E.T.S.I. TELECOMUNICACIÓN
FUNDAMENTOS DE SOFTWARE DE COMUNICACIONES
Laboratorio

APELLIDOS, NOMBRE:
TITULACIÓN:

Se requiere implementar un servicio de transferencia de ficheros redundante, en el que misma información se envía por dos canales de comunicación diferentes. En este caso, se trata de una aplicación cliente/servidor TCP donde el cliente envía nombres de fichero que se alojan en el servidor, y éste envía su contenido al cliente simultáneamente por dos sockets distintos emulando, por ejemplo, que cada uno estuviese conectado a una red distinta (redundancia).

Ejercicio 1. Servidor [2 puntos]

Implemente el servidor para dar el servicio de transferencia de ficheros redundantes como sigue.

1. El servidor dispone de dos sockets, **`s1`** y **`s2`**, por el que debe enviar los ficheros a los clientes. Los puertos de la conexión son, respectivamente, el **`5001`** y **`5002`**.
2. Cuando se conecta un cliente, lo hace consecutivamente primero por **`s1`** y después por **`s2`**.
3. Intercambio de información cliente-servidor
    a. El servidor espera a recibir el nombre del fichero a transferir por el socket **`s1`**. El nombre viene en formato longitud (**`uint16_t`**) + cadena. La transferencia de ficheros con el cliente termina si, en este paso, se detecta que éste se ha desconectado.
    b. El servidor entonces abre el fichero:
        i. Si el fichero existe, se envía por **`s1`** un código de error (**`uint8_t`**) con el valor **`0`** y pasamos al **Paso 3.c**.
        ii. Si el fichero no existe, se envía por **`s1`** el código de error (**`uint8_t)`** con el valor **`1`** y volvemos al **Paso 3.a**.
    c. Envío del contenido del fichero. Se utiliza una PDU con formato [**`tamaño de bloque (uint8_t) + bloque de datos`**]. Cada PDU se envía primero por **`s1`** y seguidamente por **`s2`**.
    d. Después de la última PDU con datos, se envía una PDU vacía, es decir, con **tamaño de bloque 0** y sin contenido. Esto notifica al cliente el el fin de la transferencia del fichero en curso.
    e. Ir al **Paso 3.a**: se vuelve a esperar un nuevo nombre de fichero.
4. Una vez terminada la sesión de envío de ficheros de un cliente, se acepta al siguiente y se sigue el procedimiento anterior.
5. Detalles de implementación:
    a. En todo caso se ha de garantizar que se lee y escribe lo esperado.
    b. Se han de liberar correctamente todos los recursos reservados.

Se entrega el fichero **ejercicio1.c/cpp.**

Ejercicio 2. Cliente [2 puntos]

Implemente el cliente para este servicio, según la siguiente especificación:

1. El cliente recibe como argumento de entrada la dirección IP del servidor.
2. Se crean dos sockets, **`s1`** y **`s2`**, que usan esa IP y los puertos donde escucha el servidor.
3. Bucle del cliente:
    a. Se solicita un nombre de fichero por teclado. Si el usuario introduce la cadena "**`fin`**", el cliente finaliza su ejecución ("**`fin`**" no se envía al servidor en ningún caso). En caso contrario, continúa con el siguiente paso.
    b. Se envía el nombre del fichero por **`s1`**, en formato longitud (**`uint16_t`**) + cadena.
    c. El contenido se almacena por separado en dos ficheros, cuyos nombres se construyen concatenando "**`.s1`**" y "**`.s2`**" al nombre leído desde teclado.
    d. Se recibe el código de error (**`uint8_t`**) por **`s1`**:
        i. En caso de ser **`0`** (ausencia de error): ir al **Paso 3.d**.
        ii. En caso de ser **`1`** (el fichero no existe), se notifica al usuario con el mensaje "**`Fichero no encontrado`**", y se va al **Paso 3.a**.
    e. Recepción del contenido del fichero.
        i. La transferencia se realiza en bloques, usando una PDU con formato [**`tamaño de bloque (uint8_t) + bloque`**].
        ii. Los bloques llegan, secuencialmente, primero por **`s1`** y después por **`s2`**. Sean **`b1`** y **`b2`** los bloques recibidos por **`s1`** y **`s2`**, respectivamente.
        iii. Comparar el contenido de **`b1`** y **`b2`**.
            1. Si coincide, se lleva su contenido a disco, cada uno en un el fichero correspondiente.
            2. Si no coincide, en lugar del contenido, se escribe en disco "**`Bloque inconsistente`**".
        iv. Independientemente del resultado de la comparación, se han de leer todos los bloques de fichero.
    f. Tras completarse la transferencia, si hay algún bloque cuyo contenido no ha coincidido, se informa al usuario con el mensaje "**`Transferencia inconsistente: descartar fichero`**".
4. Notas:
    a. En todo caso se ha de garantizar que se lee y escribe lo esperado.
    b. Se han de liberar correctamente todos los recursos reservados.

Se entrega el fichero **ejercicio2.c/cpp.**

Ejercicio 3. Máquina de estados [2 puntos]

Implemente una máquina de estados que formaría parte de un protocolo que recibe información redundante por dos canales de comunicación y debe comprobar que, efectivamente, la información que llega desde ambos canales coincide, pero además, debe hacerlo en un intervalo de tiempo dado. Es decir, se trata de un canal redundante con requisitos de sincronismo. Los detalles son los siguientes:

1. Los canales de comunicación se emulan mediante dos FIFOs, "**`c1`**" y "**`c2`**".
2. La máquina comienza en el estado **`E1`**, donde espera a recibir un bloque de datos por cualquiera de los canales. Sea este bloque **`b1`**.

3. A la llegada del bloque de datos, se pasa al estado **`E2`**. Estando en **`E2`**:
    a. Si se está durante **`5.25`** segundos sin recibir datos por el otro canal, se dispara el evento **`TIMEOUT`** y la máquina termina mostrando por pantalla el mensaje "**`Bloque de datos no recibido a tiempo por el otro canal`**".
    b. Si llega un nuevo bloque de datos por el otro canal en tiempo, se compara su contenido con el recibido en el estado **`E1`**. Si **`b2`** es este segundo bloque de datos, entonces:
        i. Si **`b1`** coincide con **`b2`**, la máquina transita a **`E1`**.
        ii. Si **`b1`** no coincide con **`b2`**, la máquina finaliza mostrando el mensaje "**`Contenido inconsistente`**".
4. La máquina está infinitamente recibiendo bloques de datos.

Código de referencia (arriba, partes de un servidor orientado a conexión; tras la línea de separación, partes de un cliente):

```
int sockfd;
struct sockaddr_in server_addr, client_addr;
socklen_t sin_size = sizeof(client_addr);

sockfd = socket(PF_INET, ???, 0);
/* ... */
memset((char *)&server_addr,0,sizeof(server_addr));
server_addr.sin_family = AF_INET;
server_addr.sin_port = ???;
server_addr.sin_addr.s_addr = INADDR_ANY;
bind(sockfd, (struct sockaddr *)&server_addr, sizeof(struct sockaddr));
/* ... */
???= listen(sockfd, 10);
/* ... */
???= accept(sockfd, (struct sockaddr *)&client_addr, &sin_size);
/* ... */

----------------------------------------

/* utilice la llamada al sistema socket() */
/*...*/
/* cambie esta IP de ejemplo por la correcta */
uint32_t dir=inet_addr("192.168.1.1");
struct sockaddr_in destino;
memcpy(&destino.sin_addr, &dir, 4);
/* siga rellenando la direccion destino */
/* y utilice la llamada al sistema connect()*/
```

```
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <sys/select.h>
#include <signal.h>
```