Lab 4

Redes de Computadoras

Sockets

Cliente y servidor de ficheros concurrente con multiplexación de e/s y control de eventos temporales.

Objetivos

El objetivo de esta práctica es aprender a utilizar la interface de programación de aplicaciones (API) para redes mediante la librería de sockets. Para ello deberán realizar una aplicación cliente y una aplicación servidor en UDP para el cálculo del tiempo de ida y vuelta (RTT, Round Trip Time) entre ambos y una aplicación cliente y una aplicación servidor en TCP para la transferencia de ficheros, utilizando el API de sockets. El protocolo para la transferencia de ficheros que se va a utilizar es una simplificación del protocolo File Transfer Protocol (FTP), RFC-959.

Descripción aplicación UDP

El servidor recibirá como argumento el puerto donde estará escuchando peticiones del cliente. Por ejemplo, si queremos que espere en el puerto 11000:

```
<nombreMaq>:~$ ./servidorRTT 11000
```

El cliente recibirá como argumento la dirección IP y puerto donde está el servidor esperando peticiones. Por ejemplo, si el servidor se está ejecutando en una máquina que tenga configurada la dirección IP 10.0.0.1 y está escuchando en el puerto 11000:

```
<nombreMaq>:~$ ./clienteRTT 10.0.0.1 11000
```

Al arrancar el cliente aparecerá un menú que muestra los tipos de operaciones que se pueden solicitar al servidor y se pide al usuario que introduzca la operación que desea realizar:

```
1. Calcular RTT
2. Almacenar RTT en el servidor
3. Obtener todos los cálculos de RTT almacenados en el servidor
Introduzca el número de la operación que desea realizar:
```

Estas operaciones consisten en:

1. Calcular RTT

El cliente envía un mensaje con una marca de tiempo que coincide con la hora del envió del mensaje. El servidor responde con la misma marca de tiempo. Cuando el cliente recibe la respuesta podrá obtener nuevamente la hora y calcular el tiempo que ha tardado el mensaje en llegar al servidor y en recibirse la respuesta en el cliente.

Se recomienda utilizar *gettimeofday* que proporciona precisión de microsegundos.

2. Almacenar RTT en el servidor

El cliente envía un mensaje al servidor para que este almacene la siguiente información:

a) RTT calculado.
b) Hora en la que el cliente hizo el cálculo de RTT. Este campo será transmitido por el cliente como una cadena de caracteres del tipo:

```
vie mar 2 06:45:26 CET 2007
```

c) Dirección IP y puerto del cliente.

3. Obtener todos los cálculos de RTT almacenados en el servidor

El cliente envía un mensaje al servidor solicitando la lista de RTTs que tiene almacenados. El cliente mostrara en la pantalla la siguiente información sobre cada uno de los RTTs:

a) Hora en la que el cliente hizo el cálculo.

b) Dirección IP y puerto del cliente al que se corresponde dicho cálculo.

c) Valor de RTT calculado en milisegundos.

A continuación se muestra un posible resultado de esta operación:

```
Hora --> IP:puerto --> RTT (ms)
vie mar 2 06:45:26 CET 2007 --> 10.0.0.1:7543 --> RTT = 200 ms
vie mar 2 06:47:36 CET 2007 --> 10.0.0.2:8000 --> RTT = 300 ms
vie mar 2 06:50:48 CET 2007 --> 10.0.0.1:7543 --> RTT = 250 ms
```

El usuario podrá solicitar las operaciones al servidor en cualquier orden, por este motivo, el cliente asignara un valor inicial de RTT = 1 segundo, para el caso en el que se solicite la operación 2 antes que la operación 1.

El servidor podrá atender a varios clientes y recibir las operaciones de los clientes en cualquier orden y entrelazadas con operaciones de diferentes clientes.

Para definir los tipos de mensajes que intercambiaran cliente y servidor, se utilizara un fichero de cabecera mensajesRTT.h que será incluido por el cliente y el servidor.

Descripción aplicación TCP

El servidor funcionará de forma iterativa, es decir, hasta que no termine de atender por completo a un cliente, no podrá aceptar nuevas peticiones de conexión de otros clientes.

El servidor recibirá como argumento el puerto donde estará escuchando peticiones del cliente. Por ejemplo, si queremos que espere en el puerto 11000:

```
<nombreMaq>:~$./srvFtp 11000
```

El cliente recibirá como argumento la dirección IP y puerto donde está el servidor esperando peticiones. Por ejemplo, si el servidor se esta ejecutando en una maquina que tenga configurada la dirección IP 10.0.0.1 y está escuchando en el puerto 11000:

```
<nombreMaq>:~$./clFtp 10.0.0.1 11000
```

Solicitudes y respuestas

Todas las operaciones que el cliente solicita al servidor son cadenas de caracteres con el siguiente formato:

```
<TIPO_OPERACION> [<parámetros>]\r\n
```

Todas las respuestas que el servidor envía al cliente son cadenas de caracteres con el siguiente formato:

```
<CODIGO_RESPUESTA> <descripción>\r\n
```

El código de respuesta será un número de 3 cifras y la descripción es una breve explicación de dicho código de respuesta.

Al arrancar el cliente se conectara al servidor de ficheros a través de un socket TCP. El servidor aceptará dicha conexión y le enviará un mensaje de saludo que consistirá en una cadena de caracteres formada por un código de respuesta y un mensaje que informe de la versión del servidor de ficheros que se está usando. Por ejemplo:

```
220 srvFtp version 1.0\r\n
```

El cliente imprimirá el mensaje recibido. A partir de ese momento el cliente estará preparado para enviar operaciones al servidor.

La primera operación que deberá realizar el cliente es la autenticación del usuario, para ello solicitara al usuario que introduzca un nombre de usuario. Por ejemplo:

```
username: lola
```

Lo que provocará el envío de un mensaje de texto al servidor de ficheros con la cadena de caracteres:

```
USER <nombreUsuario>\r\n
```

En el ejemplo anterior se enviará:

```
USER lola\r\n
```

El servidor responderá con una cadena de caracteres formada por un código de respuesta y una breve descripción de dicho código:

```
331 Password required for <nombreUsuario>\r\n
```

En el ejemplo anterior:

```
331 Password required for lola\r\n
```

El cliente imprimirá este mensaje y solicitará al usuario una palabra de paso. Por ejemplo:

```
passwd: passwd-lola
```

El cliente creara un nuevo mensaje de texto para el servidor con el siguiente formato:

```
PASS <passwdUsuario>\r\n
```

En el ejemplo anterior:

```
PASS passwd-lola\r\n
```

El servidor comprobará que el usuario y la palabra clave son válidos. Para ello leerá de un fichero llamado ftpusers los usuarios que están autorizados para realizar las operaciones de transferencia de ficheros.

Este fichero contiene líneas del tipo: **<nombreUsuario>:<passwdUsuario>**, donde cada línea corresponde a un nombre de usuario y su palabra clave separados por el caracter ':'.

Por ejemplo, un fichero ftpusers válido seria:

```
lola:passwd-lola
daniel:passwd-daniel
maria:passwd-maria
```

Si existe en el fichero ftpusers la pareja **<nombreUsuario>:<passwdUsuario>** proporcionado por el cliente el servidor responderá al cliente con el mensaje:

```
230 User <nombreUsuario> logged in\r\n
```

En el ejemplo:

```
230 User lola logged in\r\n
```

Si no existe en el fichero ftpusers la pareja **<nombreUsuario>:<passwdUsuario>**, el servidor responderá al cliente con un mensaje de error:

```
530 Login incorrect\r\n
```

Y el servidor cerrará la conexión del cliente. El cliente imprimirá el mensaje de error proporcionado por el servidor y terminará el programa cliente.

Si la autenticación ha funcionado correctamente, el usuario podrá solicitar algún fichero del servidor. Para ello, el cliente imprimirá el siguiente mensaje:

```
Operación:
```

y el usuario podrá solicitar un fichero escribiendo:

```
Operacion: get <nombreFichero>
```

El cliente enviará al servidor el siguiente mensaje de texto para solicitar el fichero:

```
RETR <nombreFichero>\r\n
```

Si el fichero existe en el directorio donde se ha arrancado el servidor, el servidor enviara la respuesta:

```
299 File <nombreFichero> size <tamaño> bytes\r\n
```

Y a continuación el servidor enviará el fichero que el cliente le ha solicitado. El envío se realizará leyendo fragmentos de 512 bytes del fichero y enviando cada uno de estos fragmentos, teniendo en cuenta que el último fragmento será de un tamaño menor o igual a 512 bytes.

Una vez enviado el fichero el servidor además enviará un nuevo mensaje para indicar que la transferencia ha terminado:

```
226 Transfer complete\r\n
```

El cliente, leerá el mensaje que indica el tamaño del fichero, leerá el fichero y lo almacenará en el directorio desde donde se ha arrancado el programa cliente y por último, leerá el mensaje que indica que la transferencia ha terminado.

Si el fichero solicitado por el cliente no existe en el servidor, el servidor devolverá un mensaje de error:

```
550 <nombreFichero>: no such file or directory\r\n
```

y el cliente imprimirá el mensaje de error recibido.

Tanto si el fichero existe como si no, el cliente presentara de nuevo el mensaje para esperar una nueva operación cuando la anterior haya terminado:

```
Operación:
```

Para salir del cliente el usuario deberá utilizar la operación quit:

```
Operación: quit
```

Esta operación provocara que el cliente envíe el mensaje de texto:

```
QUIT\r\n
```

Y el servidor contestara con:

```
221 Goodbye\r\n
```

En este momento, el servidor cerrará la conexión y el cliente terminará. El servidor quedará a la espera de nuevas peticiones de conexión de otros clientes.

Comprobación de funcionamiento

Para comprobar que la transferencia del fichero ha salido correctamente, es necesario utilizar desde el *shell* el comando *diff* para que compare el fichero original y el fichero transferido desde el servidor al cliente.

Utilize ficheros pequeños de menos de 512 bytes y ficheros grandes de hasta 1MByte.

El cliente y el servidor deberán arrancarse en directorios diferentes para que cuando se escriba el fichero que reciba el cliente no sobrescriba el fichero original que lee el servidor.

Ejemplo completo de una posible interacción cliente/servidor

```
alpha01:~$./clFtp 10.0.0.1 11000
220 srvFtp version 1.0
username: lola
```

```
331 Password required for lola
passwd: passwd-lola
230 User lola logged in
Operacion: get prueba.txt
299 File prueba.txt file size 25000 bytes
226 Transfer complete
Operacion: get prueba1.txt
550 prueba1.txt: no such file or directory
Operacion: quit
221 Goodbye
alpha01:~$
```

Servidor concurrente

La primera modificación será que el servidor funcione de forma concurrente, es decir, podrá atender simultáneamente las peticiones de diferentes clientes. Cuando un usuario desee terminar su comunicación con el servidor, tecleara el comando *quit* que provocara el cierre de la conexión y la terminación del programa cliente. Es importante que el servidor libere correctamente el/los proceso/s que había arrancado para comunicarse con dicho cliente, sin dejar procesos zombies.

Se modificará el cliente para que acepte como argumento el nombre o la dirección IP de la maquina y puerto donde está el servidor esperando peticiones. Por ejemplo, si el servidor se está ejecutando en una máquina que tenga configurado el nombre *alpha23* y está escuchando en el puerto 11000:

```
<nombreMaq>:~$ ./clFtp alpha23 11000
```

o también:

```
<nombreMaq>:~$./clFtp 212.128.4.193 11000
```

El cliente, tendrá que realizar la resolución del nombre de máquina a la dirección IP correcta antes de tratar de conectarse con el servidor, véase el apéndice B.

Conexiones diferentes para el control y para los datos

Uno de los modos de transferencia del protocolo FTP se denomina transferencia activa, y consiste en utilizar una conexión TCP para la transferencia de los comandos de control entre el cliente y el servidor y otras conexiones diferentes para el intercambio de datos. La primera conexión o conexión de control se realiza una única vez desde el programa cliente al servidor y se utiliza para realizar la autenticación y mandar al servidor las operaciones que se deseen realizar. La otras conexiones se inician desde el servidor hacia el cliente para el envió de los datos. Por ejemplo, cada vez que el usuario solicita un fichero, el servidor abrirá una nueva conexión para el envío de dicho fichero y concluirá dicha conexión.

Nuestro protocolo anterior de transferencia de ficheros solo utilizaba una conexión para realizar todas estas operaciones. Vamos a modificarlo para que el comportamiento de nuestro protocolo se parezca al modo de transferencia activa de FTP.

Dado que el servidor debe iniciar una conexión con el cliente para el envío de un fichero, es necesario que el servidor conozca la dirección IP y puerto donde el cliente espera recibir esos datos. La forma de que el servidor conozca estos valores es la siguiente: cuando el usuario solicita la transferencia de un fichero, antes de que el cliente envíe la operación RETR, el cliente abrirá un nuevo socket TCP para la transferencia de los datos desde el servidor al cliente y le enviara al servidor la información sobre la dirección IP y puerto donde esperara recibir los datos que el servidor tenga que enviarle.

El cliente utilizará la operación PORT para enviar la dirección IP y puerto donde espera recibir el fichero del servidor:

```
PORT <h1>,<h2>,<h3>,<h4>,<p1>,<p2>\r\n
```

donde cada uno de los números anteriores, separados por comas, representa grupos de 8 bits pertenecientes a la dirección IP (32 bits en total) y grupos de 8 bits pertenecientes al número de puerto (16 bits en total), considerando que h1 es el numero decimal que representa el byte más significativo de la dirección IP y p1 es el numero decimal que representa el byte más significativo del puerto.

Por ejemplo, si el cliente quiere enviar la dirección IP 127.0.0.1 y puerto 2242 utilizara la siguiente operación:

```
PORT 127,0,0,1,8,194\r\n
```

donde el valor de puerto es 8*256+194=2242.

El servidor responderá a esta operación con el siguiente código de respuesta indicando que todo ha salido bien:

```
200 PORT command successful\r\n
```

El servidor utilizará como número de puerto origen para establecer esta conexión, uno más que el valor asignado al puerto donde está esperando recibir las conexiones de los clientes. Es decir, si hemos arrancado el servidor para esperar las conexiones en el puerto 11000, el servidor utilizará el puerto origen 11001 para establecer la conexión de datos con el cliente.

A partir de este momento, el programa cliente enviara la operación *RETR* para obtener el fichero desde el servidor, utilizando la conexión creada anteriormente. Esta conexión solo se utilizará para el envío de ese fichero. Si el usuario solicita otro fichero con la orden *get*, el cliente nuevamente creara una nueva conexión de datos y enviara la operación *PORT* al servidor antes de solicitar el fichero con la operación *RETR*.

Cuando el cliente envíe la operación *RETR* por la conexión de control, el servidor responderá con el siguiente código de respuesta para indicarle que va a abrir la nueva conexión para la transferencia de datos:

```
150 Opening BINARY mode data connection for <nombreFichero> (<tamaño> bytes)\r\n
```

donde *nombreFichero* es el nombre del fichero que el servidor va a enviar y *tamaño* es el tamaño de dicho fichero.

A partir de este momento, el servidor iniciará la conexión de datos. Cuando termine de enviar el fichero, cerrara dicha conexión y enviara la siguiente respuesta a través de la conexión de control:

```
226 Transfer complete\r\n
```

El usuario podrá realizar nuevas operaciones.

Envió de ficheros desde el cliente al servidor

El usuario podía solicitar el envío de ficheros desde el servidor al cliente a través de la orden *get*. Ahora el usuario también podrá enviar ficheros desde el cliente al servidor a través de una nueva orden a implementar. La orden *put*.

```
Operación: put <nombreFichero>
```

Ante esta orden del usuario, el cliente enviara la operación *PORT* al servidor para configurar la nueva conexión de datos a través de la cuál el cliente enviará el fichero al servidor. Y el servidor le responderá con el código de respuesta *200* para indicar que todo ha ido bien. A continuación, el cliente enviará al servidor la siguiente operación:

```
STOR <nombreFichero>
```

El servidor le responderá al cliente a través de la conexión de control para indicarle que ya está dispuesto para recibir los datos:

```
150 Opening BINARY mode data connection for <nombreFichero>
```

Nótese que el servidor no responde con el tamaño del fichero porque no lo conoce.

El cliente enviara el fichero al servidor utilizando la conexión de datos, troceado en bloques de 512 bytes como máximo. Al finalizar, cerrará la conexión de datos y el servidor enviara el mensaje de transferencia finalizada al cliente a través de la conexión de control.

```
226 Transfer complete\r\n
```

El cliente quedará a la espera de una nueva orden por parte del usuario.

Comprobación de funcionamiento

Se realizarán las mismas comprobaciones con ficheros pequeños y grandes. Adicionalmente es importante comprobar con el comando *ps*, que cada vez que se hace un *fork* se cree un proceso hijo y que al terminar ese proceso hijo, se destruye y no quedan procesos zombies.

Ejemplo completo de una posible interacción entre un cliente y el servidor

```
alpha01:~$./clFtp 10.0.0.1 11000
220 srvFtp version 1.0
username: lola
331 Password required for lola
passwd: passwd-lola
230 User lola logged in
Operación: get prueba.txt
200 PORT command successful
150 Opening BINARY mode data connection for prueba.txt (25000 bytes)
226 Transfer complete
Operación: put prueba1.txt
200 PORT command successful
150 Opening BINARY mode data connection for prueba1.txt
226 Transfer complete
Operación: quit
221 Goodbye
alpha01:~$
```

Servidor multiplexado

Se pretende conseguir el mismo objetivo pero programando el servidor, multiplexando la e/s y siguiendo un modelo de programación orientado a eventos.

Existen dos funciones básicas que nos permiten multiplexar la e/s: *select* y *poll*. Para esta práctica vamos a utilizar la función select.

Utilización de la función select

La función *select* permite que un proceso pueda decirle al núcleo del SO que espere un conjunto de eventos y que despierte al proceso cuando alguno de ellos suceda o cuando haya transcurrido un plazo máximo.

```
#include <sys/select.h>
#include <sys/time.h>
#include <unistd.h>
int select(int n, fd_set *readset, fd_set *writeset, fd_set *exceptset, struct timeval *timeout);
```

n: numero de descriptor más alto configurado en select mas 1.

readset: conjunto de descriptores para operaciones de lectura.

writeset: conjunto de descriptores para operaciones de escritura.

exceptset: conjunto de descriptores para condiciones de excepción (ej: datos fuera de banda).

timeout: plazo máximo para la espera de eventos en los descriptores anteriores.

Un conjunto de descriptores de tipo *fd_set* en realidad es un mapa de bits donde cada bit se corresponde con un descriptor. Para poder utilizar los conjuntos de descriptores de tipo *fd_set* de forma sencilla, hay definidas una serie de macros que permiten activar y desactivar un descriptor en un conjunto:

```
void FD_CLR(int fd, fd_set *set); /* desactiva el descriptor fd en el conjunto set*/
int FD_ISSET(int fd, fd_set *set); /* comprueba si un descriptor fd esta activo en el conjunto set */
void FD_SET(int fd, fd_set *set); /* activa el descriptor fd en el conjunto set*/
void FD_ZERO(fd_set *set); /* limpia todos los descriptores de ese conjunto */
```

Antes de llamar a *select* es importante inicializar correctamente los conjuntos de descriptores. Para ello, primero limpiaremos todos los descriptores de ese conjunto con *FD_ZERO* y a continuación activaremos los descriptores en los que estamos interesados con *FD_SET*. Si alguno de los conjuntos de descriptores no lo vamos a usar, llamaremos a select con el valor *NULL* de ese parámetro.

La llamada *select* modifica los conjuntos de descriptores que le hemos pasado, por tanto, estos conjuntos son parámetros de entrada y salida. Cuando invocamos esta función, los conjuntos de descriptores representan los descriptores en los que estamos interesados. Cuando la función retorna, los conjuntos de descriptores representan cuáles de ellos están preparados para una operación de lectura/escritura/excepción.

Ideas para la programación

El programa servidor inicialmente deberá crear un socket para atender las peticiones de los clientes. En vez esperar las peticiones bloqueado en la llamada *accept*, utilizaremos la llamada *select* configurando adecuadamente el conjunto de descriptores preparados para realizar operaciones de lectura (*readset*), en este caso activando el descriptor del socket que espera las conexiones.

Como el programa servidor no tiene nada más que hacer, no será necesario configurar un plazo máximo de espera en la llamada *select*, queremos que *select* se quede bloqueado hasta que se reciba notificación de que hay datos en alguno de los descriptores de socket configurados.

Cuando un cliente trate de establecer la conexión, *select* despertará y nos avisará de que en ese socket tenemos datos esperando, en este caso, el establecimiento de conexión (segmento SYN de TCP). El servidor podrá invocar la llamada *accept* sin quedarse bloqueado, pues sabemos con seguridad que ha llegado una petición de establecimiento de conexión de un cliente. Al aceptar la conexión con este primer cliente tendremos dos descriptores de socket que tendremos que gestionar para manejar concurrentemente las peticiones de nuevos clientes (en el socket servidor) y el intercambio de datos con este primer cliente que se acaba de conectar (socket conectado).

Ahora habrá que configurar nuevamente la llamada *select* para que atienda a eventos de lectura que se puedan producir en estos dos descriptores. Esta situación se repetirá con cada cliente que se conecte al servidor.

En el caso en el que el servidor tenga que enviar un fichero al cliente (el servidor ha recibido la orden *RETR*), el servidor debería hacerlo estableciendo una nueva conexión con el cliente. Esta conexión solo se utiliza para enviar los datos en el sentido **servidor->cliente** y por tanto, el servidor no va a utilizar ese socket para realizar operaciones de lectura, solo de escritura. En el supuesto en el que el fichero sea muy grande, el servidor podría tardar un tiempo no despreciable en enviarlo. El servidor entraría en un bucle en

el cual realizaría operaciones de lectura del fichero y escritura en el socket y dejaría de atender el resto de los descriptores de socket que tuviera abiertos.

Para evitar esta situación sería conveniente que esta nueva conexión y envío de datos al cliente se realice fuera del hilo principal del programa, por ejemplo utilizando un nuevo proceso a través de la llamada *fork*. Cuando el proceso hijo termine enviara la señal *SIGCHLD* al proceso padre que deberá tratar adecuadamente para eliminar los procesos zombie. Puede ocurrir que esta señal se reciba en el proceso padre cuando está bloqueado en la llamada *select*. Si esto sucede *select* despertará y devolverá como valor de retorno **-1**. Es necesario comprobar si se produce esta situación "de error" por la llegada de una señal *SIGCHLD* y si es así, volver a invocar la llamada a *select*, pues es una situación que estaba prevista en nuestro programa. Para saber si es la llegada de una señal lo que ha provocado el retorno de select será necesario comprobar el contenido de la variable *errno* que en esta situación valdría *EINTR* (consultar el manual de *select*).

Nótese que cuando es el cliente el que tiene que enviar el fichero al servidor, la nueva conexión que se establece desde el servidor al cliente va a ser utilizada por el servidor para realizar operaciones de lectura de los bloques del fichero. En este caso, estas operaciones de lectura deberían estar gestionadas por la llamada *select* para que solo sean realizadas cuando verdaderamente haya algo que leer en el socket.

Comprobación del funcionamiento

Sera necesario comprobar:

1. El servidor atiende a varios clientes simultáneamente.
2. El servidor no deja procesos en estado zombie cuando se realizan operaciones *RETR*.
3. Los ficheros transferidos son iguales a los originales.
4. Los ficheros transferidos pueden ser pequeños (menor de 512 bytes) o grandes (más de 1MByte).

Mejoras al cliente

Se añadirá un nuevo comando que el usuario podrá introducir para modificar el directorio actual que se utiliza para el envío de ficheros y para la recepción de los mismos. Hasta ahora, el directorio utilizado para tal fin era el directorio desde el cual se arrancaba la aplicación cliente. Para modificar este directorio se utilizara la orden *lcd*:

```
Operación: lcd <nombreDirectorio>
```

Ante esta orden, el programa cliente utilizara el *nombreDirectorio* como directorio desde el cuál tomar los ficheros para enviarlos al servidor (cuando se ejecute la orden *put*) y como nombre de directorio donde se almacenaran los ficheros transferidos por el servidor (cuando se ejecute la orden *get*). En el caso de que el directorio no exista en la máquina donde se ejecuta la aplicación cliente, se presentará el siguiente mensaje en pantalla:

```
<nombreDirectorio>: No such file or directory
```

En el cuadro 1 del apéndice A se encuentran resumidos los mensajes que enviará la aplicación cliente al servidor.

Mejoras al servidor

Será necesario que el programa servidor compruebe el tiempo transcurrido desde que le llego el último mensaje de petición de operación de cada uno de los clientes que tiene conectados. Si ese tiempo es superior a 1 minuto, el servidor cerrará la conexión con

dicho cliente. En este caso, el cliente presentara el siguiente mensaje y terminara su ejecución:

```
Connection closed by server
```

Para controlar el tiempo transcurrido desde la recepción del último mensaje de petición de operación de un cliente se utilizara la función *gettimeofday*, que nos devuelve la hora en ese instante, cada vez que llega un mensaje desde un cliente. Para cada cliente será necesario almacenar ese valor que indicará la hora de último mensaje recibido por dicho cliente. Una vez almacenado ese valor, el servidor procesará la petición recibida y de nuevo esperará datos en los descriptores de socket que se encuentren activos utilizando la llamada select.

Para que la llamada *select* no se quede bloqueada un tiempo indefinido, utilizaremos el último argumento de dicha función que permitirá definir un plazo máximo para la espera de eventos definidos en *select*.

El valor del tiempo de espera tendrá que calcularse para que no sobrepase el minuto de tiempo que como máximo un cliente puede estar sin enviar mensajes al servidor. Por tanto, habrá que calcularse cuál es la hora más antigua en la que se recibió un mensaje de uno de los clientes y averiguar cuánto tiempo queda para que se cumpla un minuto. Ese tiempo es el que habrá que pasarle como último argumento a *select*.

Cuando la llamada a *select* retorne ahora será necesario comprobar si retorna por un evento en alguno de los descriptores o por un evento temporal, es decir, se ha cumplido el plazo definido en la variable *timeout*. Si se da este último caso, será necesario desconectar el socket con ese cliente.

Comprobación del funcionamiento

Será necesario comprobar:

1. El servidor atiende a varios clientes simultáneamente.
2. El servidor no deja procesos en estado zombie cuando se realizan operaciones RETR.
3. Los ficheros transferidos son iguales a los originales.
4. Los ficheros transferidos pueden ser pequeños (menor de 512 bytes) o grandes (más de 1Mbyte).
5. El servidor desconecta al cliente si ha transcurrido un minuto de tiempo desde que le envío el último mensaje.

Apéndice A: Resumen de mensajes

USER <nombreUsuario>\r\n	Mensaje de nombre de usuario
PASS <passwdUsuario>\r\n	Mensaje de clave del usuario
PORT <h1>,<h2>,<h3>,<h4>,<p1>,<p2>\r\n	Mensaje para configurar dirección IP y puerto donde el cliente espera los datos del servidor
RETR <nombreFichero>\r\n	Mensaje para obtener fichero
STOR <nombreFichero>\r\n	Mensaje para enviar fichero
QUIT \r\n	Mensaje de desconexión

Cuadro 1: Resumen de mensajes de solicitud del cliente

220 srvFTP version 1.0\r\n	Mensaje de saludo
331 Password required for <nombreUsuario>\r\n	Mensaje de respuesta a USER, solicitando clave
230 User <nombreFichero> logged in\r\n	Mensaje de respuesta a PASS, autenticación correcta
530 Login incorrect\r\n	Mensaje de respuesta a PASS, autenticación incorrecta
299 User File <nombreFichero> size <tamaño> in\r\n	Mensaje de respuesta a RETR con el nombre y tamaño de fichero a enviar
200 PORT command successful\r\n	Mensaje de respuesta a PORT
150 Opening BINARY mode data connection for <nombreFichero>\r\n	Mensaje de respuesta a STOR con el nombre que se va a recibir en el servidor
150 Opening BINARY mode data connection for <nombreFichero> (<tamaño> bytes)\r\n	Mensaje de respuesta a RETR con el nombre y tamaño de fichero que va a enviar el servidor
226 Transfer complete\r\n	Transferencia de fichero terminada
550 <nombreFichero>: no such file or directory\r\n	Mensaje de respuesta a RETR, el fichero no existe
221 Goodbye	Mensaje de despedida

Cuadro 2: Resumen de mensajes de respuesta del servidor

Apéndice B: Resolución de nombres

Dado un nombre de máquina podemos usar la función *getaddrinfo* para obtener la lista de direcciones IP posibles que están asociadas a dicho nombre de máquina:

```
int getaddrinfo(const char *hostname, const char *servname,
        const struct addrinfo *hints, struct addrinfo **res);
```

Proporcionaremos al parámetro *hostname* el nombre de máquina que queremos resolver y la lista de posibles direcciones se devolverá en el parámetro *res*.

El parámetro *servname* no lo utilizaremos porque se usa para convertir el nombre de un servicio en un número de puerto, nosotros estamos dando el valor de puerto ya como un dato numérico, así que no es necesario utilizar la resolución del puerto.

El parámetro *hints* se utiliza para restringir la lista de resultados devuelta en *res*. Por ejemplo, el resultado de *getaddrinfo* devolverá todas las direcciones IP asociadas a un cierto nombre. En particular, podría devolver las direcciones IPv4 e IPv6 asignadas a una máquina. En nuestro caso, solo estamos interesados en las direcciones IPv4.

Consulte el manual para ver todos los campos que tiene la estructura *addrinfo*.

Nótese que al utilizar *getaddrinfo* se devuelve en el parámetro *res* una lista de todas las posibles direcciones IP, asociadas a dicho nombre, en algún sitio alguien habrá reservado memoria para esa lista y será necesario avisar para que se libere cuando hayamos terminado de usarla. Para ello se utilizará la función *freeaddrinfo*.