



DPTO. DE INFORMÁTICA Y ANÁLISIS NUMÉRICO

UNIVERSIDAD DE CÓRDOBA



REDES – Práctica 2

Graduado en Ingeniería Informática

PRÁCTICA 2

| | |
|---|----|
| PRÁCTICA 2..... | 2 |
| 1. Fundamentos de la práctica 2 | 1 |
| 1.1 Estructura y Funciones útiles en la Interfaz Socket | 2 |
| 1.1.1 Estructuras de la interfaz socket..... | 2 |
| 1.1.2 Funciones de la interfaz socket | 3 |
| 1.2 Terminología y Conceptos del Procesado Concurrente | 6 |
| 2. Enunciado de la práctica 2 | 8 |
| 2.1 Objetivos del juego del Buscaminas | 8 |
| 2.2 Resumen paquetes | 12 |
| 2.3 Objetivo..... | 12 |

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 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:

```
struct hostent {
    char *h_name; /* nombre del host oficiales */
    char **h_aliases; /* otros alias */
    int h_addrtype; /* tipo de dirección */
    int h_length; /* 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 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.

```
/* 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 *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.

```
/* 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.

- (3° argumento, *len*), es la longitud de los datos en bytes.
- (4° 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:

- (1° argumento, *sockfd*), es el descriptor socket por donde se enviarán los datos.
- (2° argumento, *buf*), es el puntero a un buffer donde se almacenarán los datos recibidos.
- (3° argumento, *len*), es la longitud del buffer *buf*.
- (4° 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:

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

```
/* Función de recepción: read() */
read ( int sockfd, void *msg, in len )
```

dato

opción de que el **buffer** de

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*: valor 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*: 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);
```

Estructura timeval

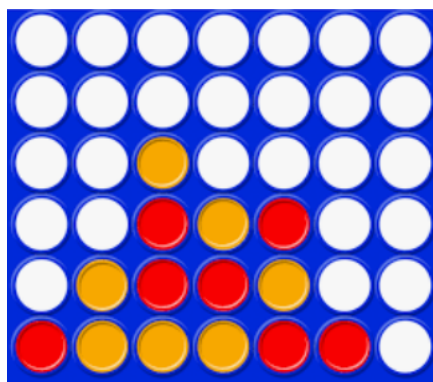
```
/* Estructura timeval */  
  
struct timeval  
{  
    unsigned long int tv_sec; /* Segundos */  
    unsigned long int tv_usec; /* Millonesimas de segundo */  
};
```

2. Enunciado de la práctica 2

Diseño e implementación del juego de 4 en línea para dos jugadores permitiendo jugar en red. El juego consiste en ser el primero en formar una línea con cuatro fichas el mismo color, ya sea horizontal, vertical o diagonal.

2.1 Objetivos del juego 4 en línea

Para jugar al 4 en línea es necesario una cuadrícula que inicialmente está vacía. Por turno, cada jugador coloca una ficha en una columna y fila determinada. El juego termina cuando un jugador consigue hacer una línea con cuatro fichas.



Al introducir una nueva ficha, el servidor comprobará si se ha formado una línea de 4 fichas del mismo color, en cuyo caso enviará un mensaje a los dos jugadores indicando que la partida ha terminado y el jugador que ha ganado.

Al no utilizarse interfaz gráfica, la representación será mediante una matriz, en la que se empleará la siguiente notación:

- “-“ para indicar que la casilla no tiene ninguna ficha.
- “o” para indicar que el jugador 1 es el que ha puesto la ficha en la casilla (el jugador 1 será el primer jugador que solicitó jugar la partida).
- “x” para indicar que el jugador 2 es el que ha puesto la ficha en la casilla (el jugador 2 será el segundo jugador que solicitó jugar esa partida).

El tablero será siempre de 6x7 y la partida termina cuando se completa una línea de 4 ‘o’ o 4 ‘x’ o bien ya no se pueden colocar fichas, en el que se termina con empate.

En la Figura 1 y 2, se puede apreciar un tablero al inicio, y uno con una partida empezada:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |

Figura 1. Tablero inicial

| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|
| | | | | | | |
| | | | | | | |
| | | | x | | | |
| | x | | o | | | |
| | o | x | x | | | |
| | o | o | x | | | |

Figura 2. Tablero con una partida empezada

2.2 Especificación del juego a implementar

La comunicación entre los clientes del juego del 4 en línea 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 que se especifican en esta sección. En el juego, 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 “+0k. Usuario conectado”.

- Cada usuario durante su turno podrá mandar el mensaje **COLOCAR-FICHA número**. Ante este mensaje, se colocará una ficha en la columna indicada en número en la primera fila que haya un hueco disponible:
 - Si no hubiese hueco disponible por estar las 6 filas cubiertas, se indicará el error al jugador que ha solicitado colocar la ficha y seguirá siendo el turno del mismo jugador. La especificación del mensaje que se enviaría sería: “-Err. Debe seleccionar otra columna que tenga alguna casilla disponible”.
 - Si hay hueco disponible, el servidor enviará a cada jugador el tablero con la información actualizada, indicando la posición de la nueva ficha. Ejemplo: “+Ok. Nuevo tablero. -, -, -, -, -, -; -, -, -, -, -, -; -, -, -, -, -, -; -, -, -, -, -, -, x, -, -, -”. De este modo, ambos usuarios estarán informados, de todos los movimientos, y el turno pasa al siguiente jugador.
 - Si colocando dicha ficha hace una línea de 4, el servidor enviará un mensaje a cada jugador, indicando que el jugador que ha realizado la línea ha ganado la partida, y finalizará dicha partida. Ejemplo: “+Ok. Jugador <nombreJugador> ha ganado la partida”.
 - Si colocando dicha ficha se completan todas las casillas disponible y no hay ninguna línea, el servidor enviará un mensaje a cada jugador, indicando que se ha producido un empate, y finalizará dicha partida. Ejemplo: “+oOk. Se ha producido un empate en la partida”.
- Se debe tener cuidado en 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 su turno y deben esperar. La especificación del mensaje que se enviaría sería: “-Err. Debe esperar su turno”.
- Para salir del servicio se usará el mensaje “SALIR”, de este modo el servidor lo eliminará de clientes conectados, si estaba jugando, finalizará la partida en la que se encontraba, avisando al otro jugador: “+Ok. Tu oponente ha salido de la partida”, si estaba a la espera de una partida, lo elimina de la espera; y si solamente estaba conectado, lo desconecta del sistema.
- Cualquier mensaje que no use uno de los especificadores detallados, generará un mensaje de “-Err. Texto informativo” 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 250 caracteres.
- El servidor aceptará servicios en el puerto 2060.
- 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.

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.
- **REGISTRO** *-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 hundir la flota.
- **COLOCAR-FICHA** *número* donde número indica la columna donde se colocará la ficha.
- **SALIR**: mensaje para solicitar salir del juego.
- Cualquier otro 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 paquete.