Ejercicio obligatorio 4

Fecha de entrega: Domingo 2 de junio

Introducción

IRC

El protocolo de Internet Relay Chat (IRC) data del año 1993 y establece un mecanismo sencillo para crear salas de chat.

El protocolo es un protocolo de texto, en el mismo hay un servidor al que los clientes se conectan y tanto los clientes como el servidor escriben y leen líneas de a líneas de texto.

La estructura de IRC es la que replican hoy en día sistemas de chat como Discord o Slack: Uno se conecta a un servidor utilizando un determinado nick. De ahí puede o enviar mensajes personales a otros usuarios o entrar a canales. En los canales todo lo que se diga llegará a todos los usuarios conectados a ese canal.

Sockets

Un socket es un canal que se puede utilizar para comunicar dos aplicaciones. Un socket de internet es abierto por una aplicación determinada y se identifica por una dirección IP y por un puerto. La aplicación que se conecte a ese socket podrá dialogar con la aplicación que lo abrió.

En el mundo de Unix todo lo que es comunicación de procesos se maneja mediante una interfaz de archivos. Es decir, una vez pasado el proceso de apertura del canal de comunicación, lo que se obtiene es un descriptor de archivo donde se pueden realizar operaciones de lectura y escritura. Desde el lado del usuario se trata de manipular archivos, mientras que es el sistema operativo el que maneja los detalles de implementación.

Unix no utiliza la interfaz de archivos de ISO-C99 basada en `FILE *` sino una muy similar (y más sencilla) basada en descriptores de archivo (file descriptors, fd). Un descriptor de archivo es apenas un `int` que representa a un archivo abierto.

Para leer y escribir de ese archivo se pueden usar dos funciones equivalentes a `fread()` y `fwrite()` que son las funciones:

```
ssize_t read(int fd, void *buff, size_t nbytes);
ssize_t write(int fd, const void *buff, size_t nbytes);
```

ambas funciones leen del archivo `fd` (o escriben respectivamente) `nbytes` en (o desde) el puntero `buff`.

Al conectar un socket con un servidor lo que se obtiene es un `fd` que comunica con éste. Si se escribe en el `fd` eso será leído por el servidor, si el servidor se comunica con nosotros podremos leer del `fd` lo que nos envió.

Poll

Ya sabemos que la llamada a funciones de lectura de archivos son bloqueantes. Es decir, si nosotros llamamos, por ejemplo, a `getchar()` esa llamada va a quedarse congelada hasta que efectivamente ingrese algún carácter en `stdin`.

Si queremos tener una aplicación que hable con el usuario a través de `stdin` y que además hable con un servidor a través de otro archivo, para atender a los dos tenemos que poder saber de antemano si hay datos en los buffers de tal manera que cuando llamemos a una función de lectura estemos seguros de que esa llamada no bloqueará y nos devolverá el dato que ya está esperando.

Unix resuelve esto de manera sencilla con un mecanismo que se llama polling. Uno puede decirle al sistema operativo que está interesado en determinados descriptores de archivos y el sistema operativo puede quedarse escuchando en todos a la vez y avisarnos cuál de todos tiene un dato, de tal manera que cuando vayamos a leer podamos hacerlo de forma no bloqueante.

Protocolo de IRC

Como ya dijimos, el protocolo de IRC es un protocolo orientado a líneas. A su vez las líneas tienen diferentes campos... y ya conocemos el formato de estos campos: Es el formato dato en el EJ2. Los campos se separan por espacios, puede haber un : al comienzo. Si un campo empieza con : todo lo que sigue hasta el final de la línea es un único campo. Ya está.

En IRC cada usuario tiene un nick que es el que lo identifica. A su vez cada canal tiene un nombre, que es el que lo identifica. Los nombres de canal empiezan con numeral (#).

Cada comando que queramos mandarle al servidor será una secuencia de comando con sus respectivos parámetros, por ejemplo:

```
PRIVMSG pepito :Hola amigo, como estas?\n
```

Este comando le manda un mensaje personal al usuario `pepito`, el mensaje es `"Hola amigo, como estas?"`.

Todos los comandos que se le envían al servidor tienen una estructura similar a esa.

A su vez el servidor nos va a responder cosas (de forma asincrónica) que pueden ser o respuestas a nuestros comandos, para avisarnos del resultado de lo que pedimos, o puede mandarnos mensajes que no solicitamos... como por ejemplo, si pepito nos respondiera.

Los mensajes del servidor tienen una estructura similar pero son un poco más sucios, por ejemplo, si yo me llamara juancito y enviara el mensaje anterior el usuario pepito recibiría del servidor algo tipo:

```
:juancito!~juanx@143.232.32.1 PRIVMSG pepito :Hola amigo, como estas?\n
```

donde la estructura es `"<remitente> PRIVMSG <destinatario> <mensaje>"` la cadena `"juancito!~juanx@143.232.32.1"` codifica el nick `juancito` pero además el ID del usuario y la IP de mismo.

Si tanto juancito como pepito estuvieran en el canal `#algoritmos` y `pepito` mandara el mensaje:

```
PRIVMSG #algoritmos :Hola companeres!\n
```

a Juancito le llegaría un mensaje:

```
:pepito!~pepe@80.32.1.2 PRIVMSG #algoritmos :Hola companeres!\n
```

misma estructura que el anterior, pero el destinatario es el canal. Este mensaje le va a llegar a todos los miembros del canal con excepción de a pepito. A pepito no va a llegarle nada. Pobre pepito.

Los mensajes que se generan por interacciones de los usuarios tienen todos ese formato. Se van a generar menesajes no sólo cuando un usuario mande un mensaje sino también cuando un usuario entre a un canal en el que estamos, cuando se vaya de ese canal, cuando cierre la sesión, cuando se cambie de nick, etc. En todos los casos el servidor nos notificará solamente sobre usuarios que compartan canal con nosotros.

Los mensajes del servidor tienen un formato diferente, por ejemplo, si cuando juancito le escribió el mensaje a pepito, pepito no hubiera estado conectado el servidor le hubiera respondido a juancito algo tipo:

```
:ta130.com.ar 401 pepito juancito :No existe ese nick\n
```

la estructura no es fija, pero el aspecto suele ser `"<servidor> <código mensaje> <usuario> <información adicional...>"`. Los códigos de mensaje están tipificados (y son un montón), la información adicional muchas veces explica en texto un error, pero también consiste en información sobre ese comando puntual. Por ejemplo, si pidiera la información de todos los canales abiertos recibiría un listado de códigos 322 donde la información sería el nombre de los canales.

Como el servidor puede hablarnos incluso cuando todavía no terminamos de conectarnos no necesariamente el tercer parámetro sea nuestro nick.

El único (*creo*) comando del servidor que no tiene esa estructura es el comando `PING`, el servidor cada un determinado tiempo nos va a mandar un ping, que nosotros tenemos que responder con un pong, si no respondemos va a asumir que no estamos ahí y nos va a desconectar. El ping del servidor tiene esta estructura:

```
PING :abcdefghi\n
```

donde el segundo campo es un código *secreto* que nosotros tenemos que utilizar en la respuesta:

```
PONG :abcdefghi\n
```

si el código no coincide el servidor va a echarnos.

Bienvenida

En todo protocolo existe un *apretón de manos* (handshake) que establece la conexión inicial. Como ya dijimos, todos los usuarios de IRC tienen un nick, en ese handshake le vamos a decir al servidor quiénes somos, y el servidor nos aceptará (o nos echará si, por ejemplo, ya hubiera otro usuario con ese nick).

Al establecer la conexión el usuario tiene que enviar estos dos comandos:

```
USER usuario host server :Nombre Real\n
NICK usuario\n
```

(¿Se acuerdan del juanx que aparecía en `juancito!~juanx`?, ese usuario se toma del comando USER, no necesariamente es el NICK que se proporcione después.).

Si todo está bien el servidor tiene que responder con:

```
:servidor 001 usuario :Bienvenido al IRC usuario\n
```

Recién cuando llega el mensaje `001` es que estamos conectados, no antes.

Comandos de IRC

He aquí una lista de comandos de IRC:

PRIVMSG destinatario mensaje:

Manda el mensaje al destinatario (usuario o canal).

NICK nick:

Se cambia el nick a nick.

JOIN canal:

Entra al canal.

PART canal:

Sale del canal.

NAMES canal:

Pide la lista de los usuarios en un canal.

TOPIC canal:

Pide el tópico (temática) del canal.

TOPIC canal topico:

Establece el tópico (temática) del canal.

LIST:

Pide la lista de todos los canales del servidor.

USERS:

Pide la lista de todos los usuarios del servidor.

QUIT:

Sale del servidor.

PING palabrasecreta:

Pinguea al servidor.

PONG palabrasecreta:

Contesta un ping del servidor.

A su vez estos son los comandos que el servidor puede enviarnos a nosotros:

remitente PRIVMSG destinatario mensaje:

Recibimos un mensaje del remitente, el destinatario somos nosotros o un canal común.

remitente NICK nick:

El usuario remitente se cambió el nick a nick. (A partir de ahora los mensajes nos llegarán con el nuevo nick como remitente.)

remitente JOIN canal:

El remitente entró al canal.

remitente PART canal:

El remitente salió del canal.

remitente QUIT canal motivo:

El remitente se desconectó.

remitente TOPIC canal topico:

El remitente le cambió el tópico (temática) al canal.

PING palabrasecreta:

Nos está pidiendo que respondamos.

PONG palabrasecreta:

Está respondiendo a un PING nuestro.

Por fuera de eso, el servidor nos puede enviar los mensajes que ya dijimos donde tienen el formato ya presentado servidor-código-nick-mensaje con la salvedad de que el código es o un número o la palabra `"NOTICE"` si el servidor nos está avisando de cosas no tabuladas.

Trabajo

El objetivo de este trabajo es implementar un cliente de IRC muy sencillo que se conecte a un servidor y permita chatear.

El comportamiento del cliente debería ser así:

La aplicación se va a iniciar con la dirección del servidor y el nick a usar.

El cliente debe conectarse al servidor.

Una vez que esté conectado al servidor y todo el tiempo que dure la aplicación el cliente tiene que estar leyendo la entrada del usuario y además debe estar mostrando los mensajes que vienen del servidor.

La entrada del usuario se lee por `stdin` en la terminal ya implementada en el EJ3. Los mensajes del servidor "interrumpen" al usuario.

Se asume que el usuario domina los comandos de IRC. Si la entrada del usuario empezara con `'/'` se asume que está ingresando un comando al servidor. Lo que el usuario haya escrito debe ser enviado al servidor, con dos modificaciones: Primero hay que omitir la barra, después, si hubiera más de 2 comandos se debe insertar un : después del segundo. Por ejemplo, si el usuario ingresa:

```
/topic #algoritmos Este cuatri la rompemos
```

el cliente debe enviarle al servidor `"topic #algoritmos :Este cuatri la rompemos"`.

El prompt empezará estando vacío. Si el usuario hace join a un canal el prompt será el nombre del canal, si el usuario le manda un mensaje privado a alguien (usuario o canal) el prompt se convertirá en ese destinatario.

Si el usuario no empieza su entrada con la `'/'` entonces se asumirá que lo que está escribiendo se lo está mandando al destinatario de su prompt. O sea, una secuencia como:

```
/join #algoritmos
Hola a todos
/privmsg juancito Excepto a vos
No, mentira ;)
```

Debería traducirse en:

```
join #algoritmos
privmsg #algoritmos :Hola a todos
privmsg juancito :Excepto a vos
privmsg juancito :No, mentira ;)
```

(Dicho sea de paso, en todos los clientes de IRC el comando para mandar mensajes es `"/msg"` y no `"/privmsg"`, como algo optativo se puede hacer esa traducción. Se agradece.)

Identificamos varios tipos de mensajes del servidor: Mensajes personales, mensajes que vienen de un canal, mensajes que me avisan de que alguien hizo algo, mensajes que vienen directo del servidor; y además están los mensajes que nosotros estamos mandando. Colorear de forma que puedan distinguirse unos de otros. Particularmente los mensajes formatearlos para que sea claro quién los envió y si son mensajes personales o de canal.

Por último el cliente tiene que ser capaz de responder al PING de forma automática sin interacción del usuario.

¿Tengo que aprender sockets, poll y compañía?

No, amigo, por supuesto que no.

Te proveemos un TDA `comunicador_t` que se encarga de conectarse y que te avisa si hay datos en `stdin` o en el descriptor de archivos del servidor.

Sólo tenés que usar el TDA.

El pseudocódigo de uso del TDA es:

```
c = cominicador_crear();

if(! comunicador_conectar(c, servidor, puerto)) {
    error = comunicador_get_error(c);
}

int fd;
while((fd = comunicador_esperar_datos(c)) != -1)) {
    if(fd == 0)
        // Sé que hay datos en stdin
    else
        // Sé que hay datos en fd para leer con read()
}

comunicador_destruir(c);
```

Dado que el servidor se comunica de a líneas, es válido asumir que si hay al menos un dato en el fd se pueden leer bytes de a uno por vez hasta eventualmente leer un `'\n'`. Reimplementar la función de lectura de línea del EJ2 para leer con read en vez de getchar. Siendo que el canal de comunicación es binario, si se lee un `'\r'` descartar el dato.

En la implementación de la cátedra se observó que a veces `comunicador_esperar_datos()` bloquea la evacuación del buffer de `stdout` y genera un delay entre que se imprime algo y se muestra. Se puede agregar una llamada a `fflush(stdout);` después de imprimir la terminal para garantizar que los datos del buffer se imprimieron realmente.

¿Cómo pruebo al cliente?

Te proveemos un servidor ya compilado que muestra por `stdout` todo lo que está pasando. El servidor se invoca como:

```
$ ./server <puerto>
```

donde el puerto por default es 6667, pero tal vez necesites cambiarlo si quedó ocupado porque algo te crasheó antes.

El servidor que te damos es bastante benevolente, tiene mucha más paciencia que un servidor de IRC real, y es bastante explícito en contarte qué está pasando.

Del servidor se sale con control + C (no, no lo estás matando, sabe cómo salir de forma limpia, y si no me creés correle Valgrind ;)).

Aplicación

Tenés que implementar una aplicación que se ejecute como:

```
$ ./cliente servidor puerto nick
```

y que permita conectarse al servidor y chatear.

Se provee el siguiente paquete con la implementación del comunicador y con el servidor: archivos_20241_ej4.tar.gz.

Nota

Para este trabajo no es necesario modularizar el problema, pero se recomienda hacerlo para practicar modularización y Makefile.

En el caso de modularizar en archivos entregar tanto los archivos como el Makefile que compila el proyecto.

Entrega

Deberá entregarse el código fuente del programa desarrollado o los fuentes y el Makefile en caso de haber modularizado.

El programa debe:

1. Compilar correctamente con los flags:

   ```
   -Wall -Werror -std=c99 -pedantic
   ```

2. permitir interactuar con el servidor provisto con los comandos que se describieron en este enunciado.

La entrega se realiza a través del sistema de entregas.

El ejercicio es de entrega individual.