



DPTO. DE CIENCIA DE LA COMPUTACIÓN E
INTELIGENCIA ARTIFICIAL



UNIVERSIDAD DE CÓRDOBA

REDES – Práctica 2

Grado en Ingeniería Informática

Índice

1. FUNDAMENTOS DE LA PRÁCTICA 2	2
1.1. Estructura y Funciones útiles en la Interfaz Socket	3
1.1.1. Estructuras de la interfaz socket	3
1.1.2. Funciones de la interfaz socket	4
1.2. Terminología y Conceptos del Procesado Concurrente	7
2. ENUNCIADO DE LA PRÁCTICA 2	10
2.1. Objetivos del juego de los números	10
2.2. Reglas del juego de los números con restricciones	10
2.3. Especificación del juego a implementar	10
2.4. Resumen paquetes	13
2.5. Objetivo	14

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. Recuérdesse 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 detallarán 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 dirección IP del `host` en binario:

```
1 struct hostent {
2     char *h_name;           /* nombre del host oficiales */
3     char **h_aliases;       /* otros alias */
4     int  h_addrtype;        /* tipo de dirección */
5     int  h_length;          /* longitud de la dirección en bytes */
6     char **h_addr_list;     /* lista de direcciones para el host */
7 }
8 #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:

```
1 struct servent {  
2     char *s_name;           /* nombre oficial del servicio */  
3     char **s_aliases;       /* otros alias */  
4     int s_port;             /* número del puerto para este servicio */  
5     char *s_proto;          /* protocolo a usar */  
6 }
```

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 práctica 1 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.

```
1 /* Se habilita el socket para recibir conexiones */  
2  
3 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.

```
1  /* Se queda a la espera hasta que lleguen conexiones */
2
3  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 `sockaddr_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.

```
1  /* Iniciar conexión con un servidor */
2
3  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.

```
1  /* Función de envío: send() */
2
3  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.

```
1  /* Función de recepción: recv() */
2
3  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.

```
1  /* Funciones de envío: write() */
2
3  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.

```
1  /* Función de recepción: read() */
2
3  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 envíen.

Función `close()`

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

```
1  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()`

```
1  /* Función de recepción:select() */
2
3  int select(int n, fd_set *readfds, fd_set *writefds, fd_set *
    exceptfds, struct timeval *timeout);
```

Los parámetros son:

- ***n***: valor incrementado en una unidad del descriptor más alto de cualquiera de los tres conjuntos.
- ***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.
- ***writefds***: conjunto de sockets que serán comprobados para ver si se puede escribir en ellos.
- ***exceptfds***: conjunto de sockets que serán comprobados para ver si ocurren excepciones.

- ***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:

```
1 //Inicializa el conjunto fd_set especificado por set.
2 FD_ZERO(fd_set *set);
3
4 //Añaden o borran un descriptor de socket dado por fd al conjunto
   dado por set.
5 FD_SET(int fd, fd_set *set);
6 FD_CLR(int fd, fd_set *set);
7
8 //Mira si el descriptor de socket dado por fd se encuentra en el
   conjunto especificado por set.
9 FD_ISSET(int fd, fd_set *set);
```

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

```
1 /* Estructura timeval */
2
3 struct timeval
4 {
5     unsigned long int tv_sec; /* Segundos */
6     unsigned long int tv_usec; /* Millonesimas de segundo */
7 };
```

2. ENUNCIADO DE LA PRÁCTICA 2

Diseño e implementación del juego de los números permitiendo jugar en red. El juego consiste en que los jugadores intenten alcanzar un número objetivo, sin pasarse.

2.1. Objetivos del juego de los números

El objetivo del juego es alcanzar un número objetivo predefinido (por ejemplo, 100), sin sobrepasarlo. Es decir, gana la partida el jugador que se acerque más al número establecido, pero si lo sobrepasa, “pierde” automáticamente.

2.2. Reglas del juego de los números con restricciones

Se implementará una versión limitada del juego, siguiendo las reglas y restricciones que se indican en esta sección. El juego se limitará a dos jugadores donde ambos jugadores seguirán las mismas reglas.

Se genera un número objetivo aleatorio entre 60 y 200. Después de esa asignación, se iniciará el juego. Cada jugador podrá realizar las siguientes opciones en su turno:

- **Tirar dados:** un jugador dispone de dos dados, pero puede decidir si tirar solo uno o los dos, por lo tanto, podrá sumar como máximo 12 a su puntuación actual.
- **No tirar dados:** un jugador puede decidir no tirar dados hasta tres veces en una partida.
- **Plantarse:** un jugador se puede plantar en cualquier momento. Una vez plantado ya no podrá volver a tirar los dados.

2.3. Especificación del juego a implementar

La comunicación entre los clientes del juego 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 con las restricciones comentadas anteriormente. En el juego

considerado los jugadores (los clientes) se conectan al servicio (el servidor). Solamente se admitirán partidas con dos jugadores y se admitirá un máximo de 10 partidas simultáneas.

El protocolo que se seguirá será el siguiente:

- Un cliente se conecta al servicio y si la conexión ha sido correcta el sistema devuelve “+Ok. Usuario conectado”.
- Para poder acceder a los servicios es necesario identificarse mediante el envío del usuario y clave para que el sistema lo valide. 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 es “PASSWORD password”, donde el servidor responderá con el mensaje de “+Ok. Usuario validado” o “-Err. Error en la validación”.
- 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 una pareja, mandará un mensaje a ambos, para indicarle que la partida va a comenzar generando un número objetivo aleatorio. “+Ok. Empieza la partida. NÚMERO OBJETIVO: [<NÚMERO>]”
 - Si no había nadie esperando, mandará un mensaje al nuevo usuario, especificando que tiene su petición y que está a la espera de la conexión de otro jugador. El usuario en este caso deberá quedarse a la espera de recibir un mensaje para jugar o también se le permitirá salir de la aplicación. “+Ok. Esperando otro jugador”.
- Una vez ha comenzado una partida, cualquiera de los jugadores podrá empezar a jugar indicando cualquiera de los comandos específicos: TIRAR-DADOS, NO-TIRAR-DADOS, PLANTARME.
- TIRAR-DADOS: El usuario podrá tirar uno o dos dados mientras que no supere el número objetivo establecido al inicio de la partida, una vez alcanzado este valor, su única opción será enviar el comando PLANTARME. Ante cada mensaje de TIRAR-DADOS, el servidor enviará al jugador el mensaje:

- Si ha tirado un dado: “+Ok.[<DADO 1>,<VALOR>]” donde valor será un número en el intervalo [1-6].
 - Si ha tirado dos dados: “+Ok.[<DADO 1>,<VALOR>; <DADO 2>,<VALOR>]” donde valor será un número en el intervalo [1-6].
 - Debe visualizarse la puntuación total junto con los dados obtenidos por el oponente en la tirada previa.
 - “-Err. Excedido el valor de <NÚMERO OBJETIVO>”, si intenta tirar más dados una vez ya superado ese número, el servidor le devolverá error.
- NO-TIRAR-DADOS: El usuario puede decidir pasar el turno y no tirar dados hasta tres veces en una partida.
 - PLANTARME. El usuario podrá plantarse en cualquier momento. Una vez se plante, habrá terminado su turno y no podrá tirar dados. Recibido este mensaje en el servidor, se le responderá:
 - +Ok. Jugador <nombre>y Jugador <nombre>habéis empatado la partida. Si se trata del último jugador en plantarse en la partida. El servidor comprobará que ninguno de los jugadores ha excedido el número objetivo y ambos jugadores tienen el mismo número de puntos.
 - +Ok. Jugador <nombre>ha ganado la partida. En caso de que se trate del segundo jugador en plantarse. El servidor comprobará el jugador que tiene el número más próximo al objetivo sin haberse pasado y ese será el ganador.
 - +Ok. Esperando que finalice el otro jugador. Si se trata del primer jugador en plantarse. El servidor informará al jugador que se está a la espera de que el otro jugador se plante para poder finalizar y hacer el recuento. El otro jugador solo tiene la opción de plantarse.
 - El servidor debe llevar el control de la situación real de cada partida. Debe controlar que solamente tengan efecto los comandos según el contexto en el que se encuentre. Por ejemplo: el usuario no puede iniciar una partida si no se ha validado correctamente, el usuario no puede ejecutar los comandos de juego si no está dentro de una partida, el usuario no puede volver a tirar dados si ya se ha plantado en esa partida...
 - Para salir del servicio se usará el mensaje “SALIR”, de este modo el servidor lo eliminará de clientes conectados. Además, si estaba jugando, finalizará la partida en la que se encontraba, avisando al otro jugador: “+Ok. Tu oponente ha terminado la partida”, si estaba a la espera de una partida, lo elimina de la espera.

- Cualquier mensaje que no utilice uno de los especificadores detallados o cuya estructura presente algún error generará un mensaje de ‘-Err’ por parte del servidor.

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

- La comunicación será mediante consola.
- El protocolo deberá permitir mandar mensajes de tamaño arbitrario. Teniendo como tamaño máximo envío una cadena de longitud 200 caracteres.
- El servidor aceptará servicios en el puerto 1865.
- 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 20 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.

2.4. Resumen paquetes

Para toda la práctica, vamos a considerar los siguientes tipos de mensajes, cada uno con su respectivo formato:

- **USUARIO usuario:** mensaje para introducir el usuario que desea.
- **PASSWORD contraseña:** mensaje para introducir la contraseña asociada al usuario.
- **REGISTRO -u usuario -p password:** mensaje mediante el cual el usuario solicita registrarse para acceder al juego de los números.
- **INICIAR-PARTIDA:** mensaje para solicitar jugar una partida.
- **TIRAR-DADOS ndados:** mensaje para tirar dados.
- **NO-TIRAR-DADOS:** mensaje para no tirar dados y pasar el turno.
- **PLANTARME:** mensaje para indicar que ya no quieres tirar dados y te plantas con la puntuación que tienes.

- **SALIR:** mensaje para solicitar salir del juego.
- Cualquier otro tipo de mensaje que se envíe al servidor, no será reconocido por el protocolo como un mensaje válido y generará su correspondiente “-Err.” por parte del servidor.

2.5. 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.