Instituto Tecnológico De Costa Rica

Sede Central

IC4700

Lenguajes de Programación, Grupo 2

Tarea Programada 1

Estudiantes:

Fabricio Ceciliano Navarro	2017111236
Carlos Esquivel Morales	2017146886
Diego Murillo Hernández	2017111111

Profesor: Andrei Fuentes Leiva

Viernes 22 de Marzo

Semestre I, 2019

Introducción

A lo largo de las lecciones del curso de Lenguajes de Programación se han visto varias características del lenguaje C. Entre ellas se incluyen: sintaxis, tipos de variables, funciones auxiliares, compilación, punteros, arreglo, *structs*, etc. Todas estas funcionalidades que presta el lenguaje mencionado facilitan la codificación de diferentes programas para diversas finalidades: manejo de memoria, manejo de caracteres, etc.

Sin embargo, el lenguaje C posee muchísimas características más, como lo son el uso de hilos (*threads*) y *sockets*. Un *socket* se refiere a un tipo de conexión abstracta por la cual dos programas (servidor y cliente) pueden intercambiar flujos de datos. En el caso de Java es posible enviar objetos, pero para el presente proyecto programado, y dado que se utilizará C, únicamente interesa el proceso de envío de texto o mensajes.

La finalidad del presente proyecto consiste en realizar un servicio de mensajería o un tipo de *Messenger* en el cual sea posible enviar y recibir mensajes entre diferentes usuarios, los cuales serán enviados a un ente principal (servidor) y que éste se encargue de distribuirlos hacia los destinatarios deseados.

Por otra parte, se hará uso de la herramienta en línea *GitHub*, la cual facilita la robustez y orden de código pues, al ser en línea, permite tener un mejor control de las diferentes versiones del proyecto, así como facilitar el acceso a cambios realizados a este. De igual forma, permite que todos aquellos usuarios que tengan acceso al repositorio puedan modificar y actualizar el código que ahí se encuentre, evitando así mal entendidos entre los cambios que cada programador realice.

Presentación y análisis del problema

¿Qué es lo que hay que resolver?

Para completar la tarea encomendada satisfactoriamente, es necesaria la aplicación de los conocimientos aprendidos en clase y, como se mencionó anteriormente, crear un programa de mensajería que cumpla con los siguientes requisitos:

a) Debe haber un servidor central en el cual se almacenen los datos de cada usuario creado. Además, éste se encargará de la transmisión de los mensajes de un usuario a otro; es decir, el mensaje de un usuario A tendrá que ser transmitido al servidor para poder ser enviado a un usuario B.

b) Debe existir un programa cliente que se conecte con el servidor central y cree un usuario. Este nuevo usuario deberá ser capaz de enviar mensajes a otros usuarios conectados al servidor. Como se mencionó anteriormente, el mensaje debe pasar por el servidor para ser redirigido al destinatario.

c) Asimismo, el destinatario debe ser capaz de visualizar el mensaje que le dirijan otros usuarios. Para esto, el cliente tendrá que introducir el usuario que recibirá el mensaje antes de redactar el cuerpo del mensaje.

Para cumplir con estos objetivos, es necesaria la investigación de distintas características que son de importancia crítica para el funcionamiento del programa: la ejecución de procesos concurrentes y el intercambio de datos mediante una conexión entre *sockets*.

- En cuanto a la ejecución de procesos concurrentes se debe utilizar la función nativa *fork*. Dicha función permite la creación de un proceso hijo de un hilo de ejecución principal, en el cual se ejecutará la instrucción inmediatamente siguiente de la llamada de *fork*. El proceso hijo utilizará los mismos recursos (registros de CPU, archivos abiertos…) que puede utilizar el proceso padre. La función *fork* provee un valor de retorno que indica la creación satisfactoria del proceso hijo.
    - El uso de esta función se debe a la cualidad del programa de poder enviar y recibir mensajes al mismo tiempo, por lo que es necesaria una bifurcación del hilo principal en sub procesos capaces de hacer dichas funciones al mismo tiempo.
- Como se mencionó anteriormente, una conexión de *socket* es aquella por la cual dos programas (nodos) pueden intercambiar flujos de datos. Para conectar los nodos es necesario conocer la dirección IP de la red y el puerto a utilizar. Cada proceso (servidor o cliente) inicializa su propio *socket* para enviar y recibir información. Para poder conectarse, ambos sockets deben encontrarse en el mismo dominio y ser del mismo tipo (de flujo o de datagrama; cada tipo utiliza un protocolo de comunicación: TCP y UDP, respectivamente).
    - Los *sockets* son cruciales para poder efectuar la conexión y la transmisión de datos de un cliente al servidor central.

Además de esto, en el caso de la escritura y visualización de mensajes y/o información en general es necesaria alguna clase de interfaz gráfica para desplegar dichos datos, ya sea mediante el uso de una librería gráfica o bien utilizando la consola nativa del sistema operativo. Puesto de forma simple, debe haber una pantalla en la cual puedan tanto leerse mensajes como escribirlos.

Una vez desarrollados ambos programas (servidor y cliente), es necesaria la creación de un MAKEFILE que contenga los pasos a seguir para facilitar la compilación del código fuente y la creación de cada programa ejecutable según corresponda.

Finalmente, se requiere el uso de un repositorio de control de versiones (en este caso, *GitHub*) en el que debe encontrarse la aplicación, así como la actividad del grupo para poder llegar a la solución final.

¿Cómo se va a resolver el problema?

Para empezar, y como se mencionó en la sección anterior, los sockets utilizan un protocolo de comunicación dependiendo del tipo que sean: Protocolo de Control de Transmisión o Protocolo de Datagramas de UNIX; TCP y UDP por sus siglas en inglés, respectivamente. Después de una ardua investigación y un proceso de prueba y error, se decidió por usar el protocolo UDP (basándose en un código existente). A diferencia del protocolo TCP, el protocolo UDP omite un proceso de aceptación por parte del servidor; solamente debe enviarse la información por el *socket* a una dirección específica. Este protocolo garantiza que los datos recibidos sean correctos pero es incapaz de leer información parcial; es decir, el mensaje DEBE arribar completo a su destino. Debido a esto, se dice que no es confiable pero que está orientado al envío de mensajes, haciéndolo apto para esta tarea.

En cuanto a la interfaz del programa, se escogió el uso de la terminal nativa del sistema operativo por motivos de simplicidad y facilidad en el desarrollo de la aplicación. Adicionalmente, los mensajes impresos en consola poseerán un formato y un color específico para hacer la aplicación más amigable con el usuario.

Al iniciar tanto el programa cliente como el servidor, se abrirá el archivo "*portNumber.ini*" en el cual se encuentra la dirección IP, el puerto al que se conectará el servidor y el puerto al que se conectará el cliente. Nótese que una vez utilizado un puerto de cliente solo un usuario puede utilizarlo; para conectar más usuarios es necesario modificar el espacio *portNumberClient* en el archivo. Además de esto, es necesario que se ejecute el servidor antes que cualquier cliente; de lo contrario, los clientes no podrán iniciar el programa, ni mucho menos enviar mensajes.

Una vez configurado el servidor, este abrirá un socket y entrará en un ciclo en el cual permanecerá escuchando cuando un nuevo usuario se conecte. Cuando esto ocurra, se efectuarán las verificaciones necesarias, que se describirán a continuación, y ejecutará una instrucción ***fork***, la cual creará un hilo concurrente en el cual se procesarán los mensajes subsecuentes del cliente correspondiente. Esto ocurrirá con cada nuevo cliente que se conecte; es decir, cada cliente contará con su propio hilo.

Al iniciar el programa cliente (con el programa servidor ejecutándose), el usuario tiene la opción de ingresar un nombre de cuenta que lo identificará en el sistema. Si no lo ingresa, el servidor creará un usuario automático para él (el servidor contará con una lista de nombres posibles, junto con una combinación numérica). En cualquier caso, se verificará que el *username* sea único en todo el sistema; si no lo es, no se le dará acceso al cliente. Al registrarse, se guardará el *username* y se vinculará con un color aleatorio en la lista de usuarios conectados. Mediante un *struct* denominado ***user***, se almacena la dirección, el nombre y el color de un usuario dentro del servidor.

- En cuanto al color aleatorio del usuario: debido a la corta gama de colores básicos de una terminal nativa, está la posibilidad de repetir colores en varios usuarios.

Una vez registrado el usuario y el proceso *fork* se haya ejecutado satisfactoriamente en el servidor, el cliente tendrá la capacidad de enviar mensajes al servidor mediante su socket. El cliente tiene la posibilidad de enviar dos tipos de mensajes:

- Mensajes a un destinatario: O en otras palabras, un mensaje normal. Para ello, el cliente debe especificar el *username* del destinatario, seguido de un ":" y luego el cuerpo del mensaje (*[destinatario]:[mensaje]*)
- Comandos: Sirven para hacer una solicitud al servidor. Se ejecutan al escribir "*comm*" seguido de una letra (*comm [simbolo]*). Existen dos comandos:
    - 'p': Imprime una lista de los usuarios actualmente en línea, tanto en el servidor como en el cliente que lo solicita.
    - 'e': Señaliza la salida del cliente y lo desconecta del servidor, a su vez cerrando la aplicación cliente.

Si el mensaje posee un formato erróneo o inexistente o se trata de enviar un mensaje a un usuario que no está registrado, el servidor le hará saber al cliente mediante un mensaje en la consola.

Al mismo tiempo que se pueden enviar mensajes, el cliente también es capaz de escuchar los mensajes que llegan a él gracias a una función *fork* propia, de la misma forma que el servidor. Al recibir mensajes de otro usuario, se imprimirá el *username* con el color respectivo (esto para identificar más fácilmente al remitente) seguido del cuerpo del mensaje en la terminal.

Finalmente, para ejecutar los programas basta con seguir las instrucciones encontradas en el MAKEFILE que viene con el código fuente de la aplicación.

Conclusiones

- El lenguaje C cuenta con una función *fork* que resulta útil a la hora de ejecutar procesos simultáneos. Sin embargo, si se desean compartir datos entre cada proceso concurrente, es más recomendable (y más común) el uso de un *thread* normal.

- A pesar de que el lenguaje C no posee las cualidades de un lenguaje orientado a objetos y carece de la capacidad de definir clases en sí, el uso de *structs* permitió cumplir con esta necesidad de manera parcial pero satisfactoriamente.
- Es posible que los colores impresos en pantalla varíen a lo largo de diferentes programas de consola en distintos sistemas operativos. Por ejemplo, la consola nativa del sistema operativo Deepin (Linux, basado en Debian) imprimirá colores distintos a la terminal de Ubuntu.
- En cuanto al manejo de archivos, se puede concluir que para guardar datos importantes como la IP y el port number, es una buena práctica utilizar archivos, en este caso un archivo de configuración, pues esto aumenta la robustez del código al no tener que recompilarlo cuando se quieran hacer cambios como es el caso del cambio del puerto del cliente.
- Por otra parte, el hecho de trabajar con punteros y arreglos en C implica tener un conocimiento un poco avanzado de cómo funciona la memoria de los programas. Esto porque aveces mostraba un error de tipo segmentation fault o que una funcion devolvía una variable local entonces mostraba error y había que enviarle esa variable como parámetro.
- Finalmente, se puede concluir que el uso del fork hace que la implementación de este programa en particular se vuelva más compleja, porque con threads no se hubieran tenido que preocupar por usar memoria compartida con *mmap*, y se hubiera podido usar TCP de manera más sencilla.

Referencias Bibliográficas

Creative Commons (2007). *Programación de Sockets en C de Unix/Linux*. Recuperado de: http://www.chuidiang.org/clinux/sockets/sockets_simp.php

Ingalls, R. (s.f.). Sockets Tutorial. Recuperado de: http://www.cs.rpi.edu/~moorthy/Courses/os98/Pgms/socket.html

Nitsch, S. (2004). *Sockets Tutorial*. Recuperado de: http://www.linuxhowtos.org/C_C++/socket.htm

Shene, C.-K. (s.f.). *The fork() system call*. Recuperado de: http://www.csl.mtu.edu/cs4411.ck/www/NOTES/process/fork/create.html