Unit 6
Communications with sockets

Computer Architecture Area (ARCOS)

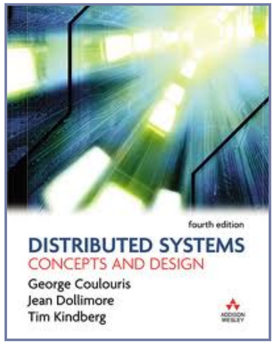
Distributed Systems

Bachelor in Informatics Engineering

Universidad Carlos III de Madrid

Objetives and Bibliography

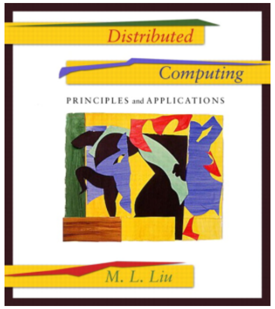
- Objetives:
  - Detailly explore the most used communication system among distributed processes.
- Basic Bibliography:



- Distributed Systems, Concepts and Design.
  G. Coulouris, J. Dollimore, T. Kindberg.

  Fourth edition, 2005.

  Addison-Wesley



- Distributed Computing: Principles and Applications
  M. L. Liu.
  2004
  Addison-Wesley

Contents

- Basic concepts about sockets
- API:
  - Sockets in C (POSIX)
  - Sockets in Java
- Concurrent servers
- Client-server applications design guide

Sockets: introduction

- Appeared in 1981 in UNIX BSD 4.2
  - Attempt to include TCP/IP in UNIX
  - Design independent from communication protocol
- A socket is an end point of a communication (IP address and port)
- Abstraction that:
  - Offers network services access interface in transport level
    - Protocol TCP
    - Protocol UDP
  - Represents a side of a bidirectional communication with an associated address

Sockets: introduction

- Under standardization processes inside POSIX (POSIX 1003.1g)
- Currently
  - Available in almost all UNIX systems
  - Available in a lot of Operating Systems
    - WinSock: Windows sockets API
  - Available in Java as a native class

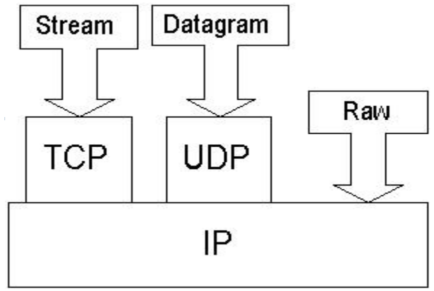
UNIX Sockets

- Communication domains
- Kinds of sockets
- Socket addresses
- Socket creation
- Addresses assignment
- Connection request
- Prepare to accept connections
- Accept connections
- Data transfer

Communication Domains

- A **domain** represents a protocol family
- A socket is associated to a domain from its creation
- Only sockets of the same domain can communicate
- Some examples:
  - PF_UNIX (or PF_LOCAL): communication inside a host
  - PF_INET: communication using TCP/IP protocols
- Socket services are independent from the domain

Kinds of sockets

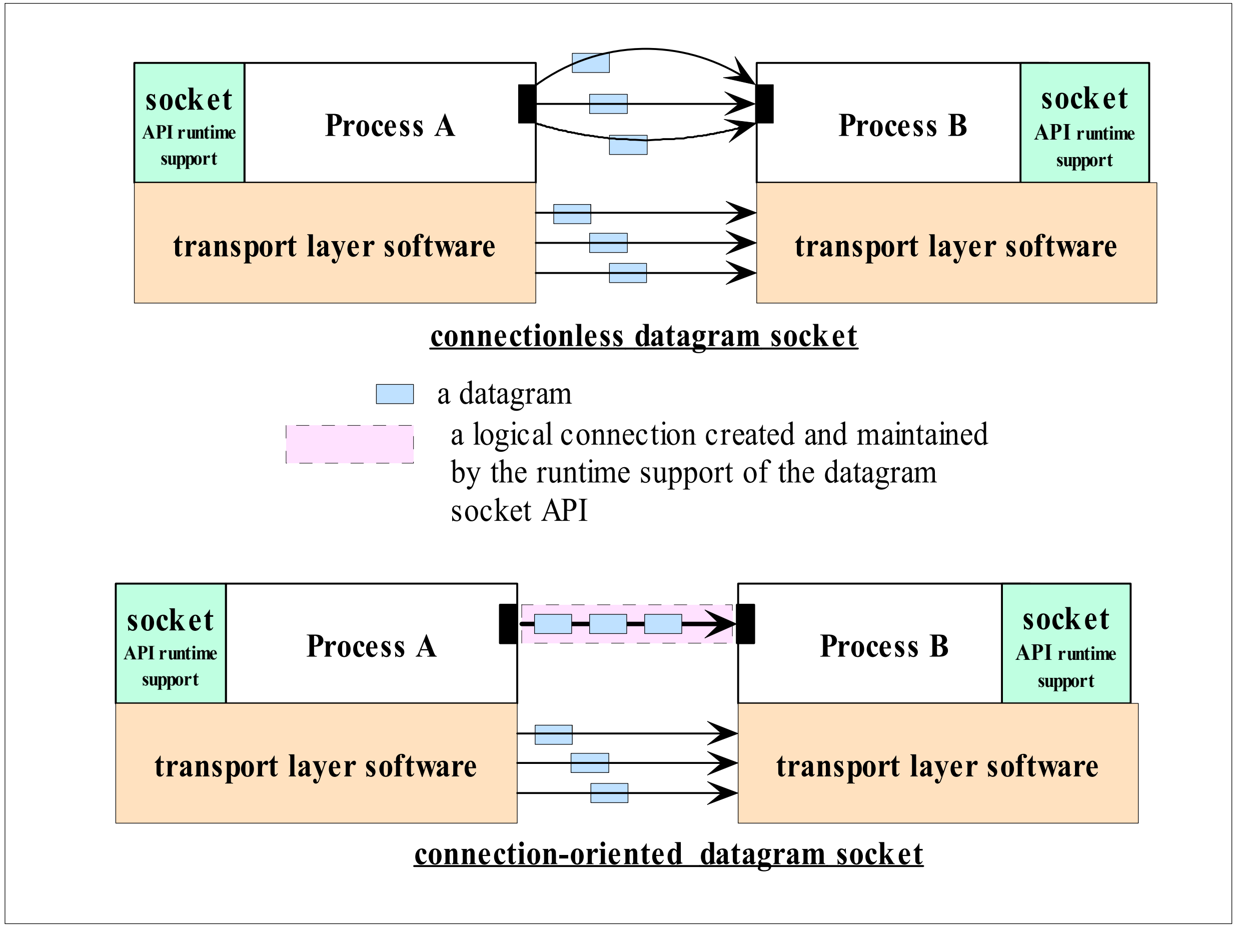


- ***Stream*** (SOCK_STREAM)
  - Connection oriented
  - Reliable, order delivery is assured
  - Do not mantain separation between messages
  - If PF_INET then it maps to TCP protocol
- ***Datagram*** (SOCK_DGRAM)
  - Connectionless
  - Not reliable, order delivery is not assured
  - Mantain separation between messages
  - If PF_INET then it maps to UDP protocol
- ***Raw*** (SOCK_RAW)
  - Protocol-less sockets

Socket addresses

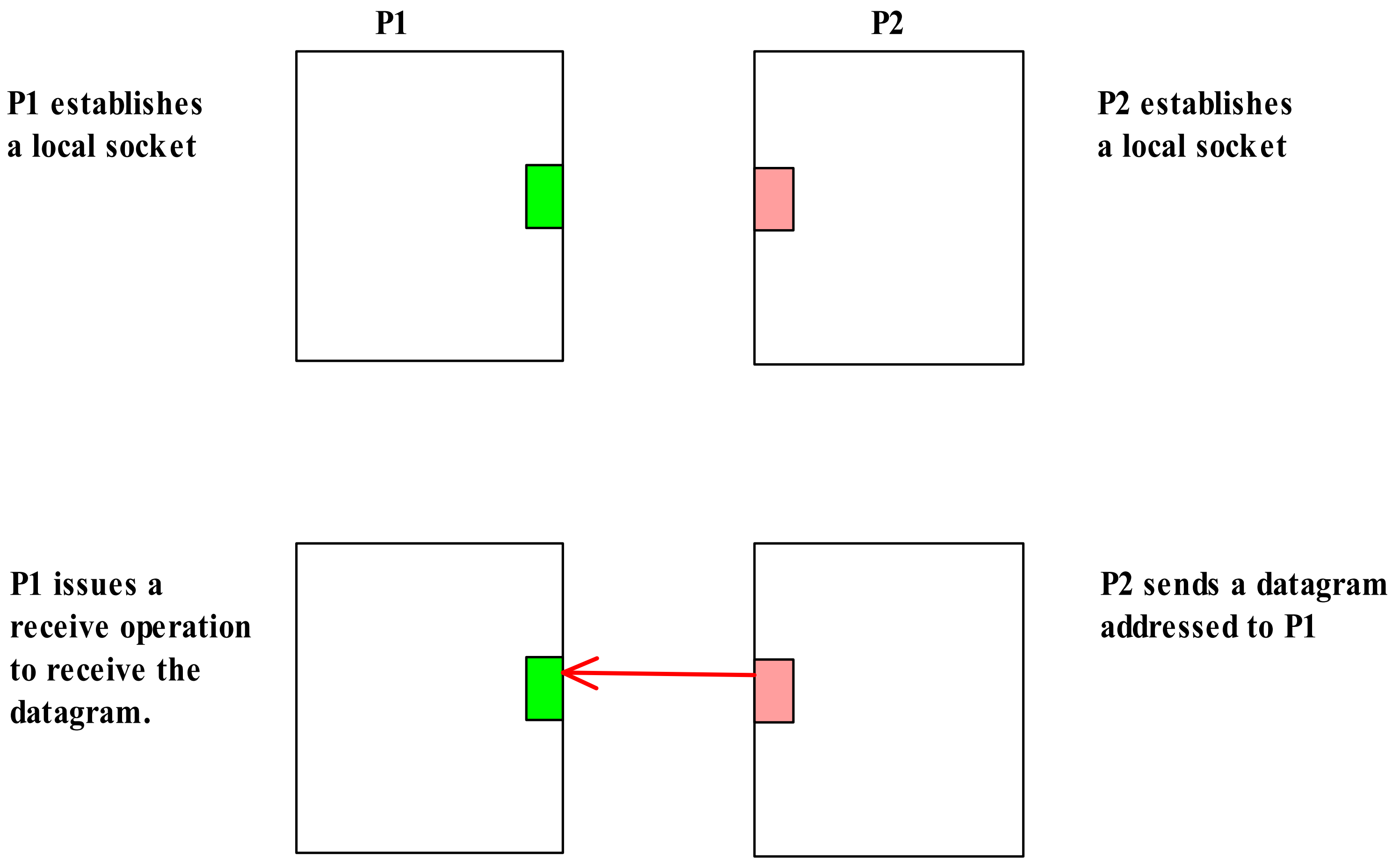
- Each socket must have a unique address assigned
  - Host (32 bits) + port (16 bits) + protocol
- Addresses are used to:
  - Assign a local address to a socket (*bind*)
  - Specify a remote address (*connect* or *sendto*)
- Domain dependents
- The generic structure *struct sockaddr* is used
- Each domain uses a specific estructure
  - Addresses in PF_UNIX (*struct sockaddr_un*)
    - File name
  - Addresses in PF_INET (*struct sockaddr_in*)
  - Use of type conversion (*casting)* in calls

Sockets: connection oriented and datagrams



© *Addison-Wesley*

Operation of a connectionless oriented service



P1 establishes a local socket

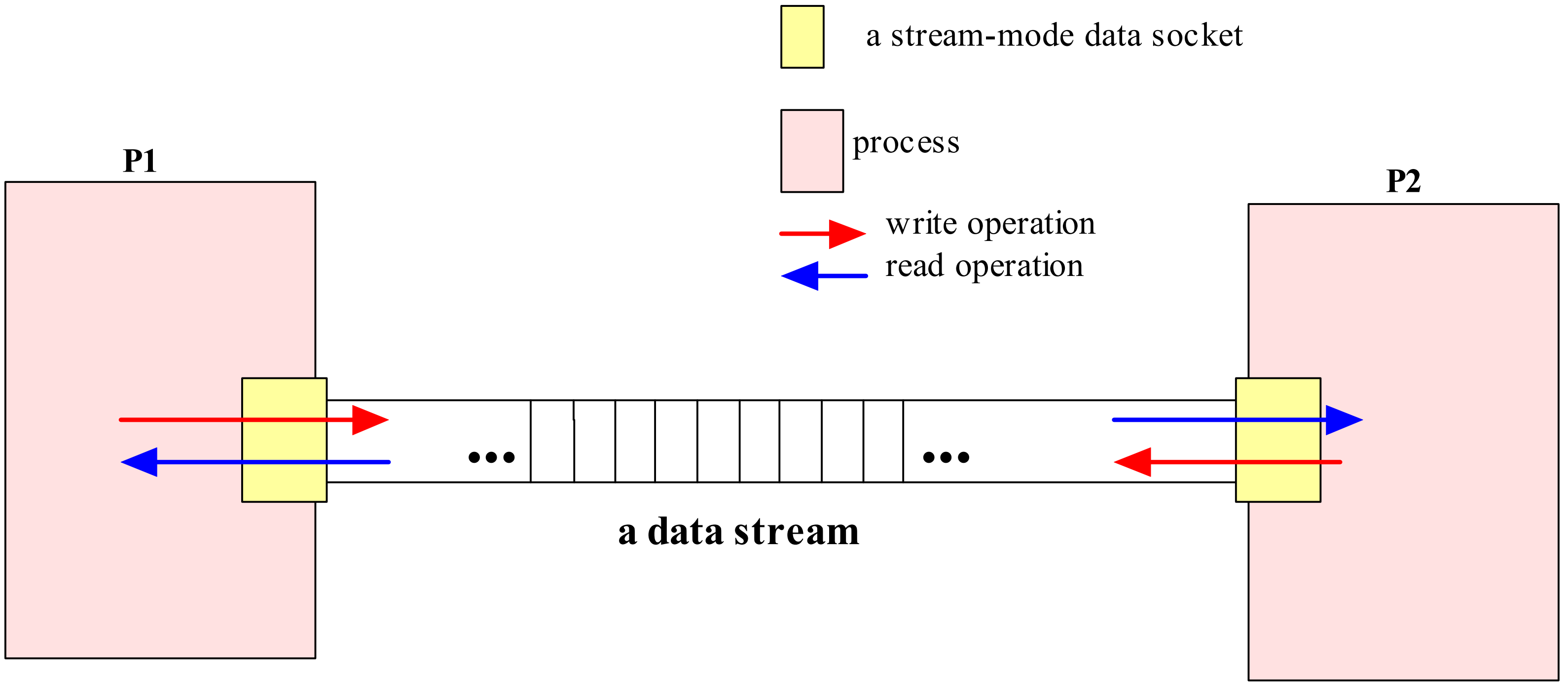
P2 establishes a local socket

P1 issues a receive operation to receive the datagram.

P2 sends a datagram addressed to P1

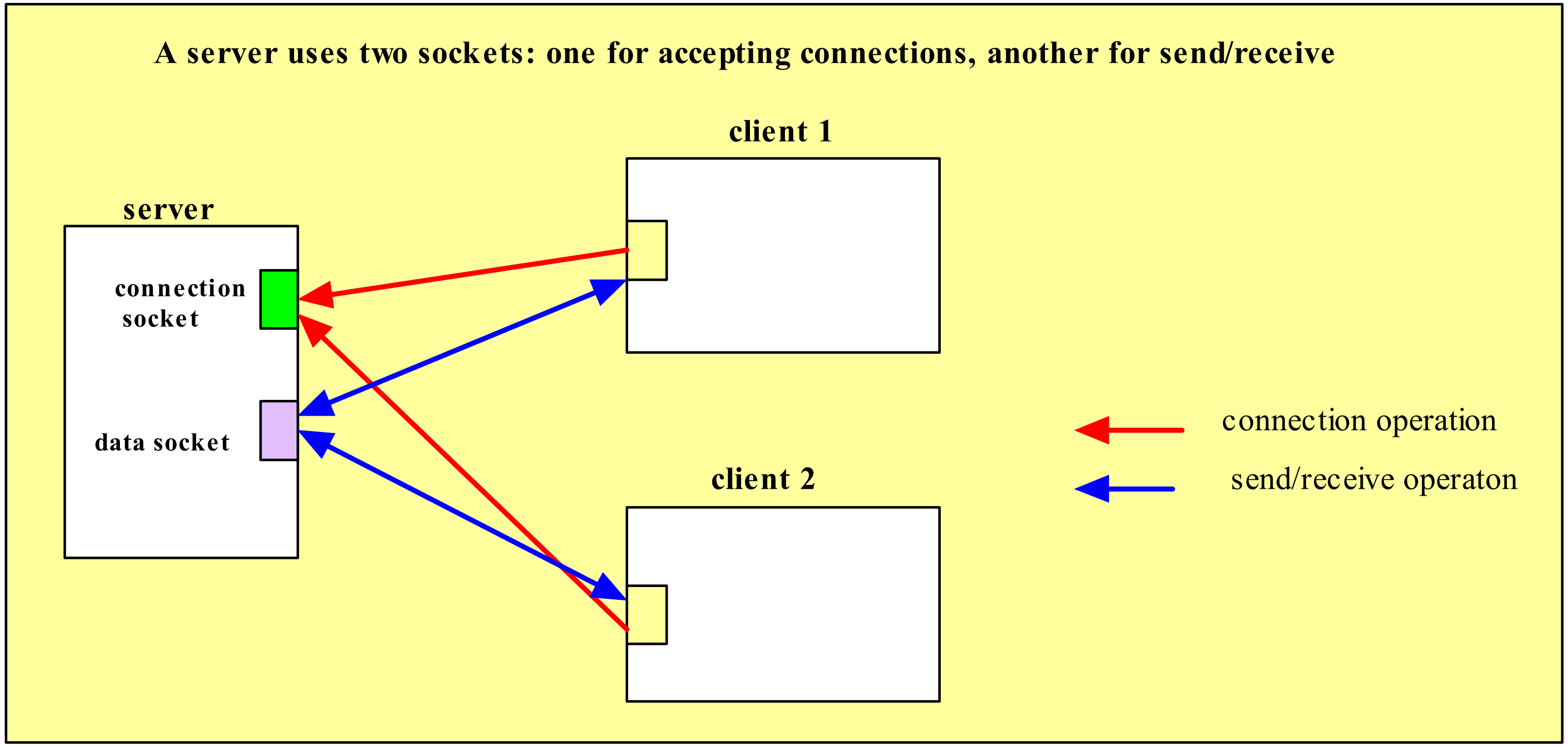
© *Addison-Wesley*

Sockets: connection oriented



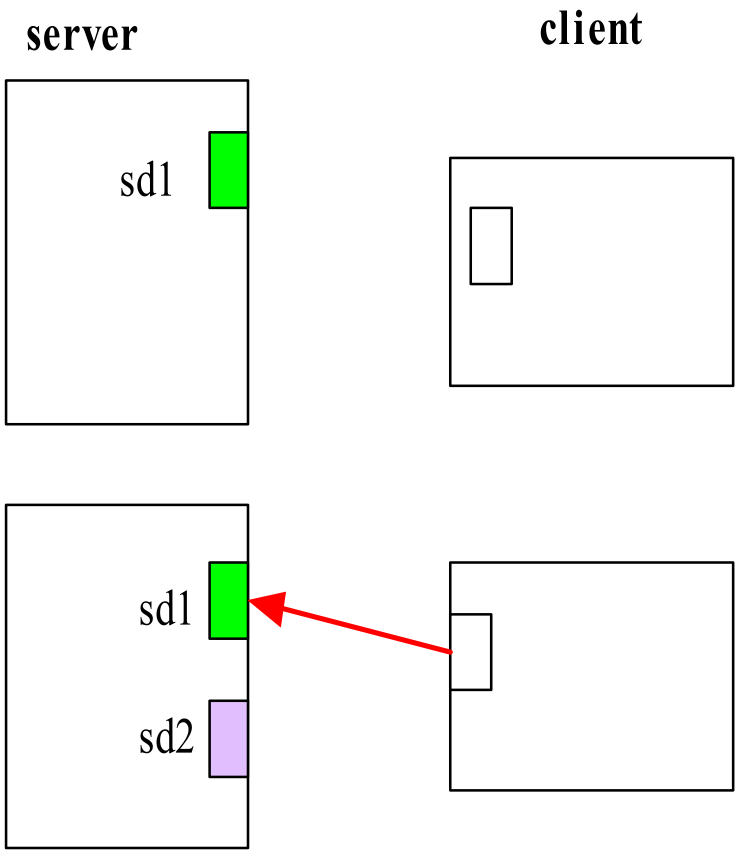
© *Addison-Wesley*

connection oriented server



Connection establishment

1. Server establishes a socket sd1 with local address, then listens for incoming connection on sd1

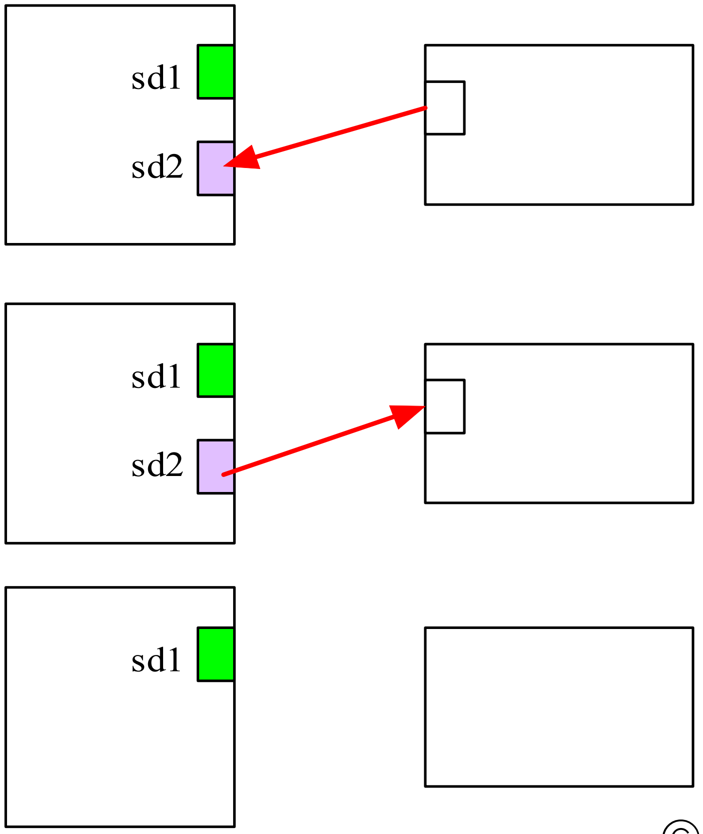


Client establishes a socket with remote (server's) address.

2. Server accepts the connection request and creates a new socket sd2 as a result.

© *Addison-Wesley*

Connection establishment



3. Server issues receive operation using sd2.

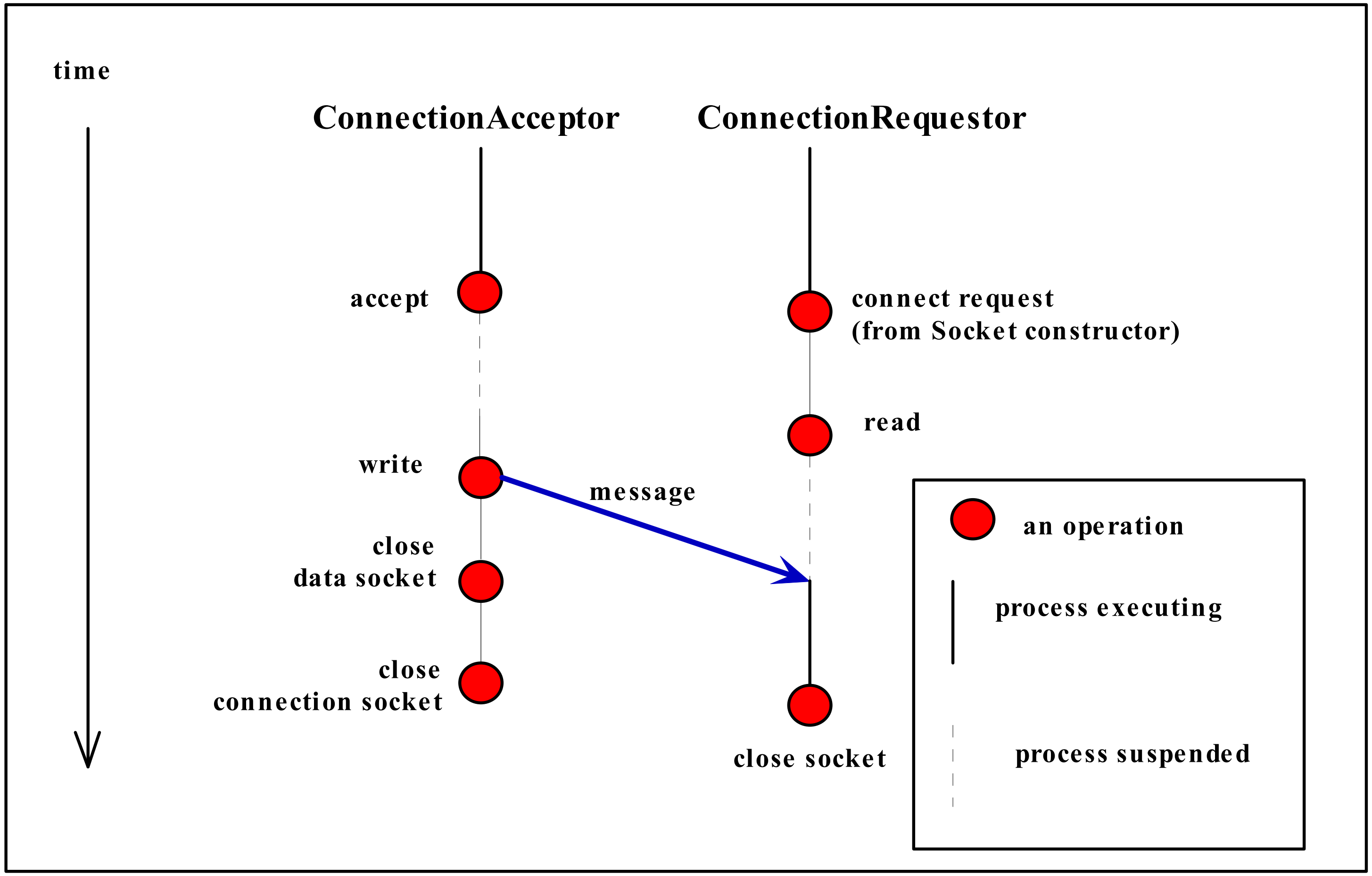
Client issues send operation.

4. Server sends response using sd2.

5. When the protocol has completed, server closes sd2; sd1 is used to accept the next connection

Client closes its socket when the protocol has completed

Operation example of a connection oriented service



© *Addison-Wesley*

IP Addresses

- An IP address is stored in a structure like:

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

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

Socket addresses in PF_INET

- Structure *struct sockaddr_in*
  - Must be initialized to 0
  - *sin_family*: domain (AF_INET)
  - *sin_port*: port
  - *sin_addr*: host address

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

struct sockaddr_in  {
         short                  sin_family;
         unsigned short         sin_port;
         struct in_addr         sin_addr;
         char                   sin_zero[8];
};
```

Socket addresses in PF_UNIX

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

struct sockaddr_un {
         short               sun_family;
         char                sun_path[108];
};
```

How to obtain a host’s name?

- This function provides the host’s name it is running on:

```
int gethostname(char *name, int namelen);
```

```
 void main ()
 {
    char host[256];
    int err;

    err = gethostname(host, 256);

    printf(“I’m running on %s\n”, host);

    exit(0);
 }
```

Obtain a host's address

- Users use addresses in text format:
  - dotted decimal: 138.100.8.100

- Functions to transform addresses

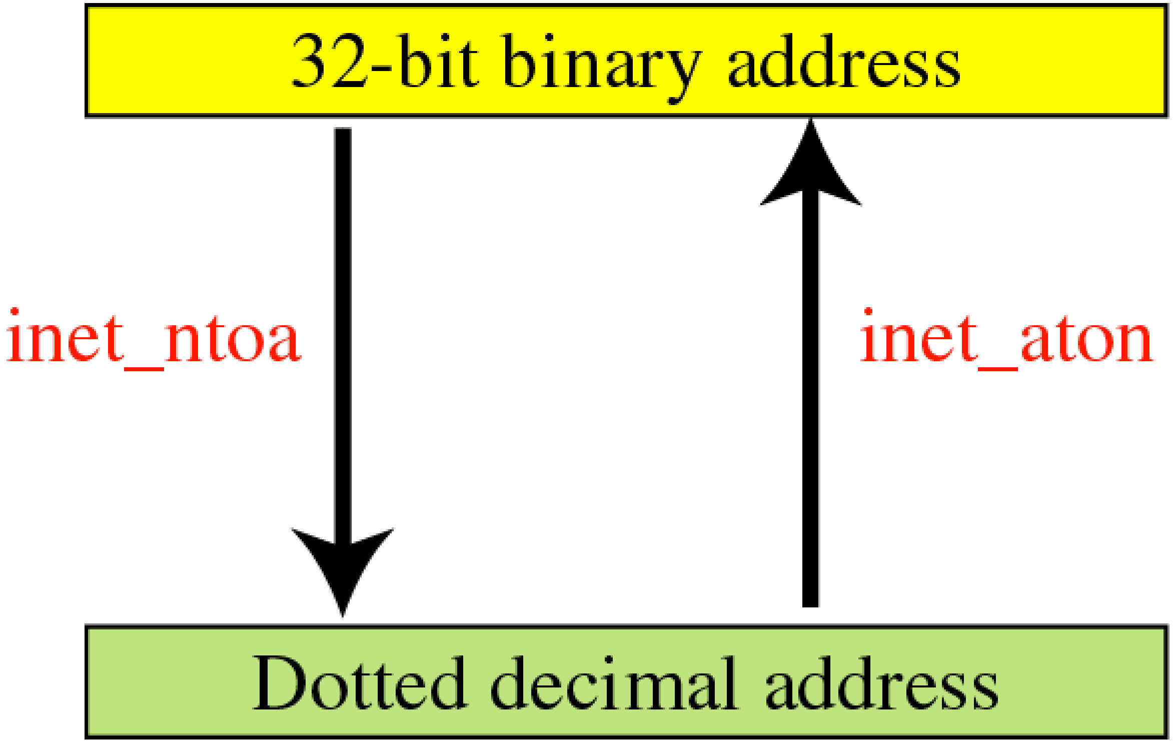
```
char *inet_ntoa(struct in_addr in);
```

  - Returns an IP address in dotted decimal notation.

```
int inet_aton(const char *cp, struct in_addr *inp);
```

  - Obtains an IP address from a dotted decimal notation

Addresses conversion



© *The McGraw-Hill*

Example of use

```
#include <netdb.h>
#include <stdio.h>

void main(int argc, char **argv){
        struct in_addr in;

         if (argc != 2) {
                printf("Use: address <dotted-decimal>\n");
                exit(0);
        }
        if (inet_aton(argv[1], &in) == 0) {
                printf("Error in address\n");
                exit(0);
        }

        printf(“The address is %s\n", inet_ntoa(in));

        exit(0);
}
```

Obtain a host’s address

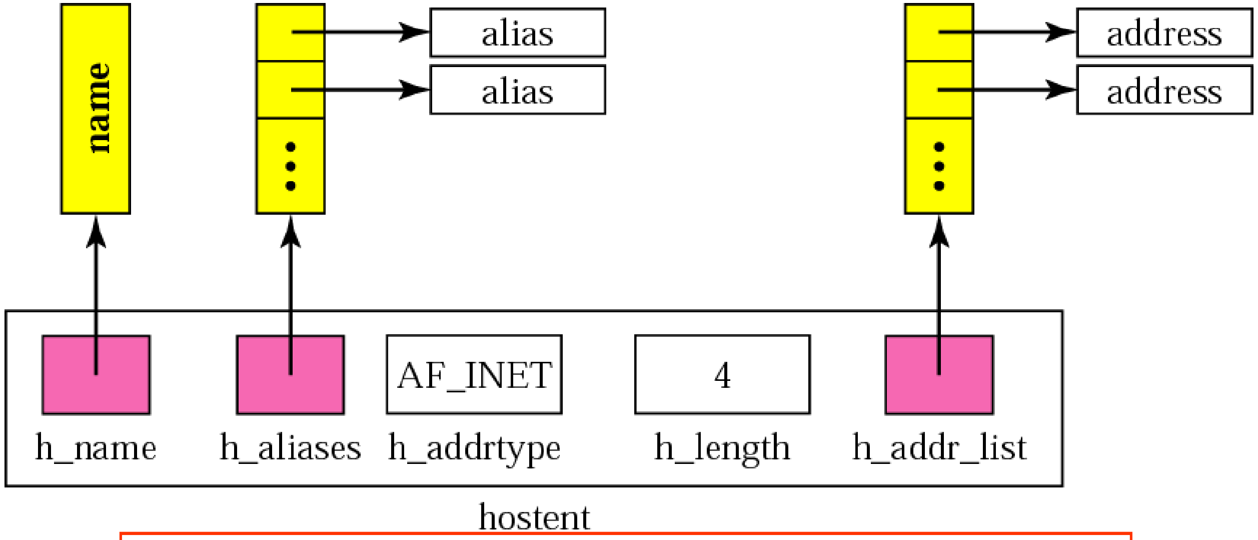
- Obtains a host’s address from a dotted domain address

```
#include <netdb.h>
struct hostent *gethostbyname(char *str);
```

- Obtains a host’s address from an IP address (`struct in_addr`)

```
#include <netdb.h>
struct hostent *gethostbyaddr(void *addr,
                                    int len,
                                    int type);
```

struct hostent



```
struct      hostent
{
            char                           *h_name ;
            char                          **h_aliases ;
            int                             h_addrtype ;
            int                             h_length ;
            char                          **h_addr_list ;
} ;
```

© *The McGraw-Hill*

Functions description

```
#include <netdb.h>
struct hostent *gethostbyname(char *str);
```

- Obtains a host's address from a dotted domain address
- *str* is the host's name

```
struct hostent *gethostbyaddr(void *addr, int len,
                              int type);
```

- Obtains a host's address from an IP address
- *addr* is a pointer to a *struct in_addr*
- *len is* the structure size
- *type* is AF_INET

Example I

- Program that obtains the dotted decimal address from a dotted domain.

```
void main(int argc, char **argv) {
   struct hostent *hp;
   struct in_addr in;

   hp = gethostbyname(argv[1]);
   if (hp == NULL) {
      printf(“Error in gethostbyname\n”);
      exit(0);
   }
   memcpy(&in.s_addr,*(hp->h_addr_list),sizeof(in.s_addr));

   printf(“%s is %s\n”, hp->h_name, inet_ntoa(in));
}
```

Example II

- Program that obtains the dotted domain address from a dotted decimal.

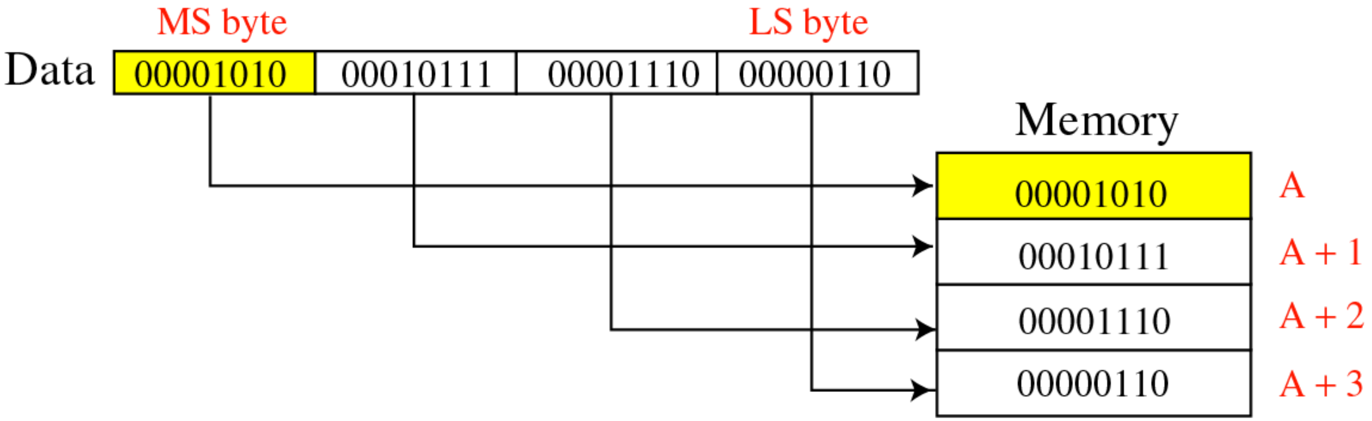
```
main(int argc, const  char **argv)
        {
        u_int addr;          struct hostent *hp;
        char **p;            struct in_addr in;
        char **q;

        if (argc != 2) {
                printf("Use: %s <IP address>\n", argv[0]);
                exit (1);
        }

        err = inet_aton(argv[1], &addr);

        if (err == 0) {
                printf(“IP address in format a.b.c.d\n");
                exit (2);
        }
```

Example II (cont.)

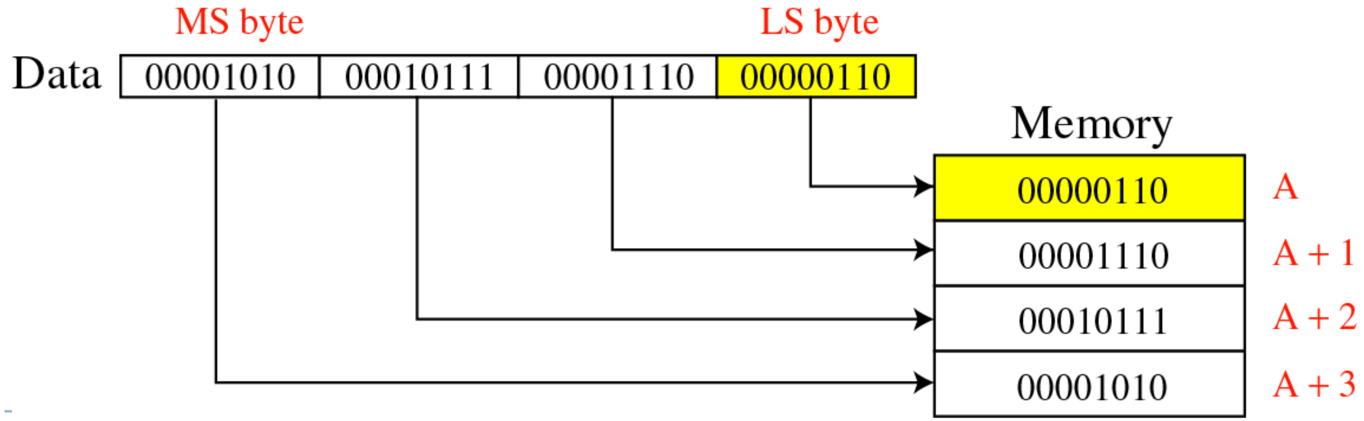
```
        hp=gethostbyaddr((char *) &addr,
                                  sizeof (addr), AF_INET);
       if (hp == NULL) {
              printf(“Error in gethostbyaddr\”n);
              exit (3);
       }

       for (p = hp->h_addr_list; *p!=0; p++){
              memcpy(&in.s_addr, *p, sizeof(in.s_addr));
              printf("%s\t%s",inet_ntoa(in), hp->h_name);
        }
       for (q=hp->h_aliases; *q != 0; q++)
               printf("%s\n", *q);

        exit(0);
}
```

Byte ordering

- In TCP/IP numbers are used in *big-endian*.
- Big-endian



- Little-endian



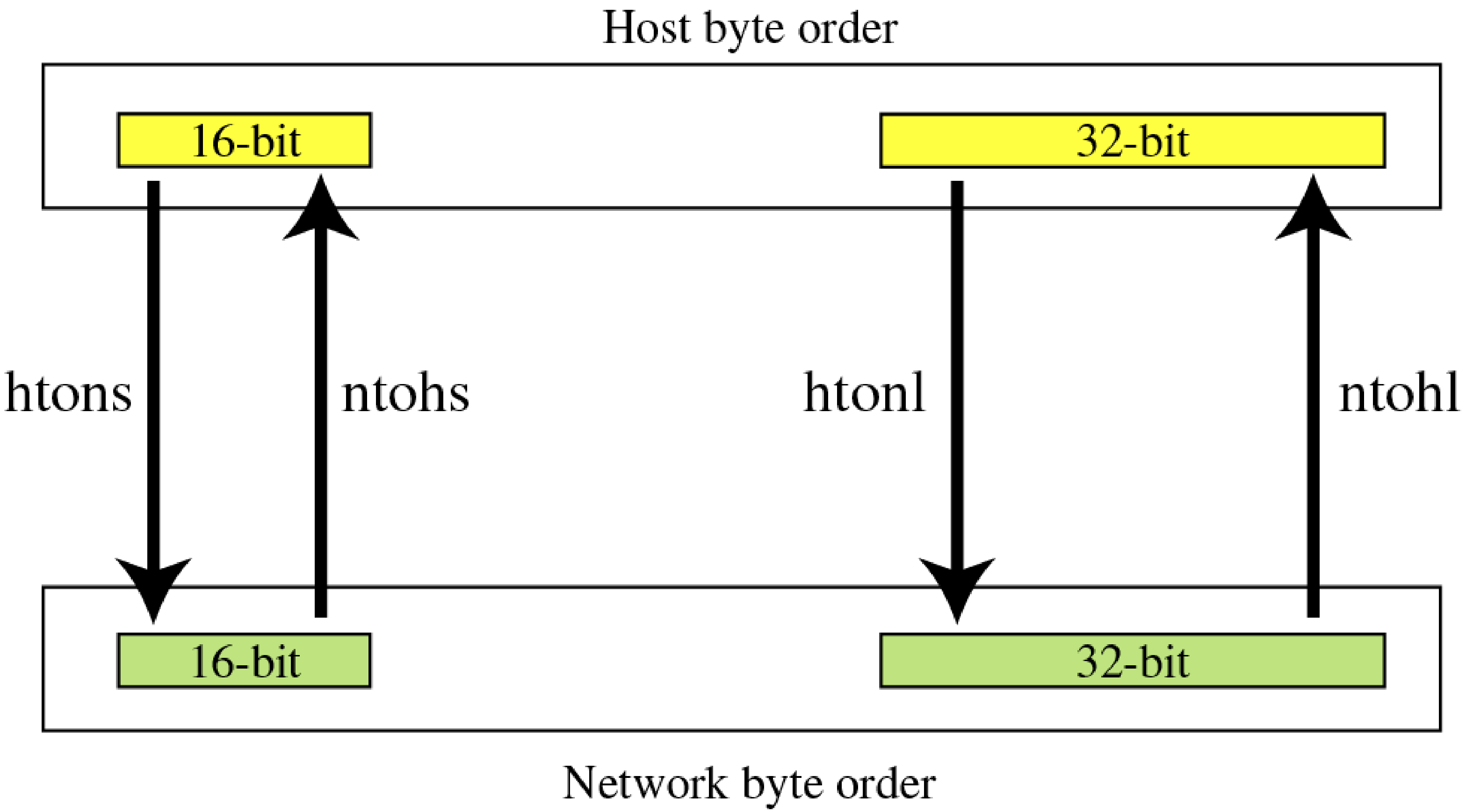
Conversion functions

- In hosts that do not use this format it is necessary to use functions to translate numbers between network (TCP/IP) format and host format:

```
u_long  htonl(u_long  hostlong)
u_short htons(u_short hostshort)
u_long  ntohl(u_long  netlong)
u_short ntohs(u_short netshort)
```

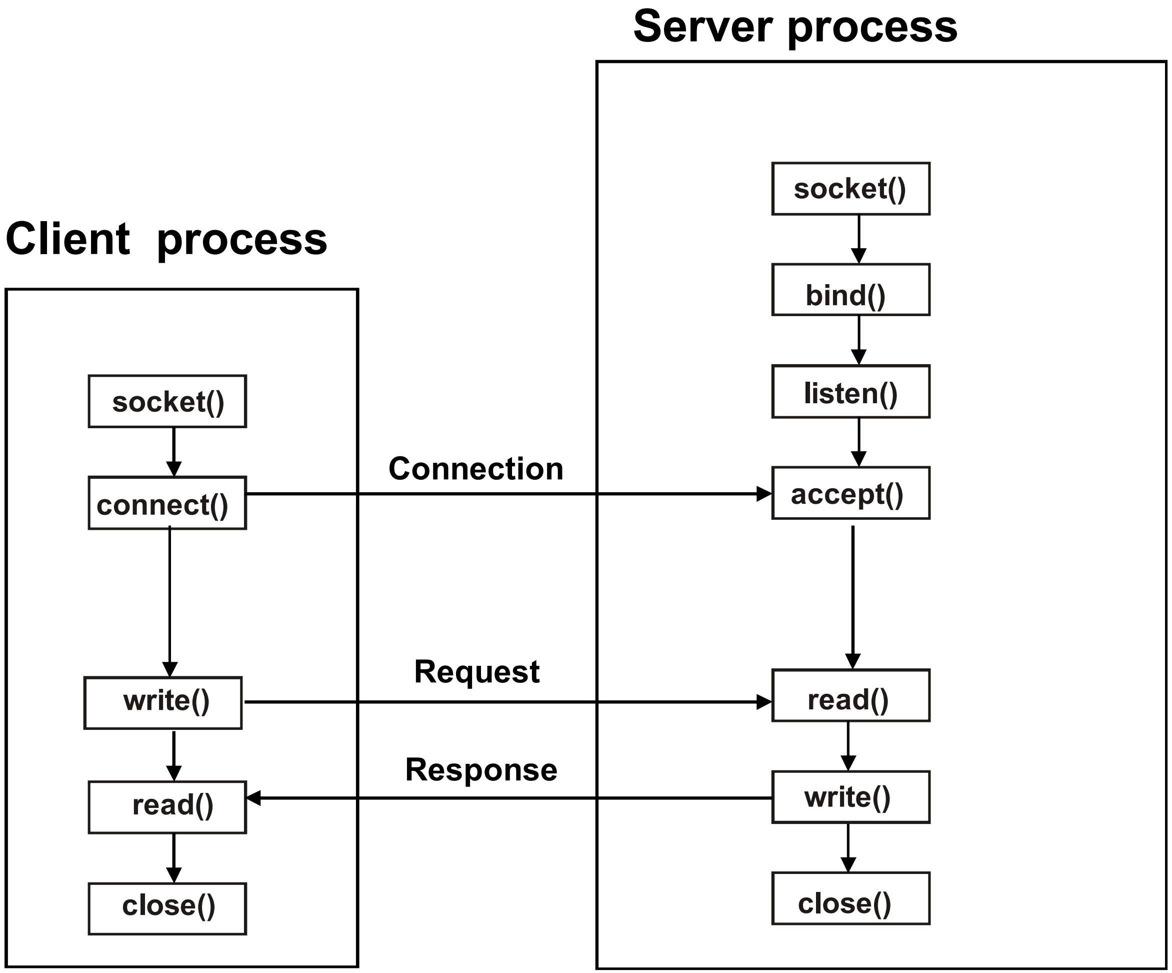
- **`hton*`**: Host->Network (TCP/IP).
- **`ntoh*`**: Network (TCP/IP)->Host.

Conversion functions

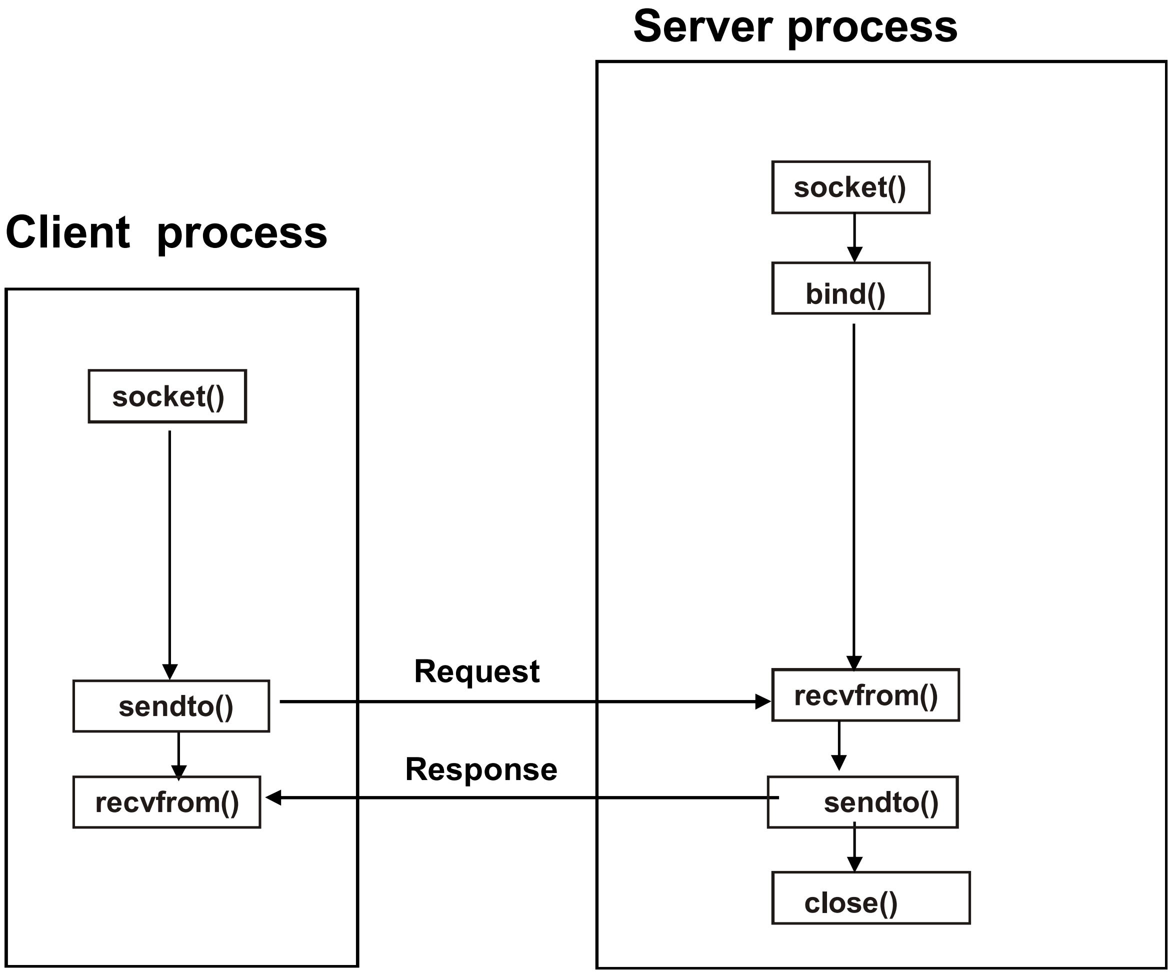


© *The McGraw-Hill*

Communication model with *stream* sockets



Communication model with *datagram* sockets



Socket creation

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol)
```

- Creates a socket
  - Returns a socket descriptor
  - Or -1 if error
- Arguments:
  - **`domain`**: PF_UNIX or PF_INET
  - **`type`**: SOCK_STREAM or SOCK_DGRAM
  - **`protocol`**: depends on domain and type
    - 0 chooses the most appropriate
    - Specified in */etc/protocols*
- The newly created socket has no address assigned

Address assignment

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sd, struct sockaddr *addr, int addrlen)
```

- Arguments:
  - **`sd`**: descriptor returned by socket()
  - **`addr`**: address to assign
  - **`addrlen`**: address length
- If an address is not assigned (like in clients)
  - One is automatically assigned (ephemeral port) when used for the first time (*connect* or *sendto*)
- Addresses in PF_INET domain (*struct sockaddr_in*)
  - Ports in range 0..65535. Reserved: 0..1023.
    - If 0, the system chooses one
  - Host: a local IP address
    - INNADDR_ANY: chooses any available address in the host
- Port spaces for *streams* and datagrams are independents
- Returns –1 if error and 0 if success

Prepare to accept connections

- Done in the *stream* sever after *socket*, and *bind*

```
#include <sys/types.h>
#include <sys/socket.h>
int listen(int sd, int baklog)
```

- Arguments:
  - **sd**: descriptor returned by socket()
  - **backlog**: maximum number of pending requests to accept that will be queued (5 or 10 recommended)
- Prepares the socket to accept connections.
- Returns –1 if error or 0 if success.

Accept connections

- Done in the *stream* sever after *socket*, *bind*, and *listen*
- When the connection is done the server obtains:
  - Client's socket address
  - A new descriptor that remains connected to client's socket
- After connecting there are two active sockets in the server:
  - The original: to accept new connections
  - The new: to send/recv data through the connection

Accept connections

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int sd, struct sockaddr *addr, int *addrlen)
```

- Arguments*:*
  - **`sd`**: descriptor returned by socket()
  - **`addr`**: client's socket address returned
  - **`addrlen`**: parameter value-result
    - Before the call: size of addr
    - After the call: size of client address returned in addr.
- Returns a new socket descriptor associated to the connection
  - Or -1 if error

Connection request

```
#include <sys/types.h>
#include <sys/socket.h>
int connect(int sd, struct sockaddr *addr, int addrlen)
```

- Arguments*:*
  - **`sd`**: descriptor returned by socket()
  - **`addr`**: remote socket address
  - **`addrlen`**: address length
- If the socket has no assigned address, it automatically receives one
- Done in the client
- Usually used with *streams*
- Returns –1 if error or 0 if success

Connection establishment in TCP

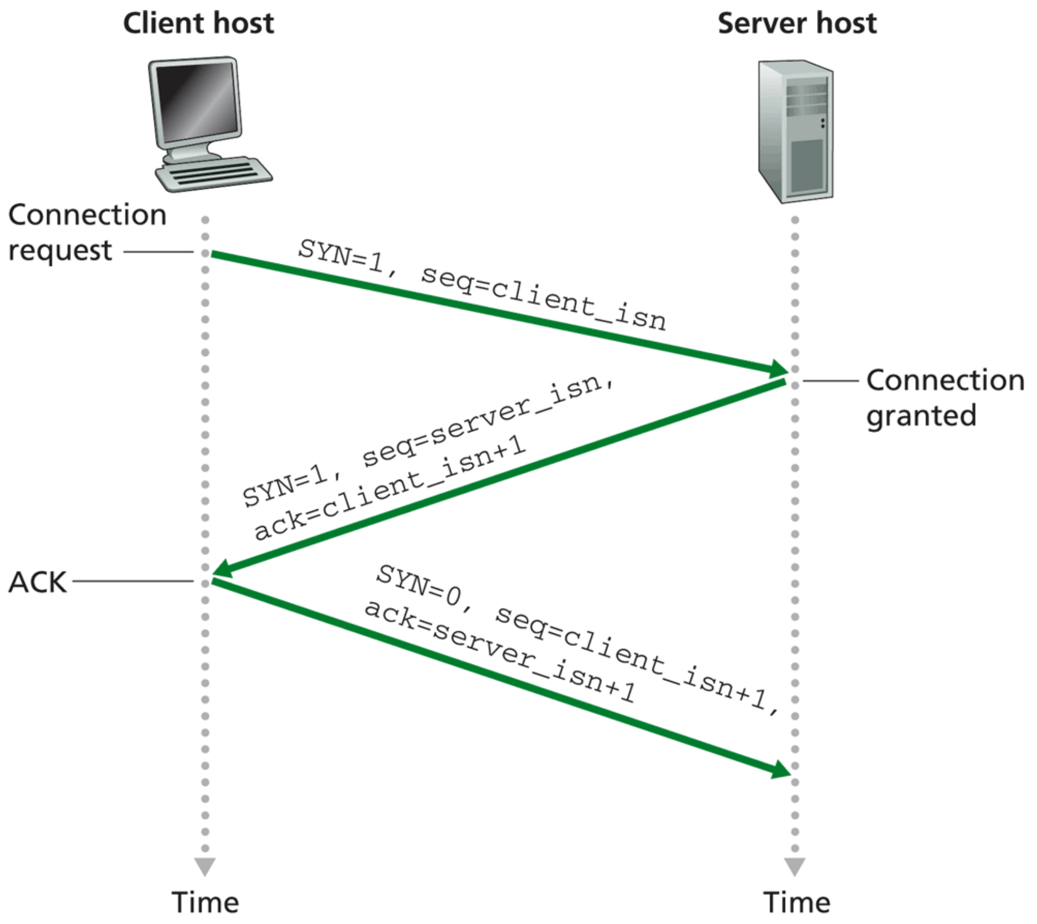


Figure 3.38 ♦ TCP three-way handshake: segment exchange

Obtain a socket address

- Obtain the current address to which a socket is bound

```
int getsockname(int sd, struct sockaddr *addr, int *addrlen)
```

  - **sd**: descriptor returned by *socket()*
  - **addr**: socket address returned
  - **addrlen**: parameter value-result (same as in *accept*)

- Obtain the address of the peer connected to a socket

```
int getpeername(int sd, struct sockaddr *addr, int *addrlen)
```

  - **sd**: returned by *socket()*
  - **addr**: remote socket address
  - **addrlen**: parameter value-result

Data transfers with *streams*

- Once connected, both sides can transfer data.
- Send:

```
int send(int sd, void *buf, int len, int flags)
```

    - Returns the number of bytes sent or –1 if error
  - It is also possible to use *write*.
- Receive:

```
int recv(int sd, void *buf, int len, int flags)
```

    - Returns the number of bytes received or –1 if error
  - It is also possible to use *read*
- It is important to always check the returned value: they might not transfer all the data

Data transfers with *streams*

- Function that sends a data block with retries:

```
int send_retry(int socket, char *message, int len)
{
    int r;
    int l = len;
    do {
           r = send(socket, message, l, 0);
           l = l - r;
           message = message + r;
    } while ((l>0) && (r>=0));

    if (r < 0)
      return (-1);   /* fail */
    else
      return(0);
}
```

Data transfers with datagrams

- Thre are not real connections
- To use a socket to transfer data it is neccessary:
  - To create it: *socket()*
  - To assign it an address: *bind()* (if not, the system will do)
- Send:

```
int sendto(int sockfd, char *buf, int len, int flags, struct sockaddr *dest_addr, int addrlen)
```

  - Returns the number of bytes sent or –1 if error
  - *dest_addr*: remote socket address and *long* is its length
- Receive:

```
int recvfrom(int sockfd, void *buf, int len, int flags, struct sockaddr *src_addr, int addrlen)
```

  - Returns the number of bytes received or –1 if error
  - *dest_addr*: remote socket address and *long* is its length

Close a socket

- We use ***close()*** to close both types of sockets

```
int close(int sd);
```

- If it is a stream socket, *close()* closes the connection in both sides
- It is possible to close only one side:

```
int shutdown(int sd, int how);
```

  - **sd**: descriptor returned by socket()
  - **how**: `SHUT_RD`, `SHUT_RW` or `SHUT_RDWR`
    - `SHUT_RD`: Further receptions will be disallowed
    - `SHUT_RW`: Further transmitions will be disallowed
    - `SHUT_RDWR`: Further receptions and transmitions will be disallowed

Information associated to a socket

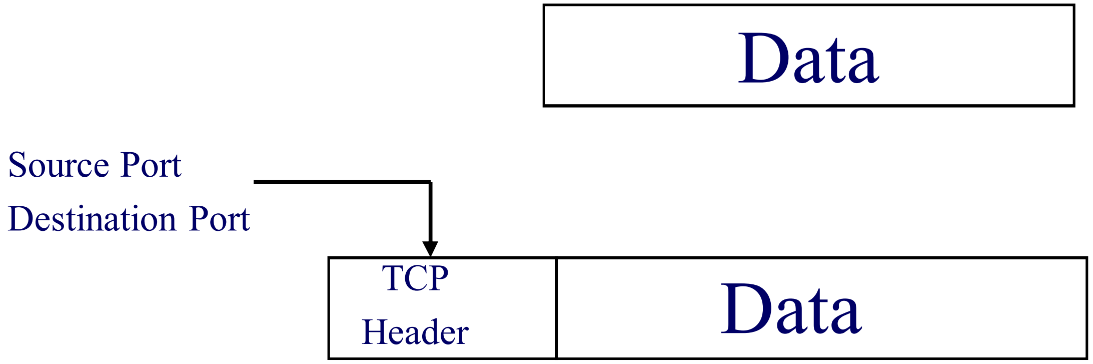
- Protocol
  - TCP, UDP
- Local IP address (source)
- Local port (source)
- Remote IP address (destination)
- Remote port (destination)

(Protocol, Local-IP, Local-Port, Remote-IP, Remote-Port)

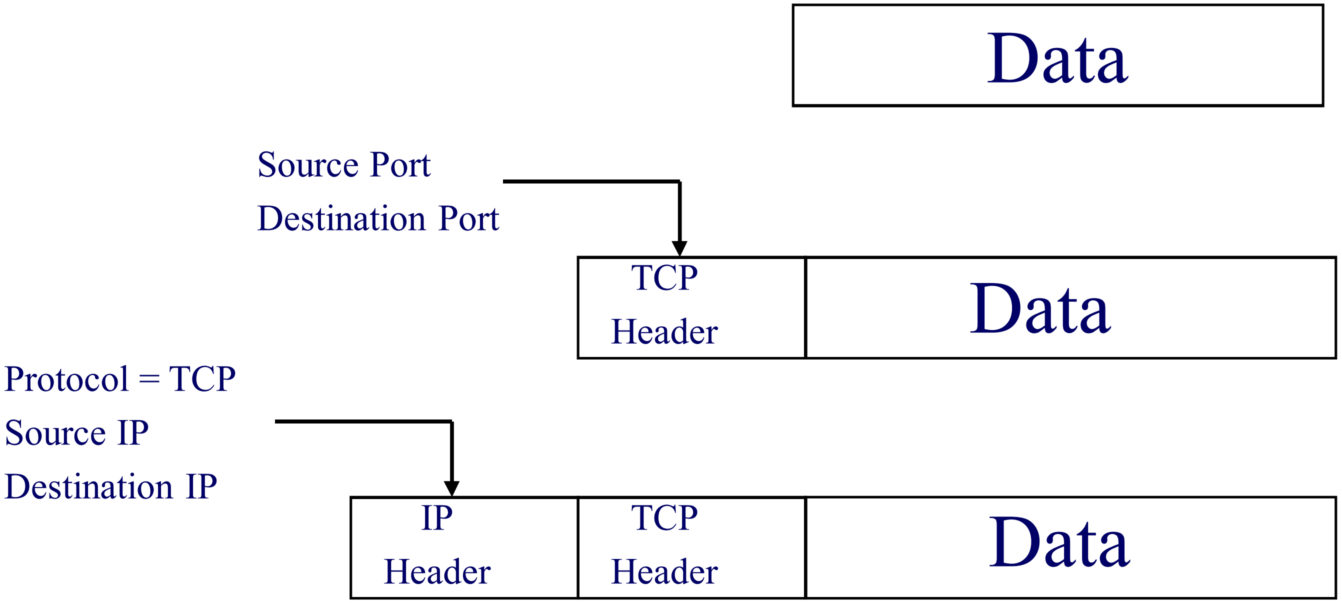
TCP packet encapsulation

Data

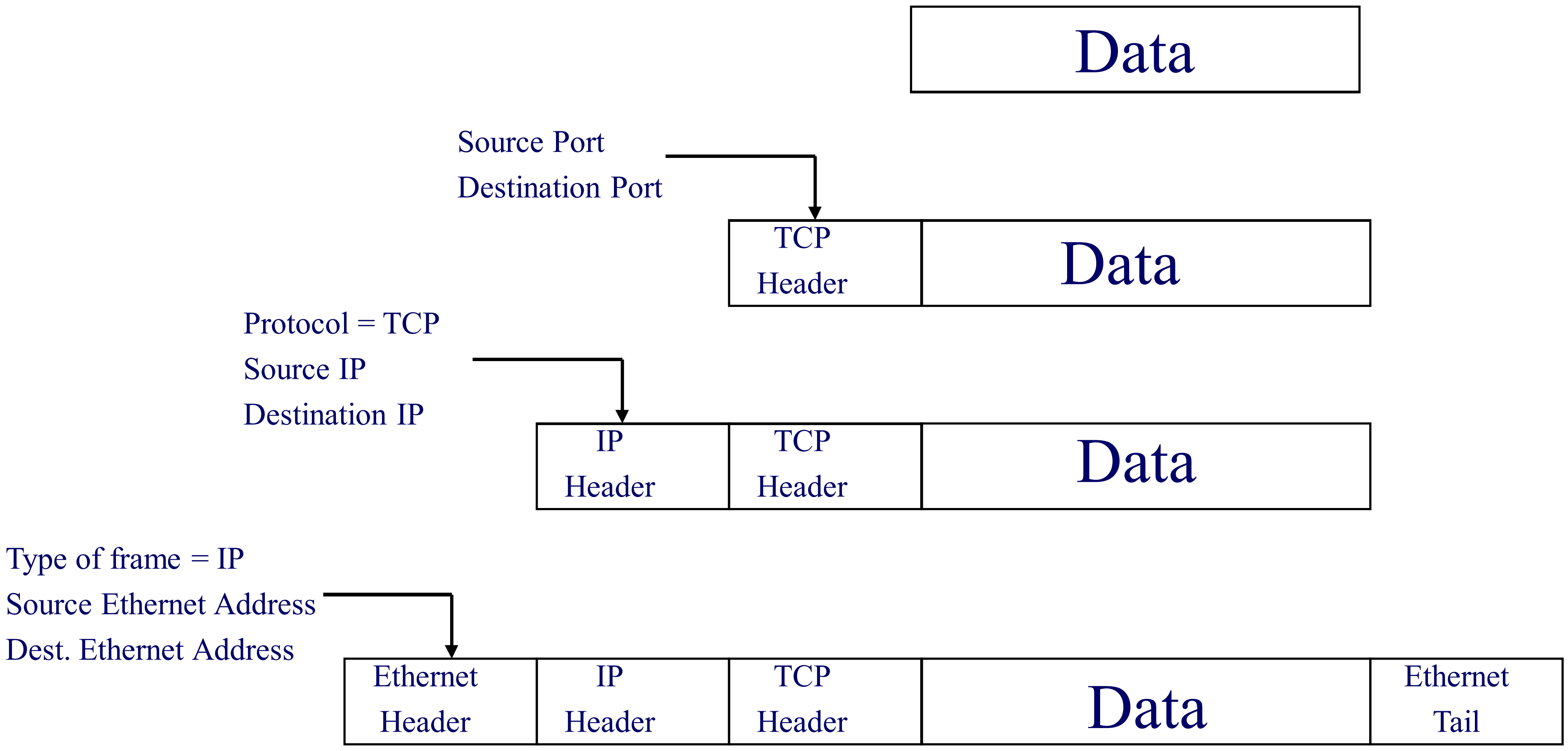
TCP packet encapsulation



TCP packet encapsulation



TCP packet encapsulation



TCP Connection

Server

(host-A, 22)

(tcp, host-A, 22, -, -)

client

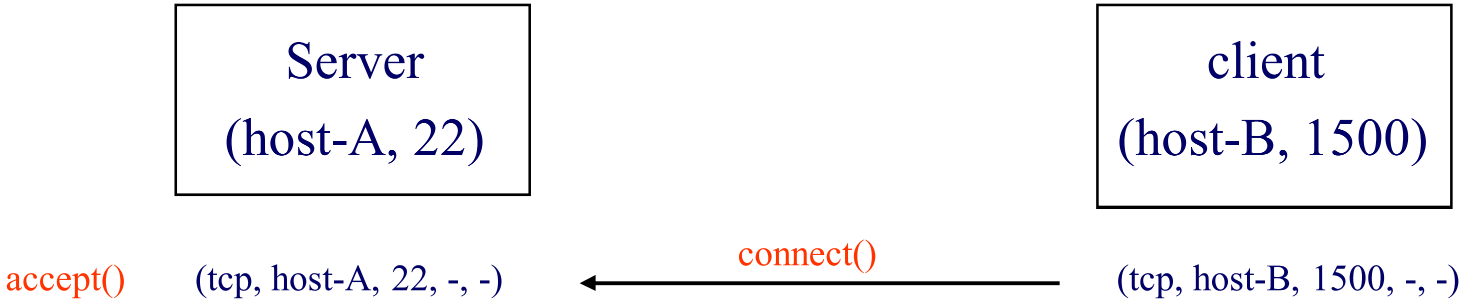
(host-B, 1500)

(tcp, host-B, 1500, -, -)

TCP Connection

	Server (host-A, 22)	client (host-B, 1500)
accept()	(tcp, host-A, 22, -, -)	(tcp, host-B, 1500, -, -)

TCP Connection



Server

(host-A, 22)

client

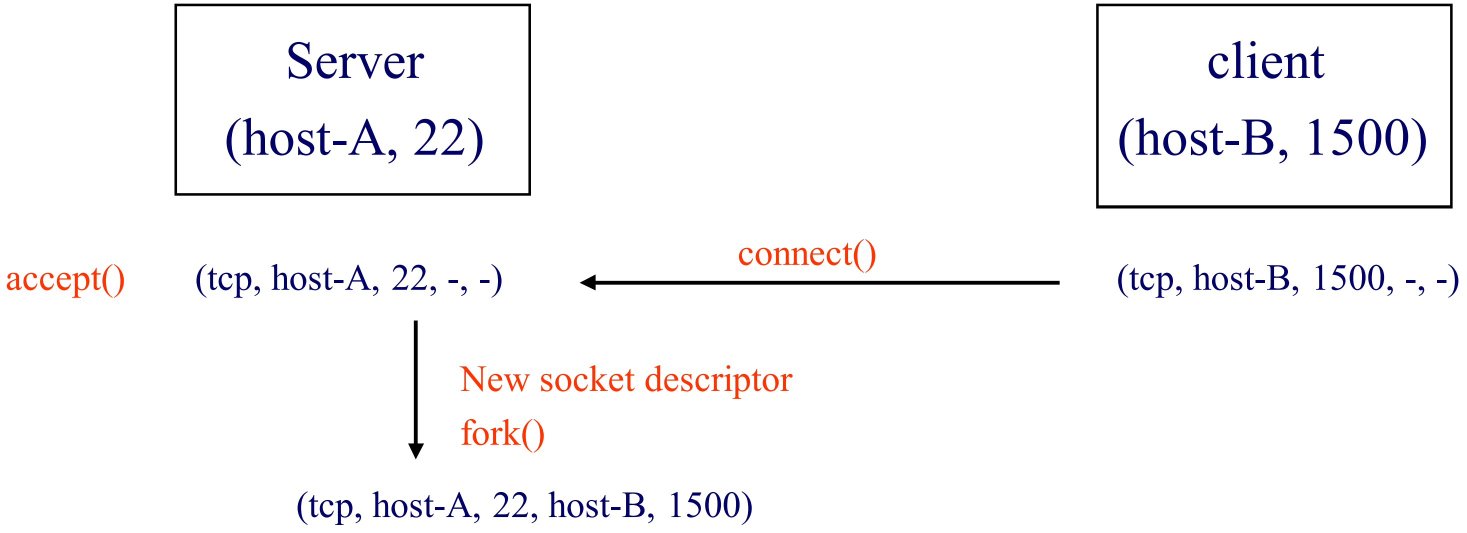
(host-B, 1500)

accept() (tcp, host-A, 22, -, -)

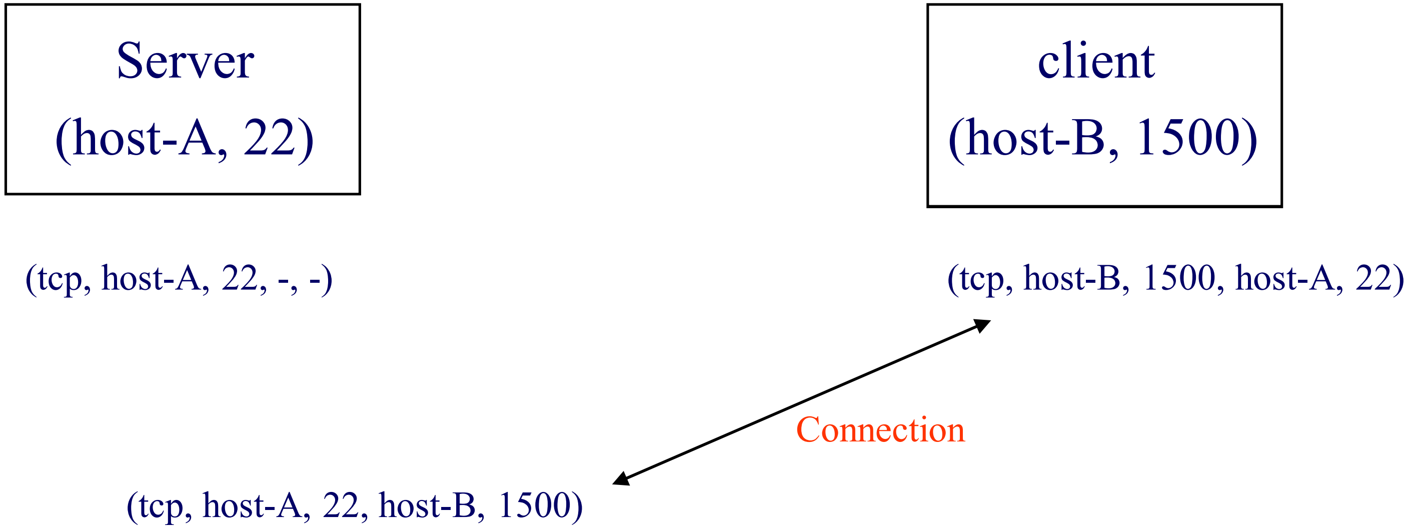
connect()

(tcp, host-B, 1500, -, -)

TCP Connection



TCP Connection



Server

(host-A, 22)

client

(host-B, 1500)

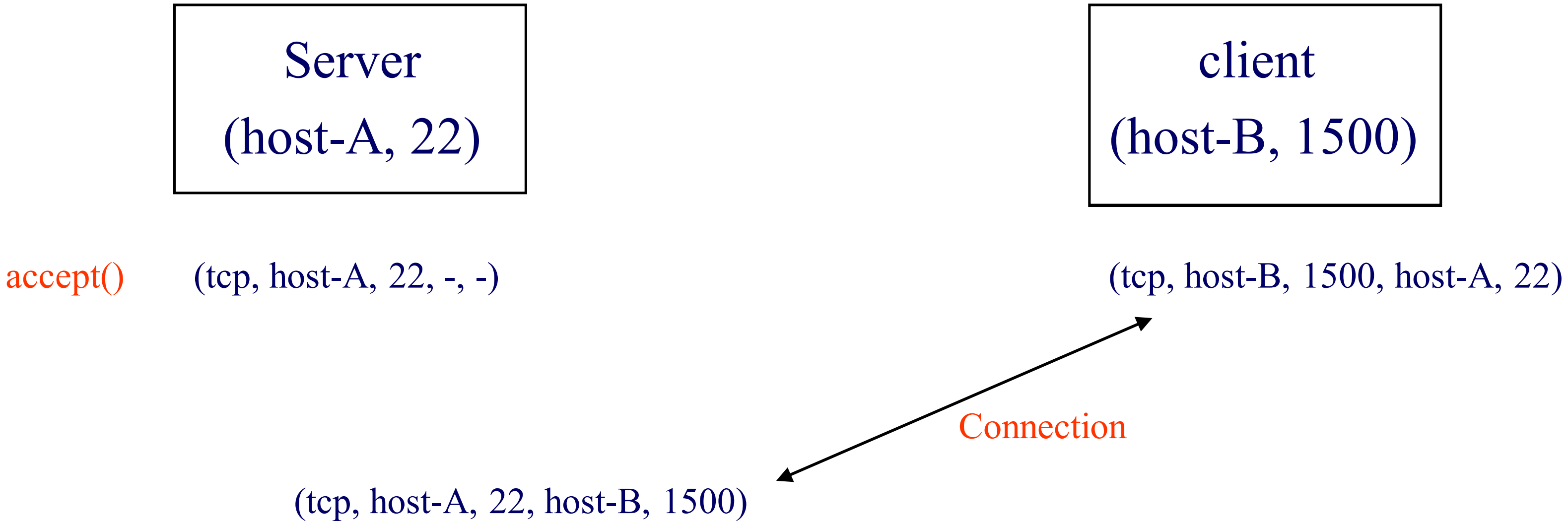
(tcp, host-A, 22, -, -)

(tcp, host-B, 1500, host-A, 22)

Connection

(tcp, host-A, 22, host-B, 1500)

Conexión con TCP



Server

(host-A, 22)

client

(host-B, 1500)

accept()

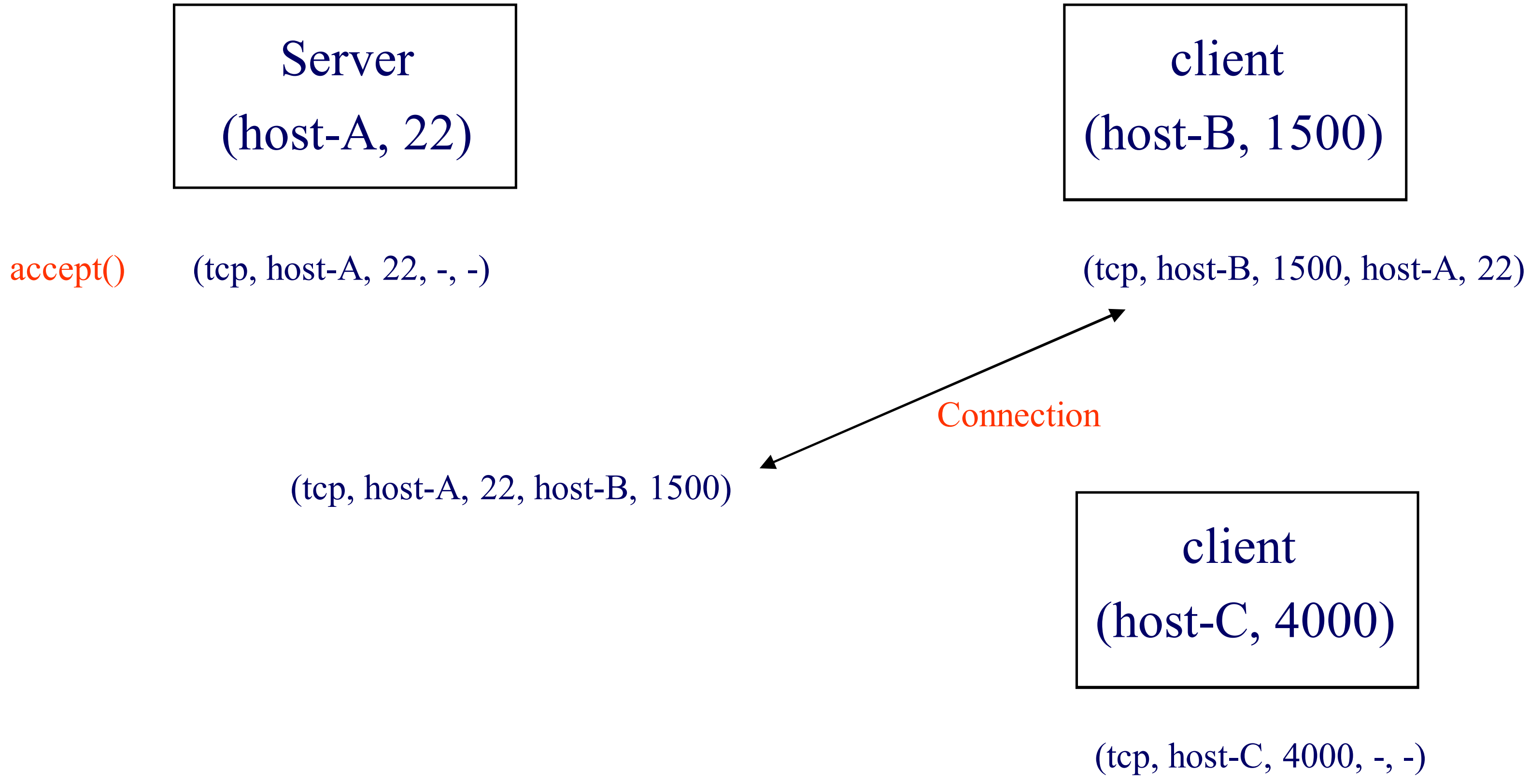
(tcp, host-A, 22, -, -)

(tcp, host-B, 1500, host-A, 22)

Connection

(tcp, host-A, 22, host-B, 1500)

Conexión con TCP



Server
(host-A, 22)

accept()

(tcp, host-A, 22, -, -)

client
(host-B, 1500)

(tcp, host-B, 1500, host-A, 22)

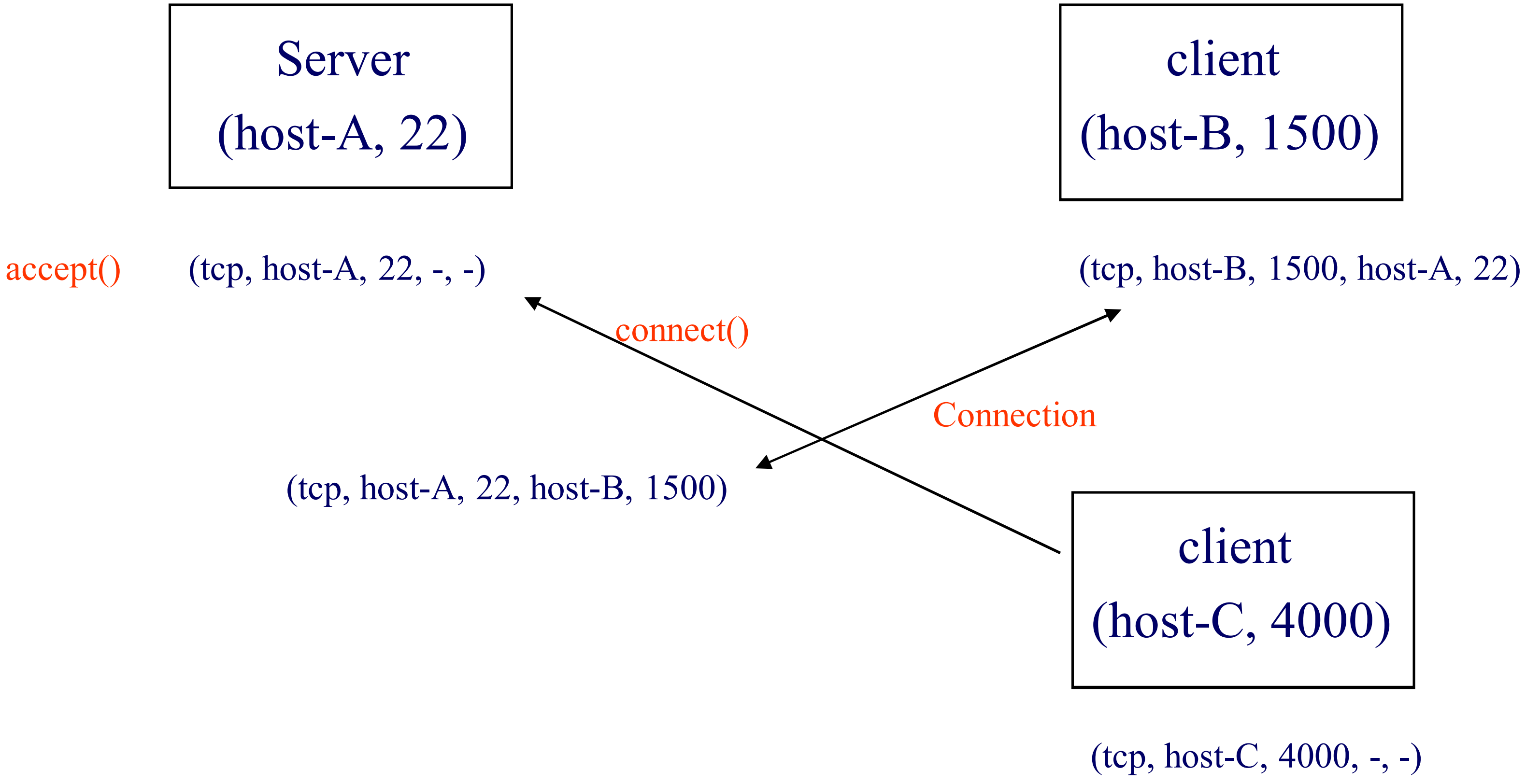
Connection

(tcp, host-A, 22, host-B, 1500)

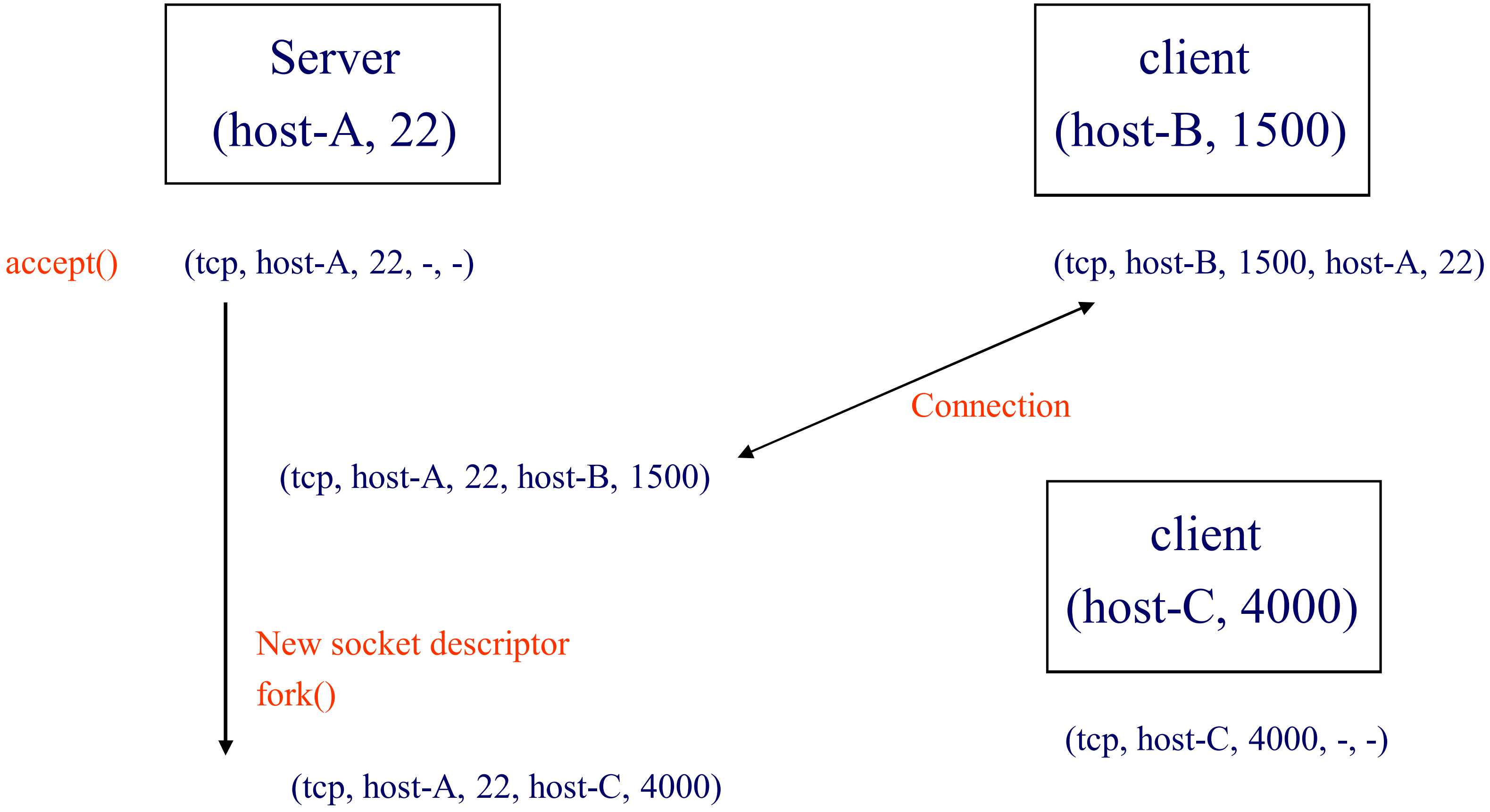
client
(host-C, 4000)

(tcp, host-C, 4000, -, -)

Conexión con TCP



TCP Connection



Server

(host-A, 22)

client

(host-B, 1500)

accept()

(tcp, host-A, 22, -, -)

(tcp, host-B, 1500, host-A, 22)

Connection

(tcp, host-A, 22, host-B, 1500)

client

(host-C, 4000)

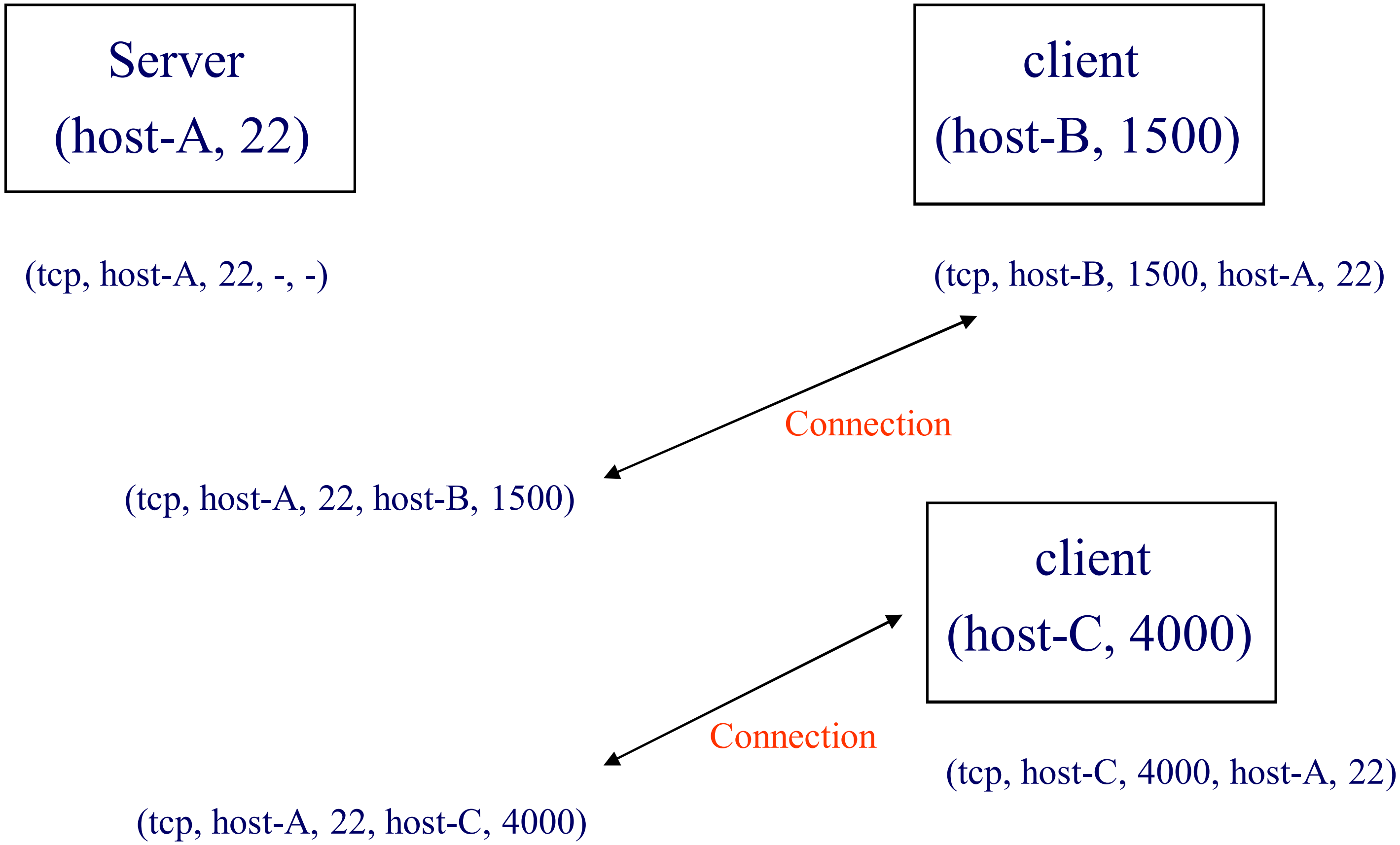
New socket descriptor

fork()

(tcp, host-C, 4000, -, -)

(tcp, host-A, 22, host-C, 4000)

TCP Connection



Server

(host-A, 22)

client

(host-B, 1500)

(tcp, host-A, 22, -, -)

(tcp, host-B, 1500, host-A, 22)

Connection

(tcp, host-A, 22, host-B, 1500)

client

(host-C, 4000)

Connection

(tcp, host-C, 4000, host-A, 22)

(tcp, host-A, 22, host-C, 4000)

Configuration of options

- There are several levels, depending on the protocol
  - SOL_SOCKET: options independent of the protocol
  - IPPROTO_TCP: options for TCP protocol
  - IPPTOTO_IP: options for IP protocol

- Obtain socket options

```
int getsockopt (int sd, int level, int optname, void *optval, int *optlen)
```

- Modify socket options

```
int setsockopt (int sd, int level, int optname, void *optval, int *optlen)
```

- Examples (SOL_SOCKET):
  - SO_REUSEADDR: allows to reuse addresses

Why use SO_REUSEADDR?

- TCP mantains the connections blocked for a period of time (`TIME_WAIT`).
- Although the connection has been already closed, and it cannot be used, associated internal tables are still alive just in case there are frames travelling through the network

```
int val = 1;
setsockopt(  sd,   SOL_SOCKET,
                   SO_REUSEADDR,
                   (void *) &val,
                   sizeof(int));
```

SO_RCVBUF, SO_SNDBUF

- Send and receive buffers
- To set the size of the transmission:

```
int size =  16384;
err = setsockopt(s, SOL_SOCKET, SO_SNDBUF,
     (char *)&size,  (int)sizeof(size));
```

- To know the size of the transmission :

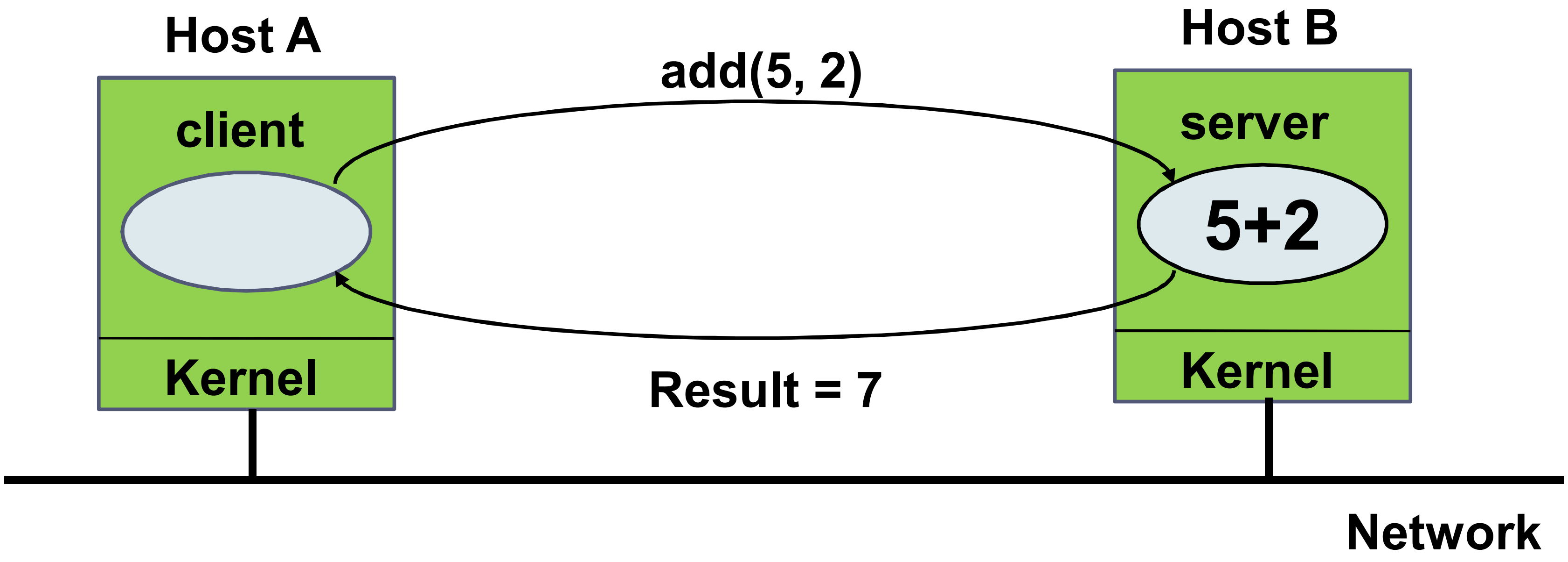
```
int size;
err = getsockopt(s, SOL_SOCKET, SO_SNDBUF,
     (char *)&size,(int)sizeof(size));
printf(“%d\n”, size)
```

TCP_NODELAY

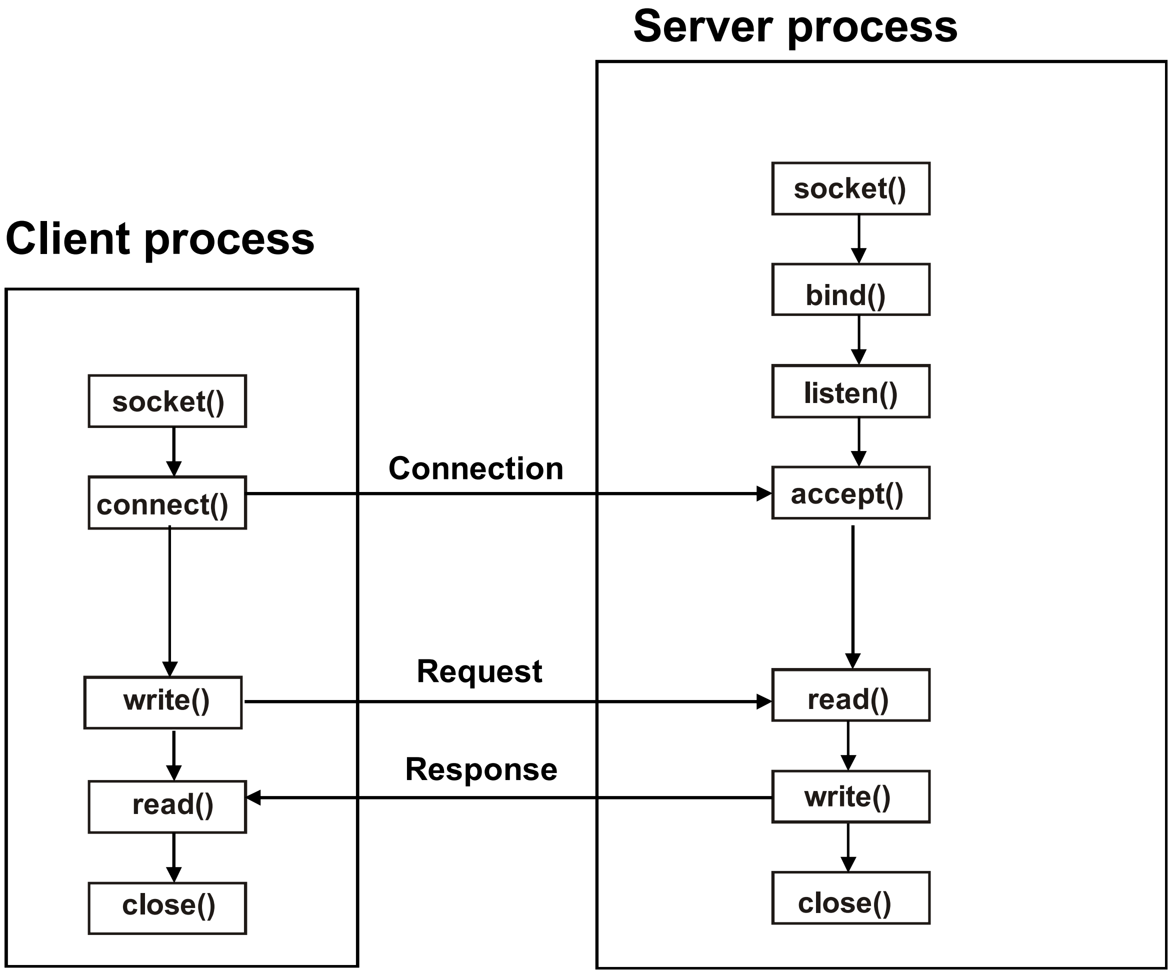
- Immediate send (does not try to group messages close together in time)

```
int option = 1;
rc = setsockopt(s, IPPROTO_TCP,
                 TCP_NODELAY,
                 &option,
                 sizeof(option));
```

Example (TCP)


Host A
client
Kernel
add(5, 2)
Result = 7
Host B
server
5+2
Kernel
Network

Communication model



Server (TCP)

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

void main(int argc, char *argv[])
{
          struct sockaddr_in server_addr,  client_addr;
          int sd, sc;
          int size, val;
          int num[2], res;

          sd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

          val = 1;
          setsockopt(sd, SOL_SOCKET, SO_REUSEADDR, (char *) &val, sizeof(int));

          bzero((char *)&server_addr, sizeof(server_addr));
          server_addr.sin_family        = AF_INET;
          server_addr.sin_addr.s_addr   = INADDR_ANY;
          server_addr.sin_port          = htons(4200);


          bind(sd, &server_addr, sizeof(server_addr));
```

Server (TCP)

```
          listen(sd, 5);

          size = sizeof(client_addr);

          while (1)
          {
                    printf(“waiting for connection\n");

                    sc = accept(sd, (struct sockaddr *) &client_addr, &size);


                    send ( sc, (char *)num, 2*sizeof(int), 0); // receives request

                    res = num[0] + num[1];                     // processes request

                    recv(sc, &res, sizeof(int), 0);            // sends result

                    close(sc);                                 // closes connection (sc)
          }

          close (sd);
          exit(0);
}
```

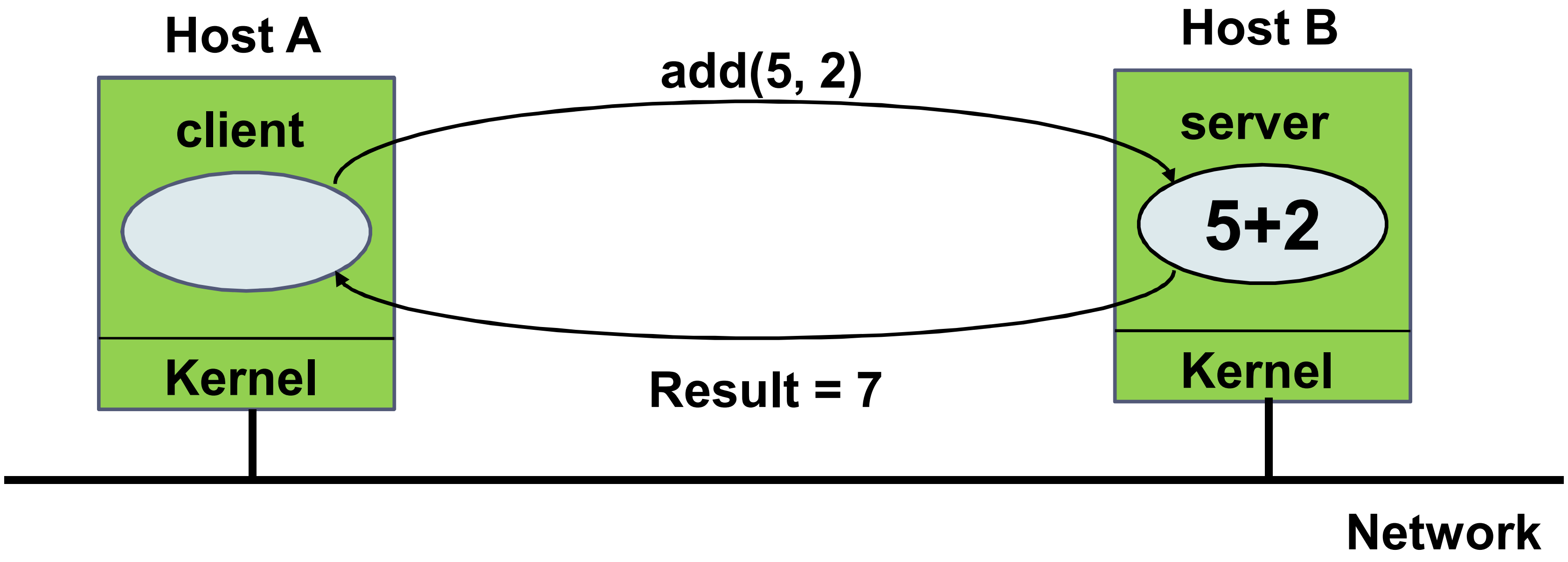
Client (TCP)

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

void main(int argc, char **argv)  // in argv[1] is the server
{
   int sd;
   struct sockaddr_in server_addr;
   struct hostent *hp;
   int num[2], res;

   if (argc != 2){
          printf("Use: client <server_address>\n");
          exit(0);
   }

   sd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

   bzero((char *)&server_addr, sizeof(server_addr));
   hp = gethostbyname (argv[1]);

   memcpy (&(server_addr.sin_addr), hp->h_addr, hp->h_length);
   server_addr.sin_family     = AF_INET;
   server_addr.sin_port       = htons(4200);
```

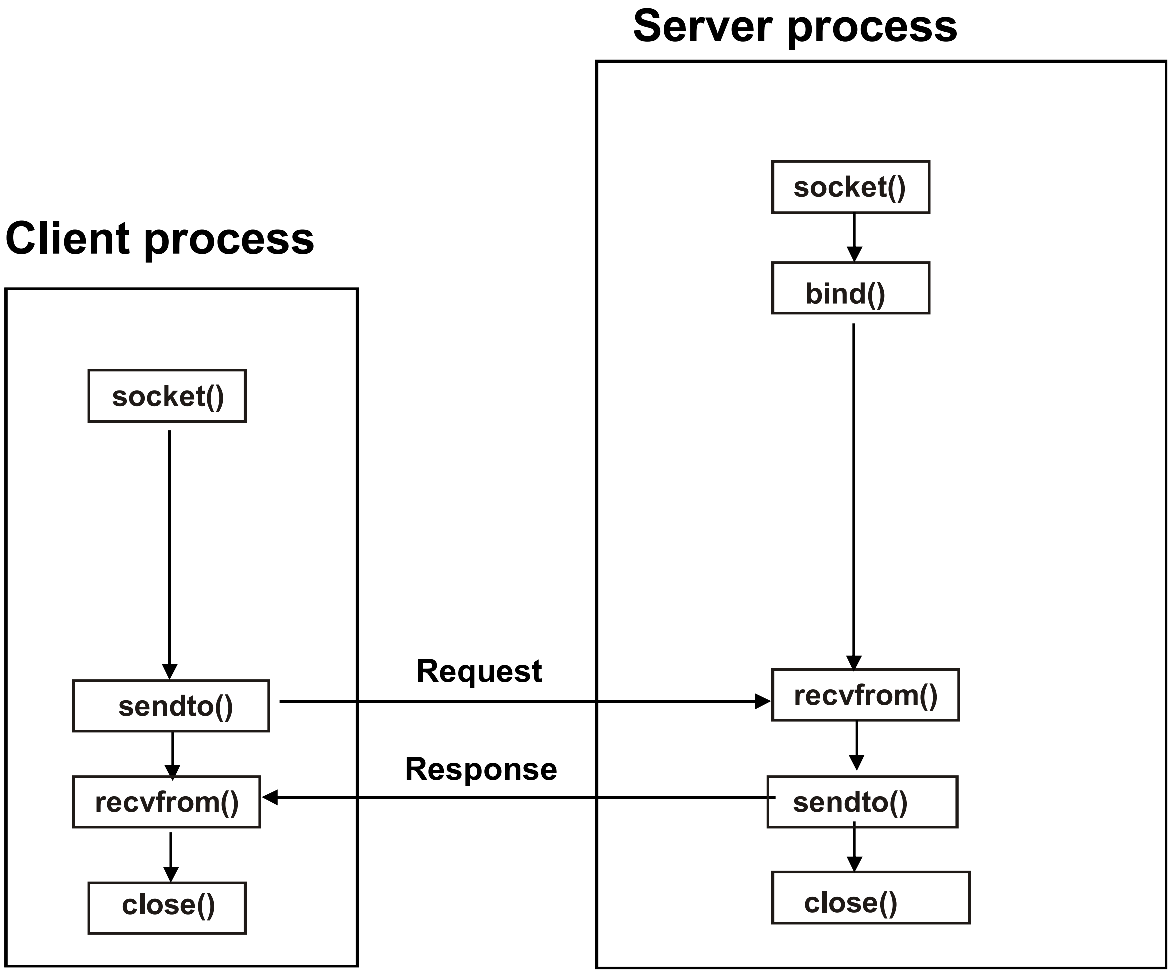
Cliente (TCP)

```
        // establish connection
        connect(sd, (struct sockaddr *) &server_addr, sizeof(server_addr));

        num[0]=5;
        num[1]=2;

        // send request
        send(sd, (char *) num, 2*sizeof(int), 0);

        // receive response
        recv(sd, &res, sizeof(int), 0);

        printf("Result = %d \n", res);

        close (sd);
        exit(0);
}
```

Example (UDP)


Host A
client
Kernel
add(5, 2)
Result = 7
Host B
server
5+2
Kernel
Network

Communication model



Server (UDP)

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

void main(void)
{
        int num[2];
        int s, res, clilen;
        struct sockaddr_in server_addr, client_addr;

        s =  socket(AF_INET, SOCK_DGRAM, 0);

        bzero((char *)&server_addr, sizeof(server_addr));
        server_addr.sin_family        = AF_INET;
        server_addr.sin_addr.s_addr   = INADDR_ANY;
        server_addr.sin_port          = htons(7200);

        bind(s, (struct sockaddr *)&server_addr,
                 sizeof(server_addr));
```

Server (UDP)

```
       clilen = sizeof(client_addr);

       while (1)
       {
               recvfrom(s, (char *) num, 2*sizeof(int), 0,
                         (struct sockaddr *)&client_addr, &clilen);

               res = num[0] + num[1];

               sendto(s, (char *)&res, sizeof(int), 0,
                       (struct sockaddr *)&client_addr,  clilen);
       }

}
```

Client (UDP)

```
void main(int argc, char *argv[])
{
        struct sockaddr_in server_addr, client_addr;
        struct hostent *hp;
        int s, num[2], res;

        if (argc != 2){
                printf("Use: client <server_address>\n");
                exit(0);
        }

        s =  socket(AF_INET, SOCK_DGRAM, 0);
        hp = gethostbyname (argv[1]);

        bzero((char *)&server_addr, sizeof(server_addr));
        memcpy (&(server_addr.sin_addr), hp->h_addr, hp->h_length);
        server_addr.sin_family = AF_INET;
        server_addr.sin_port   = htons(7200);
```

Client (UDP)

```
        bzero((char *)&client_addr, sizeof(client_addr));
        client_addr.sin_family          = AF_INET;
        client_addr.sin_addr.s_addr     = INADDR_ANY;
        client_addr.sin_port            = htons(0);

        num[0] = 5;
        num[1] = 2;

        sendto(s, (char *)num, 2 * sizeof(int), 0,
                (struct sockaddr *) &server_addr, sizeof(server_addr));

        recvfrom(s, (char *)&res, sizeof(int), 0, NULL, NULL);

        printf(“%d + %d = %d\n", num[0], num[1], res);

        close(s);
}
```

Problems in previous examples

- Error checking. Very important
- Data transfer problems
  - What happens if client is *little-endian* and server *big-endian*?
  - We have to deal with the problem of data representation. One possibility is to use these functions:

```
u_long  htonl(u_long  hostlong)
u_short htons(u_short hostshort)
u_long  ntohl(u_long  netlong)
u_short ntohs(u_short netshort)
```

  - We have to define the data representation and interchange format

Java *Sockets*

- Java package *java.net* allows to create UDP and TCP/IP *sockets*.

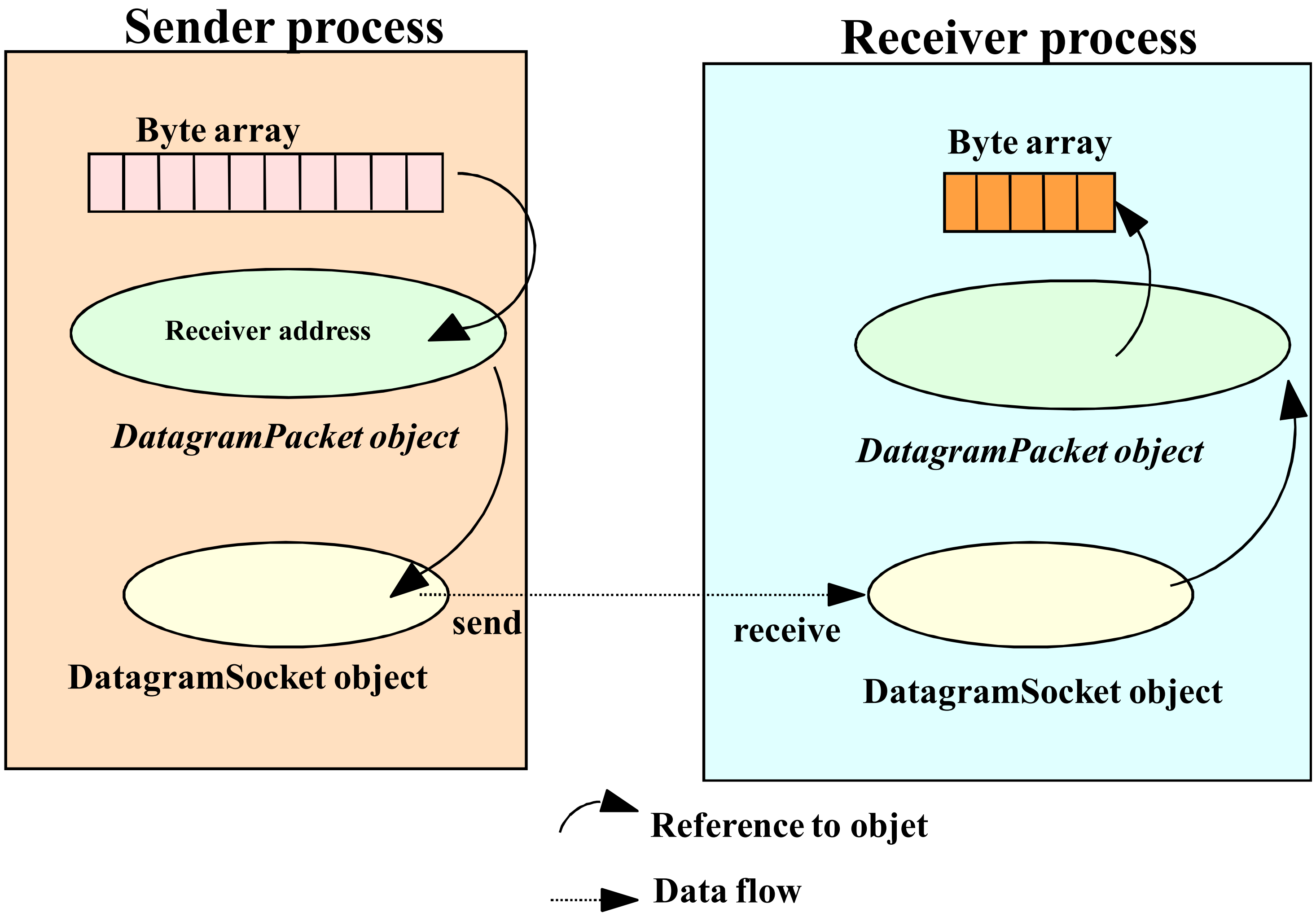
- *Datagram socket* classes:
  - *DatagramSocket*
  - *DatagramPacket*

- *Stream socket* classes:
  - *ServerSocket*
  - *Socket*

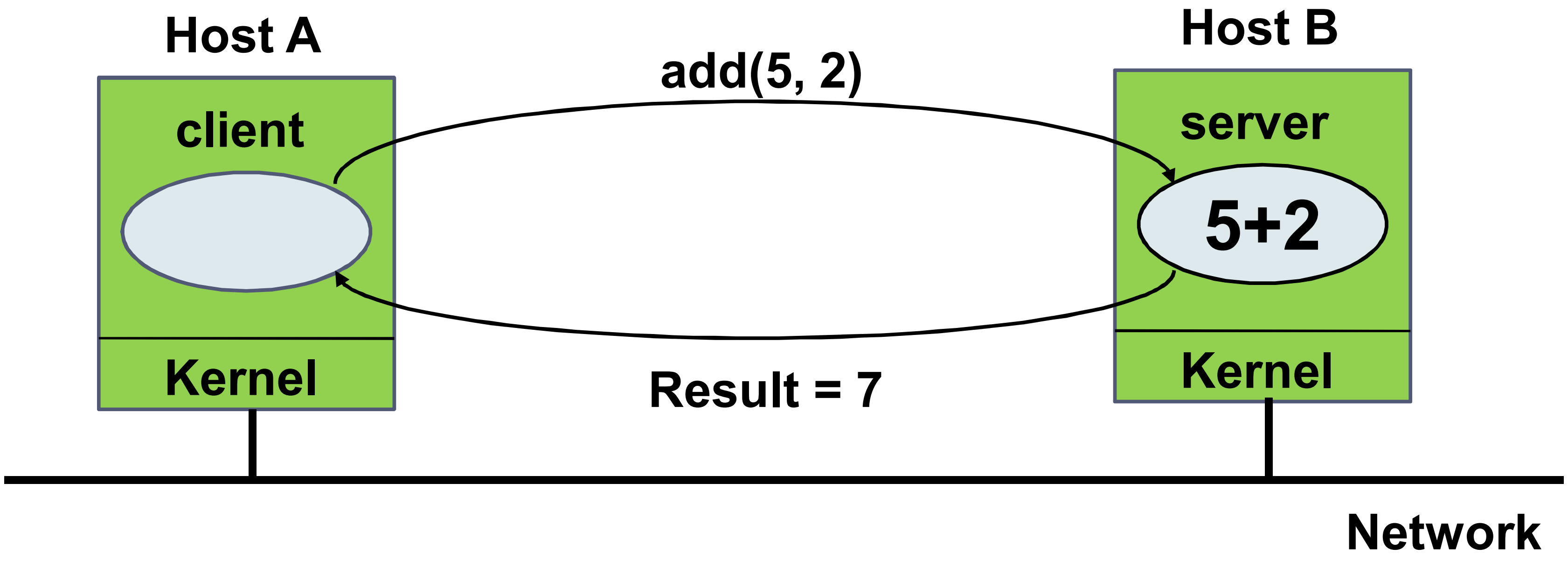
Datagram *Sockets*

- ***DatagramPacket***:
  - Implements an objet that allows to send and receive packets.
  - Constructor: *DatagramPacket.*
  - Methods: *getAddress, getPort*, ...
- ***DatagramSocket***:
  - Implements a *socket* that can be used to send and receive packets (datagramas).
  - Constructor: *DatagramSocket.*
  - Methods: *send, receive, close, setSoTimetout, getSoTimeout,...*

Datagram *Sockets*



Example (UDP)


Host A
client
Kernel
add(5, 2)
Result = 7
Host B
server
5+2
Kernel
Network

Client (UDP)

```
import java.lang.* ;
import java.io.* ;
import java.net.* ;
import java.util.* ;

public class client{
   public static void main ( String [] args)
   {

      byte bsend[] = new byte[100];
      byte brecv[] = new byte[100];

      InetAddress server_addr = null;
      DatagramSocket s = null;
      DatagramPacket in = null;
      DatagramPacket out = null;
      int res;    int num[] = new int[2];

      if (args.length != 1) {
          System.out.println("Use: client <host>");
          System.exit(0);
      }
```

Client (UDP)

```
try
    {
     // create client socket
     s = new DatagramSocket();

     // server address
     server_addr = InetAddress.getByName(args[0]);

     num[0] = 5;
     num[1] = 2;

     // pack data
     ByteArrayOutputStream baos = new ByteArrayOutputStream() ;
     ObjectOutputStream     dos = new ObjectOutputStream(baos);
     dos.writeObject(num);

     // obtain buffer (datagram)
     bsend = baos.toByteArray();

     // only one send
     out = new DatagramPacket (bsend, bsend.length, server_addr, 2500);
     s.send(out);
```

Client (UDP)

```
          // receive response datagram
          in = new DatagramPacket (brecv, 100);
          s.receive(in);

          // obtain buffer
          brecv = in.getData();

          // unpack
          ByteArrayInputStream bais = new ByteArrayInputStream(brecv) ;
          DataInputStream dis = new DataInputStream(bais);
          res = dis.readInt();
          System.out.println(“Received data " + res);
     }
     catch (Exception e)     {
          System.err.println("<<<<<Exception " + e.toString() );
          e.printStackTrace() ;
      }
   }
}
```

Server (UDP)

```
import java.lang.* ;
import java.io.* ;
import java.net.* ;
import java.util.* ;

public class server
{
   public static void main ( String [] args)
   {
      DatagramSocket s = null;
      DatagramPacket in, out;
      InetAddress client_addr = null;
      int client_port;
      byte brecv[] = new byte[100];
      byte bsend[] = new byte[100];
      int num[], res;

      try  {

         s   = new DatagramSocket(2500);
         in = new DatagramPacket(brecv, 100); // packet to receive the request
```

Server (UDP)

```
while (true) {
        // wait to receive
        s.receive(in);

        // obtain data
        brecv = in.getData();
        client_addr = in.getAddress();
        client_port = in.getPort();

       // unpack data
       ByteArrayInputStream bais = new ByteArrayInputStream(brecv);
       ObjectInputStream dis     = new ObjectInputStream(bais);

        num = (int[])dis.readObject();
        res = num[0] + num[1];
```

Server (UDP)

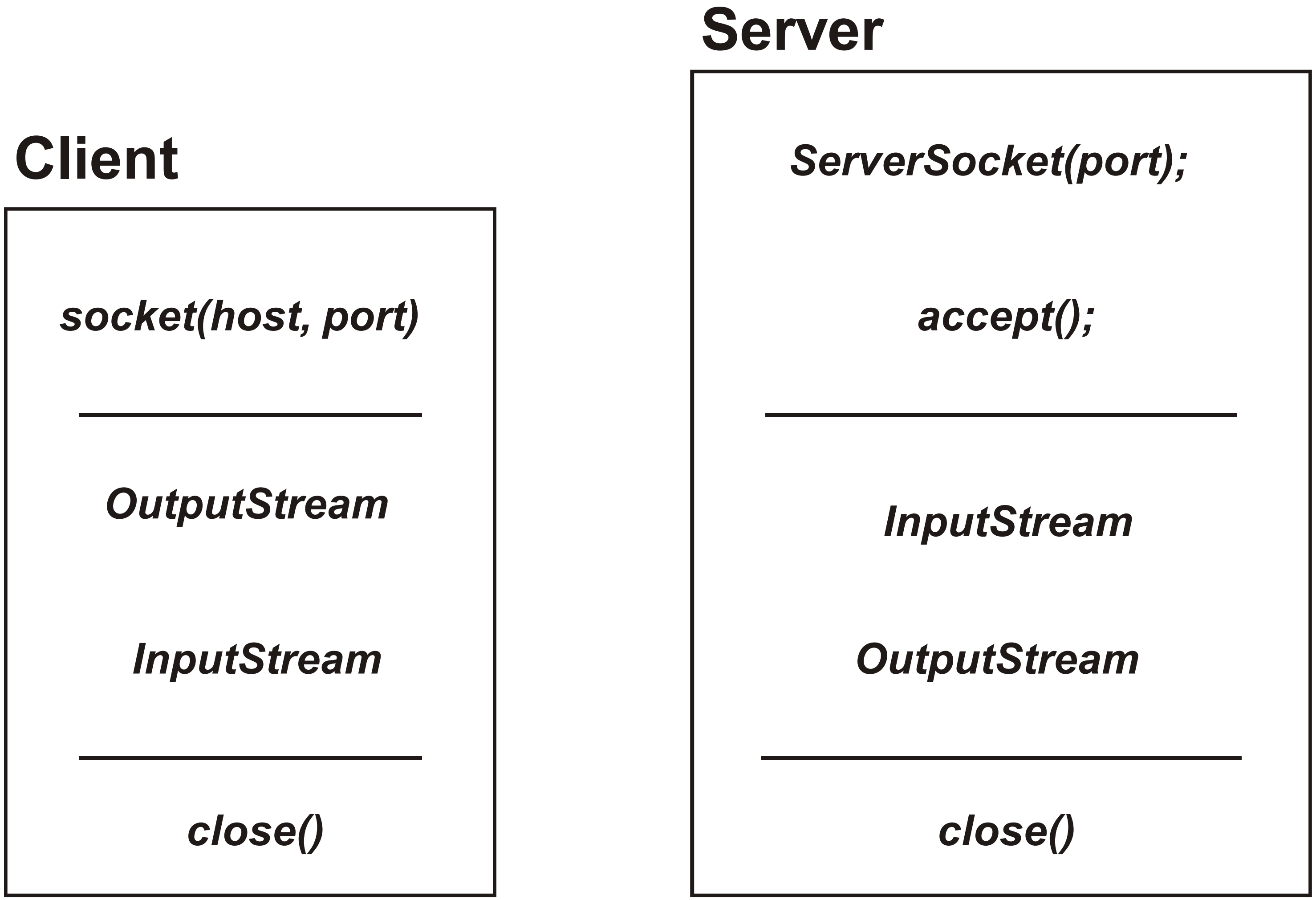
```
                 ByteArrayOutputStream baos =  new ByteArrayOutputStream();
                 DataOutputStream dos       =  new DataOutputStream(baos);

                 dos.writeInt(res);


                 bsend = baos.toByteArray();

                 out = new DatagramPacket ( bsend,
                                       bsend.length,client_addr,
                                       client_port);

                 s.send(out);
     }
  }
  catch(Exception e) {
               System.err.println(“Exception " + e.toString() );
               e.printStackTrace() ;
  }
 }
}
```

Stream sockets

- *Socket* class implements a *stream socket*
  - *Socket(InetAddress address, int port)*
  - *OutputStream getOutputStream()*
    - *flush*
  - *InputStream getInputStream()*
  - *void setSoTimeout(int wait_time)*

- *ServerSocket* class implements a *socket* to be used in servers to wait for connections
  - *Socket accept()*
  - *void close()*
  - *void setSoTimeout(int wait_time)*

Client-Server with *stream* sockets



Server

ServerSocket(port);

accept();

InputStream

OutputStream

close()

Client

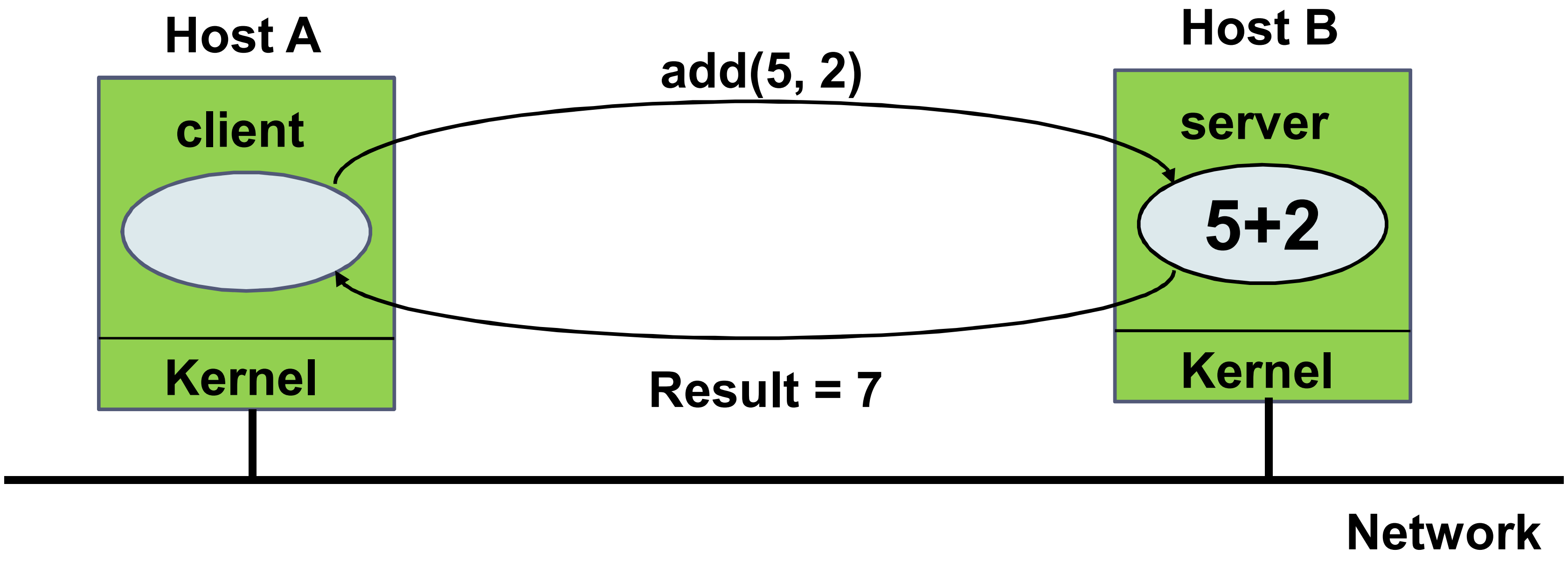
socket(host, port)

OutputStream

InputStream

close()

Example (TCP)


Host A
client
Kernel
add(5, 2)
Result = 7
Host B
server
5+2
Kernel
Network

Server (TCP)

```
import java.lang.* ;
import java.io.* ;
import java.net.* ;
import java.util.* ;

public class server
{
   public static void main (String [] args)
   {
      ServerSocket serverAddr = null;
      Socket sc = null;
      int num[];  // request
      int res;
      try {
         serverAddr = new ServerSocket(2500);
      }
      catch (Exception e){
         System.err.println("Error creating socket");
      }
```

Server (TCP)

```
while (true){
         try {
           // waiting for connection
           sc = serverAddr.accept();

           InputStream istream = sc.getInputStream();
           ObjectInput in      = new ObjectInputStream(istream);

           num = (int[]) in.readObject();
           res = num[0] + num[1];

           DataOutputStream ostream = new DataOutputStream(sc.getOutputStream());

           ostream.writeInt(res);
           ostream.flush();

           sc.close();
         }
         catch(Exception e) {
            System.err.println(“Exception " + e.toString() );
            e.printStackTrace() ;
         }
     }
   }
}
```

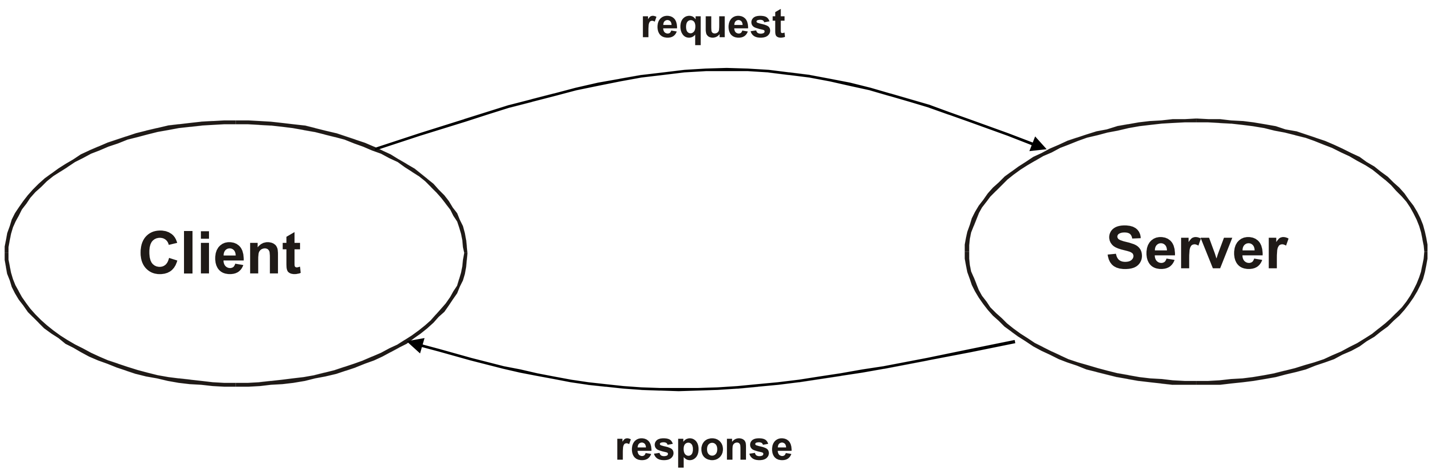
Client (TCP)

```
import java.lang.* ;
import java.io.* ;
import java.net.* ;
import java.util.* ;

public class client
{
   public static void main ( String [] args)
   {
      int  res;
      int num[] = new int[2];

      if (args.length != 1) {
         System.out.println("Use: client <host>");
         System.exit(0);
      }
      try {
                  // create connection
                 String host = args[0];
                 Socket sc = new Socket(host, 2500);
```

Client (TCP)

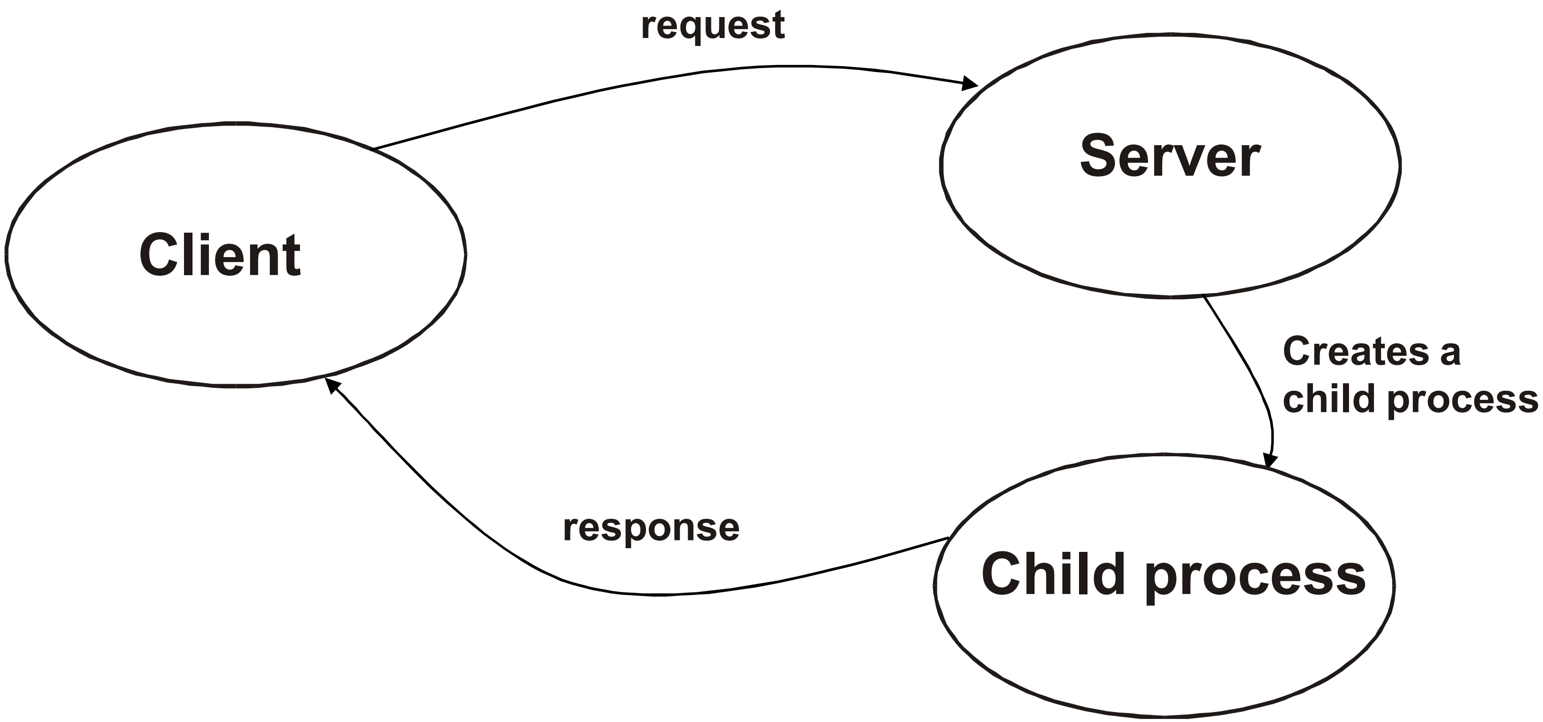
```
OutputStream ostream             = sc.getOutputStream();
         ObjectOutput s          = new ObjectOutputStream(ostream);
         DataInputStream istream = new DataInputStream(sc.getInputStream());

         num[0] = 5;   num[1] = 2; // prepare request

         s.writeObject(num);
         s.flush();

         res = istream.readInt();

         sc.close();
         System.out.println(“Result = " + res);
      }
      catch (Exception e){
         System.err.println(“Exception " + e.toString() );
         e.printStackTrace() ;
      }
   }
}
```

Sequential server model



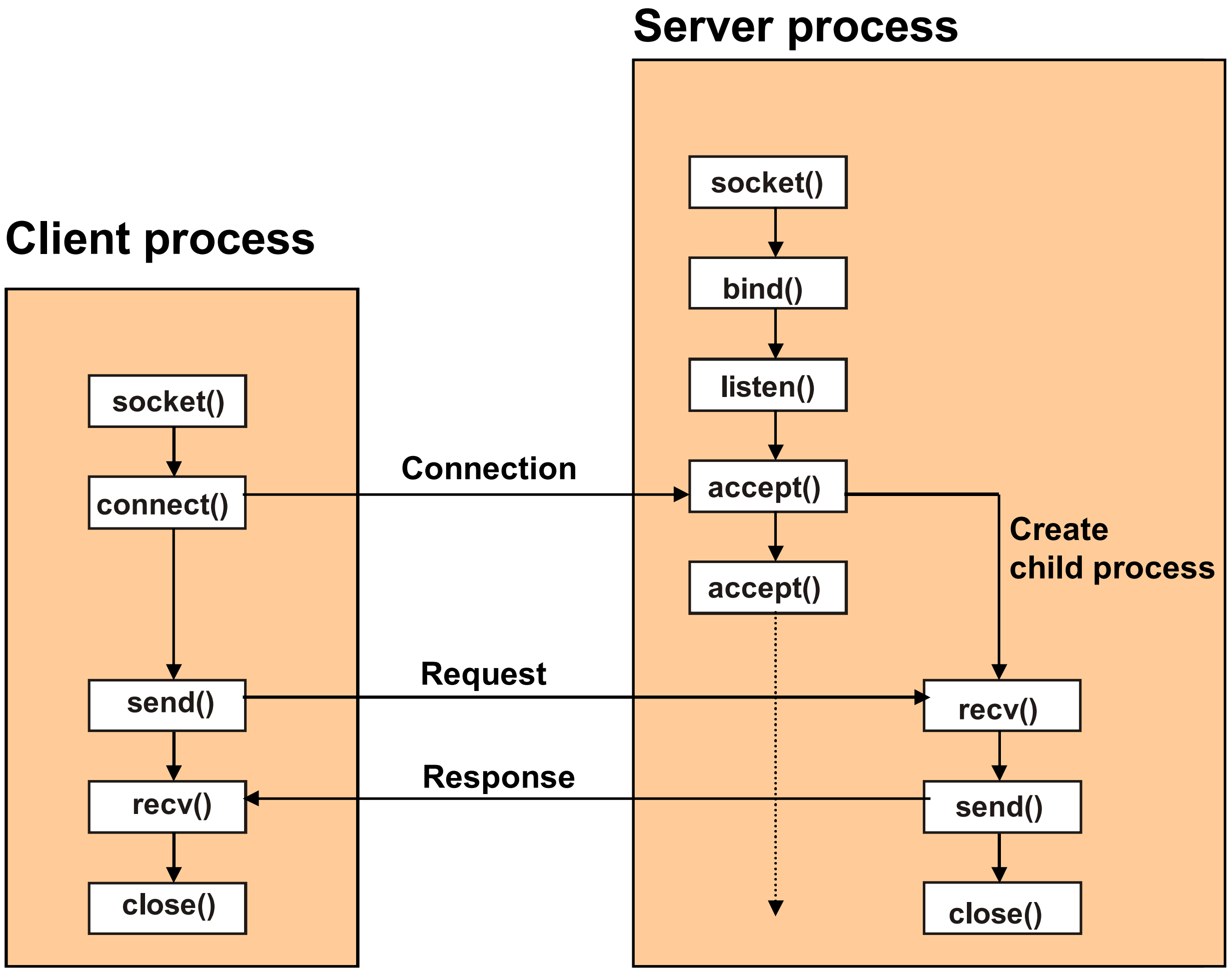
- The server processes requests sequentially.
- While it is serving a client it cannot accept other client requests.

Concurrent server model



- The server creates a child that manages the request and sends the response to the client.
- Several requests can be managed concurrently.

Concurrent servers with sockets



Types of concurrent servers

- A server process can create two types of processes:
  - Conventional processes (*fork*)
  - Lightweigth processes (thread).

Concurrent processes with *fork*

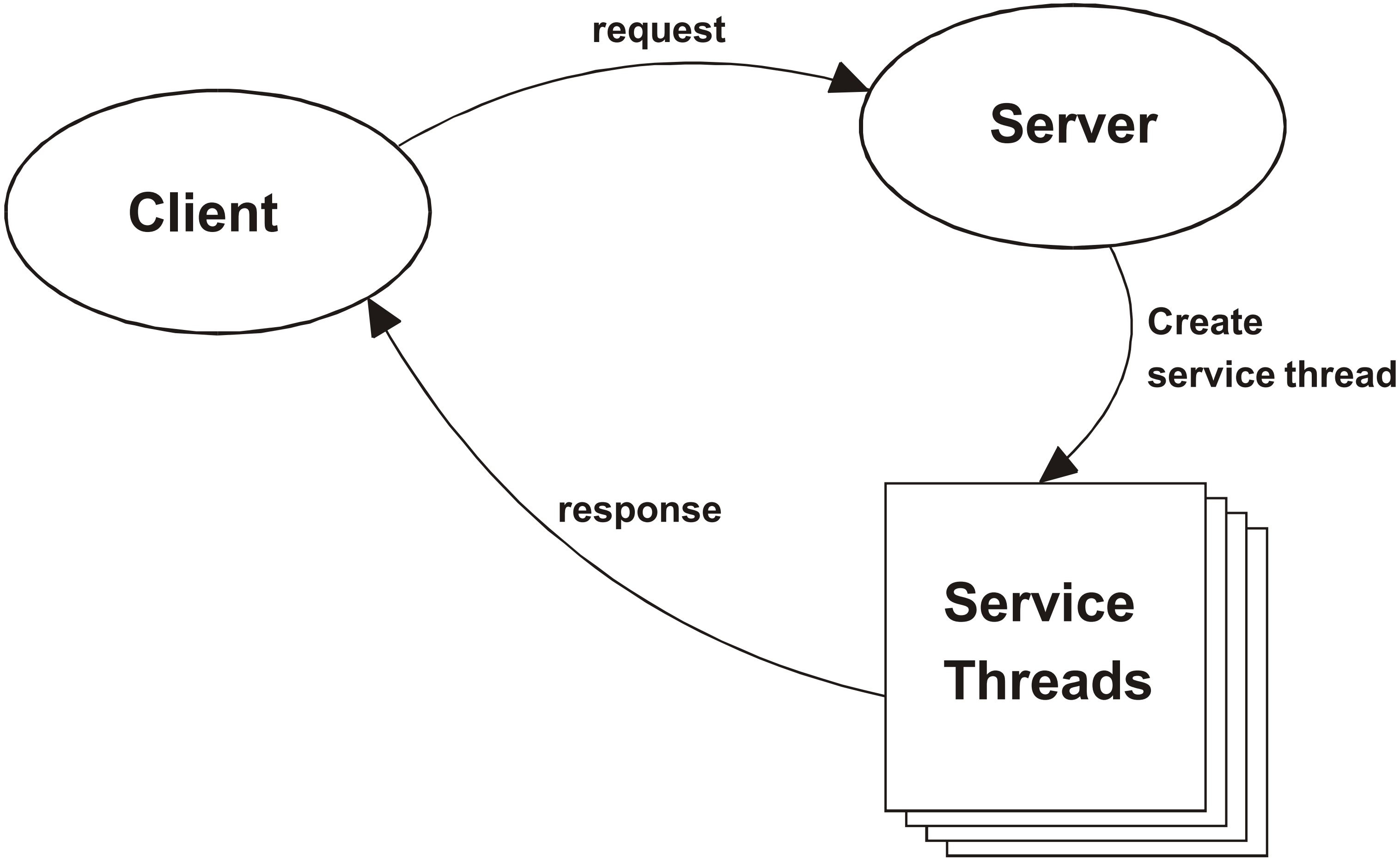
- The server creates a socket `s` and assigns it an address.
- The `main` code of the server is:

```
for(;;) {
      sd = accept(s, (struct sockaddr *)&  &client, &len);
      pid = fork();
      if (pid == -1)
               printf(“Cannot creat child\n”);
      else if (pid == 0)/* child process */
      {
               close(s);
               manage_request(sd);
               close(sd);
               exit(0);
      }
      close(sd);       /* parent */
}
```

Concurrent processes with *fork*

- In the previous model the parent process does not wait for the children to finish using *wait*.
  - Children processes remains in a *zombie* state when they die (they do not disappear).
  - To avoid the *zombie* state in children the parent can execute (only in UNIX System V and alike):
    - `signal(SIGCHLD, SIG_IGN);`

Concurrent processes with *threads*



Concurrent processes with *threads*

- The server creates a socket `s` and assigns it an address.
- The `main` code of the server is:

```
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
for(;;) {
     sd = accept(s, (struct sockaddr *) &client, &len);
     pthread_create(&thid, &attr, manage_request, &sd);
}
```

- The function the thread executes is:

```
void manage_request(int *s) {
   int s_local;

   s_local = *s;
   /* manage_request using descriptor s_local */
   close(s_local);
   pthread_exit(NULL);
}
```

Synchronization needed

- Previous solution is wrong because parent and child processes fight to access the descriptor returned by `accept`.
- It is needed to synchronize the actions using mutexes and conditional variables.
- `main` code:

```
for(;;) {
        sd = accept(s, (struct sockaddr *)&  &cliente, &len);
        pthread_create(&thid, &attr, manage_request, &sd);

        /* wait for the child to copy the descriptor */
        pthread_mutex_lock(&m);
        while(busy == TRUE)
                pthread_cond_wait(&m, &c);
        pthread_mutex_unlock(&m);
        busy = TRUE;
}
```

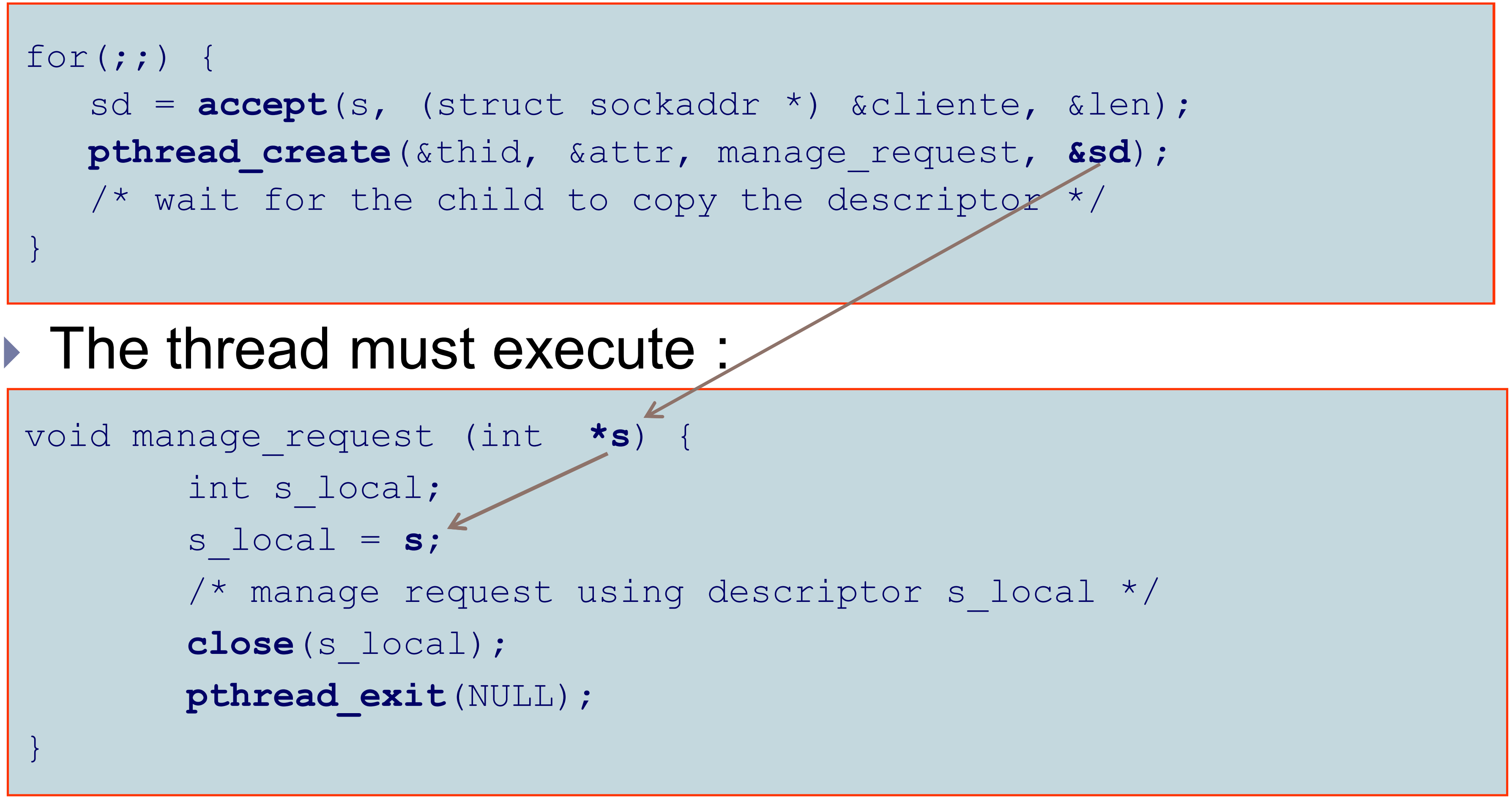
Synchronization needed

- The thread must execute:

```
void manage_request(int *s) {
        int s_local;

        pthread_mutex_lock(&m);
        s_local = *s;
        busy = FALSE;
        pthread_cond_signal(&c);
        pthread_mutex_unlock(&m);

        /* manage request using descriptor s_local */
        close(s_local);
        pthread_exit(NULL);
}
```

Synchronization needed

- main code :



```
for(;;) {
   sd = accept(s, (struct sockaddr *) &cliente, &len);
   pthread_create(&thid, &attr, manage_request, &sd);
   /* wait for the child to copy the descriptor */
}
```

- The thread must execute :

```
void manage_request (int  *s) {
        int s_local;
        s_local = *s;
        /* manage request using descriptor s_local */
        close(s_local);
        pthread_exit(NULL);
}
```

Concurrent server in Java (*streams*)

```
while (true){
         try {
           Socket client = serverAddr.accept();

            new ManageRequest(client).start();

         }
         catch(Exception e) {
            System.err.println(“Exception " + e.toString() );
            e.printStackTrace() ;
         }
     }
   }
}
```

Concurrent server in Java (*streams*)

```
class ManageRequest extend Thread {
       private Socket sc;

        ManageRequest(Socket s) {
              sc = s;
       }

       public void run() {

              // client code

       }
```

Client-Server applications design guide with sockets

- **Session**: Interaction between client and server
- Service protocol definition:
  - Service localization
  - Communication sequence among processes
  - Data representation and interpretation
- Types of servers
  - Stateful
  - Stateless

Client-Server applications design guide with sockets

1. Identify client and server
   - Client: active element, several
   - Server: passive element
2. Identify message types and message interchange sequence (requests and responses)
3. Choose the kind of socket
   1. Datagrams: stateless
   2. Streams:
      - One connection per session
      - One connection per request
4. Identify message format (data representation)
   - Independency (language, architecture, implementation, …)

Protocol comparison

	IP	**UDP**	**TCP**
Connection oriented?	No	No	Yes
Limit between messages?	Yes	Yes	No
Ack?	No	No	Yes
Timeout and retransmission?	No	No	Yes
Duplicates detection?	No	No	Yes
Sequencing?	No	No	Yes
Control flow?	No	No	Yes

Unit 6
Communications with sockets

Computer Architecture Area (ARCOS)

Distributed Systems

Bachelor in Informatics Engineering

Universidad Carlos III de Madrid