OPERATING SYSTEMS

Practical Work - Programming in C for UNIX

IVÁN ESTÉPAR REBOLLLO JIMENA ARNAIZ GONZÁLEZ

INDEX

1. Introduction

The implemented system is a platform where users can send and receive messages organized by topics. Users subscribe to topics, and when a message is sent to a topic, all subscribers receive it. The system comprises two main programs:

1. **Manager.**
2. **Feed.**

2. System architecture

- ➢ **Manager program**

  The manager.c file defines the server-side logic.

  Key components:

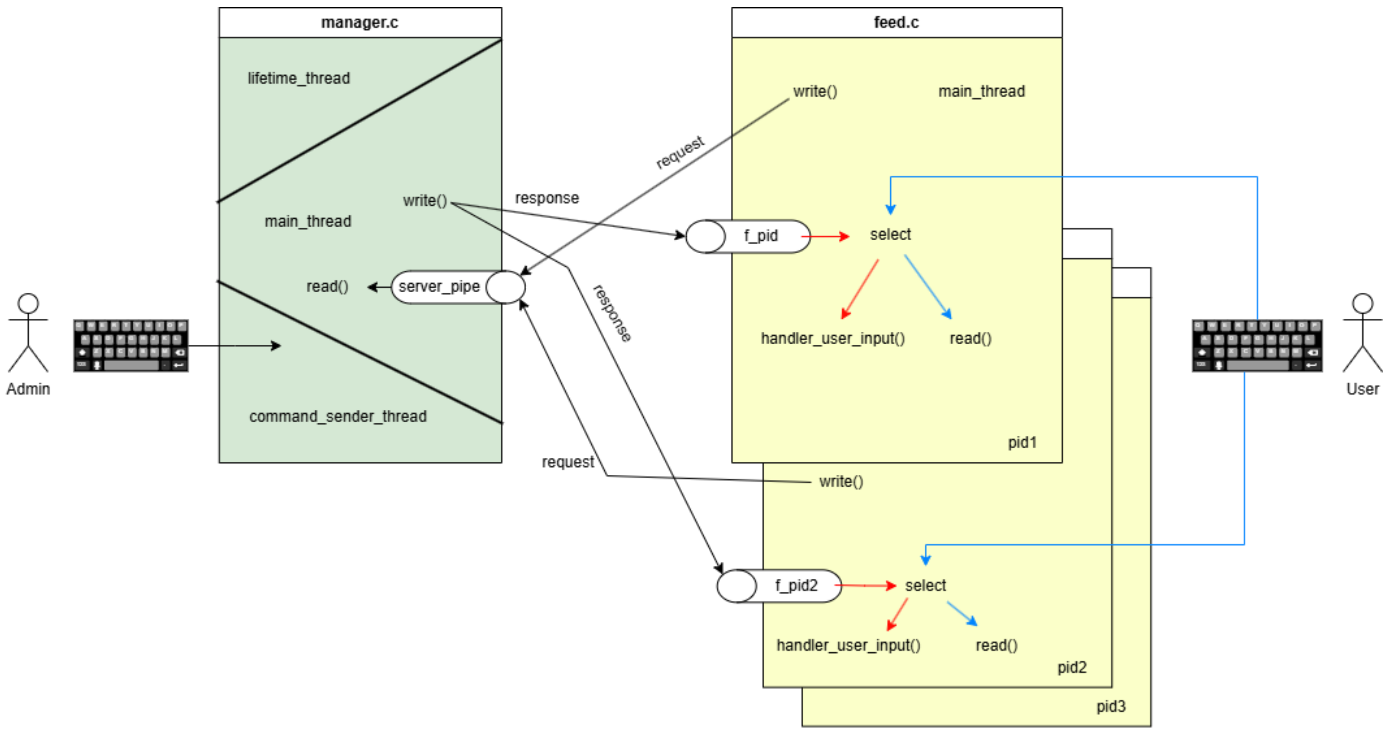
  - **Topic management**: Topics are dynamically created when messages are sent to a non-existent topic or when a user subscribes to that topic (as long as defined limits are not exceeded).
  - **Message management**: Messages are classified as persistent (stored with a specified lifetime) or non-persistent (discarded after delivery). Persistent messages are reloaded from disk upon startup.
  - **User management**: Users are identified by unique names, stored in a structure that includes their communication pipe and process ID.

- ➢ **Feed program**

  The feed.c file defines the client-side logic.

  Key operations:

  - Subscribe/Unsubscribe to topics.
  - Send messages with a lifetime parameter for persistence.
  - Receive real-time messages from subscribed topics.
  - Exit the message platform.



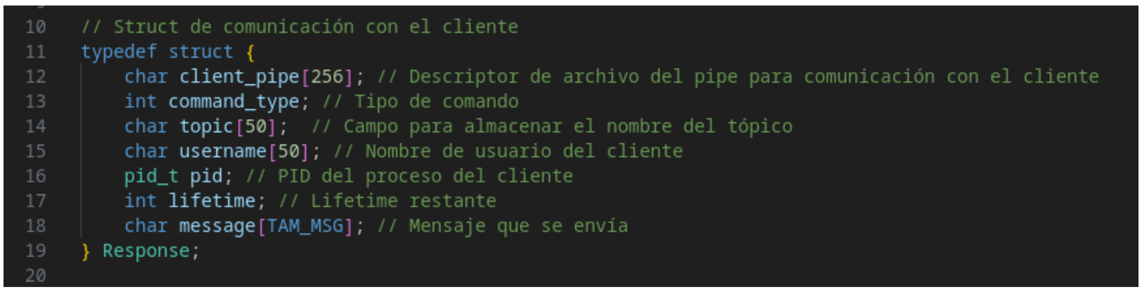
3. Implementation details

3.1 Comunication mechanism

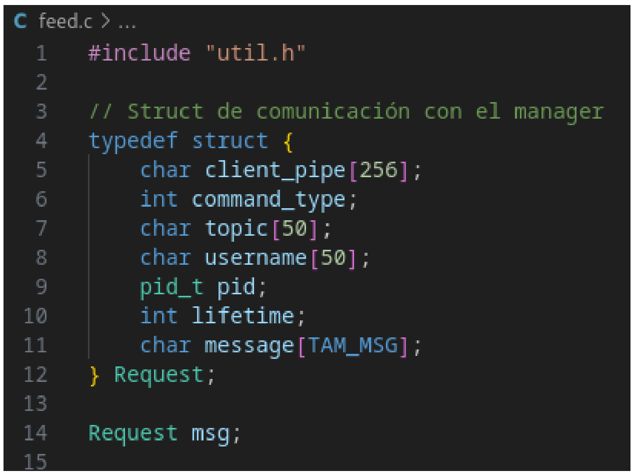
The platform uses named pipes for message exchange:

- **Server pipe (server_pipe):** Receives commands from users.
- **Client pipes (client_pipe_<pid>):** Specific to each user, used for server responses.

Communication between the manager and feed is achieved using a data structure called **Response** in manager.c and **Request** in feed.c.



```c
// Struct de comunicación con el cliente
typedef struct {
    char client_pipe[256]; // Descriptor de archivo del pipe para comunicación con el cliente
    int command_type; // Tipo de comando
    char topic[50];  // Campo para almacenar el nombre del tópico
    char username[50]; // Nombre de usuario del cliente
    pid_t pid; // PID del proceso del cliente
    int lifetime; // Lifetime restante
    char message[TAM_MSG]; // Mensaje que se envía
} Response;
```



```c
#include "util.h"

// Struct de comunicación con el manager
typedef struct {
    char client_pipe[256];
    int command_type;
    char topic[50];
    char username[50];
    pid_t pid;
    int lifetime;
    char message[TAM_MSG];
} Request;

Request msg;
```

3.2 Strategies and models followed

- **Using threads in manager.c**

  Threads were used on the server side to ensure non-blocking operations and to handle concurrency.

  - **Message Lifetime Management Thread (manage_lifetime):** This thread reduces the lifetime of persistent messages, removes expired ones, and updates the persistent message file without blocking other functions.
  - **Administrator Command Execution Thread (command_sender):** This thread processes administrator commands (e.g., lock, unlock, remove, etc.), allowing the server to accept commands without blocking.
  - **Main Thread (main):** The server performs initial setup, including establishing an environment variable to store persistent messages in a file, loading messages from the previous manager file, setting up signals, ensuring only one active instance, and creating a FIFO (SERVER_PIPE). It also launches the manage_lifetime and command_sender threads, and in its main loop, handles client requests such as connection, subscription, topic listing, message sending, and exit, synchronizing access to shared data using a mutex.

- **Using Mutex in manager.c**

Mutex was used to ensure synchronization of access to shared resources (such as user and topic data structures). Since both the server and clients may modify these resources concurrently, a mutex was used to avoid race conditions.

➢ Using SELECT() in feed.c

The feed program uses the SELECT system call to manage data input from multiple sources concurrently without the need for additional threads. This is achieved using FD_SET, which monitors multiple file descriptors (in this case, standard input and a client pipe) and performs actions based on events from these descriptors.

Role of SELECT in this context:

1. **Wait for user commands:** When the user types a command (e.g., "subscribe," "msg," "exit"), the program processes this input and sends it to the server.
2. **Read server response:** When the server sends a response, the program reads the message and displays it on the screen.

➢ Handling command_type

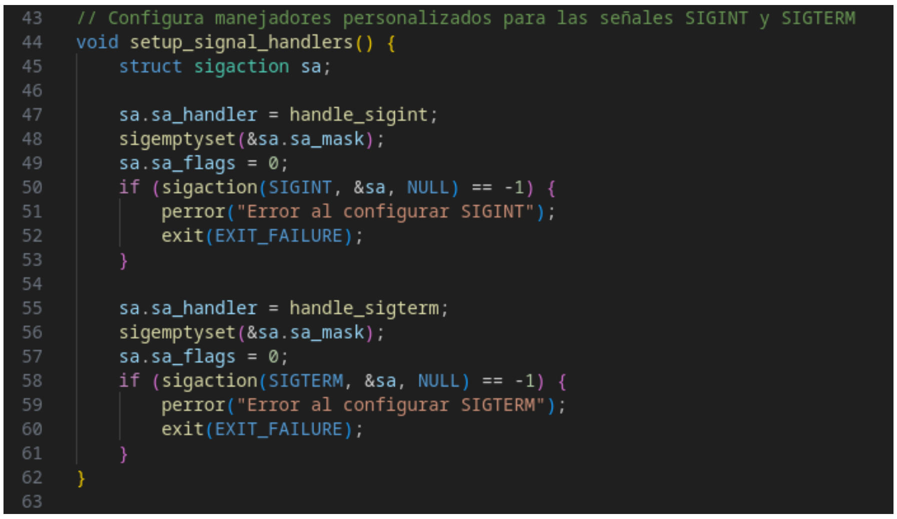
A command_type field is used in the Response and Request structures to identify the type of action the server should perform (e.g., subscription, message sending, etc.). Each action has an assigned value for the command_type field.

➢ Signal Handling

Signal handlers ensure orderly termination:

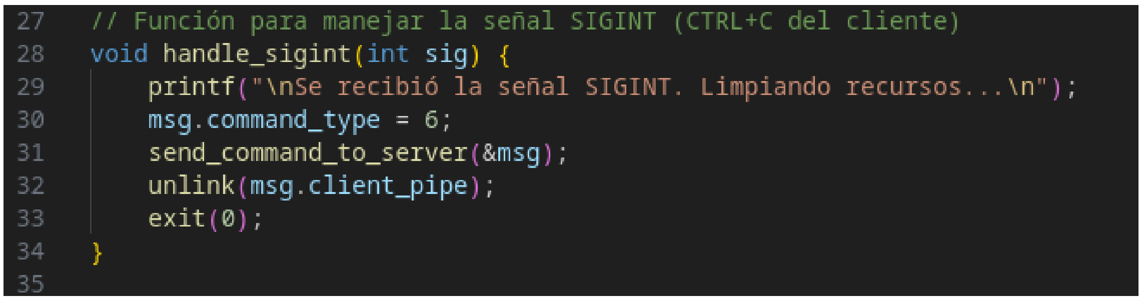
FEED:

This signal handler is responsible for choosing the method that will be executed based on the activated signal in feed.c. If the **SIGINT** signal is triggered, it will redirect the signal to the **handle_sigint()** method, and if the **SIGTERM** signal is triggered, it will redirect the signal to the **handle_sigterm()** method.



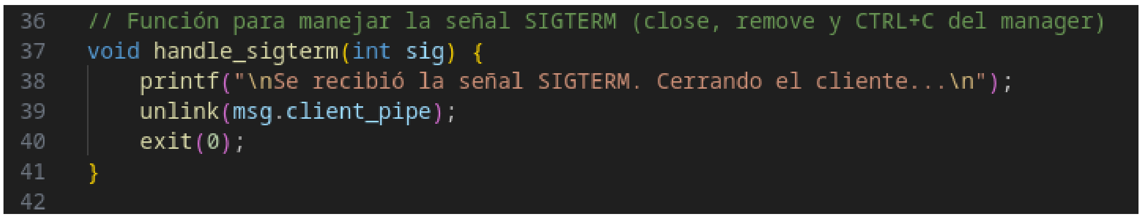
```c
// Configura manejadores personalizados para las señales SIGINT y SIGTERM
void setup_signal_handlers() {
    struct sigaction sa;

    sa.sa_handler = handle_sigint;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    if (sigaction(SIGINT, &sa, NULL) == -1) {
        perror("Error al configurar SIGINT");
        exit(EXIT_FAILURE);
    }

    sa.sa_handler = handle_sigterm;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    if (sigaction(SIGTERM, &sa, NULL) == -1) {
        perror("Error al configurar SIGTERM");
        exit(EXIT_FAILURE);
    }
}

```

The **handle_sigint()** function communicates with the manager using command_type 6 to terminate the current client's process when the **SIGINT** signal is received.



```c
// Función para manejar la señal SIGINT (CTRL+C del cliente)
void handle_sigint(int sig) {
    printf("\nSe recibió la señal SIGINT. Limpiando recursos...\n");
    msg.command_type = 6;
    send_command_to_server(&msg);
    unlink(msg.client_pipe);
    exit(0);
}

```

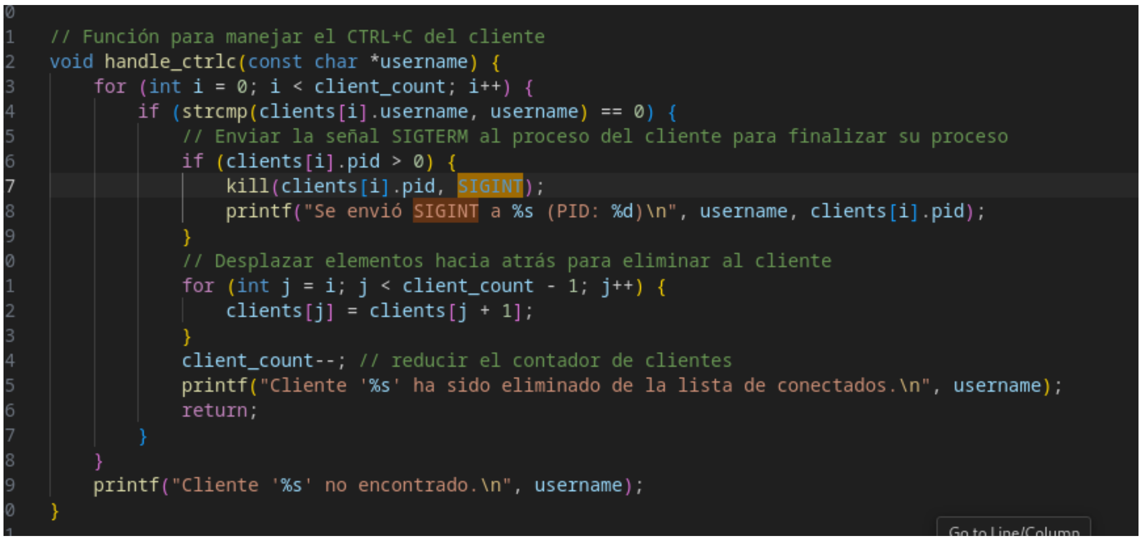
The **handle_sigterm()** function waits to receive a **SIGTERM** signal from the manager to terminate the current client's process.



```c
// Función para manejar la señal SIGTERM (close, remove y CTRL+C del manager)
void handle_sigterm(int sig) {
    printf("\nSe recibió la señal SIGTERM. Cerrando el cliente...\n");
    unlink(msg.client_pipe);
    exit(0);
}

```

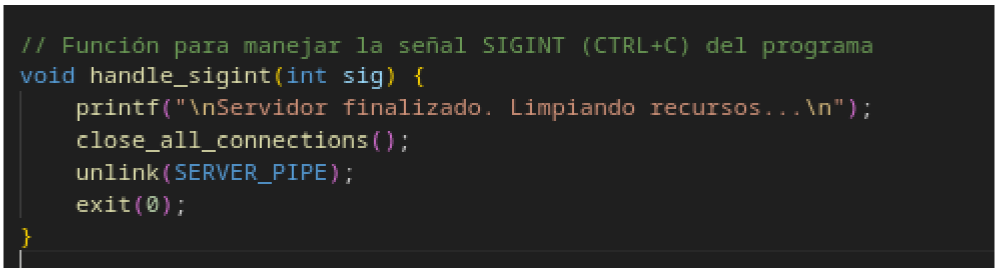
MANAGER:

When the user (feed.c) press on CTRL+C, **SIGINT** signal is sent to the user and the user exits the platform.



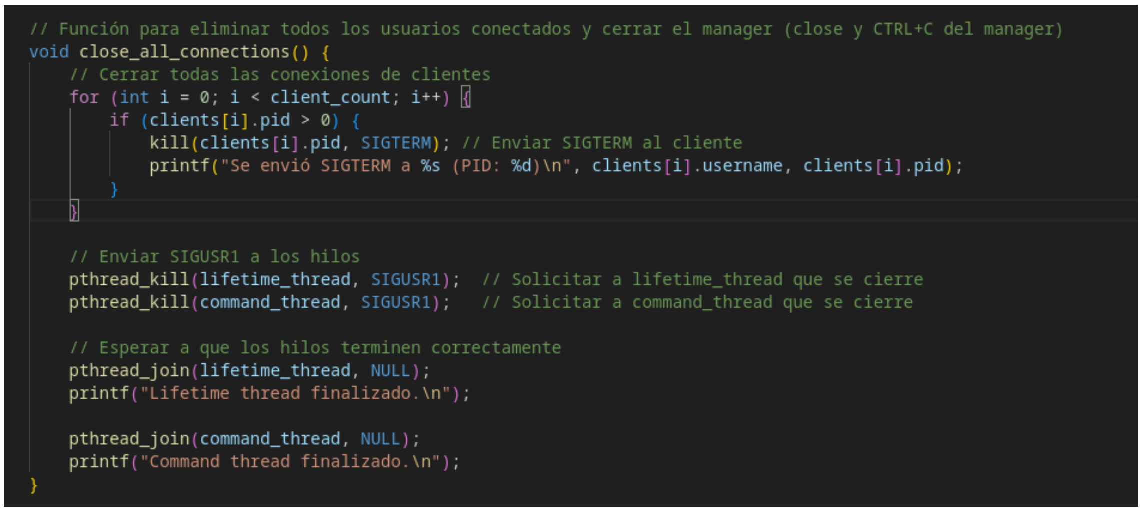
```c

// Función para manejar el CTRL+C del cliente
void handle_ctrlc(const char *username) {
    for (int i = 0; i < client_count; i++) {
        if (strcmp(clients[i].username, username) == 0) {
            // Enviar la señal SIGTERM al proceso del cliente para finalizar su proceso
            if (clients[i].pid > 0) {
                kill(clients[i].pid, SIGINT);
                printf("Se envió SIGINT a %s (PID: %d)\n", username, clients[i].pid);
            }
            // Desplazar elementos hacia atrás para eliminar al cliente
            for (int j = i; j < client_count - 1; j++) {
                clients[j] = clients[j + 1];
            }
            client_count--; // reducir el contador de clientes
            printf("Cliente '%s' ha sido eliminado de la lista de conectados.\n", username);
            return;
        }
    }
    printf("Cliente '%s' no encontrado.\n", username);
}
```

When the manager.c performs CTRL+C, the **SIGINT** signal is triggered and handled in the following function, which calls **close_all_connections()**, explained in the next point.



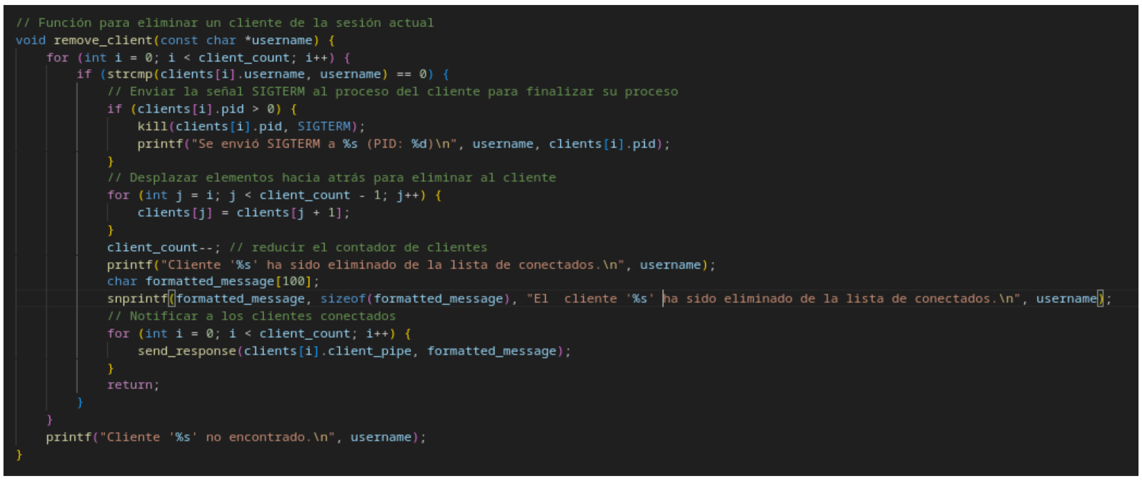
```c
// Función para manejar la señal SIGINT (CTRL+C) del programa
void handle_sigint(int sig) {
    printf("\nServidor finalizado. Limpiando recursos...\n");
    close_all_connections();
    unlink(SERVER_PIPE);
    exit(0);
}
```

This method, **close_all_connections()**, closes all active client connections and terminates the server threads. First, it sends a **SIGTERM** signal to all connected clients (identified by their PID) to terminate their execution. Then, it sends a **SIGUSR1** signal to the lifetime_thread and command_thread, indicating that they should close. Finally, it waits for both threads to complete their execution properly using pthread_join, ensuring that all resources are freed before the process ends. This function is used when manager.c performs CTRL+C or wants to close the platform with the "close" command.



```c
// Función para eliminar todos los usuarios conectados y cerrar el manager (close y CTRL+C del manager)
void close_all_connections() {
    // Cerrar todas las conexiones de clientes
    for (int i = 0; i < client_count; i++) {
        if (clients[i].pid > 0) {
            kill(clients[i].pid, SIGTERM); // Enviar SIGTERM al cliente
            printf("Se envió SIGTERM a %s (PID: %d)\n", clients[i].username, clients[i].pid);
        }
    }

    // Enviar SIGUSR1 a los hilos
    pthread_kill(lifetime_thread, SIGUSR1);  // Solicitar a lifetime_thread que se cierre
    pthread_kill(command_thread, SIGUSR1);   // Solicitar a command_thread que se cierre

    // Esperar a que los hilos terminen correctamente
    pthread_join(lifetime_thread, NULL);
    printf("Lifetime thread finalizado.\n");

    pthread_join(command_thread, NULL);
    printf("Command thread finalizado.\n");
}
```

Also, when the manager removes a user with "remove", a **SIGTERM** signal is sent to the client's process to terminate its execution.

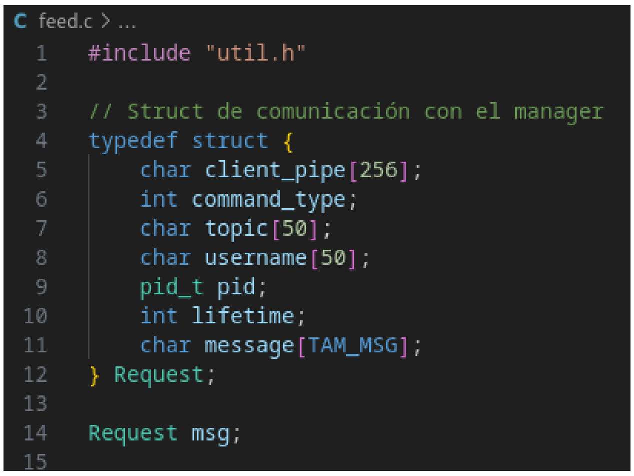


```c
// Función para eliminar un cliente de la sesión actual
void remove_client(const char *username) {
    for (int i = 0; i < client_count; i++) {
        if (strcmp(clients[i].username, username) == 0) {
            // Enviar la señal SIGTERM al proceso del cliente para finalizar su proceso
            if (clients[i].pid > 0) {
                kill(clients[i].pid, SIGTERM);
                printf("Se envió SIGTERM a %s (PID: %d)\n", username, clients[i].pid);
            }
            // Desplazar elementos hacia atrás para eliminar al cliente
            for (int j = i; j < client_count - 1; j++) {
                clients[j] = clients[j + 1];
            }
            client_count--; // reducir el contador de clientes
            printf("Cliente '%s' ha sido eliminado de la lista de conectados.\n", username);
            char formatted_message[100];
            snprintf(formatted_message, sizeof(formatted_message), "El  cliente '%s' ha sido eliminado de la lista de conectados.\n", username);
            // Notificar a los clientes conectados
            for (int i = 0; i < client_count; i++) {
                send_response(clients[i].client_pipe, formatted_message);
            }
            return;
        }
    }
    printf("Cliente '%s' no encontrado.\n", username);
}
```

3.3 Definitions of structures

- **FEED.C**

This structure enables communication between the client and the manager.

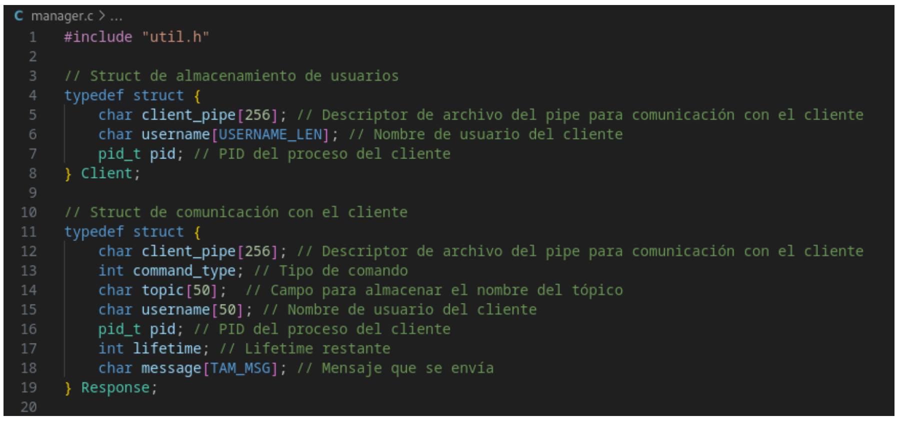


feed.c > ...

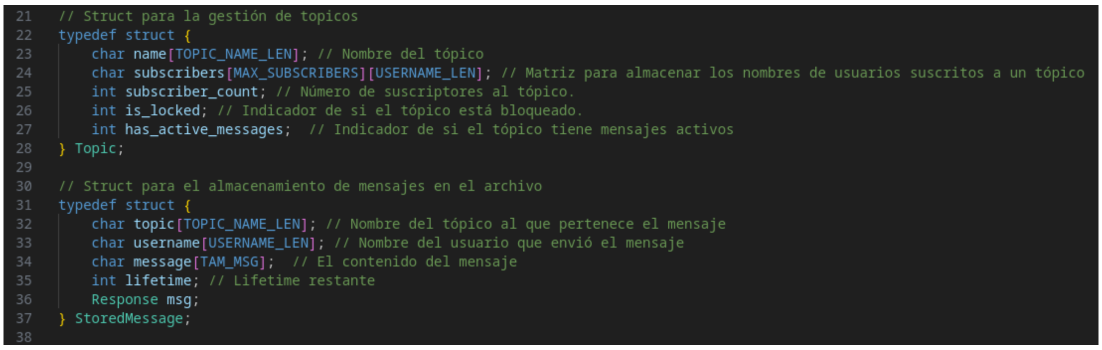
```c
#include "util.h"

// Struct de comunicación con el manager
typedef struct {
    char client_pipe[256];
    int command_type;
    char topic[50];
    char username[50];
    pid_t pid;
    int lifetime;
    char message[TAM_MSG];
} Request;

Request msg;

```

- **MANAGER.C**

The **Client** structure allows for the storage of clients.
The **Response** structure enables communication between the manager and the client.
The **Topic** structure manages topics.
The **StoredMessage** structure allows for the storage of messages in the file.

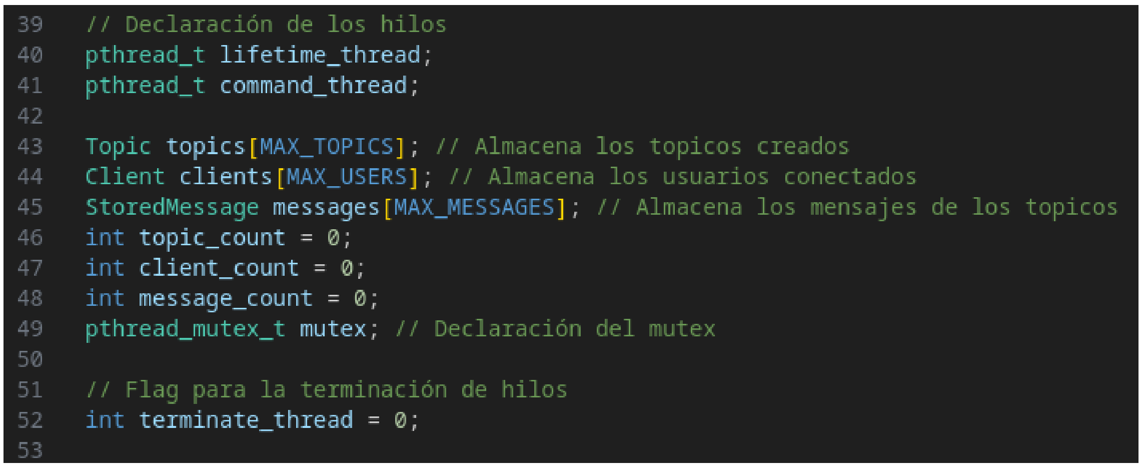


```c
#include "util.h"

// Struct de almacenamiento de usuarios
typedef struct {
    char client_pipe[256]; // Descriptor de archivo del pipe para comunicación con el cliente
    char username[USERNAME_LEN]; // Nombre de usuario del cliente
    pid_t pid; // PID del proceso del cliente
} Client;

// Struct de comunicación con el cliente
typedef struct {
    char client_pipe[256]; // Descriptor de archivo del pipe para comunicación con el cliente
    int command_type; // Tipo de comando
    char topic[50];  // Campo para almacenar el nombre del tópico
    char username[50]; // Nombre de usuario del cliente
    pid_t pid; // PID del proceso del cliente
    int lifetime; // Lifetime restante
    char message[TAM_MSG]; // Mensaje que se envía
} Response;

```



```c
// Struct para la gestión de topicos
typedef struct {
    char name[TOPIC_NAME_LEN]; // Nombre del tópico
    char subscribers[MAX_SUBSCRIBERS][USERNAME_LEN]; // Matriz para almacenar los nombres de usuarios suscritos a un tópico
    int subscriber_count; // Número de suscriptores al tópico.
    int is_locked; // Indicador de si el tópico está bloqueado.
    int has_active_messages;  // Indicador de si el tópico tiene mensajes activos
} Topic;

// Struct para el almacenamiento de mensajes en el archivo
typedef struct {
    char topic[TOPIC_NAME_LEN]; // Nombre del tópico al que pertenece el mensaje
    char username[USERNAME_LEN]; // Nombre del usuario que envió el mensaje
    char message[TAM_MSG];  // El contenido del mensaje
    int lifetime; // Lifetime restante
    Response msg;
} StoredMessage;

```

Here below are the threads used, the arrays of structures, the counters used, the declaration of the mutex, and the flag for thread termination.



```c
// Declaración de los hilos
pthread_t lifetime_thread;
pthread_t command_thread;

Topic topics[MAX_TOPICS]; // Almacena los topicos creados
Client clients[MAX_USERS]; // Almacena los usuarios conectados
StoredMessage messages[MAX_MESSAGES]; // Almacena los mensajes de los topicos
int topic_count = 0;
int client_count = 0;
int message_count = 0;
pthread_mutex_t mutex; // Declaración del mutex

// Flag para la terminación de hilos
int terminate_thread = 0;

```