

## **Semana 3 Actividad Colaborativa - Chat bidireccional usando sockets**

Andres Felipe Zapata Diaz

Politécnico Gran Colombiano

B02: Persistencia y Datos transaccionales

Oscar Campos Porras

24 de mayo de 2025

# Índice

<b>Introducción</b>	<b>1</b>
<b>Objetivos</b>	<b>2</b>
Objetivo general . . . . .	2
Objetivos específicos . . . . .	2
<b>Marco Teórico</b>	<b>2</b>
Capa de Transporte . . . . .	2
UDP . . . . .	3
TCP . . . . .	3
Socket . . . . .	5
<b>Arquitectura de Red y Verificación</b>	<b>5</b>
Diseño de la Arquitectura de Red . . . . .	6
Verificación . . . . .	6
Secuencia de Comunicación . . . . .	7
Verificación . . . . .	9
<b>Implementación</b>	<b>10</b>
1. Creación y Configuración del Socket del Servidor . . . . .	11
2. Bind y Listen del Servidor . . . . .	12
3. Aceptación de Conexiones de Clientes . . . . .	12
4. Creación del Socket del Cliente . . . . .	14
5. Establecimiento de Conexión del Cliente . . . . .	15
6. Envío y Recepción de Datos . . . . .	16
7. Manejo de Hilos para Concurrencia . . . . .	18
8. Cierre y Limpieza de Sockets . . . . .	20

<b>Video</b>	<b>22</b>
<b>Código</b>	<b>22</b>
<b>Conclusiones</b>	<b>22</b>
<b>Referencias</b>	<b>24</b>

## Índice de figuras

1.	Proceso de retransmisión TCP mostrando el mecanismo de acknowledgment y reenvío automático de paquetes perdidos para garantizar la confiabilidad en la entrega. . . . .	4
2.	Arquitectura de red del chat bidireccional con servidor TCP en puerto 8888 y múltiples clientes conectados mediante sockets. . . . .	6
3.	Salida del comando netstat -tan   grep 8888 mostrando el servidor en estado LISTEN y dos conexiones TCP establecidas con clientes en puertos dinámicos. . . . .	7
4.	Diagrama de secuencia del chat bidireccional mostrando el handshake TCP, intercambio de username y comunicación entre cliente y servidor. . . . .	8
5.	Logs del servidor mostrando el inicio del servicio, conexiones de cuatro clientes, intercambio de mensajes y proceso de desconexión. . . . .	9
6.	Interfaz de dos clientes mostrando la conexión, intercambio de mensajes, identificación de mensajes propios con "(you)" proceso de desconexión. . . . .	10

## Introducción

La programación con sockets constituye uno de los fundamentos más importantes en el desarrollo de aplicaciones de red modernas, permitiendo la comunicación entre procesos distribuidos a través de diferentes sistemas computacionales. Los sockets actúan como una interfaz de programación que abstrae la complejidad de los protocolos de red subyacentes, facilitando el intercambio de datos entre aplicaciones mediante el uso de la capa de transporte del modelo OS.

El protocolo de control de transmisión TCP representa la base para aplicaciones que requieren comunicación confiable y ordenada, garantizando la integridad de los datos mediante mecanismos de control de flujo, detección de errores y retransmisión automática. Esta confiabilidad resulta esencial en aplicaciones de tiempo real como sistemas de chat, donde la pérdida o el desorden de mensajes compromete significativamente la experiencia del usuario.

El presente trabajo desarrolla e implementa un sistema de chat bidireccional utilizando sockets TCP en lenguaje C, demostrando los principios teóricos de la programación de red a través de una aplicación práctica. La implementación emplea una arquitectura cliente-servidor que permite la conexión simultánea de múltiples usuarios, utilizando programación concurrente mediante hilos POSIX para gestionar las comunicaciones paralelas. El documento presenta inicialmente el marco teórico que fundamenta la implementación, abordando los conceptos de la capa de transporte, las diferencias entre los protocolos UDP y TCP, y la abstracción proporcionada por los sockets. Posteriormente, se detalla el diseño de la arquitectura de red implementada y se proporciona evidencia experimental del funcionamiento correcto del sistema mediante herramientas de monitoreo de red. Finalmente, se analizan las líneas más relevantes del código fuente, materializando los conceptos teóricos en una solución funcional que demuestra la aplicación práctica de los protocolos de red en el desarrollo de software de comunicaciones.

## Objetivos

### Objetivo general

Desarrollar e implementar un sistema de chat bidireccional utilizando sockets TCP en lenguaje C que demuestre la aplicación práctica de los conceptos teóricos de la capa de transporte y programación de red mediante una arquitectura cliente-servidor concurrente.

### Objetivos específicos

- **Fundamentar teóricamente** los conceptos de la capa de transporte del modelo OSI, diferenciando las características de los protocolos UDP y TCP, explicando el funcionamiento de los sockets como abstracción de programación de red.
- **Implementar una arquitectura cliente-servidor** utilizando sockets TCP que permita la comunicación bidireccional simultánea entre múltiples usuarios, empleando programación concurrente con hilos para gestionar conexiones paralelas y retransmisión de mensajes.
- **Validar y analizar** el funcionamiento del sistema mediante herramientas de monitoreo de red y análisis del código fuente, demostrando cómo los sockets materializan los conceptos teóricos en una solución práctica y funcional.

## Marco Teórico

### *Capa de Transporte*

La programación con sockets se construye sobre los cimientos de la capa de transporte L4 en el modelo OSI. Esta capa juega un papel decisivo como intermediaria entre la capa de aplicación L7 y la capa de red L, determinando a través de los números de puerto y los protocolos TCP/UDP qué aplicación debe recibir o rechazar los paquetes transmitidos entre máquinas.

La capa de transporte permite que un solo dispositivo ejecute múltiples aplicaciones de red simultáneamente, cada una identificada por su propio número de puerto. En nuestro

chat bidireccional, aunque manejamos todo en local, es la capa de transporte la que se encarga de gestionar las comunicaciones entre el servidor del chat y los distintos clientes que se encuentran conectados.

### ***UDP***

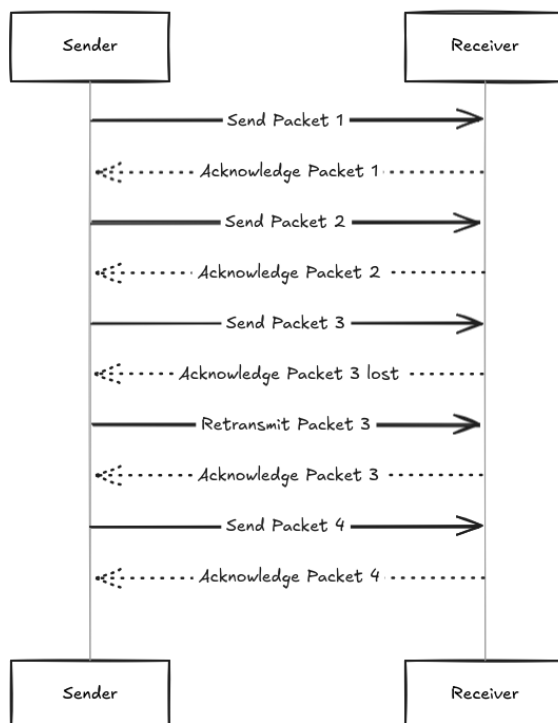
Es un protocolo de red usado donde principalmente prima la velocidad sobre la confiabilidad de la información intercambiada. Se utiliza sobre todo en servicios de streaming de video/audio, videollamadas y juegos online, donde la baja latencia es un requerimiento principal.

### ***TCP***

Este es el protocolo de red que usamos en el chat bidireccional, principalmente porque en este priman dos cosas: la confianza de entrega y el orden de los paquetes.

La confianza de entrega se logra ya que TCP en cada intercambio de información requiere que el receptor envíe un reconocimiento ACK de los paquetes que ha recibido correctamente y, en caso de que falte uno, el emisor reintenta enviar los paquetes faltantes.

En la figura 1 se ilustra el proceso:



**Figura 1**

*Proceso de retransmisión TCP mostrando el mecanismo de acknowledgment y reenvío automático de paquetes perdidos para garantizar la confiabilidad en la entrega.*

La garantía en el orden de los paquetes también se da porque TCP coloca un número de secuencia en cada paquete y, al momento de acabar una transmisión de información, los ordena sin importar si alguno falló y se reintentó en el proceso.

La conexión TCP es bidireccional, es decir que cuando se establece la conexión cliente-servidor, ambos pueden enviar y recibir información. La conexión se establece mediante un proceso de tres pasos llamado TCP handshake. En el primer paso, usando la dirección IP y el puerto del servidor destino, nuestra IP y un número de puerto aleatorio que usaremos para la comunicación, enviamos un paquete de sincronización llamado SYN. Si todo está bien, el servidor responderá con otro paquete llamado SYN-ACK. Como paso siguiente, el cliente responderá con un paquete ACK, sellando la conexión y permitiendo la comunicación bidireccional continua.



## *Socket*

Un socket es una abstracción de software que representa un endpoint de una comunicación. Permite a los programas acceder a los servicios de la capa de transporte L4, actuando como un puente entre el código de aplicación y los protocolos de red subyacentes como TCP o UDP, y se define por una tupla de cinco elementos: protocolo de transporte (TCP/UDP), dirección IP local, puerto local, dirección IP remota y puerto remoto. Esto permite enviar y recibir datos como si fueran operaciones de entrada/salida de archivos. Cuando una aplicación crea un socket, el sistema operativo asigna recursos del sistema para gestionar esa conexión, incluyendo buffers de envío y recepción.

En nuestro chat bidireccional, el socket del servidor actúa como un punto de escucha que acepta conexiones entrantes, mientras que cada socket de cliente establece una conexión punto a punto con el servidor. Esta arquitectura permite que múltiples flujos de datos independientes coexistan simultáneamente, cada uno identificado por su combinación única de direcciones y puertos. Los buffers en el caso del chat son la parte esencial porque permiten la recepción y retransmisión de los mensajes al servidor y a todos los clientes conectados.

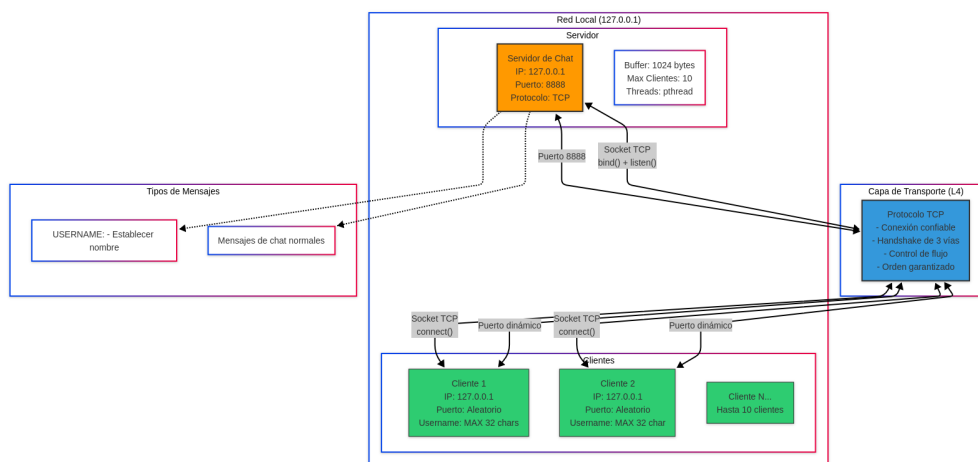
## **Arquitectura de Red y Verificación**

Esta sección presenta el diseño de la arquitectura de red implementada para el chat bidireccional, junto con la verificación experimental de su funcionamiento. La aplicación utiliza una arquitectura cliente-servidor basada en sockets TCP, donde un servidor central gestiona múltiples conexiones simultáneas de clientes a través de la capa de transporte L4.

El diseño implementado permite la comunicación bidireccional confiable entre múltiples usuarios, garantizando la entrega ordenada de mensajes mediante el protocolo TCP y la gestión concurrente de conexiones a través de hilos. La verificación experimental demuestra el correcto funcionamiento de los sockets, el establecimiento de conexiones TCP y el intercambio de datos según lo descrito en el marco teórico.

## Diseño de la Arquitectura de Red

El diagrama 2 ilustra la arquitectura de red implementada, mostrando la configuración cliente-servidor, los puertos utilizados y el flujo de datos a través de la capa de transporte:



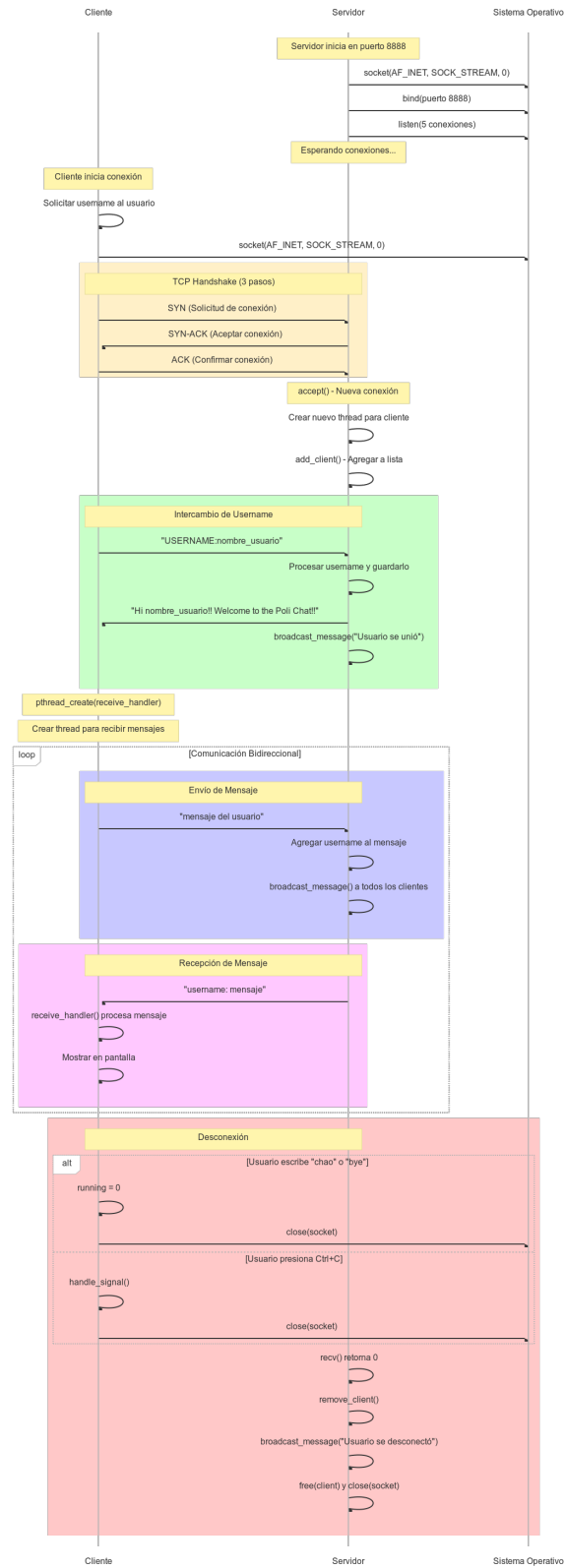
**Figura 2**

*Arquitectura de red del chat bidireccional con servidor TCP en puerto 8888 y múltiples clientes conectados mediante sockets.*

## Verificación

Para verificar el funcionamiento de la arquitectura de red propuesta, se inició el programa servidor y se conectaron dos clientes. Utilizando el comando `netstat -tan | grep 8888` se pueden observar las conexiones TCP específicas en el puerto 8888 donde inició nuestro servidor. En la imagen 3 se puede ver que en la primera fila de resultados se indica que en 0.0.0.0:8888 hay una conexión TCP en estado LISTEN esperando conexiones desde cualquier dirección del localhost. En las demás filas se observa cómo hay conexiones establecidas bidireccionalmente entre el servidor en el puerto 8888 y los clientes con puertos dinámicos 38048 y 47030, ambas en estado ESTABLISHED, confirmando que las conexiones TCP están correctamente establecidas y activas.





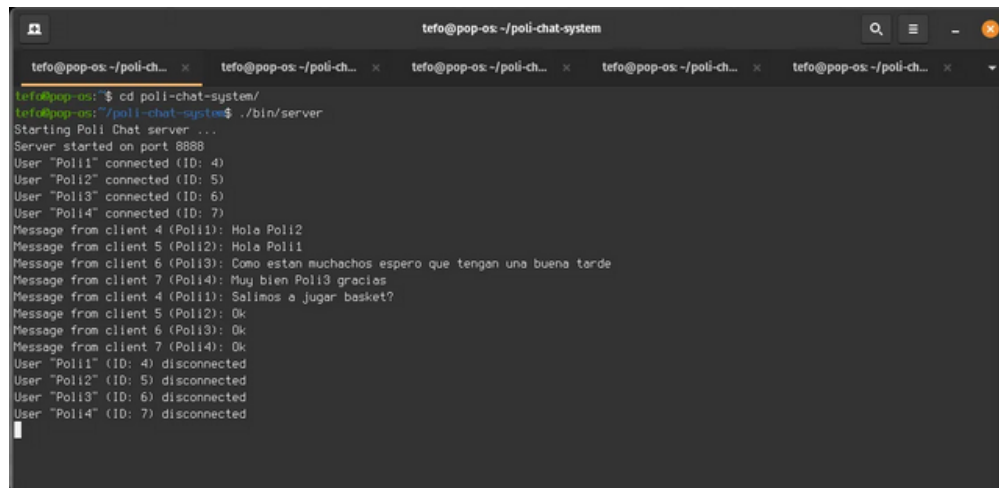
**Figura 4**

*Diagrama de secuencia del chat bidireccional mostrando el handshake TCP, intercambio de username y comunicación entre cliente y servidor.*

## Verificación

Para verificar el funcionamiento del servidor, en la figura 5 se muestran los logs capturados durante la ejecución. Se puede observar cómo inicia el servidor y comienza a escuchar en el puerto 8888, así como la conexión secuencial de cuatro clientes (Poli1, Poli2, Poli3 y Poli4), donde el servidor registra cada una de las conexiones asignándoles un ID único.

Además de capturar las conexiones, el sistema mantiene un seguimiento de los mensajes intercambiados, identificando cada mensaje por usuario e ID del cliente. Los logs también registran el proceso de desconexión de cada usuario, proporcionando una trazabilidad completa del ciclo de vida de las conexiones y la comunicación bidireccional en tiempo real.



```
tefo@pop-os: ~/poli-chat-system
tefo@pop-os: ~/poli-chat-system$ cd poli-chat-system/
tefo@pop-os: ~/poli-chat-system$ ./bin/server
Starting Poli Chat server ...
Server started on port 8888
User "Poli1" connected (ID: 4)
User "Poli2" connected (ID: 5)
User "Poli3" connected (ID: 6)
User "Poli4" connected (ID: 7)
Message from client 4 (Poli1): Hola Poli2
Message from client 5 (Poli2): Hola Poli1
Message from client 6 (Poli3): Como estan muchachos espero que tengan una buena tarde
Message from client 7 (Poli4): Muy bien Poli3 gracias
Message from client 4 (Poli1): Salimos a jugar basket?
Message from client 5 (Poli2): Ok
Message from client 6 (Poli3): Ok
Message from client 7 (Poli4): Ok
User "Poli1" (ID: 4) disconnected
User "Poli2" (ID: 5) disconnected
User "Poli3" (ID: 6) disconnected
User "Poli4" (ID: 7) disconnected
```

**Figura 5**

*Logs del servidor mostrando el inicio del servicio, conexiones de cuatro clientes, intercambio de mensajes y proceso de desconexión.*

Desde el lado del cliente en la figura 6 se puede observar cómo se recibe la solicitud para ingresar el nombre de usuario, seguido del mensaje de bienvenida personalizado por parte del servidor. Los distintos mensajes son recibidos y replicados a todos los clientes conectados, implementando un sistema de identificación que diferencia entre mensajes propios y de otros usuarios, marcando los mensajes propios con "(you):" para evitar

confusiones.

También se evidencia cómo el sistema notifica cuando un usuario abandona el servidor (User 'Poli1' has left the chat) y cómo se maneja la señal de desconexión mediante el comando chao, que permite al cliente desconectarse de manera controlada del servidor, confirmando el correcto funcionamiento de la comunicación bidireccional y la gestión de eventos de conexión y desconexión.

```

tefo@pop-os: ~/poli-chat-system
tefo@pop-os:~/poli-chat-system$ ./bin/client
Enter your username (max 31 characters):Poli2
Connecting to server at 127.0.0.1:8888...
Connected to server!
Hi Poli2!! Welcome to the Poli Chat!!
You can now start chatting. Type 'quit' or 'chao' to exit.
Poli2(you):Hola
Poli1: Hola
User "Poli1" has left the chat
>

tefo@pop-os:~/poli-chat-system$ ./bin/client
Enter your username (max 31 characters):Poli1
Connecting to server at 127.0.0.1:8888...
Connected to server!
Hi Poli1!! Welcome to the Poli Chat!!
You can now start chatting. Type 'quit' or 'chao' to exit.
Poli2: Hola
Poli1(you):Hola
> chao
Disconnection from server...
tefo@pop-os:~/poli-chat-system$

```

**Figura 6**

*Interfaz de dos clientes mostrando la conexión, intercambio de mensajes, identificación de mensajes propios con "(you)" y proceso de desconexión.*

## Implementación

Esta sección analiza las líneas de código que implementan los sockets en el chat bidireccional, siguiendo el flujo completo desde la creación hasta el cierre de las conexiones. El análisis abarca tanto la perspectiva del servidor como del cliente, explicando cómo el código utiliza los conceptos teóricos de TCP y sockets descritos anteriormente. La implementación demuestra el uso de las primitivas de UNIX para establecer conexiones confiables, manejar múltiples clientes simultáneos mediante hilos, y gestionar el intercambio bidireccional de mensajes. Cada subsección examina las funciones críticas de la API de sockets.

## 1. Creación y Configuración del Socket del Servidor

La creación del socket del servidor inicia con la función `socket(AF_INET, SOCK_STREAM, 0)` que establece un endpoint de comunicación utilizando IPv4 y el protocolo TCP, retornando un descriptor de archivo que representa el socket. Posteriormente, se configura la opción `SO_REUSEADDR` mediante `setsockopt()` para permitir la reutilización inmediata de la dirección local, evitando errores durante el desarrollo cuando se reinicia el servidor. La estructura `sockaddr_in server_addr` se configura especificando la familia de protocolos IPv4, utilizando `INADDR_ANY` para aceptar conexiones desde cualquier interfaz de red, y convirtiendo el puerto 8888 al formato de red mediante `htons()` para asegurar compatibilidad entre diferentes arquitecturas de sistema.

```
1 server_socket = socket(AF_INET, SOCK_STREAM, 0);
2 if (server_socket == -1) {
3     perror("Failed to create socket");
4     exit(EXIT_FAILURE);
5 }
6
7 int opt = 1;
8 if (setsockopt(server_socket, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt))
9     < 0) {
10     perror("Failed to set socket options");
11     exit(EXIT_FAILURE);
12 }
13
14 memset(&server_addr, 0, sizeof(server_addr));
15 server_addr.sin_family = AF_INET;
16 server_addr.sin_addr.s_addr = INADDR_ANY;
17 server_addr.sin_port = htons(PORT);
```

## 2. Bind y Listen del Servidor

El proceso de bind asocia el socket creado con una dirección específica mediante `bind(server_socket, (struct sockaddr *)&server_addr, sizeof(server_addr))`, reservando efectivamente el puerto 8888 para nuestro servidor y transformando la dirección de formato interno a formato de red. Una vez establecida la asociación, `listen(server_socket, 5)` marca el socket como pasivo y lo prepara para aceptar conexiones entrantes, estableciendo una cola de backlog de 5 conexiones pendientes que pueden esperar mientras el servidor procesa otras solicitudes. El socket transiciona del estado CLOSED al estado LISTEN, y se inicializa el array global `clients[]` con valores NULL para gestionar hasta 10 clientes simultáneos según la constante `MAX_CLIENTS` definida en el header.

```
1  if (bind(server_socket, (struct sockaddr *)&server_addr, sizeof(
    server_addr)) < 0) {
2      perror("Failed to bind socket");
3      exit(EXIT_FAILURE);
4  }
5
6  if (listen(server_socket, 5) < 0) {
7      perror("Failed to listen for connections");
8      exit(EXIT_FAILURE);
9  }
10
11  printf("Server started on port %d\n", PORT);
12
13  for (int i = 0; i < MAX_CLIENTS; i++) {
14      clients[i] = NULL;
15  }
```

## 3. Aceptación de Conexiones de Clientes

El servidor entra en un loop infinito donde `accept()` actúa como una llamada bloqueante que espera conexiones entrantes, capturando la información del cliente en



`client_addr` y retornando un nuevo descriptor de socket específico para cada cliente mientras el socket del servidor original continúa escuchando nuevas conexiones. Para cada conexión aceptada, se reserva memoria dinámica mediante `malloc()` para crear una estructura `client_t` que almacena el descriptor del socket, la dirección del cliente, un ID único basado en el descriptor, y un campo de username inicializado como string vacío. La gestión concurrente se implementa creando un hilo independiente con `pthread_create()` que ejecuta la función `handle_client()`, permitiendo que múltiples clientes se comuniquen simultáneamente, y utilizando `pthread_detach()` para que cada hilo se limpie automáticamente al terminar, con manejo de errores que incluye limpieza de memoria y cierre de conexiones si falla la creación del hilo.

```

1 while (1) {
2     client_size = sizeof(client_addr);
3     client_socket = accept(server_socket, (struct sockaddr *)&client_addr,
4                             &client_size);
5
6     if (client_socket < 0) {
7         perror("Failed to accept connection");
8         continue;
9     }
10
11     client_t *client = (client_t *)malloc(sizeof(client_t));
12     client->socket = client_socket;
13     client->address = client_addr;
14     client->id = client_socket;
15     client->username[0] = '\0';
16
17     add_client(client);
18
19     int thread_result = pthread_create(&tid, NULL, &handle_client, (void
20                                     *)client);
21
22     if (thread_result != 0) {

```

```

20     printf("Failed to create thread:%s\n", strerror(thread_result));
21     remove_client(client->id);
22     free(client);
23     close(client_socket);
24     continue;
25 }
26 pthread_detach(tid);
27 }

```

#### 4. Creación del Socket del Cliente

El cliente establece su comunicación creando un socket TCP mediante `socket(AF_INET, SOCK_STREAM, 0)` con los mismos parámetros que el servidor, almacenando el descriptor en una variable global `client_socket` que servirá como único punto de comunicación con el servidor. La configuración de la dirección del servidor se realiza mediante una estructura `sockaddr_in` donde se especifica IPv4, se convierte el puerto al formato de red con `htons()`, y se utiliza `inet_pton()` para transformar la dirección IP string ("127.0.0.1") a formato binario de manera segura. A diferencia del servidor, el cliente no requiere operaciones de `bind` o `listen` ya que el sistema operativo asigna automáticamente un puerto local aleatorio, y su propósito es conectarse a un servidor existente en lugar de aceptar conexiones entrantes.

```

1  int connect_to_server(const char *server_ip, int port) {
2      struct sockaddr_in server_addr;
3
4      client_socket = socket(AF_INET, SOCK_STREAM, 0);
5      if (client_socket == -1) {
6          perror("Could not create socket");
7          return -1;
8      }
9
10     memset(&server_addr, 0, sizeof(server_addr));

```

```

11     server_addr.sin_family = AF_INET;
12     server_addr.sin_port = htons(port);
13
14     if (inet_pton(AF_INET, server_ip, &server_addr.sin_addr) <= 0) {
15         perror("Invalid address / Address not supported");
16         return -1;
17     }
18
19     return 0;
20 }

```

### 5. Establecimiento de Conexión del Cliente

La función `connect()` inicia el handshake TCP de tres pasos SYN, SYN-ACK, ACK descrito en la teoría, estableciendo una conexión confiable con el servidor mediante una llamada bloqueante que espera hasta completar el proceso o fallar, momento en el cual el sistema operativo asigna automáticamente un puerto local aleatorio al cliente. Una vez establecida la conexión TCP, el cliente envía inmediatamente su identificación al servidor mediante `send_username()`, que concatena el prefijo `"USERNAME:"` con el nombre de usuario utilizando `sprintf()` y transmite los datos a través del socket establecido mediante `send()`. Este intercambio representa la primera comunicación a nivel de aplicación sobre la conexión TCP, estableciendo el protocolo de identificación que permitirá al servidor asociar mensajes futuros con el usuario específico y habilitar la comunicación bidireccional completa.

```

1 printf("Connecting to server at %s:%d...\n", server_ip, port);
2 if (connect(client_socket, (struct sockaddr *)&server_addr, sizeof(
    server_addr)) < 0) {
3     perror("Connection failed");
4     return -1;
5 }
6
7 printf("Connected to server!\n");

```

```

8
9 void send_username(int socket, const char *username) {
10     char buffer[BUFFER_SIZE];
11     sprintf(buffer, "%s%s", MSG_TYPE_USERNAME, username);
12     send(socket, buffer, strlen(buffer), 0);
13 }
14
15 send_username(client_socket, username);

```

## 6. Envío y Recepción de Datos

La recepción de datos en el servidor se maneja mediante `recv()`, una función bloqueante que espera datos del cliente y retorna el número de bytes recibidos, asegurando la terminación correcta del string con `buffer[bytes_received] = '\0'` y diferenciando tipos de mensajes mediante prefijos como `"USERNAME:"` para procesar la identificación inicial del usuario. El servidor implementa un sistema de broadcast mediante `broadcast_message()` que formatea cada mensaje con el username del remitente y lo envía a todos los clientes conectados utilizando `send()` en cada socket activo, protegiendo el acceso concurrente a la lista de clientes con `pthread_mutex_lock()` para prevenir condiciones de carrera. En el lado del cliente, la recepción se maneja en un thread separado mediante `receive_handler()` que ejecuta un loop continuo con `recv()`, limpiando el buffer con `memset()` antes de cada operación y implementando lógica para identificar mensajes propios marcándolos con `"(you):"` para diferenciación visual, permitiendo así comunicación bidireccional simultánea donde el usuario puede enviar y recibir mensajes al mismo tiempo.

```

1 while ((bytes_received = recv(client->socket, buffer, BUFFER_SIZE, 0)) >
2     0) {
3
4     if (strncmp(buffer, MSG_TYPE_USERNAME, strlen(MSG_TYPE_USERNAME)) == 0
5         && !username_set) {
6         char *new_username = buffer + strlen(MSG_TYPE_USERNAME);

```

```

6         strncpy(client->username, new_username, sizeof(client->username) -
              1);
7         username_set = 1;
8
9         snprintf(temp_buffer, BUFFER_SIZE, "Hi %s!! Welcome to the Poli
              Chat!!\n",
10                client->username);
11         send(client->socket, temp_buffer, strlen(temp_buffer), 0);
12     }
13
14     snprintf(temp_buffer, BUFFER_SIZE, "%s: %.*s", client->username,
15             BUFFER_SIZE - (int)strlen(client->username) - 3, buffer);
16     broadcast_message(temp_buffer, client->id, 0);
17 }
18
19 void broadcast_message(const char *message, int sender_id, int
    exclude_sender) {
20     pthread_mutex_lock(&clients_mutex);
21
22     for (int i = 0; i < MAX_CLIENTS; i++) {
23         if (clients[i]) {
24             if (exclude_sender && clients[i]->id == sender_id) {
25                 continue;
26             }
27             send(clients[i]->socket, message, strlen(message), 0);
28         }
29     }
30     pthread_mutex_unlock(&clients_mutex);
31 }
32
33 void *receive_handler(void *arg) {
34     char buffer[BUFFER_SIZE];
35     int receive_len;

```

```

36
37     while (running) {
38         memset(buffer, 0, BUFFER_SIZE);
39         receive_len = recv(client_socket, buffer, BUFFER_SIZE, 0);
40
41         if (receive_len > 0) {
42             if (strncmp(buffer, proper_username, strlen(proper_username))
43                 == 0) {
44                 snprintf(modified_buffer, BUFFER_SIZE, "%s(you):%s",
45                     username,
46                     buffer + strlen(proper_username));
47                 printf("%s", modified_buffer);
48             } else {
49                 printf("%s", buffer);
50             }
51         }
52     }
53     return NULL;
54 }

```

## 7. Manejo de Hilos para Concurrency

El servidor implementa concurrencia creando un hilo independiente por cada cliente mediante `pthread_create()` que ejecuta la función `handle_client()`, utilizando `pthread_detach()` para permitir auto-limpieza automática de hilos y manejando errores de creación con limpieza completa de memoria y conexiones, lo que permite que el hilo principal continúe aceptando nuevas conexiones mientras múltiples hilos manejan comunicaciones individuales simultáneamente. El cliente utiliza una arquitectura de dos hilos donde el hilo principal maneja el envío de mensajes desde el teclado mientras un hilo separado ejecuta `receive_handler()` para escuchar mensajes del servidor de forma asíncrona, eliminando la necesidad de bloqueo en operaciones de entrada/salida. La sincronización se garantiza mediante un mutex (`pthread_mutex_t clients_mutex`) que protege el acceso

concurrente al array global `clients[]` en las funciones `add_client()`, `remove_client()` y `broadcast_message()`, previniendo condiciones de carrera donde múltiples hilos podrían modificar la lista de clientes simultáneamente y asegurando la integridad de los datos compartidos.

```
1  int thread_result = pthread_create(&tid, NULL, &handle_client, (void *)
    client);
2  if (thread_result != 0) {
3      printf("Failed to create thread:%s\n", strerror(thread_result));
4      remove_client(client->id);
5      free(client);
6      close(client_socket);
7      continue;
8  }
9  pthread_detach(tid);
10
11 if (pthread_create(&receive_thread, NULL, receive_handler, NULL) != 0) {
12     perror("Failed to create receive thread");
13     close(client_socket);
14     return 1;
15 }
16 pthread_detach(receive_thread);
17
18 pthread_mutex_t clients_mutex = PTHREAD_MUTEX_INITIALIZER;
19
20 void add_client(client_t *client) {
21     pthread_mutex_lock(&clients_mutex);
22
23     for (int i = 0; i < MAX_CLIENTS; i++) {
24         if (clients[i] == NULL) {
25             clients[i] = client;
26             break;
27         }
```

```

28     }
29
30     pthread_mutex_unlock(&clients_mutex);
31 }
32
33 void remove_client(int id) {
34     pthread_mutex_lock(&clients_mutex);
35
36     for (int i = 0; i < MAX_CLIENTS; i++) {
37         if (clients[i] && clients[i]->id == id) {
38             clients[i] = NULL;
39             break;
40         }
41     }
42
43     pthread_mutex_unlock(&clients_mutex);
44 }

```

## 8. Cierre y Limpieza de Sockets

La detección de desconexión en el servidor ocurre cuando `recv()` retorna 0 o un valor negativo, momento en el cual se notifica a otros clientes mediante `broadcast_message()` con exclusión del remitente, se ejecuta la secuencia de limpieza que incluye `close(client->socket)` para liberar el descriptor, `remove_client()` para eliminar al cliente de la lista global, `free(client)` para liberar memoria dinámica, y `return NULL` para eliminar el hilo limpiamente. El cliente implementa desconexión controlada mediante `handle_signal()` para manejar Ctrl+C estableciendo `running = 0` y cerrando el socket, y mediante detección de comandos especiales ("**chao**", "**bye**") en el loop principal que activan `client_cleanup()` para cerrar la conexión de manera ordenada. La limpieza de recursos compartidos se gestiona mediante `remove_client()` que utiliza mutex para marcar posiciones como disponibles en el array de clientes, mientras que el cierre apropiado de sockets mediante



`close()` inicia el proceso TCP de 4-way handshake para terminación ordenada, previniendo sockets huérfanos, memory leaks y resource leaks en el sistema operativo, y asegurando que el servidor pueda aceptar nuevos clientes en el futuro.

```

1 while ((bytes_received = recv(client->socket, buffer, BUFFER_SIZE, 0)) >
    0) {
2     Procesamiento de mensajes
3 }
4
5 printf("User \"%s\" (ID: %d) disconnected\n", client->username, client->id
    );
6 snprintf(temp_buffer, BUFFER_SIZE, "User \"%s\" has left the chat\n",
    client->username);
7 broadcast_message(temp_buffer, client->id, 1);
8
9
10 close(client->socket);
11 remove_client(client->id);
12 free(client);
13 return NULL;
14
15 void handle_signal(int sig) {
16     (void)sig;
17     running = 0;
18     printf("\nDisconnecting from server...\n");
19     close(client_socket);
20     exit(EXIT_SUCCESS);
21 }
22
23
24 while (running) {
25     if (fgets(buffer, BUFFER_SIZE, stdin) == NULL) {
26         break;

```

```

27     }
28
29     if (strncmp(buffer, "chao", 4) == 0 || strncmp(buffer, "bye", 3) == 0)
30     {
31         running = 0;
32         break;
33     }
34
35     send(client_socket, buffer, strlen(buffer), 0);
36 }
37
38 void client_cleanup(void) {
39     printf("Disconnection from server...\n");
40     close(client_socket);
41 }

```

## Video

Link del video de Youtube: <https://www.youtube.com/watch?v=dq0tbqsWexA>

## Código

El código completo se puede encontrar en el siguiente repositorio:

<https://github.com/tefoos/poli-chat-system>

## Conclusiones

Se implementó exitosamente un sistema de chat bidireccional con sockets TCP que aplica los conceptos de programación de red en una solución funcional. TCP fue la mejor opción para este proyecto porque garantiza la entrega ordenada y confiable de mensajes, algo esencial para mantener conversaciones coherentes entre usuarios. Su mecanismo de ACK y retransmisión automática evita que se pierdan mensajes durante la comunicación.

Los hilos fueron fundamentales para el funcionamiento del sistema. En el servidor, cada cliente es manejado por un hilo independiente que evita interferencias entre usuarios,

mientras que la sincronización mediante mutex previene problemas durante la modificación de datos compartidos como la lista de clientes conectados. En el cliente, separar los hilos de envío y recepción permite escribir y recibir mensajes simultáneamente sin bloqueos.

La validación experimental confirmó el funcionamiento correcto del protocolo TCP, mostrando el establecimiento apropiado de conexiones, intercambio ordenado de datos y terminación limpia de sesiones.

## Referencias

Birch, J. (2024). *Foundations of network technology* (1st ed.). Björkulturförbundet EF. ISBN 978-91-52-74980-6.