AEC2 - Mini Server

Creado	@30 de octubre de 2025 13:40
Etiquetas	
Progreso	50

Descripción General

Esta actividad consiste en desarrollar un sistema de chat básico tipo cliente-servidor con interfaz de terminal. La idea es bastante directa: varios clientes pueden conectarse a un servidor central, enviar mensajes que se retransmiten a todos los demás usuarios conectados, y recibir los mensajes que escriben otros.

Funcionalidades Implementadas

Parte Básica Obligatoria (5 puntos)

He completado toda la funcionalidad básica que se pedía en el enunciado:

Lado del Cliente

Al arrancar el programa, lo primero que hace es pedir tu nombre de usuario para identificarte en el sistema. Después de eso, puedes escribir mensajes que se envían al servidor. Para poder recibir mensajes mientras escribes, he implementado un hilo separado que está constantemente escuchando nuevos mensajes del servidor y los muestra en pantalla.

Todo el envío de datos lo he hecho usando las funciones de empaquetado y desempaquetado de la librería libUtils que nos dieron. Básicamente serializo cada mensaje con `pack()` antes de enviarlo, y al recibirlo uso `unpack()` para recuperar los datos originales.

Lado del Servidor

El servidor se queda en un bucle infinito esperando que lleguen nuevos clientes. Cada vez que se conecta uno, creo un hilo nuevo para gestionarlo de forma independiente. El servidor mantiene una lista con todos los usuarios conectados, y cuando recibe un mensaje de cualquiera de ellos, lo reenvía al resto en modo broadcast.

Como hay varios hilos tocando la misma lista de usuarios, he usado un mutex para asegurarme de que no haya problemas de concurrencia y todo funcione correctamente.

Bonus 1: Solución al Error “Lost Connection” (2 puntos)

El problema que había era evidente cuando alguien escribía `exit()` para salirse del chat. El cliente que se desconectaba terminaba mostrando un error de conexión perdida. Esto pasaba porque el hilo que recibe mensajes estaba esperando un nuevo mensaje justo cuando el servidor cerraba la conexión de golpe.

La solución que implementé fue crear un protocolo de desconexión ordenada. Ahora cuando escribes `exit()`, el cliente avisa al servidor con un mensaje especial de tipo `MSG_DISCONNECT`. El servidor recibe esto, notifica a los demás usuarios que te has ido, y lo importante: antes de cerrar

tu conexión, te envía un mensaje de confirmación. Cuando tu cliente recibe esa confirmación, sabe que puede terminar limpiamente sin errores.

Con esto conseguí que las desconexiones sean limpias y no aparezca ningún error molesto en la terminal.

Bonus 2: Implementación de Mensajes Privados (3 puntos)

Para los mensajes privados tuve que hacer varios cambios en el diseño original. Lo primero fue definir bien los tipos de mensajes que se pueden enviar: públicos, privados, de conexión y de desconexión. Creé una estructura `Message` que guarda toda la información necesaria: quién envía, qué dice, y si es privado, a quién va dirigido.

En el cliente añadí el comando `/privado <usuario> <mensaje>` para que sea fácil enviar mensajes privados. También metí un `/ayuda` por si alguien se pierde con los comandos. Cuando recibes un mensaje privado, aparece claramente marcado como `[PRIVADO]` para que no haya confusión.

La parte más interesante fue en el servidor. Tuve que cambiar la lista simple de IDs de clientes por una estructura que guarde pares de información: el ID del socket y el nombre de usuario. Así cuando llega un mensaje privado, puedo buscar el usuario destinatario por su nombre, encontrar su ID de conexión, y enviarle el mensaje solo a él. Si el usuario no existe, el servidor te avisa con un mensaje de error.

Arquitectura del Sistema

Protocolo de Comunicación

Para que el cliente y el servidor se entiendan, definí un protocolo con cuatro tipos de mensajes diferentes: mensajes públicos que van para todos, mensajes privados que van solo al destinatario, mensajes de conexión que se envían cuando alguien se une al chat, y mensajes de desconexión para avisar cuando alguien se va.

Empaquetado de Mensajes

Cada mensaje se empaqueta en un formato específico. Primero va el tipo de mensaje como un número entero, después el nombre del usuario que lo envía, luego el contenido del mensaje, y si es un mensaje privado, también incluyo al final el nombre del destinatario. Todo esto usando las funciones pack y unpack de la librería que teníamos que usar.

Concurrencia y Sincronización

El cliente tiene dos hilos funcionando en paralelo: el hilo principal que lee lo que escribes y lo envía, y otro hilo que está pendiente de los mensajes que llegan del servidor para mostrarlos. En el servidor la cosa es más compleja porque hay un hilo principal esperando nuevas conexiones, y luego un hilo independiente para cada cliente conectado.

Para que no haya problemas con varios hilos tocando la lista de usuarios a la vez, protejo todos los accesos con un mutex. También uso una variable atómica para controlar cuándo hay que parar los hilos de forma limpia.

Compilación y Ejecución

Compilar el proyecto

El proyecto usa CMake. Para compilar, hay que entrar en la carpeta build y ejecutar cmake seguido de make:

```
cd build
cmake ..
make
```

Ejecutar el servidor

Una vez compilado, el servidor se arranca directamente:

```
./server
```

Se queda escuchando en localhost puerto 3000 esperando que se conecten clientes.

Ejecutar clientes

Puedes abrir varias terminales y ejecutar el cliente en cada una:

```
./client
```

Te preguntará tu nombre y después ya puedes empezar a chatear con el resto de gente conectada.

Comandos Disponibles

Cuando estás conectado como cliente, estos son los comandos que puedes usar:

- Simplemente escribe cualquier cosa y se envía como mensaje público a todos
- `/privado <usuario> <mensaje>` para enviar un mensaje privado a alguien en concreto
- `/ayuda` para que te recuerde los comandos disponibles
- `exit()` para salir del chat limpiamente

Estructura del Proyecto

```
AEC2/
├── CMakeLists.txt        # Configuración de CMake
├── include/
│   ├── protocol.h        # Definición del protocolo y estructuras
│   └── utils.h           # Funciones de librería (libUtils)
├── src/
│   ├── client.cpp        # Implementación del cliente
│   ├── server.cpp        # Implementación del servidor
│   └── utils.cpp         # Implementación de libUtils
└── build/                # Directorio de compilación
    ├── client            # Ejecutable del cliente
    └── server            # Ejecutable del servidor
```

Detalles de Implementación

Thread Safety

Como varios hilos acceden a la misma lista de usuarios conectados, tuve que proteger todos los accesos con un mutex. Básicamente uso un lock_guard que automáticamente bloquea el mutex al entrar en un bloque de código y lo libera al salir. Así me aseguro de que no haya dos hilos tocando la lista a la vez y evito problemas de race conditions.

Gestión de Errores

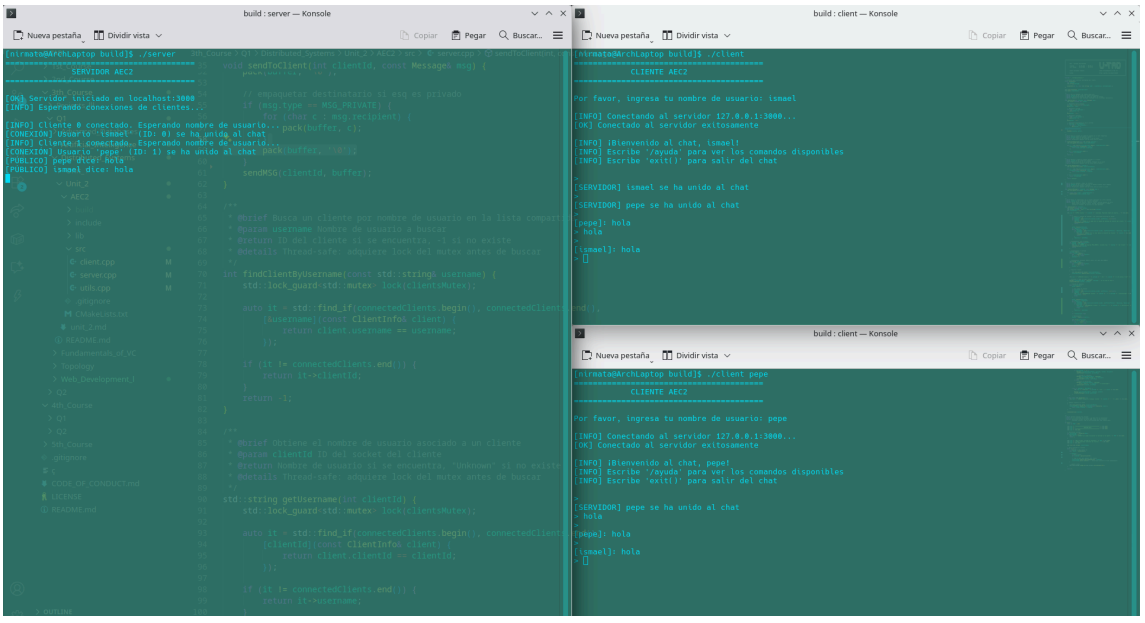
He intentado validar todo lo que puedo: que el usuario no envíe comandos mal formateados, que si intentas mandar un mensaje privado el destinatario exista, que cuando se pierda una conexión se maneje sin petar el programa, etc. El servidor y el cliente muestran mensajes informativos cuando pasa algo para que quede claro qué está ocurriendo.

Interfaz de Usuario

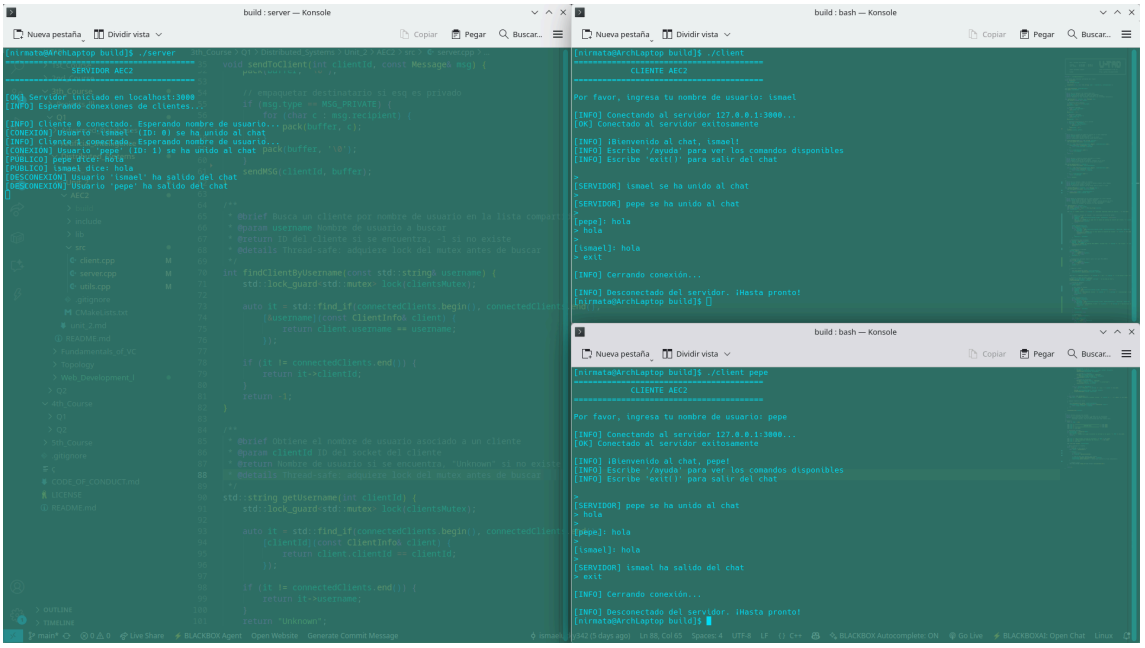
Para que sea más fácil de usar, todos los mensajes del sistema tienen prefijos claros: INFO para información general, ERROR cuando algo va mal, PRIVADO para mensajes privados, etc. También mantuve el símbolo `>` como prompt para que quede claro cuándo el programa está esperando que escribas algo.

Ejemplos de Ejecución

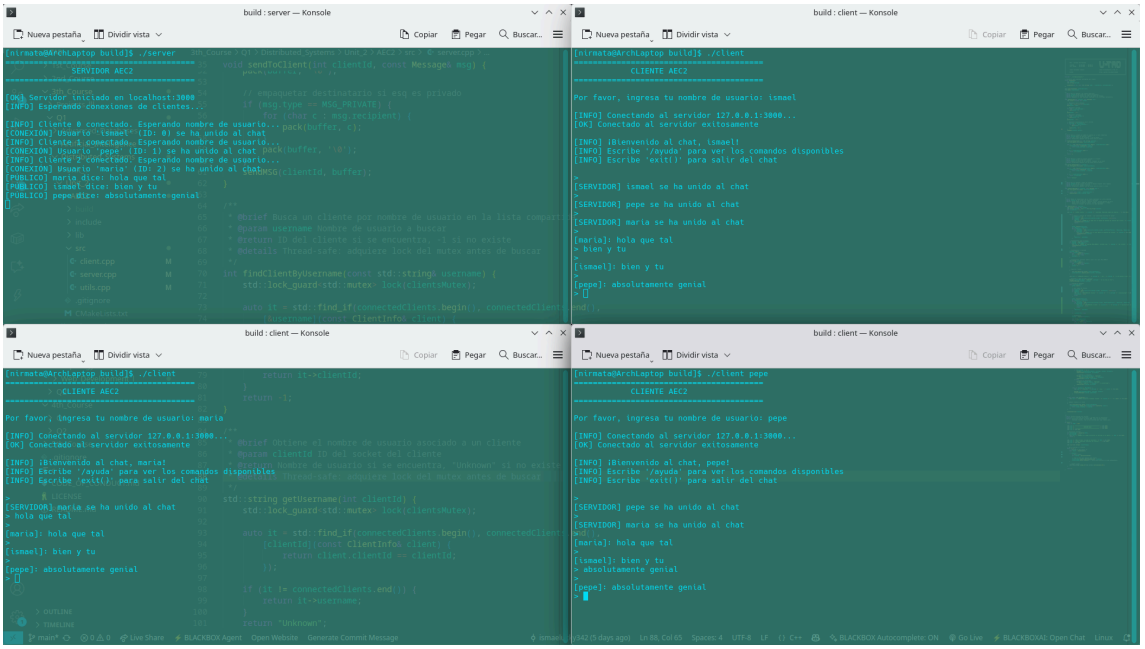
Ejemplo 1: Conexión de 2 clientes al servidor



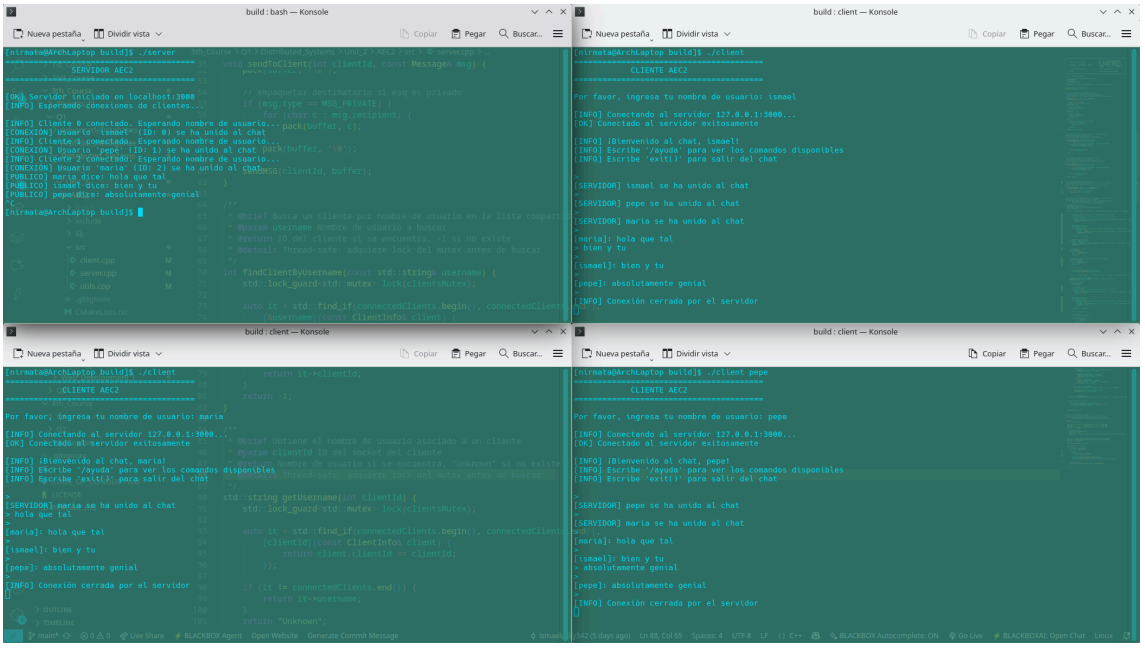
Y luego la posterior salida de ambos clientes sin problemas:



Ejemplo 2: Conexión de 3 clientes a un servidor y conversación de varios mensajes:

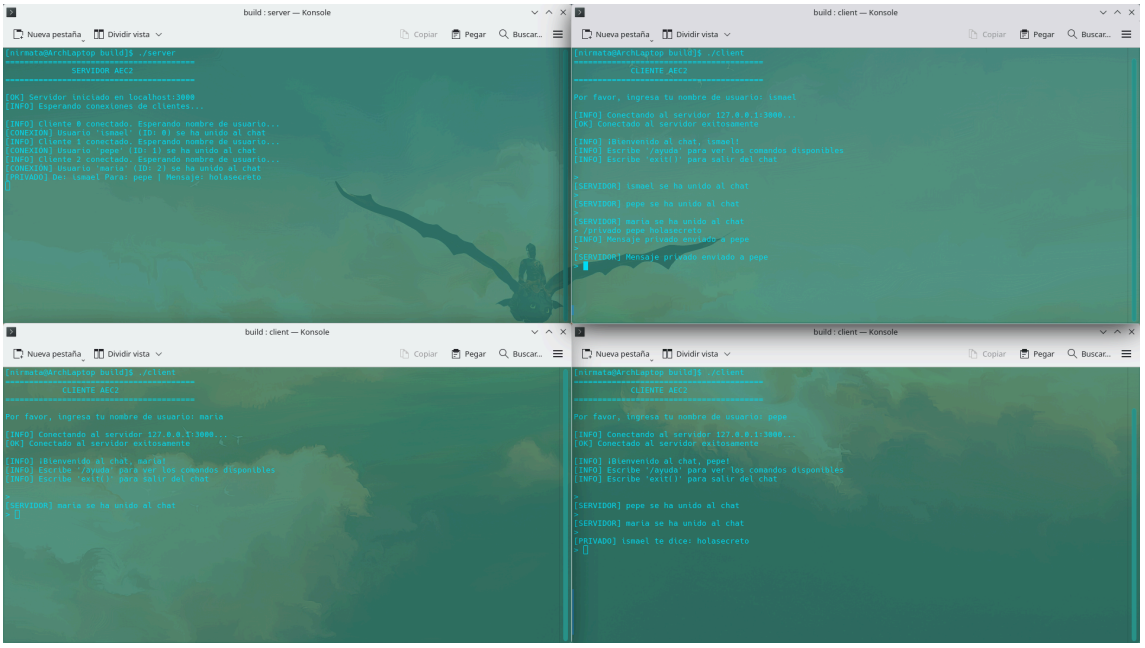


Si se cierra el servidor los clientes emiten un mensaje de desconexión de forma segura:



y viceversa si los que se desconectan son los clientes el servidor lo registra:

Ejemplo 3: Mensajes privados (bonus)



He realizado el bonus de implementar mensajes privados por lo que el programa como se ve en el ejemplo soporta una comunicación cerrada entre dos clientes sin que un tercero lo pueda ver. En el ejemplo de la imagen el cliente 1 Ismael lanza el mensaje al cliente 2 pepe y solo ellos dos y evidentemente el servidor pueden verlo, el cliente 3, maría no puede ver esta conversación.