Ejemplos java y C/linux

Twittear

Tutoriales

- Java
- C/C++ de Linux
- Metodologías y diseño orientado a objetos
- CSS

Enlaces

- Diario de Programación
- Chuwiki
- Micro entradas
- Foro de Java y C++
- Mis proyectos
- Pasatiempos

Programación de sockets en C de Unix/Linux

En los puntos 1 a 3 se cuentan de forma muy sencilla unos conceptos básicos sobre comunicaciones en red y los sockets y la arquitectura cliente/servidor. Si ya sabes de qué va el tema, te lo puedes saltar. De todas formas, no estaría de más un ojo al punto 2, ya que se introduce alguna cosa avanzadilla.

Del 4 al 7 se cuentan por encima los pasos que deberían seguir nuestros programas cliente y servidor para conectarse y hablar entre ellos, las funciones de c/linux a las que deberían irse llamando, así como la configuración necesaria de los ficheros de linux implicados.

Del 8 al 10 se muestra un código de ejemplo que funciona y se explica paso a paso.

El 11 y 12 para introducirse en temas más complejos dentro de sockets.

Temas algo más avanzados: una mini-librería para facilitar el uso de sockets. Atender a varios clientes utilizando select().

INTRODUCCIÓN

En una red de ordenadores hay varios ordenadores que están conectados entre si por un cable. A través de dicho cable pueden transmitirse información. Es claro que deben estar de acuerdo en cómo transmitir esa información, de forma que cualquiera de ellos pueda entender lo que están transmitiendo los otros, de la misma forma que nosotros nos ponemos de acuerdo para hablar en inglés cuando uno es italiano, el otro francés, el otro español y el otro alemán.

Al "idioma" que utilizan los ordenadores para comunicarse cuando están en red se le denomina **protocolo**. Hay muchísimos protocolos de comunicación, entre los cuales el más extendido es el **TCP/IP**. El más extendido porque es el que se utiliza en **Internet**.

Aunque todo esto pueda parecer complicado y que no podemos hacer mucho con ello, lo cierto es que podemos aprovecharlo para comunicar dos programas nuestros que estén corriendo en ordenadores distintos. De hecho, con C en Linux/Unix tenemos una serie de funciones que nos permiten enviar y recibir datos de otros programas, en C o en otros lenguajes de programación, que estén corriendo en otros ordenadores de la misma red.

En este artículo no se pretende dar una descripción detallada y rigurosa del protocolo **TCP/IP** y lo que va alrededor de él. Simplemente se darán unas nociones básicas de manera informal, con la intención de que se pueda comprender un pequeño ejemplo de programación en C.

Está, por tanto, orientado a personas que tengan unos conocimientos básicos de C en Linux y deseen o necesiten comunicar dos programas en C que corren simultáneamente en dos ordenadores distintos conectados en red. El ejemplo propuesto puede servir como guía inicial que se pude complicar todo lo que se desee.

LOS SOCKETS

Una forma de conseguir que dos programas se transmitan datos, basada en el protocolo **TCP/IP**, es la programación de **sockets**. Un **socket** no es más que un "canal de comunicación" entre dos programas que corren sobre ordenadores distintos o incluso en el mismo ordenador.

Desde el punto de vista de programación, un **socket** no es más que un "fichero" que se abre de una manera especial. Una vez abierto se pueden escribir y leer datos de él con las habituales funciones de **read()** y **write()** del lenguaje C. Hablaremos de todo esto con detalle más adelante.

Existen básicamente dos tipos de "canales de comunicación" o **sockets**, los **orientados a conexión** y los **no orientados a conexión**.

En el primer caso ambos programas deben conectarse entre ellos con un **socket** y hasta que no esté establecida correctamente la conexión, ninguno de los dos puede transmitir datos. Esta es la parte **TCP** del protocolo **TCP/IP**, y garantiza que todos los datos van a llegar de un programa al otro correctamente. Se utiliza cuando la información a transmitir es importante, no se puede perder ningún dato y no importa que los programas se queden "bloqueados" esperando o transmitiendo datos. Si uno de los programas está atareado en otra cosa y no atiende la comunicación, el otro quedará bloqueado hasta que el primero lea o escriba los datos.

En el segundo caso, no es necesario que los programas se conecten. Cualquiera de ellos puede transmitir datos en cualquier momento, independientemente de que el otro programa esté "escuchando" o no. Es el llamado protocolo **UDP**, y garantiza que los datos que lleguen son correctos, pero no garantiza que lleguen todos. Se utiliza cuando es muy importante que el programa no se quede bloqueado y no importa que se pierdan datos. Imaginemos, por ejemplo, un programa que está controlando la temperatura de un horno y envía dicha temperatura a un ordenador en una sala de control para que éste presente unos gráficos de temperatura. Obviamente es más importante el control del horno que el perfecto refresco de los gráficos. El programa no se puede quedar bloqueado sin atender al horno simplemente porque el ordenador que muestra los gráficos esté ocupado en otra cosa.

En el ejemplo y a partir de este momento nos referimos únicamente a sockets **TCP**. Los **UDP** son básicamente iguales, aunque hay pequeñas diferencias en la forma de abrirlos y de enviar los mensajes. Puedes ver un ejemplo de programación de socket UDP.

ARQUITECTURA CLIENTE / SERVIDOR

A la hora de comunicar dos programas, existen varias posibilidades para establecer la conexión inicialmente. Una de ellas es la utilizada aquí. Uno de los programas debe estar arrancado y en espera de que otro quiera conectarse a él. Nunca da "el primer paso" en la conexión. Al programa que actúa de esta forma se le conoce como **servidor**. Su nombre se

debe a que normalmente es el que tiene la información que sea disponible y la "sirve" al que se la pida. Por ejemplo, el servidor de páginas web tiene las páginas web y se las envía al navegador que se lo solcite.

El otro programa es el que da el primer paso. En el momento de arrancarlo o cuando lo necesite, intenta conectarse al servidor. Este programa se denomina **cliente**. Su nombre se debe a que es el que solicita información al **servidor**. El navegador de Internet pide la página web al servidor de Internet.

En este ejemplo, el servidor de páginas web se llama **servidor** porque está (o debería de estar) siempre encendido y pendiente de que alguien se conecte a él y le pida una página. El navegador de Internet es el **cliente**, puesto que se arranca cuando nosotros lo arrancamos y solicita conexión con el servidor cuando nosotros escribimos, por ejemplo, www.chuidiang.com

En el juego del Quake, debe haber un **servidor** que es el que tiene el escenario del juego y la situación de todos los jugadores en él. Cuando un nuevo jugador arranca el juego en su ordenador, se conecta al servidor y le pide el escenario del juego para presentarlo en la pantalla. Los movimientos que realiza el jugador se transmiten al servidor y este actualiza escenarios a todos los jugadores.

Resumiendo, **servidor** es el programa que permanece pasivo a la espera de que alguien solicite conexión con él, normalmente, para pedirle algún dato. **Cliente** es el programa que solicita la conexión para, normalmente, pedir datos al servidor.

LA CONEXIÓN

Para poder realizar la conexión entre ambos programas son necesarias varias cosas:

- **Dirección IP del servidor**.

  Cada ordenador de una red tiene asignado un número único, que sirve para identificarle y distinguirle de los demás, de forma que cuando un ordenador quiere hablar con otro, manda la información a dicho número. Es similar a nuestros números de teléfono. Si quiero hablar con mi amigo "Josechu", primero marco su número de teléfono y luego hablo con él.

  El servidor no necesita la dirección de ninguno de los dos ordenadores, al igual que nosotros, para recibir una llamada por teléfono, no necesitamos saber el número de nadie, ni siquiera el nuestro. El cliente sí necesita saber el número del servidor, al igual que nosotros para llamar a alguien necesitamos saber su número de teléfono.

  La dirección IP es un número del estilo 192.100.23.4. ¡Todos lo hemos visto en Internet!. En resumidas cuentas, el cliente debe conocer a qué ordenador desea conectarse. En nuestro navegador de Internet facilitamos la dirección IP del servidor al que queremos conectarnos a través de su nombre (www.chuidiang.com). Obviamente este nombre hay que traducirlo a una dirección IP, pero nuestro navegador e Internet se encargan de eso por nosotros.

- **Servicio** que queremos crear / utilizar.

  Si llamamos a una empresa, puede haber en ella muchas personas, cada una con su extensión de teléfono propia. Normalmente la persona en concreto con la que hablemos nos da igual, lo que queremos es alguien que nos atienda y nos de un determinado "servicio", como recoger una queja, darnos una información, tomarnos nota de un pedido,

etc.

De la misma forma, en un mismo ordenador pueden estar corriendo varios programas servidores, cada uno de ellos dando un servicio distinto. Por ejemplo, un ordenador puede tener un servidor de Quake y un servidor de páginas web corriendo a la vez. Cuando un cliente desea conectarse, debe indicar qué servicio quiere, igual que al llamar a la empresa necesitamos decir la extensión de la persona con la que queremos hablar o, al menos, decir su nombre o el departamento al que pertenece para que la telefonista nos ponga con la persona adecuada.

Por ello, cada servicio dentro del ordenador debe tener un número único que lo identifique (como la extensión de teléfono). Estos números son enteros normales y van de 1 a 65535. Los número bajos, desde 1 a 1023 están reservados para servicios habituales de los sistemas operativos (www, ftp, mail, ping, etc). El resto están a disposición del programador y sirven para cosas como Quake.

Tanto el servidor como el cliente deben conocer el número del servicio al que atienden o se conectan. El servidor le indica al sistema operativo qué servicio quiere atender, al igual que en una empresa el empleado recién contratado (o alguien en su lugar) debe informar a la telefonista en qué extensión se encuentra.

El cliente, cuando llame a la empresa, debe dar el número de extensión (o nombre de empleado), de forma que la telefonista le ponga con la persona adecuada. En el caso del navegador de Internet, estamos indicando el servicio con la www en www.chuidiang.com, servicio de páginas web. También es posible, por ejemplo "ftp.chuidiang.com", si chuidiang.com admite clientes ftp. Nuestro ordenador es lo suficientemente listo como para saber a qué número corresponden esos servicios habituales.

EL SERVIDOR

A partir de este punto comenzamos con lo que es la programación en C de los sockets. Si no tienes unos mínimos conocimientos de C, es mejor que los adquieras antes de seguir.

Con C en Unix/Linux, los pasos que debe seguir un programa servidor son los siguientes:

- **Apertura de un socket**, mediante la función **socket()**. Esta función devuelve un descriptor de fichero normal, como puede devolverlo open(). La función socket() no hace absolutamente nada, salvo devolvernos y preparar un descriptor de fichero que el sistema posteriormente asociará a una conexión en red.
- **Avisar al sistema operativo** de que hemos abierto un socket y queremos que asocie nuestro programa a dicho socket. Se consigue mediante la función **bind()**. El sistema todavía no atenderá a las conexiones de clientes, simplemente anota que cuando empiece a hacerlo, tendrá que avisarnos a nosotros. Es en esta llamada cuando se debe indicar el número de servicio al que se quiere atender.
- Avisar al sistema de que **comience a atender dicha conexión** de red. Se consigue mediante la función **listen()**. A partir de este momento el sistema operativo anotará la conexión de cualquier cliente para pasárnosla cuando se lo pidamos. Si llegan clientes más rápido de lo que somos capaces de atenderlos, el sistema operativo hace una "cola" con ellos y nos los irá pasando según vayamos pidiéndolo.
- Pedir y **aceptar las conexiones** de clientes al sistema operativo. Para ello hacemos una llamada a la función **accept()**. Esta función le indica al sistema operativo que nos dé al siguiente cliente de la cola. Si no hay clientes se quedará bloqueada hasta que algún cliente se conecte.
- **Escribir y recibir datos** del cliente, por medio de las funciones **write()** y **read()**, que son exactamente las mismas que usamos para escribir o leer de un fichero. Obviamente,

tanto cliente como servidor deben saber qué datos esperan recibir, qué datos deben enviar y en qué formato. Puedes ver cómo se pueden poner de acuerdo en estos mensajes en el apartado de mensajes.
- **Cierre de la comunicación** y del socket, por medio de la función **close()**, que es la misma que sirve para cerrar un fichero.

EL CLIENTE

Los pasos que debe seguir un programa cliente son los siguientes:

- **Apertura de un socket**, como el servidor, por medio de la función **socket()**
- **Solicitar conexión** con el servidor por medio de la función **connect()**. Dicha función quedará bloqueada hasta que el servidor acepte nuestra conexión o bien si no hay servidor en el sitio indicado, saldrá dando un error. En esta llamada se debe facilitar la dirección IP del servidor y el número de servicio que se desea.
- **Escribir y recibir datos** del servidor por medio de las funciones **write()** y **read()**.
- **Cerrar la comunicación** por medio de **close()**.

FICHEROS UNIX/LINUX IMPLICADOS

Para la programación de **socket** no es estrictamente necesario ningún fichero. Sabiendo la **dirección IP** y el **número de servicio**, se ponen directamente en código y todo resuelto. Sin embargo, esto no es lo más cómodo ni lo más portable de unos ordenadores a otros.

Hay dos ficheros en Unix/Linux que nos facilitan esta tarea, aunque hay que tener permisos de root para modificarlos. Estos ficheros serían el equivalente a una agenda de teléfonos, en uno tenemos apuntados el nombre de la empresa con su número de teléfono y en el otro fichero el nombre de la persona y su extensión (EmpresaGorda, tlfn 123.456.789; JoseGordo, extensión 2245; ...)

/etc/hosts : Esta es la agenda en la que tenemos las empresas y sus números de teléfono. En este fichero hay una lista de nombres de ordenadores conectados en red y dirección IP de cada uno. Habitualmente en el /etc/hosts del cliente se suele colocar el nombre del servidor y su dirección IP. Luego, desde programa, se hace una llamada a la función **gethostbyname()**, a la que pasándole el nombre del ordenador como una cadena de caracteres, devuelve una estructura de datos entre los que está la dirección IP.

Una línea de lo que puede aparecer en este fichero es la siguiente, en el que vemos la dirección IP y el nombre del ordenador que nos da el servicio de Quake.

```
192.30.10.1 Ordenador_Quake
```

En Windows, este fichero suele encontrarse en C:\WINNT\system32\drivers\etc\hosts.

/etc/services : Este fichero es el equivalente a la agenda donde tenemos apuntados los distintos departamentos/personas de la empresa y sus números de extensión telefónica. En este fichero hay una lista de servicios disponibles, indicando nombre de servicio, número de servicio y tipo (ftp/udp).

Tanto el servidor como el cliente deben tener en este fichero el servicio que están atendiendo / solicitando con el mismo número y tipo de servicio. El nombre puede ser distinto, igual que cada uno en su agenda pone el nombre que quiere, pero no es lo habitual.

Desde programa, tanto cliente como servidor, deben hacer una llamada a la función **getservbyname()**, a la que pasándole el nombre del servicio, devuelve una estructura de datos entre los que está el número de servicio y el tipo.

Un ejemplo de lo que puede aparecer en un fichero /etc/services es el siguiente, en el que vemos en cada línea nombre del servicio, número / tipo y un comentario opcional detrás del símbolo #. "Casualmente", se ve el servicio www, cuyo número de servicio conocido por todos los ordenadores de Internet, es el 80.

```
tftp 69/udp
gopher 70/tcp # Internet Gopher
gopher 70/udp
rje 77/tcp
finger 79/tcp
www 80/tcp http # Worldwide Web HTTP
www 80/udp # HyperText Transfer Protocol
link 87/tcp ttylink
```

En Windows, el ficherito de marras está en C:\WINNT\system32\drivers\etc\services

EJEMPLO

Sentadas las bases de los **sockets**, vamos a ver un pequeño ejemplo.zip de programa **servidor** y **cliente**, realizado con C en Linux. El servidor esperará la conexión del cliente. Una vez conectados, se enviarán una cadena de texto el uno al otro y ambos escribirán en pantalla la cadena recibida.

Para ejecutar el ejemplo no es necesario tener dos ordenadores. Se pude ejecutar el servidor desde una ventana y el cliente desde otra. En la ventana del servidor veremos la cadena que ha enviado el cliente y al revés.

DETALLES DEL SERVIDOR

En este apartado vamos a detallar las llamadas a las funciones del servidor que indicamos anteriormente. Explicaremos con cierto detalle qué parámetros se deben pasar y cual es el resultado de dichas llamadas.

En primer lugar el servidor debe **obtener el número del servicio** al que quiere atender, haciendo la llamada a **getservbyname()**. Esta función devuelve una estructura (en realidad puntero a la estructura) en el que uno de sus campos contiene el número de servicio solicitado.

```
struct servent Puerto; /* Estructura devuelta */

/* La llamada a la función */
Puerto = getservbyname ("Nombre_Servicio", "tcp");
```

Los parámetros de la función son dos cadenas de caracteres. La primera es el **nombre del servicio**, tal cual lo escribimos en el fichero **/etc/services**. El segundo es el tipo de protocolo que queremos usar. "**tcp**" o "**udp**", Dentro de lo que devuelve la función, el número de servicio que nos interesa está en un campo de **Puerto**:

```
Puerto->s_port
```

Ya tenemos todos los datos que necesita el servidor para abrir el socket, así que procedemos a hacerlo. El socket se abre mediante la llamada a la función **socket()** y devuelve un entero que es el **descriptor de fichero** o –1 si ha habido algún error.

```
int Descriptor;
Descriptor = socket (AF_INET, SOCK_STREAM, 0);
if (Descriptor == -1)
    printf ("Error\n");
```

El primer parámetro es **AF_INET** o AF_UNIX para indicar si los clientes pueden estar en otros ordenadores distintos del servidor o van a correr en el mismo ordenador. **AF_INET** vale para los dos casos. **AF_UNIX** sólo para el caso de que el cliente corra en el mismo ordenador que el servidor, pero lo implementa de forma más eficiente. Si ponemos **AF_UNIX**, el resto de las funciones varía ligeramente.

El segundo parámetro indica si el socket es orientado a conexión (**SOCK_STREAM**) o no lo es (**SOCK_DGRAM**). De esta forma podremos hacer sockets de red o de Unix tanto orientados a conexión como no orientados a conexión.

El tercer parámetro indica el **protocolo** a emplear. Habitualmente se pone 0.

Si se ha obtenido un descriptor correcto, se puede indicar al sistema que ya lo tenemos abierto y que vamos a atender a ese servicio. Para ello utilizamos la función **bind()**. El problema de la función **bind()** es que lleva un parámetro bastante complejo que debemos rellenar.

```
struct sockaddr_in Direccion;
Direccion.sin_family = AF_INET;
Direccion.sin_port = Puerto->s_port;
Direccion.sin_addr.s_addr =INADDR_ANY;

if (bind (Descriptor, (struct sockaddr *)&Direccion, sizeof
(Direccion)) == -1)
{
    printf ("Error\n");
}
```

El parámetro que necesita es una estructura **sockaddr**. Lleva varios campos, entre los que es obligatorio rellenar los indicados en el código.

sin_family es el tipo de conexión (por red o interna), igual que el primer parámetro de **socket()**.
sin_port es el número correspondiente al servicio que obtuvimos con **getservbyname()**. El valor está en el campo **s_port** de Puerto.
Finalmente **sin_addr.s_addr** es la dirección del cliente al que queremos atender. Colocando en ese campo el valor **INADDR_ANY**, atenderemos a cualquier cliente.

La llamada a **bind()** lleva tres parámetros:

- **Descriptor del socket** que hemos abierto
- Puntero a la **estructura Direccion** con los datos indicados anteriormente. La estructura admitida por este parámetro es general y valida para cualquier tipo de socket y es del tipo **struct sockaddr**. Cada socket concreto lleva su propia estructura. Los **AF_INET** como este caso llevan **struct sockaddr_in**, los **AF_UNIX** llevan la estructura **struct sockaddr_un**. Por eso, a la hora de pasar el parámetro, debemos hacer un "cast" al tipo **struct sockaddr**. La operación cast es un pequeño truco que admite C. Cuando tenemos un dato de un determinado tipo, por ejemplo, un entero int, es posible convertirlo a otro tipo que nos venga mejor poniendo el nuevo tipo deseado entre paréntesis y detrás la variable del tipo no deseado. Por ejemplo, para convertir de entero a flotante:

```
Variable_Flotante = (float)Variable_Entera;
```

Esto es válido siempre y cuando los dos tipos tengan algún tipo de relación y sea posible convertir uno en el otro. En nuestro ejemplo las estructuras **sockaddr_in** y **sockaddr_un** pueden convertirse sin problemas al tipo **sockaddr**.

- **Longitud** de la estructura Direccion.

La función devuelve -1 en caso de error.

Una vez hecho esto, podemos decir al sistema que empiece a atender las llamadas de los clientes por medio de la función **listen()**

```
if (listen (Descriptor, 1) == -1)
{
    printf ("Error\n");
}
```

La función **listen()** admite dos parámetros, afortunadamente sencillos:

- **Descriptor del socket**.
- **Número máximo de clientes encolados**. Supongamos que recibimos la conexión de un primer cliente y le atendemos. Mientras lo estamos haciendo, se conecta un segundo cliente, al que no atendemos puesto que estamos ejecutando el código necesario para atender al primero. Mientras sucede todo esto, llega un tercer cliente que también se conecta. Estamos atendiendo al primero y tenemos dos en la "lista de espera". El segundo parámetro de **listen()** indica cuántos clientes máximo podemos tener en la lista de espera. Cuando un cliente entra en la lista de espera, su llamada a **connect()** queda bloqueada hasta que se le atiende. Si la lista de espera está llena, el nuevo cliente que llama a **connect()** recibirá un error de dicha función. Algo así como un "vuelva usted mañana".

La función **listen()** devuelve –1 en caso de error.

Con todo esto ya sólo queda recoger los clientes de la lista de espera por medio de la función **accept()**. Si no hay ningún cliente, la llamada quedará bloqueada hasta que lo haya. Esta función devuelve un **descriptor de fichero** que es el que se tiene que usar para "hablar" con el cliente. El descriptor anterior corresponde al servicio y sólo sirve para encolar a los clientes. Digamos que el primer descriptor es el aparato de teléfono de la telefonista de la empresa y el segundo descriptor es el aparato de teléfono del que está atendiendo al cliente.

```
struct sockaddr Cliente;
int Descriptor_Cliente;
int Longitud_Cliente;

Descriptor_Cliente = accept (Descriptor, &Cliente,
&Longitud_Cliente);
if (Descriptor_Cliente == -1)
{
    printf ("Error\n");
}
```

La función **accept()** es otra que lleva un parámetro complejo, pero que no debemos rellenar. Los parámetros que admite son

- **Descriptor del socket** abierto.
- **Puntero a estructura sockaddr**. A la vuelta de la función, esta estructura contendrá la dirección y demás datos del ordenador cliente que se ha conectado a nosotros.

- **Puntero a un entero**, en el que se nos devolverá la longitud útil del campo anterior.

La función devuelve un –1 en caso de error.

Si todo ha sido correcto, ya podemos "hablar" con el cliente. Para ello se utilizan las funciones **read()** y **write()** de forma similar a como se haría con un fichero. Supongamos que sabemos que al conectarse un cliente, nos va a mandar una cadena de cinco caracteres. Para leerla sería

```
int Leido = 0;/* Número de caracteres leídos hasta el momento */
int Aux = 0; /* Guardaremos el valor devuelto por read() */
int Longitud = 5; /* Número de caracteres a leer */
char Datos[5]; /* Buffer donde guardaremos los caracteres */

/* Bucle hasta que hayamos leído todos los caracteres que
estamos esperando */
while (Leido < Longitud)
{
    /* Se leen los caracteres */
    Aux = read (Descriptor, Datos + Leido, Longitud - Leido);

    /* Si hemos conseguido leer algún carácter */
    if (Aux > 0)
    {
        /* Se incrementa el número de caracteres leídos */
        Leido = Leido + Aux;
    }
    else
    {
        /* Si no se ha leído ningún carácter, se comprueba
        la condición de socket cerrado */
        if (Aux == 0)
        {
            printf ("Socket cerrado\n");
        }
        else
        /* y la condición de error */
        if (Aux == -1)
        {
            printf ("Error\n");
        }
    }
}
```

Parece un poco complicado. A ver si somos capaces de aclararlo. La función **read()** admite como parámetros

- **Descriptor del fichero** / socket del que se quiere leer
- **Puntero a char** donde se almacenarán los datos leídos
- **Número de caracteres** que se quieren leer.

La función devuelve –1 si ha habido error, 0 si el socket se ha cerrado o el fichero ha llegado a fin de fichero, o bien el número de caracteres si se ha conseguido leer alguno. En el caso de socket, si no hay errores y el socket sigue abierto y no hay caracteres para leer (no los han enviado desde el otro extremo), la llamada queda bloqueada.

En el caso de leer de socket, tenemos un pequeño problema añadido y es que se leen los caracteres disponibles y se vuelve. Si pedimos 5 caracteres, entra dentro de lo posible que **read()** lea 3 caracteres y vuelva, sin dar error, pero sin leer todos los que hemos pedido. Por ese motivo, al leer de socket es casi obligatorio (totalmente obligatorio si queremos transmitir muchos caracteres de un solo golpe) el **leer con un bucle**, comprobando cada vez cuantos caracteres se han leído y cuantos faltan.

Por este motivo, **read()** va dentro de un **while()** en el que se mira si el total de caracteres leídos es menor que el número de caracteres que queremos leer. La lectura del **read()** debe además pasar como parámetro cada vez la posición dentro del buffer de lectura en la que queremos situar los caracteres **(Datos + Leido)** y la longitud de caracteres a leer, **(Longitud - Leido)** que también cambia según se va leyendo.

En cuanto a escribir caracteres, la función **write()** admite los mismos parámetros, con la excepción de que el **buffer** de datos debe estar previamente relleno con la información que queremos enviar. Tiene el mismo problema, que escribe lo que puede y vuelve, pudiendo no haber escrito toda la información, por lo que hay que hacer un trozo de código similar.

```
int Escrito = 0; /* Núm. de caracteres que ya se han escrito */
int Aux = 0; /* Número de caracteres escritos en cada pasada */
int Longitud = 5; /* Total de caracteres a escribir */
char Datos[] = "Hola"; /* El 5º carácter es el \0 del final */

/* Bucle mientras no se hayan escrito todos los caracteres
deseados */
while (Escrito < Longitud)
{
    /* Se escriben los caracteres */
    Aux = write (Descriptor, Datos + Escrito, Longitud - Escrito);

    /* Si hemos conseguido escribir algún carácter */
    if (Aux > 0)
    {
        /* Incrementamos el número de caracteres escritos */
        Escrito = Escrito + Aux;
    }
    else
    {
        /* Si no hemos podido escribir caracteres,
        comprobamos la condición de socket cerrado */
        if (Aux == 0)
        {
            printf ("Socket cerrado\n");
        }
        else
        /* y la condición de error */
        if (Aux == -1)
        {
            printf ("Error\n");
        }
    }
}
```

En un programa más serio, ni el cliente ni el servidor saben a priori qué es lo que tienen que leer. Normalmente hay una serie de mensajes que los dos conocen precedidos de una "**cabecera**", en la que suelen ir campos del estilo "**Identificador** de mensaje" y "**Longitud**

del mensaje". De esta forma, primero se leen estos dos campos y sabiendo qué mensaje va a llegar y su longitud, se procede a leerlo. A la hora de escribir, primero se manda la cabecera diciendo qué mensaje se va a mandar y qué longitud tiene, y luego se manda el mensaje en sí.

Una vez que se han leído / enviado todos los datos necesarios, se procede a **cerrar el socket**. Para ello se llama a la función **close()**, que admite como parámetro el descriptor del socket que se quiere cerrar.

```
close (Descriptor_Cliente);
```

Normalmente el servidor cierra el descriptor del cliente (Descriptor_Cliente), no el del socket (Descriptor), ya que este se suele dejar abierto para volver a realizar una llamada a **accept()** y sacar al siguiente cliente de la cola. Es como si una vez que un amable señor de una empresa nos ha atendido le dice a la telefonista que no atienda más el teléfono. Esas cosas sólo las hacen los jefes.

DETALLES DEL CLIENTE

Puesto que la escritura / lectura y cierre de sockets es idéntica a la del servidor, únicamente contaremos la apertura del socket que, cómo no, es más compleja que la del servidor, puesto que además del **número de servicio**, debe obtener la **dirección IP** del servidor.

En primer lugar, como en el servidor, obtenemos el número del servicio.

```
struct servent *Puerto;
Puerto = getservbyname ("Nombre_Servicio", "tcp");

if (Puerto == NULL)
{
    printf ("Error\n");
}
```

Después obtenemos la dirección **IP** del servidor. El parámetro que hay que pasar es una cadena de caracteres con el **nombre del servidor**, tal cual lo pusimos en el fichero /etc/hosts. Devuelve los datos del servidor en una estructura hostent.

```
struct hostent *Host;

Host = gethostbyname ("Nombre_Servidor");
if (Host == NULL)
{
    printf ("Error\n");
}
```

Una vez que tenemos todos los datos necesarios, abrimos el socket igual que en el servidor.

```
Descriptor = socket (AF_INET, SOCK_STREAM, 0);

if (Descriptor == -1)
{
    printf ("Error\n");
}
```

Ya tenemos todo lo necesario para solicitar conexión con el servidor. El parámetro de **connect()** ya es conocido y sólo tenemos que rellenarlo, igual que en **bind()** del servidor.

```
struct sockaddr_in Direccion;
Direccion.sin_family = AF_INET;
Direccion.sin_addr.s_addr = ((structin_addr*)(Host->h_addr))-
>s_addr;
Direccion.sin_port = Puerto->s_port;

if (connect (Descriptor, (struct sockaddr *)&Direccion,sizeof
(Direccion)) == -1)
{
    printf ("Error\n");
}
```

La única novedad es que la dirección del servidor se coloca en **Direccion.sin_addr.s_addr** y no como pusimos antes, **INADDR_ANY**. Dicha dirección la tenemos dentro de la estructura obtenida con **gethostbyname()**, en el campo **((structin_addr*)(Host->h_addr))->s_addr**. ¡Vaya por Dios!, ¡Menudo campo!. Nos pasa como dijimos antes, los tipos no son exactamente los mismos y tenemos estructuras anidadas en estructuras, con lo que la sintaxis queda de esa forma tan horrible. La explicación es la siguiente:

La estructura **Host** tiene un campo **h_addr**, de ahí la parte **Host->h_addr**.

Este campo no es del tipo deseado, así que se convierte con un cast a **struct in_addr**, de ahí lo de **(struct in_addr*)(Host->h_addr)**.

Bueno, pues todo esto es a su vez una estructura, de la que nos interesa el campo **s_addr**, así que metemos todo entre paréntesis, cogemos el campo y nos queda lo que tenemos puesto en el código **((structin_addr*)(Host->h_addr))->s_addr**

El resto es exactamente igual que en el servidor.

ALGUNAS CONSIDERACIONES

Puede haber variaciones con otro tipo de sockets.

El ejemplo aquí expuesto funciona correctamente, pero si se quieren hacer otras cosas un poco más serias hay que tener en cuenta varios puntos que indicamos aquí por encima.

En primer lugar el ejemplo es con **sockets orientados a conexión** y de tipo **AF_INET** (para que pueda funcionar con servidor en un ordenador y cliente en otro). Si cambiamos de tipo de socket o de conexión, la idea básica es la misma, pero la sintaxis del código cambia ligeramente. Antes de realizar dicho cambio, conviene saber qué cosas hay que cambiar en el código. Un ejemplo claro, si usamos **sockets no orientados a conexión**, sobra la llamada a **connect()** que hace el cliente y la de **accept()** que hace el servidor, pero en su lugar el cliente debe hacer una llamada a **bind()**. Puedes ver aquí un ejemplo de socket **udp**.

Si usamos sockets **AF_UNIX**, sobra pedir la **dirección IP** del servidor, ya que es el mismo ordenador en el que corren ambos programas.

Organización de los enteros.

En el mundo de los ordenadores, están los micros de intel (y los que los emulan) y los demás. La diferencia, entre otras muchas, es que organizan los enteros de forma distinta; uno pone antes el byte más significativo del entero y el otro lo pone al final. Si conectamos dos ordenadores con sockets, una de las informaciones que se transmiten en la conexión es

un entero con el número de servicio. ¡Ya la hemos liado! hay micros que no se entienden entre sí. Para evitar este tipo de problemas están la funciones **htons()**, **htonl()** y similares. Esta función convierte los enteros a un formato "standard" de red, con lo que se garantiza que nos podemos entender con cualquier entero. Eso implica que algunos de los campos de las estructuras que hemos rellenado arriba, debemos aplicarles esta función antes de realizar la conexión. Si enviamos después enteros con **write()** o los leemos con **read()**, debemos convertirlos y desconvertirlos a **formato red**.

Puedes ver un ejemplo de todo esto al conectar un socket en Java con uno en C. Aunque ambos corran en el mismo ordenador, java tiene su propia máquina virtual, por lo que es como si corriera en su propio microprocesados, distinto de los pentium.

Atención al cliente

Una técnica habitual en el servidor es que cree nuevos procesos (o hilos) para cada cliente. Puedes echar un ojo a la función **fork()**. Si no se quieren crear procesos, se puede usar la función **select()**. A esta función se le dicen todos los sockets que estamos atendiendo y cuando la llamemos, nos quedamos bloqueados hasta que en alguno de ellos haya "actividad". Esto nos evita estar en un bucle mirando todos los sockets clientes uno por uno para ver si alguno quiere algo.

Se suelen crear procesos con **fork()** cuando el servidor no es capaz de atender todas las peticiones de los clientes a suficiente velocidad. Al tener un proceso dedicado a cada cliente, se puede atender a varios "simultaneamente". Se suele usar **select()** cuando podemos atender a los clientes lo suficientemente rápido como para hacerlo de uno en uno, sin hacer esperar demasiado a nadie.

Poniendo un ejemplo, si en una ventanilla de un banco "gotean" los clientes y se despacha rápido a cada uno, basta con una única ventanilla y una persona en ella que "duerma disimuladamente" mientras llega un cliente. Si los clientes llegan a mogollón y se tarda en atender a cada uno de ellos, es mejor que haya muchas ventanillas, lo ideal, una por cada cliente.

BIBLIOGRAFÍA

Un excelente libro para programación avanzada en C sobre entorno Unix es "UNIX, Programación avanzada" de Fco. Manuel Márquez García, editorial ra-ma.

Tienes una guia bastante simple, pero más detallada de sockets en http://www.arrakis.es/~dmrq/beej/index.html

Estadísticas y comentarios

Numero de visitas desde el 4 Feb 2007:

- Esta pagina este mes: 41
- Total de esta pagina: 372626
- Total del sitio: 19739017