DPTO. DE INFORMÁTICA Y ANÁLISIS NUMÉRICO
UNIVERSIDAD DE CÓRDOBA

REDES – Práctica 2

Graduado en Ingeniería Informática

PRÁCTICA 2

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 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

Diseño e implementación del juego de la ruleta de la suerte permitiendo jugar de manera individual o bien en grupos. La finalidad del juego es averiguar la frase o palabra secreta que se muestra en cada partida.

2.1 Objetivos del juego de la ruleta

Para jugar a la ruleta de la suerte es necesario un tablón donde aparecen marcadas las letras de las que consta la frase a adivinar. El objetivo del juego es adivinar la frase consiguiendo el mayor número de puntos. Se jugará compitiendo con otros usuarios. Gana el jugador que consigue resolver satisfactoriamente la frase.

Todas las partidas versarán sobre la categoría de refranes.

2.2 Especificación del juego a implementar

La comunicación entre los clientes del juego de la ruleta de la suerte se realizará bajo el protocolo de transporte TCP. La práctica que se propone consiste en la realización de una aplicación cliente/servidor que implemente el **juego de la ruleta de la suerte** con algunas restricciones. En el juego considerado los jugadores (los clientes) se conectan al servicio (el servidor). Solamente se admitirán partidas con dos jugadores. Se admiten hasta 10 partidas simultáneas, y hasta 30 jugadores conectados simultáneamente en el servidor.

El procedimiento que se seguirá será el siguiente:

- Un cliente se conecta al servicio y si la conexión ha sido correcta el sistema devuelve "*+0k. Usuario conectado"*.
- Para poder acceder a los servicios es necesario identificarse mediante el envío del usuario y clave para que el sistema lo valide[1]. Los mensajes que deben indicarse son: "*USUARIO usuario*" para indicar el usuario, tras el cual el servidor enviará "*+Ok. Usuario correcto*" o "*–Err. Usuario incorrecto*". En caso de ser correcto el siguiente mensaje que se espera recibir de dicho usuario es "*PASSWORD password*", donde el servidor responderá con el mensaje de "*+Ok. Usuario validado*" o "*–ERR. Error en la validación*".

[1] El control de usuarios y claves en el servidor se llevará simplemente mediante un fichero de texto plano y no se codificará ningún tipo de encriptación.

- Un usuario nuevo podrá registrarse mediante el mensaje “REGISTRO –u usuario –p password”. Se llevará un control para evitar colisiones con los nombres de usuarios ya existentes.
- Una vez conectado y validado, el cliente podrá llevar a cabo una partida en el juego indicando un mensaje de **“**INICIAR-PARTIDA**”**. Recibido este mensaje en el servidor, éste se encargará de comprobar las personas que tiene pendiente para comenzar una partida:
    - Si con esta petición, ya se forma un grupo de dos jugadores, mandará un mensaje a cada uno de ellos, para indicarle que la partida va a comenzar “+Ok. Empieza la partida. FRASE: _ _ _ _ _ _ _ _ _ _ _ _ _ _”. El orden de los usuarios en el juego será el mismo orden en el que se ha realizado la conexión.
    - Si todavía falta un jugador para iniciar la partida, mandará un mensaje al nuevo usuario, especificando que tiene su petición y que está a la espera de la conexión de otro jugador “+*Ok. Petición Recibida. Quedamos a la espera de más jugadores*”.
- Una vez comenzada la partida, en el envío de la frase a adivinar, se envía la frase que se debe adivinar con la que inicia la partida. Se indicarán todas las posiciones que representan una consonante o vocal mediante el símbolo “-“ y el espacio en blanco “ “, se utilizará para representar la separación entre las palabras de la frase. No se tendrán en cuenta en la frase los signos de puntuación como elemento a adivinar por el jugador, estos elementos aparecerán directamente en la frase.
- El servidor será el que indique el turno al jugador oportuno, con el mensaje “+Ok. Turno de partida”, al otro jugador le indicará que es el turno del compañero “+Ok. Turno del otro jugador”.
- En su turno, cada usuario podrá ir mandando mensajes de CONSONANTE/VOCAL *letra,* mientras la frase contenga la letra que va especificando. Esta información se va mostrando a cada uno de los usuarios, al igual que los mensajes que envía el usuario que está jugando en cada momento. Cuando falle, se pasará a otro usuario, así hasta que un usuario escoja la opción de RESOLVER la frase. En este punto son admitidos cuatro tipos de mensajes que se puede enviar al servidor:
    - CONSONANTE *letra,* donde letra indica una consonante que se piensa que puede estar en la frase. La puntuación que se obtiene sería 50 puntos por cada vez que la consonante aparezca en la frase a resolver.

Si la frase desconocida no contiene ninguna consonante de la que has elegido, se pasará el turno al siguiente jugador.

- Se recibirá por cada jugador de La partida:
- "+Ok. <consonante> aparece <n> veces. FRASE: <frase>.

- VOCAL *letra*, para poder mandar este mensaje necesitas tener al menos 50 puntos, que se restarán por cada vocal que solicites, con independencia del número de veces que aparezca. Si no se tiene suficientes puntos, se recibirá: "+Ok. No tienes puntuación suficiente". Si la frase desconocida no contiene ninguna vocal de la que has elegido, se pasará el turno al siguiente jugador, aunque se restarán los 50 puntos.
  - Si se tiene puntuación suficiente, se recibirá por cada jugador de La partida:
    - "+Ok. <vocal> aparece <n> veces. FRASE: <frase>"
- PUNTUACION, se le mandará al jugador la puntuación que tiene en ese momento.
- RESOLVER *frase*, donde *frase* representará una cadena que contiene el refrán que queremos resolver.

- Se debe tener cuidado con el servidor cuando lleguen peticiones de usuarios, cuando no sea su turno de jugar. En caso de que envíen un mensaje cuando no es su turno, deberá responder que todavía no es turno y deben esperar. La especificación del mensaje que se enviaría sería: "*-Err. Debe esperar su turno*".
- La partida continúa con los jugadores colocando sus fichas hasta que se presenta alguna de las situaciones siguientes:
  - En caso de que un usuario envíe la petición RESOLVER <frase>.
    - Si acierta el refrán, ha ganado el jugador que ha mandado RESOLVER, se mandará un mensaje a los jugadores: "*+Ok. Partida finalizada. FRASE: <refrán>. Ha ganado el jugador <nombre> con <puntos> puntos.*".
    - Si falla el refrán, ha perdido el jugador que ha mandado RESOLVER, se mandará un mensaje a los jugadores: "*+Ok. Partida finalizada. FRASE: <refrán>. No se ha acertado la frase.*".
  - Cuando ya se ha completado todos los huecos del refrán, sin haber indicado "RESOLVER":
    - Ha ganado el jugador que ha colocado la última consonante o vocal necesaria para cumplimentar el panel. Se mandará un mensaje a los jugadores: "*+Ok. Partida finalizada. FRASE:*

<refrán>. Ha ganado el jugado <nombre> con <puntos> puntos.".

- Un jugador siempre podrá salir de la aplicación en cualquier momento. De este modo, el comando "SALIR" al servidor implicará:
    - Si estaba jugando, finalizará la partida en la que se encontraba, avisando al otro jugador. El otro jugador será informado: "+Ok. Ha salido el otro jugador. Finaliza la partida." y seguirá en el sistema pudiendo iniciar una nueva partida.
    - Si estaba a la espera, lo elimina de la espera.
    - Si no había indicado que quería seleccionar una partida, solamente sale del sistema.
    - En todos los casos, mandará "+Ok. Desconexión procesada.".

- Cualquier mensaje que no use uno de los especificadores detallados, generará un mensaje de "-*Err*" por parte del servidor.

Algunas restricciones a tener en cuenta en esta nueva versión que se debe implementar es:

- La comunicación será mediante consola.
- El cliente deberá aceptar como argumento una dirección IP que será la dirección del servidor al que se conectará.
- El protocolo deberá permitir mandar mensajes de tamaño arbitrario. Teniendo como tamaño máximo envío una cadena de longitud 350 caracteres. No se permitirá jugar con frases de un número mayor de 250 caracteres.
- El servidor aceptará servicios en el puerto 2050.
- El servidor debe permitir la conexión de varios clientes simultáneamente. Se utilizará la función *select()* para permitir esta posibilidad.
- El número máximo de clientes conectados será de 30 usuarios.
- Todos los mensajes devueltos por el servidor que se necesiten contemplar, seguirán el criterio de ir precedidos por *+Ok. Texto informativo*, si la petición se ha podido aceptar sin problemas y *–Err. Texto informativo*, si ha habido algún tipo de error.

- Las frases que se van a resolver no tendrán en cuenta los signos de puntuación. Si la frase cuenta con alguno de ellos, aparecerá directamente cuando se indique la vocal acentuada.

2.2 Resumen paquetes

Considerando la práctica completa, vamos a considerar los siguientes tipos de mensajes con el siguiente formato cada uno:

- **USUARIO** *usuario***:** mensaje para introducir el usuario que desea.
- **PASSWORD** *contraseña***:** mensaje para introducir la contraseña asociada al usuario.
- **REGISTER** *–u usuario –p password*: mensaje mediante el cual el usuario solicita registrarse para acceder al juego de la ruleta que escucha en el puerto TCP 2050.
- **INICIAR_PARTIDA:** mensaje para solicitar jugar una partida de la ruleta de la suerte en grupo.
- **CONSONANTE** *letra,* donde letra indica una consonante que se piensa que puede estar en la frase.
- **VOCAL** *letra*, para poder mandar este mensaje necesitas tener al menos 50 puntos, que se restarán por cada vocal que solicites, con independencia del número de veces que aparezca.
- **RESOLVER** *frase*, donde *frase* representará una cadena que contiene el refrán que queremos resolver.
- **PUNTUACION**, solicitar la puntuación en el juego.
- **SALIR:** mensaje para solicitar salir del juego.
- Cualquier otra tipo de mensaje que se envíe al servidor, no será reconocida por el protocolo como un mensaje válido y generará su correspondiente "*-Err*." por parte del servidor.

2.3 Objetivo

- Conocer las funciones básicas para trabajar con sockets y el procedimiento a seguir para conectar dos procesos mediante un socket.
- Comprender el funcionamiento de un servicio orientado a conexión y confiable del envío de paquetes entre una aplicación cliente y otra servidora utilizando sockets.
- Comprender las características o aspectos claves de un protocolo:
    - **Sintaxis:** formato de los paquetes
    - **Semántica:** definiciones de cada uno de los tipos de paquetes