INFORMATICA I

Uso de los Sockets

Ing.Juan Carlos Cuttitta

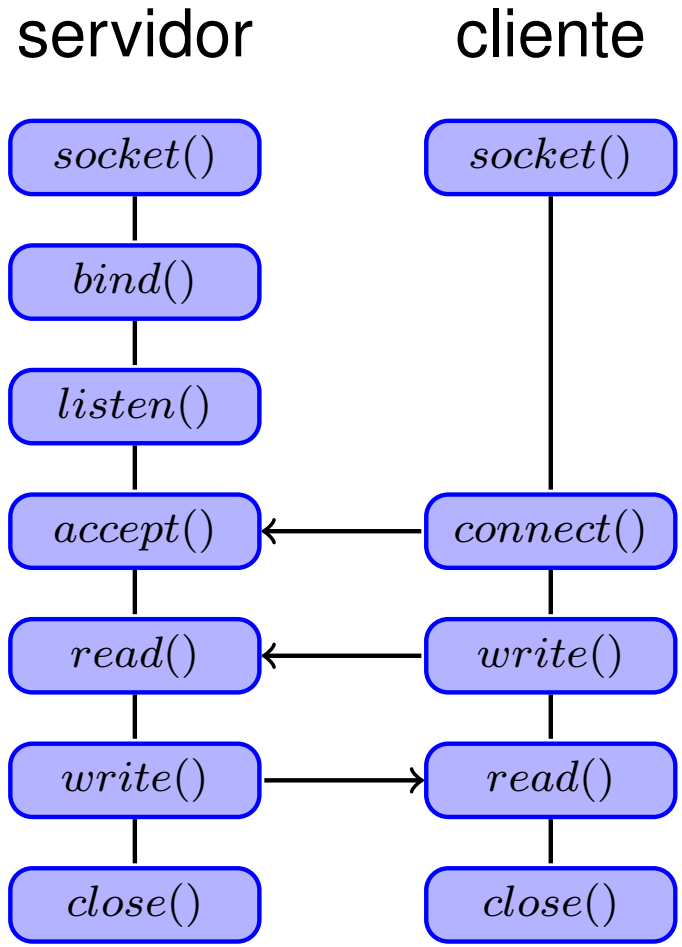
Universidad Tecnológica Nacional
Facultad Regional Buenos Aires
Departamento de Ingeniería Electrónica

13 de octubre de 2020

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.

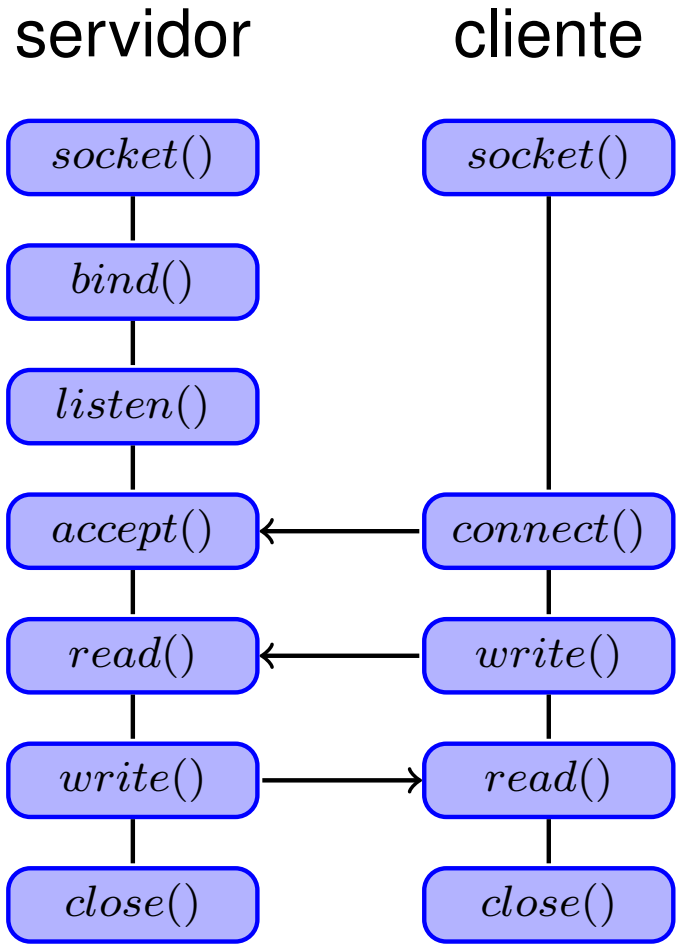
1. Crear el socket mediante la función socket().
2. Asignar una dirección final al socket a la que pueda referirse el otro interlocutor. Función bind().
3. En sockets tipo stream, es necesario conectar con otro socket cuya dirección debemos conocer y se logra con la función connect() para el proceso 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.

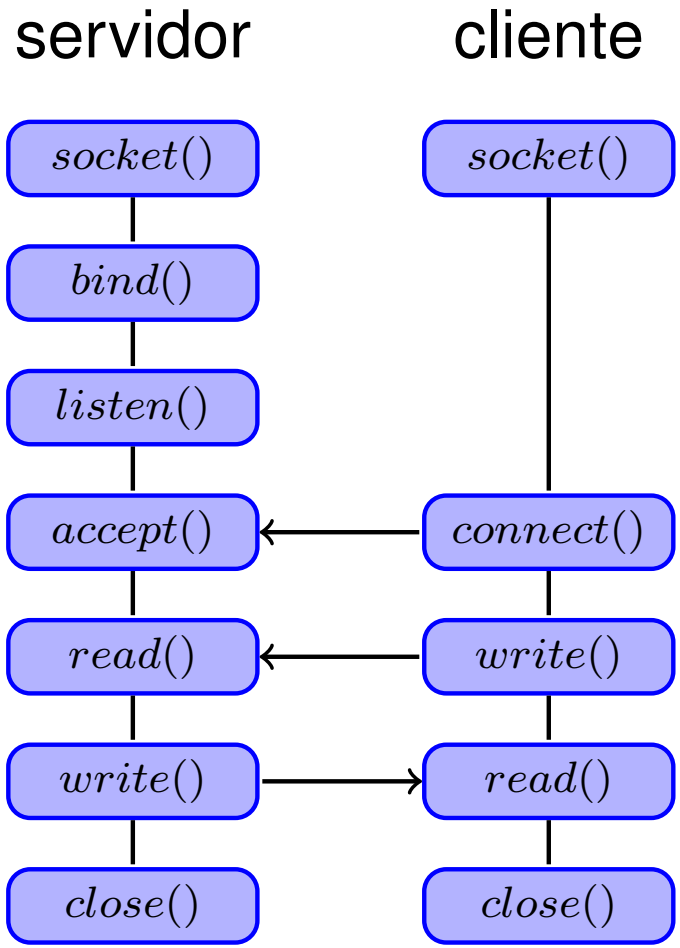
1. Crear el socket mediante la función socket().
2. Asignar una dirección final al socket a la que pueda referirse el otro interlocutor. Función bind().
3. En sockets tipo stream, es necesario conectar con otro socket cuya dirección debemos conocer y se logra con la función connect() para el proceso 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.

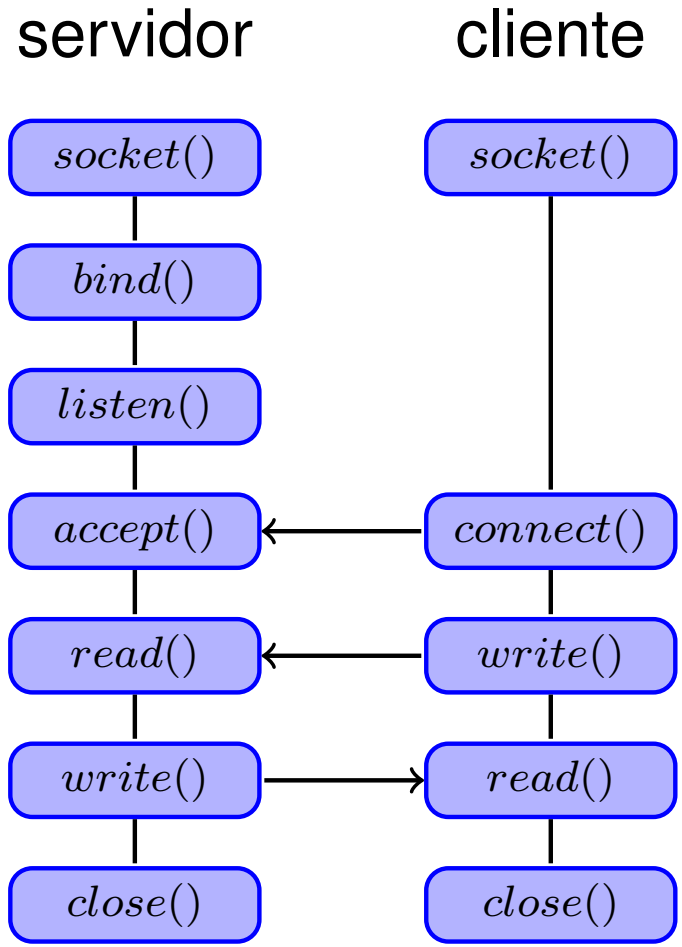
1. Crear el socket mediante la función socket().
2. Asignar una dirección final al socket a la que pueda referirse el otro interlocutor. Función bind().
3. En sockets tipo stream, es necesario conectar con otro socket cuya dirección debemos conocer y se logra con la función connect() para el proceso 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.

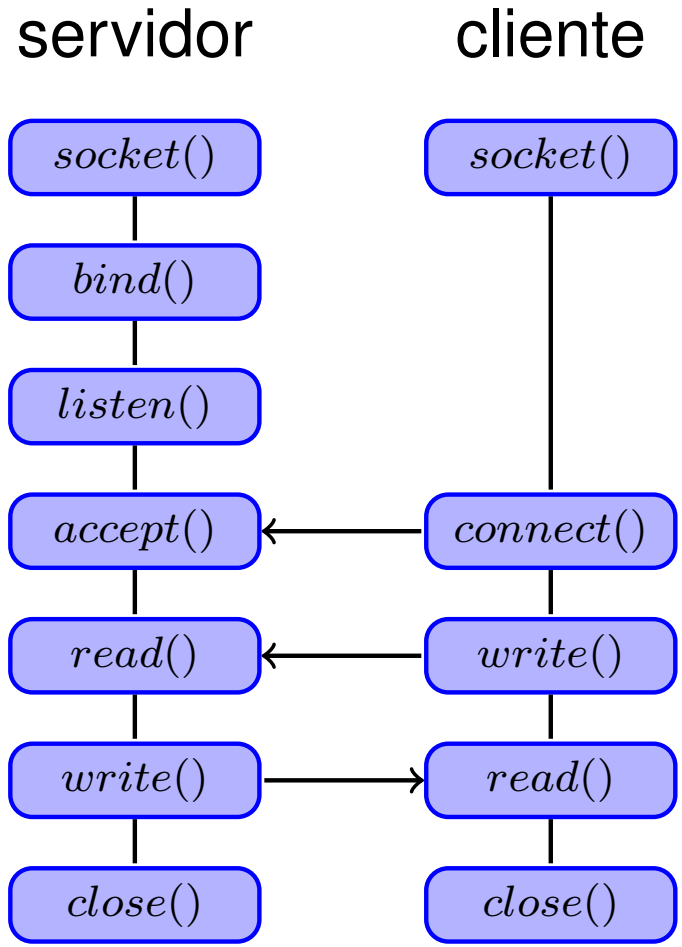
1. Crear el socket mediante la función socket().
2. Asignar una dirección final al socket a la que pueda referirse el otro interlocutor. Función bind().
3. En sockets tipo stream, es necesario conectar con otro socket cuya dirección debemos conocer y se logra con la función connect() para el proceso 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);
```

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);
```

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. 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 );
```

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*

1. 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.
2. 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.
3. sin_addr Es una estructura bastante sencilla.

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

4. sin_zero: Es un campo de relleno que no utilizaremos.

La estructura *sockaddr_in*

1. 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.
2. 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.
3. sin_addr Es una estructura bastante sencilla.

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

4. sin_zero: Es un campo de relleno que no utilizaremos.

La estructura *sockaddr_in*

1. 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.
2. 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.
3. sin_addr Es una estructura bastante sencilla.

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

4. sin_zero: Es un campo de relleno que no utilizaremos.

La estructura *sockaddr_in*

1. 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.
2. 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.
3. sin_addr Es una estructura bastante sencilla.

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

4. sin_zero: Es un campo de relleno que no utilizaremos.

La estructura *sockaddr_in*

1. 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.
2. 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.
3. sin_addr Es una estructura bastante sencilla.

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

4. 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_ANYestamos 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:

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.

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:

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).

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, num_cola);
```

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, num_cola);
```

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()

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);
```

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.

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);
```

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);
```

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);
```

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()**

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

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 creado el socket(), 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 es sabido 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 creado el socket(), 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 es sabido 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);
```

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()*

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 cuestion (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 cuestion (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 cuestion (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

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

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

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

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

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