Universidad Carlos III

Curso Sistemas Distribuidos

Curso 2022-23

Práctica final

GRUPO: **80**

Rodrigo Valderrey Tarrero - 100451271

Carlos Sánchez Arroyo - 100451282

ÍNDICE

1. Introducción

En este documento se detalla la implementación realizada en la práctica final del grupo. Para el funcionamiento del servidor, se explican las funcionalidades implementadas en cada una de las funciones, y cómo estas se conectan entre sí, además de las estructuras utilizadas en el servidor. En cuanto al cliente, se explica el funcionamiento implementado en cada posible acción a realizar, y las funciones que se han creado para enviar/ recibir mensajes. Tras todo esto, se explicará la manera en la que se conectan servidor y cliente, es decir, los sockets empleados y su fin.

Una vez explicado todo lo relacionado con la estructura del código, se expondrá la manera en la que se ha implementado el código multihilo, y las como se han evitado posibles condiciones de carrera, utilizando cerrojos para bloquear la modificación de estas zonas.

Finalmente, se explicarán los pasos que se han realizado para compilar y ejecutar el código, y se indicarán las pruebas realizadas durante las ejecuciones para asegurar el correcto funcionamiento de todas las funcionalidades.

2. Consideraciones en la implementación

Previo a la explicación del contenido de la práctica, se exponen las consideraciones que el grupo ha efectuado en cuanto al enunciado disponible. Estas, han modificado el enfoque del grupo en cuanto a realizar varias acciones, que se explican más adelante.

En cuanto a la información de los clientes y los mensajes almacenados en el servidor, en un principio se pensó en tenerlos en archivos físicos (por ejemplo .JSON). Sin embargo, para una implementación más sencilla en el servidor y que se entienda con mayor claridad, se ha supuesto que el servidor nunca deja de estar conectado. De ser esto así, no haría falta almacenar la información en ficheros, ya que cuando se comience a ejecutar el servidor no habrá datos anteriores que se deban recuperar. Tras aclarar esto, la información de clientes y mensajes se almacena en estructuras declaradas, las cuales forman una lista simplemente enlazada. De esta manera se puede acceder a todas estas estructuras de una forma sencilla (su implementación se detalla más adelante).

En el enunciado (concretamente en el cliente) se ha comprobado que hay funcionalidades que no se llegan a dar, debido a que el código de ayuda proporcionado ya cubre esos casos. Por ejemplo, a la hora de conectar un cliente, si el usuario no está registrado previamente debería realizarse una excepción y especificarse por la salida de la interfaz del cliente. Sin embargo, antes de realizar la operación ya se comprueba si está o no registrado y aparece una pestaña indicando que el usuario está sin registrar. Debido a esto, existen casos de error que no hemos podido llegar a comprobar tal cual se describen en la memoria, pero no llegan a dar error por estas implementaciones ya realizadas. Ya que el enunciado especificaba la salida que debía tener por la interfaz cada uno de estos casos, se ha realizado el código que se encarga de mostrar todas las salidas correspondientes, aunque varios de estos casos nunca vayan a ser ejecutados.

Se considera que un usuario no se registrará dos veces en la misma interfaz. Esta consideración se comentó con el profesorado y fue aprobada.

Para la implementación del cliente, se han encontrado algunos apartados del enunciado que eran contradictorios entre sí. Todos estos apartados son los que indican una salida con un formato especificado en la interfaz. Hay algunos casos en los que, en el apartado del cliente se menciona un formato que, aunque es parecido, no se corresponde con el formato indicado en el servidor. Es por esto que, se ha decidido explicar en este apartado las decisiones tomadas por el grupo para cada una de las salidas. A continuación se indica el formato escogido de los existentes en el enunciado para cada una de las posibles salidas.

- ***Registrar:***

  - En la salida de la izquierda se mostrará: `c> REGISTER <userName>`

  En la salida de la derecha habrá diferentes salidas dependiendo del resultado de la operación. Los diferentes resultados son:

  -OK: `s> REGISTER OK`

  -Si el usuario ya se encuentra registrado: `s> USERNAME IN USE`

  -Si se produce fallo por otro motivo: `s> REGISTER FAIL`

- ***Desregistrar:***

  -En la salida de la izquierda se mostrará: `c> UNREGISTER <userName>`

  En la salida de la derecha habrá diferentes salidas dependiendo del resultado de la operación. Los diferentes resultados son:

  -OK: `s> UNREGISTER OK`

  -Si el usuario no existe: `s> USER DOES NOT EXIST`

  -Si se produce fallo por otro motivo: `s> UNREGISTER FAIL`

- ***Connect:***

  -En la salida de la izquierda se mostrará: `c> CONNECT <userName>`

  En la salida de la derecha habrá diferentes salidas dependiendo del resultado de la operación. Los diferentes resultados son:

  -OK: `s> CONNECT OK`

  -Si el usuario no existe: `s> CONNECT FAIL, USER DOES NOT EXIST`

  -Si el usuario ya está conectado: `s> USER ALREADY CONNECTED`

  -Si se produce fallo por otro motivo: `s> CONNECT FAIL`

- ***Disconnect:***

-En la salida de la izquierda se mostrará: `c> DISCONNECT <userName>`

En la salida de la derecha habrá diferentes salidas dependiendo del resultado de la operación. Los diferentes resultados son:

-OK: `s> DISCONNECT OK`

-Si el usuario no existe: `s> DISCONNECT FAIL / USER DOES NOT EXIST`

-Si el usuario no está conectado: `s> DISCONNECT FAIL / USER NOT CONNECTED`

-Si se produce fallo por otro motivo: `s> DISCONNECT FAIL`

- ***Send:***

-En la salida de la izquierda se mostrará: `c> SEND <userName> <message>`

En la salida de la derecha habrá diferentes salidas dependiendo del resultado de la operación. Los diferentes resultados son:

-OK: `s> SEND OK - MESSAGE <id>`

-Si el usuario al que se ha enviado no existe: `s> SEND FAIL / USER DOES NOT EXIST`

-Si se produce fallo por otro motivo: `s> SEND FAIL`

-Si llega un ACK desde el servidor de un mensaje ya enviado: `s> SEND MESSAGE <id> OK`

-Si le llega un mensaje al cliente:

```
s> MESSAGE <id> FROM <userName>
<message>
END
```

- ***Connected users:***

-En la salida de la izquierda se mostrará: `c> CONNECTEDUSERS`

En la salida de la derecha habrá diferentes salidas dependiendo del resultado de la operación. Los diferentes resultados son:

-Si no hay error: `s> CONNECTED USERS (N users connected) OK - <user1>, <user2>, ... <userN>`

-Si el usuario no está conectado: `s> CONNECTED USERS FAIL / USER IS NOT CONNECTED`

-Si llega un ACK desde el servidor de un mensaje ya enviado: `s> CONNECTED USERS FAIL`

Además de todo esto, en las funcionalidades implementadas, se ha añadido algún otro valor que se envía al cliente/ servidor de los especificados en el enunciado. Por ejemplo en el caso de connected users, el servidor debe comprobar que el usuario esté conectado para continuar con la funcionalidad, pero no se indica que el cliente deba enviar su alias. En este caso lo enviamos para que pueda conocer el estado (conectado o desconectado) sin problema, y al igual que en este caso se han añadido algunos valores en los envíos en unas pocas funcionalidades más.

3. Explicación del código

a. Cliente

i. Funciones de envío/ escucha

Envío de mensajes:

Para desarrollar un código más refactorizado, el grupo ha creado una función llamada "enviar_mensaje". Gracias a esta función se cubren todas las posibilidades de un envío. Si se proporciona la lista de mensajes que se quiere enviar, y ciertos parámetros para especificar si se trata de una operación "Send" o "ConnectedUsers" (en cuyo caso se debe proceder de otra forma) internamente la función genera toda la casuística. De esta forma generamos un código más ordenado en las funcionalidades de la clase cliente evitando repetir secciones de código. Podemos diferenciar tres casos:

- Send: Si se quiere hacer un "send", el cuarto mensaje se envía previamente al servidor web para eliminar los espacios. Además, la recepción de mensajes es diferente, ya que no solo debe recogerse el código de error, sino que también debe recogerse el id del mensaje. Se devuelven ambos valores y acaba la ejecución del código.
- ConnectedUsers: Si tiene lugar esta situación, se trata de una forma diferenciada la recepción de mensajes. En caso de que se tenga un error simplemente se imprime la situación por pantalla y se devuelve. En caso de éxito primero se captura el número de usuarios y después se establece un bucle que espera recibir tantos mensajes como usuarios conectados.
- Otros casos: El resto de casos es generalizable debido a que la recepción de mensajes es la misma. Tan solo se captura un código de error para notificar del éxito o error en la operación.

Dada esta explicación, se considera que no es relevante dar más detalles sobre cómo se gestiona el envío de mensajes por parte del cliente.

Hilo de escucha:

Como se explicará más adelante, el programa lanza un hilo de escucha al conectarse. Este hilo sirve para recibir mensajes de otros usuarios y para recoger los ACK se envía el servidor indicando que se envió el mensaje al cliente receptor. La estructura de la función es la siguiente:

Se tiene una sentencia try para controlar la posibilidad de que el hilo no deba seguir ejecutándose. En caso de que deba acabar su ejecución, se cierra el socket y no se retorna nada.

En otro caso, se debe seguir escuchando y se recibe un mensaje que indica si la información recogida se trata de un mensaje o un ACK. Si se trata de un ACK, se recoge el id del mensaje y se muestra esta información por pantalla siguiendo el estándar indicado en las especificaciones.

Si se trata de un mensaje se procede obteniendo más datos (emisor, id, mensaje) y se muestra como corresponde.

ii. Funcionalidades implementadas

Para implementar las funcionalidades descritas en el cliente, se han completado las acciones a realizar cuando se pulsan los diferentes botones de la interfaz. Esto se ha hecho en las funciones que ya se encontraban declaradas para este fin en el código inicial proporcionado.

Funciones: register, unregister.

Se envía la operación seleccionada. Si se trata de "register", también se envían los datos introducidos del usuario que se va a registrar para que sean almacenados. Para el caso de "unregister", se envía solamente el alias. Tras esto se comprueba el error y se muestra el feedback correspondiente del resultado por la interfaz.

Función connect:

En este caso, se adquiere la variable global "cortar_hilo" que permite gestionar el control sobre la escucha. Asimismo, se comienza la ejecución de otro hilo para activar la permanente escucha del cliente. Esta función altera su valor en función del error recogido, ya que si hay un error debido a que el usuario ya estaba conectado no se debe dejar de escuchar, pero en otro caso si.

Función disconnect:

El envío de la información es similar al de los otros casos. El único punto que especial que debe ser mencionado es la finalización del hilo de escucha, que es controlada al establecer "cortar_hilo" a True.

Función Send:

El envío de los datos en esta función no supone un trato muy diferente, y la recogida de la respuesta se ha tratado en el apartado destinado a "enviar_mensaje". La recepción del ACK asociado a esta operación se ha explicado en el apartado destinado al hilo de escucha.

Función ConnectedUsers:

Esta función envía la operación como en los otros casos (se recuerda que el grupo envía el alias por motivos previamente mencionados con la aprobación de la docencia). La captura de la información de respuesta ha sido explicada anteriormente.

b. Servidor

i. Estructuras implementadas

Las estructuras empleadas para el desarrollo de la práctica se encuentran en "lista_server.h". Se explican a continuación:

- Lista_mensajes: estructura que guarda el puntero al primer nodo del conjunto de mensajes. Cada usuario tiene una lista asociada para poder almacenar los mensajes pendientes.
- Nodo_mensajes: estructura que permite almacenar contador, mensaje, receptor y emisor, además de un puntero al siguiente nodo. Cuando un mensaje se envía, se libera la memoria del nodo que lo contenía.
- Cliente_servidor: Guarda la información relevante asociada a un cliente en el servidor. En ella encontramos los campos: nombre del usuario (nombre); alias proporcionado (alias); fecha de nacimiento (fecha); si el usuario está conectado o desconectado, representado con 1 o 0 (estado); IP del cliente (IP); puerto del cliente (puerto); contador propio del usuario, que guarda el número de mensajes que tiene enviados (contador_user); el número de mensajes pendientes (pendientes) y por último una estructura de tipo Lista_mensajes para guardar los que tiene ese cliente aún por leer.
- Lista_cliente: guarda un puntero al primer nodo de la lista de clientes.
- Nodo: nodo de la lista de clientes que guarda un puntero al cliente y otro puntero al siguiente nodo. La base de datos del servidor funciona gracias a este tipo de nodos que guardan toda la información.
- Argumentos: Esta estructura sirve para guardar la información que reciben los hilos (salvo el hilo que envía mensajes, que tiene su propia estructura). Se guarda el descriptor del socket por el que se debe enviar la respuesta y un objeto de tipo cliente con la información que corresponda.
- Argumentos_send: Argumentos que recibe un send. Estos son el descriptor del socket, el emisor, receptor y mensaje. Estos tres últimos son char de tamaño 256 bytes y no estructuras ya que solo se guardan el mensaje estrictamente y los alias.
- Leer_si: estructura que sirve para decidir qué campos se leerán al obtener los datos del cliente.

ii. Funciones implementadas

Para facilitar la comprensión del servidor se ha dividido su funcionalidad en dos archivos con extensión ".c". Por un lado contamos con server.c que tiene un main que lee la operación a realizar. Seguidamente llama a una función (la que corresponde dependiendo del caso) que termina de leer los datos. Para ello se servirá, en todos los casos, de "leer_cliente". Esta devuelve un objeto cliente con los campos de interés. Una vez se tiene esta información, se pasa al hilo la estructura argumento y se ejecuta. Veremos a continuación el flujo de las instrucciones en función de cada caso:

- **Caso 1: Registro de un cliente:**
  Si el servidor recibe “REGISTER” tendrá lugar la lectura de argumentos y se creará un hilo que usa la función “agregar_cliente”, (“lista_server.c”). Esta agrega un nodo a la lista de clientes de forma segura evitando condiciones de carrera.
- **Caso 2: Desregistro de un cliente:**
  Si el servidor recibe “UNREGISTER” se leerán los argumentos y se creará un hilo que ejecuta “eliminar_user” (“lista_server.c”). Se elimina un nodo para un alias específico.
- **Caso 3: Conexión de un cliente:**
  Si se recibe “CONNECT” se leen los argumentos y se modifican los atributos del cliente como se solicita (ip, puerto y estado). Para ello se crea un hilo que ejecuta la función “conectar_user” (“lista_server.c”). Dentro de esta función se ejecuta “vaciar_mensajes” (“lista_server.c”) que permite enviar al hilo de escucha del cliente cada mensaje almacenado en la lista que le corresponde. Si no tuviese ninguno no se envía.

- **Caso 4: Desconexión de un cliente:**
  Si se recibe “DISCONNECT” se leen los argumentos y se modifican los atributos del cliente como se solicita (ip, puerto y estado, dejando los dos primeros vacíos y el último a 0). Para ello se crea un hilo que ejecuta la función “desconectar_user” (“lista_server.c”).

- **Caso 5: Send:**
  Este es el caso más complejo. Como siempre, se recibe el resto de información y se crea un hilo que ejecuta “tomar_decision_mensaje”. En este momento se decide si almacenar el mensaje mediante la función “agregar_mensaje_a_cliente” o se envía directamente haciendo uso de “enviar_mensaje_conectado”. Esta última será llamada siempre que se decida finalmente enviar un mensaje (el otro caso es durante el vaciado de mensajes que se realiza en la conexión). Si se envía un mensaje (o cuando se envíe, si es que se envía durante el vaciado) salta la función “enviar_ack_mensaje”. Gracias a esta el cliente emisor recibe la confirmación de que el envío del mensaje sucedió. Todos las funciones mencionadas en este apartado corresponden a “lista_server.c”.

- **Caso 6: ConnectedUsers:**
  Si tiene lugar esta petición, se lee lo que corresponde y se comienza la ejecución de un hilo que realiza “efectuar_connect_users”. Esta función es solo de lectura y no tiene ningún mecanismo de cerrojo. En primer lugar se comprueba si el usuario está registrado y si está conectado mediante funciones auxiliares cuyo único fin es refactorizar estas comprobaciones. Si se debe notificar al cliente de algún error por estas situaciones, se envía la notificación. Si todo es correcto se envía un 0 indicando que se procede a enviar el resto de información. Después, se cuantifican los usuarios conectados y se envía el contador. Por último se envía uno a uno cada usuario conectado.

c. Conexión cliente - servidor

Para la conexión entre el cliente y el servidor ha tenido lugar la creación de los siguientes sockets:

- Socket número 1: este socket permite establecer una primera conexión con el cliente. Posteriormente se irán creando más para ir atendiendo cada petición.
- Socket número 2: Este es el que se crea para cada petición, que es recogida y enviada a un hilo. Cada hilo recoge el descriptor de este socket y envía la respuesta que ha generado.
- Socket número 3: Generado al enviar un mensaje. Se conecta al hilo de escucha del receptor del mensaje y lo transmite.
- Socket número 4: Este se crea para enviar el ack al hilo de escucha del emisor. Gracias a él se envía la información que verifica la entrega del mensaje.

d. Servidor web

Nota importante: para el correcto funcionamiento del servidor web debe especificarse la ip en la variable de entorno "ip_servidor_web". Este punto se explica con más detalle en el apartado de ejecución y compilación.

Para implementar el servidor web descrito, se ha creado otro fichero de código en python, el cual se encargará de transformar los mensajes que le lleguen de los clientes. En este fichero, se describe el objeto "Application()" correspondiente, el cual contiene la clase "Quitar_espacios()" (la cual se define al inicio, y contiene una función que elimina los espacios repetidos). Con este objeto se crea un nuevo objeto de la clase "WsgiApplication()". Tras todo esto, se describe la función principal, la cual define el servidor con el objeto creado anteriormente, el puerto y la IP. Finalmente, se inicializa el servidor para escuchar los mensajes que le manden los clientes.

Para acceder a este servidor web, los clientes tienen en su código la función "Transformar_mensaje()", la cual se encarga de enviar el mensaje a la dirección especificada del servidor y ejecutar la función del servidor para eliminar espacios repetidos. Esta función se ejecuta antes de enviar un mensaje, para que se envíe en el formato correcto al servidor.

Se debe tener en cuenta que se reserva el puerto 8000 para que sea por el que el servicio web realiza la escucha de las peticiones de los clientes. Por esto, no puede coincidir con el puerto de escucha del servidor en caso de que ambos se ejecuten en el mismo dispositivo.

4. Paralelización

El código desarrollado ha recibido una implementación paralela. Para ello se ha incorporado un sistema de control de la concurrencia mediante sistemas de exclusión mutua.

Los hilos se crean como se ha indicado anteriormente, justo después de leer los argumentos, cuando la función solicitada debe ejecutarse. La incorporación de un único mutex "mutex_mensaje" ha permitido controlar las secciones críticas. Este mutex tan solo es llamado en operaciones de escritura, y es por ello que al recorrer bucles simplemente para leer o en operaciones atómicas no se hace uso de pthread_mutex_lock() ni del consiguiente pthread_mutex_unlock().

5. Compilación y ejecución

Para la correcta ejecución del programa se deben seguir los siguientes pasos:

1. Abrir un número de terminales superior en dos unidades al número de clientes deseados. Estas dos terminales extras serán destinadas para el servidor y el servidor_web.
2. Especificar en todas las terminales de los clientes el valor de la variable de entorno "ip_servidor_web". El grupo ha probado a otorgar el valor de "localhost" y de la dirección IP asignada a nuestros dispositivos en nuestra red local. En ambos casos el resultado fue satisfactorio.
3. Ejecutar make. De esta forma se generará el ejecutable que se requiere para el servidor.
4. En la terminal que se quiere destinar al servidor ejecutar "./ server -p <puerto>". Se sugiere utilizar algún puerto estrictamente superior a 1023 para evitar que esté reservado, usando de esta forma un puerto efímero.
5. En la terminal del servidor web debe ejecutarse "python3 ./servidor_web.py"
6. En cada terminal de cliente se debe ejecutar "python3 ./client.py -s <ip> -p <puerto>" donde <puerto> debe ser el mismo especificado en el paso 4.

Nota importante: Se recuerda que es importante no pulsar dos veces en registrar para evitar problemas. Suponer que el usuario no efectuaría esta acción fue una decisión aprobada por el profesorado.

6. Pruebas realizadas

Para la correcta comprobación de los casos se ha decidido evaluar cada posibilidad que el grupo ha considerado posible. Tan solo se han considerado casos que haya tenido que implementar el grupo dado que la interfaz proporcionada ya contemplaba algunos:

Descripción de la	**Resultado esperado**	**Resultado obtenido**	**Observaciones**

situación			
Registrar un usuario: registrar un usuario con los datos introducidos correctamente. Nombre: Paco Alias: Paco Fecha 16/05/2023	c> REGISTER Paco s> REGISTER OK	c> REGISTER Paco s> REGISTER OK	Todo sucedió correctamente.
Registrar un usuario: registrar un usuario con el mismo alias que otro ya existente.	c> REGISTER a s> USERNAME IN USE	c> REGISTER a s> USERNAME IN USE	Se impide el registro como se esperaba.
Conectar un usuario: Conectar un usuario estando registrado.	c> CONNECT Paco s> CONNECT OK	c> CONNECT Paco s> CONNECT OK	Se conecta un usuario correctamente registrado y se tiene éxito.
Conectar un usuario que ya está conectado: Conectar un usuario estando ya conectado.	c> CONNECT Paco s> USER ALREADY CONNECTED	c> CONNECT Paco s> USER ALREADY CONNECTED	Se deniega la solicitud dado que ya estaba en ese estado. Todo es correcto.
Conectar un usuario no registrado: Conectar un usuario sin estar registrado.	c> CONNECT Paco s> CONNECT FAIL, USER DOES NOT EXIST	c> CONNECT Paco s> CONNECT FAIL, USER DOES NOT EXIST	Para esto, se debe registrar un usuario, dar de baja y luego intentar conectarlo. Todo es correcto.
Desregistrar un usuario registrado: Se registra un usuario y se da de baja	c> UNREGISTER Paco s> UNREGISTER OK	c> UNREGISTER Paco s> UNREGISTER OK	Se da de baja con éxito
Desregistrar un usuario no registrado: Se trata de borrar de la base de datos un cliente.	c> UNREGISTER Paco s> USER DOES NOT EXIST	c> UNREGISTER Paco s> USER DOES NOT EXIST	Se impide el proceso. No se puede dar de baja si no está en la base de datos.
Desconectar un usuario conectado: Se trata de desconectar un usuario cuyo estado es: “conectado”.	c> DISCONNECT Paco s> DISCONNECT OK	c> DISCONNECT Paco s> DISCONNECT OK	Se desconecta exitosamente.

Desconectar un usuario que ya está desconectado: Se desconecta el usuario estando desconectado.	c> DISCONNECT a s> DISCONNECT FAIL / USER NOT CONNECTED	c> DISCONNECT a s> DISCONNECT FAIL / USER NOT CONNECTED	No se permite desconectar un usuario desconectado.
Desconectar un usuario que no existe: Se desconecta el usuario no perteneciente a la BBDD.	c> DISCONNECT a s> DISCONNECT FAIL / USER NOT CONNECTED	c> DISCONNECT a s> DISCONNECT FAIL / USER NOT CONNECTED	No se puede cambiar el estado de un cliente inexistente.
Enviar un mensaje desde a cuando a no está conectado: Se intenta enviar sin estar conectado.	c> SEND a Hola s> SEND FAIL	c> SEND a Hola s> SEND FAIL	Hay un fallo debido a que el usuario no existe. Todo es correcto
Enviar un mensaje desde a, hasta b cuando b está conectado:	Interfaz de a: c> SEND a hola s> SEND OK - MESSAGE 1 s> SEND MESSAGE 1 OK Interfaz de b: s> MESSAGE 1 FROM b hola END	Interfaz de a: c> SEND a hola s> SEND OK - MESSAGE 1 s> SEND MESSAGE 1 OK Interfaz de b: s> MESSAGE 1 FROM b hola END	Llegan con éxito todos los mensajes, y el ACK. La prueba es un éxito y se comunica adecuadamente.
Enviar un mensaje desde “a”, hasta “b” cuando “b” no está conectado: Se envía un mensaje cuando el usuario aún no está conectado	**Antes de conectarlo:** Interfaz de a: SEND OK - MESSAGE 2 Interfaz de b: -sin cambios- **Después de conectar b:** Interfaz de a: s> SEND MESSAGE 2 OK Interfaz de b: c> CONNECT b s> CONNECT OK s> MESSAGE 1 FROM a hola END	**Antes de conectarlo:** Interfaz de a: SEND OK - MESSAGE 2 b: -sin cambios- **Después de conectar b:** Interfaz de a: s> SEND MESSAGE 2 OK Interfaz de b: c> CONNECT b s> CONNECT OK s> MESSAGE 1 FROM a hola END	Todo ha sucedido con éxito. El mensaje quedó almacenado y se envió al destinatario. El ack por su parte también llegó.

<u>Sometemos al destinatario a 20 mensajes pendientes:</u> Enviamos desde “a” hasta “b” 20 mensajes sin estar conectado “b”. Estos 20 mensajes deberán ser recibidos nada más se conecte “b”.	antes de conectarse b: a: s> SEND OK - MESSAGE 1 … s> SEND OK - MESSAGE 20 b: -sin cambios- Se conecta b: a: c> SEND b hola s> SEND MESSAGE 1 OK … s> SEND MESSAGE 20 OK b: c> CONNECT b s> CONNECT OK s> MESSAGE 1 FROM a hola END … s> MESSAGE 20 FROM a hola END	antes de conectarse b: a: s> SEND OK - MESSAGE 1 … s> SEND OK - MESSAGE 20 b: -sin cambios- Se conecta b: a: c> SEND b hola s> SEND MESSAGE 1 OK … s> SEND MESSAGE 20 OK b: c> CONNECT b s> CONNECT OK s> MESSAGE 1 FROM a hola END … s> MESSAGE 20 FROM a hola END	Todos los mensajes se almacenaron con éxito hasta, incluso, muchos fueron acumulados.
<u>Llamamos a connectedUsers sin estar conectados</u>	c> CONNECTED USERS s> CONNECTED USERS FAIL / USER IS NOT CONNECTED	c> CONNECTED USERS s> CONNECTED USERS FAIL / USER IS NOT CONNECTED	No se permite ejecutar esta acción desconectado.
<u>Llamamos a connectedUsers estando conectados tres usuarios</u> conectamos a,b y c y ejecutamos connectedUsers	c> CONNECTEDUSERS s> CONNECTED USERS (3) OK -c, b, a	c> CONNECTEDUSERS s> CONNECTED USERS (3) OK -c, b, a	El número de clientes conectados y la lista de nombres son correctos.
<u>Llamamos a connectedUsers estando conectados dos usuarios y tres registrados</u> conectamos a,b. A c lo desconectamos	c> CONNECTEDUSERS s> CONNECTED USERS (2) OK -b, a	c> CONNECTEDUSERS s> CONNECTED USERS (2) OK -b, a	No se cuentan ni se envían usuarios no conectados.
<u>Evaluamos el servidor web:</u> Probamos a enviar la cadena “Texto con separacion” y vemos si en el otro lado se recibe sin separación.	<u>Interfaz de a:</u> c> SEND a Texto con separacion s> SEND MESSAGE 3 OK <u>Interfaz de b:</u> s> MESSAGE 3 FROM b Texto con separacion END	<u>Interfaz de a:</u> c> SEND a Texto con separacion s> SEND MESSAGE 3 OK <u>Interfaz de b:</u> s> MESSAGE 3 FROM b Texto con separacion END	Observamos que en todos lados el servidor web actúa como debería. Tanto en la consola del emisor como en la del

			receptor.

7. Conclusiones

La realización de la práctica ha permitido a los estudiantes familiarizarse con la implementación de sockets tanto en C como en Python. Destaca, especialmente, la simplicidad de la creación de este tipo de mecanismos de comunicación en Python con respecto a la más complicada elaboración de los mismos en C. Sucede lo mismo con la creación de hilos.

El grupo ha encontrado numerosas dificultades relacionadas con la correcta comunicación de los sockets. En primer lugar se deben al tamaño variable de lo que se envía y el requerimiento de que se enviase en campos separados y no concatenados como habíamos planteado al principio. Tampoco se podían enviar enteros para indicar lo que ocupaba el próximo campo. Después, se implementó un sistema de sleeps para recibir lo deseado y no más, pero creímos que se trataba de una implementación poco elegante. Finalmente se implementó un sistema de padding que pareció arrojar buenos resultados y por ello es lo que permanece.

Otro problema que encontramos fue cómo debía efectuarse el bind del cliente para el socket de escucha. Así pues optamos por hacer un bind('0.0.0.0', 0) permite enlazar el socket a todas las interfaces de red y a un puerto seleccionado automáticamente por el sistema operativo.

Por último nos enfrentamos a problemas relacionados con la lista enlazada debido a memoria que no se gestionaba adecuadamente, pero finalmente encontramos las zonas que no estábamos manipulando bien y las corregimos.