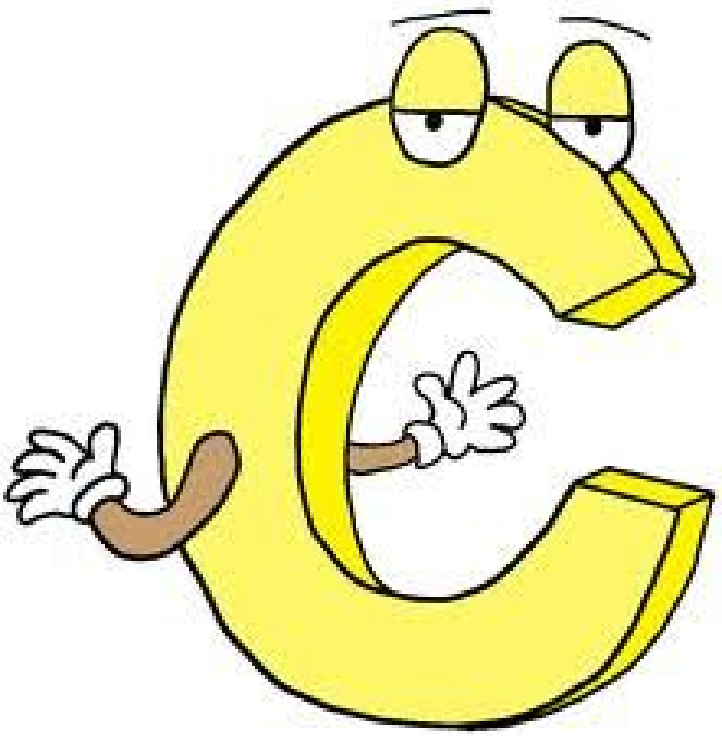

Trabajo Práctico 0

Sistemas Operativos

UTN-FRBA

Versión 1.0

¿De qué se trata?

El TP0 es una práctica inicial para empezar a familiarizarse con algunas de las herramientas necesarias para el trabajo práctico cuatrimestral como C, sockets y serialización. Es un ejercicio corto y sirve como base para empezar el TP luego.

Requisitos

Antes de empezar es necesario:

- Descargar una de las VMs https://www.utnso.com/recursos/maquinas-virtuales/
- Conocimientos mínimos de C
- Conocimientos básicos de sockets: Leer y tener a mano http://beej.us/guide/bgnet/
- Tener instaladas la biblioteca de TADs de uso común (“las commons”, para los amigos) https://github.com/sisoputnfrba/so-commons-library

Desarrollo

El objetivo del ejercicio es crear un programa simple en C que utilice sockets y serialización para conectarse a un servidor remoto e intercambiar algunos mensajes.
Luego de cada mensaje que el cliente envíe, el servidor validará que sea correcto y permitirá continuar. En caso que se envíe un mensaje incorrecto, cerrará la conexión. Al finalizar la cadena de mensajes, el servidor responderá con un mensaje de éxito.

Inicio

Para empezar con el ejemplo es necesario bajase el esqueleto del TP del repositorio https://github.com/sisoputnfrba/tp0 mediante **`git clone`**. Allí vamos a tener los siguientes archivos:

- El archivo fuente **tp0.c**, sobre el que vamos a trabajar.
- El archivo de encabezado **tp0.h**, que tiene varias definiciones que vamos a usar.
- El archivo **makefile**, que nos va a permitir compilar el tp0 mediante el comando **`make`**. Esto va a ser especialmente útil para instancias finales del TP general.

Abramos el archivo fuente y empecemos...

Creando la configuración Inicial

Durante todo el tp iremos logueando en un archivo de texto las diferentes acciones que el programa vaya realizando, tanto las correctas, como los errores. Para ello utilizaremos las funciones de logging que proveen las commons. Las funciones **`configure_logger`** y **`exit_gracefully`** están pensadas para crear nuestro logger y cerrarlo cuando terminemos de usarlo. Para el caso del la primera función (1.) si revisamos el header de logs de las

commons, habla de la función **`log_create()`**, que nos devuelve un logger listo para usar. Usaremos esa función para configurarlo para que:

1. Loguee en el archivo tp0.log
2. Muestre los logs por pantalla y no solo los escriba en el archivo.
3. Muestre solo los logs del nivel info para arriba.

Finalmente, en **`exit_gracefully()`** debemos aprovechar la función contraparte de **`log_create()`** para eliminar el logger y liberar los recursos. Además, dado su comportamiento, esta función deberá terminar la ejecución del programa.

El Protocolo a usar

`connect_to_server(2)`

Para iniciar la comunicación el cliente debe crear un socket TCP/IP (2.) de la conexión mediante su función homónima con los siguiente datos:

- Host: ***tp0.utnso.com***
- Puerto: ***8080***

Luego, tenemos que usar ese socket para conectarnos (3.) al servidor mediante la función **`connect()`**[1]. Para chequear que se efectuó correctamente (3.1.), vamos a tener que revisar el valor de retorno de las funciones y si es necesario, salir del programa con la función que hablábamos antes, **`exit_gracefully()`**.

`wait_hello(1)`

Al conectarse deberá recibir un mensaje de 16 bytes e imprimirlo por pantalla (5).

`send_hello(2)`

Después, nos toca mandar a nosotros un mensaje. El mensaje que vamos a mandar va a ser la información del alumno almacenado en una estructura con la siguiente forma:

```
struct typedef {
  int id_mensaje,
  int legajo,
  char nombre[40],
  char apellido [40]
} __attribute__((packed)) Alumno;
```

Más adelante, vamos a explicar como conseguir el resto de la información, pero por ahora para no meter ruido, sigamos con el intercambio de mensajes. Mientras tanto, vamos a necesitar completar el campo *id_mensaje*, que para el protocolo, debe ser el valor 99 (11.).

[1] http://beej.us/guide/bgnet/html/single/bgnet.html#connect

Si te estás preguntando porqué funciona el send y no es necesario *"serializar"* nada de la estructura, es porque usa el atributo **"__attribute__((packed))"**. Esto solamente funciona siempre y cuando la estructura no tenga punteros y los datos se envíen a procesadores con arquitecturas compatibles. En caso que alguna de las condiciones no se cumplan, es necesario crear un buffer y serializar los datos antes de poder enviarlos.

Finalmente, nos toca enviar la estructura con **send()**. Tengan en cuenta que recibe un puntero a los datos que va a mandar, así que la variable alumno hay que pasarla con el operador de dirección **&**.

wait_content(1)

A continuación deberá esperar la respuesta del servidor. La misma va a tener un tamaño variable, por lo que nuestro protocolo va a definir el mensaje de la siguiente forma:

ID	**Tamaño de datos**	**Datos**
4 Bytes	4 Bytes	n Bytes

- El identificador será el entero *18*
- El campo tamaño de datos indicará la cantidad de bytes necesarios para almacenar el campo datos.
- El campo datos será el contenido de un archivo que deberá almacenar localmente en el disco.
- El tamaño mínimo serán 60 bytes y no está definido un tamaño máximo.

Primero tendremos que recibir la primera parte del mensaje, compuesta por el ID y el tamaño de los datos (13.) y después, con el tamaño, nos toca recibir el resto del mensaje (14.).

Nota: es importante tener en cuenta que el servidor enviará un string como dato pero no incluirá '\0' al final. El proceso deberá agregarlo para que el MD5 coincida.

Calcular el MD5

send_md5(1)

Una vez recibido el mensaje vamos a calcular el MD5 y enviarlo al servidor, con el objetivo que valide que hemos recibido bien el mismo. Para calcular el MD5 vamos a utilizar la biblioteca openssl (16.) para luego enviarlo (18.).

Para mandar el MD5, vamos a usar un mensaje de tamaño variable (17.) definido en nuestro protocolo (aunque podríamos usar un mensaje de tamaño fijo). El formato es similar al que recibimos anteriormente:

ID	**Tamaño de datos**	**Datos**

4 Bytes	4 Bytes	n Bytes

- El identificador, para mandar info variable, ser el entero *33*.
- El campo tamaño será el tamaño de los datos a enviar
- El campo datos contendrá el resultado efectuar el MD5 sobre la cadena recibida.

`wait_confirmation(1)`

Para finalizar recibirá un último mensaje (19.) de 4 bytes indicando si el proceso fue satisfactorio (valor: 1) o si hubo algún inconveniente (valor: 0).

Obtener los datos del alumno

Volvamos a la función **`read_hello`**, en la que vamos a obtener los datos del alumno. ¿Cómo vamos a obtener esa data? Vamos a ingresarla por consola.

Para ello, primero tenemos que definir la estructura que mencionamos antes (6.) dejando los campos de nombre y apellido como cadenas vacías, para asegurarnos que todo el array de chars que define la estructura para ambos campos tengan un estado definido (y no "basura").

Existen varias formas de leer de la consola, pero por comodidad, vamos a usar la biblioteca **`readline`** (7.). De ahí vamos a ir pidiendo por consola que ingresen el legajo (8.), el nombre (9.) y, finalmente, el apellido (9.1.).

Para los últimos 2 datos, vamos a necesitar hacer uso de la función **`memcpy()`**[2] para copiar los valores de la porción de memoria devuelta por **`readline()`**, a nuestra estructura alumno.

Notas Finales

La biblioteca **readline** nos ahorra un montón de trabajo, porque nos reserva la memoria justa para leer de consola y lee hasta el fin de línea, todo de forma automágica mientras que invoquemos su función.
Pero todo lo mágico tiene algún pequeño inconveniente, *no todo es gratis*. Si usamos readline en nuestro código, si se hace un análisis de la memoria usada por el programa mediante valgrind, no va a reportar errores, peeero va a informar que algunos bytes todavía son "still reachable".
¿Qué significa still reachable? Significa que es memoria a la cual podríamos haber liberado justito antes de terminar nuestro programa, *pero no lo hicimos*. En este caso, no fuimos nosotros, sino que fue **readline**. Si revisan el siguiente foro (en ingles), van a ver como van deduciendo por donde viene el error, hasta llegar a que **es un bug de readline** (algo que incluso, tienen reportado).

[2] http://man7.org/linux/man-pages/man3/memcpy.3.html#DESCRIPTION

Por lo tanto, *no se preocupen* si al evaluar sus programas con valgrind (con los flags para ver los “still reachable”, que son “--leak-check=full --show-reachable=yes”), encuentran que valgrind les comenta que readline no libera toda la memoria, ya que, a fines del trabajo práctico, **no nos importa.**