socket creation failed\n");
    exit(1);
  }
```

Collegamento socket server

- **Il socket creato deve essere collegato a un indirizzo locale (porta)**

```
int bind(int sockfd, struct sockaddr *localaddr,
                     int localaddrlen)
```

  - **`sockfd`**
    Socket descriptor di un socket creato in precedenza
  - **`localaddr`**
    Puntatore alla struttura dati che descrive l'indirizzo locale del socket
  - **`localaddrlen`**
    Dimensione dell'indirizzo socket locale (per gestire protocolli diversi)

- **La funzione ritorna 0 se ha successo, -1 in caso di errore**

Attivazione dell'ascolto

- **Dopo aver creato il socket e averlo collegato ad una porta si può attivare l'ascolto (socket di tipo SOCK_STREAM)**

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

  - **`sockfd`**
    E' il socket descriptor del socket creato in precedenza
  - **`backlog`**
    E' la dimensione della coda di attesa per le richieste di connessione

- **La funzione ritorna 0 in caso di successo, -1 in caso di errore**

Esempio di attivazione server

```
#define PROTOPORT  5193  /* default protocol port number */
#define QLEN  6   /* size of request queue        */

struct  sockaddr_in sad; /* server address  */

sad.sin_family = AF_INET;          /* set family to Internet
   */
sad.sin_addr.s_addr = INADDR_ANY; /* set the local IP address
   */
sad.sin_port = htons((u_short)PROTOPORT); /* set local TCP
   port */

/* Bind a local address to the socket */
if(bind(sd, (struct sockaddr *)&sad, sizeof(sad)) < 0) {
  fprintf(stderr,"bind failed\n");
  exit(1);
}

/* Specify size of request queue */
if (listen(sd, QLEN) < 0) {
  fprintf(stderr,"listen failed\n");
  exit(1);
}
```

Ricezione sul server

- **Attesa (bloccante) della prossima richiesta di connessione**

```
int accept(int sockfd, struct sockaddr *client_addr,
                       int *clientaddrlen)
```

  - **`sockfd`**
    Socket descriptor del socket creato in precedenza
  - **`client_addr`**
    puntatore ad una variabile di tipo struct sockaddr in cui viene memorizzato l'indirizzo del client che si è connesso
  - **`clientaddrlen`**
    Puntatore ad una variabile in cui viene scritta la dimensione dell'indirizzo del client (prima della chiamata deve contenere la dimensione effettiva)

- **Restituisce il nuovo socket descriptor da utilizzare per la comunicazione, -1 in caso di errore**

Esempio di ricezione (server)

```
struct sockaddr_in cad; /* client address  */
int sdc;                /* connection socket descriptor
   */
int alen;               /* length of address */

alen = sizeof(cad);

/* accept connections to a listening socket */
if ((sdc=accept(sd, (struct sockaddr *)&cad, &alen)) <
   0) {
  fprintf(stderr, "accept failed\n");
   exit(1);
}
/* sdc can be used to send/receive data from client */
```

Connessione client

- **Dopo aver creato un socket con la funzione socket, il client può connettersi ad un server con la funzione**

```
int connect(int sockfd, const struct sockaddr
        *serveraddr, int serveraddrlen)
```

  - **`sockfd`**
    descrittore del socket creato in precedenza
  - **`serveraddr`**
    puntatore ad una struttura sockaddr_in in cui è memorizzato l'indirizzo del server da contattare (IP e porta)
  - **`serveraddrlen`**
    lunghezza dell'indirizzo in byte (per gestire protocolli diversi)
- **Restituisce 0 in caso di successo, -1 in caso di errore**

Esempio di connessione (client)

```
#define PROTOPORT 5193 /* default protocol port number */
char localhost[] =  "localhost";  /* server hostname */
struct  hostent  *ptrh;  /* pointer to a host table entry */
struct  sockaddr_in sad; /* server TCP address */
int sd; /* socket descriptor */

sad.sin_family = AF_INET;  /* set family to Internet */
sad.sin_port = htons((u_short)PROTOPORT); /* set port number
   */

/* Convert host name to IP address and copy to sad */
ptrh = gethostbyname(host);
if (((char *)ptrh) == NULL ) {
  fprintf(stderr,"invalid host: %s\n", host); exit(1);
}
memcpy(&sad.sin_addr, ptrh->h_addr, ptrh->h_length);

if (connect(sd, (struct sockaddr *)&sad, sizeof(sad)) < 0) {
  fprintf(stderr,"connect failed\n"); exit(1);
} /* use sd to receive/send data */
```

Invio dati con connessione

- **Invio dati su una connessione aperta**

```
int send(int sockfd, const void *buf, int buflen, int
                                                flags)
  int write(int sockfd, const void *buf, int buflen)
```

  - **`sockfd`**
    Descrittore del socket ottenuto con la connessione
  - **`buf`**
    puntatore all'area di memoria che contiene i dati
  - **`buflen`**
    numero di byte da inviare
  - **`flags`**
    Opzioni di trasmissione (normalmente 0)
- **Ritorna il numero di byte inviati, -1 in caso di errore**

Ricezione dati con connessione

- **Ricezione dati su una connessione aperta**

```
int recv(int sockfd, const void *buf, int buflen, int
                                                flags)
  int read(int sockfd, const void *buf, int buflen)
```

  - **`sockfd`**
    Descrittore del socket ottenuto con la connessione
  - **`buf`**
    puntatore all'area di memoria in cui memorizzare i dati ricevuti
  - **`buflen`**
    dimensione del buffer (numero massimo di caratteri nel buffer)
  - **`flags`**
    Opzioni di ricezione (normalmente 0)
- **Ritorna il numero di byte ricevuti, -1 in caso di errore**

Chiusura di un socket

- **Il socket deve essere chiuso se non è più usato**
  - la chiusura libera le risorse allocate per la gestione del socket

```
int close(int sockfd)
```

  - **`sockfd`**
    descrittore del socket da chiudere

- **Restituisce 0 in caso di successo, -1 in caso di errore**
- **Dopo la chiusura il socket non è più disponibile**

Esempio di invio/ricezione

```
int     sd;  /* socket descriptor */
int     n; /* number of read characters */
int     rplylen; /* number of read characters */
char    buf[1000]; /* buffer for transmit data */

/* Read data from socket */
while(1) {
  n = recv(sd, buf, sizeof(buf), 0);
  if(n<0) break;
  rplylen = process(buf,n); /* process data and build reply in
   buf */
  send(sd,buf,rplylen,0);
}
/* Close the socket */
close(sd);
```

Invio per datagram

- **Se si utilizza il protocollo basato su datagram (UDP) non si apre la connessione (connect) ma si invia direttamente il pacchetto**

```
int sendto(int sockfd, const void *buf, int buflen,
   int flags, const struct sockaddr *toaddr, int
                        toaddrlen)
```

  - **`sockfd`** descrittore di socket ottenuto da socket(..)
  - **`buf`** buffer con i dati da trasmettere
  - **`buflen`** quantità in byte di dati da trasmettere
  - **`flags`** normalmente 0
  - **`toaddr`** indirizzo a cui inviare il pacchetto (IP+porta UDP)
  - **`toaddrlen`** dimensione dell'indirizzo (sizeof(struct sockaddr_in))
- **Restituisce il numero di byte inviati, -1 in caso di errore**

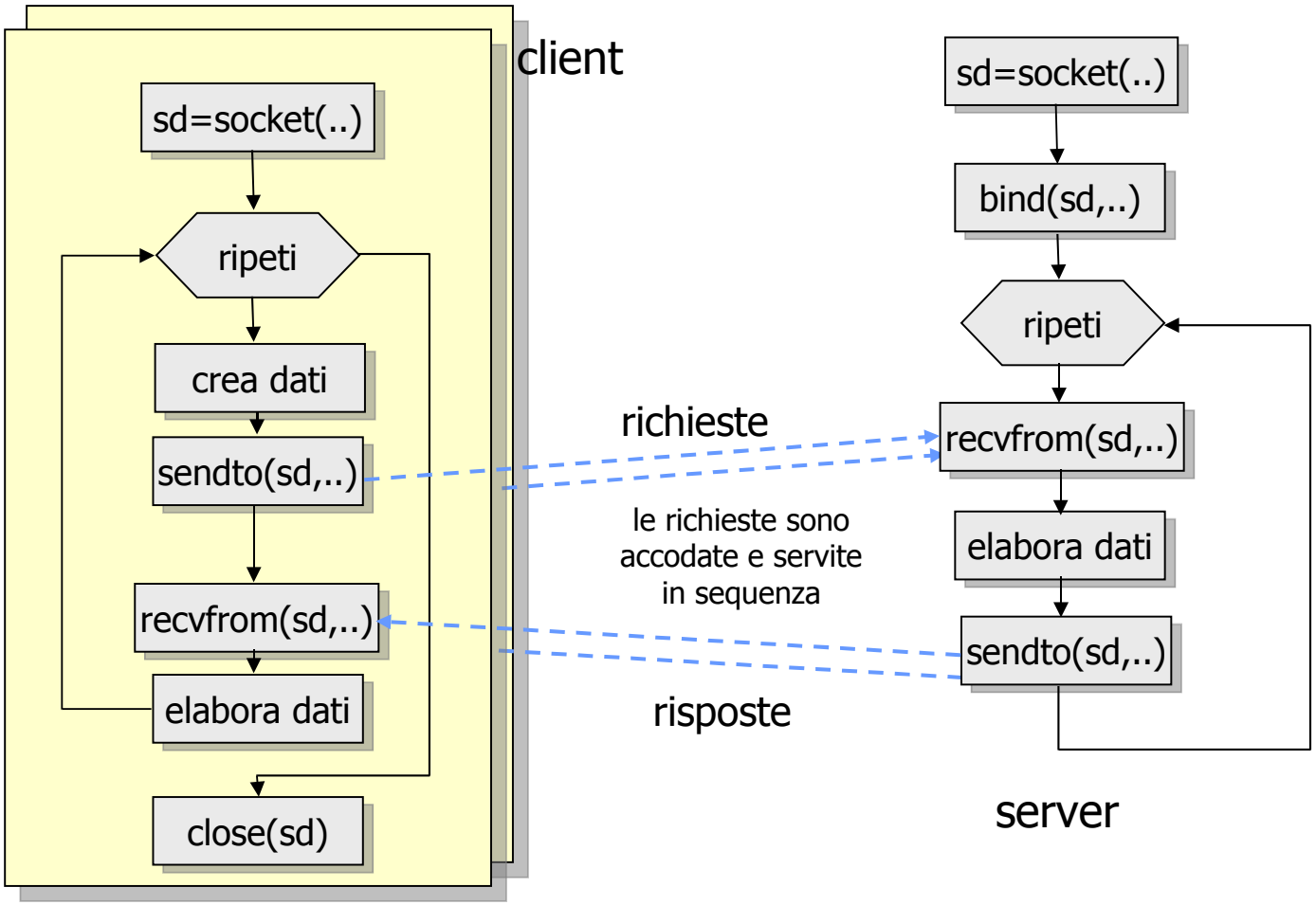
Ricezione per datagram

- **Se si utilizza il protocollo basato su datagram (UDP) non è necessario aprire la connessione in ascolto (listen)**

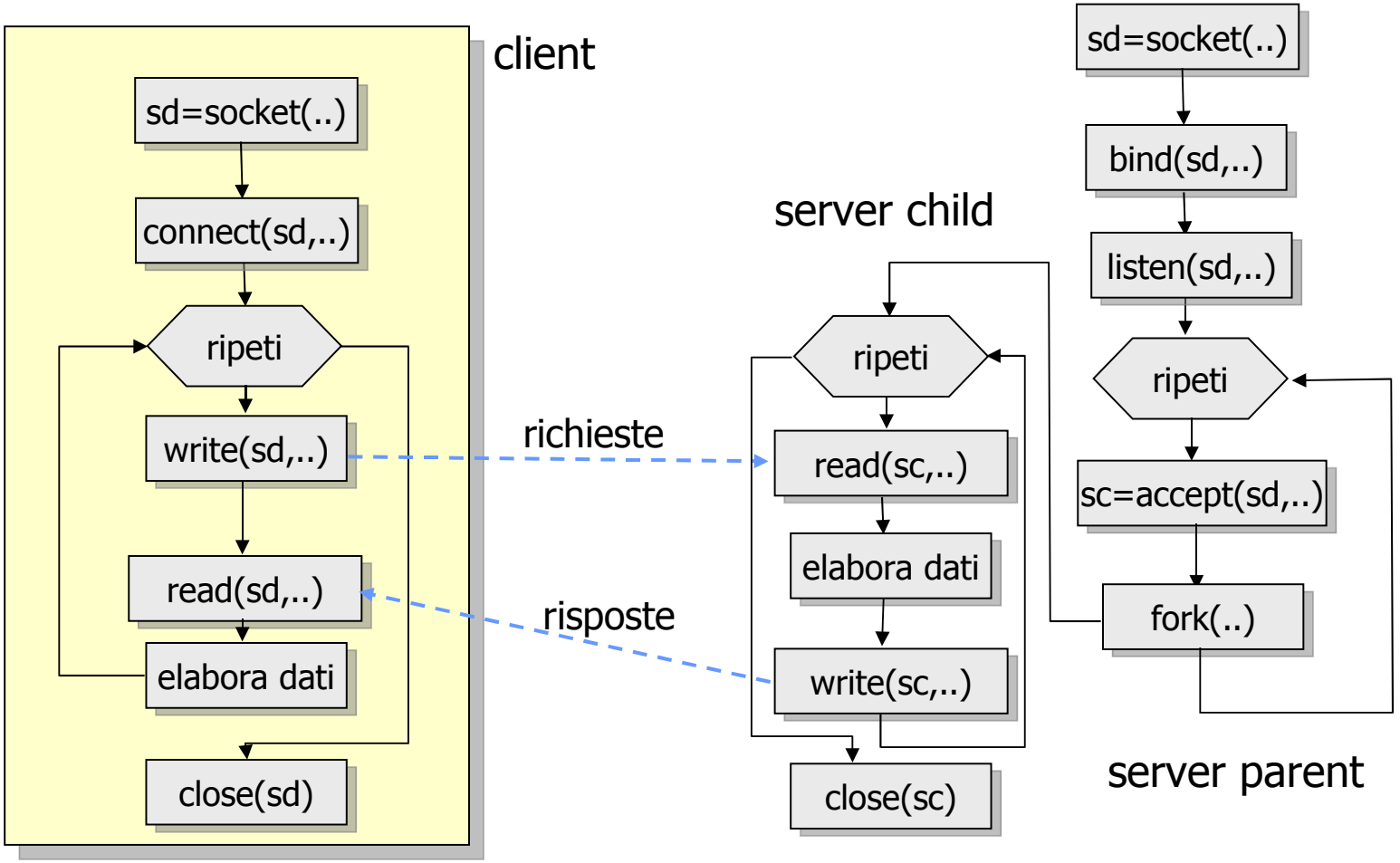
```
int recvfrom(int sockfd, const void *buf, int buflen,
int flags, struct sockaddr *fromaddr, int *fromaddrlen)
```

  - **`sockfd`** descrittore di socket ottenuto da socket(..)
  - **`buf`** buffer in cui memorizzare i dati ricevuti
  - **`buflen`** dimensione del buffer
  - **`flags`** normalmente 0
  - **`fromaddr`** puntatore alla struttura per l'indirizzo da cui proviene il pacchetto (IP+porta UDP)
  - **`fromaddrlen`** puntatore alla dimensione dell'indirizzo
- **Restituisce il numero di byte ricevuti, -1 in caso di errore**

Server sequenziale datagram



Server parallelo stream



Computer Networks

Socket Programming in 9 steps

Luigi Vetrano

1st step

- Set up a couple of variables to hold information on the IP address and port on which the socket server will run.
- You can set up your server to use any port in the numeric range 1-65535, so long as that port is not already in use.

```
<?php
// set some variables
  $host = "192.168.1.99";
  $port = 1234;
?>
```

2nd step

- Since this is a server, it's also a good idea to use the set_time_limit() function to ensure that PHP doesn't time out and die() while waiting for incoming client connections.

```
<?php
 // don't timeout!
    set_time_limit(0);
?>
```

3rd step

- **With the preliminaries out of the way, it's time to create a socket with the socket_create() function**
  - this function returns a socket handle that must be used in all subsequent function calls.

```
<?php
// create TCP socket
 $socket =
  socket_create(AF_INET,SOCK_STREAM,0)or
  die("Could not create socket\n");
?>
```

Note

- In case you're wondering what this is all about, don't worry too much about it. The AF_INET parameter specifies the domain, while the SOCK_STREAM parameter tells the function what type of socket to create (in this case, TCP).
  If you wanted to create a UDP socket, you could use the following line of code instead:

```
<?
// create UDP socket
 $socket =
  socket_create(AF_INET,SOCK_DGRAM, 0) or
  die("Could not create socket\n");
?>
```

4th step

- Once a socket handle has been created, the next step is to attach, or "bind", it to the specified address and port. This is accomplished via the socket_bind() function.

```
<?php
 // bind socket to port
$result = socket_bind($socket,$host, $port)
  or die("Could not bind to socket\n");
?>
```

5th step

- With the socket created and bound to a port, it's time to start listening for incoming connections. PHP allows you to set the socket up as a listener via its socket_listen() function, which also allows you to specify the number of queued connections to allow as a second parameter (3, in this example).

```
<?php
// start listening for connections
  $result = socket_listen($socket, 3) or
  die("Could not set up socket listener\n");
?>
```

6th step

- At this point, your server is basically doing nothing, waiting for incoming client connections. Once a client connection is received, the socket_accept() function springs into action, accepting the connection request and spawning another child socket to handle messaging between the client and the server.

```
<?php
// accept incoming connections
// spawn another socket to handle communication
   $spawn = socket_accept($socket) or die("Could
            not accept incoming connection\n");
?>
```

- This child socket will now be used for all subsequent communication between the client and server.

7th step

- **With a connection established, the server now waits for the client to send it some input**
  - this input is read via the socket_read() function, and assigned to the PHP variable $input.

```
<?php
 // read client input
$input = socket_read($spawn, 1024) or die("Could
   not read input\n");
?>
```

The second parameter to socket_read() specifies the number of bytes of input to read - you can use this to limit the size of the data stream read from the client.

Note that the socket_read() function continues to read data from the client until it encounters a carriage return (\n), a tab (\t) or a \0 character. This character is treated as the end-of-input character, and triggers the next line of the PHP script.

8th step

- **The server now must process the data sent by the client**
  - in this example, this processing merely involves reversing the input string and sending it back to the client. This is accomplished via the socket_write() function, which makes it possible to send a data stream back to the client via the communication socket.

```
<?php
// reverse client input and send back
$output = strrev($input) . "\n";
socket_write($spawn, $output, strlen($output))
  or die("Could not write output\n");
?>
```

  - The socket_write() function needs three parameters:
    - a reference to the socket,
    - the string to be written to it, and
    - the number of bytes to be written.

9th step

- Once the output has been sent back to the client, both generated sockets are terminated via the socket_close() function.

```
<?php
// close sockets
socket_close($spawn);
socket_close($socket);
?>
```

- And that's it - socket creation, in nine easy steps!

How about seeing it in action?

Since this script generates an "always-on" socket, it isn't a good idea to run it via your Web server; instead, you might prefer to run it from the command line via the PHP binary:

```
$ /usr/local/bin/php -q server.php
```

Note the additional -q parameter to PHP - this tells the program to suppress the "Content-Type: text/html" header that it usually adds when executing a script (I don't need this header here because the output of this script isn't going to a browser).

Once the script has been executed and the socket server is active, you can simply telnet to it using any standard telnet application, and send it a string of characters as input. The server should respond with the reversed string, and then terminate the connection.

Here's what it looks like:

```
$ telnet 192.168.1.99 1234
Trying 192.168.1.99...
Connected to medusa.
Escape character is '^]'.
Dado e Cice
eciC e odaD
Connection closed by foreign host.
```