Trabajo Práctico 1

Fecha de entrega: Lunes 24 de junio

Introducción

Objetivo

El objetivo del presente trabajo práctico es implementar una servidor de IRC simplificado, haciendo uso de listas enlazadas.

El trabajo consiste en la resolución de un problema mediante el diseño y uso de TDAs y en la reutilización de algunos de los tipos, funciones y conceptos desarrollados a lo largo del curso en trabajos anteriores.

Alcances

Mediante el presente TP se busca que el estudiante adquiera y aplique conocimientos sobre los siguientes temas:

- Encapsulamiento en TDAs,
- Listas enlazadas,
- Contenedores y tablas de búsqueda,
- Punteros a funciones,
- Archivos,
- Modularización,
- Técnicas de abstracción,

además de los temas ya evaluados en trabajos anteriores.

Nota

Este trabajo es un trabajo integrador de final de cursada. En el mismo se van a aplicar todos los temas aprendidos y además se va a evaluar el diseño de la aplicación.

Es imprescindible entender los temas de TDA y modularización antes de empezar con el trabajo.

> Es imprescindible diseñar el trabajo como etapa **previa** a la implementación.
>
> Cualquier abordaje que pretenda empezar a codificar sin haber ordenado el trabajo primero está condenado a fracasar contra la complejidad del problema.

El servidor

Para estas alturas ya sabemos qué es lo que hace el servidor. El servidor recibe mensajes de clientes y según el mensaje que recibió puede decidir entre responderle al cliente que le escribió, o enviar a su vez un mensaje a un grupo seleccionado de otros clientes.

(

En el EJ4 vimos la lógica de conexión de un cliente. Un cliente abre un socket conectándolo a una determinada IP y un determinado puerto y cuando el servidor acepta esa conexión se abre un canal que se representa en un fd.

Desde el lado del servidor se usa la misma API con un algoritmo parecido pero diferente. En primer lugar el servidor abre un socket escuchando en determinada IP y en determinado puerto, ese socket es un fd. El servidor va a escuchar en ese socket hasta que se cierre. Cada cliente que intente conectarse va a generar un evento en el fd del socket. Cuando llega ese evento el servidor va a aceptar a ese cliente en particular, la acción de aceptar genera un nuevo fd que va a ser exclusivo entre la comunicación del servidor con ese cliente.

Entonces, desde el lado del servidor se abre un socket con su fd y cada vez que se conecte un nuevo cliente se generará un fd exclusivo para ese cliente. El servidor todo el tiempo tiene que estar esperando datos en todos los fds que tiene abiertos. Si el dato es en el fd del socket esto significa que se conectó un nuevo cliente, si el dato es en el fd de un cliente significa que ese cliente está hablando con el servidor.

)

Volviendo del paréntesis de los últimos tres párrafos un servidor de IRC tiene que poder atender múltiples clientes conectados en simultaneo y encargarse de darle lógica a la comunicación entre ellos.

En un servidor de IRC hay dos cosas: Usuarios y canales. Cada conexión entrante al servidor representa un usuario diferente, identificado por su nick. Luego los usuarios pueden participar o no de canales.

Todo el tiempo los usuarios están generando eventos, sea porque hablaron, porque entraron o salieron de un canal, porque pidieron información específica de algo, etc. o sea porque se desconectaron. Según el tipo de esos eventos el servidor tendrá o que actualizar el estado del usuario, o de los canales, o responderle al cliente con información, o avisarle a todos los clientes que se vean afectados con ese evento de la nueva información.

A estas alturas sabemos que el servidor, si nadie lo molesta, sólo genera un único evento por su cuenta, sin que nadie se lo pida: Cada una determinada cantidad de tiempo le envía un ping a cada uno de los usuarios. Si alguno de los usuarios responde en determinado tiempo, entonces lo desconecta del servidor.

El protocolo

No viene de más repetirlo, el protocolo de IRC es un protocolo ASCII de líneas. Cada mensaje de uno y otro lado es una línea finalizada en `'\n'`. Y además esas líneas representan campos como los que ya estamos trabajando desde el EJ2.

Cuando un cliente se conecta al servidor inicialmente es sólo un fd abierto. El servidor sólo conoce lo físico de esa conexión, como por ejemplo la IP del cliente. Entonces lo primero que se espera es que el cliente se identifique. (Dicho sea de paso: Es el cliente el que tiene que tomar la iniciativa, cuando un cliente se conecta a un servidor que habla determinado protocolo se asume que el cliente conoce ese protocolo. Si el cliente no cumple con el protocolo el servidor puede cerrarle la conexión sin dar ninguna explicación. En el servidor provisto en el EJ4 el mismo nos daba indicaciones, esas indicaciones eran de cortesía y no deberían ser necesarias.) Ya lo sabemos, la identificación se da por una secuencia del comando USER y del comando NICK (podemos asumir que vienen en ese orden, si bien no todos los clientes lo mandan en ese orden.)

Hasta que un cliente no complete enviar el comando USER y el comando NICK (y ser aceptado ese nick) no debería permitírsele enviar otro comando. (Y como se dijo "no permitírsele" puede ser ignorarlo o directamente desconectarlo, porque no conoce el protocolo.)

(

Sólo como recordatorio, no olvidarse de que el tag que representa a cada uno de los clientes en el protocolo tiene la forma `nick!~usuario@ip`, como se dijo la ip se obtiene en el momento de la conexión. Los otros parámetros son los que se obtienen de USER y NICK.

Como otro recordatorio los mensajes que el servidor le responde al usuario siempre tienen el formato `idservidor xxx nick mensaje` donde `xxx` es el código del mensaje.

)

Sin más presentamos la lista de comandos del protocolo:

USER usuario host server nombrereal:

Debería ser el primer comando a recibir. El campo usuario es el que se usa en el tag de identificación. El servidor puede no responder nada a este mensaje.

NICK nick:

Si el nick está ocupado el servidor tiene que responder con un mensaje 433.

Hay dos casos diferentes del comando NICK, cuando un usuario recién llega tiene que informar su nick como parte del proceso de conexión, pero también un usuario puede cambiarse el nick cuando quiera.

En el caso de usuario nuevo, si ya mandó previamente el comando USER tenemos que reponderle con un mensaje 001 de bienvenida. En la bienvenida vamos a decirle algo así como "Bienvenido tag", donde el tag es el que vamos a usar para referirnos a ese usuario.

En el caso de que el usuario ya esté conectado, mandaremos un mensaje `tagviejo NICK nicknuevo` a todos los usuarios que estén en canales compartidos con el usuario. (Notar que como el tag contiene el nick el mismo cambia si cambia el nick.)

Además, obviamente, tiene que actualizarse el nick de ese cliente en el servidor.

JOIN canal:

Si el nombre de canal es inválido, esto es, no comienza con `'#'` debe responderse un mensaje 476.

Si el canal no existe debe crearse y sumarse al usuario a ese canal como administrador.

Si el canal existe debe sumarse al usuario.

En ambos casos debe mandarse un mensaje de `tag JOIN canal` a todos los usuarios del canal (este paso viene después del anterior, así que esto incluye al usuario que acaba de sumarse).

Y finalmente debe enviársele al usuario la lista de usuarios del canal. La lista de usuarios es lo mismo que el usuario recibiría si invocara el comando NAMES.

LIST canal:

El servidor lista todos los canales, esta lista es una sucesión de:

Mensaje 321, para informar que empieza la lista.

Un mensaje 322 por cada canal donde el primer campo es el nombre y el segundo el topic del canal (si lo tiene).

Un mensaje 323 para informar que termina la lista.

NAMES canal:

Lista todos los usuarios en un canal.

Si el canal existe es una sucesión de mensajes 353 con contenido `canal = nick1 nick2 nick3...`. Si un usuario es administrador del canal se debe preceder su nick con `'@'`. ¿Por qué dije sucesión?, porque la línea no debería superar los 500 caracteres... puede omitirse esto y devolver todos los nicks en una sola línea.

Finalmente debe venir un mensaje 366 con el canal como mensaje que indique que ahí terminó la lista.

Si un canal no existe entonces no hay usuarios, se manda sólo el 366.

PART canal:

Saca a un usuario de un canal.

Si el canal no existe se responde un mensaje 403.

Si el usuario no está en ese canal se responde un mensaje 442.

Si todo está bien se le manda a todos los usuarios del canal un mensaje `tag PART canal` y luego se retira al usuario del canal.

(Si el usuario era administrador pierde los permisos y nadie lo reemplaza.)

PING codigo:

Simplemente se le responde al usuario con `idservidor PONG nick codigo`

PONG codigo:

Este mensaje debería ser la respuesta a un PING del servidor.

Recordemos que se dijo que el servidor todo el tiempo está acosando a los usuarios con sus pings. El servidor tiene que ser capaz de identificar para cada usuario dos cosas: cuándo fue la última vez que el usuario respondió un ping y cuándo fue la última vez que el servidor mandó un ping. En realidad tres: además cuál fue el código que le mandó en su momento.

Cuando un usuario responde al ping, y el código es el correcto, entonces se debe consignar que ese usuario está respondiendo y cuándo.

Por fuera del ciclo de escuchadas el servidor tiene que ser capaz de identificar a los usuarios que hace mucho que no se los molesta con el ping para mandarles un nuevo ping, y también debe poder identificar a los usuarios que fueron pingueados y no respondieron en un tiempo razonable para desconectarlos.

Si el código no coincide puede ignorarse o puede desconectarse al usuario.

PRIVMSG destinatario mensaje:

El destinatario puede ser un canal o un usuario (los canales arrancan por numeral).

Para canales:

Si el canal no existe mensaje 403.

Si el canal existe pero el usuario no está en ese canal mensaje 404.

Caso contrario mandarle a todos los usuarios del canal **menos** al usuario que está mandando el mensaje `tag PRIVMSG canal mensaje`.

Para usuarios:

Si el usuario no existe mensaje 401.

Caso contrario mandarle `tag PRIVMSG usuario mensaje`.

QUIT:

Desconecta al usuario del servidor.

Debe mandarle el mensaje `tag QUIT :Mandó un quit` (el mensaje es el que se desee, pero diferenciarlo de veces que usuarios se van por otros motivos) a todos los usuarios que estén en canales compartidos.

Además debe retirar al usuario de todos los canales donde participaba, pero como ya se mandó QUIT no se manda PART.

Y además debe cerrar el fd asociado al usuario.

TOPIC canal [tópico]:

(Corchetes: Optativo.) Cambia el tópico de un canal.

Si el canal no existe, es un mensaje 403.

Si el usuario no está en el canal mensaje 404.

Si el usuario no mandó el tópico se debe informar el tópico del canal y hay dos opciones:

El canal tiene tópico: Mensaje 332 con contenido `canal tópico`

El canal no tiene tópico: Mensaje 331 con contenido `canal :No hay tópico` (O el texto que prefieran.)

Si el usuario mandó tópico entonces hay que cambiárselo al canal, y se mandará el mensaje `tag TOPIC canal topico` a todos los usuarios del canal.

USERS:

Lista todos los usuarios del servidor.

La lista comienza con un mensaje 392.

Luego por cada usuario conectado (que no sea el que está preguntando) debe mandarse un mensaje 393 con contenido `nick usuario ip`.

Si no hubiera ningún usuario conectado (salvo el que pregunta) debería mandarse un mensaje 395 para indicar que no hay nadie conectado. (El servidor del EJ4 creo que tenía un bug y no lo mandaba.)

Finalmente un mensaje 394 de fin de lista de usuarios.

Esos son todos los comandos que es necesario implementar. Cuando se dice que es una versión simplificada de servidor es porque se dejaron afuera un montón de detalles que tiene un servidor real como los permisos de los canales, los privilegios de los usuarios, la posibilidad de echar o de bannear a usuarios, etc. Un servidor de IRC real tiene muchísimas más funcionalidades que estamos ignorando.

Volviendo al servidor explicamos algo que se dijo en QUIT y en PONG que no se aclaró. El mecanismo de QUIT puede ser disparado por el usuario pero también puede ser disparado por el servidor en dos casos: Si el usuario no responde al ping debe ser desconectado y lo mismo ocurre si el fd se desconecta. En ambos casos hay que avisarle a todos los usuarios de canales compartidos, sacarlo del canal, y cerrar el fd asociado.

Implementación

TDAs

Deben desarrollarse TDAs al menos para representar un usuario y para representar un canal. Todos los atributos tanto del usuario como del canal deben estar contenidos en su respectivo TDA.

Listas enlazadas

Deben utilizarse las listas enlazadas provistas por la cátedra para representar las listas de usuarios y listas de canales del servidor. No está permitido modificar la interfaz de estas listas, deben utilizarse como Bárbara.

Manejo de tiempos

Notar que la implementación del algoritmo de ping requiere manipular tiempos.

En la sección de Material de la página web se encuentra el apunte sobre manejo de fechas.

De forma simplificada, se puede obtener la fecha actual con:

```
time_t tiempo = time(NULL);
```

luego pueden compararse dos `time_t` con la función `difftime(tiempo1, tiempo0);` que devuelve la diferencia en segundos desde el `tiempo0` al `tiempo1`.

while(1)

Un servidor es un programa que está diseñado para correr en forma no interactiva por siempre. ¿Cómo se termina entonces un servidor?, la única manera es interrumpiendo el proceso, lo que hacemos cuando apretamos control + C.

En C y en Unix las excepcionalidades en los procesos se manejan con un mecanismo llamado señales. Un acceso a memoria indebida dispara una señal (SIGSEGV), una división por cero dispara una señal (SIGFPE), etc. Hay algunas señales que son fatales, como SIGSEGV, hay otras que pueden ser ignoradas, y hay señales que no pueden ser ignoradas pero que puedo intentar acomodar las cosas antes de que la aplicación se termine.

Cuando se aprieta control + C se dispara una señal que se llama SIGINT. Esta señal no puede ser ignorada, pero puedo tomarme un tiempo adicional para dejar las cosas bien. Como por ejemplo, liberar toda la memoria que tengo ocupada, cerrar archivos, despedirme de la gente, etc...

Las señales en C se manejan con `<signal.h>`. Ahí está definida la función:

```
void (*signal(int sig, void (*func)(int)))(int);
```

... y la realidad es que nunca le había prestado atención a ese prototipo hasta que lo pegué ahí, ¿devuelve un puntero a función?, qué espanto. Lo que importa es esto:

```
signal(int sig, void (*func)(int));
```

Si invocamos `signal(SIGINT, funcion_que_va_a_manejar_las_cosas_cuando_alguien_apriete_control_c);` la función esa va a ser invocada cuando alguien apriete control + C. El entero que reciba la función va a ser `SIGINT`.

Si quisiéramos liberar memoria, ¿cómo hacemos para que una función pueda liberar? No nos va a quedar otra que que las cosas que haya que liberar sean globales, así la función puede verlas.

En este TP se permite que haya una estructura global (o un TDA) que represente las cosas que tienen que liberarse al matar la aplicación. A esta estructura global sólo puede acceder el `main()` y la función de liberación. El resto de la aplicación tiene que manejarse con parámetros a funciones como corresponde.

Apertura del socket del servidor

La creación e inicialización del socket del servidor es similar a la ya vista en el comunicador:

1. Se crea el socket con los mismos parámetros.
2. Se inicializa la estructura `servaddr`.
3. Vamos a variar la inicialización de `servaddr.sin_addr.s_addr` y vamos a inicializarla en `htonl(INADDR_ANY)`, esto es básicamente decirle al servidor que escuche en 0.0.0.0, que son todas las IPs disponibles en el equipo.

Ahora bien, no vamos a hacer `connect()` porque no vamos a conectarnos a algo existente si no que queremos abrir nosotros ese canal de escucha. Para esto se utiliza `bind()` con los mismos parámetros que connect.

Y finalmente hay que ponernos a prestar atención con `listen()`.

La secuencia completa queda:

```
int sockfd;
sockfd = socket(AF_INET, SOCK_STREAM, 0);
if(sockfd == -1)
    // ERROR Creación del socket

struct sockaddr_in servaddr;
memset(&servaddr, 0, sizeof(servaddr));

servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(puerto);

if((bind(sockfd, (struct sockaddr*)&servaddr, sizeof(servaddr))) != 0)
    // ERROR No pudo conectarse el socket al puerto

if((listen(sockfd, 5)) != 0)
    // ERROR Falla al escuchar

// Servidor creado y escuchando!
```

El servidor ahora está escuchando en el fd `sockfd`.

Poll

En el EJ4 teníamos dos fuentes de datos: `stdin` y el fd del servidor, y usamos `poll()` para que nos avisara si había datos en uno de los dos.

En este caso usaremos lo mismo, pero nuestra fuente de datos serán o el fd del server o los fds de los distintos clientes que se fueron conectando.

La firma de la función `poll()` es:

```
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

Recibe un arreglo de estructuras, la cantidad de elementos en el arreglo y un timepo máximo. Devuelve la cantidad de fds que recibieron datos. Si la función fallara devolvería -1.

El arreglo es de estructuras:

```
struct pollfd {
    int   fd;         /* file descriptor */
    short events;     /* requested events */
    short revents;    /* returned events */
};
```

En cada elemento tendremos que cargar qué `fd` queremos escuchar y qué eventos `events` queremos atender. Después de llamar a `poll()` la función va a escribir en `revents` qué eventos registró para ese `fd`.

Para el fd del servidor nos interesa escuchar las conexiones entrantes, por lo tanto cargaremos el `events` en `POLLIN`.

Para los clientes nos interesa saber si el cliente nos habló o si se desconectó así que iniciaremos `events` en `POLLIN | POLLHUP | POLLRDHUP`.

Si recordamos del EJ4, cuando llamámabos a `comunicador_esperar_datos()` esta llamada era bloqueante hasta que hubiera un dato. Eso en este contexto no nos serviría, porque si nadie nos hablara no podríamos hacer nuestra ronda de acoso de pings. Para esto sirve el parámetro `timeout`. En el EJ4 el comunicador llamaba con `-1` lo que significa "esperá datos por siempre". `timeout` es el número de milisegundos que `poll()` va a esperar por un dato. Si pasamos un valor

positivo la función va a retornar o cuando haya habido un evento (en ese caso devolverá la cantidad de eventos que hubo) o hasta que se alcance el tiempo (en ese caso devolverá 0, porque no hubo eventos).

Los eventos en los fds de clientes siempre van a ser o un evento `POLLIN`, que significa que hay una línea nueva a ser leída, o `POLLRDHUP` que significa que se perdió la conexión con ese cliente. En el caso de que sea eso último hay que hacer lo que se haría en caso de un QUIT.

Aceptando nuevos clientes

Si se registra un evento de `POLLIN` en el fd del servidor entonces eso significa que hay un nuevo cliente queriéndose conectar. En ese caso hay que aceptarlo, la secuencia es la siguiente (`sockfd` es el socket del servidor, igual que antes):

```
struct sockaddr_in cli;
int fd = accept(sockfd, (struct sockaddr*)&cli, sizeof(cli));
if(fd < 0)
    // ERROR Falló aceptar al cliente

int direccion = ntohl(cli.sin_addr.s_addr);
int puerto = ntohs(cli.sin_port);
```

El nuevo cliente está escuchando en el `fd`. La dirección IP, que es un número de 32 bits está en `direccion`. Como sabemos, las IPs se imprimen en decimal en formato x.x.x.x, donde cada uno de los octetos es el valor decimal de ese byte. Ya sabemos manejo de bits a estas alturas del partido.

Cuando se acepta una conexión cliente y servidor se comunican por un nuevo puerto, este parámetro no tiene la menor importancia para este TP, pero si interesara ese valor en la última línea se ve cómo obtenerlo.

Trabajo

Ya se dijo todo lo que había por decir.

El problema que se plantea en este trabajo deberá ser resuelto con las herramientas ya adquiridas, extendiendo de forma consistente los TDAs ya planteados de ser necesario y diseñando los TDAs nuevos que hagan falta. Se evaluará además el correcto uso de los TDAs respetando la abstracción y el "modelo" de Alan-Bárbara.

A esta altura del cuatrimestre es una obviedad decirlo pero: Todo lo que pueda ser iterativo o implementarse de forma indexada, debe ser hecho de esa manera. Todo lo que amerite tablas de búsqueda debe utilizarlas.

Se pide diseñar una aplicación que pueda ser ejecutada como:

```
$ ./servidor [puerto]
```

que lance un servidor en el puerto indicado (o en el 6667 si no se indica).

El servidor debe funcionar hasta que se apriete control + C, imprimiendo tanto los mensajes entrantes como los salientes.

Se deben implementar TDAs donde se considere que es necesario.

Al menos el usuario y el canal deben ser implementados como TDAs.

El alcance es lo especificado en este enunciado. En cualquier aplicación amplia como esta siempre pueden implementarse detalles adicionales, los mismos no forman parte del enunciado y no hay que hacer nada adicional para alcanzar la máxima nota. La prioridad es resolver completa la funcionalidad descrita para terminar con eso la materia.

Material

Los fuentes de la lista enlazada de la cátedra están disponibles en la sección de Material de la página web.

Requisitos

Los únicos requisitos son que el programa resuelva correctamente la funcionalidad pedida y que lo haga utilizando TDAs con un diseño planificado a conciencia. Si hacés eso nada puede salir mal.

Entregables

Deberá entregarse:

1. El código fuente del trabajo debidamente documentado.
2. El archivo `Makefile` para compilar el proyecto.

Nota

El programa debe:

- estar programado en C,
- estar completo,
- compilar...
- sin errores...
- sin warnings,
- correr...
- sin romperse...
- sin fugar memoria...
- resolviendo el problema pedido...

Trabajos que no cumplan con lo anterior no serán corregidos.

La entrega se realiza a través del sistema de entregas.

El ejercicio es grupal permitiéndose grupos de hasta 2 integrantes.

Si elegís hacer el trabajo en grupo te pedimos que nos avises o por mail o por Discord quién es tu compañero de grupo así lo cargamos en el sistema de entregas. (Si el grupo sufriera alteraciones avisanos también.)