

# UNIDAD 2

# ACTIVIDAD PRÁCTICA

# PROGRAMACIÓN

# CLIENTE-SERVIDOR

## ÍNDICE

ORGANIZACIÓN DE LA ACTIVIDAD PRÁCTICA.....	3
DESCRIPCIÓN DE LA ACTIVIDAD PRÁCTICA .....	3
RECOMENDACIONES E INDICACIONES PARA LA ENTREGA.....	3
RÚBRICA DE CORRECCIÓN .....	8
COMO REALIZAR MI ENTREGA .....	9

## ORGANIZACIÓN DE LA ACTIVIDAD PRÁCTICA

Nombre de la práctica	
Tipo de tarea	Individual
Entregables	Proyecto de Linux/CMake Documento con imágenes de ejemplos de ejecución con hasta tres clientes y un servidor.

## DESCRIPCIÓN DE LA ACTIVIDAD PRÁCTICA

En este ejercicio se pedirá desarrollar un programa tipo cliente-servidor que permita simular un sistema sencillo tipo Chat-Room con interfaz por terminal de texto. La idea principal de estos programas es la siguiente:

- Programas tipo cliente: Estos programas ofrecen una interfaz sencilla de usuario para escribir y leer mensajes. Al iniciar el programa se pide un nombre de usuario para identificarse en el sistema, y más adelante permite escribir los mensajes que el usuario mandará al sistema. En caso de haber más clientes conectados al sistema, reciben los mensajes escritos por ellos para mostrarlos al usuario.
- Programa tipo servidor: Este programa se mantiene en bucle infinito esperando conexiones de clientes y recibiendo/reenviando los mensajes. Cada cliente se conecta al servidor, y éste crea una lista de usuarios que compartirán mensajes. El servidor recibirá los mensajes de cada cliente, y se encargará de reenviarlos al resto de clientes conectados en modo “broadcast”.

Para implementar el sistema pedido, se proporciona la plantilla de proyecto “libUtils.zip” y el pseudocódigo de los programas pedidos, que se deberá implementar en los archivos “client.cpp” y “server.cpp”. El pseudocódigo proporcionado está pensado para realizar una implementación sencilla del programa, no tiene todas las funcionalidades de programas tipo chat maduros. Adicionalmente, se da libertad al alumno para añadir funcionalidades extra que considere necesario. En caso de añadir funcionalidad no pedida al sistema, se valorará en una de las secciones de la rúbrica en función de lo añadido.

Contenido archivo “client.cpp”:

```
#include "utils.h"
#include <string>
#include <iostream>
#include <string>

using namespace std;

//Función para recibir en paralelo mensajes reenviados por el servidor
//Recibe por parámetros el ID del servidor para recibir datos, y una variable compartida
//"salir"
void recibeMensajesClientes(int serverId, bool& salir)
{
    vector<unsigned char> buffer; //Buffer de datos que se recibirán desde el cliente.
    //Desempaquetaremos todas las variables que
    //empaquetó el cliente en el mismo orden
    string nombreUsuario;//Variable para almacenar el nombre del usuario que escribió el
                        //mensaje
    string mensaje; //Variable para almacenar el mensaje
    //bucle mientras no salir
    while (!salir) {
```

```

//Recibir mensaje del servidor

//si hay datos en el buffer:
if (buffer.size() > 0) {
    // - desempaquetar del buffer la longitud del nombreUsuario
    // - redimensionar variable nombreUsuario con esa longitud, para poder
    //   almacenar el nombreUsuario
    // - desempaquetar del buffer la longitud del mensaje
    // - redimensionar variable mensaje con esa longitud, para poder almacenar
    //   el mensaje

    //mostrar mensaje recibido
    cout << "Mensaje recibido:" << nombreUsuario << " dice:" << mensaje << endl;
}
else {
    //conexión cerrada, salir
}
}

int main(int argc, char** argv)
{
    vector<unsigned char> buffer; //Buffer de datos que se enviará al server.
        //Empaquetaremos todas las variables que
        //queremos enviar dentro de este buffer para
        //realizar un único envío.

    string nombreUsuario;// Nombre del usuario que entra al chat
    string mensaje;// Mensaje que escribe el usuario
    bool salir = false;

    //Pedir nombre de usuario por terminal
    cout << "Introduzca nombre de usuario:" << endl;
    getline(cin, nombreUsuario);

    //Iniciar conexión al server en localhost:3000
    auto conn = initClient("127.0.0.1", 3000);

    //Iniciar thread "recibeMensajesClientes". Debe pasarse por parámetros las
    //variables "conn.serverId" y una referencia compartida a "salir"
    thread* th = new thread(recibeMensajesClientes, conn.serverId, ref(salir));

    //crear buffer de envío que contenga datos de nombre de usuario y texto
    //empaquetar tamaño de nombreUsuario
    //empaquetar datos de nombreUsuario
    //Enviar buffer al servidor

    //IMPORTANTE limpiar buffer una vez usado:
    buffer.clear();

    //Bucle hasta que el usuario escribe "exit()"
    do {
        //Ler mensaje de texto del usuario
        getline(cin, mensaje);

        //crear buffer de envío que contenga datos de nombre de usuario y texto
        //empaquetar tamaño de mensaje
        //empaquetar datos de mensaje
        //Enviar buffer al servidor

        //IMPORTANTE limpiar buffer una vez usado:
        buffer.clear();
    }
}

```

```

} while (mensaje != "exit()");
//Marcar variable "salir" para detener el hilo de recepción de mensajes
salir = true;
th->join(); //sincronizar con el thread antes de cerrar la conexión

//Cerrar conexión con el servidor

return 0;
}
  
```

#### Contenido del archivo “server.cpp”:

```

#include "utils.h"
#include <iostream>
#include <string>
#include <thread>
#include <list>
#include <algorithm>

using namespace std;

//Función para recibir en paralelo mensajes de un cliente que inició conexión con el servidor
//Recibe por parámetros el ID del cliente para recibir datos, y una variable compartida "users"
//La variable "users" contendrá la lista de usuarios que se han conectado hasta ahora, y se usará
//para reenviar los mensajes
void atiendeConexion(int clientId, list<int>& users)
{
    vector<unsigned char> buffer; //Buffer de datos que se recibirán desde el cliente.
    //Desempaquetaremos todas las variables que
    //empaquetó el cliente en el mismo orden
    string nombreUsuario;//Nombre de usuario
    string mensaje;// Mensaje de texto enviado por el usuario

    //Recibir nombre de usuario del cliente en el buffer
    //Desempaquetar nombre de usuario:
    //Para reconstruir un nombre:
    // - desempaquetar del buffer la longitud del nombre
    // - redimensionar variable nombreUsuario con esa longitud, para poder almacenar el nombre
    // - desempaquetar el texto del nombre en los datos de la variable nombreUsuario

    //mostrar mensaje de conexión
    cout << "Usuario Conectado:" << nombreUsuario << endl;
    //añadir clientId a la vector "users" compartido por todos los hilos de clientes
    users.push_back(clientId);

    //Bucle repetir mientras no reciba mensaje "exit()" del cliente.
    //El mensaje "exit()" se usará como señal de salida del cliente, en ese momento se cierra conexión
    //con él y acaba este thread
    do
    {
        //Recibir mensaje de texto del cliente
        recvMSG(clientId, buffer);
        //si hay datos en el buffer, recibir
        if (buffer.size() > 0) {
            //Para reconstruir un mensaje de texto:
            // - desempaquetar del buffer la longitud del mensaje
            // - redimensionar variable mensaje con esa longitud, para poder almacenar el mensaje
            // - desempaquetar el texto del mensaje en los datos de la variable mensaje

            //mostrar mensaje recibido
            cout << "Mensaje recibido:" << nombreUsuario << ":" << mensaje << endl;

        }
    }
    //reenviar a cada uno de los usuarios
  
```

```

//crear buffer de envío que contenga datos de nombre de usuario y texto
//empaquetar tamaño de nombreUsuario
//empaquetar tamaño de mensaje
//empaquetar datos de nombreUsuario
//empaquetar datos de mensaje
//bucle por cada identificador almacenado en la lista "users"
for (auto& userId : users)
{
    //enviar nuevo buffer usando "userId", sin reenviar al que lo envió
}

//IMPORTANTE limpiar buffer una vez usado:
buffer.clear();
}
else {
    //si no hay datos, error de conexión (el cliente cerró sin enviar mensaje de "exit()")
    //añadir código para enseñar el error y salir sin cerrar el servidor
}

} while (mensaje != "exit()");

//eliminar al cliente de la lista
users.erase(find(users.begin(), users.end(), clientId));

//cerrar conexión con el cliente
}

int main(int argc, char** argv)
{
    auto conn = initServer(3000); //Iniciar el server en el puerto 3000
    list<int> usersList;
    while (1) //bucle infinito
    {
        //Esperar conexión de un cliente
        while (!checkClient()) usleep(100);
        //Una vez conectado, conseguir su identificador (puede haber varios clientes)
        auto clientId = getLastClientId();
        //Crear hilo paralelo en el que se ejecuta "atiendeConexion"
        //recibe el identificador de cliente y una referencia a la lista compartida de usuarios
        thread* th = new thread(atiendeConexion, clientId, ref(usersList));
    }
    close(conn); //cerrar el servidor
    return 0;
}

```

### Parte básica obligatoria del ejercicio: (5 puntos)

El alumno deberá completar el código suministrado con las llamadas necesarias a las funciones pack/unpack/send/recvMSG de la librería libUtils que se presentó en los ejemplos de la unidad. Para ello se han dejado comentarios en varias partes, el alumno deberá interpretarlos y añadir las funciones necesarias para que se ejecute el programa correctamente. Si se completa correctamente el ejercicio, el programa tendrá una salida similar a la presentada en la siguiente imagen:

The image shows three terminal windows titled "Maquina1". The left window displays the server's log, which includes user connections, data read counts, and messages sent by users. The right window shows two client sessions. Each client logs in, sends a message to all users ("Hola a todos!!"), and then exits, resulting in a connection error message.

```

Maquina1
:~/PSDI_Dist/chat1/bin $ ./server
DatosLeidos : 10
Usuario Conectado:Carlos
DatosLeidos : 10
Usuario Conectado:Marcos
DatosLeidos : 18
Mensaje recibido:Marcos:Hola a todos!!
DatosLeidos : 17
Mensaje recibido:Carlos:Hola Marcos!!
DatosLeidos : 10
Mensaje recibido:Carlos:exit()
DatosLeidos : 10
Mensaje recibido:Marcos:exit()
[...]
Maquina1
Marcos
Hola a todos!!
DatosLeidos : 27
Mensaje recibido:Carlos dice:Hola Marcos!!
DatosLeidos : 20
Mensaje recibido:Carlos dice:exit()
exit()
DatosLeidos : 0
ERROR: recvMSG -- line : 83 lost connection
:~/PSDI_Dist/chat1/bin $ 
Maquina1
:~/PSDI_Dist/chat1/bin $ ./client
Introduzca nombre de usuario:
Carlos
DatosLeidos : 28
Mensaje recibido:Marcos dice:Hola a todos!!
Hola Marcos!!
exit()
DatosLeidos : 0
ERROR: recvMSG -- line : 83 lost connection
:~/PSDI_Dist/chat1/bin $ 

```

En la imagen se aprecian tres terminales:

- Izquierda: Terminal para el servidor. Este programa queda esperando conexiones de cada usuario. Cuando inicia una conexión, muestra el nombre de usuario y un log con los mensajes que ha escrito cada uno.
- Derecha: Terminales para los clientes (dos clientes en este caso). El programa cliente inicia pidiendo un usuario para identificarse. Cada vez que recibe un mensaje del servidor, lo muestra con el prefijo “Mensaje recibido”

#### Partes opcionales para sumar 5 puntos extra:

Se piden añadir las siguientes funcionalidades para llegar al 10 en este ejercicio:

- Solucionar error “lost connection” (2 puntos):
  - Se puede observar en las terminales de los clientes que, al terminar su ejecución escribiendo “exit()”, el programa cliente (sólo uno, el que invocó “exit()”) acaba con errores de conexión. Eso es debido a lo siguiente:
    - El bucle de la función “recibeMensajesClientes” está esperando un mensaje mientras el servidor cierra la conexión. Eso es un estado inválido, el servidor debe avisar del cierre de conexión al cliente enviándole algún tipo de mensaje antes de cerrarla. Se valorará con hasta 2 puntos extra resolver correctamente este problema. Se recuerda, está prohibido modificar las librerías “utils” y sus funcionalidades, no es suficiente con “borrar” ese mensaje.
- Implementar “mensajes privados” (3 puntos):
  - Con la implementación anterior, todo el mundo puede leer los mensajes enviados. Se pide implementar un tipo de mensaje “privado” entre dos clientes. Consejo:
    - Crear “tipos de mensajes” que se envían entre el cliente y el servidor:
      - Empaquetar el tipo de mensaje (público o privado)
      - Empaquetar el texto
      - Si es privado, empaquetar el destinatario
    - Añadir en el cliente algún comando o palabra clave para iniciar un “mensaje privado”, que pida los datos necesarios.
    - En el servidor, cambiar el tipo de lista “users” que recibe cada thread por una que permita almacenar el par de datos “nombre”, “clientId”. Así se pueden buscar los identificadores de los clientes usando su nombre.

- Con la nueva lista compartida de usuarios, si el tipo de mensaje es privado, se puede buscar el usuario y reenviarlo únicamente a él.

## RECOMENDACIONES E INDICACIONES PARA LA ENTREGA

Se puede realizar la práctica usando un único ordenador de EC2, pero se aconseja realizar la totalidad de la práctica usando al menos dos instancias (una para cada tipo de programa). Eso permitirá al alumno trabajar con un entorno en el que haya que especificar direcciones IP de forma explícita, y podrá lanzar varias instancias de los programas “cliente” para comprobar mejor su funcionalidad.

También se aconseja usar repositorios y programas de control de versiones tipo GitHub. Eso es debido a que estas instancias son volátiles (pueden desaparecer si AWS lo requiere), por lo que es una buena costumbre tener copias de la versión final del código para evitar su pérdida.

## RÚBRICA DE CORRECCIÓN

La rúbrica para corregir el ejercicio seguirá los criterios listados a continuación, que tienen distintos pesos respecto al total de la nota.

Criterios	Excelente	Satisfactorio	No satisfactorio	Insuficiente
<b>Implementación ejercicio básico obligatorio</b> <b>Máximo 5 puntos</b>	Compila y manda/recibe mensajes entre todos los clientes sin errores importantes. El código es óptimo.  (de 3.75 a 5)	El código es mínimamente funcional pero presenta errores de ejecución. Llega a enviar al menos mensajes del cliente al servidor sin errores.  (de 2.5 a 3.75)	El código llega a compilar pero no llegan mensajes entre cliente-servidor.  (de 1.25 a 2.5)	El problema no está bien resuelto. Código y funcionalidades insuficientes.  (de 0 a 0.5 puntos)
<b>Implementación Solucionar error “lost connection”</b> <b>Máximo 2 puntos</b>	Se ha solucionado el problema de la forma más óptima  (de 1.5 a 2 puntos)	Se ha solucionado el problema, pero no es la forma más óptima  (de 1 a 1.5 puntos)	Se ha intentado solucionar el problema, pero no funciona correctamente  (de 0.5 a 1 puntos)	No se han seguido ninguna de las guías indicadas  (de 0 a 0.5 puntos)
<b>Implementación “Mensajes privados”</b>	Se ha solucionado el problema de la forma más óptima.	Se ha solucionado el problema, pero no es la forma	Se ha intentado solucionar el problema, pero	No se han seguido ninguna de las guías

<b>Máximo 3 puntos</b>	(de 2.25 a 3 puntos)	más óptima (de 1.5 a 2.25 puntos)	no funciona correctamente (de 0.75 a 1,5 puntos)	indicadas (de 0 a 0.75 puntos)
------------------------	----------------------	---	---	--------------------------------------

## COMO REALIZAR LA ENTREGA

El alumno deberá entregar un archivo ZIP nombrado como “P1SISDISD\_Nombre\_Apellido.zip” y con el siguiente contenido:

- Proyecto CMAKE con la solución a los ejercicios indicados. Sólo es necesario entregar el código fuente (archivos “.h”, “.cpp” y archivo CMakeLists.txt”
- Documento (docx o pdf) con capturas de pantalla que muestren la correcta ejecución de los programas. No es necesario realizar un documento con descripciones muy complejas, sólo lo suficiente para demostrar las funcionalidades implementadas:
  - Ejecución con un servidor y dos clientes, donde se muestre que lo escrito por un cliente pasa por el servidor y es recibido por el segundo cliente.
  - Ejecución sin errores “lost connection” (en caso de haberlo implementado)
  - Ejecución con mensajes privados: En caso de haberlo implementado, se deberá mostrar las terminales del servidor y al menos tres clientes, donde se pueda ver que dos de ellos hablan de forma privada.

No se debe entregar el directorio “bin” que contiene los archivos compilados y ejecutables.