Práctica 1

Servidor IRC

Grupo 2311 - Pareja 02

Francisco Andreu Sanz

Javier Martínez Hernández

Francisco Andreu Sanz
Javier Martínez Hernández

Índice

Francisco Andreu Sanz
Javier Martínez Hernández

Introducción:

Esta práctica consiste en la implementación de un servidor implementado **en lenguaje C** que siga el protocolo **IRC** utilizando las funciones de la librería **eps-redes2 .**

Aunque el objetivo no sea un servidor que cumpla cada uno de los comandos del protocolo sí se pide que sea robusto y ejecute correctamente los comandos más comunes.

Para probar el correcto funcionamiento de nuestro servidor hemos hecho uso del cliente **XChat** el cual, evidentemente, también sigue el protocolo según los estándares **RFCs-1459, 2812, 2813 y 2811.**

Así mismo, la manera en la que se pide que lancemos el servidor será en modo **demonio(**dejando el proceso corriendo en segundo plano**).**

Diseño:

Nuestra práctica está compuesta del módulo **main.c** principal del programa y un módulo que contiene las funciones implementadas llamado **connectServer.c** , así como su fichero de cabecera comentado siguiendo las pautas de **doxygen.**

La implementación del servidor ha sido con **sockets,** siendo específicamente orientado a conexión haciendo uso de las funciones:

- **recv(...)** para el recibo de mensajes a través de sockets.
- **send(...)** para el envío de mensajes a través de sockets.
- **socket(...), bind(...), listen(...)** para la creación, conexión y establecimiento de modo escucha de los sockets del servidor.
- **accept(...)** para confirmar el establecimiento de conexión entre socket del cliente y del servidor.

La manera con la que operamos es la siguiente:

1. Al ejecutar el programa inicializamos como vacía la lista de clientes máximos que podrá atender el servidor, así como los semáforos relativos a cada cliente.
2. Inicializamos y ponemos en modo escucha al **master_socket**(es decir, dejamos al socket servidor a la espera de recibir peticiones de sockets de clientes), y dejamos al programa *daemonizado*(en segundo plano).
3. Entramos en un bucle infinito en el que verificamos en cada uno de los sockets clientes si se produce alguna actividad. Ésto lo hacemos de forma relativamente eficaz gracias a la función **select(...),** la cual sin necesidad dividirse en hilos ó procesos comprueba si alguno de los sockets de la lista **readfds** efectúa algún cambio.

   Ésta función es muy útil puesto que sino tendríamos que tener un proceso para cada posible cliente a la espera de un **recv()** constante.
4. Una vez que se detecta un cambio en el **master socket** significa que **un nuevo cliente desea conectarse al servidor,** luego lo primero que hacemos es agregar al cliente a la lista de sockets.
5. En caso de que el cambio se detecte en alguno de los sockets cliente, creamos una estructura con datos como el ID del descriptor del socket, la posición en el array de sockets, o la dirección del mismo. Ésta estructura la enviaremos a **un hilo** que trabajará con la petición de dicho cliente. Para asegurarnos de que **sólo una petición del mismo cliente pueda ser atendida,** hacemos uso de un **mutex para cada cliente.** Dicho semáforo hará un *down* justo antes de lanzarse el hilo y el *up* nada más finalizar la petición del cliente.
6. De ésta forma, no tendremos el problema de que distintos hilos ejecuten labores de un mismo cliente, lo cual podría dar lugar a problemas, **pero sí permite que distintos hilos trabajen de forma concurrente con peticiones de distintos clientes.**
7. Cabe destacar por último que cuando un cliente se desconecta, el hilo cierra el descriptor del socket **y elimina al cliente** de la lista de usuarios. Para asegurarnos de que no se intente cerrar el mismo socket y eliminar al cliente

en distintos hilos(no debería ocurrir) hemos implementado un segundo *mutex* el cual cubre la **región crítica** de la eliminación del cliente y usuario.

El hilo que atiende cada cliente operará con la cadena que ha enviado el cliente.

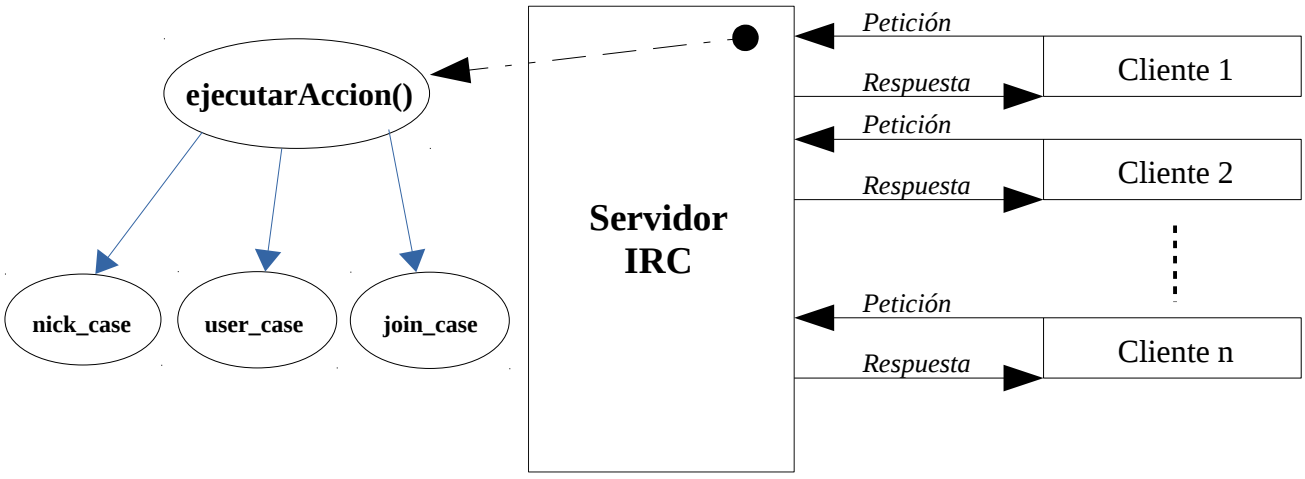
1. Para ello, lo primero que hacemos es **descomponer** el mensaje recibido en el servidor en caso de que un sólo mensaje pueda contener más de un comando(como ocurre al iniciar sesión con NICK y USER). Ésto lo logramos con **UnPipelineCommands.**

2. Una vez tenemos la cadena a operar, comprobamos de qué comando se trata con **IRC_CommandQuery.**

3. Enviamos la cadena junto con el tipo de operación a la función **ejecutarAccion(...).** Dicha función consistirá en un **switch** de los diferentes comandos. EjecutarAccion(...) se encargará de llamar a distintas subfunciones de más bajo nivel que trabajarán con cada comando implementado.

Al final el modo de operación es casi siempre el mismo independientemente del comando. **Parseamos la cadena** que nos envían dependiendo del comando que sea, **creamos un prefijo,** y elaboramos un **mensaje de respuesta** que devolveremos al cliente que nos lo envía ó a otros dependiendo del comando. Además, **algunas operaciones requieren modificaciones en la lista de usuarios ó canales.** Para ello hacemos uso de las funciones **IRCTAD_XXXX.**

Un esquema muy simplificado del funcionamiento del servidor podría ser el siguiente:



Hay que tener en cuenta que la respuesta del servidor **no siempre va a ser al mismo cliente**. De hecho, la respuesta puede ser a todos los clientes menos al que se lo envió(como ocurre con los **mensajes grupales**).

Asimismo, todos los *socket_client* estarían conectados al *master_socket(*el del servidor*).*

Por último en el diseño, mencionar que en los TAD que se nos proporcionaban, no existía ninguna relación entre **socket-usuario.** Por ello, decidimos no complicarnos y crear **dos arrays globales** para todo el programa. Los dos tienen la misma longitud y comparten índices. Uno de ellos guarda los *nicks registrados y* el otro el *socket* correspondiente a cada nick. Así, teniendo `client_socket[i],` para acceder a su nick bastaría con llamar a `nickList[i].`

Funcionalidad IRC:

Los comandos implementados han sido:

- **NICK:**
  El comando NICK tiene dos usos principales.
  1. El primero de ellos es inicializar el “nick”del usuario. Ésta es la opción que se ejecuta justo al conectar un cliente al servidor.
  2. El segundo es establecer **un nuevo nick** para ese usuario. A diferencia del otro, **este sí devolverá un mensaje de respuesta.**

- **USER:**
  Este comando se recibe tras dividir la primera cadena que envía el cliente. El primer comando a ejecutar es NICK(ya explicado arriba) y el segundo(dividido gracias a **UnPipeLine**), recibe datos como el realname, hostname, username y los añade como un nuevo usuario a la **lista** de usuarios que almacena el programa. Además devuelve un mensaje de respuesta de bienvenida.

- **JOIN:** El comando JOIN tiene a su vez dos usos.
  1. Si el target de JOIN es un canal que **no existe**, lo **crea,** establece al usuario que lo llama como CREATOR y OPERATOR y devuelve un mensaje de respuesta.
  2. En caso contrario, se une a ese canal(si fuera necesaria una clave se llamaría a **TestPassword** para ver si coinciden), y se notifica de la llegada de ese usuario a **todos los que pertenecen al canal.**

- **LIST:** Lista los canales disponibles con sus topic correspondientes. En caso de no haber **topic** definido no imprimirá nada en ese campo.

- **NAMES:** Muestra una lista de todos los usuarios visibles a un usuario determinado.

- **WHOIS:** Devuelve una serie de mensajes de respuesta mostrando distintos tipos de información relativa a un usuario como:
  - Lista de canales en los que se encuentra.
  - Realname, hostname, Username y Nick del usuario.
  - Tiempo inactivo(al no haber implementada en la librería un recuento del tiempo de cada usuario decidimos no incluir esta información).

  Además de estos también devuelve una respuesta de **inicio y end del whois.** En caso de no incluir un <*target*> se devuelve un mensaje de error.

- **WHO:** Comando que periódicamente le pide el cliente al servidor para saber la lista de usuarios disponible. Muestra información sobre clientes que cumplan una condición especificada en comando.

- **PRIVMSG:** Tiene dos funcionalidades.
  - Si el <target> comienza por '#' deducimos que es un canal y por tanto el mensaje será enviado a todos los usuarios de dicho canal.
  - Si el <target> no comienza por # entonces será un usuario, y mandamos un mensaje al mismo.
  - Si el <target> no es ninguna de las dos opciones anteriores devolvemos un mensaje de error.

- **PART:** Se sale del canal actual. Se actualiza la lista de usuarios del canal, sacando al usuario que ejecuta el comando. Se informa a los usuarios del canal con un mensaje quién se ha ido.

- **TOPIC:** Se usa para dos cosas:
  - Si no se adjunta ningún parámetro al comando, se entiende que se desea saber el topic actual.
  - En caso contrario, se establece el topic de canal **siempre y cuando el canal no sea modo TOPICOP, en cuyo caso se requiere que sea el operador el que cambie el topic.**

- **KICK:** Sirve para echar a un usuario de un canal. Adicionalmente se puede incluir un mensaje. Se informa a los usuarios del canal a quién hemos echado.

- **AWAY:** Away permite dejar a un usuario en modo inactivo, de forma que siempre que reciba un mensaje se imprimirá el argumento que se ha adjuntado con el comando AWAY.

- **QUIT:** Quit es usado por un cliente para notificar su salida del programa. Se libera ese cliente de la lista de usuarios y lo quitamos de los canales en los que estuviese. Además notificamos a los usuarios que compartían canal con él que se ha marchado.

- **MODE:** Permite establecer distintos modos a un canal. Los implementados son:
  - **+s:** Canal en modo secreto.
  - **+t:** Modo **TOPICOP.** Sólo el operador podrá cambiar el topic.
  - **/+k:** Modo con contraseña. La contraseña será pasada como parámetro. Si un usuario desea unirse deberá introducir en el JOIN la contraseña correcta.

- **MOTD:** Imprime el mensaje del día de un servidor dado por argumento. Si ningún servidor se introduce por arg.
  Imprime el del mismo.

Conclusiones técnicas:

En esta práctica hemos aprendido a trabajar con funciones externas y su correspondiente implementación en nuestro código, siendo conscientes de la importancia de una buena documentación para desarrollar cualquier aplicación.

También hemos comprendido cómo funciona un protocolo como es el de IRC, que implementamos en una aplicación **cliente-servidor** de chat.

En cuanto a conocimientos relativos a programación, hemos aprendido a **crear librerías,** documentar correctamente el código con **páginas manual de doxygen,** etcétera.

Además hemos repasado y recordado el cómo trabajar con múltiples **hilos** y saber gestionar sus secciones críticas con **semáforos.**

Conclusiones personales:

Ha sido una de las prácticas que más trabajo nos ha llevado realizar, en parte debido a su longitud en cuanto a código, pero mucho más importante, en cuanto a comprender lo que se nos pedía(la primera semana quizá no teníamos demasiado claros los conceptos). U

na vez que comprendimos el problema, la dificultad residía en saber **qué funciones** utilizar de la librería que se nos proporcionaba, además de saber qué parámetros pasarle a cada función en un determinado momento, ayudándonos del cliente **XChat** y viendo en su registro plano cómo se formaban los mensajes.

Al final el método de trabajo era rutinario, y probamos **uno a uno** los tests que requería el autocorrector con éxito.

El mayor problema lo hemos tenido cuando hemos visto(al contrario de lo que pensábamos), al correr **todos los tests juntos.** Tuvimos varios problemas(probablemente debido a *punteros locos*) puesto que el comando **QUIT** no sólo no lo pasaba, sino que bloqueaba los siguientes tests. Por ello, decidimos que lo mejor era comentarlo e ignorarlo para que al menos los siguientes sí los hiciera bien.

Por otra parte, probamos el funcionamiento con **XChat** haciendo diferentes pruebas como:

- Enviar mensaje privado a otro usuario.
- Crear un canal y que se una un usuario.
- Ejecución correcta de comando **AWAY**.
- Ejecución correcta de **PART**.
- Probamos a enviar **mensajes a un grupo y** vemos que les llega a todos(menos al que lo manda)
- Establecemos diferentes modos para un canal y comprobamos que funcionan corectamente(por ejemplo, el de canal con contraseña o el de TOPICOP).
- Implementamos también el comando **WHO(**no se pedía**).**

- Pruebas con WHOIS.

Además, realizamos más controles de errores de los que pedía el corrector.

Por ejemplo MOTD devolverá un **ERRNOMOTD** cuando no exista el fichero *MOTD.txt* en el directorio de la práctica.