# Tema 2. Comunicación entre procesos

SCS – Sistemas Cliente/Servidor
4º informática

`http://ccia.ei.uvigo.es/docencia/SCS`

octubre 2008

# 2.1 Requisitos y alternativas

- Sistemas distribuidos requieren dar soporte a la comunicación entre procesos
  - Permite que 2 procesos colaboren en una tarea
- Comunicación entre procesos en una misma máquina
  - Ejemplos: pipes, memoria compartida, señales, semáforos, etc
- Comunicación entre procesos en máquinas distintas
  - Mediante **intercambio de mensajes** entre emisor y receptor/es
    - uno a uno (unicast)
    - uno a muchos (multicast)
- Esquema típico: mecanismo **petición-respuesta**
  - Distintos niveles de abstracción
  - Ejemplos: interfaz sockets, mecanismos RPC (llamada procedim. remoto)

## (a) Sincronización en mecanismos de paso de mensajes

- Conceptualmente todo mecanismos de paso de mensaje contará con las siguientes primitivas básicas:
  - ENVIAR:: proceso emisor transmite datos a un proceso receptor
  - RECIBIR:: proceso receptor acepta los datos de un emisor
  - INICIAR CONEXIÓN: (opcional, en sist. basados en conexión) un proceso indica que desea iniciar una conexión con otro
    - proceso activo, típicamente un cliente
  - ESPERAR CONEXIÓN: (opcional, en sist. basados en conexión) un proceso indica que está dispuesto a recibir conexiones
    - proceso pasivo, típicamente un servidor
  - ACEPTAR CONEXIÓN: (opcional, en sist. basados en conexión) un proceso acepta la comunicación con otro
    - proceso pasivo, típicamente un servidor

## Sincronización

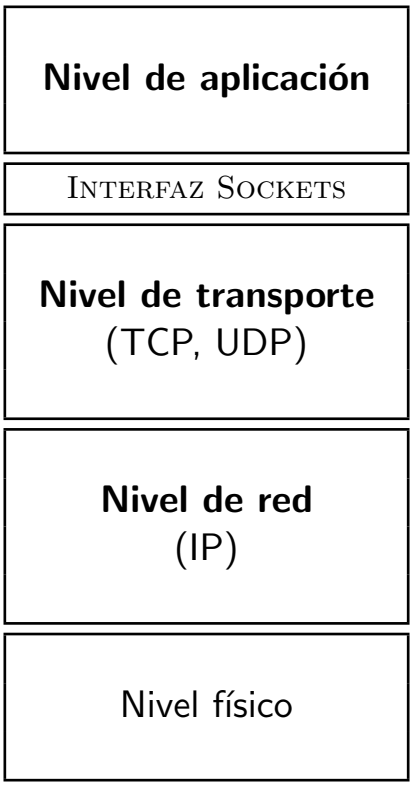
- Para asegurar el establecimiento de la conexión toda primitiva INICIAR CONEXIÓN debe tener la correspondiente ACEPTAR CONEXIÓN de una máquina que hubiera realizado esperar conexión
- Para asegurar la transferencia de un mensaje toda primitiva ENVIAR debe tener una primitiva RECIBIR en el otro extremo
- En esquemas petición-respuesta se intercambian 2 mensajes (petición+respuesta) con sus correspondientes pares ENVIAR y RECIBIR
- Dependiendo del modo en que se implementen las primitivas anteriores existen diversos esquemas de sincronización entre procesos
  - **Primitivas bloqueantes:** el proceso que las invoca queda bloqueado hasta que se verifica una condición (primitiva se completa)
  - **Primitivas no bloqueantes** el proceso que las invoca continúa su ejecución
- Ejemplos:
  - ENVIAR *bloqueante*: el proceso emisor queda en espera hasta que el receptor reciba correctamente el mensaje (receptor completa la correspondiente primitiva RECIBIR)
  - ENVIAR *no bloqueante*: el proceso emisor "coloca" dos datos en una zona de envío (buffer) y continúa su ejecución
  - RECIBIR *bloqueante*: el proceso receptor queda en espera hasta que haya datos disponibles del emisor que corresponda (emisor completa la correspondiente primitiva RECIBIR
  - RECIBIR *no bloqueante*: el proceso receptor recoge los datos disponibles, si no los hay lo indica y continua su ejecución
  - INICIAR CONEXIÓN *bloqueante*: el proceso que inicia la conexión quede a la espera hasta que el otro extremo invoque el correspondiente ACEPTAR CONEXIÓN
  - ACEPTAR CONEXIÓN *bloqueante*: hace que el proceso quede en espera hasta que desde el otro extremo se invoque un INICAR CONEXIÓN

- Casos típicos:
  - **Comunicaciones síncronas**: ENVIAR y RECIBIR bloqueantes
    - se garantiza sincronización entre los mensajes enviados y recibidos
    - simplifica la programación, pero penalizan el rendimiento
  - **Comunicaciones asíncronas**: ENVIAR no bloqueante
    - evitan bloqueos indefinidos y ofrecen mejor rendimiento
    - RECIBIR puede ser bloqueante (lo más usual) o no bloqueante
      - ◇ implementación compleja con ENVIAR y RECIBIR no bloqueantes
    - **Diseño habitual**: ubicar las operaciones bloqueantes en un hilo de ejecución (*thread*) propio
      - ◇ Ejemplo: servidores multihilo, un hilo para esperar y atender cada petición

# 2.2 Interfaz de *Sockets*

- **Socket:** interfaz de programación (API) sobre el nivel de transporte
  - abstracción que facilita al programador el acceso a los servicios y recursos del nivel de transporte
  - ofrece un servicio punto a punto entre emisor y receptor
- Estándar de facto: parte del S.O. o de las librerías del sistema
  - inicialmente interfaz en C para sistemas Unix
  - adaptado a otros S.O. y lenguajes

**Pila de protocolos TCP/IP**




- **Nivel de transporte:** ofrece servicio de envío de datos extremo a extremo
  - hace uso de servicos del nivel de red (IP)
- **Protocolo TCP:** ofrece un servicio orientado a conexión, fiable, ordenado, con control de flujo y errores
  - requiere establecimiento previo de una conexión entre ambos extremos
  - ofrece un flujo permanente entre los extremos en ambas direcciones ($\approx$ tubería punto a punto)
  - controla la recepción en orden, completa y sin errores, gestionando el reenvío de paquetes perdidos
- **Protocolo UDP:** ofrece un servicio no orientado a conexión, no fiable y sin control de flujo y errores
  - cada mensaje UDP (datagrama) es independiente y se trata de forma aislada
  - no se garantiza la entrega de los datagramas enviados ni que estos lleguen en orden

- **Puertos:** identificadores usados para asociar los datos entrantes a un proceso concreto de la máquina
  - usados tanto en TCP y UDP
  - números de 16 bits
  - 0-1023: reservados por convenio ("puertos bien conocidos" )
    - ◇ asignados a los servidores de servicios básicos

| Servicio | Puerto | Servicio | Puerto |
|---|---|---|---|
| ftp | 21 | dns | 53 |
| telnet | 22 | http | 80 |
| ssh | 23 | pop3 | 110 |
| smtp | 25 | https | 443 |

  - 1024-65535: uso libre
    - ◇ son los usados a los clientes al establecer conexiones
    - ◇ suelen asignarse de forma aleatoria

## Direccionamiento

- Para comunicarse con otro proceso usando sockets debe conocerse
  1. *Dirección IP* (32 bits) de la máquina donde se ejecuta el proceso
     - Alternativamente: su nombre para consultar servicio de nombre DNS (traducción dominio-IP)
  2. *Núm. de puerto* (TCP ó UDP) que utiliza el procesa en su máquina

  Conclusión: Interfaz de Sockets no ofrece transparencia de localización

# 2.2.1 Primitivas básicas del interfaz Sockets

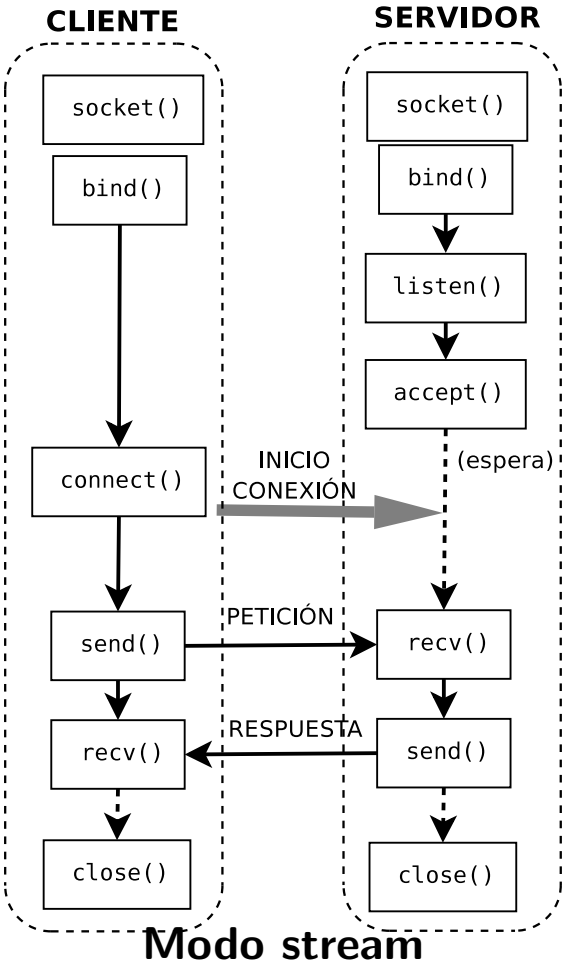
## (a) Modos de operación

- El interfaz de Sockets soporta dos modos de operación
  - **modo stream (modo TCP)**: establece una conexión entre los dos extremos que permite enviar flujos de bytes en ambas direcciones
    - Existen dos papeles definidos implicitamente:
      **sockets de servidor:** esperan recibir conexiones
      **sockets de tráfico:** { inician conexiones (en los clientes); envían/reciben datos (en clientes y servidor) }
    - Funciones: { `socket()`, `bind()`; `listen()`, `accept()`; `connect()`; `send()`, `recv()` }
  - **modo datagrama (modo UDP)**: no se establece conexión, cada mensaje ($datagrama$) es independiente
    - Funciones: { `socket()`, `bind()`; `sendto()`, `revfrom()` }
- En ambos casos se ofrece al programador un interfaz análogo al interfaz de acceso a ficheros del S.O.
  - lectura $\longrightarrow$ recepción
  - escritura $\longrightarrow$ envío
- **Importante:** la información intercambiada entre los extremos depende de los procesos
  - La interfaz socket no impone la sintaxis o semántica de los mensajes intercambiados
  - Los procesos deben acordar y gestionar el formato de los mensajes
  - Los procesos deben manejar explícitamente los tipos de datos y realizar los cambios de codificación necesarios
    - Cliente y servidor deben usar el mismo esquema para empaquetar y aplanar los datos ($marshaling$)

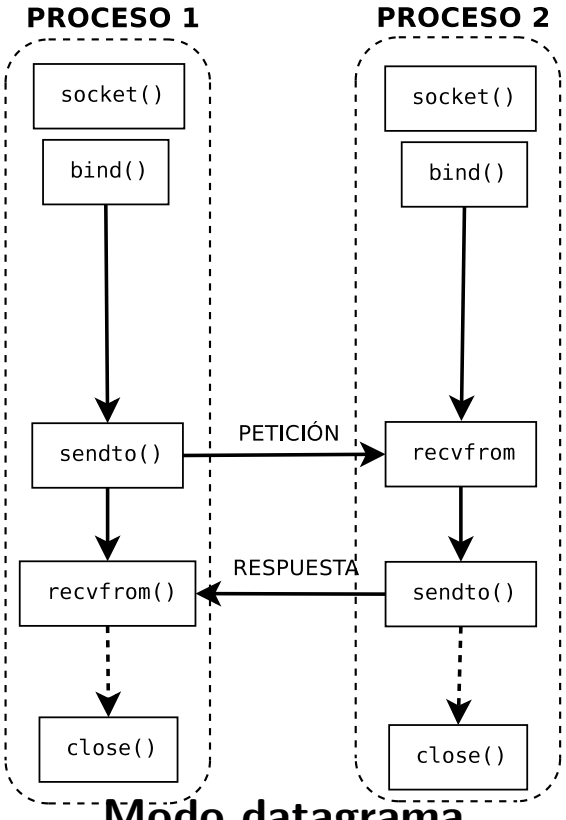
## (b) Primitivas básicas

- **socket()**: crea un socket (de tipo *stream* o *datagrama*)
- **bind()**: asocia la dirección local (IP+nº puerto) al socket
  - Sólo es obligatorio asignar dir. local (nº puerto) en servidores en modo stream
  - En clientes si no se asigna se hará automáticamente en su primer uso [connect() o sendto()]
- **listen()**: pone un socket de servidor en modo espera [no bloqueante]
- **connect()**: establece la conexión con una máquina remota
  - exige indicar dirección del proceso servidor destino (IP+nº puerto)
- **accept()**: indica que se ha recibido y aceptado una conexión de un cliente (tomada de la cola de espera) y genera un nuevo socket para que el servidor intercambie información con el cliente [bloqueante]
- **send(), sendto()**: envío de datos (stream, datagrama) [no bloqueante]
- **recv(), revfrom()**: recepción de datos (stream, datagrama) [bloqueante]

## (c) Esquema de funcionamiento




Modo stream




Modo datagrama

## Ventajas uso de Sockets

- Rendimiento y mínima sobrecarga (control total sobre los datos enviados)
- Máxima flexibilidad a la hora de desarrollar aplicaciones (control total sobre los datos enviados)
- Disponibilidad: interfaz socket disponible en múltiples S.O. y lenguajes de programación

## Inconvenientes uso de Sockets

- Necesidad de realizar un manejo explícito de los datos (formateo, codificación, etc)
  - Requiere acordar/negociar los formatos de transmisión
  - Requiere aplanar/desaplanar los datos estructurados antes de ser enviados/recibidos (*marshaling* o serialización)
- Escasa transparencia de localización
  - Necesario manejar explicitamente dir. IP, puertos, protocolo, etc
- Complejidad y dificultad de programación

# 2.2.2 Uso de sockets en C

## (a) Creación y nombrado de sockets

1. Creación de sockets: función **socket()**
   - Crea un socket del tipo especificado (stream o datagrama)
   - Devuelve un entero que sirve como descriptor del socket creado.

```
#include <sys/socket.h>
int socket(int dominio, int tipo, int protocolo);
```

   - `dominio`: familia de protocolos
     - `PF_UNIX`: comunic. procesos misma máquina
     - `PF_INET`: protocolos Internet [TCP ó UDP]
     - otros: `PF_INET6`,`PF_IPX`,`PF_NETLINK`...
   - `tipo`: tipo de conexión
     - `SOCK_STREAM` (conexión TCP, streams)
     - `SOCK_DGRAM` (conexión UDP, datagramas)
     - `SOCK_RAW` (acceso a la red a bajo nivel)
   - `protocolo`: identificador del protocolo (0: protocolo por defecto)

2. Asignación dirección local, función **bind()**
   - Asocia una dirección local a un socket previamente creado.
   - Devuelve 0 si no hubo errores.

   **Nota:** en el caso de los clientes se puede omitir **bind()**, se asignará la dir. local en primer uso (**connect()** o **sendto()**) del socket

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen);
```

   - `sockf`: descriptor del socket creado con la función socket()
   - `my_addr`: información con la dirección de la máquina propia
   - `adrrlen`: tamaño de la estructura `my_addr` (`size_of(my_addr)`)

   **Nota:** Con sockets de la familia `AF_INET` se usa `struct sockaddr_in`

```
#include <netinet/in.h>
struct sockaddr_in {
  short          sin_family; /* familia de la dirección: AF_INET */
  u_short        sin_port;   /* num. de puerto (0 asigna uno aleatorio )*/
  struct in_addr sin_addr;   /* dir. de internet (INADDR_ANY asigna una de la propia
  char           sin_zero[8];/* campo de 8 ceros (bytes de relleno)*/
};
struct in_addr {
   u_long     s_addr; /* 32 bits de la dir. IP*/
};
```

## (b) Sockets en modo stream ((TCP)

## Funciones del cliente

1. Establecimiento de conexión: función **connect()**
   - Usa el socket creado para establecer una conexión a una máquina especificada por una direccion IP + nº de puerto

```
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen);
```

   - `sockfd`: descriptor del socket a usar en la conexión
   - `serv_addr`: información con la dirección del servidor de destino (IP+puerto)
   - `addrlen`: tamaño de la estructura `serv_addr` (`size_of(serv_addr)`)

**Ejemplo:** socket() + bind() + connect()

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

#include <sys/types.h>
#include <strings.h>
...

int conexion, resultado;
struct sockaddr_in dir_propia, dir_remota;

/* creación de un socket TCP de Internet (stream) */
conexion = socket(PF_INET, SOCK_STREAM, 0);

/* bind() con direccion local (puerto aleatorio) [no imprescindible en clientes] */
memset(&dir_remota, 0, sizeof(struct sockaddr_in)); /* inicializar a 0 */
dir_propia.sin_family = AF_INET;
dir_propia.sin_addr.s_addr = htonl(INADDR_ANY); /* asina una dir. local de la maquina*/
dir_propia.sin_port = 0;  /* asigna un puerto libre al azar*/
resultado = bind(conexion, (struct sockaddr *) &dir_propia, sizeof(dir_propia);

/* connect() a puerto 110 de 193.147.87.47 */
memset(&dir_remota, 0, sizeof(struct sockaddr_in)); /* inicializar a 0 */
dir_remota.sin_family = AF_INET;
dir_remota.sin_addr.s_addr = inet_addr("193.147.87.47");
dir_remota.sin_port = htons(110);
resultado = connect(conexion, (struct sockaddr *) &dir_remota, sizeof(dir_remota);
```

## Manejo de direcciones IP textuales y nombres de dominio

- Conversión cadenas con dir. IP en formato `xxx.xxx.xxx.xxx`

```
#include <arpa/inet.h>
struct in_addr inet_addr(const char *cp)
```

- Traducción de nombres de dominio

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

struct hostent {
  char *h_name;                 /* Official name of host.  */
  char **h_aliases;             /* Alias list.  */
  int h_addrtype;               /* Host address type.  */
  int h_length;                 /* Length of address.  */
  char **h_addr_list;           /* List of addresses from name server.  */
  #define h_addr  h_addr_list[0]  /* Address, for backward compatibility.  */
};
```

  - Para la traducción se consulta el fichero `/etc/hosts` y/o el servidor DNS
  - En `h_addr_list` se devuelve un array con las direcciones asociadas

### Ejemplo: conexión con nombre de máquina remota

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

#include <sys/types.h>
#include <string.h>
...
int conexion, resultado;
struct sockaddr_in dir_remota;
struct hostent * datos_host;

/* creación de un socket TCP de Internet (stream) */
conexion = socket(PF_INET, SOCK_STREAM, 0);

/* consulta del nombre ccia.ei.uvigo.es */
datos_host = gethostbyname("ccia.ei.uvigo.es");
memset(&dir_remota, 0, sizeof(struct sockaddr_in));  /* inicializar a 0 */
dir_remota.sin_family = datos_host->h_addrtype;      /* copiar familia   */
memcpy(&dir_remota.sin_addr.s_addr,                  /* copiar direccion IP*/
       datos_host->h_addr_list[0], datos_host->h_length);
dir_remota.sin_port = htons(80);                      /* conexion al puerto 80 */

resultado = connect(conexion, (struct sockaddr *) &dir_remota, sizeof(dir_remota));
```

## Funciones del servidor

1. Modo escucha: función **listen()**
   - El servidor indica que desea aceptar conexiones entrantes a través del socket y estable el límite de conexiones entrantes que permite en cola
     - **Nota:** antes de poner un socket en espera de conexión es imprescindible haber realizado **bind** para indicar el puerto de escucha

```
#include <sys/socket.h>
int listen(int sockfd, int backlog);
```

   - `sockfd`: descriptor del socket servidor a poner en espera
   - `backlog`: nº máximo de peticiones a mantener en la cola de espera
2. Recepción de conexiones: función **accept()**
   - Toma la primera petición de conexión de la cola creada en la llamada **listen()**
   - Crea un nuevo socket que se usará para comunicarse con el cliente
     - Devuelve un entero que funciona como descriptor del socket creado
     - Rellena una estructura `struct sockaddr` con información sobre la dir. del cliente conectado
     - El socket de escucha no es alterado y queda disponible para aceptar nuevas peticiones

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

   - `sockfd`: Descriptor del socket en modo escucha que recibió la petición de conexión
   - `addr`: Puntero a la estructura donde dejará la información sobre la dirección de la máquina que realizó la petición de conexión
   - `addrlen`: Tamaño de esa dirección

## Envio/recepción (modo stream TCP)

1. Envio de mensajes: función **send()**
   - Envía el contenido de un buffer al otro extremo de un socket conectado
   - Devuelve el nº de bytes realmente enviados ó -1 si hubo algún error

```
#include <sys/socket.h>
int send(int sockfd, const void *msg, size_t len, int flags);
```

   - `sockfd`: Descriptor del socket usado para el envio
   - `msg`: Puntero al buffer donde están los datos a enviar
   - `len`: Tamaño (en bytes) del buffer con los datos a enviar
   - `flags`: Modificadores

   **Ejemplo**:

```
char * mensaje = "hola";
enviados = send(conexion, mensaje, strlen(mensaje)+1,0);
```

2. Recepción de mensajes: función **recv()**
   - Queda a la espera de recibir datos del otro extremo de la conexión y los escribe en el buffer indicado
   - Por defecto es una operación bloqueante, el flag `MSG_DONTWAIT` habilita el modo no bloqueante
   - Devuelve la longitud del mensaje recibido o -1 si hubo un error

```
#include <sys/socket.h>
int recv(int sockfd, void *buf, size_t len, int flags);
```

   - `sockfd`: Descriptor del socket usado para la recepción
   - `buf`: Puntero al buffer donde se escribirán los datos leidos
   - `len`: Capacidad del buffer (nº máximo de bytes que caben en el buffer)
   - `flags`: Modificadores

   **Ejemplo**:

```
char * buffer = malloc(512);
leidos = recv(conexion, buffer, 512, 0);
```

# (d) Esquema típico para clientes y servidores en modo stream

## esquema clientes

```
...
int conexion;
int resultado;
struct sockaddr_in dir_remota;
struct hostent * datos_host;

/* CREACION de un socket TCP  */
conexion = socket(PF_INET, SOCK_STREAM, 0);

/* CONSULTA NOMBRE servidor y rellenar dir remota*/
datos_host = gethostbyname(...<nombre>...);
memset(&dir_remota, 0, sizeof(struct sockaddr_in));
dir_remota.sin_family = datos_host->h_addrtype;
memcpy(&dir_remota.sin_addr.s_addr,datos_host->h_addr_list[0],datos_host->h_length);
dir_remota.sin_port = htons(...);   /* indicar PUERTO REMOTO */

/* estabecer CONEXION */
resultado = connect(conexion,(struct sockaddr *) &dir_remota,sizeof(dir_remota));
...
while(! final_dialogo) { /* dialogo con servidor */
   ...
   peticion = crear_peticion(....);
   enviados = send(conexion,peticion,tam_peticion,0);    /* ENVIO     */
   ...
   recibidos = recv(conexion,respuesta,tam_respuesta,0); /* RECEPCION */
   ...
   procesar_respuesta(respuesta, ....);
   ...
}
close(conexion);  /* CERRAR conexion */
...
```

## **esquema servidor** (un único cliente a la vez)

```
...
int conexion, conexion_cliente;
int resultado, tam_dir_cliente;
struct sockaddr_in dir_propia, dir_cliente;

/* CREACION de un socket TCP  */
conexion = socket(PF_INET, SOCK_STREAM, 0);
...
/* BIND local (establecer PUERTO) */
memset(&dir_propia, 0, sizeof(struct sockaddr_in));
dir_propia.sin_family = AF_INET;
dir_propia.sin_addr.s_addr = htonl(INADDR_ANY);
dir_propia.sin_port = htons(...);       /* establecer PUERTO de escucha*/
resultado = bind(conexion,(struct sockaddr *) &dir_propia,sizeof(dir_propia));
...
/* poner en MODO ESCUCHA (tamano cola = 20) */
resultado = listen(conexion, 20);
...
while(1) {  /* Bucle principal aceptando conexiones*/
   /* ACEPTAR conexion del cliente */
   tam_dir_cliente = sizeof(dir_cliente);
   conexion_cliente = accept(conexion,(struct sockaddr *) &dir_cliente,&tam_dir_cliente);
   while (! fin_dialogo) {  /* Bucle de DIALOGO con cliente */
      recibidos= recv(conexion_cliente,peticion, tam_peticion,0);  /* RECEPCION */
      ...
      respuesta= procesar_peticion(peticion);
      ...
      enviados= send(conexion_cliente,respuesta,tam_respuesta,0); /* ENVIO     */
      ...
   }
   close(conexion_cliente);  /* CERRAR conexion con cliente */
}
close(conexion);   /* CERAR conexion */
...
```

## esquema servidor multiproceso

```
...
int conexion, conexion_cliente;
int resultado, tam_dir_cliente, pid;
struct sockaddr_in dir_propia, dir_cliente;

/* CREACION de un socket TCP  */
conexion = socket(PF_INET, SOCK_STREAM, 0);
...
/* BIND local (establecer PUERTO) */
memset(&dir_propia, 0, sizeof(struct sockaddr_in));
dir_propia.sin_family = AF_INET;
dir_propia.sin_addr.s_addr = htonl(INADDR_ANY);
dir_propia.sin_port = htons(...);       /* establecer PUERTO de escucha*/
resultado = bind(conexion,(struct sockaddr *) &dir_propia,sizeof(dir_propia));
...
/* poner en MODO ESCUCHA (tamano cola = 20) */
resultado = listen(conexion, 20);
...
while(1) {  /* Bucle principal aceptando conexiones*/
   /* ACEPTAR conexion del cliente */
   tam_dir_cliente = sizeof(dir_cliente);
   conexion_cliente = accept(conexion,(struct sockaddr *) &dir_cliente,&tam_dir_cliente);

   /* Crear un proceso hijo para procesar la conexion de cada cliente */
   pid = fork();
   if (pid == -1) { /* error */
      ...
   }
   else if (pid == 0) { /* PROCESO HIJO */
      close(conexion);  /* liberar conexion */
      while (! fin_dialogo) {  /* Bucle de DIALOGO con cliente */
         recibidos = recv(conexion_cliente,peticion,tam_peticion,0);  /* RECEPCION */
         ...
         respuesta = procesar_peticion(peticion);
         ...
         enviados = send(conexion_cliente,respuesta,tam_respuesta,0); /* ENVIO     */
         ...
      }
      close(conexion_cliente);  /* CERRAR conexion con cliente */
      exit(0);  /* finaliza proceso hijo */
   }
   else {   /* Proceso PADRE */
      close(conexion_cliente);  /* liberar conexion con cliente */
   }
}
close(conexion);   /* CERAR conexion */
...
```

## (d) Sockets en modo datagrama ((UDP)

- No se establece conexión, cada mensaje se trata de forma aislada
  - Los sockets se crean especificando `SOCK_DGRAM` como `tipo`
- En cada mensaje enviado/recibido se especifica la dirección de origen/destino (IP + nº puerto).

1. Envio de datagramas UDP: función **sendto()**
   - Envia un mensaje a la máquina remota indicada (IP + puerto)
   - Devuelve el nº de bytes enviados, o -1 si hubo algún error
   - Si el mensaje es demasiado largo para ser enviado a través del protocolo inferior, se devuelve el error `EMSGSIZE` y el mensaje no es transmitido

```
#include <sys/socket.h>
int  sendto(int sockfd, const void *msg, size_t len, int flags,
            const struct sockaddr *to, socklen_t tolen);
```

   - `sockfd`: Descriptor del socket usado para el envio
   - `msg`: Puntero al buffer donde están los datos a enviar
   - `len`: Tamaño del buffer con los datos a enviar (datagrama)
   - `flags`: Modificadores
   - `to`: Información sobre la dirección de la máquina a la que se envia el datagrama
   - `tolem`: Tamaño de la dirección
2. Recepción de datagramas UDP: función **recvfrom()**
   - Recibe un mensaje e indica la dir. de máquina remota emisora (IP + puerto)
   - Devuelve el nº de bytes recibidos, o -1 si hubo algún error
   - Si un mensaje es demasiado largo para caber en el buffer de recepción, los bytes que sobran pueden ser descartardos
   - Por defecto es una operación bloqueante, el flag `MSG_DONTWAIT` habilita el modo no bloqueante

```
#include <sys/socket.h>
int  recvfrom(int  sockfd, void *buf, size_t len, int flags,
              struct sockaddr *from, socklen_t *fromlen);
```

   - `sockfd`: Descriptor del socket usado para la recepción
   - `buf`: Puntero al buffer donde se escribirán los datos leidos (datagrama)
   - `len`: Núm. máximo de bytes que caben en el buffer
   - `flags`: Modificadores
   - `from`: Puntero a la estructura donde dejará la información de la dirección de la máquina desde la que se envió el datagrama
   - `fromlen`: Tamaño de la dirección

# (e) Esquema típico para clientes y servidores en modo datagrama

## esquema clientes

```
...
int conexion;
int resultado;
struct sockaddr_in dir_propia, dir_remota, dir_remota2;
struct hostent * datos_host;

/* CREACION de un socket UDP  */
conexion = socket(PF_INET, SOCK_DGRAM, 0);
...
/* BIND con dirección local */
memset(&dir_local, 0, sizeof(struct sockaddr_in));
dir_propia.sin_family = AF_INET;
dir_propia.sin_addr.s_addr = htonl(INADDR_ANY); /* cq. dir. de la maquina */
dir_propia.sin_port = htons(0),                 /* puerto aleatorio */
resultado = bind(conexion,(struct sockaddr *) &dir_propia,sizeof(dir_propia));
...
/* CONSULTA NOMBRE servidor y rellenar dir remota */
datos_host = gethostbyname(...<nombre>...);
memset(&dir_remota, 0, sizeof(struct sockaddr_in));
dir_remota.sin_family = datos_host->h_addrtype;
memcpy(&dir_remota.sin_addr.s_addr,datos_host->h_addr_list[0],datos_host->h_length);
dir_remota.sin_port = htons(...);   /* indicar PUERTO REMOTO */
...
while(! final_dialogo) { /* intercambio con servidor */
   ...
   peticion = crear_peticion(....);
   /* ENVIO     */
   enviados = sendto(conexion, peticion, tam_peticion, 0,
                    (struct sockaddr *) &dir_remota, sizeof(dir_remota));
   ...
   /* RECEPCION */
   tam_dir_remota2 = sizeof(dir_remota2);
   recibidos = recvfrom(conexion, respuesta, tam_respuesta, 0,
                        (struct sockaddr *) &dir_remota2, &tam_dir_remota);
   ...
   procesar_respuesta(respuesta, ....);
   ...
}
close(conexion);  /* CERRAR conexion */
...
```

## esquema servidor

```
...
int conexion;
int resultado;
struct sockaddr_in dir_propia, dir_cliente;
struct hostent * datos_host;

/* CREACION de un socket UDP  */
conexion = socket(PF_INET, SOCK_DGRAM, 0);
...
/* BIND con dirección local y PUERTO DE ESCHUCA REMOTO */
memset(&dir_local, 0, sizeof(struct sockaddr_in));
dir_propia.sin_family = AF_INET;
dir_propia.sin_addr.s_addr = htonl(INADDR_ANY); /* cq. dir. de la maquina */
dir_propia.sin_port = htons(....);              /* PUERTO de ESCUCHA  */
resultado = bind(conexion,(struct sockaddr *) &dir_propia,sizeof(dir_propia));
...
while(1) {  /* Bucle principal aceptando mensajes */
   ...
   /* RECEPCION */
   tam_dir_cliente = sizeof(dir_cliente);
   recibidos = recvfrom(conexion, peticion, tam_peticion, 0,
                        (struct sockaddr *) &dir_cliente, &tam_dir_cliente);
   ...
   procesar_respuesta(respuesta, ....);
   ...
   /* ENVIO    */
   enviados = sendto(conexion, respuesta, tam_respuesta, 0,
                     (struct sockaddr *) &dir_cliente, sizeof(dir_cliente));
   ...
}
close(conexion);   /* CERAR conexion */
...
```

## 2.2.3 Uso de sockets en Java

El paquete **java.net** ofrece la infraestructura y las clases necesarias para soportar el uso de la interfaz de sockets.

### Clases relevantes del paquete java.net

- Manejo de direcciones
  - `InetAddress`: encapsula la dirección IP de una máquina
    - No tiene constructores
    - Ofrece métodos estáticos para generar direcciones (`InetAddress`) a partir de Strings (nombres de dominio, direcciones en formato `xxx.xxx.xxx.xxx`) y para recuperar la dirección de la propia máquina.
      - ◇ `InetAddress getByName(String host)`
      - ◇ `InetAddress getLocalHost()`
  - `InetSocketAddress`: encapsula la dirección de un socket (dir. IP + nº puerto)
    - Constructores
      - ◇ `InetSocketAddress(InetAddress addr, int port)`
      - ◇ `InetSocketAddress(String hostname, int port)`
      - ◇ `InetSocketAddress(int port)`
- Sockets en modo stream (TCP)
  - `ServerSocket`: encapsula un Socket de servidor (espera conexiones)
    - Implementa las primitivas **bind()** y **accept()**
    - Ofrece métodos para controlar las conexiones en espera
    - Crea objetos `Socket` para intercambio de datos al aceptar la conexiones de los clientes
  - `Socket`: encapsula un socket de comunicación
    - Implementa las primitivas **bind()** y **connect()**
    - El envio/recepción de datos (primitivas **send()** y **recv()**) se realiza a través de objetos `InputStream` y `OutputStream` asociados al objeto `Socket`
- Sockets en modo datagrama (UDP)
  - `DatagramPacket`: Encapsula un mensaje (datagrama) a recibir o a enviar hacia un destino (dir. IP + puerto) usando UDP
  - `DatagramSocket`: Encapsula un socket en modo datagrama que permite enviar o recibir los datos asociados a un `DatagramPacket`

## (a) Sockets java en modo stream ((TCP)

1. Clase **SocketServer**: socket servidor, escucha conex. en un puerto
   - **Constructores:**
     - `ServerSocket(int port)`: socket de servidor enlazado [bind()] al puerto de escucha indicado
     - `ServerSocket(int port, int backlog)`: *idem*, indicando tamaño de la cola de espera
     - `ServerSocket()`: socket de servidor no enlazado [bind()]
   - **Métodos:**
     - `Socket accept()`: escucha/aceptación de conexiones
       - el servidor se bloquea y queda a la espera en el puerto indicado
       - cuando reciba una conexión devuelve un objeto `Socket` para comunicarse con el cliente
     - `void setSoTimeout(int timeout)`: establece tiempo máximo de espera para las conexiones entrantes
     - `void close()`: libera el puerto y cierra las conexiones establecidas
   - Ejemplo: escucha en puerto 80

```
ServerSocket servidor = new ServerSocket(80);
Socket cliente = servidor.accept();
...
```

2. Clase **Socket**: socket para conexión y envío/recepción
   - **Constructores:**
     - `Socket(InetAddress address, int port)`: crea un socket y lo conecta a la dirección y puerto indicados
     - `Socket(String host, int port)`: *idem*, con dirección en formato String
     - `Socket()`: crea un socket pero no lo conecta
   - **Métodos:**
     - Métodos para enlazar y conectar { `void bind(SocketAddress bindpoint)` ; `void connect(SocketAddress endpoint)`
     - `void close()`: cierra el socket y libera recursos
     - `InputStream getInputStream()`: devuelve un objeto `InputStream` a través del cual **leer** los datos enviados desde el otro extremo del socket
     - `OutputStream getOutputStream()`: devuelve un objeto `OutputStream` a través del cual **escribir** los datos a enviar hacia el otro extremo del socket

   **Nota:** El resultado de `getInputStream()` y `getOutputStream()` se suele usar para crear otros "*streams*" definidos en el paquete **java.io**
   - `DataInputStream` y `DataOutputStream` leen/escriben datos de tipos primitivos [`writeInt()/readInt()`, `writeDouble()/readDouble()`]
   - `ObjectInputStream` y `ObjectOutputStream` leen/escriben objetos java serializados (implementan interfaz `java.io.Serializable`) [`writeObject()/readObject()`]

# (b) Esquema general cliente/servidor java stream

## esquema clientes

```
import java.net.*;
import java.io.*;
...
InetAddress direccionServidor = InetAddress.getByName(..<nombre>..);
Socket conexion = new Socket(direccionServidor, ..<puerto>..);

DataOutputStream out = new DataOutputStream(socket.getOutputStream());
DataInputStream in = new DataInputStream(socket.getInputStream());

while (! finDialogo) {  // bucle DIALOGO con servidor
   Peticion peticion = crearPeticion(....);
   out.write(peticion.getBytes(),0, peticion.getTamano())  // ENVIO
   leidos = in.read(respuesta);   // RECEPCION
   procesarRespuesta(respuesta, ....);
   ...
}
out.close();
in.close();
conexion.close();
...
```

## esquema servidor

```
import java.net.*;
import java.io.*;
...
ServerSocket servidor = new ServerSocket(..<puerto>..);
while(true) {  // bucle aceptando conexiones
    Socket cliente = servidor.accept();
    DataOutputStream out = new DataOutputStream(cliente.getOutputStream());
    DataInputStream in = new DataInputStream(cliente.getInputStream());

    while (! finDialogo) {  // bucle de DIALOGO con servidor
       leido = in.read(peticion);    // RECEPCION
       Respuesta respuesta = procesarPeticion(peticion,...);
       out.write(respuesta.getBytes(),0, respuesta.getTamano())  // ENVIO
       ...
    }
    out.close();
    in.close();
    cliente.close();
}
servidor.close();
...
```

## esquema servidor multihilo

```
import java.net.*;
import java.io.*;
...
ServerSocket servidor = new ServerSocket(..<puerto>..);
while(true) {  // bucle aceptando conexiones
    Socket cliente = servidor.accept();
    HiloServidor hilo = new HiloServidor(cliente);
    new Thread(hilo).start();
}
servidor.close();
...


public class HiloServidor implements Runnable {
    private Socket socket;

    public HiloServidor (Socket s) {this.socket = s;}

    public void run() {
       DataOutputStream out = new DataOutputStream(socket.getOutputStream());
       DataInputStream in = new DataInputStream(socket.getInputStream());

       while (! finDialogo) {  // bucle de DIALOGO con servidor
          leido = in.read(peticion);    // RECEPCION
          Respuesta respuesta = procesarPeticion(peticion,...);
          out.write(respuesta.getBytes(),0, respuesta.getTamano())  // ENVIO
          ...
       }
       out.close();
       in.close();
       socket.close();
    }
...
}  // Fin clase HiloServidor
```

## (c) Sockets java en modo datagrama ((UDP)

1. Clase **DatagramPacket**: representa los mensajes (datagramas) que se envía/reciben
   - Almacena:
     - dirección + puerto (de origen o de destino)
     - array de bytes con los datos enviados/recibidos [buffer]
     - longitud de los datos+ posición de inicio de los datos [offset]
   - En envio:
     - se debe indicar dirección + puerto de destino
     - se escribe en el buffer los datos a enviar
   - En recepción:
     - devolverá dirección + puerto de origen
     - en el buffer estarán los datos recibidos
   - **Constructores:**
     - `DatagramPacket(byte[] buf, int length)`: construye un datagrama para recibir paquetes de la longitud indicada
     - `DatagramPacket(byte[] buf, int length, InetAddress address, int port)`: construye un datagrama para enviar datagramas del tamaño indicado a la dir. IP y puerto especificados
   - **Métodos:**
     - `byte[] getData()`: devuelve el buffer con los datos recibidos
     - `int getLength()`: longitud de los datos a enviar o recibir
     - `void setData(byte[])`: esteblece el buffer con los datos a enviar
     - `void setLength(int l)`: longitud de los datos a enviar o recibir
2. Clase **DatagramSocket**: representa un socket para enviar/recibir datagramas
   - Siempre asociados [bind()] a una dirección local
   - dir+puerto destino se debe especificar en el `DatagramPacket` a enviar
   - dir+puerto origen vendrá indicado en el `DatagramPacket` recibido
   - **Constructores:**
     - `DatagramSocket()`: crea socket UDP enlazado a un puerto aleatorio local
     - `DatagramSocket(int port)`: crea socket UDP enlazado al puerto local indicado
   - **Métodos:**
     - `void close()`: cierra el socket y libera recursos
     - `void setSoTimeout(int timeout)`: establece el tiempo máx. de espera en las recepciones
     - `void receive(DatagramPacket p)`: espera a recibir un datagrama dejando los datos en `p`
       - el datagrama contendrá dir+puerto de origen
       - si no se recibe en el tiempo límite genera `SocketTimeoutException`
     - `void send(DatagramPacket p)`: envia un datagrama
       - el datagrama debe indicar dir+puerto de destino

**Nota:** Para simplificar la escritura/lectura sobre los buffers de envio/recepción se pueden crear objetos `ByteArrayOutputStream/ByteArrayInputStream` sobre los cuales construir otros "*streams*" **java.io** (`DataOutputStream/DataInputStream` o `ObjectOutputStream/ObjectInputStream`)

## Ejemplos

- envío de datagramas

```
byte[] buffer = new byte[512];
< escritura en buffer >
...
DatagramPacket datagramaEnvio = new DatagramPacket(buffer, 512,
                    IntetAddres.getByName(<nombre>), <puerto>);
DatagramSocket socket = new DatagramSocket();
socket.send(datagramaEnvio);
...
socket.close();
```

- recepción de datagramas

```
byte[] buffer = new byte[512];

DatagramPacket datagramaRecepcion = new DatagramPacket(buffer, 512);
DatagramSocket socket = new DatagramSocket();
socket.receive(datagramaRecepcion);

InetAddress dirOrigen = datagramaRecepcion.getAddress();
int puertoOrigen = datagramaRecepcion.getPort();

< lectura desde buffer >
...
socket.close();
```

- envío de datagramas (usando ByteArrayOutputStream)

```
ByteArrayOutputStream outBuffer = new ByteArrayOutputStream(512);
DataOutputStream out = new DataOutputStream();
out.writeInt(...);
out.writeDouble(...);
...
DatagramPacket datagramaEnvio = new DatagramPacket(outBuffer.toByteArray(),
                outBuffer.size(), InetAddress.getByName(<nombre>), <puerto>);
DatagramSocket socket = new DatagramSocket();
socket.send(datagramaEnvio);
...
out.close();
socket.close();
```

- recepción de datagramas (usando ByteArrayInputStream)

```
DatagramPacket datagramaRecepcion = new DatagramPacket(buffer, 512);
DatagramSocket socket = new DatagramSocket();
socket.receive(datagramaRecepcion);

InetAddress dirOrigen = datagramaRecepcion.getAddress();
int puertoOrigen = datagramaRecepcion.getPort();

DataInputStream in = new DataInputStream(
                      new ByteArrayInputStream(datagramaRecepcion.getData()));


int valor1 = in.readInt();
double valor2 = in.readDouble();
...

DatagramPacket datagramaEnvio = new DatagramPacket(outBuffer.toByteArray(),
                outBuffer.size(),InetAddresS.getByName(<nombre>), <puerto>);
DatagramSocket socket = new DatagramSocket();
socket.send(datagramaEnvio);
...
out.close();
socket.close();
```