Instituto Tecnológico de Costa Rica

Campus Tecnológico de Costa Rica

Guía Semana 2

IC4700 Lenguajes de Programación

Alonso Durán Muñoz c.2023044780

Ana Isabel Hernández Muñoz c.2023057623

Fabricio Picado Alvarado c.2022104933

Prof. Bryan Hernández Sibaja

I Semestre, 2025

Enlace de GitHub:

https://github.com/FabriBott/ProyUnoLenguajes

Pasos de la instalación del programa

Requisitos previos:

- Un sistema operativo Linux
- Un compilador con soporte para C++
- Instalar libreria nlohmann/json:
  - `(Fedora) sudo dnf install nlohmann-json-devel`
  - `(Ubuntu) sudo apt install nlohmann-json3-dev`

Clonar el repositorio:

- `git clone https://github.com/usuario/ProyUnoLenguajes.git`
- La terminal debe estar ubicada en el directorio del proyecto: `cd ProyUnoLenguajes`

Compilación del proyecto:

Cuando ya se encuentra dentro del directorio del proyecto se debe correr en la terminal el comando "`make clean`". Posterior a esto, se utiliza el comando "`make`" para generar los archivos ejecutables.

Ejecución del servidor y los clientes:

- Ejecución del servidor:

  ./server

- Ejecución de los clientes:

  ./client

Manual de usuario

Al ya tener instalado correctamente el programa, entonces puede proceder a abrir una terminal que se encuentre en el mismo directorio. Debe ejecutar los comandos de las secciones Compilación del proyecto y Ejecución del servidor y los clientes. Cuando inicia el sistema muestra un mensaje de bienvenida y le pregunta al usuario si desea iniciar sesión (1), registrarse (2) o salir (). El usuario debe ingresar su elección escribiendo un número que representa la opción deseada.

Si el usuario elige registrarse, se le solicita que introduzca un nombre de usuario y una contraseña. El sistema intenta registrar al nuevo usuario y luego le informa si el registro fue exitoso o si ocurrió un problema, como por ejemplo que el nombre de usuario ya existe. Si el usuario opta por iniciar sesión, el sistema le solicita que ingrese su nombre de usuario y contraseña. Estos datos son verificados con los usuarios almacenados en el servidor. Si las credenciales son correctas, el usuario podrá acceder al sistema. En caso contrario, el sistema mostrará un mensaje de error.

Si el puerto se encuentra ocupado, se le da la opción al usuario de cambiarlo. Si decide hacerlo, deberá digitar un nuevo puerto que se encuentre disponible. Al ya estar en la parte de mensajería del programa, el usuario decide a quién enviarle un mensaje y el contenido con el siguiente formato:

destinario: mensaje

Al enviar un mensaje exitosamente, también se le imprimen los mensajes anteriores del usuario. También, se le indicará si el usuario al que desea contactar no existe.

Arquitectura lógica utilizada

Diagrama general de la arquitectura

Explicación del funcionamiento

Lógica de client

El sistema cliente para esta aplicación de chat está compuesto principalmente por dos archivos: `client.hpp` (que contiene las declaraciones de las clases) y `client.cpp` (con las implementaciones). Juntos forman un módulo que gestiona toda la funcionalidad del lado del cliente, desde la configuración inicial hasta la comunicación en tiempo real con el servidor.

Estructura y Clases Principales

El diseño del cliente sigue un patrón de separación de responsabilidades a través de tres clases principales. La clase `ConfigManager` se encarga del la gestión de la configuración del cliente, particularmente del puerto de comunicación. Utiliza un archivo JSON para persistir estos ajustes, lo que permite mantener la configuración entre sesiones. La clase `MessageHandler` actúa como el núcleo de comunicación, manejando tanto el envío como la recepción de mensajes a través de sockets UDP. Finalmente, la clase `Client` coordina todas las operaciones, sirviendo como punto de integración entre la configuración, la comunicación y la interacción con el usuario.

Funcionamiento del Subsistema de Configuración

El `ConfigManager` implementa un mecanismo robusto para manejar la configuración del puerto. Al inicializarse, intenta cargar la configuración desde un archivo JSON en el directorio `config`. Si el archivo no existe, crea uno nuevo con valores por defecto. Este enfoque asegura que el cliente siempre tenga valores válidos para operar, incluso en la primera ejecución. La clase también proporciona métodos para modificar dinámicamente el puerto y persistir estos cambios, para cuando el puerto predeterminado está en uso.

Manejo de Comunicación y Mensajería

El `MessageHandler` implementa la lógica completa para la comunicación con el servidor. Utiliza sockets UDP para enviar y recibir mensajes en formato JSON. Durante la autenticación, construye mensajes específicos para login o registro y procesa las respuestas del servidor. Para el envío de mensajes normales, implementa un formato estandarizado que incluye remitente, destinatario y contenido. La recepción de mensajes se maneja en un bucle continuo que filtra los mensajes dirigidos específicamente al usuario actual, mostrándolos en la consola con colores distintivos para mejor legibilidad.

Flujo Principal de la Aplicación Cliente

En la clase `client` se encuentra un menú interactivo que permite al usuario elegir entre iniciar sesión, registrarse o salir. Una vez capturadas las credenciales, establece la conexión UDP y gestiona el proceso de autenticación con el servidor. Para manejar concurrentemente la entrada del usuario y la recepción de mensajes, implementa un modelo de bifurcación (fork) donde el proceso padre se encarga de la interfaz de usuario y el proceso hijo de recibir mensajes. Este diseño permite una experiencia de usuario receptiva donde se pueden enviar y recibir mensajes simultáneamente.

Manejo de Errores

El sistema puede detectar cuando el puerto está en uso y ofrece al usuario la posibilidad de seleccionar uno diferente. Gestiona adecuadamente las señales del sistema (como `SIGINT`) para limpiar recursos antes de terminar. Además, implementa validaciones básicas para las entradas del usuario, como la confirmación de contraseña durante el registro o el formato correcto para enviar mensajes. Los errores de comunicación son claramente reportados al usuario mediante mensajes de consola con codificación de colores.

Lógica del server

Estructura y Clases Principales

El módulo del servidor está compuesto por dos archivos fundamentales: server.hpp y server.cpp. En el primero se declaran las clases centrales que permiten

gestionar tanto los usuarios como los mensajes, y en el segundo se implementa la funcionalidad. Las tres clases principales son *`UserManager`*, `MessageManager` y `Server`.

- **UserManager:** Se encarga de cargar, guardar y administrar la información de los usuarios almacenados en un archivo JSON.

- **MessageManager:** Gestiona el historial de mensajes, permitiendo cargar, guardar y añadir mensajes a un archivo de datos (también en formato JSON).

- **Server:** Integra la funcionalidad del sistema, gestionando la creación del socket UDP, el enlace con el puerto del servidor y la recepción y procesamiento de distintos tipos de mensajes (login, registro y mensajes de chat).

Gestión de Usuarios

La clase `UserManager` se encarga de la administración de los usuarios. En su constructor se inicializa la lista de usuarios leyendo los datos almacenados en un archivo JSON. Si dicho archivo no existe, se crea el directorio necesario y se procede a utilizar valores predeterminados.
El método `loadUsers()` se encarga de leer el archivo JSON, parsear cada registro y almacenar los usuarios en un vector. Cuando se realizan cambios, el método `saveUsers()` recorre la lista de usuarios y genera un JSON formateado con indentación para escribirlo en el archivo, garantizando la persistencia de la información. Además, esta clase ofrece métodos para buscar un usuario, autenticarlo (comparando la contraseña) y registrar nuevos usuarios, actualizando en caso de existir ya información previa (como la IP o el puerto).

Manejo de Mensajes

El `MessageManager` se centra en la gestión del historial de mensajes. Al inicializarse, intenta cargar los mensajes desde el archivo JSON correspondiente, almacenándolos en un vector. El método `loadMessages()` se encarga de parsear cada mensaje y asegurarse de que todos tengan la consistencia de tipo "`message`".
Por otra parte, `saveMessages()` recorre el vector de mensajes y, asignando un identificador secuencial a cada uno, genera un JSON legible que se guarda en el archivo. El método `addMessage()` permite agregar un nuevo mensaje al historial y guardar inmediatamente los cambios. De este modo, se preserva el historial de conversaciones de forma persistente entre ejecuciones del servidor.

Flujo Principal del Servidor

La clase `Server` coordina la interacción entre el manejo de usuarios y la gestión de mensajes. En su método `init()`, el servidor crea un socket UDP, configura la dirección (aceptando conexiones en todas las interfaces mediante `INADDR_ANY`) y enlaza el socket al puerto definido para el servidor. Si el enlace es exitoso, se muestra un mensaje indicando que el servidor se ha iniciado correctamente.

El método `run()` entra en un bucle infinito en el que se reciben datos a través del socket. Cada bloque de datos recibido se interpreta como un objeto JSON y, según el valor de la propiedad "type", se redirige a la función correspondiente:

- **processLogin():** Se encarga de procesar las solicitudes de inicio de sesión, verificando la existencia del usuario y la validez de la contraseña, y actualizando la información de conexión (IP y puerto) si la autenticación es exitosa.

- **processRegistration():** Se utiliza para registrar nuevos usuarios, comprobando primero si el nombre de usuario ya existe y, de no ser así, creando el nuevo registro.

- **processMessage():** Gestiona el envío de mensajes entre usuarios. Para ello, verifica la existencia del destinatario, actualiza las listas de contactos de ambos usuarios en caso de nuevo intercambio, guarda el mensaje en el historial y lo reenvía al cliente destinatario utilizando su IP y puerto.

El servidor no necesita usar fork() porque tiene un modelo de operación diferente. Utiliza un socket UDP y maneja todas las solicitudes entrantes secuencialmente en un solo bucle (while (true) en el método run()). Como UDP es un protocolo sin conexión, el servidor puede procesar cada solicitud a medida que llega y no necesita mantener múltiples conexiones simultáneas como lo haría un servidor TCP.

Manejo de Errores y Validación

El módulo de servidor incorpora varios mecanismos para manejar errores y validar datos. Durante la carga de usuarios y mensajes, se capturan excepciones derivadas del parseo de JSON, lo que permite reportar y corregir errores de lectura.

En el proceso de creación del socket y el enlace a un puerto, se verifica el éxito de cada operación, y en caso de fallo se informa mediante mensajes de error en la consola y se cierra el socket si es necesario.

Finalmente, en el procesamiento de mensajes, se realizan comprobaciones para asegurar que tanto la IP como el puerto del usuario destinatario sean válidos antes de intentar enviar el mensaje, evitando así errores durante la comunicación.

common.hpp

Inclusión de Librerías Estándar y Externas

El archivo incluye varias bibliotecas estándar de C++ (como `<string>`, `<vector>`, `<iostream>`, `<fstream>` y `<cstring>`) que proporcionan funcionalidades esenciales para el manejo de cadenas, vectores, entrada/salida, manejo de archivos y funciones de manipulación de memoria. Además, se incluyen cabeceras específicas para programación de sockets y redes en sistemas Unix, tales como `<unistd.h>`, `<arpa/inet.h>` y `<sys/socket.h>`, lo que permite gestionar la comunicación a través de la red. La inclusión de la biblioteca `nlohmann/json.hpp` permite trabajar con objetos JSON de manera sencilla, facilitando la serialización y deserialización de datos para la comunicación entre cliente y servidor.

Definición de Constantes y Macros

El archivo define varias macros que se utilizan a lo largo del programa para mantener consistencia y facilitar su mantenimiento:

- **`MAX_BUFFER_SIZE`:** Define el tamaño máximo del buffer para la recepción y envío de datos, en este caso 1024 bytes.

- **`SERVER_PORT`** y **`DEFAULT_CLIENT_PORT`:** Especifican los puertos predeterminados para el servidor (7777) y para el cliente (8888), permitiendo configurar la comunicación de red.

- **`SERVER_IP`:** Define la dirección IP predeterminada del servidor ("127.0.0.1"), que se utiliza para conexiones locales durante las pruebas.

Definición de Colores para la Consola

Se establecen una serie de macros que permiten aplicar colores a la salida de texto en la consola. Estas definiciones (por ejemplo, **`COLOR_RED`**, **`COLOR_GREEN`**, **`COLOR_YELLOW`**, **`COLOR_BLUE`**, entre otras) utilizan secuencias de escape ANSI para colorear la salida. Esto es útil para resaltar mensajes de error, éxito o información en la terminal y mejora la legibilidad y experiencia de usuario durante la ejecución del programa.

message.hpp

Este archivo define la clase `Message`, que representa la estructura de los mensajes intercambiados entre el cliente y el servidor. Los mensajes pueden contener distintos tipos de información según su propósito (registro, envío de texto, respuestas del servidor, etc.), y son serializados/deserializados usando JSON para facilitar la comunicación en red.

Atributos principales

- **type**: Cadena que indica el tipo de mensaje.
- **sender**: Nombre del remitente del mensaje.
- **receiver**: Nombre del destinatario del mensaje.
- **content**: Contenido textual del mensaje. Puede ser un mensaje entre usuarios o una notificación del servidor.
- **timestamp**: Marca de tiempo que indica cuándo fue creado el mensaje. Se almacena como `time_t`, generado automáticamente con time(nullptr) en el constructor por defecto.
- **delivered**: Bandera booleana que indica si el mensaje fue entregado con éxito.
- **status**: Campo opcional que almacena información sobre el estado del mensaje. Se incluye principalmente en mensajes de respuesta del servidor.

Comportamiento y métodos

1. Constructor por defecto

Inicializa automáticamente la marca de tiempo con la hora actual y marca el mensaje como no entregado (delivered = false).

2. Método toJson()

Convierte un objeto `Message` a un objeto JSON serializable, ideal para enviarlo por red.

- Los campos básicos se incluyen siempre.
- El campo status se agrega solo si no está vacío, para evitar información innecesaria.

3. Método estático fromJson()

Permite reconstruir un objeto Message a partir de un JSON recibido (por ejemplo, desde el socket).

- Se verifica la existencia de cada campo antes de asignarlo, evitando errores en caso de mensajes incompletos o mal formateados.

4. Método estático createResponse()

Genera un mensaje tipo respuesta, útil cuando el servidor quiere enviar confirmaciones o errores al cliente.

- Se establece el type, el content, el status, y la marca de tiempo.
- No requiere especificar sender o receiver, ya que es una respuesta genérica.

user.hpp

Este archivo define la clase `User`, la cual representa a cada usuario del sistema de mensajería. Contiene la información necesaria para identificar al usuario, validar su autenticidad y gestionar sus contactos.

Atributos principales

- **`username:`** Nombre único del usuario. Es utilizado como identificador principal en el sistema.
- **`password:`** Contraseña del usuario. Se usa para autenticación. (Nota: en un entorno de producción real, esta debería almacenarse de forma cifrada, pero en este proyecto educativo se almacena en texto plano para facilitar el desarrollo).
- **`ip:`** Dirección IP actual del usuario. Se registra al momento del registro o inicio de sesión.
- **`port:`** Número de puerto que el usuario utiliza para recibir mensajes. Se establece por defecto a DEFAULT_CLIENT_PORT si no se especifica.
- **`contacts:`** Lista de nombres de otros usuarios con los que este usuario ha establecido contacto. Esto permite implementar funcionalidades como historial de conversaciones o mensajes frecuentes.

Comportamientos y métodos

1. Constructores

- El constructor por defecto inicializa el puerto con el valor por defecto definido globalmente.
- El constructor con parámetros permite inicializar directamente los datos del usuario al momento de su creación, incluyendo el nombre, contraseña, IP y

puerto.

2. Método `toJson()`

Este método convierte el objeto User en un objeto JSON, permitiendo:

- Guardar la información del usuario en disco (persistencia).
- Transmitir los datos por red si fuese necesario.

Incluye todos los atributos, incluyendo la lista de contactos convertida en un arreglo JSON.

3. Método estático `fromJson()`

Permite reconstruir un objeto User desde un archivo o mensaje JSON. Se usa principalmente al cargar usuarios desde archivos durante el arranque del servidor.

- Utiliza value("password", "") para manejar casos en los que el archivo JSON no incluya el campo password (por ejemplo, versiones anteriores del sistema).
- También reconstruye la lista de contactos si está presente en el JSON.

4. Método `checkCredentials()`

Este método compara el nombre de usuario y la contraseña proporcionados contra los almacenados. Es útil durante el proceso de autenticación para validar el acceso de un cliente.

Almacenamiento de Datos

Para este proyecto se utilizaron dos archivos formato json para el almacenamiento de datos de los usuarios (credenciales) y sus mensajes. Se muestra el formato exacto para cada uno.

Formato messages.json

```
{
    {
        "content": "mensaje de texto",
        "delivered": false,
```

```
            "id": 1,
            "receiver": "receptor",
            "sender": "emisor",
            "timestamp": 1742531567,
            "type": "message"
        }
    }
```


Formato users.json


```
    {
        {
            "contacts": [],
            "ip": "127.0.0.1",
            "password": "hola",
            "port": 8888,
            "username": "nombreDeUsuario"
        }
    }
```