The Socket Interface

- Client and server use the transport protocol to communicate.
- When it interacts with protocol, an application must specify :whether it is a server or a client (that is, wether it will wait passively or actively initiate communication)
- In addition, the sender must specify the data to be sent, and the receiver must specify where incoming data should be placed

- The interface used by an application is known as an *Application Program Interface* (API).
- An API defines **a set of operations** that an application can perform when interacting with protocol software and details such as arguments required.
- Usually an API contains a separate procedure for each logical function

- The protocols specify the general operations that should be provided, and **allow each O.S. to define the specific API** an application use to perform the operation.

- **The socket API is a standard.** They are available for many O.S. (e.g. Microsoft’s windows system, various Unix systems).

- **The socket API** originated as part of the BSD Unix O.S.

Socket and Socket Library

- In BSD UNIX and in the systems derived from it, socket functions are **part of the O.S. itself**.

- In different O.S., instead to modifying their basic O.S. vendors created a **socket library** that provides the socket API. That is, the vendor created a library of procedures that each have the same name and arguments as one of the socket functions.

- A **socket library** can provide applications with a socket API on a computer system that does **not provide native sockets**. When an application calls one of the socket procedures the control passes to a library routine that makes one or more calls to the underlyng operating system to implement the socket function.

Socket properties

• **Communication domain**.
PF-INET :internet domain
PF-UNIX :Unix O.S. domain

• **Semantic characteristics of the communication.**
reliability, one to one communication, one to many communication .

• **Local and remote addresses representation**
A generic format is used to represent the addresses to be assigned to the sockets

Socket type

Type: indicates the communication properties.

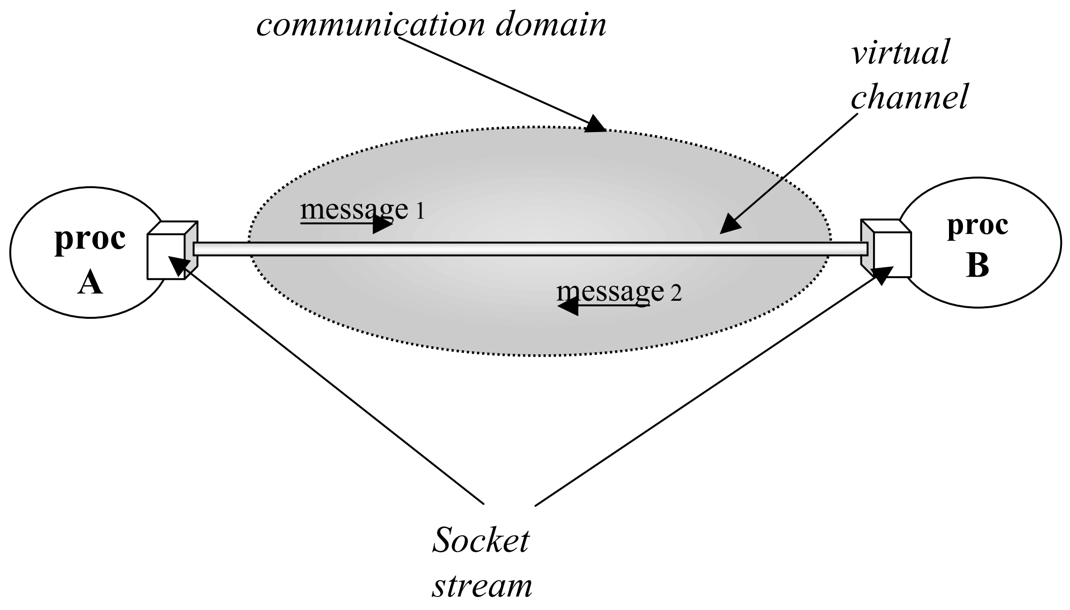
connection-oriented (virtual channel utilization)
connection-less (no virtual channel)

Socket stream

- *Connection oriented*
- *one-to-one simmetric communication*
- *connection creation before the beginning of communication*
- *connection termination at the end of communication*

Socket datagram

- *Connectionless*
- *one-to- many asimmetric communication*


communication domain
virtual channel
proc A
proc B
message 1
message 2
Socket stream

Data structure associated to a socket

• In the **Internet domain** (PF-INET) to each socket is assigned an **address** constituded by:

> **the IP address** of the node in which is running the process owner of the socket.
> **the number of the port** to which the socket is associated .

• in order to communicate with others processes each process must:

> **to create** a socket. Each socket is locally represented by a *file descriptor*.
> **to specify the socket address** at wich the server will accept

contacts

In the case of PF-INET domain the **socket address** is represented by the following structure :

```
struct sockaddr-in {
sa-family-t         sin-family;          /*family of the address*/
in-port-t           sin-port;            /*port number*/
struct-in-addr      sin-addr;            /*IP address of the node*/
char                sin.zero [8]         /*not used (set to zero)*/
}
```

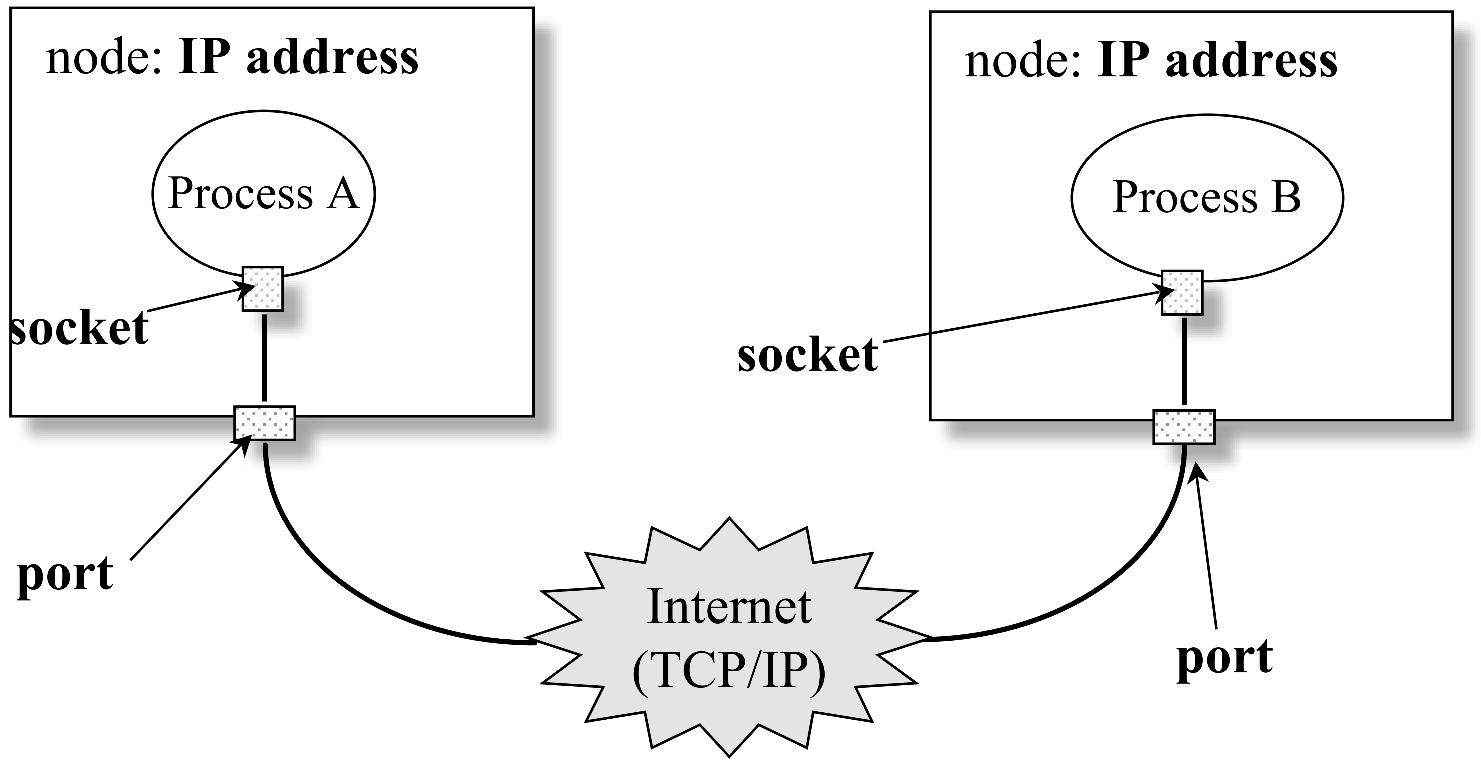
where the type of sin_addr, which represents the node address, is represented in the following way:

```
struct in_addr { uint32_t  s_addr};
```

The non primitive data types used in the previous definitions are described in the following. Are also indicated the header files in which they are declared.

Type	**Description**	**Header file**
sa_family_t	Type associated to the domain	<sys/types.h>
in_port_t	Type associeted to the port (16 bit unsigned int)	<netinet/in.h>
uint32_t	32 bit unsigned int	<sys/types.h>
sockaddr	Type associated to the socket address	<sys/socket.h>
sockaddr_in	Specific type associated to a socket address in the internet domain (IP v.4)	<sys/socket.h>

Socket



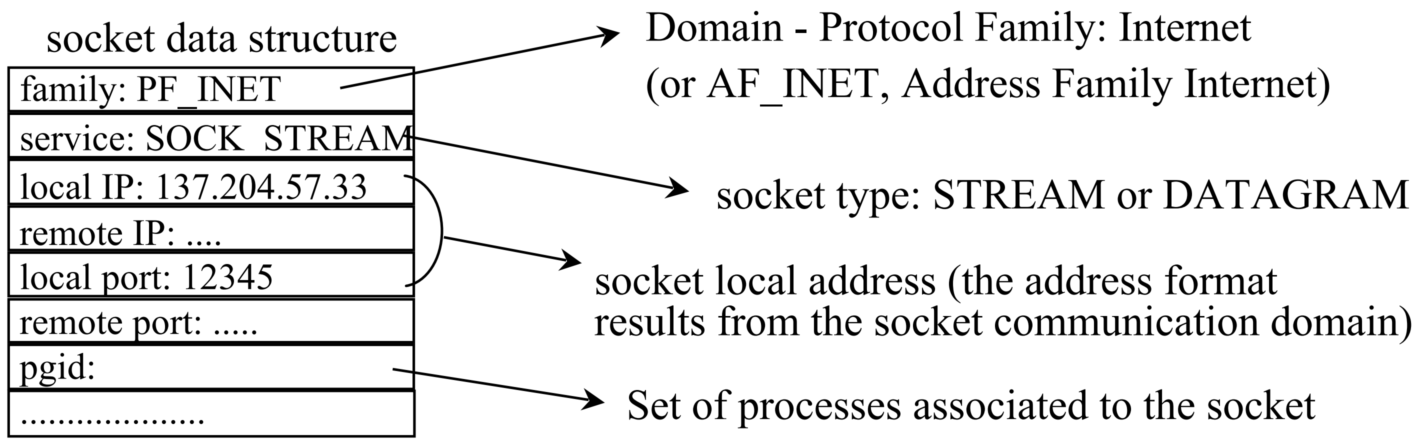
The **communication channel** between the process A and the process B is defined by

*<**protocol**; **IP local address**; **local port**; **IP remote address**; **remote port** >*

Socket data structure

A socket is created into a communication domain(Internet or Unix domains)

.



socket data structure
family: PF_INET
service: SOCK_STREAM
local IP: 137.204.57.33
remote IP: ....
local port: 12345
remote port: .....
pgid:
....
Domain - Protocol Family: Internet
(or AF_INET, Address Family Internet)
socket type: STREAM or DATAGRAM
socket local address (the address format results from the socket communication domain)
Set of processes associated to the socket

Address format

UNIX domain : the address format is the format of a file name (pathname).

AF_INET domain:Internet address
IP address of the node (32 bit); port number (16 bit)

socket creation

sd = **socket** (domain, type, protocol);

The *socket procedure* creates a socket and returns an integer descriptor

domain specifies the communication domain (es. AF_INET)

type specifies the type of communication the socket will use (es. SOCK_STREAM o SOCK_DGRAM)

protocol specifies a particular transport protocol used with the socket

In the communication channel the **procedure specifies the used protocol**
<***protocol***; *IP local address; local port; IP remote address; remote port*
> ⇧

bind procedure

When created, a socket has neither a local address nor a remote address

error = **bind** (sd, localaddr, addrlen);

A process uses the the **bind** procedure to supply the local address at wich the process will wait for contacts.

sd is the descriptor of the socket that has been created, but not previously bound.
localaddr is a structure that specifies the local address to be assigned to the socket.
addrlen is an integer tha specifies the length of the address.

In the communication channel the **procedure specifies** :

*<protocol; **IP local address**; **local port*** ; *IP remote address; remote port>*

connection-oriented communication
(socket STREAM or TCP)

client and server: **asimmetric communication**

1) server and client must creat **creare each one a socket** and specify their address (socket and bind procedures)

2) a virtual channel must be created betweeen the two sockets

3) **communication**

4) socket shutdown

listen procedure
(Server)

After creation the socket is placed **in passive mode** so it can be used to wait for contacts from clients.

```
error  =  listen (sd ,  dim);
int  error ,  sd , dim;
```

.

sd is the descriptor of a socket that has been created and bound to a local address.

dim specifies a length for the socket’s request queue.

The O.S.builds a separate **request queue** for each socket. Initially the queue is empty.

As requests arrive from clients, **each is placed in the queue**; when the server asks to retrieve an incoming request from the socket, the system returns the next request from the queue.

If the queue is full when a request arrives, the system rejects the request.

connect procedure
(Client)

Clients use **procedure connect** to establish connection with a specific server

```
error  =  connect (sd,  saddress,  saddresslen) ;
```

- **sd** is the descriptor of a socket on the client computer to use for the connection.
- **saddress** is a sockaddress structure that specifies the s**erver address and port number**
- **saddresslen** specifies the length of the server's address measured in bytes
- the client process may be suspended as a consequence of a call of **connect**

In the communication channel **the procedure specifies** :
<protocol; IP local address; local port ; ***IP remote address; remote port>***

⇧ ⇧

accept procedure
(Server)

A server that use a connection-oriented transport must call procedure **accept** to accept **a connection request**

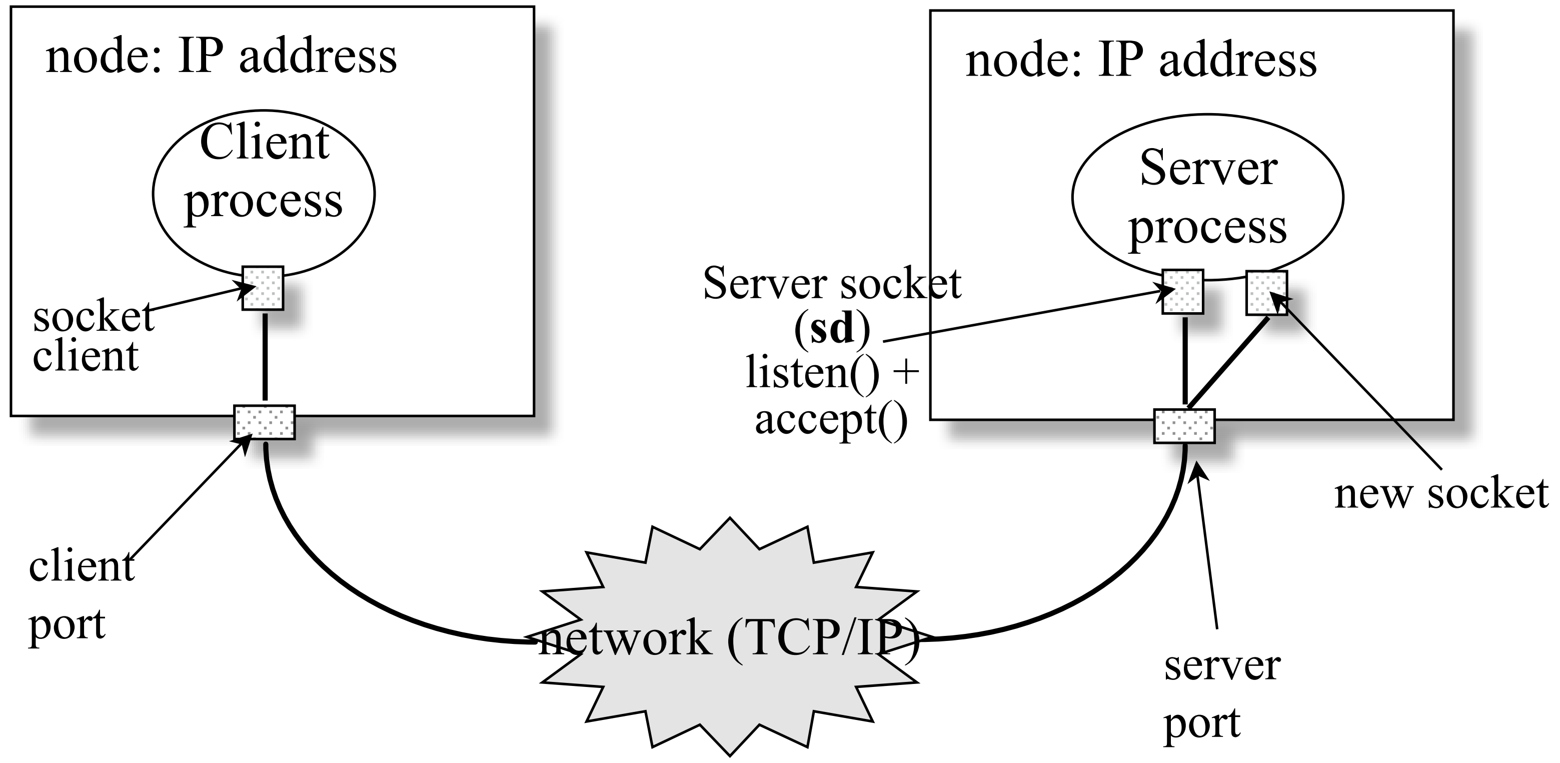
int = accept (sd , caddress, caddreslen);

sd is the descriptor of a socket the server has created and bound to a specific port.
caddress is the address of a structure of type sockaddress**.**
caddresslen is a pointer to an integer.

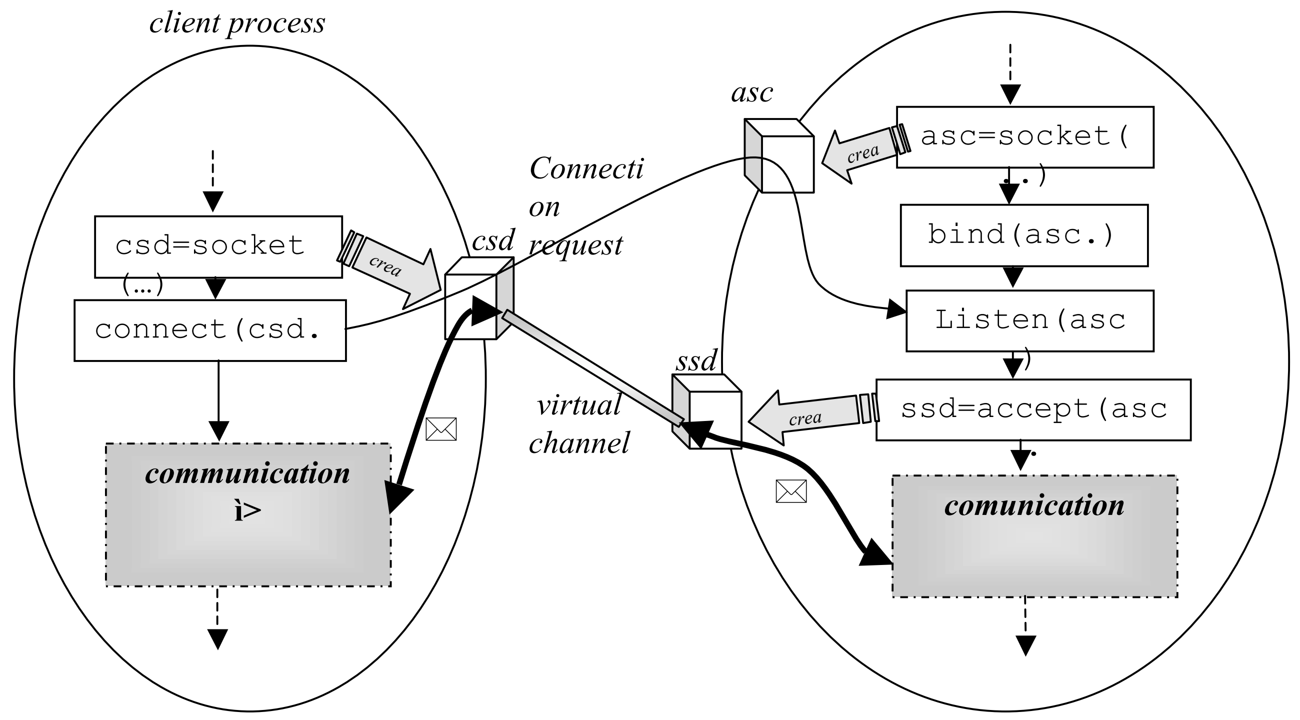
-If a request is present in the queue, *accept* estracts from it the first request and creates a new socket for the connection and returns the descriptor of the new socket to the caller

- If the queue is empty the server process is suspended.

connection oriented communication



virtual channel creation



Read and write with sockets

Socket API was originally designed to be part of UNIX, which uses read and write fot I/O. Consequently, sockets also allow applications to use **read** and **write** to transfer data.

Read and write do not have arguments tha permit the caller **to specify** a **destination**. (one to one communication schema). It is sufficient to specify only the socket descriptor associated to the local socket.

Read and write have **three arguments**: a socket descriptor, the location of a buffer in memory used to store data and the length of the memory buffer.
A server can send a message to a client in the following way:

```
int asc, sd;
char msg[6] = “Ciao!”;
<creazione socket ed apertura del canale>
write (sd,msg,6)
```

Client process

```
int csd;
char msg[6];
< socket creation and channel definition>;
read (ssd,msg,6);
```

The sockets are blocking: if the read is executed when no messages are present in the channel,the process is suspended.

In TCP protocol messages are not separated. The channel contents is consided as a non structured sequence of bytes.

Agreement between the two processes on the characteristics of the messages (for ins., prefixed constant lenght).

Close procedure

The *close* procedure tells the system to terminate the use of a socket. It has the form:

close(socket)

. Closing the socket immediately terminates its use . The descriptor is released, preventing the process from sending and receiving more data.

If the socket is using a connection oriented transport protocol, *close* terminates the connection before closing the socket.

• At the end of a communication session, the connection can be closed using the procedure:

shutdown (socket, mode)

sd is the socket descripror associated to the channel terminal.
mode defines the closure modalities.

• It is possible to close the channel **only in a direction** (value 0 for the reception, value 1 for the trasmission) or in both directions (value 2).

• If both directions are closed the sd socket is deleted.

Client process

```
# include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
struct sockaddr_in *D, *server;
char msg[2000];
int sd, l;
int main()
{
            sd=socket(AF_INET,SOCK_STREAM,0);
            <server address initialization>
            /* establish connection*/
            connect(sd,&server,l);

<msg message creation>;

            write(sd, msg,2000); /* message sending*/
            read(sd,ris, 2000);  /* answer receiving */
            shutdown (sd,2);    /* connection closing*/
}
```

Server process

```
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
struct sockaddr_in *M, *my;
char msg[BUFFERSIZE], ris[2000];
int asc, l,sd, addrlen ;
int main()
{
        asc=socket(AF_INET,SOCK_STREAM,0);
        <address initialitation>
        l=sizeof(struct sockaddr_in);
        bind (asc,&my,l); /* address  assigning*/
        listen(asc, 100);  /*request queue creating */
        sd=accept(asc, M, &addrlen); /* channel opening*/
        read (sd, msg, 2000); /* message receiving*.
        < creation of  ris answer>
        write (sd, ris, 2000); /* answer sending*/ }
```

Socket datagram

As the communication is **connection-less,** the procedures called by a process are **create and bind.**

Procedures **sendto** e **recvfrom** are used by processes to send and receive messages**.**

sendto(sd, data, length, flags,destaddress, addresslen**)**

sd is the descriptor of a socket to use
data is the address in memory of the data to be send;
length is an integer that specifies the number of bytes of data
flags contains bits that request special options about the message transport;
destaddress specifies the address of a destination;
addresslength is the length of the address

recvfrom(sd, **buffer,length,flags**,sndraddr,saddrln)

sd is the descriptor **of a socket from wich data is to be received**
buffer is the address in memory in which the incoming message should be placed;
length specifies the size of the buffer;
flags allow the caller to control details about the message transport.
sndraddr and **saddrln** are used to record the sender’s address

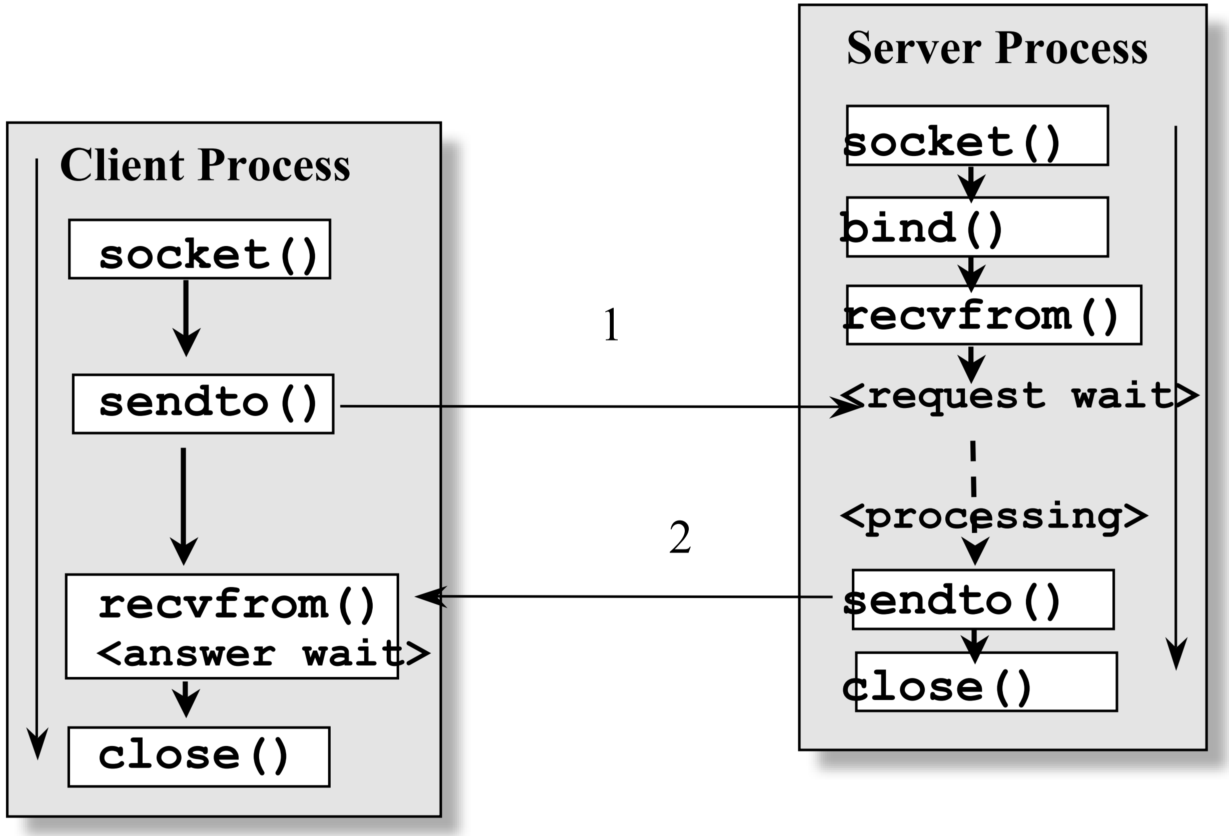
• The **recvfrom** can block the process, if the message is not available;

• Differently from the case of connection oriented communication the communication procedures require specific parameters to identify the addresses of the partnes in the communication.

• to send a message it is necessary to specify the address of the receiver; to receive a message it is necessary that the **recvfrom** provides the address of the sending process.

• to close a datagram socket it is possible to utilize the system call **shutdown**, or the system call **close**

Connection-less communication
(socket DATAGRAM)



Client

```
include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
struct sockaddr_in *D, *my;
char msg[2000], ris[BUFFERSIZE];
int sd, l, addrlen;
main()
{
      sd=socket(AF_INET,SOCK_DGRAM,0);
      <my address initialization>
      l=sizeof(struct sockaddr_in);
      bind (sd,&my,l);
      /* message sending to the server : */
      sendto (sd, msg, 2000, 0, D,l);
      /* answer receiving
      recvfrom (sd, ris, BUFFERSIZE,0, D, &addrlen);
      ...
      close(sd);
}
```

server

```
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
struct sockaddr_in *M, *my;
char msg[BUFFERSIZE], ris[2000];
int sd, l, addrlen;
main() {
        sd=socket(AF_INET,SOCK_DGRAM,0);
        < my address initialization >
        l=sizeof(struct sockaddr_in);
        bind (sd,&mio,l);
        addrlen =l;
        /* message receiving:*/
        recvfrom (sd, msg, BUFFERSIZE,0, M, &addrlen);
        < answer ris evaluation>
        /*answer sending:/
        sendto (sd, ris, 2000, 0, M, addrlen);
         ...
        close(sd); }
```

Socket properties

• STREAM sockets require a connection
DATAGRAM sockets are connectionless

$\longleftrightarrow$

• **Reliability problems** : STREAM sockets are based on TCP and the are reliable. DATAGRAM sockets are based on UDP and then they are not reliable.

• **Performance**: STREAM sockets are more expensive with reference to the DATAGRAM sockets.

Quale tipo di Socket? Quale livello di trasporto, UDP o TCP?

Connection oriented:

•reliability is fundamental
• a correct sequency of messages is important
•at-most-once semantic

:

Connectionless:
• broadcast/multicast
• performance is relevant
• there are not sequency of messages problems
• may-be semantic

Example: remote execution of comands

• Remote execution of simple comands sent by a client to a server and local display of the output of the executed comands.

•The client must require the creation of the connection and then send a message including the comand

•The clien waits the answer, constituted by the output of the executed comand
Each byte received is represented on the standard output

client process :

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
main(int argc, char **argv)
{
        int  sock, retval, i;
        char mess[10], ris[1000];
        struct sockaddr_in rem_ind; /* remote socket address */
        /* address server  */
        rem_ind.sin_family = PF_INET; /*domain*/
        /* for example: if the node internet address is
        137.204.57.115:*/
        rem_ind.sin_addr.s_addr=inet_addr("137.204.57.115");
rem_ind.sin_port = 22375; /* server port  number */
/* message creation*/
        strcpy(mess, argv[1]);
        /* socket creation */
        sock=socket(PF_INET, SOCK_STREAM, 0);
        connect(sock, &rem_ind, sizeof(struct sockaddr_in));
        write(sock,mess, 10); /* message sending*/
        while (i=read(sock,ris, 1)>0) /*response receiving */
                write(1,ris,i);         /* writing on  std. output */
        shutdown(sock,2);           /* closing connection*/
}
```

Processo server

• Each server process has typically a cyclic structure. A particular connection request is managed for each cycle.

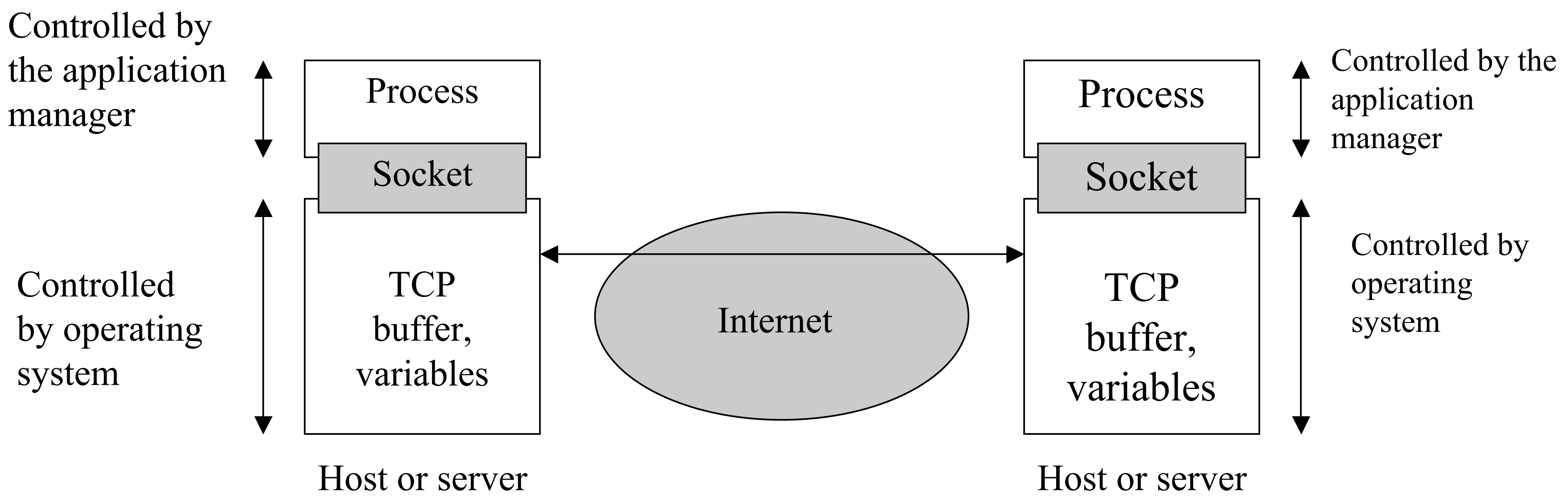
• The server process may manage the requests following a sequential or concurrent modality. In the example the server is sequentially managed.

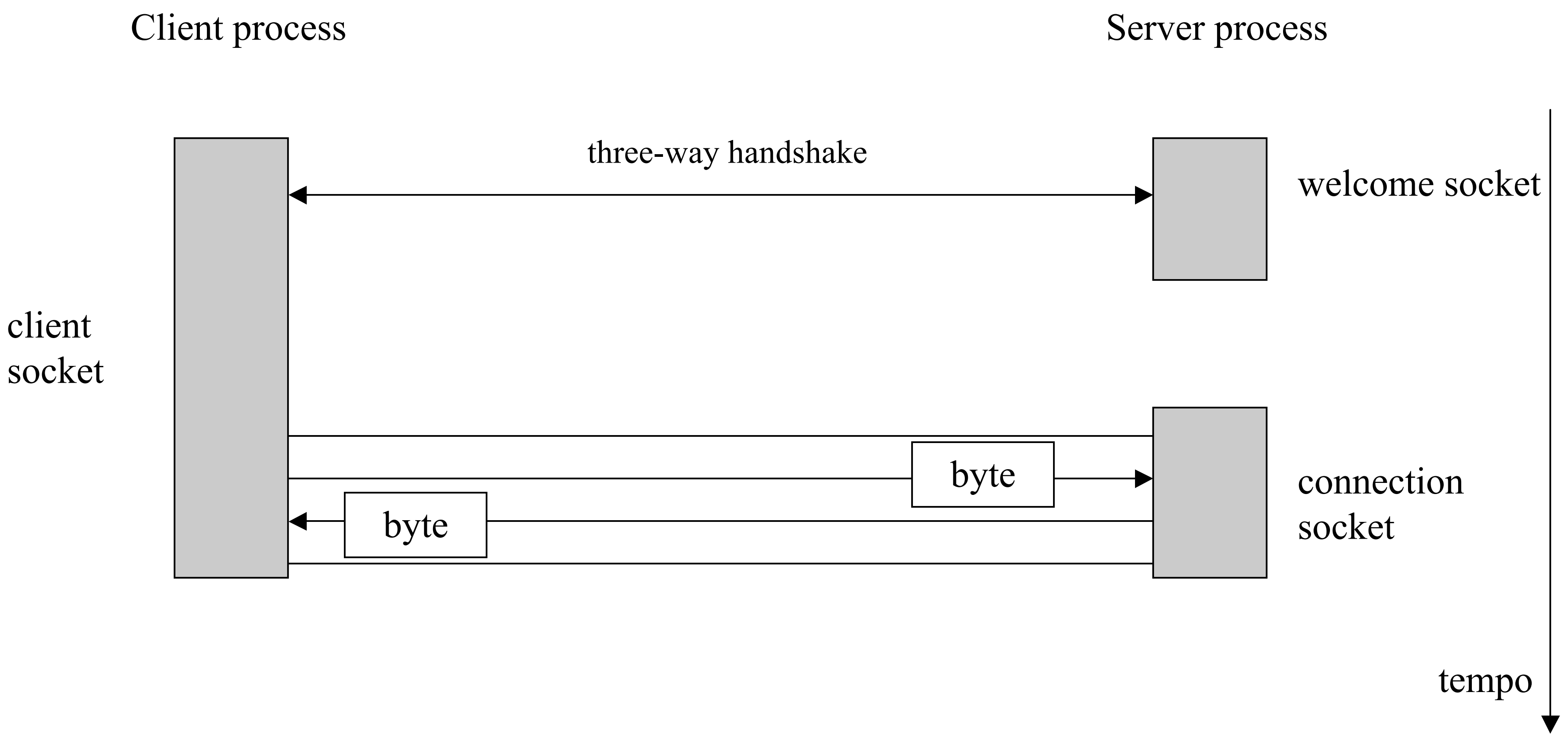
Before to begin the service, the server creates the socket that will receive the client requests. The *bind procedure* is called in order to supply the local address at wich the process will wait for contacts.
The listen procedure is called to associate to the socket a queue in which the connection requests will be inserted.

• For each request, after the procedure **accept** has been called, a new process will be create . The process will receive from the channel the name of the service to execute and the communication socket will be redirected on the standard output. Then, the comand will be executed by a system call of the *exec* family. At the end of the comand execution the server process will close the socket and will begin the execution of new requests.

```
Processo server:*/
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
main()
{
            char comand [20];
            int  newsock, sock,   son, status, i;
            struct sockaddr_in my_ind;
            /* listening socket creation */
            sock=socket(PF_INET, SOCK_STREAM, 0);
            /* socket address : */
            my_ind.sin_family=PF_INET;
            my_ind.sin_addr.s_addr = INADDR_ANY;
            my_ind.sin_port=22375;
```

```
/* Binding:*/
          if(bind(sock, &my_ind, sizeof(struct sockaddr_in))<0)
          {
                    perror("bind");
                    exit(1);
          }
          /* sock will receive the connection requests */
   listen(sock, 5); /*connection requests creation*/
   for (;;)/* service cycle */
   {      /* extraction of a new request from the queue
newsock=accept(sock,(struct sockaddr_in *) 0, 0);
          if ((son=fork())==0)
          {  /* son */
                    close(sock);
          read(newsock, comand, 10); /* comand receiving*/
         /* output ridirection output on the socket  */
                              close(1);
         dup(newsock);
/* comand esecution */
            if((i=execlp(comand, comand, (char )0))<0)
        {   write(1,"error", 7);
              exit(-1);
            }
         } /* son*
         else /*father*/
         {              wait(&status); /* waiting son  */
                        shutdown(newsock, 1); /* closing socket: */
              close(newsock);
         }
    }
    close(sock);
}
```


Controlled by the application manager
Process
Socket
Controlled by operating system
TCP buffer, variables
Host or server
Internet
Process
Socket
TCP buffer, variables
Host or server
Controlled by the application manager
Controlled by operating system


Client process
Server process
three-way handshake
welcome socket
client
socket
byte
byte
connection
socket
tempo