UNIVERSIDAD DEL VALLE DE GUATEMALA

Sistemas Operativos

Sección 21

Catedrática: Sebastían Galindo

Informe Proyecto

Chat

Abner Iván García Alegría - 21285

Oscar Esteban Donis Martínez - 21610

Dariel Eduardo Villatoro - 20776

Índice

Resumen

Este trabajo presenta la implementación de un sistema de chat en C++ utilizando sockets y multithreading. Se exploraron las bibliotecas y tecnologías necesarias para establecer una comunicación eficiente y concurrente entre múltiples usuarios a través de una red. Las direcciones IP y los sockets se utilizan para la comunicación en red, mientras que el multithreading permite manejar múltiples conexiones de clientes simultáneamente. Protobuf se emplea para la serialización de mensajes, garantizando una transmisión de datos estructurada eficiente y compatible. Se concluye que la combinación de estas tecnologías proporciona una base sólida para la construcción de aplicaciones de red robustas y eficientes.

Introducción

En la revolución tecnológica que estamos viviendo hoy en día, podemos observar como la comunicación ha mejorado increíblemente, permitiéndonos comunicarnos con cualquier persona, en cualquier parte del mundo, en cuestión de segundos. Sin embargo, también estamos más ocupados que nunca, por lo que mandar mensajes se ha convertido en la vía de comunicación preferida por las personas a nuestro alrededor. +

El presente trabajo aborda la implementación de un sistema de chat basado en sockets y multithreading en C++. Este sistema permite la comunicación en tiempo real entre múltiples usuarios a través de una red. La implementación se apoya en diversas bibliotecas y tecnologías, incluyendo Protocol Buffers (Protobuf) para la serialización de datos, y el uso de hilos para manejar múltiples conexiones de clientes simultáneamente. Se detallan tanto los aspectos teóricos de los componentes involucrados, como las técnicas prácticas empleadas para el desarrollo del sistema de chat.

Marco Teórico

Sockets

Los sockets son canales de comunicación que permiten que procesos no relacionados intercambien datos localmente y entre redes. Un único socket es el punto final de un canal de comunicación bidireccional. (AIX 7.3, 2023)

IP Address

Una dirección IP es una dirección única que identifica a un dispositivo en Internet o en una red local. IP significa “protocolo de Internet”, que es el conjunto de reglas que rigen el formato de los datos enviados a través de Internet o la red local.

En esencia, las direcciones IP son el identificador que permite el envío de información entre dispositivos en una red. Contienen información de la ubicación y brindan a los dispositivos acceso de comunicación. Internet necesita una forma de diferenciar entre distintas computadoras, enrutadores y sitios web. Las direcciones IP proporcionan una forma de hacerlo y forman una parte esencial de cómo funciona Internet. (Kaspersky, 2024)

Multithreading

Para aumentar la velocidad del núcleo del procesador sin tener que cambiar la frecuencia del reloj, el multithreading permite a la CPU procesar varias tareas simultáneamente. Siendo más precisos: se procesan varios hilos al mismo tiempo. Un hilo puede entenderse como una hebra de un proceso. Los programas pueden dividirse en procesos y éstos, a su vez, en hilos individuales. Cada proceso consta de al menos un hilo. (de, 2021)

Funcionamiento Sockets C/C++

Los sockets son simplemente un "puente", como hemos dicho antes, hacen la función de comunicarnos con otro ordenador. Para que ésto ocurra deben existir dos nodos: **Cliente** y **Servidor**. Para que ambos se comuniquen deben de enviarse información entre ellos para determinar con quién van a hablar. Un ejemplo práctico es el correo (no electrónico): Si quiere enviar una carta debe saber el destino donde va a enviar la carta, cuál es la ciudad donde vive el destinatario, su dirección, nombre, etc... Acto seguido, el remitente (cliente) al enviar la carta (mensaje) a una administrador de correo (servidor), éste leerá los datos y los enviará a su destino (otro cliente). Este es el caso más particular de intercambio de datos que existe. También podemos enviar datos directamente al servidor, y que el servidor nos conteste. (Programación En C/Sockets - Wikilibros, 2022)

Materiales y Métodos

Lenguaje Utilizado

C++: es un ejemplo de lenguaje de programación compilado, multiparadigma, principalmente de tipo imperativo y orientado a objetos, incluyendo también programación genérica y funcional, características estas últimas que comentaremos más adelante en el curso.

Librerías

Protobuf: Protocol Buffers, comúnmente conocido como Protobuf, es un formato de serialización binaria independiente del lenguaje desarrollado por Google. Está diseñado para serializar de manera eficiente datos estructurados y permitir la comunicación entre diferentes sistemas y plataformas. Protobuf proporciona una forma neutral de lenguaje y plataforma para definir la estructura de los datos utilizando un lenguaje de esquema

específico, y luego genera código en varios lenguajes de programación para serializar y deserializar los datos.

Abseil: Es una biblioteca de código abierto desarrollada por Google que proporciona una colección de componentes y utilidades de C++ para facilitar el desarrollo de software. Abseil se centra en mejorar la productividad y la calidad del código al proporcionar abstracciones de alto nivel, algoritmos comunes, manejo de errores, manejo de cadenas, estructuras de datos, entre otros.

<iostream>: Es una biblioteca estándar de C++ que proporciona las clases y funciones necesarias para realizar operaciones de entrada y salida de datos en la consola.

<cstring>: Es una biblioteca estándar de C++ que proporciona funciones y utilidades para manipular cadenas de caracteres (cadenas de texto). Esta biblioteca es parte del núcleo del lenguaje C++ y ofrece operaciones para trabajar con cadenas de caracteres de forma eficiente.

<sys/socket.h>: Es un archivo encabezado en el lenguaje de programación C que forma parte de la biblioteca de funciones del sistema operativo POSIX. Este archivo proporciona las definiciones y funciones necesarias para trabajar con sockets en aplicaciones de red.

<arpa/inet.h>: Es un archivo de encabezado en el lenguaje de programación C que forma parte de la biblioteca de funciones del sistema operativo POSIX. Este archivo proporciona las definiciones y funciones necesarias para la manipulación y conversión de direcciones IP en formato de red. Incluye funciones para convertir direcciones IP entre formatos de texto y binario, así como para realizar otras operaciones relacionadas con las direcciones IP.

<unistd.h>: es un archivo encabezado en el lenguaje de programación C que forma parte de la biblioteca de funciones del sistema operativo POSIX. Este archivo proporciona definiciones y funciones relacionadas con llamadas al sistema y operaciones de bajo nivel en el entorno de ejecución del programa.

<string>: Es un archivo encabezado en el lenguaje de programación C++ que forma parte de la biblioteca estándar del lenguaje. Este archivo proporciona la clase std::string y otras funciones y utilidades relacionadas con la manipulación de cadenas de caracteres (cadenas de texto). Biblioteca para manipulación de cadenas C++.

<unordered_map>: Es un archivo de encabezado en el lenguaje de programación C++ que forma parte de la biblioteca estándar del lenguaje. Este archivo proporciona la plantilla de clase std::unordered_map y otras funcionalidades relacionadas con la implementación de mapas hash (tablas hash) en C++.

<unordered_set>: Es un archivo de encabezado en el lenguaje de programación C++ que forma parte de la biblioteca estándar del lenguaje. Este archivo proporciona la plantilla de clase std::unordered_set y otras funcionalidades relacionadas con la implementación de conjuntos hash (tablas hash) en C++.

<thread>: Es un archivo encabezado en el lenguaje de programación C++ que forma parte de la biblioteca estándar del lenguaje. Este archivo proporciona funcionalidades relacionadas con la creación y administración de hilos de ejecución en C++.

<vector>: Es un archivo encabezado en el lenguaje de programación C++ que forma parte de la biblioteca estándar del lenguaje. Este archivo proporciona la plantilla de clase std::vector y otras funcionalidades relacionadas con la implementación de arreglos dinámicos en C++.

<atomic>: es un archivo encabezado en el lenguaje de programación C++ que forma parte de la biblioteca estándar del lenguaje. Este archivo proporciona tipos y operaciones para la programación concurrente y la manipulación de variables atómicas en C++.

<deque>: Es un archivo encabezado en el lenguaje de programación C++ que forma parte de la biblioteca estándar del lenguaje. Este archivo proporciona la plantilla de clase std::deque y otras funcionalidades relacionadas con la implementación de colas doblemente terminadas (deque) en C++.

<mutex>: Es un archivo encabezado en el lenguaje de programación C++ que forma parte de la biblioteca estándar del lenguaje. Este archivo proporciona tipos y funciones relacionadas con la sincronización de hilos mediante el uso de mutex (mutual exclusions) en C++.

<condition_variable>: Es un archivo de encabezado en el lenguaje de programación C++ que forma parte de la biblioteca estándar del lenguaje. Este archivo proporciona la clase std::condition_variable y funciones relacionadas para la sincronización de hilos utilizando variables de condición en C++.

<sstream>: Es un archivo de encabezado en el lenguaje de programación C++ que forma parte de la biblioteca estándar del lenguaje. Este archivo proporciona clases y funciones relacionadas con la manipulación de cadenas y flujos de datos en C++.

Protocolo

```
syntax = "proto3";
package chat;

// Enumeration of potential user statuses to clearly define possible states a user can have.
// This state is not functional but represent the user's availability to receive messages.
enum UserStatus {
   ONLINE = 0;  // The user is online and available to receive messages.
   BUSY = 1;    // The user is online but marked as busy, may not respond promptly.
   OFFLINE = 2; // The user is offline and cannot receive messages.
}

// User represents the essential information about a chat user.
message User {
   string username = 1;  // Unique identifier for the user.
```

```
    UserStatus status = 2;  // Current status of the user, indicating availability.
}

// NewUserRequest is used to register a new user on the chat server.
message NewUserRequest {
    string username = 1;  // Desired username for the new user. Must be unique across all
users.
}

// MessageRequest represents a request to send a chat message.
message SendMessageRequest {
    string recipient = 1;  // Username of the recipient. If empty, the message is broadcast to all
online users.
    string content = 2;  // Content of the message being sent.
}

enum MessageType {
    BROADCAST = 0;  // Message is broadcast to all online users.
    DIRECT = 1;  // Message is sent to a specific user.
}

message IncomingMessageResponse {
    string sender = 1;  // Username of the user who sent the message.
    string content = 2;  // Content of the message.
    // Type of message
    MessageType type = 3;
}

enum UserListType {
    ALL = 0;  // Fetch all connected users.
    SINGLE = 1;  // Fetch details for a single user.
}

// UserListRequest is used to fetch a list of currently connected users.
message UserListRequest {
    string username = 1;  // Specific username to fetch details for. If empty, fetches all
connected users.
}

// UserListResponse returns a list of users.
message UserListResponse {
    repeated User users = 1;  // List of users meeting the criteria specified in UserListRequest.
    UserListType type = 2;
}

// UpdateStatusRequest is used to change the status of a user.
message UpdateStatusRequest {
    string username = 1;  // Username of the user whose status is to be updated.
```

```
    UserStatus new_status = 2;  // The new status to be applied to the user.
}

enum Operation {
    REGISTER_USER = 0;
    SEND_MESSAGE = 1;
    UPDATE_STATUS = 2;
    GET_USERS = 3;
    UNREGISTER_USER = 4;
    INCOMING_MESSAGE = 5;
}

// Request types consolidated into a unified structure with a type indicator.
message Request {
    // Indicates the type of request being made.
    Operation operation = 1;

    // Different request payloads, only one of these is filled based on 'type'.
    oneof payload {
        NewUserRequest register_user = 2;
        SendMessageRequest send_message = 3;
        UpdateStatusRequest update_status = 4;
        UserListRequest get_users = 5;
        User unregister_user = 6;
    }
}

enum StatusCode {
    UNKNOWN_STATUS = 0;             // Default value, should not be used in normal
operations
    OK = 200;                       // Request has succeeded
    BAD_REQUEST = 400;              // Request cannot be fulfilled due to bad syntax (este
podría ser el utilizado general)
    INTERNAL_SERVER_ERROR = 500;    // A generic error message, given when no more
specific message is suitable
}


// Response is a generalized structure used for all responses from the server.
message Response {
    Operation operation = 1;  // Indicates the type of operation being performed.
    StatusCode status_code = 2;  // Status code indicating the outcome of the operation.
    string message = 3;  // Human-readable (We XD) message providing more details about
the result.
    oneof result {
        UserListResponse user_list = 4;  // Details specific to user list requests.
        IncomingMessageResponse incoming_message = 5;  // Details specific to incoming
chat messages.
```

```
    }
}
```

Métodos

Client

Variables

std::deque<std::string> messages: Cola para ir guardando los mensajes broadcast
std::atomic<bool> receivingResponse{true}: Variable que detemina si se sigue recibiendo responses del servidor
std::unordered_map<std::string, std::deque<std::string>> privateMessages: Mapa para guardar los mensajes privados
std::mutex messagesMutex: Mutex para evitar problemas de concurrencia en la cola de mensajes
std::string currentStatus: Variable para guardar el status actual del usuario
std::string tempUserStatus: Variable para almacenar el nuevo estado por cambiar
std::string tempMessage: Variable para almacenar el mensaje por guardar
std::string tempRecipient: Variable que guarda el recibidor de nuestro mensaje

Funciones

void messageReceiver: Escucha los responses del servidor
@param clientSocket Interger que posee el socket utilizado por nuestro cliente.

void BroadcastMessagesPrinter: Imprime los mensajes de broadcast

void DirectMessagesPrinter: Imprime los mensajes directos

void unregisterUser: Envia mensaje al servidor para desregistrar al usuario
@param clientSocket Interger que posee el socket utilizado por nuestro cliente.
@param userName variable tipo string que posee el nombre de usuario de nuestro cliente.

void sendMessageBroadcast: Envia mensaje al servidor
@param clientSocket Interger que posee el socket utilizado por nuestro cliente.
@param message Mensaje a enviar por canal broadcast.

void sendMessageDirect: Envia mensaje al servidor el request del mensaje directo

@param clientSocket Interger que posee el socket utilizado por nuestro cliente.
@param userName Username de nuestro cliente.

void requestUsersList: Hace el request al servidor para obtener la lista de usuarios
@param clientSocket Interger que posee el socket utilizado por nuestro cliente.

void requestUserInfo: Hace el request al servidor para obtener la información de un usuario en particular
@param clientSocket Interger que posee el socket utilizado por nuestro cliente.

void changeStatus: Cambia el status del usuario
@param clientSocket Interger que posee el socket utilizado por nuestro cliente.
@param status Entero que posee el status del usuario.

void printHelp: Imprime la explicacion de las diferentes opciones del cliente

int main: Función principal del cliente.
@param argc Cantidad de argumentos de la línea de comandos.
@param argv Argumentos de la línea de comandos.

Server

Variables

std::vector<std::thread> clientThreads: Vector donde se almacenan los threads de los clientes
std::deque<std::string> messagesBroadcast: Variable donde almacenamos todos los mensajes en broadcast
std::unordered_map<std::string, chat::UserStatus> usersState: Variable donde almacenamos el status de los usuarios
std::unordered_map<std::string, int> userSockets: Variable donde almacenamos los sockets de los usuarios
std::unordered_map<std::string, std::string> ipsUsers: Variable donde almacenamos las ips de los usuarios
std::mutex clientsMutex: Mutex para proteger las variables compartidas
std::unordered_map<std::string, int> usersTiming: Variable donde almacenamos el tiempo de inactividad de los usuarios
int waitTime: Variable donde se guarda el tiempo de inactividad predeterminado

Funciones

void broadcastMessage: Funcion que maneja el envío de mensajes broadcast a través de un socket
@param message Mensaje a enviar en broadcast
@param userSender Usuario que envía el mensaje

void directMessage: Función que maneja el envío de mensajes directos a través de un socket
@param message Mensaje a enviar en directo
@param userSender Usuario que envía el mensaje
@param recipient Usuario que recibe el mensaje

void returnAllUsers: Funcion que envia todos los usuarios conectados
@param clientSocket Socket del cliente a enviar los usuarios

void returnUserInfo: Funcion que envia la informacion de un usuario en especifico
@param clientSocket Socket del cliente a enviar la informacion del usuario
@param username Nombre del usuario que se pidio la informacion

void changeStatus: Maneja el cambio de estado de un usuario
@param clientSocket Socket del cliente a cambiar el estado
@param status Nuevo estado del usuario

void userScanner: Función que maneja el tiempo de inactividad de los usuarios

void handleClient: Maneja los requests de los clientes
@param clientSocket Socket del cliente
@param clientIp IP del cliente

int main: Funcion principal del servidor
@param argc Cantidad de argumentos
@param argv Argumentos

Message

Variables

constexpr size_t BufferSize: Tamaño del buffer donde se guarda el mensaje

Funciones

bool sendMessage: Función para manejar el envío de mensajes entre el servidor y el cliente
@param socket Socket a donde se enviar el mensaje
@param message Mensaje a enviar

bool receiveMessage: Función para manejar el recibir de mensajes entre el servidor y el cliente
@param socket Socket a donde se enviar el mensaje
@param message Mensaje a recibir

Discusión

Los sockets son fundamentales para la comunicación en red, actuando como puntos finales de un canal de comunicación bidireccional. Permiten que procesos no relacionados intercambien datos, ya sea localmente o a través de una red. En la implementación, los sockets facilitan la comunicación entre el cliente y el servidor, estableciendo una conexión mediante la cual se intercambian mensajes.

Las direcciones IP juegan un papel crucial al identificar de manera única a cada dispositivo en la red. Este identificador es esencial para dirigir los datos hacia el destinatario correcto. La correcta manipulación y conversión de estas direcciones son manejadas a través de las bibliotecas <arpa/inet.h> y <unistd.h> en C++.

El multithreading mejora significativamente la eficiencia del servidor, permitiendo la gestión concurrente de múltiples conexiones de clientes. Cada cliente es manejado por un hilo separado, asegurando que el servidor pueda atender varias solicitudes simultáneamente sin que una conexión bloquee a las demás. La biblioteca <thread> en C++ se utiliza para la creación y administración de estos hilos, mientras que <mutex> y <condition_variable> aseguran la correcta sincronización entre ellos.

La estructura de datos y la lógica de comunicación del sistema de chat se definieron utilizando Protocol Buffers (Protobuf), que proporciona un formato eficiente para la serialización de mensajes. Este enfoque garantiza la compatibilidad y eficiencia en la transmisión de datos estructurados entre el cliente y el servidor.

Conclusiones

El desarrollo de un sistema de chat basado en sockets y multithreading en C++ demuestra la efectividad de estas tecnologías para construir aplicaciones de red robustas y eficientes. Los sockets permiten la comunicación entre procesos distribuidos, mientras que el multithreading asegura que el servidor pueda manejar múltiples conexiones de manera

simultánea y eficiente. La utilización de Protobuf facilita la serialización de datos, mejorando la interoperabilidad y eficiencia del sistema.

Recomendaciones

Para futuras mejoras, se recomienda:

1. Seguridad: Implementar medidas de seguridad adicionales, como cifrado de datos y autenticación de usuarios, para proteger la información transmitida.
2. Escalabilidad: Evaluar y optimizar el sistema para manejar un mayor número de conexiones concurrentes, posiblemente mediante técnicas de balanceo de carga y servidores distribuidos.
3. Interface de usuario: Desarrollar una interfaz gráfica de usuario (GUI) para mejorar la experiencia del usuario final, facilitando la interacción con el sistema de chat.
4. Mantenimiento: Establecer pruebas unitarias y de integración automatizadas para asegurar la estabilidad del sistema con cada actualización o cambio en el código.

Referencias:

AIX 7.3. (2023, March 24). Ibm.com. https://www.ibm.com/docs/es/aix/7.3?topic=concepts-sockets

Kaspersky. (2024, May 24). Qué es una dirección IP. Latam.kaspersky.com. https://latam.kaspersky.com/resource-center/definitions/what-is-an-ip-address

de. (2021, September 13). Multithreading: más potencia para los procesadores. IONOS Digital Guide; IONOS. https://www.ionos.com/es-us/digitalguide/servidores/know-how/explicacion-del-multithreading/

Programación en C/Sockets - Wikilibros. (2022). Wikibooks.org. https://es.wikibooks.org/wiki/Programaci%C3%B3n_en_C/Sockets

Mensajes entre Sockets. (2024). Chuidiang.org. https://old.chuidiang.org/clinux/sockets/mensajes.php