1. Introducción

Muchas aplicaciones que utilizan sockets emplean protocolos de aplicación que se basan en enviar cadenas de caracteres entre procesos clientes y servidores. Cualquier dato se codifica en una cadena de caracteres y está se envía como petición y respuesta. Suele ser habitual que estas cadenas de caracteres se representen como un conjunto de caracteres que finalizan con el código ASCII 0 (8 bis a 0), el cual sirve como delimitador de la cadena, de igual forma que ocurre en lenguajes de programación como C. En otras ocasiones la cadena de caracteres finaliza con el código ASCII de salto de línea ('\n'), en este caso el protocolo está orientado al envío y recepción de líneas.

El objetivo de este laboratorio, que se realizará durante dos sesiones, es realizar pequeñas aplicaciones con sockets TCP que utilizan cadenas de caracteres como datos a intercambiar en el proceso de comunicación.

En aula global se dispone del material de apoyo para el este laboratorio. El material incluye los siguientes archivos:

```
Makefile
lines.h
lines.c
test.c
client.c
server.c
client.java
```

El archivo `lines.c` incluye la definición de las siguientes funciones:

```
int sendMessage(int socket, char *buffer, int len);
```

Esta función incluye el código para enviar un mensaje de una determinada longitud por un socket.

```
int recevMessage(int socket, char *buffer, int len);
```

Esta función incluye el código para recibir un mensaje de una determinada longitud por un socket.

```
ssize_t readLine(int fd, void *buffer, size_t n);
```

Esta función lee de un descriptor (de fichero o de socket) una cadena de una longitud máxima de `n` bytes (incluido el código terminador de la cadena). Si se sobrepasa este número, el resto de los caracteres se descartan. La función devuelve una cadena de caracteres que finaliza con el código ASCII 0. Esta función lee una secuencia de bytes y los almacena en el buffer pasado como segundo argumento hasta que se han leído `n-1` bytes. El resto de los bytes leídos o recibidos se descartan. La función finaliza retornando una cadena de caracteres (con el código terminador incluido) cuando ocurre alguna de las siguientes circunstancias:

- El descriptor de fichero hace referencia a un fichero y se alcanza el fin de fichero.
- El descriptor de fichero hace referencia a un fichero y se lee el código ASCII '`\0`' o el salto de línea ('`\n`'). La cadena devuelta no incluye el salto de línea.
- El descriptor de fichero hace referencia a un socket y se lee el código ASCII '`\0`' o el salto de línea

('\n'). La cadena devuelta no incluye el salto de línea, pero sí el código ASCII '\0'.

Cuando se quiere enviar por un socket una cadena de caracteres (por ejemplo, `cadena`) incluido el código de fin de cadena (código ASCII 0) se puede utilizar la función `sendMessage` de la siguiente forma:

```
char cadena[256];
strcpy(cadena, "Cadena a enviar");

enviar(cadena, strlen(cadena)+1);
// envía todos los caracteres de la cadena y el código ASCII 0.
```

El archivo `test.c` incluye también el código que hace uso de la función `readLine`. Compile y pruebe este programa para entender el funcionamiento de esta función. Este programa, que se muestra a continuación, lee líneas que finalizan con el salto de línea ('\n') por la entrada estándar y las imprime por pantalla con `printf()`.

```
#define MAX_LINE        256

int main(int argc, char **argv)
{
        char buffer[MAX_LINE];
        int err = 0;
        int n;

        while (err != -1) {
                n = readLine(0, buffer, MAX_LINE);
                if (n!=-1)
                        printf("%s\n", buffer);
        }
        return (0);
}
```

2. Laboratorio de la primera sesión

Ejercicio 1. El objetivo de la primera parte del laboratorio consiste en programar un cliente (`client.c`) y un servidor (`server.c`) utilizando sockets de tipo *stream* en C. El cliente y el servidor se envían cadenas de caracteres finalizadas con el código ASCII '\0'; Se proporcionan los archivos `cliente.c` y `servidor.c` que se utilizarán como base para desarrollar estos programas. Utilice y modifique los archivos `client.c` y `server.c` proporcionados como material de apoyo.

El cliente realizará las siguientes acciones:

1. El cliente recibe como argumento en la línea de mandatos la dirección y el puerto del servidor. Para probarlo de forma local con un servidor ejecutando en el puerto 2000, puede invocar al cliente de esta forma:
   ```
   ./cliente    localhost    2000
   ```
2. Se conecta con el servidor arrancado en la dirección y puerto pasados como argumentos.
3. Una vez conectado, ejecuta un bucle infinito. En cada iteración lee de la entrada estándar una cadena utilizando la función `readLine` y la envía al servidor utilizando `sendMessage`. A continuación, recibe del servidor la misma cadena utilizando `readLine` y la imprimirá por pantalla utilizando `printf`. Cuando el usuario teclee "`EXIT`", la cadena se enviará al servidor, se recibirá la respuesta, que se mostrará por pantalla, y el cliente saldrá del bucle, cerrará la conexión y finalizará la ejecución.

Observe que el cliente utiliza la función `readLine` para leer de la entrada estándar y del socket. Lo único que cambia es el descriptor pasado como argumento. En el primer caso utiliza el descriptor 0 asociado a la entrada estándar y en el segundo el descriptor asociado al socket.

El servidor (en esta primera tarea será secuencial) acepta una conexión de un cliente y entrará en un bucle en el que recibirá una cadena del cliente (`readLine`) y le devolverá la misma cadena de vuelta al cliente (`sendMessage`). Del bucle se saldrá cuando se reciba del cliente la cadena "`EXIT`". A continuación, el servidor volverá a ejecutar la llamada `accept` para aceptar la conexión de otro cliente. El servidor **recibe** el puerto en el que tiene que aceptar las conexiones como argumento en la línea de mandatos.

Ejercicio 2. En el código desarrollado en el ejercicio 1, el servidor solo puede atender a un cliente. Mientras está en el bucle dialogando con el cliente, no ejecuta la llamada `accept` y, por tanto, no podrá atender a otros clientes. El objetivo de la segunda parte es convertir el código del servidor en concurrente utilizando procesos ligeros. De esta forma el servidor podrá dialogar simultáneamente con varios clientes.

3. Laboratorio de la segunda sesión

Ejercicio 3. En la primera parte de la segunda sesión de laboratorio se realizará la implementación de un cliente en Python que utilizará el servidor previamente implementado.

La forma de enviar un string a través de un socket TCP es la siguiente:

```
message = "Cadena a enviar"
message = message + "\0"
connection.sendall(message.encode())
```

La siguiente función se puede utilizar para leer un string de un socket:

```
def readString(sock):
    a = ''
    while True:
        msg = sock.recv(1)
        if (msg == b'\0'):
            break;
        a += msg.decode()

    return(a)
```

Ejercicio 4. Escriba un programa servidor en Python (`server.py`) con la misma funcionalidad que el desarrollado en C en la primera sesión de laboratorio. Prueba el servidor desarrollador con el cliente C y el cliente Python.