DPTO. DE CIENCIA DE LA COMPUTACIÓN E
INTELIGENCIA ARTIFICIAL

UNIVERSIDAD DE CÓRDOBA

REDES – Práctica 1

Grado en Ingeniería Informática

Índice

1. FUNDAMENTOS DE LA PRÁCTICA 1

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 DE LA INTERFAZ SOCKET

Para el desarrollo de aplicaciones, el sistema proporciona una serie de funciones y utilidades que permiten el manejo de los sockets.

Hay que incluir siempre los siguientes ficheros de cabecera:

```
1 # include<stdio.h>
2 # include<sys/types.h>
3 # include<sys/sockets.h>
4 # include<netinet/in.h>
5 # include<netdb.h>
```

Repasemos las estructuras de datos importantes. Comenzamos con el problema de los direccionamientos. Los programas de aplicación usan una estructura predefinida cuando necesitan declarar variables que contengan direcciones finales. La estructura más general para tal fin es la conocida por `sockaddr`, mediante la cual el kernel almacena la mayoría de las direcciones.

```
1 struct sockaddr {
2   u_short sa_family; /* familia de direcciones */
3   char sa_data[14];  /* hasta 14 bytes de dirección */
4 };
```

El campo de 2 bytes *sa_family* contiene un identificador de la familia de direcciones, que para el caso de usar TCP/IP debe contener la constante simbólica *AF_INET* (ver el fichero de cabecera *socket.h* para otras familias de direcciones).

No obstante, para mantener los programas portables se recomienda no utilizar la es-

tructura `sockaddr` en las declaraciones. Otra posibilidad que se recomienda es usar la estructura `sockaddr_in` (definida en el fichero de cabecera `#include linux/in.h` incluido en el fichero `/usr/incluye/netinet/in.h`) que permite almacenar la información relacionada con un socket en el siguiente formato:

```
struct sockaddr_in{
    short sin_family;          /* tipo de dirección */
    u_short sin_port;          /* número del puerto */
    struct in_addr sin_addr    /* dirección IP */
    char sin_zero[8];          /* no se usa */
}
```

El campo `sin_family` especifica la familia de protocolos a usar, en nuestro caso, como se trata de TCP/IP, dicho campo debe ser igual a la constante *AF_INET*. Igualmente, en el campo sin_addr (que es a su vez otra estructura) debe utilizarse sólo el subcampo `u_long s_addr`, que contendrá los 4 bytes de la dirección IP (esto que puede parecer innecesariamente complicado, es así para mantener compatibilidad con versiones antiguas del 4.2BSD). Para aclaración supongamos que definimos la variable `sockname` tipo estructura `sockaddr_in` dentro del main:

```
struct sockaddr_in sockname;
```

Para asignarle un valor deberemos rellenar los siguientes campos:

```
sockname.sin_family = familia_de_direcciones;
sockname.sin_addr.s_addr = direccion_IP;
sockname.sin_port = numero_de_puerto;
```

Puede ser útil usar la función `inet_addr`, que realiza una conversión de las cadenas de caracteres de la dirección IP en el formato convencional al formato adecuado para ser utilizado en la estructura `sockaddr_in`.

Otra función de utilidad es `htons`, que cambia el orden de los bytes del entero que se le pasa como argumento. La conversión se realiza del llamado “host byte order”, en el que el byte LSB precede al MSB, al llamado “network byte order” (usado en internet), donde el byte MSB precede al LSB.

1.2. EJEMPLO DE APLICACIÓN

En los sockets orientados a conexión se envían mensajes con `write()` o `send()` y se reciben con `read()` o `recv()`. En un socket no orientado a conexión hay que indicar el destinatario, así que se usan las funciones `sendto()` y `recvfrom()`.

1.2.1. DESCRIPCIÓN DE LOS PASOS EN EL SERVIDOR

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

- Abrir un socket con la función `socket()`
- Asociar el socket a un puerto con `bind()`
- Leer mensaje con `recvfrom()`
- Responder mensaje con `sendto()`
- Cerrar el socket con `socket()`

1.2.1.1 Abrir el socket

Se abre el socket con la función `socket()`. Esta función nos devuelve un descriptor de socket. La forma de llamarla sería la siguiente:

```
1 int Descriptor;
2 Descriptor = socket (AF_INET, SOCK_DGRAM, 0);
```

- El primer parámetro indica la familia, en nuestro caso será *AF_INET*.
- El segundo indica que es *UDP* (*SOCK_STREAM* indicaría un socket *TCP* orientado a conexión).
- El tercero es el protocolo que queremos utilizar. Hay varios disponibles, pero poniendo un *0* dejamos al sistema que elija este detalle.

1.2.1.2 Asociar el socket con un puerto

En unix para el establecimiento de conexiones con sockets hay 65536 puertos disponibles, del 0 al 65535. Del 0 al 1023 están reservados para el sistema. El resto están disponibles.

Para decir al sistema operativo que deseamos atender a un determinado servicio, de forma que cuando llegue un mensaje por ese servicio nos avise, debemos llamar a la función `bind()`. La forma de llamarla es la siguiente:

```
struct sockaddr_in Direccion;

/* Hay que rellenar la estructura Direccion */
Direccion = ... (la veremos a continuacion)
bind (Descriptor, (struct sockaddr *)&Direccion, sizeof (
   Direccion));
```

- El primer parámetro es el descriptor de socket obtenido con la función `socket()`.
- El segundo parámetro es un puntero a una estructura `sockaddr` que debemos rellenar adecuadamente (se explicará en la siguiente sección). La Dirección la hemos declarado como `struct sockaddr_in` porque como se ha explicado, es una estructura más adecuada y es compatible con la estructura `sockaddr` (podemos hacer cast de una a otra). Para los sockets que comunican procesos en la misma máquina `AF_UNIX`, tenemos la estructura `sockaddr_un`, que también es compatible con sockaddr).
- El tercer parámetro es el tamaño de la estructura `sockaddr_in`.

1.2.1.3 Rellenar la estructura `sockaddr_in`

Hay tres campos que debemos rellenar:

```
Direccion.sin_family = AF_INET;
Direccion.sin_port =htons(numero_puerto) ;
Direccion.sin_addr.s_addr = INADDR_ANY; (Direccion));
```

- El campo `sin_family` se rellena con el tipo de socket que estamos tratando, `AF_INET` en nuestro caso.

- El campo `sin_port` es el número de puerto en el que el servidor escuchará conexiones.
- El campo `s_addr` es la dirección IP del cliente al que queremos atender. Poniendo `INADDR_ANY` atenderemos a cualquier cliente.

1.2.1.4 Envío y Recepción de mensajes

Con los pasos anteriores, el socket del servidor está dispuesto para recibir y enviar mensajes.

1.2.1.4.1 Recibir mensajes en el servidor

La función para leer un mensaje por un socket *UDP* es `recvfrom()`. Esta función admite seis parámetros. Vamos a verlos:

- *(1º argumento, int)*, es el descriptor del socket que queremos leer. Lo obtuvimos con `socket()`.
- *(2º argumento, char *)*, es el buffer donde queremos que nos devuelva el mensaje. Podemos pasar cualquier estructura o array que tenga el tamaño suficiente en bytes para contener el mensaje. Debemos pasar un puntero y hacer el cast a `char *`.
- *(3º argumento, int)*, es el número de bytes que queremos leer y que compondrán el mensaje. El buffer pasado en el campo anterior debe tener al menos tantos bytes como indiquemos aquí.
- *(4º argumento, int)*, con opciones de recepción. De momento nos vale un *0*.
- *(5º argumento, struct sockaddr)*, esta estructura se pasa vacía y `recvfrom()` nos devolverá en ella los datos del que nos ha enviado el mensaje. Si los guardamos, luego podremos responderle con otro mensaje. Si no queremos responder, en este parámetro podemos pasar *NULL* (sabiendo que de este modo, no tenemos forma de saber quién nos ha enviado el mensaje ni de responderle).
- *(6º argumento, int *)*, debemos especificar con este parámetro el tamaño de la estructura `sockaddr_in`. La función nos lo devolverá con el tamaño de los datos contenidos en dicha estructura.

El código de nuestro ejemplo, para recibir un mensaje, se muestra a continuación:

```
/* Contendrá los datos del que nos envía el mensaje */
struct sockaddr_in Cliente;

/* Tamaño de la estructura anterior */
int longitudCliente = sizeof(Cliente);

/* El mensaje es simplemente un entero, 4 bytes. */
int buffer;

recvfrom (Descriptor, (char *)&buffer, sizeof(buffer), 0, (struct
    sockaddr *)&Cliente, &longitudCliente);
```

La función se quedará bloqueada hasta que llegue un mensaje. Esta función nos devolverá el número de bytes leídos o -1 si ha habido algún error.

1.2.1.4.2 Responder con un mensaje al cliente

La función para envío de mensajes es `sendto()`. Esta función admite seis parámetros, que son los mismos que la función `recvfrom()`. Su significado cambia un poco, así que vamos a verlos:

- *(1º argumento, int)*, con el descriptor del socket por el que queremos enviar el mensaje. Lo obtuvimos con `socket()`.
- *(2º argumento, char *)*, con el buffer de datos que queremos enviar. En este caso, al llamar a `sendto()` ya debe estar relleno con los datos a enviar.
- *(3º argumento, int)*, con el tamaño del mensaje anterior, en bytes.
- *(4º argumento, int)*, existen diferentes opciones. De momento nos vale poner un *0*.
- *(5º argumento, struct sockaddr)*, en este caso, como es respuesta a la petición del cliente, podemos especificar la misma estructura que nos rellenó la función `recvfrom()`, estaremos enviando el mensaje al cliente que nos lo envío a nosotros previamente.
- *(6º argumento, int)*, con el tamaño de la estructura `sockaddr`. Vale el mismo entero que nos devolvió la función `recvfrom()` como sexto parámetro.

El código, después de haber recibido el mensaje del cliente, quedaría como se especifica a continuación:

```
/* Rellenamos el mensaje que se va a mandar con los datos que
   queramos */
buffer = ...;

sendto (Descriptor, (char *)&buffer, sizeof(buffer), 0, (struct
   sockaddr *)&Cliente, longitudCliente);
```

La llamada envía el mensaje y devuelve el número de bytes escritos o -1 en caso de error.

1.2.1.5 Cerrar el socket

La función para cerrar el socket es `close()`. La forma de llamarla sería la siguiente:

```
close (descriptor);
```

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

1.2.2. DESCRIPCIÓN DE LOS PASOS EN EL CLIENTE

Los pasos que debe seguir un cliente son los siguientes:

- Abrir un socket con `socket()`
- Enviar mensaje al servidor con `sendto()`
- Leer respuesta con `recvfrom()`
- Cerrar el socket con `socket()`

1.2.2.1 Abrir el socket

Se trata de la misma especificación que en el servidor.

1.2.2.2 Envío y Recepción de mensajes

1.2.2.2.1 Enviar un mensaje al servidor

Para enviar un mensaje al servidor la función es `sendto()`. Los parámetros y forma de funcionamiento es igual que en el servidor. Para llamar a esta función tenemos que rellenar la estructura `sockaddr` del quinto parámetro, indicando los datos del servidor.

```
Direccion.sin_family = AF_INET;
Direccion.sin_port = ...; (El puerto del servicio)
Direccion.sin_addr.s_addr = ...; (La IP del servidor)
```

1.2.2.2.2 Recibir un mensaje del servidor

Es igual que en el servidor para recibir mensajes, con la función recvfrom(), los argumentos serían los mismos.

1.2.2.3 Cerrar el socket

La función para cerrar el socket es `close()`. La forma de llamarla sería la siguiente:

```
close (descriptor);
```

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

1.3. ATENDER VARIAS PETICIONES DE CLIENTES. FUNCIÓN SELECT

Para que en un programa servidor se pueden conectar **varios clientes** simultáneamente, podemos realizar dos opciones posibles.

- Una opción es crear un nuevo proceso o hilo por cada cliente que llegue, más el proceso o hilo principal, pendiente de aceptar nuevos clientes.

- La otra opción sería recibir un aviso 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. Esta será la opción con la que trabajaremos en clase.

La función `select()` nos permite simularemos la atención de varios clientes sin crear procesos, aunque realmente no se está realizando de manera concurrente. 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. Esta opción es adecuada 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.

```
1 /* Descripción función select */
2
3 int select(int n, fd_set *readfds, fd_set *writefds, fd_set *
    exceptfds, struct timeval *timeout);
```

Esta función admite cinco parámetros:

1. ***n*:** valor incrementado en una unidad, del descriptor más alto de cualquiera de los tres conjuntos.
2. ***readfds*:** conjunto de sockets que serán comprobados para ver si existen caracteres para leer. Si el socket es de tipo *SOCK_STREAM* y no está conectado, también se modificará este conjunto si llega una petición de conexión.
3. ***writefds*:** conjunto de sockets que serán comprobados para ver si se puede escribir en ellos.
4. ***exceptfds*:** conjunto de sockets que serán comprobados para ver si ocurren excepciones.
5. ***timeout*:** límite 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);
```

La estructura de tipo *timeval* que se emplea está definida:

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

timeout.tv_sec=1;
timeout.tv_usec=0;
```

2. ENUNCIADO DE LA PRÁCTICA 1

La práctica 1 consiste en crear un protocolo de aplicación similar al servicio proporcionado por DAYTIME. Este servicio proporciona información sobre el día y la hora del ordenador remoto. El servicio puede ofrecerse tanto a través de TCP como UDP, en esta práctica nos basaremos en el protocolo UDP.

2.1. Descripción

Se pide diseñar un servidor UDP que implemente un servicio similar al proporcionado por daytime, así como un cliente que haga uso de su servicio. Como el servicio que ofrece UDP es no fiable tanto la petición como la respuesta pueden perderse, por lo que el cliente puede no recibir contestación. El programa cliente realizará una petición al servidor y esperará la respuesta un tiempo limitado (5 segundos). Si recibe la respuesta, enviará a la salida estándar el mensaje proporcionado por el servidor. Si después de ese tiempo no recibe una respuesta, volverá a mandar el mensaje al servidor para intentar recibir, si después de tres intentos no consigue nada, mostrará al usuario un mensaje de error. Tanto la dirección IP como el timeout se pasarán como argumentos. El servicio admitirá la siguiente especificación de los paquetes:

- **DAY**: paquete para solicitar el día de la maquina remota. La salida será con el siguiente formato: *Lunes, 17 de Septiembre de 2018*
- **TIME**: paquete para solicitar la hora de la maquina remota. La salida será con el siguiente formato: *16:00:00*
- **DAYTIME**: paquete para solicitar tanto el día como la hora de la maquina remota. La salida será con el siguiente formato: *Lunes, 17 de Septiembre de 2018; 16:00:00*

El funcionamiento bajo UDP es el siguiente:

1. El cliente envía un datagrama UDP con el tipo de servicio que desea al servidor.
2. El servidor recibe el datagrama UDP, a partir del cual obtiene la dirección IP y el puerto UDP del cliente que solicita la información.
3. El servidor realiza una consulta para obtener la información solicitada por el cliente. Se puede hacer uso de las funciones *strftime* y *asctime* para obtener la información

necesaria para ofrecer este servicio.

4. El servidor crea un datagrama UDP con la información obtenida en el punto 2, datagrama que tiene como único dato la cadena de texto a devolver.

5. El datagrama UDP llega al cliente, el cual puede leerlo y extraer la información del día y hora del servidor.

2.2. Objetivo

- Conocer las funciones básicas para trabajar con sockets y el procedimiento a seguir para conectar dos procesos mediante un socket bajo UDP.
- Comprender el funcionamiento de un servicio no orientado a la conexión confiable y no confiable a través del envío de paquetes entre una aplicación cliente y otra servidora utilizando sockets.
- Comprender como se añade confiabilidad al servicio proporcionado.
- Comprender las características o aspectos claves de un protocolo:
  - **Sintaxis**: formato de los paquetes.
  - **Semántica**: definiciones de cada uno de los tipos de paquetes.