DPTO. DE INFORMÁTICA Y ANÁLISIS NUMÉRICO

UNIVERSIDAD DE CÓRDOBA

REDES – Práctica 2

Graduado en Ingeniería Informática

1. Fundamentos de la práctica 2

La programación de aplicaciones sobre TCP/IP se basa en el llamado modelo cliente-servidor. Básicamente, la idea consiste en que al indicar un intercambio de información, una de las partes debe "iniciar" el diálogo (cliente) mientras que la otra debe estar indefinidamente preparada a recibir peticiones de establecimiento de dicho diálogo (servidor). Cada vez que un usuario cliente desee entablar un diálogo, primero deberá contactar con el servidor, mandar una petición y posteriormente esperar la respuesta.

Los servidores pueden clasificarse atendiendo a si están diseñados para admitir múltiples conexiones simultáneas (servidores concurrentes), por oposición a los servidores que admiten una sola conexión por aplicación ejecutada (llamados iterativos). Evidentemente, el diseño de estos últimos será más sencillo que el de los primeros.

Otro concepto importante es la determinación del tipo de interacción entre el cliente y el servidor, que puede ser orientada a conexión o no orientada a conexión. Éstas corresponden, respectivamente, con los dos protocolos característicos de la capa de transporte: TCP y UDP. TCP (orientado a conexión) garantiza toda la "fiabilidad" requerida para que la transmisión esté libre de errores: verifica que todos los datos se reciben, automáticamente retransmite aquellos que no fueron recibidos, garantiza que no hay errores de transmisión y además, numera los datos para garantizar que se reciben en el mismo orden con el que fueron transmitidos. Igualmente, elimina los datos que por algún motivo aparecen repetidos, realiza un control de flujo para evitar que el emisor envíe más rápido de lo que el receptor puede consumir y, finalmente, informa a las aplicaciones (tanto cliente como al servidor) si los niveles inferiores de red no pueden entablar la conexión. Por el contrario, UDP (no orientada a conexión) no introduce ningún procedimiento que garantice la seguridad de los datos transmitidos, siendo en este caso responsabilidad de las aplicaciones la realización de los procedimientos necesarios para subsanar cualquier tipo de error.

En el desarrollo de aplicaciones sobre TCP/IP es imprescindible conocer como éstas pueden intercambiar información con los niveles inferiores; es decir, conocer la interfaz con los protocolos TCP o UDP. Esta interfaz es bastante análoga al procedimiento de entrada/salida ordinario en el sistema operativo UNIX que, como se sabe, está basado en la secuencia abrir-leer/escribir-cerrar. En particular, la interfaz es muy similar a los descriptores de fichero usados en las operaciones convencionales de entrada/salida en UNIX. Recuérdese que en las operaciones entrada/salida es necesario realizar la apertura del fichero (*open*) antes de que la aplicación pueda acceder a dicho fichero a través del ente abstracto "descriptor de fichero". En

la interacción de las aplicaciones con los protocolos TCP o UDP, es necesario que éstas obtengan antes el descriptor o "*socket*", y a partir de ese momento, dichas aplicaciones intercambiarán información con el nivel inferior a través del socket creado. Una vez creados, los sockets pueden ser usados por el servidor para esperar indefinidamente el establecimiento de una conexión (*sockets pasivos*) o, por el contrario, pueden ser usados por el cliente para iniciar la conexión (*sockets activos*).

1.1 Estructura y Funciones útiles en la Interfaz Socket

Para el desarrollo de aplicaciones, el sistema proporciona una serie de funciones y utilidades que permiten el manejo de los sockets. Puesto que muchas de las funciones y estructuras son iguales que las desarrolladas para los sockets UDP y éstas han sido explicadas en la práctica 1. En esta sección se detallaran solamente aquellas funciones nuevas.

1.1.1 Estructuras de la interfaz socket

Se parte de las estructuras sockaddr, sockaddr_in definidas en la práctica anterior, estudiaremos aquí otras estructuras interesantes.

Estructura hostent

La estructura hostent definida en el fichero /usr/include/netdb.h, contiene entre otras, la direccion IP del *host* en binario:

```
struct hostent {
        char *h_name;  /* nombre del host oficials */
        char  **h_aliases;  /* otros alias */
        int  h_addrtype;  /* tipo de dirección */
        int h_lenght;   /* longitud de la dirección en bytes */
        char **h_addr_list; /* lista de direcciones para el host */
}
#define h_addr h_addr_list[0]
```

Asociado con la estructura hostent está la función gethostbyname, que permite la conversión entre un nombre de host del tipo *www.uco.es* a su representación en binario en el campo *h_addr* de la estructura hostent.

Estructura servent

La estructura servent (también definida en el fichero netdb.h) contiene, entre otros, como campo el número del puerto con el que se desea comunicar:

```
struct servent {
        char *s_name;  /* nombre oficial del servicio */
        char  **s_aliases;  /* otros alias */
        int  s_port;   /* número del puerto para este servicio */
        char *s_proto;   /* protocolo a usar */
}
```

Con la estructura servent se relaciona la función getservbyname que permite a un cliente o servidor buscar el número oficial de puerto asociado a una aplicación estándar.

1.1.2 Funciones de la interfaz socket

TCP se caracteriza por tener un paso previo de establecimiento de la conexión, con lo que existen una serie de primitivas que no existían en el caso de UDP y que se estudiarán en este apartado.

Ambos, cliente y servidor, deben crear un socket mediante la función socket(), para poder comunicarse. El uso de esta función es igual que el descrito en la práctica 1 con la especificación de que se va a usar el protocolo SOCK_STREAM.

Otras funciones que no se han visto en la práctica1 y que se emplearán en TCP se detallan en este apartado.

Función listen()

Se llama desde el servidor, habilita el socket para que pueda recibir conexiones.

```
/* Se habilita el socket  para recibir conexiones */

int listen ( int sockfd, int backlog)
```

Esta función admite dos parámetros:

- *(1º argumento, sockfd),* es el descriptor del socket devuelto por la función socket() que será utilizado para recibir conexiones.
- *(2º argumento, backlog),* es el número máximo de conexiones en la cola de entrada de conexiones. Las conexiones entrantes quedan en estado de espera en esta cola hasta que se aceptan.

Función accept()

Se utiliza en el servidor, con un socket habilitado para recibir conexiones (listen()). Esta función retorna un nuevo descriptor de socket al recibir la conexión del cliente en el puerto configurado. La llamada a accept() no retornará hasta que se produce una conexión o es interrumpida por una señal.

```
/* Se queda a la espera hasta que lleguen conexiones */
int accept ( int sockfd, struct sockaddr *addr, socklen_t *addrlen)
```

Esta función admite tres parámetros:

- *(1º argumento, sockfd),* es el descriptor del socket habilitado para recibir conexiones.
- *(2º argumento, addr),* puntero a una estructura sockadd_in. Aquí se almacenará información de la conexión entrante. Se utiliza para determinar que host está llamando y desde qué número de puerto.
- *(3º argumento, addrlen),* debe ser establecido al tamaño de la estructura sockaddr. sizeof(struct sockaddr).

Función connect()

Inicia la conexión con el servidor remoto, lo utiliza el cliente para conectarse.

```
/* Iniciar conexión con un servidor */
int connect ( int sockfd, struct sockaddr *serv_addr, socklen_t addrlen )
```

Esta función admite tres parámetros:

- *(1º argumento, sockfd),* es el descriptor del socket devuelto por la función socket().
- *(2º argumento, serv_addr),* estructura sockaddr que contiene la dirección IP y número de puerto destino.
- *(3º argumento, serv_addrlen),* debe ser inicializado al tamaño de struct sockaddr. sizeof(struct sockaddr).

Funciones de Envío/Recepción

Después de establecer la conexión, se puede comenzar con la transferencia de datos. Podremos usar 4 funciones para realizar transferencia de datos.

```
/* Función de envío: send() */
send ( int sockfd,  void *msg, int len, int flags )
```

Esta función admite cuatro parámetros:

- *(1º argumento, sockfd),* es el descriptor socket por donde se enviarán los datos.
- *(2º argumento,* msg*),* es el puntero a los datos a ser enviados.
- *(3º argumento,* len*),* es la longitud de los datos en bytes.
- *(4º argumento, flags),* para ver las diferentes opciones consultar man send (la usaremos con el valor 0).

La función *send()* retorna la cantidad de datos enviados, la cual podrá ser menor que la cantidad de datos que se escribieron en el buffer para enviar.

```
/* Función de recepción: recv() */
recv ( int sockfd, void *buf, int len,  int flags )
```

Esta función admite cuatro parámetros:

- *(1º argumento, sockfd),* es el descriptor socket por donde se enviarán los datos.
- *(2º argumento, buf),* es el puntero a un buffer donde se almacenarán los datos recibidos.
- *(3º argumento, len),* es la longitud del buffer buf.

- *(4º argumento, flags),* para ver las diferentes opciones consultar man recv (la usaremos con el valor 0).

Si no hay datos a recibir en el socket , la llamada a recv() no retorna (bloquea) hasta que llegan datos. Recv() retorna el número de bytes recibidos.

```
/* Funciones de envío: write() */

write ( int sockfd, const void *msg, int len )
```

Esta función admite tres parámetros:

- *(1º argumento, sockfd),* es el descriptor socket por donde se enviarán los datos.
- *(2º argumento, msg),* es el puntero a los datos a ser enviados.
- *(3º argumento, len),* es la longitud de los datos en bytes.

```
/* Función de recepción: read() */

 read ( int sockfd, void *msg, in len )
```

Esta función admite los mismos parámetros que write, con la excepción de que el **buffer** de datos es donde se almacenará la información que nos envien.

Función close()

Finaliza la conexión del descriptor del socket. La función para cerrar el socket es close().

```
close (descriptor);
```

El argumento es el descriptor del socket que se desea liberar**.**

1.2 Ejemplo de Servidor Iterativo

El siguiente programa es un ejemplo muy sencillo de un servidor orientado a conexión, es decir usando TCP, que proporciona un servicio de envío de caracteres orientado hasta que se recibe la cadena de caracteres "FIN". El fichero está disponible en el moodle.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>


/*
 * El servidor ofrece el servicio de incrementar un número recibido de un
cliente
 */


main ( )
{

       /*-----------------------------------------------------
              Descriptor del socket y buffer de datos
       -----------------------------------------------------*/
       int sd, new_sd;
       struct sockaddr_in sockname, from;
       char buffer[100];
       socklen_t from_len;


       /* -----------------------------------------------------
              Se abre el socket
       -----------------------------------------------------*/
       sd = socket (AF_INET, SOCK_STREAM, 0);
       if (sd == -1)
       {
              perror("No se puede abrir el socket cliente\n");
              exit (1);
       }

       /*
 * El servidor ofrece el servicio de incrementar un número recibido de
   un cliente
       */

       sockname.sin_family = AF_INET;
       sockname.sin_port = htons(2000);
       sockname.sin_addr.s_addr =  INADDR_ANY;

       if (bind (sd, (struct sockaddr *)&sockname, sizeof (sockname)) == -1)
       {
              perror("Error en la operación bind");
              exit(1);
       }


       /*-----------------------------------------------------------------
              Del las peticiones que vamos a aceptar sólo necesitamos el
              tamaño de su estructura, el resto de información (familia,puerto,
              ip), nos la proporcionará el método que recibe las peticiones.
       -----------------------------------------------------------------*/
       from_len = sizeof (from);
```

```
	if(listen(sd,1) == -1){
		perror("Error en la operación de listen");
		exit(1);
	}

	/*-----------------------------------------------------------------
		El servidor acepta una petición
	-----------------------------------------------------------------*/

	if((new_sd = accept(sd, (struct sockaddr *) &from, &from_len)) == -1){
		perror("Error aceptando peticiones");
		exit(1);
	}
	do
	{
		if(recv(new_sd, buffer, 100, 0) == -1)
			perror("Error en la operación de recv");

printf("el mensaje recibido fue: \n%s\n", buffer );

}while(strcmp(buffer, "FIN")!=0);


	close(new_sd);
	close(sd);

}
```

1.3 Ejemplo de Cliente

El siguiente programa es un ejemplo de cliente muy sencillo, que solicita un servicio orientado a conexión, de envío de caracteres orientado a conexión. El fichero está disponible en el moodle.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>


main ( )
{

	/*-----------------------------------------------------
		Descriptor del socket y buffer de datos
	-----------------------------------------------------*/
	int sd;
	struct sockaddr_in sockname;
	char buffer[100];
	socklen_t len_sockname;


	/* ----------------------------------------------------
		Se abre el socket
	----------------------------------------------------*/
	sd = socket (AF_INET, SOCK_STREAM, 0);
	if (sd == -1)
	{
		perror("No se puede abrir el socket cliente\n");
		exit (1);
	}
```

```
        /* ------------------------------------------------------------------
               Se rellenan los campos de la estructura con la IP del
               servidor y el puerto del servicio que solicitamos
        ------------------------------------------------------------------*/
        sockname.sin_family = AF_INET;
        sockname.sin_port = htons(2000);
        sockname.sin_addr.s_addr =  inet_addr("127.0.0.1");

        /* ------------------------------------------------------------------
               Se solicita la conexión con el servidor
        ------------------------------------------------------------------*/
        len_sockname = sizeof(sockname);

        if (connect(sd, (struct sockaddr *)&sockname, len_sockname) == -1)
        {
               perror ("Error de conexión");
               exit(1);
        }

        /* ------------------------------------------------------------------
               Se transmite la información
        ------------------------------------------------------------------*/
        do
        {
                     puts("Teclee el mensaje a transmitir");
                     gets(buffer);

                     if(send(sd,buffer,100,0) == -1)
                            perror("Error enviando datos");

        }while(strcmp(buffer, "FIN") != 0);


        close(sd);
}
```

1.5 Terminología y Conceptos del Procesado Concurrente

Normalmente a un programa servidor se pueden conectar **varios clientes** simultáneamente. Hay dos opciones posibles para realizar esta tarea:

- Crear un nuevo proceso por cada cliente que llegue, estableciendo el proceso principal para estar pendiente de aceptar nuevos clientes.
- Establecer un mecanismo que nos avise si algún cliente quiere conectarse o si algún cliente ya conectado quiere algo de nuestro servidor. De esta manera, nuestro programa servidor podría estar "dormido", a la espera de que sucediera alguno de estos eventos.

La primera opción, la de múltiples procesos/hilos, es adecuada cuando las peticiones de los clientes son muy numerosas y nuestro servidor no es lo bastante rápido para atenderlas consecutivamente. Si, por ejemplo, los clientes nos hacen en promedio una petición por segundo y tardamos cinco segundos en atender cada petición, es mejor opción la de un proceso por cliente. Así, por lo menos, sólo sentirá el retraso el cliente que más pida.

La segunda es buena opción cuando recibimos peticiones de los clientes que podemos atender más rápidamente de lo que nos llegan. Si los clientes nos hacen una petición por segundo y tardamos un milisegundo en atenderla, nos bastará con un único proceso pendiente de todos. Esta opción será la que se implemente en la práctica, usando para ello la función *select()*.

Función select

```
/* Función de recepción:select() */

int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
```

- Los parámetros son:
  - *n: v*alor incrementado en una unidad del descriptor más alto de cualquiera de los tres conjuntos.
  - *readfds:* conjunto de sockets que será comprobado para ver si existen caracteres para leer. Si el socket es de tipo *SOCK_STREAM* y no esta conectado, también se modificará este conjunto si llega una petición de conexión.
  - *writefds:* conjunto de sockets que será comprobado para ver si se puede escribir en ellos.
  - *exceptfds:* conjunto de sockets que será comprobado para ver si ocurren excepciones.
  - *timeout: l*imite superior de tiempo antes de que la llamada a *select* termine. Si *timeout* es *NULL*, la función *select* no termina hasta que se produzca algún cambio en uno de los conjuntos (llamada bloqueante a *select*).

Para manejar el conjunto *fd_set* se proporcionan cuatro macros:

//Inicializa el conjunto *fd_set* especificado por *set*.
*FD_ZERO(fd_set *set);*

//Añaden o borran un descriptor de socket dado por *fd* al conjunto dado por *set*.
*FD_SET(int fd, fd_set *set);*
*FD_CLR(int fd, fd_set *set);*

//Mira si el descriptor de socket dado por *fd* se encuentra en el conjunto especificado por *set*.
*FD_ISSET(int fd, fd_set *est);*

Estructura timeval

```
/* Estructura timeval */

struct timeval
{
        unsigned long int tv_sec; /* Segundos */
        unsigned long int tv_usec; /* Millonesimas de segundo */
};
```

2. Enunciado de la práctica 2

El objetivo de esta práctica es que, a partir de un código de partida, se practique y se comprenda el funcionamiento básico de la programación con sockets, tanto para TCP como para UDP.

2.1 Estudio del código de partida

Comentar el código que se proporciona, de manera que quede constancia de que se entiende el funcionamiento, tanto en TCP como en UDP. Incluir los comentarios dentro del mismo fichero de código y utilizarlo para realizar posteriormente la segunda parte de la práctica y la tercera.

Conteste a las siguientes preguntas:

TCP

1. ¿Qué llamada sirve para crear el socket? ¿Cuántos parámetros necesita y qué significa cada uno de ellos? ¿Existe alguna diferencia entre las llamadas que efectúan el cliente y el servidor?
2. ¿Qué llamada hace el servidor para admitir la conexión de un nuevo cliente? ¿Qué devuelve esta llamada? ¿Qué tiene de particular y por qué es necesario este comportamiento?
3. Tal como está programado el servidor, ¿cuántos clientes concurrentes pueden estar conectados al mismo momento?
4. Cuando se ejecuten los programas, comprobar con netstat que se han creado los sockets correspondientes, y adjuntar una captura de pantalla.

UDP

5. ¿Cuál es la unidad de intercambio de datos entre el cliente y el servidor? ¿Cuántos datos puede transporter, a partir de lo que se observa en el código, cada paquete de datos?
6. Dado que en UDP no se establece conexión entre el cliente y el servidor, ¿dónde define el cliente la dirección y el Puerto del servidor al cuál se quieren enviar los datos?
7. ¿Qué llamadas usa el cliente para enviar los datos? ¿Qué llamadas usa el servidor para recibir los datos a través de los sockets UDP? ¿Cuántos parámetros tiene cada llamada?
8. ¿Cómo puede saber el servidor quién envía un paquete?
9. Ejecutar los programas, y comprobar con netstat que se han creado los sockets correspondientes. Adjuntar una captura.

2.2 Modificación del código TCP

Modificar el código de partida de tal forma que funcione como un servidor de reservas de butacas para un teatro. Para ello:

- El cliente recibirá como parámetros del programa el nombre de la máquina donde se encuentra el servidor (por ejemplo, localhost si se ejecutan ambos programas en la misma máquina). El servidor no recibirá ningún parámetro.
- El cliente se conectará al servidor y acto seguido le enviará un saludo formado por la cadena "HELLO".
- El servidor analizará la cadena recibida y mostrará por pantalla el mensaje "HELLO".
- El servidor devolverá el saludo, contestando también con la cadena "HELLO".
- El cliente imprimirá por pantalla la cadena recibida.
- A partir de aquí, el usuario introducirá por teclado un código de tipo de butaca, y el cliente lo enviará al servidor.
- El servidor añadirá detrás de la cadena recibida un espacio en blanco y el número de la butaca asignada, lo imprimirá por pantalla, y después lo devolverá al cliente.
- El cliente lo imprimirá por pantalla y volverá a repetir el mismo proceso (enviando un mensaje al servidor, esperando que le conteste el servidor) tantas veces como sesee.
- Los números iniciales de butacas disponibles para cada tipo estarán prefijados en el código tal y como se especifica en el apartado de Notas de este enunciado.
- El número de butacas disponibles para cada tipo nunca sera menor que 0.
- El número de butacas disponibles para todos los tipos se podrá consultar mediante el comando "DIS".
- Cuando el usuario quiera finalizar el programa, escribirá "FIN" por teclado. El cliente lo indicará al servidor, enviándole un mensaje de fin. El servidor le responderá con el mismo mensaje y el cliente finalizará la ejecución del programa después de mostrarlo por pantalla.
- El servidor, al recibir "FIN", cerrará la conexión y se pondrá de nuevo a escuchar peticiones de otros clientes.

- Han de contemplarse los casos en que tanto las lecturas como escrituras sobre los sockets devuelvan error, mostrando un mensaje indicative por pantalla y, a continuación, finalizando la ejecución del programa.
- El cliente no tiene que comprobar si las órdenes introducidas son o no correctas, o si existen. El servidor sí que debe comprobar la existencia de los comandos. En caso de que reciba cualquier orden desconocida, devolverá la cadena "ERROR".

Notas:

- Los tipos de butaca a tener en cuenta son (y por tanto las órdenes que recibirá el servidor):
  - PLA Platea
  - PI1 Piso 1
  - PI2 Piso 2
  - PI3 Piso 3
  - GAL Gallinero
- También existen las órdenes siguientes:
  - HELLO Saludo
  - DIS Consulta del número de butacas de un tipo disponibles
- El número inicial de butacas disponibles de cada tipo se fijará dentro del código del servido con los siguientes valores:
  - PLA 100
  - PI1 30
  - PI2 30
  - PI3 30
  - GAL 50
- El número de butacas disponibles de un tipo se decrementará en 1 cada vez que se reciba una petición de reserva de dicho tipo.
- El número de butacas disponibles de todos los tipos se puede consultar mediante la orden "DIS". El resultado será en el format: "0:100; 1:30; 2:30; 3:30; 4:50"

A continuación se muestra una ejecución de ejemplo. En azul tenemos lo que se muestra por pantalla, en verde lo que se introduce por teclado, y en rojo lo que se envían entre ellos.

CLIENTE	SERVIDOR
HELLO ->	HELLO <- HELLO
PLA PLA ->	PLA 100 <- PLA 100
PLA 100 PLA PLA ->	PLA 99 <- PLA 99
PLA 99 GAL GAL ->	GAL 50 <- GAL 50
GAL 50 DIS DIS ->	DIS 0:98; 1:30; 2:30; 3:30; 4:49 <- DIS 0:98; 1:30; 2:30; 3:30; 4:49
DIS 0:98; 1:30; 2:30; 3:30; 4:49 XXX XXX ->	XXX ERROR <- XXX ERROR
XXX ERROR FIN FIN ->	FIN <- FIN
FIN	

2.3 *Modificación del código UDP*

Repetir el apartado 2 pero modificando la versión UDP del código.

2.4 *Parte opcional (para optar a la nota máxima)*

Se debe modificar el código, tanto TCP como UDP para permitir que varios clientes accedan simultáneamente al servidor y reserven butacas.