**Authors:**

**Zana SHABANI**

**Leutrim HAJDINI**

**Client-Server Messaging Program Using Message Queues**

This is a client-server messaging program implemented in C using message queues (message passing through processes). The program allows multiple clients to connect to a server and exchange messages.

**Instructions**

1. Compile the server and client programs separately using a C compiler (e.g., gcc).

2. Run the server program first by executing the server executable.

3. Run the client program by executing the client executable.

4. Enter a request in the client program to send it to the server.

5. The server receives the request and then proccesses it.

6. To exit the client program, enter "exit" as the request.

7. The server handles disconnections and allows other clients to connect.

**Files**

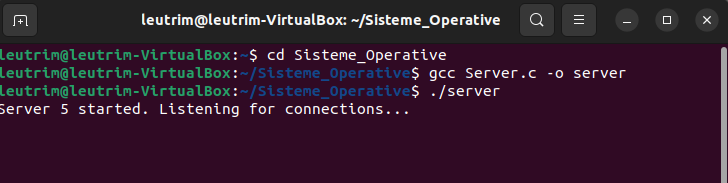
1. Client.c This file contains the implementation of the client program. It establishes a connection with the server and sends multiple requests until the user types exit to that server.

2. Server.c: This file contains the implementation of the server program. It listens for connections from clients, handles multiple client threads, and processes requests from clients.

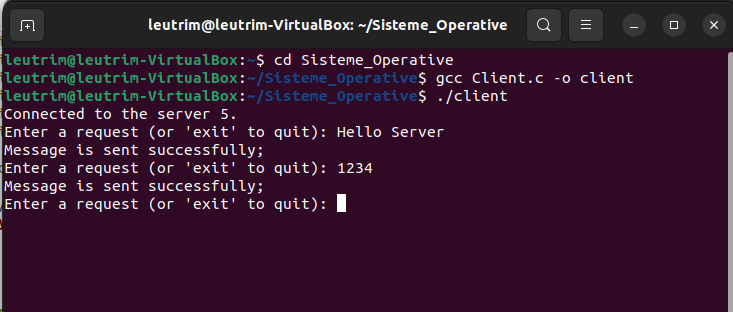
**Compilation and Execution**

Compile the server first and then the client program on the terminal on the Ubuntu OS:

1. gcc Server.c -o server

./server

2. gcc Client.c -o client

./client

3. Enter requests in the client program to send them to the server.

4. View the responses received from the server in the client program.

5. To exit type ‘exit’ on the client terminal.

**Program Flow**

1. The server program starts by creating a message queue and initializing an array of clients. It then enters a loop to listen for connections from clients.

2. When a client sends a connection request, the server accepts the connection and assigns a client queue ID to it. A new thread is created to handle communication with the client.

3. The client program starts by creating its own message queue and establishing a connection with the server by sending its client queue ID.

4. The client enters a loop where it prompts the user to enter a request or type "exit" to quit. The request is sent to the server for processing.

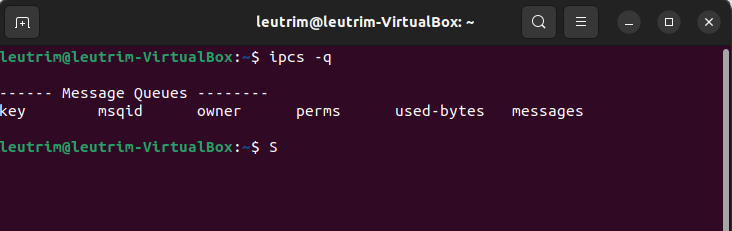
7. When the client program exits, it sends a termination message to the server to notify its disconnection.

8. The server handles the client disconnection, removes the client from the array, and terminates the corresponding thread.

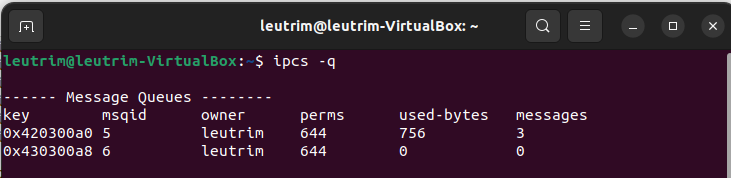
9. The server continues listening for new connections from other clients.

You can see the program flow if we type **ipcs -q** on the terminal to see the message queues.  
  
The server and the client are succesfully connected as you can see on the feedback from **ipcs -q:**

This is what is shown before the server starts:



As you can see there is nothing on the message queues.  
And this is what is shown after the client has sent some messages to the server number 5 as seen in the terminals shown when the client is executed:



So this means that with the key of the server number 5, the client has send 3 messages and can send many more if needed. This shows that the server and client are successfully connected on the message queues.

**Communication Protocol**

1. The client and server communicate using message queues.

2. The messages have a predefined structure represented by the `Message` struct, which contains a message type and a message text.

3. The client sends requests to the server by populating the message text with the request content and sending it with a message type of 1.

4. The server and client use their respective message queues to send and receive messages.

**Error Handling**

1. The programs handle various error conditions, such as failure to create message queues, message sending/receiving errors, and disconnections.

2. Error messages are displayed when errors occur, and the programs exit gracefully.

**Additional Notes**

- The programs assume a maximum limit of 10 clients. If this limit is reached, new connections will be rejected.

- The programs utilize POSIX threads (`pthread`) to handle multiple client connections concurrently.

- The client and server programs use the `ftok` function to generate unique message queue keys based on file paths.

- The server program utilizes a mutex (`pthread\_mutex\_t`) to synchronize access to the shared client array.

**Compatibility**

The program is written in C and should be compatible with most C compilers and operating systems that support POSIX message queues and threads.

Please make sure to compile and run the programs on a compatible system (this program of ours has only been tested on Ubuntu OS).