Sistemas Distribuidos
Grado en Ingeniería Informática en Sistemas de Información

Enseñanzas de Prácticas y Desarrollo
EPD 1: Sockets I

Objetivos

1. Aprender a intercambiar mensajes entre procesos distribuidos.
2. Aprender a manejar correctamente los sockets.
3. Diferenciar entre protocolos UDP y TCP.

Conceptos

Un socket es un mecanismo de comunicación punto a punto entre dos procesos sobre el protocolo TCP/IP. 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 podemos leer y escribir con las funciones específicas para tal fin.

Existen dos tipos de comunicación: la orientada a la conexión y la no orientada a la conexión. En el primer caso se debe establecer la conexión entre ambos procesos mediante un socket para poder transmitir. En este caso se utiliza el protocolo TCP, el cual es un protocolo confiable que asegura que los datos transmitidos llegan de un proceso a otro correctamente. En el segundo caso se utiliza el protocolo UDP, que garantiza que los datos que llegan son correctos pero no que lleguen todos. En este caso un proceso envía datos sin la necesidad de que el otro se encuentre escuchando.

En nuestro caso uno de los procesos será el **Servidor** y se encontrará en estado de escucha a la espera de recibir una conexión del proceso **Cliente**. Para realizar una conexión entre un cliente y un servidor mediante sockets debemos conocer la dirección IP de la máquina servidor y el puerto que identifica el servicio al cual nos queremos conectar (desde el 1 al 1023 están reservados para servicios conocidos quedando libres hasta el 65535).

Por tanto, los pasos necesarios para crear una aplicación cliente/servidor basada en sockets serán:

1. Programación de un servidor.
2. Programación de un cliente.

Para esta EPD utilizaremos el lenguaje de programación C, sobre el sistema operativo Linux.

En la Figura 1, se muestra en esquema general de las funciones implicadas en una comunicación concurrente mediante sockets. Como se puede ver, con fork() conseguiremos la concurrencia y con las demás funciones el establecimiento y liberación de la comunicación, así como el envío y recepción de datos

Implementación del servidor

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

La función socket sirve para abrir una conexión y devuelve un descriptor al fichero o -1 si se ha producido un error. Esta función posee tres parámetros.

1. El primer parámetro es AF_INET o AF_UNIX e indica 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 (utiliza protocolos tales como TCP o UDP). AF_UNIX sólo para el caso en el 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.
2. El segundo parámetro indica si el socket está orientado a la conexión (SOCK_STREAM) o no lo está (SOCK_DGRAM).
3. El tercer parámetro indica el protocolo a emplear. Habitualmente se pone 0 dejando al sistema que se encargue de la elección del protocolo.

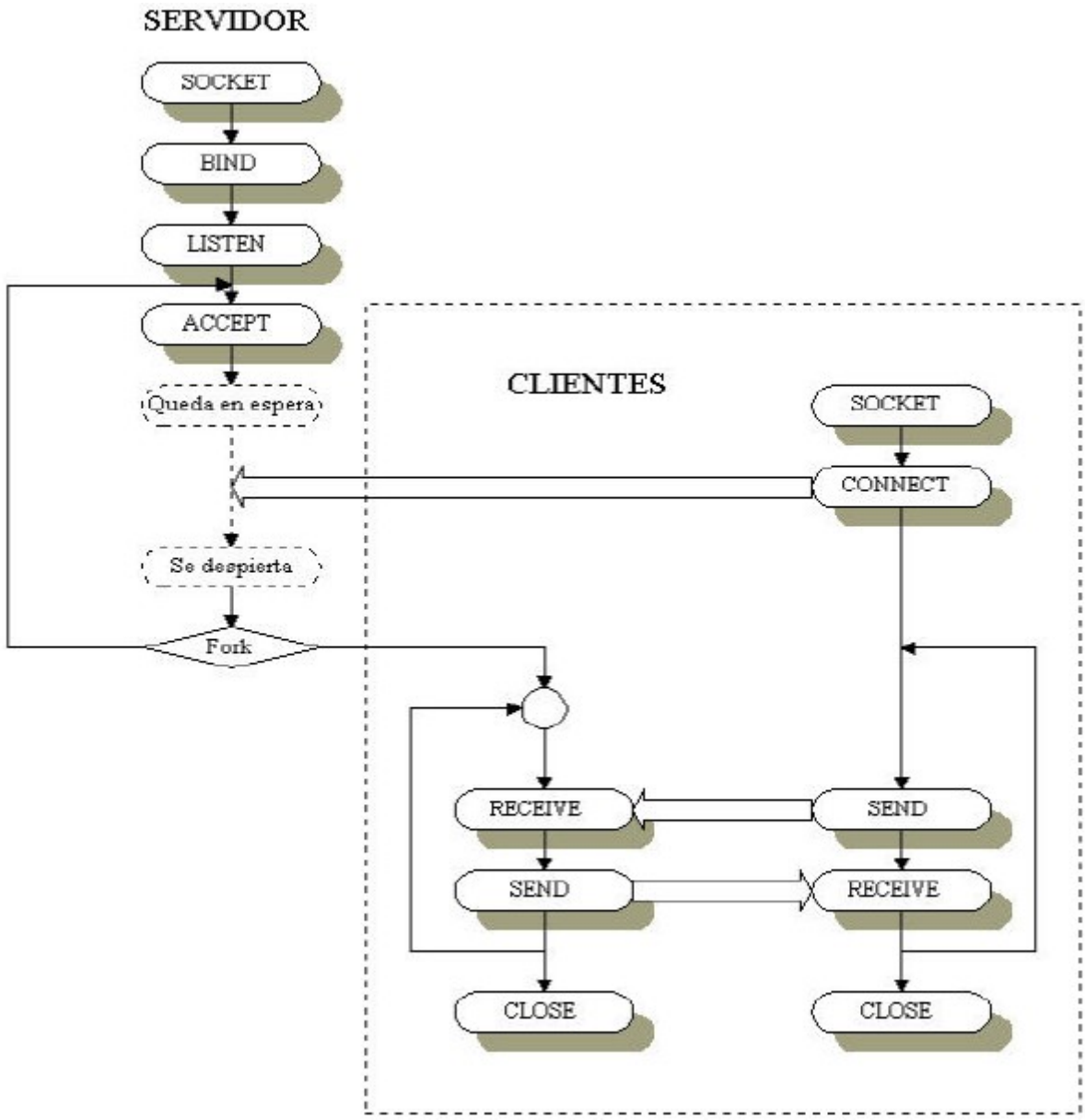

Figura 1. Esquema de comunicación concurrente mediante sockets.

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

1. sin_family: es el tipo de conexión (por red o interna), igual que el primer parámetro de socket().
2. sin_port: es el número correspondiente al servicio.
3. 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.

```
struct sockaddr_in Direccion;
Direccion.sin_family = AF_INET;
Direccion.sin_port = htons(atoi(puerto));
Direccion.sin_addr.s_addr =INADDR_ANY;
if (bind (Descriptor, (struct sockaddr *)&Direccion, sizeof (Direccion)) == -1)
   printf ("Error\n");
```

La llamada a bind() lleva tres parámetros:

1. Descriptor del socket que hemos abierto.

2. Puntero a la estructura *Direccion* con los datos indicados anteriormente.
3. 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(). La función listen() admite dos parámetros:

1. Descriptor del socket.
2. Número máximo de clientes encolados.

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

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, devuelto por la función socket(), corresponde al servicio y sólo sirve para encolar a los clientes.

```
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 función con parámetros complejos. Los parámetros que admite son:

1. Descriptor del socket abierto.
2. 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.
3. 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 queremos que nuestro servidor esté continuamente escuchando peticiones de clientes debemos incluir la llamada a accept() en un bucle infinito. Posteriormente tenemos que leer los datos que nos envíe un cliente.

```
void Lee_Socket (int fd, char *Datos, int Longitud)
{
 if (recv(fd,Datos,Longitud,0)==-1)
    printf("\nSRV: Error de lectura en el socket");
}
```

Hemos creado una función con la cual leeremos los datos. En dicha función se nos pasa el descriptor asociado al socket, una cadena de caracteres donde guardaremos los datos que nos envía el cliente y la longitud máxima de caracteres a leer. La función recv nos devolverá el número total de caracteres leídos que podría ser menor que la longitud máxima (si la cadena es menor).

Si queremos que el servidor acepte **peticiones de clientes de forma concurrente** debemos crear un proceso hijo por cada petición (después de Accept). Este proceso se encargará de la comunicación con el cliente correspondiente.

```
if ((pid = fork()) == -1)
   printf("\nSRV: Error al crear el proceso hijo");

else
{
   if (pid == 0)
   {
      close(Socket_Servidor);
      while (resultado != -1)
      {

// Se lee la informacion del cliente, suponiendo que va a enviar 5 caracteres.
            Lee_Socket(Socket_Cliente, Cadena, 50);
```

```
// Se escribe en pantalla la informacion que se ha recibido del cliente
            resultado = realizarOperacion(Cadena);
//Se prepara una cadena de texto para enviar al cliente. La longitud de la
//cadena es 5 letras + \0 al final de la cadena = 6 caracteres
            resultado = sprintf(Cadena,"%d",resultado);
            Escribe_Socket (Socket_Cliente, Cadena, 50);
      }
      close(Socket_Cliente);
      exit(0);
   }
}
```

Una vez realizada la operación particular de cada servidor, lo único que queda es enviar el resultado al cliente.

```
void Escribe_Socket (int fd, char *Datos, int Longitud)
{
   int n;
   n = send(fd,Datos,Longitud,0);
}
```

Cuando el cliente indique la opción de no realizar más operaciones se enviará un mensaje de "fin" al servidor y cerrará la conexión con el cliente. El cliente realizará la misma función en su parte de la conexión.

```
//Se cierran los sockets en el proceso hijo del servidor
close(Socket_Cliente);
exit(0);
```

```
//Se cierran los sockets en el proceso hijo del servidor
Escribe_Socket (Socket_Con_Servidor, "fin", strlen("fin"));
Lee_Socket (Socket_Con_Servidor, Cadena, 50);
sscanf(Cadena,"%d",&resultado);
if (resultado == -1)
{
   close (Socket_Con_Servidor);
   printf("\nConexion cerrada.");
}
```

Para la lectura y la escritura, como hemos comentado anteriormente, se puede crear procesos hijos que se encarguen de atender a cada cliente que realizase una petición, de manera que el proceso padre no se quedase bloqueado resolviendo él mismo una petición de un cliente.

Implementación del cliente

Abrimos el socket igual que en el servidor.

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

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 = inet_addr(host);
Direccion.sin_port = htons(atoi(puerto));
if (connect (Descriptor, (struct sockaddr *)&Direccion,sizeof (Direccion)) ==
-1)
   printf ("Error\n");
```

El único cambio es que ahora debemos indicar la dirección del servidor al que nos queremos conectar. Posteriormente seguiremos las indicaciones realizadas por la interfaz del cliente. El cliente mantendrá la conexión abierta con el servidor mientras se deseen realizar más operaciones.

Compilación y ejecución

```
$ gcc -g -o servidor Servidor.c
$ gcc -g -o cliente Cliente.c
$ ./Servidor puerto &
$ ./Cliente dir_ip_servidor puerto
```

Para probarlo podemos lanzar el cliente en una máquina y el servidor en otra.