



INFORMATICA I

Uso de los Sockets

Ing. Juan Carlos Cuttitta

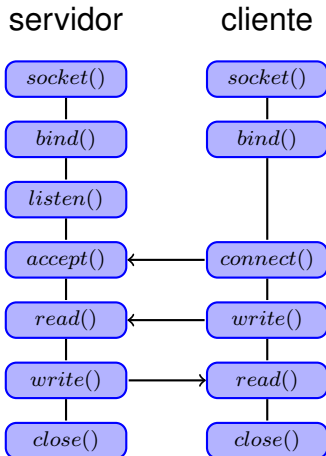
*Universidad Tecnológica Nacional
Facultad Regional Buenos Aires
Departamento de Ingeniería Electrónica*

9 de septiembre de 2019

Uso de los **sockets orientados a la conexión**

Para comunicar dos programas escritos en lenguaje C mediante sockets, es necesario seguir los siguientes pasos, que se ilustra en la figura.

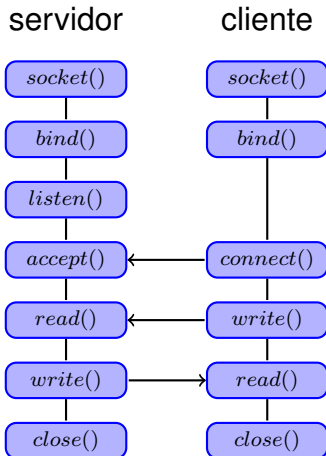
- 1 **Crear el socket mediante la función `socket()`.**
- 2 Asignar una dirección final al socket a la que pueda referirse el otro interlocutor. Función `bind()`.
- 3 En sockets tipo stream, es necesario conectar con otro socket cuya dirección debemos conocer y se logra con la función `connect()` para el proceso cliente y las funciones `listen()` y `accept()` para el proceso servidor.
- 4 Comunicarse. En sockets tipo stream, basta usar `write()` para volcar datos en el socket que el otro extremo puede leer mediante la función `read()`, y a la inversa.
- 5 cuando la comunicación se da por finalizada, ambos deben cerrar el socket con la función `close()`.



Uso de los **sockets orientados a la conexión**

Para comunicar dos programas escritos en lenguaje C mediante sockets, es necesario seguir los siguientes pasos, que se ilustra en la figura.

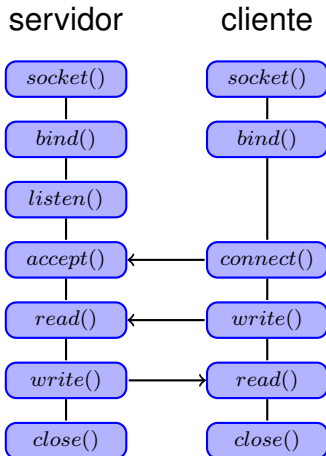
- 1 Crear el socket mediante la función `socket()`.
- 2 Asignar una dirección final al socket a la que pueda referirse el otro interlocutor. Función `bind()`.
- 3 En sockets tipo stream, es necesario conectar con otro socket cuya dirección debemos conocer y se logra con la función `connect()` para el proceso cliente y las funciones `listen()` y `accept()` para el proceso servidor.
- 4 Comunicarse. En sockets tipo stream, basta usar `write()` para volcar datos en el socket que el otro extremo puede leer mediante la función `read()`, y a la inversa.
- 5 cuando la comunicación se da por finalizada, ambos deben cerrar el socket con la función `close()`.



Uso de los **sockets orientados a la conexión**

Para comunicar dos programas escritos en lenguaje C mediante sockets, es necesario seguir los siguientes pasos, que se ilustra en la figura.

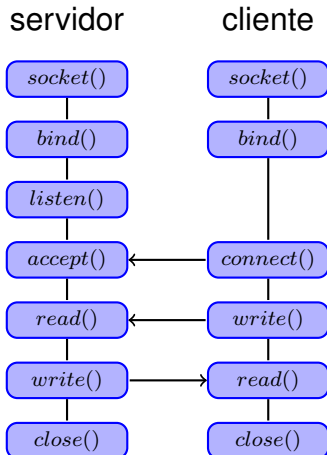
- 1 Crear el socket mediante la función `socket()`.
- 2 Asignar una dirección final al socket a la que pueda referirse el otro interlocutor. Función `bind()`.
- 3 En sockets tipo stream, es necesario conectar con otro socket cuya dirección debemos conocer y se logra con la función `connect()` para el proceso cliente y las funciones `listen()` y `accept()` para el proceso servidor.
- 4 Comunicarse. En sockets tipo stream, basta usar `write()` para volcar datos en el socket que el otro extremo puede leer mediante la función `read()`, y a la inversa.
- 5 cuando la comunicación se da por finalizada, ambos deben cerrar el socket con la función `close()`.



Uso de los **sockets orientados a la conexión**

Para comunicar dos programas escritos en lenguaje C mediante sockets, es necesario seguir los siguientes pasos, que se ilustra en la figura.

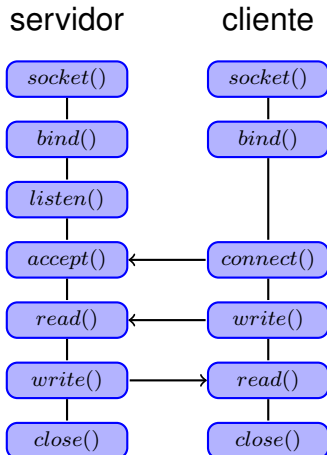
- 1 Crear el socket mediante la función `socket()`.
- 2 Asignar una dirección final al socket a la que pueda referirse el otro interlocutor. Función `bind()`.
- 3 En sockets tipo stream, es necesario conectar con otro socket cuya dirección debemos conocer y se logra con la función `connect()` para el proceso cliente y las funciones `listen()` y `accept()` para el proceso servidor.
- 4 Comunicarse. En sockets tipo stream, basta usar `write()` para volcar datos en el socket que el otro extremo puede leer mediante la función `read()`, y a la inversa.
- 5 cuando la comunicación se da por finalizada, ambos deben cerrar el socket con la función `close()`.



Uso de los **sockets orientados a la conexión**

Para comunicar dos programas escritos en lenguaje C mediante sockets, es necesario seguir los siguientes pasos, que se ilustra en la figura.

- 1 Crear el socket mediante la función `socket()`.
- 2 Asignar una dirección final al socket a la que pueda referirse el otro interlocutor. Función `bind()`.
- 3 En sockets tipo stream, es necesario conectar con otro socket cuya dirección debemos conocer y se logra con la función `connect()` para el proceso cliente y las funciones `listen()` y `accept()` para el proceso servidor.
- 4 Comunicarse. En sockets tipo stream, basta usar `write()` para volcar datos en el socket que el otro extremo puede leer mediante la función `read()`, y a la inversa.
- 5 cuando la comunicación se da por finalizada, ambos deben cerrar el socket con la función `close()`.



Creación del socket: **socket()**

La llamada a `socket()` es de la siguiente forma:

```
s = socket(fam_pro, tipo_ser, protocolo);
```

- ❶ **s**: Valor devuelto por la función. Tiene que ser de tipo entero.
 - Un valor negativo significa que ha ocurrido un error.
 - Un valor positivo será el descriptor del socket y es el que se usará en las funciones siguientes.
- ❷ **fam_pro**: Especifica la familia de protocolos que se usará en la comunicación.
 - `PF_UNIX`: para comunicar procesos UNIX en la misma máquina.
 - `PF_INET`: para comunicar procesos en diferentes máquinas a través de Internet.
 - Estas constantes están definidas en el fichero `<sys/socket.h>`.

Creación del socket: **socket()**

La llamada a `socket()` es de la siguiente forma:

```
s = socket(fam_pro, tipo_ser, protocolo);
```

- 1 **s**: Valor devuelto por la función. Tiene que ser de tipo entero.
 - Un valor negativo significa que ha ocurrido un error.
 - Un valor positivo será el descriptor del socket y es el que se usará en las funciones siguientes.
- 2 **fam_pro**: Especifica la familia de protocolos que se usará en la comunicación.
 - PF_UNIX: para comunicar procesos UNIX en la misma máquina.
 - PF_INET: para comunicar procesos en diferentes máquinas a través de Internet.
 - Estas constantes están definidas en el fichero `<sys/socket.h>`.

Creación del socket: **socket()**

La llamada a `socket()` es de la siguiente forma:

```
s = socket(fam_pro, tipo_ser, protocolo);
```

- 1 **s**: Valor devuelto por la función. Tiene que ser de tipo entero.
 - Un valor negativo significa que ha ocurrido un error.
 - Un valor positivo será el descriptor del socket y es el que se usará en las funciones siguientes.
- 2 **fam_pro**: Especifica la familia de protocolos que se usará en la comunicación.
 - PF_UNIX: para comunicar procesos UNIX en la misma máquina.
 - PF_INET: para comunicar procesos en diferentes máquinas a través de Internet.
 - Estas constantes están definidas en el fichero `<sys/socket.h>`.

Creación del socket: **socket()**

La llamada a `socket()` es de la siguiente forma:

```
s = socket(fam_pro, tipo_ser, protocolo);
```

- 1 **s**: Valor devuelto por la función. Tiene que ser de tipo entero.
 - Un valor negativo significa que ha ocurrido un error.
 - Un valor positivo será el descriptor del socket y es el que se usará en las funciones siguientes.
- 2 **fam_pro**: Especifica la familia de protocolos que se usará en la comunicación.
 - `PF_UNIX`: para comunicar procesos UNIX en la misma máquina.
 - `PF_INET`: para comunicar procesos en diferentes máquinas a través de Internet.
 - Estas constantes están definidas en el fichero `<sys/socket.h>`.

Creación del socket: **socket()**

La llamada a `socket()` es de la siguiente forma:

```
s = socket(fam_pro, tipo_ser, protocolo);
```

- 1 **s**: Valor devuelto por la función. Tiene que ser de tipo entero.
 - Un valor negativo significa que ha ocurrido un error.
 - Un valor positivo será el descriptor del socket y es el que se usará en las funciones siguientes.
- 2 **fam_pro**: Especifica la familia de protocolos que se usará en la comunicación.
 - **PF_UNIX**: para comunicar procesos UNIX en la misma máquina.
 - **PF_INET**: para comunicar procesos en diferentes máquinas a través de Internet.
 - Estas constantes están definidas en el fichero `<sys/socket.h>`.

Creación del socket: **socket()**

La llamada a `socket()` es de la siguiente forma:

```
s = socket(fam_pro, tipo_ser, protocolo);
```

- 1 **s**: Valor devuelto por la función. Tiene que ser de tipo entero.
 - Un valor negativo significa que ha ocurrido un error.
 - Un valor positivo será el descriptor del socket y es el que se usará en las funciones siguientes.
- 2 **fam_pro**: Especifica la familia de protocolos que se usará en la comunicación.
 - PF_UNIX: para comunicar procesos UNIX en la misma máquina.
 - PF_INET: para comunicar procesos en diferentes máquinas a través de Internet.
 - Estas constantes están definidas en el fichero `<sys/socket.h>`.

Creación del socket: **socket()**

La llamada a `socket()` es de la siguiente forma:

```
s = socket(fam_pro, tipo_ser, protocolo);
```

- 1 **s**: Valor devuelto por la función. Tiene que ser de tipo entero.
 - Un valor negativo significa que ha ocurrido un error.
 - Un valor positivo será el descriptor del socket y es el que se usará en las funciones siguientes.
- 2 **fam_pro**: Especifica la familia de protocolos que se usará en la comunicación.
 - PF_UNIX: para comunicar procesos UNIX en la misma máquina.
 - PF_INET: para comunicar procesos en diferentes máquinas a través de Internet.
 - **Estas constantes están definidas en el fichero `<sys/socket.h>`.**

Creación del socket: **socket()**

La llamada a `socket()` es de la siguiente forma:

```
s = socket(fam_pro, tipo_ser, protocolo);
```

❶ **tipo_ser**: El socket será:

- `SOCK_STREAM` para tipo stream.
- `SOCK_DGRAM` para tipo datagram.

❷ **protocolo**: Especifica que tipo de protocolo:

- Para el servicio stream (TCP)
- Para el servicio datagram (UDP).
- Lo mejor es poner un cero que significa que el sistema elija el protocolo más adecuado para el tipo solicitado.

Para crear un socket de tipo stream para una comunicación a través de la red, la llamada sería:

```
s = socket( PF_INET, SOCK_STREAM, 0 );
```

Creación del socket: **socket()**

La llamada a `socket()` es de la siguiente forma:

```
s = socket(fam_pro, tipo_ser, protocolo);
```

- 1 **tipo_ser**: El socket será:
 - **SOCK_STREAM** para tipo stream.
 - **SOCK_DGRAM** para tipo datagram.
- 2 **protocolo**: Especifica que tipo de protocolo:
 - Para el servicio stream (TCP)
 - Para el servicio datagram (UDP).
 - Lo mejor es **poner un cero** que significa que el sistema elija el protocolo más adecuado para el tipo solicitado.

Para crear un socket de tipo stream para una comunicación a través de la red, la llamada sería:

```
s = socket( PF_INET, SOCK_STREAM, 0 );
```

Creación del socket: **socket()**

La llamada a `socket()` es de la siguiente forma:

```
s = socket(fam_pro, tipo_ser, protocolo);
```

- 1 **tipo_ser**: El socket será:
 - **SOCK_STREAM** para tipo stream.
 - **SOCK_DGRAM** para tipo datagram.
- 2 **protocolo**: Especifica que tipo de protocolo:
 - Para el servicio stream (TCP)
 - Para el servicio datagram (UDP).
 - Lo mejor es poner un cero que significa que el sistema elija el protocolo más adecuado para el tipo solicitado.

Para crear un socket de tipo stream para una comunicación a través de la red, la llamada sería:

```
s = socket( PF_INET, SOCK_STREAM, 0 );
```


Creación del socket: **socket()**

La llamada a `socket()` es de la siguiente forma:

```
s = socket(fam_pro, tipo_ser, protocolo);
```

- 1 **tipo_ser**: El socket será:
 - **SOCK_STREAM** para tipo stream.
 - **SOCK_DGRAM** para tipo datagram.
- 2 **protocolo**: Especifica que tipo de protocolo:
 - Para el servicio stream (TCP)
 - Para el servicio datagram (UDP).
 - Lo mejor es poner un **cero** que significa que el sistema elija el protocolo más adecuado para el tipo solicitado.

Para crear un socket de tipo stream para una comunicación a través de la red, la llamada sería:

```
s = socket( PF_INET, SOCK_STREAM, 0 );
```

Creación del socket: **socket()**

La llamada a `socket()` es de la siguiente forma:

```
s = socket(fam_pro, tipo_ser, protocolo);
```

- 1 **tipo_ser**: El socket será:
 - **SOCK_STREAM** para tipo stream.
 - **SOCK_DGRAM** para tipo datagram.
- 2 **protocolo**: Especifica que tipo de protocolo:
 - **Para el servicio stream (TCP)**
 - Para el servicio datagram (UDP).
 - Lo mejor es **poner un cero** que significa que el sistema elija el protocolo más adecuado para el tipo solicitado.

Para crear un socket de tipo stream para una comunicación a través de la red, la llamada sería:

```
s = socket( PF_INET, SOCK_STREAM, 0 );
```

Creación del socket: **socket()**

La llamada a `socket()` es de la siguiente forma:

```
s = socket(fam_pro, tipo_ser, protocolo);
```

- 1 **tipo_ser**: El socket será:
 - **SOCK_STREAM** para tipo stream.
 - **SOCK_DGRAM** para tipo datagram.
- 2 **protocolo**: Especifica que tipo de protocolo:
 - Para el servicio stream (TCP)
 - **Para el servicio datagram (UDP).**
 - Lo mejor es **poner un cero** que significa que el sistema elija el protocolo más adecuado para el tipo solicitado.

Para crear un socket de tipo stream para una comunicación a través de la red, la llamada sería:

```
s = socket( PF_INET, SOCK_STREAM, 0 );
```

Creación del socket: **socket()**

La llamada a `socket()` es de la siguiente forma:

```
s = socket(fam_pro, tipo_ser, protocolo);
```

- 1 **tipo_ser**: El socket será:
 - **SOCK_STREAM** para tipo stream.
 - **SOCK_DGRAM** para tipo datagram.
- 2 **protocolo**: Especifica que tipo de protocolo:
 - Para el servicio stream (TCP)
 - Para el servicio datagram (UDP).
 - **Lo mejor es poner un cero que significa que el sistema elija el protocolo más adecuado para el tipo solicitado.**

Para crear un socket de tipo stream para una comunicación a través de la red, la llamada sería:

```
s = socket( PF_INET, SOCK_STREAM, 0 );
```

Creación del socket: **socket()**

La llamada a `socket()` es de la siguiente forma:

```
s = socket(fam_pro, tipo_ser, protocolo);
```

- 1 **tipo_ser**: El socket será:
 - **SOCK_STREAM** para tipo stream.
 - **SOCK_DGRAM** para tipo datagram.
- 2 **protocolo**: Especifica que tipo de protocolo:
 - Para el servicio stream (TCP)
 - Para el servicio datagram (UDP).
 - Lo mejor es **poner un cero** que significa que el sistema elija el protocolo más adecuado para el tipo solicitado.

Para crear un socket de tipo stream para una comunicación a través de la red, la llamada sería:

```
s = socket( PF_INET, SOCK_STREAM, 0 );
```

Asignar una dirección al socket: **bind()**

- 1 Un socket queda definido cuando contiene a todos los parámetros siguientes:
 - protocolo
 - dirección IP máquina local
 - número puerto de protocolo local
 - dirección IP máquina remota
 - número puerto de protocolo remoto
- 2 Con la función `socket()` se crea un socket y se le asigna el protocolo, pero quedan sin asignar los restantes cuatro parámetros.
- 3 la función `bind()` asignará los parámetros
 - dirección IP máquina local
 - número puerto de protocolo local

La sintaxis de `bind()` es la siguiente:

```
retcod=bind(s, dir_fin, lon_dir_fin);
```

Asignar una dirección al socket: **bind()**

- 1 Un socket queda definido cuando contiene a todos los parámetros siguientes:
 - **protocolo**
 - dirección IP máquina local
 - número puerto de protocolo local
 - dirección IP máquina remota
 - número puerto de protocolo remoto
- 2 Con la función **socket()** se crea un socket y se le asigna el protocolo, pero quedan sin asignar los restantes cuatro parámetros.
- 3 la función **bind()** asignará los parámetros
 - dirección IP máquina local
 - número puerto de protocolo local

La sintaxis de **bind()** es la siguiente:

```
retcod=bind(s, dir_fin, lon_dir_fin);
```

Asignar una dirección al socket: **bind()**

- 1 Un socket queda definido cuando contiene a todos los parámetros siguientes:
 - protocolo
 - **dirección IP máquina local**
 - número puerto de protocolo local
 - dirección IP máquina remota
 - número puerto de protocolo remoto
- 2 Con la función `socket()` se crea un socket y se le asigna el protocolo, pero quedan sin asignar los restantes cuatro parámetros.
- 3 la función `bind()` asignará los parámetros
 - dirección IP máquina local
 - número puerto de protocolo local

La sintaxis de `bind()` es la siguiente:

```
retcod=bind(s, dir_fin, lon_dir_fin);
```


Asignar una dirección al socket: **bind()**

- 1 Un socket queda definido cuando contiene a todos los parámetros siguientes:
 - protocolo
 - dirección IP máquina local
 - **número puerto de protocolo local**
 - dirección IP máquina remota
 - número puerto de protocolo remoto
- 2 Con la función `socket()` se crea un socket y se le asigna el protocolo, pero quedan sin asignar los restantes cuatro parámetros.
- 3 la función `bind()` asignará los parámetros
 - dirección IP máquina local
 - número puerto de protocolo local

La sintaxis de `bind()` es la siguiente:

```
retcod=bind(s, dir_fin, lon_dir_fin);
```

Asignar una dirección al socket: **bind()**

- 1 Un socket queda definido cuando contiene a todos los parámetros siguientes:
 - protocolo
 - dirección IP máquina local
 - número puerto de protocolo local
 - **dirección IP máquina remota**
 - número puerto de protocolo remoto
- 2 Con la función `socket()` se crea un socket y se le asigna el protocolo, pero quedan sin asignar los restantes cuatro parámetros.
- 3 la función `bind()` asignará los parámetros
 - dirección IP máquina local
 - número puerto de protocolo local

La sintaxis de `bind()` es la siguiente:

```
retcod=bind(s, dir_fin, lon_dir_fin);
```

Asignar una dirección al socket: **bind()**

- 1 Un socket queda definido cuando contiene a todos los parámetros siguientes:
 - protocolo
 - dirección IP máquina local
 - número puerto de protocolo local
 - dirección IP máquina remota
 - **número puerto de protocolo remoto**
- 2 Con la función `socket()` se crea un socket y se le asigna el protocolo, pero quedan sin asignar los restantes cuatro parámetros.
- 3 la función `bind()` asignará los parámetros
 - dirección IP máquina local
 - número puerto de protocolo local

La sintaxis de `bind()` es la siguiente:

```
retcod=bind(s, dir_fin, lon_dir_fin);
```

Asignar una dirección al socket: **bind()**

- 1 Un socket queda definido cuando contiene a todos los parámetros siguientes:
 - protocolo
 - dirección IP máquina local
 - número puerto de protocolo local
 - dirección IP máquina remota
 - número puerto de protocolo remoto
- 2 Con la función `socket()` se crea un socket y se le asigna el protocolo, pero quedan sin asignar los restantes cuatro parámetros.
- 3 la función `bind()` asignará los parámetros
 - dirección IP máquina local
 - número puerto de protocolo local

La sintaxis de `bind()` es la siguiente:

```
retcod=bind(s, dir_fin, lon_dir_fin);
```

Asignar una dirección al socket: **bind()**

- 1 Un socket queda definido cuando contiene a todos los parámetros siguientes:
 - protocolo
 - dirección IP máquina local
 - número puerto de protocolo local
 - dirección IP máquina remota
 - número puerto de protocolo remoto
- 2 Con la función **socket()** se crea un socket y se le asigna el protocolo, pero quedan sin asignar los restantes cuatro parámetros.
- 3 la función **bind()** asignará los parámetros
 - dirección IP máquina local
 - número puerto de protocolo local

La sintaxis de **bind()** es la siguiente:

```
retcod=bind(s, dir_fin, lon_dir_fin);
```

Asignar una dirección al socket: **bind()**

- 1 Un socket queda definido cuando contiene a todos los parámetros siguientes:
 - protocolo
 - dirección IP máquina local
 - número puerto de protocolo local
 - dirección IP máquina remota
 - número puerto de protocolo remoto
- 2 Con la función **socket()** se crea un socket y se le asigna el protocolo, pero quedan sin asignar los restantes cuatro parámetros.
- 3 la función **bind()** asignará los parámetros
 - **dirección IP máquina local**
 - número puerto de protocolo local

La sintaxis de **bind()** es la siguiente:

```
retcod=bind(s, dir_fin, lon_dir_fin);
```

Asignar una dirección al socket: **bind()**

- 1 Un socket queda definido cuando contiene a todos los parámetros siguientes:
 - protocolo
 - dirección IP máquina local
 - número puerto de protocolo local
 - dirección IP máquina remota
 - número puerto de protocolo remoto
- 2 Con la función **socket()** se crea un socket y se le asigna el protocolo, pero quedan sin asignar los restantes cuatro parámetros.
- 3 la función **bind()** asignará los parámetros
 - dirección IP máquina local
 - **número puerto de protocolo local**

La sintaxis de **bind()** es la siguiente:

```
retcod=bind(s, dir_fin, lon_dir_fin);
```

Asignar una dirección al socket: **bind()**

- 1 Un socket queda definido cuando contiene a todos los parámetros siguientes:
 - protocolo
 - dirección IP máquina local
 - número puerto de protocolo local
 - dirección IP máquina remota
 - número puerto de protocolo remoto
- 2 Con la función **socket()** se crea un socket y se le asigna el protocolo, pero quedan sin asignar los restantes cuatro parámetros.
- 3 la función **bind()** asignará los parámetros
 - dirección IP máquina local
 - número puerto de protocolo local

La sintaxis de **bind()** es la siguiente:

```
retcod=bind(s, dir_fin, lon_dir_fin);
```


Asignar una dirección al socket: **bind()**

```
retcod=bind(s, dir_fin, lon_dir_fin);
```

- 1 **s**: Es el descriptor devuelto por la función **socket()**.
- 2 **dir_fin**: Es la dirección y puerto local que queremos asignar al socket. Este parámetro es un puntero a una estructura de tipo **sockaddr**.
 - **sockaddr_in** para las conexiones de la familia **AF_INET**
 - **sockaddr_un** para las conexiones de la familia **AF_UNIX**
- 3 **lon_dir_fin**: Es el tamaño en bytes del parámetro anterior. Lo habitual es usar la macro **sizeof** para averiguar este tercer parámetro.
- 4 **retcod**: Código de retorno de la función.
 - Si retorna 0, la operación se ha completado con éxito.
 - Si retorna -1 significa que ha ocurrido un error.

Asignar una dirección al socket: **bind()**

```
retcod=bind(s, dir_fin, lon_dir_fin);
```

- 1 **s**: Es el descriptor devuelto por la función **socket()**.
- 2 **dir_fin**: Es la dirección y puerto local que queremos asignar al socket. Este parámetro es un puntero a una estructura de tipo **sockaddr**.
 - **sockaddr_in** para las conexiones de la familia **AF_INET**
 - **sockaddr_un** para las conexiones de la familia **AF_UNIX**
- 3 **lon_dir_fin**: Es el tamaño en bytes del parámetro anterior. Lo habitual es usar la macro **sizeof** para averiguar este tercer parámetro.
- 4 **retcod**: Código de retorno de la función.
 - Si retorna 0, la operación se ha completado con éxito.
 - Si retorna -1 significa que ha ocurrido un error.

Asignar una dirección al socket: **bind()**

```
retcod=bind(s, dir_fin, lon_dir_fin);
```

- 1 **s**: Es el descriptor devuelto por la función **socket()**.
- 2 **dir_fin**: Es la dirección y puerto local que queremos asignar al socket. Este parámetro es un puntero a una estructura de tipo **sockaddr**.
 - **sockaddr_in** para las conexiones de la familia **AF_INET**
 - **sockaddr_un** para las conexiones de la familia **AF_UNIX**
- 3 **lon_dir_fin**: Es el tamaño en bytes del parámetro anterior. Lo habitual es usar la macro **sizeof** para averiguar este tercer parámetro.
- 4 **retcod**: Código de retorno de la función.
 - Si retorna 0, la operación se ha completado con éxito.
 - Si retorna -1 significa que ha ocurrido un error.

Asignar una dirección al socket: **bind()**

```
retcod=bind(s, dir_fin, lon_dir_fin);
```

- 1 **s**: Es el descriptor devuelto por la función **socket()**.
- 2 **dir_fin**: Es la dirección y puerto local que queremos asignar al socket. Este parámetro es un puntero a una estructura de tipo **sockaddr**.
 - **sockaddr_in** para las conexiones de la familia **AF_INET**
 - **sockaddr_un** para las conexiones de la familia **AF_UNIX**
- 3 **lon_dir_fin**: Es el tamaño en bytes del parámetro anterior. Lo habitual es usar la macro **sizeof** para averiguar este tercer parámetro.
- 4 **retcod**: Código de retorno de la función.
 - Si retorna 0, la operación se ha completado con éxito.
 - Si retorna -1 significa que ha ocurrido un error.

Asignar una dirección al socket: **bind()**

```
retcod=bind(s, dir_fin, lon_dir_fin);
```

- 1 **s**: Es el descriptor devuelto por la función **socket()**.
- 2 **dir_fin**: Es la dirección y puerto local que queremos asignar al socket. Este parámetro es un puntero a una estructura de tipo **sockaddr**.
 - **sockaddr_in** para las conexiones de la familia **AF_INET**
 - **sockaddr_un** para las conexiones de la familia **AF_UNIX**
- 3 **lon_dir_fin**: Es el tamaño en bytes del parámetro anterior. Lo habitual es usar la macro **sizeof** para averiguar este tercer parámetro.
- 4 **retcod**: Código de retorno de la función.
 - Si retorna 0, la operación se ha completado con éxito.
 - Si retorna -1 significa que ha ocurrido un error.

Asignar una dirección al socket: **bind()**

```
retcod=bind(s, dir_fin, lon_dir_fin);
```

- 1 **s**: Es el descriptor devuelto por la función **socket()**.
- 2 **dir_fin**: Es la dirección y puerto local que queremos asignar al socket. Este parámetro es un puntero a una estructura de tipo **sockaddr**.
 - **sockaddr_in** para las conexiones de la familia **AF_INET**
 - **sockaddr_un** para las conexiones de la familia **AF_UNIX**
- 3 **lon_dir_fin**: Es el tamaño en bytes del parámetro anterior. Lo habitual es usar la macro **sizeof** para averiguar este tercer parámetro.
- 4 **retcod**: Código de retorno de la función.
 - Si retorna 0, la operación se ha completado con éxito.
 - Si retorna -1 significa que ha ocurrido un error.

Asignar una dirección al socket: **bind()**

```
retcod=bind(s, dir_fin, lon_dir_fin);
```

- 1 **s**: Es el descriptor devuelto por la función **socket()**.
- 2 **dir_fin**: Es la dirección y puerto local que queremos asignar al socket. Este parámetro es un puntero a una estructura de tipo **sockaddr**.
 - **sockaddr_in** para las conexiones de la familia **AF_INET**
 - **sockaddr_un** para las conexiones de la familia **AF_UNIX**
- 3 **lon_dir_fin**: Es el tamaño en bytes del parámetro anterior. Lo habitual es usar la macro **sizeof** para averiguar este tercer parámetro.
- 4 **retcod**: Código de retorno de la función.
 - Si retorna 0, la operación se ha completado con éxito.
 - Si retorna -1 significa que ha ocurrido un error.

Asignar una dirección al socket: **bind()**

```
retcod=bind(s, dir_fin, lon_dir_fin);
```

- 1 **s**: Es el descriptor devuelto por la función **socket()**.
- 2 **dir_fin**: Es la dirección y puerto local que queremos asignar al socket. Este parámetro es un puntero a una estructura de tipo **sockaddr**.
 - **sockaddr_in** para las conexiones de la familia **AF_INET**
 - **sockaddr_un** para las conexiones de la familia **AF_UNIX**
- 3 **lon_dir_fin**: Es el tamaño en bytes del parámetro anterior. Lo habitual es usar la macro **sizeof** para averiguar este tercer parámetro.
- 4 **retcod**: Código de retorno de la función.
 - Si retorna 0, la operación se ha completado con éxito.
 - Si retorna -1 significa que ha ocurrido un error.

La estructura *sockaddr_in*

La función `bind()` recibe como segundo parámetro un puntero a una estructura de tipo `struct sockaddr_in`, por lo que es necesario hacer un cast. El tipo `struct sockaddr_in` está definido en `<netinet/in.h>` . Su aspecto es como sigue:

```
struct sockaddr_in {  
    short            sin_family;  
    u_short          sin_port;  
    struct in_addr   sin_addr;  
    u_long           sin_zero[8];  
}
```

`sin_addr` En algunas máquinas es de tipo `u_long`

La estructura *sockaddr_in*

- ❶ **sin_family**: Es la familia del socket, **AF_INET** en nuestro caso, ya que estamos interesados sólo en sockets que comuniquen a través de Internet.
- ❷ **sin_port**: Es el puerto de protocolo local. Este es un número que podemos elegir libremente, siempre que no esté en uso ese mismo puerto por otro proceso. Si usamos puertos libres por encima de 1000 no tendremos conflictos.
- ❸ **sin_addr** Es una estructura bastante sencilla.

```
struct in_addr {  
    unsigned long    s_addr;  
};
```

- ❹ **sin_zero**: Es un campo de relleno que no utilizaremos.

La estructura *sockaddr_in*

- ❶ **sin_family**: Es la familia del socket, **AF_INET** en nuestro caso, ya que estamos interesados sólo en sockets que comuniquen a través de Internet.
- ❷ **sin_port**: Es el puerto de protocolo local. Este es un número que podemos elegir libremente, siempre que no esté en uso ese mismo puerto por otro proceso. Si usamos puertos libres por encima de 1000 no tendremos conflictos.
- ❸ **sin_addr** Es una estructura bastante sencilla.

```
struct in_addr {  
    unsigned long    s_addr;  
};
```

- ❹ **sin_zero**: Es un campo de relleno que no utilizaremos.

La estructura *sockaddr_in*

- ❶ **sin_family**: Es la familia del socket, **AF_INET** en nuestro caso, ya que estamos interesados sólo en sockets que comuniquen a través de Internet.
- ❷ **sin_port**: Es el puerto de protocolo local. Este es un número que podemos elegir libremente, siempre que no esté en uso ese mismo puerto por otro proceso. Si usamos puertos libres por encima de 1000 no tendremos conflictos.
- ❸ **sin_addr** Es una estructura bastante sencilla.

```
struct in_addr {  
    unsigned long    s_addr;  
};
```

- ❹ **sin_zero**: Es un campo de relleno que no utilizaremos.

La estructura *sockaddr_in*

- ❶ **sin_family**: Es la familia del socket, **AF_INET** en nuestro caso, ya que estamos interesados sólo en sockets que comuniquen a través de Internet.
- ❷ **sin_port**: Es el puerto de protocolo local. Este es un número que podemos elegir libremente, siempre que no esté en uso ese mismo puerto por otro proceso. Si usamos puertos libres por encima de 1000 no tendremos conflictos.
- ❸ **sin_addr** Es una estructura bastante sencilla.

```
struct in_addr {  
    unsigned long    s_addr;  
}
```

- ❹ **sin_zero**: Es un campo de relleno que no utilizaremos.

La estructura *sockaddr_in*

- ❶ **sin_family**: Es la familia del socket, **AF_INET** en nuestro caso, ya que estamos interesados sólo en sockets que comuniquen a través de Internet.
- ❷ **sin_port**: Es el puerto de protocolo local. Este es un número que podemos elegir libremente, siempre que no esté en uso ese mismo puerto por otro proceso. Si usamos puertos libres por encima de 1000 no tendremos conflictos.
- ❸ **sin_addr** Es una estructura bastante sencilla.

```
struct in_addr {  
    unsigned long    s_addr;  
}
```

- ❹ **sin_zero**: Es un campo de relleno que no utilizaremos.

La estructura *in_addr*

s_addr es la dirección IP de nuestra máquina.

Lo normal es poner aquí la constante predefinida *INADDR_ANY*, ya que una misma máquina puede tener varias direcciones IP (si está conectada a varias redes simultáneamente). Al poner *INADDR_ANY* estamos dejando indeterminado el dato de **dirección local** con lo que nuestro programa podrá aceptar conexiones de todas las redes a las que pertenezca la máquina.

Lo habitual es que cada máquina pertenezca sólo a una red, por lo que poner *INADDR_ANY* es igual a poner la dirección IP de la máquina en cuestión, con la ventaja de que no necesitamos conocer esta dirección, y además el programa es más portable, ya que el mismo código fuente compilado en otra máquina (con otra dirección IP) funcionaría igual.

Ejemplo de uso de la estructura **sockaddr_in**

```
1 #include <sys/types.h>
2 #include <netinet/in.h>
3 #include <sys/socket.h>
4 ...
5 int s;
6 struct sockaddr_in local;
7 ...
8 s = socket(PF_INET, SOCK_STREAM, 0);
9 ... /* Comprobacion de errores */
10 local.sin_family=AF_INET;
11 local.sin_port=htons(15001);
12 local.sin_addr.s_addr = htonl(INADDR_ANY);
13 bind(s, (struct sockaddr *) &local, sizeof local);
14 ...
```


Ejemplo de uso de la estructura *sockaddr_in*

Observar las siguientes peculiaridades:

- ❶ Es necesario hacer un cast en el segundo parámetro de `bind()`.
- ❷ El tercer parámetro de `bind()` es el tamaño en bytes del segundo parámetro. Lo más cómodo es usar la macro `sizeof` que nos devuelve este valor.
- ❸ Está prohibido no comprobar errores !!
- ❹ El puerto que hemos usado (15001) lo hemos elegido arbitrariamente, pero deberá ser conocido también por el otro interlocutor.
- ❺ Las funciones `htons()` y `htonl()` se usan para traducir cantidades de más de 1 byte entre el formato utilizado por nuestra computadora y el utilizado por la red. Existen otras dos funciones para la traducción inversa.

Ejemplo de uso de la estructura *sockaddr_in*

Observar las siguientes peculiaridades:

- 1 Es necesario hacer un cast en el segundo parámetro de `bind()`.
- 2 El tercer parámetro de `bind()` es el tamaño en bytes del segundo parámetro. Lo más cómodo es usar la macro `sizeof` que nos devuelve este valor.
- 3 Está prohibido no comprobar errores !!
- 4 El puerto que hemos usado (15001) lo hemos elegido arbitrariamente, pero deberá ser conocido también por el otro interlocutor.
- 5 Las funciones `htons()` y `htonl()` se usan para traducir cantidades de más de 1 byte entre el formato utilizado por nuestra computadora y el utilizado por la red. Existen otras dos funciones para la traducción inversa.

Ejemplo de uso de la estructura *sockaddr_in*

Observar las siguientes peculiaridades:

- 1 Es necesario hacer un cast en el segundo parámetro de `bind()`.
- 2 El tercer parámetro de `bind()` es el tamaño en bytes del segundo parámetro. Lo más cómodo es usar la macro `sizeof` que nos devuelve este valor.
- 3 **Está prohibido no comprobar errores !!**
- 4 El puerto que hemos usado (15001) lo hemos elegido arbitrariamente, pero deberá ser conocido también por el otro interlocutor.
- 5 Las funciones `htons()` y `htonl()` se usan para traducir cantidades de más de 1 byte entre el formato utilizado por nuestra computadora y el utilizado por la red. Existen otras dos funciones para la traducción inversa.

Ejemplo de uso de la estructura *sockaddr_in*

Observar las siguientes peculiaridades:

- 1 Es necesario hacer un cast en el segundo parámetro de `bind()`.
- 2 El tercer parámetro de `bind()` es el tamaño en bytes del segundo parámetro. Lo más cómodo es usar la macro `sizeof` que nos devuelve este valor.
- 3 Está prohibido no comprobar errores !!
- 4 El puerto que hemos usado (15001) lo hemos elegido arbitrariamente, pero deberá ser conocido también por el otro interlocutor.
- 5 Las funciones `htons()` y `htonl()` se usan para traducir cantidades de más de 1 byte entre el formato utilizado por nuestra computadora y el utilizado por la red. Existen otras dos funciones para la traducción inversa.

Ejemplo de uso de la estructura *sockaddr_in*

Observar las siguientes peculiaridades:

- 1 Es necesario hacer un cast en el segundo parámetro de `bind()`.
- 2 El tercer parámetro de `bind()` es el tamaño en bytes del segundo parámetro. Lo más cómodo es usar la macro `sizeof` que nos devuelve este valor.
- 3 Está prohibido no comprobar errores !!
- 4 El puerto que hemos usado (15001) lo hemos elegido arbitrariamente, pero deberá ser conocido también por el otro interlocutor.
- 5 Las funciones `htons()` y `htonl()` se usan para traducir cantidades de más de 1 byte entre el formato utilizado por nuestra computadora y el utilizado por la red. Existen otras dos funciones para la traducción inversa.

Funciones **htons()**, **htonl()**, **ntohs()** y **ntohl()**

Es necesario hacer estas conversiones porque cada máquina puede tener su propio criterio para ordenar los bytes de estas cantidades (Little Endian o Big Endian), pero el criterio de la red es único, en concreto **Big Endian**.

El significado de todas estas funciones es:

- ➊ **htons()**: Host to Network Short. Convierte un valor corto (2 bytes) del formato usado por el host (nuestra máquina) al usado por la red.
- ➋ **htonl()**: Host to Network Long. Convierte un valor largo (4 bytes) del formato usado por el host (nuestra máquina) al usado por la red.
- ➌ **ntohs()**: Network to Host Short. Convierte un valor corto (2 bytes) del formato usado por la red al usado por el host (nuestra máquina).
- ➍ **ntohl()**: Network to Host Long. Convierte un valor largo (4 bytes) del formato usado por la red al usado por el host (nuestra máquina).

Funciones **htons()**, **htonl()**, **ntohs()** y **ntohl()**

Es necesario hacer estas conversiones porque cada máquina puede tener su propio criterio para ordenar los bytes de estas cantidades (Little Endian o Big Endian), pero el criterio de la red es único, en concreto **Big Endian**.

El significado de todas estas funciones es:

- ➊ **htons()**: Host to Network Short. Convierte un valor corto (2 bytes) del formato usado por el host (nuestra máquina) al usado por la red.
- ➋ **htonl()**: Host to Network Long. Convierte un valor largo (4 bytes) del formato usado por el host (nuestra máquina) al usado por la red.
- ➌ **ntohs()**: Network to Host Short. Convierte un valor corto (2 bytes) del formato usado por la red al usado por el host (nuestra máquina).
- ➍ **ntohl()**: Network to Host Long. Convierte un valor largo (4 bytes) del formato usado por la red al usado por el host (nuestra máquina).

Funciones **htons()**, **htonl()**, **ntohs()** y **ntohl()**

Es necesario hacer estas conversiones porque cada máquina puede tener su propio criterio para ordenar los bytes de estas cantidades (Little Endian o Big Endian), pero el criterio de la red es único, en concreto **Big Endian**.

El significado de todas estas funciones es:

- 1 **htons()**: Host to Network Short. Convierte un valor corto (2 bytes) del formato usado por el host (nuestra máquina) al usado por la red.
- 2 **htonl()**: Host to Network Long. Convierte un valor largo (4 bytes) del formato usado por el host (nuestra máquina) al usado por la red.
- 3 **ntohs()**: Network to Host Short. Convierte un valor corto (2 bytes) del formato usado por la red al usado por el host (nuestra máquina).
- 4 **ntohl()**: Network to Host Long. Convierte un valor largo (4 bytes) del formato usado por la red al usado por el host (nuestra máquina).

Funciones **htons()**, **htonl()**, **ntohs()** y **ntohl()**

Es necesario hacer estas conversiones porque cada máquina puede tener su propio criterio para ordenar los bytes de estas cantidades (Little Endian o Big Endian), pero el criterio de la red es único, en concreto **Big Endian**.

El significado de todas estas funciones es:

- 1 **htons()**: Host to Network Short. Convierte un valor corto (2 bytes) del formato usado por el host (nuestra máquina) al usado por la red.
- 2 **htonl()**: Host to Network Long. Convierte un valor largo (4 bytes) del formato usado por el host (nuestra máquina) al usado por la red.
- 3 **ntohs()**: Network to Host Short. Convierte un valor corto (2 bytes) del formato usado por la red al usado por el host (nuestra máquina).
- 4 **ntohl()**: Network to Host Long. Convierte un valor largo (4 bytes) del formato usado por la red al usado por el host (nuestra máquina).

Funciones **htons()**, **htonl()**, **ntohs()** y **ntohl()**

Es necesario hacer estas conversiones porque cada máquina puede tener su propio criterio para ordenar los bytes de estas cantidades (Little Endian o Big Endian), pero el criterio de la red es único, en concreto **Big Endian**.

El significado de todas estas funciones es:

- 1 **htons()**: Host to Network Short. Convierte un valor corto (2 bytes) del formato usado por el host (nuestra máquina) al usado por la red.
- 2 **htonl()**: Host to Network Long. Convierte un valor largo (4 bytes) del formato usado por el host (nuestra máquina) al usado por la red.
- 3 **ntohs()**: Network to Host Short. Convierte un valor corto (2 bytes) del formato usado por la red al usado por el host (nuestra máquina).
- 4 **ntohl()**: Network to Host Long. Convierte un valor largo (4 bytes) del formato usado por la red al usado por el host (nuestra máquina).

El servidor: *listen()* y *accept()*

Después de ejecutar la función *bind()* es necesario llamar a la función *listen()* para que el socket quede marcado por el sistema como listo para recibir conexiones. Esta llamada tiene una función doble:

- Pone el socket en modo **pasivo** a la espera de conexiones.
- Fija el tamaño máximo de la cola de peticiones para ese socket.

La sintaxis de *listen()* es la siguiente:

```
retcod=listen(s, numCola);
```

El servidor: *listen()* y *accept()*

Después de ejecutar la función *bind()* es necesario llamar a la función *listen()* para que el socket quede marcado por el sistema como listo para recibir conexiones. Esta llamada tiene una función doble:

- Pone el socket en modo **pasivo** a la espera de conexiones.
- Fija el tamaño máximo de la cola de peticiones para ese socket.

La sintaxis de *listen()* es la siguiente:

```
retcod=listen(s, numCola);
```

El servidor: *listen()* y *accept()*

- ❶ **s**: Es el descriptor devuelto por *socket()*.
- ❷ **numCola**: Es el máximo número de pedidos de conexión que puede esperar en la cola.
Lo normal es que el servidor esté en un bucle aceptando conexiones de diferentes clientes.
Si hay una conexión ya establecida cuando otro cliente intenta conectarse, este segundo cliente es puesto en una cola, y la próxima vez que el servidor llame a *accept()* se establecerá la conexión con este cliente.
El tamaño máximo que puede tener la cola de pedidos de conexión está determinado para cada sistema por el valor de la constante *SOMAXCONN*.
- ❸ **retcod**: Es el valor retornado por la función.
 - 0 indica que se ha completado con éxito
 - -1 indica que ha ocurrido algún error

El servidor: *listen()* y *accept()*

- 1 **s**: Es el descriptor devuelto por *socket()*.
- 2 **numCola**: Es el máximo número de pedidos de conexión que puede esperar en la cola.
Lo normal es que el servidor esté en un bucle aceptando conexiones de diferentes clientes.
Si hay una conexión ya establecida cuando otro cliente intenta conectarse, este segundo cliente es puesto en una cola, y la próxima vez que el servidor llame a *accept()* se establecerá la conexión con este cliente.
El tamaño máximo que puede tener la cola de pedidos de conexión está determinado para cada sistema por el valor de la constante **SOMAXCONN**.
- 3 **retcod**: Es el valor retornado por la función.
 - 0 indica que se ha completado con éxito
 - -1 indica que ha ocurrido algún error

El servidor: *listen()* y *accept()*

- 1 **s**: Es el descriptor devuelto por *socket()*.
- 2 **num_cola**: Es el máximo número de pedidos de conexión que puede esperar en la cola.
Lo normal es que el servidor esté en un bucle aceptando conexiones de diferentes clientes.
Si hay una conexión ya establecida cuando otro cliente intenta conectarse, este segundo cliente es puesto en una cola, y la próxima vez que el servidor llame a *accept()* se establecerá la conexión con este cliente.
El tamaño máximo que puede tener la cola de pedidos de conexión está determinado para cada sistema por el valor de la constante SOMAXCONN.
- 3 **retcod**: Es el valor retornado por la función.
 - 0 indica que se ha completado con éxito
 - -1 indica que ha ocurrido algún error

El servidor: *listen()* y *accept()*

- 1 **s**: Es el descriptor devuelto por *socket()*.
- 2 **num_cola**: Es el máximo número de pedidos de conexión que puede esperar en la cola.
Lo normal es que el servidor esté en un bucle aceptando conexiones de diferentes clientes.
Si hay una conexión ya establecida cuando otro cliente intenta conectarse, este segundo cliente es puesto en una cola, y la próxima vez que el servidor llame a *accept()* se establecerá la conexión con este cliente.
El tamaño máximo que puede tener la cola de pedidos de conexión está determinado para cada sistema por el valor de la constante *SOMAXCONN*.
- 3 **retcod**: Es el valor retornado por la función.
 - 0 indica que se ha completado con éxito
 - -1 indica que ha ocurrido algún error

El servidor: *listen()* y *accept()*

- 1 **s**: Es el descriptor devuelto por *socket()*.
- 2 **num_cola**: Es el máximo número de pedidos de conexión que puede esperar en la cola.
Lo normal es que el servidor esté en un bucle aceptando conexiones de diferentes clientes.
Si hay una conexión ya establecida cuando otro cliente intenta conectarse, este segundo cliente es puesto en una cola, y la próxima vez que el servidor llame a *accept()* se establecerá la conexión con este cliente.
El tamaño máximo que puede tener la cola de pedidos de conexión está determinado para cada sistema por el valor de la constante SOMAXCONN.
- 3 **retcod**: Es el valor retornado por la función.
 - 0 indica que se ha completado con éxito
 - -1 indica que ha ocurrido algún error

El servidor: listen() y **accept()**

Una vez que se ejecuta la función `listen()`, el servidor debe quedar bloqueado esperando que un cliente intente conectarse. Esto lo hace ejecutando la función `accept()`.

```
n_sock=accept(s, quien, l_quien);
```

- ❶ `n_sock`: Descriptor de archivo (tipo `int`) y es el que se usará en las funciones `read()` o `write()` para transmitir la información a través del socket.
- ❷ `s`: Es el descriptor devuelto por `socket()` en el que estamos esperando las conexiones
- ❸ `quien`: Puntero a estructura de tipo `sockaddr_in` y devuelve información sobre el cliente que se ha conectado (dirección IP y el número de puerto del cliente). `accept` aceptará conexiones de cualquier proceso que intente conectarse con la dirección de nuestro socket. Si no nos interesa la información de quién se ha conectado, podemos pasar `NULL` como segundo parámetro de `accept`.
- ❹ `l_quien`: Es un puntero a entero a través del cual se nos devuelve el tamaño en bytes del parámetro anterior. Pondremos `NULL` si hemos puesto `NULL` en el parámetro anterior.

El servidor: listen() y **accept()**

Una vez que se ejecuta la función `listen()`, el servidor debe quedar bloqueado esperando que un cliente intente conectarse. Esto lo hace ejecutando la función `accept()`.

```
n_sock=accept(s, quien, l_quien);
```

- 1 **n_sock**: Descriptor de archivo (tipo int) y es el que se usará en las funciones `read()` o `write()` para transmitir la información a través del socket.
- 2 **s**: Es el descriptor devuelto por `socket()` en el que estamos esperando las conexiones
- 3 **quien**: Puntero a estructura de tipo `sockaddr_in` y devuelve información sobre el cliente que se ha conectado (**dirección IP** y el **número de puerto del cliente**). `accept` aceptará conexiones de cualquier proceso que intente conectarse con la dirección de nuestro socket. Si no nos interesa la información de quién se ha conectado, podemos pasar NULL como segundo parámetro de `accept`.
- 4 **l_quien**: Es un puntero a entero a través del cual se nos devuelve el tamaño en bytes del parámetro anterior. Pondremos NULL si hemos puesto NULL en el parámetro anterior.

El servidor: listen() y **accept()**

Una vez que se ejecuta la función `listen()`, el servidor debe quedar bloqueado esperando que un cliente intente conectarse. Esto lo hace ejecutando la función `accept()`.

```
n_sock=accept(s, quien, l_quien);
```

- 1 **n_sock**: Descriptor de archivo (tipo int) y es el que se usará en las funciones `read()` o `write()` para transmitir la información a través del socket.
- 2 **s**: Es el descriptor devuelto por `socket()` en el que estamos esperando las conexiones
- 3 **quien**: Puntero a estructura de tipo `sockaddr_in` y devuelve información sobre el cliente que se ha conectado (**dirección IP** y el **número de puerto del cliente**). `accept` aceptará conexiones de cualquier proceso que intente conectarse con la dirección de nuestro socket. Si no nos interesa la información de quién se ha conectado, podemos pasar NULL como segundo parámetro de `accept`.
- 4 **l_quien**: Es un puntero a entero a través del cual se nos devuelve el tamaño en bytes del parámetro anterior. Pondremos NULL si hemos puesto NULL en el parámetro anterior.

El servidor: listen() y **accept()**

Una vez que se ejecuta la función `listen()`, el servidor debe quedar bloqueado esperando que un cliente intente conectarse. Esto lo hace ejecutando la función `accept()`.

```
n_sock=accept(s, quien, l_quien);
```

- 1 **n_sock**: Descriptor de archivo (tipo int) y es el que se usará en las funciones `read()` o `write()` para transmitir la información a través del socket.
- 2 **s**: Es el descriptor devuelto por `socket()` en el que estamos esperando las conexiones
- 3 **quien**: Puntero a estructura de tipo `sockaddr_in` y devuelve información sobre el cliente que se ha conectado (dirección IP y el número de puerto del cliente). `accept` aceptará conexiones de cualquier proceso que intente conectarse con la dirección de nuestro socket. Si no nos interesa la información de quién se ha conectado, podemos pasar NULL como segundo parámetro de `accept`.
- 4 **l_quien**: Es un puntero a entero a través del cual se nos devuelve el tamaño en bytes del parámetro anterior. Pondremos NULL si hemos puesto NULL en el parámetro anterior.

El servidor: listen() y **accept()**

Una vez que se ejecuta la función `listen()`, el servidor debe quedar bloqueado esperando que un cliente intente conectarse. Esto lo hace ejecutando la función `accept()`.

```
n_sock=accept(s, quien, l_quien);
```

- 1 **n_sock**: Descriptor de archivo (tipo int) y es el que se usará en las funciones `read()` o `write()` para transmitir la información a través del socket.
- 2 **s**: Es el descriptor devuelto por `socket()` en el que estamos esperando las conexiones
- 3 **quien**: Puntero a estructura de tipo `sockaddr_in` y devuelve información sobre el cliente que se ha conectado (**dirección IP** y el **número de puerto del cliente**). `accept` aceptará conexiones de cualquier proceso que intente conectarse con la dirección de nuestro socket. Si no nos interesa la información de quién se ha conectado, podemos pasar NULL como segundo parámetro de `accept`.
- 4 **l_quien**: Es un puntero a entero a través del cual se nos devuelve el tamaño en bytes del parámetro anterior. Pondremos NULL si hemos puesto NULL en el parámetro anterior.

Información en sockets tipo stream: **write()** y read()

Una vez que la conexión quedó establecida, el descriptor del socket (`n_sock`) lo usaremos como un descriptor de fichero para las funciones `read()` y `write()`.

La sintaxis de `write()` es la siguiente:

```
retcod=write(n_sock, buffer, l_buffer);
```

- ❶ `n_sock`: Descriptor de socket devuelto por `accept()`
- ❷ `buffer`: Puntero a carácter que apunta a los datos que queremos transmitir.
- ❸ `l_buffer`: Entero que indica la cantidad de bytes a transmitir.
- ❹ `retcod`: Si la función se ha ejecutado correctamente

Información en sockets tipo stream: **write()** y read()

Una vez que la conexión quedó establecida, el descriptor del socket (`n_sock`) lo usaremos como un descriptor de fichero para las funciones `read()` y `write()`.

La sintaxis de `write()` es la siguiente:

```
retcod=write(n_sock, buffer, l_buffer);
```

- 1 **n_sock**: Descriptor de socket devuelto por `accept()`
- 2 **buffer**: Puntero a carácter que apunta a los datos que queremos transmitir.
- 3 **l_buffer**: Entero que indica la cantidad de bytes a transmitir.
- 4 **retcod**: Si la función se ha ejecutado correctamente
 - contendrá el número de bytes transferidos
 - En caso contrario, devolverá el valor -1
 - Si (bytes enviado < bytes que se querían enviar) será necesario volver a escribir en el socket a partir del punto en que se haya detenido la transmisión, hasta completar la cantidad de bytes esperada
 - Si (`l_buffer` = bytes devuelto por `write()`), los datos han llegado a su destino.

Información en sockets tipo stream: **write()** y read()

Una vez que la conexión quedó establecida, el descriptor del socket (`n_sock`) lo usaremos como un descriptor de fichero para las funciones `read()` y `write()`.

La sintaxis de `write()` es la siguiente:

```
retcod=write(n_sock, buffer, l_buffer);
```

- 1 **n_sock**: Descriptor de socket devuelto por `accept()`
- 2 **buffer**: Puntero a carácter que apunta a los datos que queremos transmitir.
- 3 **l_buffer**: Entero que indica la cantidad de bytes a transmitir.
- 4 **retcod**: Si la función se ha ejecutado correctamente
 - contendrá el número de bytes transferidos
 - En caso contrario, devolverá el valor -1
 - Si (bytes enviado < bytes que se querían enviar) será necesario volver a escribir en el socket a partir del punto en que se haya detenido la transmisión, hasta completar la cantidad de bytes esperada
 - Si (l_buffer = bytes devuelto por write()), los datos han llegado a su destino.

Información en sockets tipo stream: **write()** y read()

Una vez que la conexión quedó establecida, el descriptor del socket (`n_sock`) lo usaremos como un descriptor de fichero para las funciones `read()` y `write()`.

La sintaxis de `write()` es la siguiente:

```
retcod=write(n_sock, buffer, l_buffer);
```

- 1 **n_sock**: Descriptor de socket devuelto por `accept()`
- 2 **buffer**: Puntero a carácter que apunta a los datos que queremos transmitir.
- 3 **l_buffer**: Entero que indica la cantidad de bytes a transmitir.
- 4 **retcod**: Si la función se ha ejecutado correctamente
 - contendrá el número de bytes transferidos
 - En caso contrario, devolverá el valor -1
 - Si (bytes enviado < bytes que se querían enviar) será necesario volver a escribir en el socket a partir del punto en que se haya detenido la transmisión, hasta completar la cantidad de bytes esperada
 - Si (l_buffer = bytes devuelto por write()), los datos han llegado a su destino.

Información en sockets tipo stream: **write()** y read()

Una vez que la conexión quedó establecida, el descriptor del socket (`n_sock`) lo usaremos como un descriptor de fichero para las funciones `read()` y `write()`.

La sintaxis de `write()` es la siguiente:

```
retcod=write(n_sock, buffer, l_buffer);
```

- 1 **n_sock**: Descriptor de socket devuelto por `accept()`
- 2 **buffer**: Puntero a carácter que apunta a los datos que queremos transmitir.
- 3 **l_buffer**: Entero que indica la cantidad de bytes a transmitir.
- 4 **retcod**: Si la función se ha ejecutado correctamente
 - contendrá el número de bytes transferidos
 - En caso contrario, devolverá el valor -1
 - Si (bytes enviado < bytes que se querían enviar) será necesario volver a escribir en el socket a partir del punto en que se haya detenido la transmisión, hasta completar la cantidad de bytes esperada
 - Si (`l_buffer` = bytes devuelto por `write()`), los datos han llegado a su destino.

Información en sockets tipo stream: **write()** y read()

Una vez que la conexión quedó establecida, el descriptor del socket (`n_sock`) lo usaremos como un descriptor de fichero para las funciones `read()` y `write()`.

La sintaxis de `write()` es la siguiente:

```
retcod=write(n_sock, buffer, l_buffer);
```

- ❶ **n_sock**: Descriptor de socket devuelto por `accept()`
- ❷ **buffer**: Puntero a carácter que apunta a los datos que queremos transmitir.
- ❸ **l_buffer**: Entero que indica la cantidad de bytes a transmitir.
- ❹ **retcod**: Si la función se ha ejecutado correctamente
 - **contendrá el número de bytes transferidos**
 - En caso contrario, devolverá el valor -1
 - Si (`bytes enviado < bytes que se querían enviar`) será necesario volver a escribir en el socket a partir del punto en que se haya detenido la transmisión, hasta completar la cantidad de bytes esperada
 - Si (`l_buffer = bytes devuelto por write()`), los datos han llegado a su destino.

Información en sockets tipo stream: **write()** y read()

Una vez que la conexión quedó establecida, el descriptor del socket (`n_sock`) lo usaremos como un descriptor de fichero para las funciones `read()` y `write()`.

La sintaxis de `write()` es la siguiente:

```
retcod=write(n_sock, buffer, l_buffer);
```

- ❶ **n_sock**: Descriptor de socket devuelto por `accept()`
- ❷ **buffer**: Puntero a carácter que apunta a los datos que queremos transmitir.
- ❸ **l_buffer**: Entero que indica la cantidad de bytes a transmitir.
- ❹ **retcod**: Si la función se ha ejecutado correctamente
 - contendrá el número de bytes transferidos
 - **En caso contrario, devolverá el valor -1**
 - Si (`bytes enviado < bytes que se querían enviar`) será necesario volver a escribir en el socket a partir del punto en que se haya detenido la transmisión, hasta completar la cantidad de bytes esperada
 - Si (`l_buffer = bytes devuelto por write()`), los datos han llegado a su destino.

Información en sockets tipo stream: **write()** y read()

Una vez que la conexión quedó establecida, el descriptor del socket (`n_sock`) lo usaremos como un descriptor de fichero para las funciones `read()` y `write()`.

La sintaxis de `write()` es la siguiente:

```
retcod=write(n_sock, buffer, l_buffer);
```

- 1 **n_sock**: Descriptor de socket devuelto por `accept()`
- 2 **buffer**: Puntero a carácter que apunta a los datos que queremos transmitir.
- 3 **l_buffer**: Entero que indica la cantidad de bytes a transmitir.
- 4 **retcod**: Si la función se ha ejecutado correctamente
 - contendrá el número de bytes transferidos
 - En caso contrario, devolverá el valor -1
 - Si (bytes enviado < bytes que se querían enviar) será necesario volver a escribir en el socket a partir del punto en que se haya detenido la transmisión, hasta completar la cantidad de bytes esperada
 - Si (`l_buffer` = bytes devuelto por `write()`), los datos han llegado a su destino.

Información en sockets tipo stream: **write()** y read()

Una vez que la conexión quedó establecida, el descriptor del socket (`n_sock`) lo usaremos como un descriptor de fichero para las funciones `read()` y `write()`.

La sintaxis de `write()` es la siguiente:

```
retcod=write(n_sock, buffer, l_buffer);
```

- ❶ **n_sock**: Descriptor de socket devuelto por `accept()`
- ❷ **buffer**: Puntero a carácter que apunta a los datos que queremos transmitir.
- ❸ **l_buffer**: Entero que indica la cantidad de bytes a transmitir.
- ❹ **retcod**: Si la función se ha ejecutado correctamente
 - contendrá el número de bytes transferidos
 - En caso contrario, devolverá el valor -1
 - Si (`bytes enviado < bytes que se querían enviar`) será necesario volver a escribir en el socket a partir del punto en que se haya detenido la transmisión, hasta completar la cantidad de bytes esperada
 - Si (`l_buffer = bytes devuelto por write()`), los datos han llegado a su destino.

Información en sockets tipo stream: **write()** y **read()**

Para el caso de **read()** la sintaxis es la siguiente:

```
retcod=read(n_sock, buffer, l_buffer);
```

- ❶ **n_sock**: Descriptor de socket devuelto por **accept()**
 - ❷ **buffer**: Puntero a carácter que apunta a la memoria donde se dejarán los datos leídos del socket
 - ❸ **l_buffer**: Tamaño de la memoria reservada para leer datos del socket. Si llegan más datos de los que caben en nuestro buffer, quedarán a la espera de la siguiente llamada a **read()**.
 - ❹ **retcod**: La función devuelve un valor de:
 - El número de bytes leídos
 - En caso de error un número negativo
 - 0 significa que la conexión ha sido cerrada por el otro interlocutor (es como si hubiéramos leído un '\0'). La conexión se ha podido cerrar por cualquiera y podemos cerrar el socket.
- El código de ejemplo de la siguiente diapositiva muestra la implementación de un servidor de sockets.

Información en sockets tipo stream: **write()** y **read()**

Para el caso de **read()** la sintaxis es la siguiente:

```
retcod=read(n_sock, buffer, l_buffer);
```

- 1 **n_sock**: Descriptor de socket devuelto por **accept()**
- 2 **buffer**: Puntero a carácter que apunta a la memoria donde se dejarán los datos leídos del socket
- 3 **l_buffer**: Tamaño de la memoria reservada para leer datos del socket. Si llegan más datos de los que caben en nuestro buffer, quedarán a la espera de la siguiente llamada a **read()**.
- 4 **retcod**: La función devuelve un valor de
 - El número de bytes leídos
 - En caso de error un número negativo
 - 0 significa que la conexión ha sido cerrada por el otro interlocutor (es como si hubiéramos leído un `\0`), la comunicación puede darse por terminada y podemos cerrar el socket.
 - Si (`bytes leídos < bytes esperados`) será necesario volver a leer del socket hasta que hayamos recibido la cantidad de bytes esperada

Información en sockets tipo stream: **write()** y **read()**

Para el caso de **read()** la sintaxis es la siguiente:

```
retcod=read(n_sock, buffer, l_buffer);
```

- 1 **n_sock**: Descriptor de socket devuelto por **accept()**
- 2 **buffer**: Puntero a carácter que apunta a la memoria donde se dejarán los datos leídos del socket
- 3 **l_buffer**: Tamaño de la memoria reservada para leer datos del socket. Si llegan más datos de los que caben en nuestro buffer, quedarán a la espera de la siguiente llamada a **read()**.
- 4 **retcod**: La función devuelve un valor de
 - El número de bytes leídos
 - En caso de error un número negativo
 - 0 significa que la conexión ha sido cerrada por el otro interlocutor (es como si hubiéramos leído un `\0`), la comunicación puede darse por terminada y podemos cerrar el socket.
 - Si (`bytes leídos < bytes esperados`) será necesario volver a leer del socket hasta que hayamos recibido la cantidad de bytes esperada.

Información en sockets tipo stream: **write()** y **read()**

Para el caso de **read()** la sintaxis es la siguiente:

```
retcod=read(n_sock, buffer, l_buffer);
```

- 1 **n_sock**: Descriptor de socket devuelto por **accept()**
- 2 **buffer**: Puntero a carácter que apunta a la memoria donde se dejarán los datos leídos del socket
- 3 **l_buffer**: Tamaño de la memoria reservada para leer datos del socket. Si llegan más datos de los que caben en nuestro buffer, quedarán a la espera de la siguiente llamada a **read()**.
- 4 **retcod**: La función devuelve un valor de
 - El número de bytes leídos
 - En caso de error un número negativo
 - 0 significa que la conexión ha sido cerrada por el otro interlocutor (es como si hubiéramos leído un `\0`), la comunicación puede darse por terminada y podemos cerrar el socket.
 - Si (`bytes leídos < bytes esperados`) será necesario volver a leer del socket hasta que hayamos recibido la cantidad de bytes esperada

Información en sockets tipo stream: **write()** y **read()**

Para el caso de **read()** la sintaxis es la siguiente:

```
retcod=read(n_sock, buffer, l_buffer);
```

- 1 **n_sock**: Descriptor de socket devuelto por **accept()**
- 2 **buffer**: Puntero a carácter que apunta a la memoria donde se dejarán los datos leídos del socket
- 3 **l_buffer**: Tamaño de la memoria reservada para leer datos del socket. Si llegan más datos de los que caben en nuestro buffer, quedarán a la espera de la siguiente llamada a **read()**.
- 4 **retcod**: La función devuelve un valor de
 - El número de bytes leídos
 - En caso de error un número negativo
 - 0 significa que la conexión ha sido cerrada por el otro interlocutor (es como si hubiéramos leído un `\0`), la comunicación puede darse por terminada y podemos cerrar el socket.
 - Si (`bytes leídos < bytes esperados`) será necesario volver a leer del socket hasta que hayamos recibido la cantidad de bytes esperada

Información en sockets tipo stream: **write()** y **read()**

Para el caso de **read()** la sintaxis es la siguiente:

```
retcod=read(n_sock, buffer, l_buffer);
```

- 1 **n_sock**: Descriptor de socket devuelto por **accept()**
- 2 **buffer**: Puntero a carácter que apunta a la memoria donde se dejarán los datos leídos del socket
- 3 **l_buffer**: Tamaño de la memoria reservada para leer datos del socket. Si llegan más datos de los que caben en nuestro buffer, quedarán a la espera de la siguiente llamada a **read()**.
- 4 **retcod**: La función devuelve un valor de
 - **El número de bytes leídos**
 - En caso de error un número negativo
 - 0 significa que la conexión ha sido cerrada por el otro interlocutor (es como si hubiéramos leído un `\0`), la comunicación puede darse por terminada y podemos cerrar el socket.
 - Si (**bytes leídos** < **bytes esperados**) será necesario volver a leer del socket hasta que hayamos recibido la cantidad de bytes esperada

Información en sockets tipo stream: **write()** y **read()**

Para el caso de **read()** la sintaxis es la siguiente:

```
retcod=read(n_sock, buffer, l_buffer);
```

- 1 **n_sock**: Descriptor de socket devuelto por **accept()**
- 2 **buffer**: Puntero a carácter que apunta a la memoria donde se dejarán los datos leídos del socket
- 3 **l_buffer**: Tamaño de la memoria reservada para leer datos del socket. Si llegan más datos de los que caben en nuestro buffer, quedarán a la espera de la siguiente llamada a **read()**.
- 4 **retcod**: La función devuelve un valor de
 - El número de bytes leídos
 - **En caso de error un número negativo**
 - 0 significa que la conexión ha sido cerrada por el otro interlocutor (es como si hubiéramos leído un `\0`), la comunicación puede darse por terminada y podemos cerrar el socket.
 - Si (**bytes leídos < bytes esperados**) será necesario volver a leer del socket hasta que hayamos recibido la cantidad de bytes esperada

Información en sockets tipo stream: `write()` y `read()`

Para el caso de `read()` la sintaxis es la siguiente:

```
retcod=read(n_sock, buffer, l_buffer);
```

- 1 `n_sock`: Descriptor de socket devuelto por `accept()`
- 2 `buffer`: Puntero a carácter que apunta a la memoria donde se dejarán los datos leídos del socket
- 3 `l_buffer`: Tamaño de la memoria reservada para leer datos del socket. Si llegan más datos de los que caben en nuestro buffer, quedarán a la espera de la siguiente llamada a `read()`.
- 4 `retcod`: La función devuelve un valor de
 - El número de bytes leídos
 - En caso de error un número negativo
 - 0 significa que la conexión ha sido cerrada por el otro interlocutor (es como si hubiéramos leído un `\0`), la comunicación puede darse por terminada y podemos cerrar el socket.
 - Si (`bytes leídos < bytes esperados`) será necesario volver a leer del socket hasta que hayamos recibido la cantidad de bytes esperada

Información en sockets tipo stream: `write()` y `read()`

Para el caso de `read()` la sintaxis es la siguiente:

```
retcod=read(n_sock, buffer, l_buffer);
```

- 1 `n_sock`: Descriptor de socket devuelto por `accept()`
- 2 `buffer`: Puntero a carácter que apunta a la memoria donde se dejarán los datos leídos del socket
- 3 `l_buffer`: Tamaño de la memoria reservada para leer datos del socket. Si llegan más datos de los que caben en nuestro buffer, quedarán a la espera de la siguiente llamada a `read()`.
- 4 `retcod`: La función devuelve un valor de
 - El número de bytes leídos
 - En caso de error un número negativo
 - 0 significa que la conexión ha sido cerrada por el otro interlocutor (es como si hubiéramos leído un `\0`), la comunicación puede darse por terminada y podemos cerrar el socket.
 - Si (bytes leídos < bytes esperados) será necesario volver a leer del socket hasta que hayamos recibido la cantidad de bytes esperada

Cerrar la conexión: **close()**

En los sockets tipo stream cuando los interlocutores no van a intercambiar más información conviene cerrar la conexión.

Esto se hace mediante la función `close()`.

Observar que el servidor cerrará el socket por el cual **dialogaban** (el que llamamos `n_sock`), pero no cerrará el socket por el que recibe las peticiones de conexión (el que llamamos `s`) si quiere seguir aceptando más clientes.

Cuando un proceso se **mata** (recibe una señal SIGTERM) o cuando sale normalmente con una llamada a `return()`, todos los sockets se cierran automáticamente por el sistema operativo.

Si cuando cerramos un socket aún quedaban en él datos por transmitir, el sistema operativo nos garantiza que los transmitirá todos antes de cerrar.

Si tras varios intentos no lo logra (por ejemplo, porque el otro interlocutor ha cerrado ya o porque hay un fallo en la red) cerrará finalmente el socket y los datos se perderán.

Cerrar la conexión: **close()**

La sintaxis de `close()` es muy sencilla:

```
retcod=close(socket);
```

- 1 **socket**: Descriptor del socket que queremos cerrar. Normalmente se tratará del valor devuelto por `accept()` (`n_sock` para nuestro caso). El socket `s` inicial no suele ser cerrado explícitamente, sino que se deja que el sistema lo cierre cuando nuestro proceso muere.
- 2 **retcod**: Valor de retorno.
 - 0 si la operación se ha completado con éxito
 - -1 si se ha producido algún tipo de error.

Cerrar la conexión: **close()**

La sintaxis de `close()` es muy sencilla:

```
retcod=close(socket);
```

- 1 **socket**: Descriptor del socket que queremos cerrar. Normalmente se tratará del valor devuelto por `accept()` (`n_sock` para nuestro caso). El socket `s` inicial no suele ser cerrado explícitamente, sino que se deja que el sistema lo cierre cuando nuestro proceso muere.
- 2 **retcod: Valor de retorno.**
 - 0 si la operación se ha completado con éxito
 - -1 si se ha producido algún tipo de error.

Cerrar la conexión: **close()**

La sintaxis de `close()` es muy sencilla:

```
retcod=close(socket);
```

- 1 **socket**: Descriptor del socket que queremos cerrar. Normalmente se tratará del valor devuelto por `accept()` (`n_sock` para nuestro caso). El socket `s` inicial no suele ser cerrado explícitamente, sino que se deja que el sistema lo cierre cuando nuestro proceso muere.
- 2 **retcod**: Valor de retorno.
 - 0 si la operación se ha completado con éxito
 - -1 si se ha producido algún tipo de error.

Cerrar la conexión: **close()**

La sintaxis de `close()` es muy sencilla:

```
retcod=close(socket);
```

- 1 **socket**: Descriptor del socket que queremos cerrar. Normalmente se tratará del valor devuelto por `accept()` (`n_sock` para nuestro caso). El socket `s` inicial no suele ser cerrado explícitamente, sino que se deja que el sistema lo cierre cuando nuestro proceso muere.
- 2 **retcod**: Valor de retorno.
 - 0 si la operación se ha completado con éxito
 - -1 si se ha producido algún tipo de error.

El cliente: *connect()*

El cliente, una vez ejecutado `bind()`, debe intentar establecer conexión con un servidor.

Para ello debe conocer:

- la dirección IP del servidor
- el puerto del servidor.

La dirección IP puede averiguarse mediante la función `gethostbyname()` si se conoce el nombre del nodo en el cual se está ejecutando el servidor.

El número de puerto se supone ya conocido porque estamos consultando a un servidor **bien conocido** (cuyo número está documentado en manuales) o bien porque estamos conectando con un servidor programado por nosotros mismos.

El cliente: *connect()*

El cliente, una vez ejecutado `bind()`, debe intentar establecer conexión con un servidor.

Para ello debe conocer:

- la dirección IP del servidor
- **el puerto del servidor.**

La dirección IP puede averiguarse mediante la función `gethostbyname()` si se conoce el nombre del nodo en el cual se está ejecutando el servidor.

El número de puerto se supone ya conocido porque estamos consultando a un servidor **bien conocido** (cuyo número está documentado en manuales) o bien porque estamos conectando con un servidor programado por nosotros mismos.

El cliente: *connect()*

La sintaxis de `connect()` es:

```
retcod=connect(s, servidor, l_servidor);
```

- 1 **s**: Es el valor devuelto por `socket()`.
- 2 **servidor**: Puntero a estructura del tipo `sockaddr` en cuyos campos se especificará la dirección del otro interlocutor del socket. En las comunicaciones a través de Internet este parámetro es en realidad de tipo `(struct sockaddr_in *)`, y contiene la dirección IP y el puerto del interlocutor.
- 3 **l_servidor**: Es el tamaño en bytes del parámetro anterior.
- 4 **retcod**: Es el código de retorno de la función
 - 0 si la operación se ha completado con éxito
 - -1 si se ha producido algún tipo de error.

El cliente: *connect()*

La sintaxis de `connect()` es:

```
retcod=connect(s, servidor, l_servidor);
```

- 1 **s**: Es el valor devuelto por `socket()`.
- 2 **servidor**: Puntero a estructura del tipo `sockaddr` en cuyos campos se especificará la dirección del otro interlocutor del socket. En las comunicaciones a través de Internet este parámetro es en realidad de tipo `(struct sockaddr_in *)`, y contiene la dirección IP y el puerto del interlocutor.
- 3 **l_servidor**: Es el tamaño en bytes del parámetro anterior.
- 4 **retcod**: Es el código de retorno de la función
 - 0 si la operación se ha completado con éxito
 - -1 si se ha producido algún tipo de error.

El cliente: **connect()**

La sintaxis de `connect()` es:

```
retcod=connect(s, servidor, l_servidor);
```

- 1 **s**: Es el valor devuelto por `socket()`.
- 2 **servidor**: Puntero a estructura del tipo `sockaddr` en cuyos campos se especificará la dirección del otro interlocutor del socket. En las comunicaciones a través de Internet este parámetro es en realidad de tipo `(struct sockaddr_in *)`, y contiene la dirección IP y el puerto del interlocutor.
- 3 **l_servidor**: Es el tamaño en bytes del parámetro anterior.
- 4 **retcod**: Es el código de retorno de la función
 - 0 si la operación se ha completado con éxito
 - -1 si se ha producido algún tipo de error.

El cliente: **connect()**

La sintaxis de `connect()` es:

```
retcod=connect(s, servidor, l_servidor);
```

- 1 **s**: Es el valor devuelto por `socket()`.
- 2 **servidor**: Puntero a estructura del tipo `sockaddr` en cuyos campos se especificará la dirección del otro interlocutor del socket. En las comunicaciones a través de Internet este parámetro es en realidad de tipo `(struct sockaddr_in *)`, y contiene la dirección IP y el puerto del interlocutor.
- 3 **l_servidor**: Es el tamaño en bytes del parámetro anterior.
- 4 **retcod**: Es el código de retorno de la función
 - 0 si la operación se ha completado con éxito
 - -1 si se ha producido algún tipo de error.

El cliente: **connect()**

La sintaxis de `connect()` es:

```
retcod=connect(s, servidor, l_servidor);
```

- 1 **s**: Es el valor devuelto por `socket()`.
- 2 **servidor**: Puntero a estructura del tipo `sockaddr` en cuyos campos se especificará la dirección del otro interlocutor del socket. En las comunicaciones a través de Internet este parámetro es en realidad de tipo `(struct sockaddr_in *)`, y contiene la dirección IP y el puerto del interlocutor.
- 3 **l_servidor**: Es el tamaño en bytes del parámetro anterior.
- 4 **retcod**: Es el código de retorno de la función
 - 0 si la operación se ha completado con éxito
 - -1 si se ha producido algún tipo de error.

El cliente: **connect()**

La sintaxis de `connect()` es:

```
retcod=connect(s, servidor, l_servidor);
```

- ❶ **s**: Es el valor devuelto por `socket()`.
- ❷ **servidor**: Puntero a estructura del tipo `sockaddr` en cuyos campos se especificará la dirección del otro interlocutor del socket. En las comunicaciones a través de Internet este parámetro es en realidad de tipo `(struct sockaddr_in *)`, y contiene la dirección IP y el puerto del interlocutor.
- ❸ **l_servidor**: Es el tamaño en bytes del parámetro anterior.
- ❹ **retcod**: Es el código de retorno de la función
 - 0 si la operación se ha completado con éxito
 - -1 si se ha producido algún tipo de error.

El cliente: **connect()**

Antes de llamar a `connect()` hay que rellenar correctamente todos los campos de la estructura `sockaddr_in` que hay que pasarle como parámetro.

- 1 `servidor.sin_family` debe contener el valor `AF_INET` para las comunicaciones a través de Internet, que son el caso que nos ocupa.
- 2 `servidor.sin_port` debe contener el número del puerto al que queremos conectarnos. Este tiene que ser el puerto del servidor en cuestión (es el que obtuvo el servidor con la función `bind()`). El número de puerto debe ser convertido al formato red mediante la función `htons()`.
- 3 `servidor.sin_addr.s_addr` Si conocemos la dirección IP de la máquina donde se halla el servidor, podemos llenar este campo traduciéndolo mediante la función `inet_addr()`. Si sólo conocemos el nombre, debemos llamar a `gethostbyname()` para que consiga la dirección IP.

El cliente: **connect()**

Antes de llamar a `connect()` hay que rellenar correctamente todos los campos de la estructura `sockaddr_in` que hay que pasarle como parámetro.

- 1 `servidor.sin_family` debe contener el valor `AF_INET` para las comunicaciones a través de Internet, que son el caso que nos ocupa.
- 2 `servidor.sin_port` debe contener el número del puerto al que queremos conectarnos. Este tiene que ser el puerto del servidor en cuestión (es el que obtuvo el servidor con la función `bind()`). El número de puerto debe ser convertido al formato red mediante la función `htons()`.
- 3 `servidor.sin_addr.s_addr` Si conocemos la dirección IP de la máquina donde se halla el servidor, podemos llenar este campo traduciéndolo mediante la función `inet_addr()`. Si sólo conocemos el nombre, debemos llamar a `gethostbyname()` para que consiga la dirección IP.

El cliente: **connect()**

Antes de llamar a `connect()` hay que rellenar correctamente todos los campos de la estructura `sockaddr_in` que hay que pasarle como parámetro.

- 1 `servidor.sin_family` debe contener el valor `AF_INET` para las comunicaciones a través de Internet, que son el caso que nos ocupa.
- 2 `servidor.sin_port` debe contener el número del puerto al que queremos conectarnos. Este tiene que ser el puerto del servidor en cuestión (es el que obtuvo el servidor con la función `bind()`). El número de puerto debe ser convertido al formato red mediante la función `htons()`.
- 3 `servidor.sin_addr.s_addr` Si conocemos la dirección IP de la máquina donde se halla el servidor, podemos llenar este campo traduciéndolo mediante la función `inet_addr()`. Si sólo conocemos el nombre, debemos llamar a `gethostbyname()` para que consiga la dirección IP.

Ejemplo con *sockets tipo stream*

El ejemplo es lo más sencillo posible.

El servidor

- 1 Espera a que cualquier cliente se conecte con él
- 2 Cuando esto ocurre, lee datos de la conexión y los muestra por pantalla
- 3 Se desconecta de ese cliente, quedando listo para admitir otro.

El cliente

- 1 Se conecta al servidor
- 2 Le envía una cadena de texto
- 3 Se desconecta

El puerto del servidor, que debe ser conocido por el cliente, será el 15002.

El servidor corre en la máquina llamada [carlos-R430-R480-R440](#).

Los clientes pueden conectarse a él desde cualquier máquina.

Ejemplo con *sockets tipo stream*

El ejemplo es lo más sencillo posible.

El servidor

- 1 Espera a que cualquier cliente se conecte con él
- 2 Cuando esto ocurre, lee datos de la conexión y los muestra por pantalla
- 3 Se desconecta de ese cliente, quedando listo para admitir otro.

El cliente

- 1 Se conecta al servidor
- 2 Le envía una cadena de texto
- 3 Se desconecta

El puerto del servidor, que debe ser conocido por el cliente, será el 15002.

El servidor corre en la máquina llamada [carlos-R430-R480-R440](#).

Los clientes pueden conectarse a él desde cualquier máquina.

Ejemplo con *sockets tipo stream*

El ejemplo es lo más sencillo posible.

El servidor

- 1 Espera a que cualquier cliente se conecte con él
- 2 Cuando esto ocurre, lee datos de la conexión y los muestra por pantalla
- 3 Se desconecta de ese cliente, quedando listo para admitir otro.

El cliente

- 1 Se conecta al servidor
- 2 Le envía una cadena de texto
- 3 Se desconecta

El puerto del servidor, que debe ser conocido por el cliente, será el 15002.

El servidor corre en la máquina llamada [carlos-R430-R480-R440](#).

Los clientes pueden conectarse a él desde cualquier máquina.

Ejemplo con *sockets tipo stream*

El ejemplo es lo más sencillo posible.

El servidor

- 1 Espera a que cualquier cliente se conecte con él
- 2 Cuando esto ocurre, lee datos de la conexión y los muestra por pantalla
- 3 Se desconecta de ese cliente, quedando listo para admitir otro.

El cliente

- 1 Se conecta al servidor
- 2 Le envía una cadena de texto
- 3 Se desconecta

El puerto del servidor, que debe ser conocido por el cliente, será el 15002.

El servidor corre en la máquina llamada [carlos-R430-R480-R440](#).

Los clientes pueden conectarse a él desde cualquier máquina.

Ejemplo con *sockets tipo stream*

El ejemplo es lo más sencillo posible.

El servidor

- 1 Espera a que cualquier cliente se conecte con él
- 2 Cuando esto ocurre, lee datos de la conexión y los muestra por pantalla
- 3 Se desconecta de ese cliente, quedando listo para admitir otro.

El cliente

- 1 Se conecta al servidor
- 2 Le envía una cadena de texto
- 3 Se desconecta

El puerto del servidor, que debe ser conocido por el cliente, será el 15002.

El servidor corre en la máquina llamada [carlos-R430-R480-R440](#).

Los clientes pueden conectarse a él desde cualquier máquina.

Ejemplo con *sockets tipo stream*

El ejemplo es lo más sencillo posible.

El servidor

- 1 Espera a que cualquier cliente se conecte con él
- 2 Cuando esto ocurre, lee datos de la conexión y los muestra por pantalla
- 3 Se desconecta de ese cliente, quedando listo para admitir otro.

El cliente

- 1 Se conecta al servidor
- 2 Le envía una cadena de texto
- 3 Se desconecta

El puerto del servidor, que debe ser conocido por el cliente, será el 15002.

El servidor corre en la máquina llamada [carlos-R430-R480-R440](#).

Los clientes pueden conectarse a él desde cualquier máquina.

Ejemplo **servidor.c** con **sockets tipo stream**

librerías necesarias

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

```
#define PUERTO 15002
```

```
int main(void)
{
    int s;
    int n_sock;
    struct sockaddr_in local;
    struct sockaddr_in cliente;
    socklen_t l_cliente;
    char buffer[81];
    int leidos;

    s = socket(PF_INET, SOCK_STREAM, 0);
    if (s < 0) {
        perror("creando socket:");
        return(1);
    }
    local.sin_family = AF_INET;
    local.sin_port = htons(PUERTO);
    local.sin_addr.s_addr = htonl(INADDR_ANY);
    if (bind(s, (struct sockaddr *) &local, sizeof local) < 0) {
        perror("asignando direccion:");
        return(2);
    }
    listen(s, SOMAXCONN);
```

usada para printf
usada para memset
usada para write, read y close
usada para struct sockaddr_in
socket, bind, listen
accept, PF_INET
INADDR_ANY

protocolo

IP local y puerto local

IP remota y puerto remoto

Ejemplo **servidor.c** con **sockets tipo stream**

librerias necesarias

```
{ #include <stdio.h>
  #include <string.h>
  #include <unistd.h>
  #include <netdb.h>
```

definimos puerto
para servidor

```
#define PUERTO 15002
```

```
int main(void)
{
    int s;
    int n_sock;
    struct sockaddr_in local;
    struct sockaddr_in cliente;
    socklen_t l_cliente;
    char buffer[81];
    int leidos;
    s = socket(PF_INET, SOCK_STREAM, 0);
    if (s<0) {
        perror("creando socket:");
        return(1);
    }
    local.sin_family=AF_INET;
    local.sin_port=htons(PUERTO);
    local.sin_addr.s_addr = htonl(INADDR_ANY);
    if (bind(s,(struct sockaddr *) &local, sizeof local)<0) {
        perror("asignando direccion:");
        return(2);
    }
    listen(s, SOMAXCONN);
```

protocolo
IP local y puerto local
IP remota y puerto remoto

Ejemplo **servidor.c** con **sockets tipo stream**

librerias necesarias

```
{ #include <stdio.h>
  #include <string.h>
  #include <unistd.h>

  #include <netdb.h>

  #define PUERTO 15002
```

variable para guardar
el file descriptor
del socket de conexion
usada por **socket()**

```
int main(void)
{
  int s;
  int n_sock;
  struct sockaddr_in local;
  struct sockaddr_in cliente;
  socklen_t l_cliente;
  char buffer[81];
  int leidos;

  s = socket(PF_INET, SOCK_STREAM, 0);
  if (s < 0) {
    perror("creando socket:");
    return(1);
  }
  local.sin_family = AF_INET;
  local.sin_port = htons(PUERTO);
  local.sin_addr.s_addr = htonl(INADDR_ANY);
  if (bind(s, (struct sockaddr *) &local, sizeof local) < 0) {
    perror("asignando direccion:");
    return(2);
  }
  listen(s, SOMAXCONN);
```

protocolo
IP local y puerto local
IP remota y puerto remoto

Ejemplo **servidor.c** con **sockets tipo stream**

librerías necesarias

```
{ #include <stdio.h>
  #include <string.h>
  #include <unistd.h>

  #include <netdb.h>

  #define PUERTO 15002
```

variable para guardar
el file descriptor
del canal de datos
usada por **accept()**

```
int main(void)
{
  int s;
  int n_sock;
  struct sockaddr_in local;
  struct sockaddr_in cliente;
  socklen_t l_cliente;
  char buffer[81];
  int leidos;

  s = socket(PF_INET, SOCK_STREAM, 0);
  if (s < 0) {
    perror("creando socket:");
    return(1);
  }
  local.sin_family = AF_INET;
  local.sin_port = htons(PUERTO);
  local.sin_addr.s_addr = htonl(INADDR_ANY);
  if (bind(s, (struct sockaddr *) &local, sizeof local) < 0) {
    perror("asignando direccion:");
    return(2);
  }
  listen(s, SOMAXCONN);
```

protocolo
IP local y puerto local
IP remota y puerto remoto

Ejemplo **servidor.c** con **sockets tipo stream**

librerias necesarias

```
{ #include <stdio.h>
  #include <string.h>
  #include <unistd.h>
  #include <netdb.h>

  #define PUERTO 15002

  int main(void)
  {
    int s;
    int n_sock;
    struct sockaddr_in local;
    struct sockaddr_in cliente;
    socklen_t l_cliente;
    char buffer[81];
    int leidos;

    s = socket(PF_INET, SOCK_STREAM, 0);
    if (s<0) {
      perror("creando socket:");
      return(1);
    }
    local.sin_family=AF_INET;
    local.sin_port=htons(PUERTO);
    local.sin_addr.s_addr = htonl(INADDR_ANY);
    if (bind(s,(struct sockaddr *) &local, sizeof local)<0) {
      perror("asignando direccion:");
      return(2);
    }
    listen(s, SOMAXCONN);
```

estructura para
colocar datos
con **bind()**

protocolo
IP local y puerto local
IP remota y puerto remoto

Ejemplo **servidor.c** con **sockets tipo stream**

librerias necesarias

```
{ #include <stdio.h>
  #include <string.h>
  #include <unistd.h>
  #include <netdb.h>

  #define PUERTO 15002

  int main(void)
  {
    int s;
    int n_sock;
    struct sockaddr_in local;
    struct sockaddr_in cliente;
    socklen_t l_cliente;
    char buffer[81];
    int leidos;

    s = socket(PF_INET, SOCK_STREAM, 0);
    if (s < 0) {
      perror("creando socket:");
      return(1);
    }
    local.sin_family = AF_INET;
    local.sin_port = htons(PUERTO);
    local.sin_addr.s_addr = htonl(INADDR_ANY);
    if (bind(s, (struct sockaddr *) &local, sizeof local) < 0) {
      perror("asignando direccion:");
      return(2);
    }
    listen(s, SOMAXCONN);
```

estructura para
colocar datos
con **accept()**

protocolo
IP local y puerto local
IP remota y puerto remoto

Ejemplo **servidor.c** con **sockets tipo stream**

librerias necesarias

```
{ #include <stdio.h>
  #include <string.h>
  #include <unistd.h>
  #include <netdb.h>

  #define PUERTO 15002

  int main(void)
  {
    int s;
    int n_sock;
    struct sockaddr_in local;
    struct sockaddr_in cliente;
    socklen_t l_cliente;
    char buffer[81];
    int leidos;

    s = socket(PF_INET, SOCK_STREAM, 0);
    if (s < 0) {
      perror("creando socket:");
      return(1);
    }
    local.sin_family = AF_INET;
    local.sin_port = htons(PUERTO);
    local.sin_addr.s_addr = htonl(INADDR_ANY);
    if (bind(s, (struct sockaddr *) &local, sizeof local) < 0) {
      perror("asignando direccion:");
      return(2);
    }
    listen(s, SOMAXCONN);
```

variable para guardar
tamaño de estructura
cliente usada en accept()

protocolo
IP local y puerto local
IP remota y puerto remoto

Ejemplo **servidor.c** con **sockets tipo stream**

librerias necesarias

```
{ #include <stdio.h>
  #include <string.h>
  #include <unistd.h>

  #include <netdb.h>

  #define PUERTO 15002

  int main(void)
  {
    int s;
    int n_sock;
    struct sockaddr_in local;
    struct sockaddr_in cliente;
    socklen_t l_cliente;
    char buffer[81];
    int leidos;

    s = socket(PF_INET, SOCK_STREAM, 0);
    if (s<0) {
      perror("creando socket:");
      return(1);
    }
    local.sin_family=AF_INET;
    local.sin_port=htons(PUERTO);
    local.sin_addr.s_addr = htonl(INADDR_ANY);
    if (bind(s,(struct sockaddr *) &local, sizeof local)<0) {
      perror("asignando direccion:");
      return(2);
    }
    listen(s, SOMAXCONN);
```

tamaño del buffer
usado en función read()

protocolo
IP local y puerto local
IP remota y puerto remoto

Ejemplo **servidor.c** con **sockets tipo stream**

librerias necesarias

```
{ #include <stdio.h>
  #include <string.h>
  #include <unistd.h>
  #include <netdb.h>

  #define PUERTO 15002

  int main(void)
  {
    int s;
    int n_sock;
    struct sockaddr_in local;
    struct sockaddr_in cliente;
    socklen_t l_cliente;
    char buffer[81];
    int leidos;
    s = socket(PF_INET, SOCK_STREAM, 0);
    if (s<0) {
      perror("creando socket:");
      return(1);
    }
    local.sin_family=AF_INET;
    local.sin_port=htons(PUERTO);
    local.sin_addr.s_addr = htonl(INADDR_ANY);
    if (bind(s,(struct sockaddr *) &local, sizeof local)<0) {
      perror("asignando direccion:");
      return(2);
    }
    listen(s, SOMAXCONN);
```

variable usada para
guardar la cantidad de
caracteres leídos por
la función read()

protocolo
IP local y puerto local
IP remota y puerto remoto

Ejemplo **servidor.c** con **sockets tipo stream**

librerias necesarias

```
{ #include <stdio.h>
  #include <string.h>
  #include <unistd.h>

  #include <netdb.h>

  #define PUERTO 15002
```

función **socket()** que
devuelve su file descriptor
en **s** y protocolo **TCP/IP**

```
int main(void)
{
  int s;
  int n_sock;
  struct sockaddr_in local;
  struct sockaddr_in cliente;
  socklen_t l_cliente;
  char buffer[81];
  int leidos;
  s = socket(PF_INET, SOCK_STREAM, 0);
  if (s<0) {
    perror("creando socket:");
    return(1);
  }
  local.sin_family=AF_INET;
  local.sin_port=htons(PUERTO);
  local.sin_addr.s_addr = htonl(INADDR_ANY);
  if (bind(s, (struct sockaddr *) &local, sizeof local)<0) {
    perror("asignando direccion:");
    return(2);
  }
  listen(s, SOMAXCONN);
```

TCP/IP

IP local y puerto local

IP remota y puerto remoto

Ejemplo **servidor.c** con **sockets tipo stream**

librerias necesarias

```
{
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <netdb.h>

#define PUERTO 15002

int main(void)
{
    int s;
    int n_sock;
    struct sockaddr_in local;
    struct sockaddr_in cliente;
    socklen_t l_cliente;
    char buffer[81];
    int leidos;
    s = socket(PF_INET, SOCK_STREAM, 0);
    if (s < 0) {
        perror("creando socket:");
        return(1);
    }
    local.sin_family = AF_INET;
    local.sin_port = htons(PUERTO);
    local.sin_addr.s_addr = htonl(INADDR_ANY);
    if (bind(s, (struct sockaddr *) &local, sizeof local) < 0) {
        perror("asignando direccion:");
        return(2);
    }
    listen(s, SOMAXCONN);
}
```

verifica errores al
crear el **socket()**

TCP/IP

IP local y puerto local

IP remota y puerto remoto

Ejemplo **servidor.c** con **sockets tipo stream**

librerias necesarias

```
{ #include <stdio.h>
  #include <string.h>
  #include <unistd.h>

  #include <netdb.h>

  #define PUERTO 15002
```

completa datos en la
estructura `sockaddr_in`
que usa `bind()`

```
int main(void)
{
    int s;
    int n_sock;
    struct sockaddr_in local;
    struct sockaddr_in cliente;
    socklen_t l_cliente;
    char buffer[81];
    int leidos;

    s = socket(PF_INET, SOCK_STREAM, 0);
    if (s < 0) {
        perror("creando socket:");
        return(1);
    }
    local.sin_family = AF_INET;
    local.sin_port = htons(PUERTO);
    local.sin_addr.s_addr = htonl(INADDR_ANY);
    if (bind(s, (struct sockaddr *) &local, sizeof local) < 0) {
        perror("asignando direccion:");
        return(2);
    }
    listen(s, SOMAXCONN);
```

TCP/IP

IP local y puerto local

IP remota y puerto remoto

Ejemplo **servidor.c** con **sockets tipo stream**

librerías necesarias

```
{#include <stdio.h>
#include <string.h>
#include <unistd.h>

#include <netdb.h>

#define PUERTO 15002

int main(void)
{
    int s;
    int n_sock;
    struct sockaddr_in local;
    struct sockaddr_in cliente;
    socklen_t l_cliente;
    char buffer[81];
    int leidos;

    s = socket(PF_INET, SOCK_STREAM, 0);
    if (s < 0) {
        perror("creando socket:");
        return(1);
    }
    local.sin_family = AF_INET;
    local.sin_port = htons(PUERTO);
    local.sin_addr.s_addr = htonl(INADDR_ANY);
    if (bind(s, (struct sockaddr *) &local, sizeof local) < 0) {
        perror("asignando direccion:");
        return(2);
    }
    listen(s, SOMAXCONN);
```

función **bind()** que
completa los campos
IP y puerto local
del socket **s**

TCP/IP
INADDR_ANY y **PUERTO**
IP remota y puerto remoto

Ejemplo **servidor.c** con **sockets tipo stream**

librerias necesarias

```
{#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <netdb.h>

#define PUERTO 15002

int main(void)
{
    int s;
    int n_sock;
    struct sockaddr_in local;
    struct sockaddr_in cliente;
    socklen_t l_cliente;
    char buffer[81];
    int leidos;

    s = socket(PF_INET, SOCK_STREAM, 0);
    if (s<0) {
        perror("creando socket:");
        return(1);
    }
    local.sin_family=AF_INET;
    local.sin_port=htons(PUERTO);
    local.sin_addr.s_addr = htonl(INADDR_ANY);
    if (bind(s,(struct sockaddr *)&local,sizeof local)<0) {
        perror("asignando direccion:");
        return(2);
    }
    listen(s, SOMAXCONN);
```

listen() deja al socket
de conexión s listo para
aceptar conexiones

TCP/IP
INADDR_ANY y PUERTO
IP remota y puerto remoto

Ejemplo **servidor.c** con **sockets tipo stream**

loop donde solo se sale con Ctrl C

accept() completa los datos del cliente en el socket de datos **n_sock**

TCP/IP
INADDR_ANY y PUERTO
IP remota y puerto remoto

```
{ while (1) {  
    printf("Esperando nueva conexion...\n");  
    l_cliente=sizeof (cliente);  
    n_sock=accept(s,(struct sockaddr *)&cliente,&l_cliente);  
    if (n_sock<0) {  
        perror("aceptando:");  
    }  
    else {  
        memset(buffer, 0, sizeof buffer);  
        while ((leidos=read(n_sock,&buffer,sizeof buffer))!=0){  
            if (leidos < 0)  
                perror("leyendo:");  
            else  
                printf("Cadena recibida -> %s\n", buffer);  
            memset(buffer, 0, sizeof buffer);  
        }  
        printf("Conexion terminada. \n");  
        close(n_sock);  
    }  
}
```

Ejemplo **servidor.c** con **sockets tipo stream**

loop donde solo se
sale con *Ctrl C*

inicializa el buffer con 0

TCP/IP
INADDR_ANY y PUERTO
IP remota y puerto remoto

```
{ while (1) {  
    printf("Esperando nueva conexion...\n");  
    l_cliente=sizeof (cliente);  
    n_sock=accept(s,( struct sockaddr *)&cliente,&l_cliente);  
    if (n_sock<0) {  
        perror("aceptando:");  
    }  
    else {  
        memset(buffer, 0, sizeof buffer);  
        while ((leidos=read(n_sock,&buffer,sizeof buffer))!=0){  
            if (leidos < 0)  
                perror("leyendo:");  
            else  
                printf("Cadena recibida -> %s\n", buffer);  
            memset(buffer, 0, sizeof buffer);  
        }  
        printf("Conexion terminada. \n");  
        close(n_sock);  
    }  
}
```

Ejemplo **servidor.c** con **sockets tipo stream**

*loop donde solo se
sale con Ctrl C*

lee datos desde **n_sock**
y los guarda en **buffer**

TCP/IP
INADDR_ANY y PUERTO
IP remota y puerto remoto

```
{ while (1) {  
    printf("Esperando nueva conexion...\n");  
    l_cliente=sizeof (cliente);  
    n_sock=accept(s,( struct sockaddr *)&cliente,&l_cliente);  
    if (n_sock<0) {  
        perror("aceptando:");  
    }  
    else {  
        memset(buffer, 0, sizeof buffer);  
        while ((leidos=read(n_sock,&buffer,sizeof buffer))!=0){  
            if (leidos < 0)  
                perror("leyendo:");  
            else  
                printf("Cadena recibida -> %s\n", buffer);  
            memset(buffer, 0, sizeof buffer);  
        }  
        printf("Conexion terminada. \n");  
        close(n_sock);  
    }  
}
```

Ejemplo **servidor.c** con **sockets tipo stream**

loop donde solo se
sale con *Ctrl C*

cierra el socket de
datos `n_sock`
pero mantiene el
socket de conexión `s`

TCP/IP
`INADDR_ANY` y `PUERTO`
IP remota y puerto remoto

```
{ while (1) {  
    printf("Esperando nueva conexion...\n");  
    l_cliente=sizeof (cliente);  
    n_sock=accept(s,( struct sockaddr *)&cliente,&l_cliente);  
    if (n_sock<0) {  
        perror("aceptando:");  
    }  
    else {  
        memset(buffer, 0, sizeof buffer);  
        while ((leidos=read(n_sock,&buffer,sizeof buffer))!=0){  
            if (leidos < 0)  
                perror("leyendo:");  
            else  
                printf("Cadena recibida -> %s\n", buffer);  
            memset(buffer, 0, sizeof buffer);  
        }  
        printf("Conexion terminada. \n");  
        close(n_sock);  
    }  
}
```


Ejemplo **servidor.c** con **sockets tipo stream**

loop donde solo se
sale con *Ctrl C*

el sistema operativo se
encarga de cerrar el
socket de conexión **s**
cuando termina con *Ctrl C*

TCP/IP
INADDR_ANY y PUERTO
IP remoto y puerto remoto

```
{ while (1) {  
    printf("Esperando nueva conexion...\n");  
    l_cliente=sizeof (cliente);  
    n_sock=accept(s,( struct sockaddr *)&cliente,&l_cliente);  
    if (n_sock<0) {  
        perror("aceptando:");  
    }  
    else {  
        memset(buffer, 0, sizeof buffer);  
        while ((leidos=read(n_sock,&buffer,sizeof buffer))!=0){  
            if (leidos < 0)  
                perror("leyendo:");  
            else  
                printf("Cadena recibida -> %s\n", buffer);  
            memset(buffer, 0, sizeof buffer);  
        }  
        printf("Conexion terminada. \n");  
        close(n_sock);  
    }  
}
```

Ejemplo cliente.c con *sockets tipo stream*

librerias necesarias

```
{  
#include <stdio.h>  
#include <string.h>  
#include <unistd.h>  
  
#include <netdb.h>
```

```
#define PUERTO 15002  
#define DATOS "Un, dos, tres ... probando, probando ..."
```

define *puerto* del servidor
y *datos* a enviar

```
int main(void)  
{  
    int s;  
    struct sockaddr_in local;  
    struct sockaddr_in serv;  
    struct hostent *sv;  
    s = socket(PF_INET, SOCK_STREAM, 0);  
    if (s<0) {  
        perror("creando socket:");  
        return(1);  
    }  
    local.sin_family=AF_INET;  
    local.sin_port=htons(0);  
    local.sin_addr.s_addr = htonl(INADDR_ANY);  
    if (bind(s,(struct sockaddr *) &local, sizeof local)<0){  
        perror("asignando direccion:");  
        return(2);  
    }  
}
```

protocolo
IP cliente y puerto cliente
IP server y puerto server

Ejemplo cliente.c con *sockets tipo stream*

librerias necesarias

variable para guardar
el file descriptor
del socket de conexion
usada por `socket()`

protocolo
IP cliente y puerto cliente
IP server y puerto server

```
{
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#include <netdb.h>

#define PUERTO 15002
#define DATOS "Un, dos, tres... probando, probando . . ."

int main(void)
{
    int s;
    struct sockaddr_in local;
    struct sockaddr_in serv;
    struct hostent *sv;
    s = socket(PF_INET, SOCK_STREAM, 0);
    if (s<0) {
        perror("creando socket:");
        return(1);
    }
    local.sin_family=AF_INET;
    local.sin_port=htons(0);
    local.sin_addr.s_addr = htonl(INADDR_ANY);
    if (bind(s,(struct sockaddr *) &local, sizeof local)<0){
        perror("asignando direccion:");
        return(2);
    }
}
```

Ejemplo cliente.c con *sockets tipo stream*

librerías necesarias

```
{
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#include <netdb.h>

#define PUERTO 15002
#define DATOS "Un, dos, tres... probando, probando . . ."

int main(void)
{
    int s;
    struct sockaddr_in local;
    struct sockaddr_in serv;
    struct hostent *sv;
    s = socket(PF_INET, SOCK_STREAM, 0);
    if (s<0) {
        perror("creando socket:");
        return(1);
    }
    local.sin_family=AF_INET;
    local.sin_port=htons(0);
    local.sin_addr.s_addr = htonl(INADDR_ANY);
    if (bind(s,(struct sockaddr *) &local, sizeof local)<0){
        perror("asignando direccion:");
        return(2);
    }
}
```

estructura para usar
en función `bind()`

protocolo
IP cliente y puerto cliente
IP server y puerto server

Ejemplo cliente.c con *sockets tipo stream*

librerías necesarias

```
{
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#include <netdb.h>

#define PUERTO 15002
#define DATOS "Un, dos, tres... probando, probando . . ."

int main(void)
{
    int s;
    struct sockaddr_in local;
    struct sockaddr_in serv;
    struct hostent *sv;
    s = socket(PF_INET, SOCK_STREAM, 0);
    if (s<0) {
        perror("creando socket:");
        return(1);
    }
    local.sin_family=AF_INET;
    local.sin_port=htons(0);
    local.sin_addr.s_addr = htonl(INADDR_ANY);
    if (bind(s,(struct sockaddr *) &local,sizeof local)<0){
        perror("asignando direccion:");
        return(2);
    }
}
```

estructura para usar
en función connect()

protocolo
IP cliente y puerto cliente
IP server y puerto server

Ejemplo cliente.c con *sockets tipo stream*

librerias necesarias

```
{
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#include <netdb.h>

#define PUERTO 15002
#define DATOS "Un, dos, tres... probando, probando . . ."

int main(void)
{
    int s;
    struct sockaddr_in local;
    struct sockaddr_in serv;
    struct hostent *sv;
    s = socket(PF_INET, SOCK_STREAM, 0);
    if (s<0) {
        perror("creando socket:");
        return(1);
    }
    local.sin_family=AF_INET;
    local.sin_port=htons(0);
    local.sin_addr.s_addr = htonl(INADDR_ANY);
    if (bind(s,(struct sockaddr *) &local, sizeof local)<0){
        perror("asignando direccion:");
        return(2);
    }
}
```

estructura para usar en
función gethostbyname()

protocolo
IP cliente y puerto cliente
IP server y puerto server

Ejemplo cliente.c con *sockets tipo stream*

librerías necesarias

```
{
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#include <netdb.h>

#define PUERTO 15002
#define DATOS "Un, dos, tres... probando, probando . . . ."

int main(void)
{
    int s;
    struct sockaddr_in local;
    struct sockaddr_in serv;
    struct hostent *sv;
    s = socket(PF_INET, SOCK_STREAM, 0);
    if (s < 0) {
        perror("creando socket:");
        return(1);
    }
    local.sin_family = AF_INET;
    local.sin_port = htons(0);
    local.sin_addr.s_addr = htonl(INADDR_ANY);
    if (bind(s, (struct sockaddr *) &local, sizeof local) < 0) {
        perror("asignando direccion:");
        return(2);
    }
}
```

función `socket()` que
devuelve su file descriptor
en `s` y protocolo TCP/IP

TCP/IP
IP cliente y puerto cliente
IP server y puerto server

Ejemplo cliente.c con *sockets tipo stream*

librerias necesarias

```
{
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#include <netdb.h>

#define PUERTO 15002
#define DATOS "Un, dos, tres... probando, probando . . ."

int main(void)
{
    int s;
    struct sockaddr_in local;
    struct sockaddr_in serv;
    struct hostent *sv;
    s = socket(PF_INET, SOCK_STREAM, 0);
    if (s<0) {
        perror("creando socket:");
        return(1);
    }
    local.sin_family=AF_INET;
    local.sin_port=htons(0);
    local.sin_addr.s_addr = htonl(INADDR_ANY);
    if (bind(s,(struct sockaddr *) &local, sizeof local)<0){
        perror("asignando direccion:");
        return(2);
    }
}
```

completa datos en la
estructura `sockaddr_in`
que usa `bind()`

TCP/IP

IP cliente y puerto cliente
IP server y puerto server

Ejemplo cliente.c con *sockets tipo stream*

librerías necesarias

```
{
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#include <netdb.h>

#define PUERTO 15002
#define DATOS "Un, dos, tres... probando, probando . . ."

int main(void)
{
    int s;
    struct sockaddr_in local;
    struct sockaddr_in serv;
    struct hostent *sv;
    s = socket(PF_INET, SOCK_STREAM, 0);
    if (s < 0) {
        perror("creando socket:");
        return (1);
    }
    local.sin_family = AF_INET;
    local.sin_port = htons(0);
    local.sin_addr.s_addr = htonl(INADDR_ANY);
    if (bind(s, (struct sockaddr *) &local, sizeof local) < 0) {
        perror("asignando direccion:");
        return (2);
    }
}
```

función `bind()` que
completa los campos
IP y puerto local
del socket de conexión `s`

`htons(0)` elije un puerto local
libre aleatoriamente

TCP/IP
IP local y puerto local
IP remota y puerto remoto

Ejemplo cliente.c con *sockets tipo stream*

conocer *nombre* o *IP*
donde *esta el servidor*
y el *PUERTO*

obtiene *dirección IP*
conociendo el *nombre*
de la *PC* donde
está el *servidor*

TCP/IP
IP local y *puerto local*
IP remota y *puerto remoto*

```
sv = gethostbyname("carlos-R430-R480-R440");  
if (sv==NULL) {  
    fprintf(stderr, "carlos-R430-R480-R440 \n");  
    return(2);  
}  
memcpy(&serv.sin_addr.s_addr,sv->h_addr,sv->h_length);  
serv.sin_family = AF_INET;  
serv.sin_port = htons(PUERTO);  
if (connect(s, (struct sockaddr *) &serv, sizeof serv)<0) {  
    perror("conectando:");  
    return(3);  
}  
if (write(s, DATOS, sizeof DATOS)<0) {  
    perror("escribiendo el socket:");  
    return(3);  
}  
close(s);  
return(0);  
}
```

Ejemplo cliente.c con *sockets tipo stream*

conocer *nombre* o *IP*
donde esta el servidor
y el *PUERTO*

copia datos *IP* de la
estructura *sv* a la
estructura *serv* que
necesita *connect()*

TCP/IP
IP local y puerto local
IP remota y puerto remoto

```
sv = gethostbyname("carlos-R430-R480-R440");
if (sv==NULL) {
    fprintf(stderr, "carlos-R430-R480-R440 \n");
    return(2);
}
memcpy(&serv.sin_addr.s_addr,sv->h_addr,sv->h_length);
serv.sin_family = AF_INET;
serv.sin_port = htons(PUERTO);
if (connect(s, (struct sockaddr *) &serv, sizeof serv)<0) {
    perror("conectando:");
    return(3);
}
if (write(s, DATOS, sizeof DATOS)<0) {
    perror("escribiendo el socket:");
    return(3);
}
close(s);
return(0);
}
```

Ejemplo cliente.c con *sockets tipo stream*

conocer *nombre* o *IP*
donde *esta el servidor*
y el *PUERTO*

completa los datos
restantes en la
estructura *serv*

TCP/IP
IP local y puerto local
IP remota y puerto remoto

```
sv = gethostbyname("carlos-R430-R480-R440");
if (sv==NULL) {
    fprintf(stderr, "carlos-R430-R480-R440 \n");
    return(2);
}
memcpy(&serv.sin_addr.s_addr,sv->h_addr,sv->h_length);
serv.sin_family = AF_INET;
serv.sin_port = htons(PUERTO);
if (connect(s, (struct sockaddr *) &serv, sizeof serv)<0) {
    perror("conectando:");
    return(3);
}
if (write(s, DATOS, sizeof DATOS)<0) {
    perror("escribiendo el socket:");
    return(3);
}
close(s);
return(0);
}
```

Ejemplo cliente.c con *sockets tipo stream*

conocer *nombre* o *IP*
donde *esta el servidor*
y el *PUERTO*

`connect()` establece
la conexión con el
servidor

TCP/IP
IP local y puerto local
IP remota y puerto remoto

```
sv = gethostbyname("carlos-R430-R480-R440");
if (sv==NULL) {
    fprintf(stderr, "carlos-R430-R480-R440 \n");
    return(2);
}
memcpy(&serv.sin_addr.s_addr,sv->h_addr,sv->h_length);
serv.sin_family = AF_INET;
serv.sin_port = htons(PUERTO);
if (connect(s, (struct sockaddr *) &serv, sizeof serv)<0) {
    perror("conectando:");
    return(3);
}
if (write(s, DATOS, sizeof DATOS)<0) {
    perror("escribiendo el socket:");
    return(3);
}
close(s);
return(0);
}
```

Ejemplo cliente.c con *sockets tipo stream*

conocer *nombre* o *IP*
donde *esta el servidor*
y el *PUERTO*

si la conexión
fué returnosa
ejecuta write()

TCP/IP
IP local y puerto local
IP remota y puerto remoto

```
sv = gethostbyname("carlos-R430-R480-R440");
if (sv==NULL) {
    fprintf(stderr, "carlos-R430-R480-R440 \n");
    return(2);
}
memcpy(&serv.sin_addr.s_addr,sv->h_addr,sv->h_length);
serv.sin_family = AF_INET;
serv.sin_port = htons(PUERTO);
if (connect(s, (struct sockaddr *) &serv, sizeof serv)<0) {
    perror("conectando:");
    return(3);
}
if (write(s, DATOS, sizeof DATOS)<0) {
    perror("escribiendo el socket:");
    return(3);
}
close(s);
return(0);
}
```

Ejemplo cliente.c con *sockets tipo stream*

conocer *nombre* o *IP*
donde *esta el servidor*
y el *PUERTO*

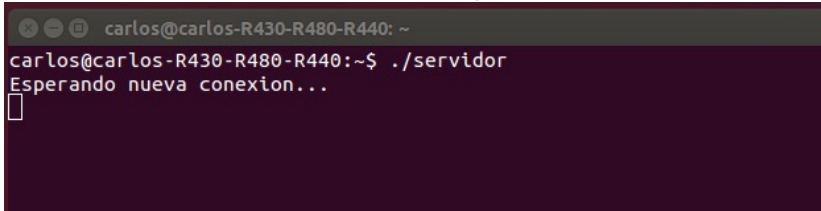
si `write()` no
devuelve error
cerramos el socket `s`
y salimos

TCP/IP
IP local y puerto local
IP remota y puerto remoto

```
sv = gethostbyname("carlos-R430-R480-R440");
if (sv==NULL) {
    fprintf(stderr, "carlos-R430-R480-R440 \n");
    return(2);
}
memcpy(&serv.sin_addr.s_addr,sv->h_addr,sv->h_length);
serv.sin_family = AF_INET;
serv.sin_port = htons(PUERTO);
if (connect(s, (struct sockaddr *) &serv, sizeof serv)<0) {
    perror("conectando:");
    return(3);
}
if (write(s, DATOS, sizeof DATOS)<0) {
    perror("escribiendo el socket:");
    return(3);
}
close(s);
return(0);
}
```

Servidor

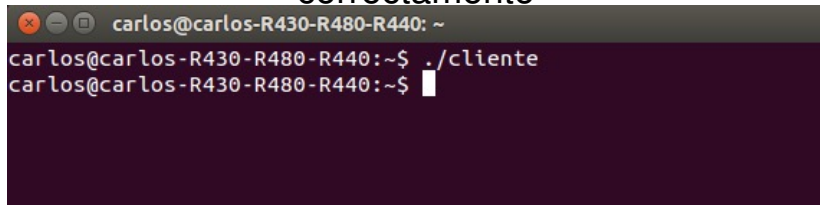
El servidor se encuentra esperando conexiones

A terminal window with a dark background and light text. The title bar shows 'carlos@carlos-R430-R480-R440: ~'. The prompt is 'carlos@carlos-R430-R480-R440:~\$'. The user has entered './servidor' and the output is 'Esperando nueva conexion...'. A cursor is visible on the line below the output.

```
carlos@carlos-R430-R480-R440: ~  
carlos@carlos-R430-R480-R440:~$ ./servidor  
Esperando nueva conexion...  
█
```


Cliente

El cliente ya se ejecutó y el mensaje fué enviado correctamente

A terminal window with a dark background and light text. The title bar shows a window icon, a close button, and the text 'carlos@carlos-R430-R480-R440: ~'. The terminal content shows the prompt 'carlos@carlos-R430-R480-R440:~\$' followed by the command './cliente' on the next line. The prompt 'carlos@carlos-R430-R480-R440:~\$' appears again on the third line, followed by a white cursor block.

```
carlos@carlos-R430-R480-R440: ~  
carlos@carlos-R430-R480-R440:~$ ./cliente  
carlos@carlos-R430-R480-R440:~$ █
```

Servidor

El servidor ya atendió a un cliente y se encuentra esperando nuevas conexiones

```
carlos@carlos-R430-R480-R440: ~  
carlos@carlos-R430-R480-R440:~$ ./servidor  
Esperando nueva conexion...  
Cadena recibida -> Un, dos, tres... probando, probando . . .  
Conexion terminada.  
Esperando nueva conexion...  
█
```