Redes de Comunicaciones II

PRÁCTICA 1: Servidor IRC

SERGIO FUENTES DE UÑA
DANIEL PERDICES BURRERO

15 de marzo de 2016

Redes de Comunicaciones II

PRÁCTICA 1: Servidor IRC

1. Objetivos de la práctica

El objetivo de la práctica es desarrollar un servidor *IRC* (*Internet Relay Chat*) que se comporte según lo descrito en los *RFC* (*Request For Comments*), principalmente en los RFC 2812 y 1459 y diversos más sobre dicho protocolo para completar la información sobre la implementación.

2. Breve descripción del protocolo IRC

El funcionamiento detallado está escrito en los mencionados RFC. Resumidamente, el protocolo consiste en un sistema de comunicación en el que los usuarios se conectan a servidores para intercambiar mensajes con otros usuarios. Cada servidor IRC se encarga de gestionar las operaciones que el cliente solicite (JOIN, NICK, NOTICE, PRIVMSG…) y de llevarlas a cabo enviando si es necesario mensajes a otros clientes aludidos por la operación. Los servidores también se conectan entre sí formando redes IRC y permitiendo a los usuarios comunicarse dentro de la misma red.

El conjunto de características implementadas en esta práctica se ciñe a la comunicación usuario-servidor, por lo que no se implementan comandos de comunicación servidor-servidor. La red IRC a la que se permite el acceso, por tanto, está limitada únicamente al servidor desarrollado y a los clientes conectados a éste.

3. Principios de diseño e implementación del proyecto

El servidor se ha codificado siguiendo la normativa de la asignatura: se compone de diversos módulos organizados mediante un diseño estructurado en librerías y por capas de abstracción, implementando desde un módulo de TCP/IP para el manejo de sockets hasta un módulo de gestión de comandos IRC. A continuación se detalla el diseño de todos los módulos utilizados para esta práctica (y funcionalidades futuras para otras prácticas).

3.1. Módulo TCP

Se implementa una librería de alto nivel de manejo de sockets, la cual básicamente se centra en dos objetivos:

- Crear una API más simplificada respecto a la que provee el sistema operativo, agrupar funciones y ocultar ciertas estructuras restringiéndolas al envío y recepción de mensajes TCP.
- Controlar tanto la robustez del módulo ante argumentos inválidos como la gestión clara y precisa de errores, simplificando así la utilización y depuración de estas funciones en módulos superiores.

Se pretende además dar la mayor portabilidad y autonomía posible al módulo con el fin de poder incorporarlo a futuras prácticas o proyectos personales.

3.2. Módulo *Server*

Se pretende con esta parte de la librería abstraer el concepto de servidor de hilos al máximo nivel, haciéndolo totalmente independiente de su funcionalidad a nivel de aplicación (i.e.: pueda emplearse genéricamente para implementar un servidor de *echo*, un servidor de *ping* o un servidor IRC). Nótese la diferencia con el módulo anterior: no se pretende encapsular funcionalidad del SO, sino proveer funciones básicas para crear un servidor de hilos para cualquier propósito y de manera transparente.

Al ser un módulo más complejo, aparecen modelos de implementación distintos:

A. Servidor mono-hilo: El planteamiento más sencillo. Todos los mensajes son atendidos de manera secuencial (uno tras otro), por lo que no requiere el uso de mecanismos IPC (semáforos y otros mecanismos de protección de los recursos compartidos). El rendimiento es bastante pobre y limita la concurrencia.
B. Servidor multi-hilo con un hilo por cliente: Otra implementación sencilla que lanza un hilo de atención por cada cliente, necesitando así IPC y mecanismos de protección (*mutex*). No requiere sin embargo el máximo nivel de sincronización pues existe la limitación de que no se pueden atender dos mensajes del mismo cliente al mismo tiempo. En cuanto a rendimiento esta opción es bastante aceptable, si bien puede verse reducido cuando los clientes envían muchos mensajes consecutivamente, aumentando además el uso de recursos (los asociados por hilo): puede pasar mucho tiempo sin requerir una atención y por tanto ocupar espacio en las tablas del sistema operativo (vulnerabilidad frente ataques DoS [*Denial of Service*]).
C. Servidor multi-hilo con un hilo por mensaje (utilizando *select*): Esta opción se basa en utilizar un hilo por mensaje: cada hilo procesa el mensaje y se destruye al terminar, es decir, una *pool* de hilos (en principio ilimitada) que atienden los mensajes. Esto originaría la ventaja de poder atender varios mensajes del mismo cliente en paralelo (sin embargo, está opción está limitada por razones estructurales y se bloquea un socket cuando un hilo ya lo está atendiendo). Por ello, en rendimiento es comparable y en muchos casos superior al modelo anterior (procesos de atención más cortos). En consumo de recursos, es prácticamente óptima pues se mantienen estructuras en memoria sólo del número de hilos ejecutándose en cada momento y no del número de clientes (mayor robustez ante ataques DoS [*Denial of Service*]).
D. Servidor con *pool* de procesos multi-hilo (por mensaje): El modelo más complejo y eficiente. Consistiría en crear una *pool* (limitada) de procesos que a su vez reciben conexiones y lanzan hilos de atención por cada mensaje (*pool* de hilos).

Se ha decidido implementar la opción C por optimización de rendimiento y uso de recursos respecto a los modelos A y B, y por falta de tiempo para perfeccionar el sistema D –no se descarta usar esta última en el futuro–.

El funcionamiento es bastante sencillo: el proceso principal (que inicializa el servidor) se encarga de atender conexiones entrantes y lanzar hilos de atención. Por decisión de diseño no se lanza un hilo distinto del principal para atender conexiones entrantes, pues el servidor desarrollado en estas prácticas espera un número moderado de conexiones y esta optimización no es necesaria.

Cada hilo de atención requiere para ser llamado por el nivel superior dos funciones que se pasan como argumento:

- *handler*: función que atiende cada mensaje. Recibe los argumentos de la conexión (sockets) y estructuras auxiliares que se pueden pasar como punteros genéricos (*void* *).
- *do_on_disconnect*: función que recibe los mismos argumentos y atiende el cierre de una conexión TCP. Es de gran utilidad al manejar desconexiones abruptas, para evitar SIGPIPE (*Broken Pipes*) y otros errores de escritura/lectura en sockets cerrados.

Se incluyen funciones de bloqueo/desbloqueo de sockets (habilitando lectura concurrente del mismo socket), así como funciones para añadir o borrar de manera artificial conexiones que están siendo vigiladas por *select* para dar soporte completo en la librería.
Para más detalle, se incluye documentación *man* y comentarios en el código.

3.3. Módulo del servidor IRC

Sobre el módulo anteriormente descrito se crea un servidor IRC que emplea función de atención (*handler*) a la que llegan mensajes de protocolo IRC separados por "\r\n" (CR-LF), por lo que hay que separarlos antes de procesarlos.

Cada comando se procesa de manera prácticamente idéntica:

1. Tras bloquear la lectura del socket, se identifica el comando con ayuda de la librería.
2. Se accede a la rutina correspondiente de la librería de comandos IRC (descrita en el siguiente apartado).
3. La rutina de atención hace las operaciones oportunas y termina.
4. Se desbloquea la lectura del socket y se destruye el hilo.

Además de esto, periódicamente se envía un comando PING y se espera respuesta de cada cliente. En caso de no recibir PONG de algún cliente en un intervalo de tiempo determinado, dicho cliente es desconectado.

Como comentario sobre la implementación, los comandos se guardan en una tabla indexada por el identificador numérico del comando (similar a una tabla *Hash* con acceso aleatorio), evitando así la búsqueda lineal. La función *nocommand* del módulo siguiente se utiliza como caso por defecto para funcionalidad no contemplada.

Para mayor detalle del API y la implementación se puede consultar el código y la documentación *man* entregada.

3.4. Módulo IRC de atención a comandos

Se implementa la lógica de atención y envío de respuestas necesarios para cada comando de acuerdo a los requisitos definidos en los RFC.

El funcionamiento es el siguiente:

1. *Parsear* el comando recibido.
2. Lógica de control.
3. Construcción del mensaje de respuesta.
4. Envío del mensaje de respuesta.

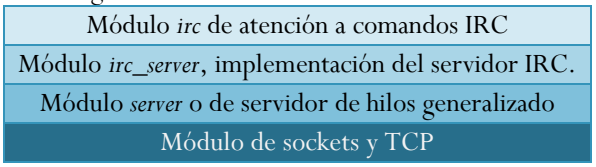
Se ha codificado una selección de comandos lo más amplia posible para proveer una correcta funcionalidad de acuerdo con el límite de tiempo para desarrollar el servidor.

3.5. Otros módulos:

Se incluyen además otros módulos parte de la librería de cara a facilitar el futuro desarrollo:

- Módulo de demonios: la función *daemonize* que convierte el proceso que la llama en demonio del sistema.
- Módulo de herramientas (*tools*): funciones de propósito general, por ejemplo, sobrecargas de la librería del sistema *string.h* que toleran punteros a NULL y similares.
- Módulo UDP: en construcción y de funcionalidad limitada.

En resumen, se ha implementado la siguiente estructura de módulos:



Como siempre en el desarrollo de software, el diseño estructurado y modular facilita al desarrollador la posibilidad de actualizar, extender o modificar el funcionamiento interno de cada módulo de manera independiente. Por ejemplo, el mejorar el sistema para emplear el modelo de *pool* procesos-hilos descrito anteriormente para lograr un mejor rendimiento o añadir una librería de atención a comandos IRC avanzados.

4. Problemas encontrados

Los problemas se han centrado principalmente en torno a tres factores:

1. Comprensión del protocolo IRC a través de los RFC: siendo el objetivo de la práctica el enfrentarse a los documentos técnicos de descripción de protocolos de telecomunicaciones parece normal el haber tenido que leer los RFC en detalle.
2. Utilización de la librería proporcionada y del corrector: aunque pueda parecer que se pretende decir que la librería ha sido un obstáculo, lo que realmente expresa es la dificultad de mantener el código, documentación y pruebas así como comunicar de manera eficiente y precisa el funcionamiento de una API, lo cual no solo se aprecia en la librería de la que disponíamos (y cuyo uso ha tenido una curva de aprendizaje) sino también en nuestro código, que a primera vista, puede suponer difícil de usar o comprender. Mencionar en esta parte que actualizar el corrector y/o la librería a pocos días de la entrega sí que lo hemos considerado perjudicial, pues pese a tener fallos localizados y con solución, se debe dar de cara a la entrega y evaluación un criterio bien definido del entorno de ejecución, facilitando a los alumnos las pruebas y la entrega, evitándonos a todos cambios de última hora.
3. Errores de codificación: este apartado es el más obvio de todos y se podría considerar implícito al concepto de práctica. Estos errores han variado en intensidad y persistencia, sin embargo, utilizando las herramientas adecuadas de depuración, se ha logrado enfrentarse a esta problemática.

De manera autocrítica, hay que evaluar el trabajo realizado, y lo primero que hay que advertir es que es un proyecto de ámbito didáctico, luego precisamente en muchos entornos es desaconsejable hasta que no cumpla unos especificaciones más robustas y extendidas acordes con el uso industrial de un producto. Sin embargo, como pequeño proyecto para ámbitos menos exigentes se podría decir que tenemos una alternativa a grandes proyectos que nos proporciona un rendimiento, funcionalidad y facilidad de uso más acorde a este escenario.

El resultado obtenido es un servidor IRC de funcionalidad limitada pero con una robustez y tolerancia de errores más desarrolladas de los esperado. La razón de esto es sencilla, aunque admite discusiones, y consiste en valorar el aumentar la funcionalidad sacrificando en quizás pruebas y recuperación de errores o el reducir la funcionalidad y responder antes unos mínimos estándares de estabilidad.

5. Conclusión

En el desarrollo de la práctica se ha experimentado un entorno profesional en el que se piden una serie de objetivos y se ha de trabajar en base a documentos de especificaciones técnicas –RFC en este caso–, empleando de manera opaca librerías mantenidas por otros desarrolladores y reportando de forma adecuada los posibles *bugs* encontrados.

Asimismo, se ha hecho hincapié en la necesidad una correcta organización y modularización del código en librerías para reutilizar recursos y poder solucionar problemas en escenarios controlados. De esta manera, se hace una aproximación al método de trabajo y convenciones que se utilizan en los proyectos de desarrollo de software.