Sistemas Operativos

Segundo cuatrimestre 2016

Trabajo Práctico Nro. 1: Inter Process Communication

Integrantes:
Barruffaldi, Carla 55421
Bianchi, Luciano 56398
Cerdá, Tomás 56281

INTRODUCCIÓN

Para este trabajo práctico se eligió recrear un entorno similar al de la red social 'Twitter' haciendo uso de IPC[1] para controlar el flujo de datos entre un servidor y varios clientes concurrentes, así como entre el servidor y la base de datos.

El programa se basa en que cada cliente puede ingresar con un nombre de usuario a la 'red social' para realizar distintas acciones.
Puede escribir un mensaje y compartirlo con otros usuarios, además puede ver 'tweets' o mensajes que compartieron otros usuarios.

Cada tweet tiene un número identificador (id) para realizar acciones sobre él. Por ejemplo las acciones 'like', 'delete', 'show', agregan un 'like' al tweet, lo eliminan, o lo muestran en pantalla respectivamente.
Para ver los mensajes nuevos en pantalla se usa el comando 'refresh'.
Para más información sobre los comandos disponibles puede usarse el comando 'help'.

CAPA IPC

La comunicación entre cliente y servidor fue realizada con dos implementaciones: Named Pipes(FIFO) y Sockets. Se puede elegir cual usar en el momento de compilación y linkedición (Para ayuda con la compilación y ejecución ver archivo README). Estas implementaciones fueron abstraídas creando una capa de comunicación única, cuya interfaz está dada por IPC.h. Allí se puede encontrar información sobre cada una de sus funciones.

El peer que desea leer puede aceptar una conexión a través de la función accept_peer(). A su vez, el peer que desea establecer una conexión lo

[1] Inter Process Communication

hace a través de la función connect_peer(). Se optó por este diseño pues es flexible y es claro para el peer que desea leer, el momento en el cual se acepta una conexión. Esto provee gran facilidad si se desean realizar ciertas acciones en el momento que se acepta una conexión, como por ejemplo abrir un nuevo thread que se encargue de dicha conexión.

Una vez establecidas las conexiones, el peer que escribe simplemente crea un request una única vez, settea su contenido cuantas veces quiera y lo envía mediante send_request(), función que a su vez devuelve la respuesta. El otro peer espera y recibe un request mediante read_request() y lo responde mediante send_response(). Estas tres funciones son bloqueantes.

Por último se proveen destructores para cada ADT que haya sido creado a través de un constructor. Por lo tanto, los requests **respondidos** y las respuestas **leídas** son destruidas por la API a partir del último momento en el que ya no son necesitadas. Esto sería a través de las funciones send_response() (destruye el request) y get_response_msg().

Los bytes enviados en los requests y los leídos en los response tienen un tamaño fijo de BUFSIZE. A contraposición de usar tamaños variables, esto nos permite enviar estructuras fácil y claramente. Más importante aún, si el contenido de las estructuras cambian, el código de la capa de IPC que escribe y lee no necesita cambiar, pues se envía la estructura entera (siempre y cuando entre en BUFSIZE). En cambio, con tamaños variables se hubiese tenido que enviar cada campo de la estructura precedido por la cantidad de bytes a leer, lo cual hubiese terminado en un código menos claro y adaptable, sin mencionar la necesidad de hacer múltiples mallocs y frees.

CAPAS MARSHALLING/UNMARSHALLING

Dado que la capa de comunicación solo trabaja con bytes, fue necesario tener otra capa que se encargue de ‘traducir’ esos bytes para cada aplicación. Esta es la capa de marshalling/unmarshalling. El cliente, el servidor y la base de datos tienen distintas implementaciones según las acciones que realizan.
Para todos los casos, esta capa es además la encargada de establecer las conexiones necesarias para la comunicación. Para manejar estas conexiones, también proveen una abstracción por encima de la interfaz de IPC de la capa de comunicación denominada session.

La aplicación del cliente envía un comando junto con la información necesaria para su ejecución, luego la capa de marshalling del cliente genera una cadena de caracteres con un opcode correspondiente al número

del comando y los datos requeridos y la envía al servidor mediante la capa de comunicación. La capa de marshalling del servidor recibe esta cadena y la interpreta dando la información necesaria a la aplicación del servidor para que se ejecute lo pedido.

Luego, la aplicación del servidor envía a través de su otra capa de marshalling, 'db_marshalling', la información que necesita obtener de la base de datos y la obtiene por allí también.

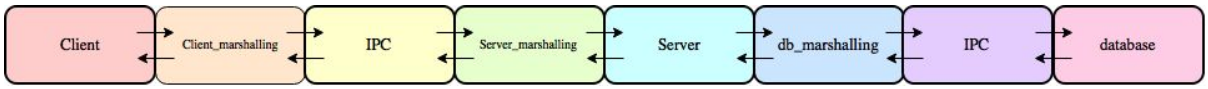

Fig.1 Flujo de datos

LOGGING

Al inicializar el servidor, se ejecuta un daemon logging independiente con el cual se comunicará. Este log fue implementado mediante la interfaz POSIX de Message Queues y tiene tres niveles: info, warning y error. Siendo 'info' mensajes de prioridad baja pues solo informan un acontecimiento sucedido, 'warning' mensajes en los cuales se genera un error pero aun así la aplicación sigue funcionando correctamente (por ejemplo error al enviar un tweet) y 'error' para casos críticos en los que se debe cerrar la aplicación. Estos últimos tienen la prioridad más alta.

APLICACIÓN SERVIDOR Y THREADS

Para que el servidor pueda manejar múltiples clientes concurrentemente, se decidió crear un thread del lado del servidor para atender a cada uno de ellos. El proceso principal del servidor espera nuevas conexiones, y cada vez que acepta una nueva, crea un thread para atenderla. A su vez, este nuevo thread tiene una conexión a la base de datos, que funciona de la misma manera, esperando nuevas conexiones de servidores, y creando threads para atenderlos.

Se prefirió el uso de threads sobre forks debido a que al compartir el heap resultó práctico enviar punteros, como el ADT de session devuelto por la capa de marshalling del servidor al recibir una nueva conexión. Además, ya que se espera la conexión de muchos clientes, forkear hubiese tomado mucho lugar en la tabla de procesos.

No se utiliza un mutex para sincronizar los threads del servidor pues la base de datos asegura sincronización a través de un mutex propio como se explica más abajo. Por lo tanto cada query a la base de datos podría

considerarse una acción atómica desde un punto de vista semántico, y como cada acción del servidor se resuelve con un única query no es necesario un mutex del lado del servidor.

Para almacenar la información de los tweets se implementó con la librería SQLite una base de datos que se comunica con el servidor. Para abstraer del servidor la codificación de operaciones (queries), decodificación de resultados y conexiones hacia la base de datos, se optó por implementar una capa que se denominó 'db_marshalling', análoga a las capas de marshalling del servidor y cliente.

BASE DE DATOS

La base de datos funciona como una aplicación independiente, y lee las solicitudes que recibe de los servidores de manera concurrente. Para evitar problemas con dicha concurrencia, se usó un mutex de la librería pthread.h, para guardar que no se pueda realizar más de una query en la base de datos al mismo tiempo.

Se decidió aprovechar la misma interfaz de IPC usada en la comunicación cliente-servidor, para conectar al servidor con la base de datos. Como la BD es un proceso aparte, era una solución lógica usar el sistema de IPC que ya se había implementado, reusando el código y aprovechando la generalidad de la abstracción diseñada.

Shared Memory, la otra opción propuesta en la consigna, podría haber sido una solución razonable si la base de datos hubiese estado en el mismo proceso que el servidor. En ese caso, todos los threads del servidor hubiesen usado mutexes para coordinar sus accesos a ella y realizar las queries concurrentemente. Sin embargo, al tener la BD como un componente separado del servidor, SM hubiese tenido que ser usado como una especie de método de comunicación, lo cual hubiese sido sumamente inconveniente e ineficiente.

Cabe destacar que solo se usa la interfaz de IPC diseñada (independiente de la implementación), por lo que el servidor y la BD se pueden comunicar por cualquiera de los métodos elegidos en la compilación, sean fifos o sockets (usarán la misma implementación que cliente-servidor).

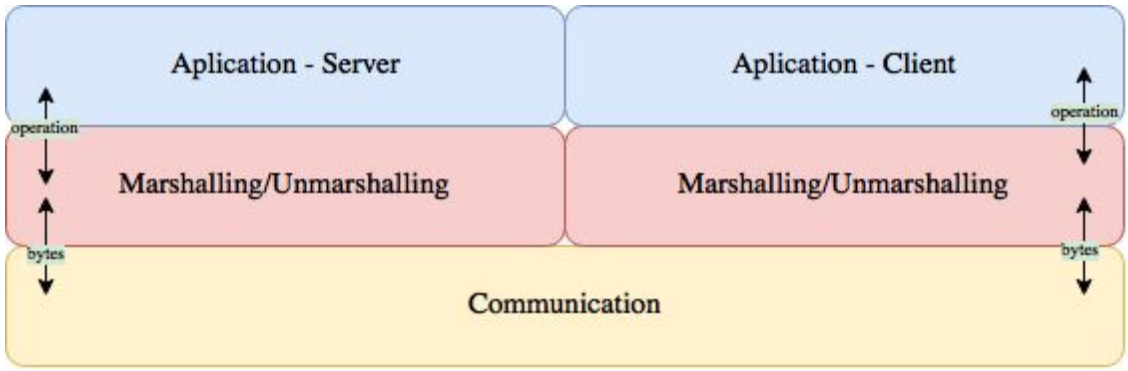

Fig.2 Capas

OBSERVACIONES

Debido a que la IPC puede enviar una cantidad BUFSIZE máxima de bytes, para la operación de refresh se envían varios requests pidiendo tweets a partir de un id hasta que todos hayan sido pedidos. Esto significa que mientras un cliente está refresheando, otros pueden postear tweets y estos son recibidos por el cliente refresheando, a pesar de que el comando de refresh haya sido ejecutado antes que los de postear. Esto no lo consideramos un inconveniente, al contrario, es un comportamiento positivo pues permite conseguir tweets más recientes.

README

FIFO

1- Hacer make fifo.

2- Ejecutar "./database.bin (db_fifo_name)"

3- Ejecutar "./server.bin (sv_fifo_name) (db_fifo_name)"

4- Ejecutar n clients en otras terminales pasando como argumento fifo del servidor ("./client.bin (sv_fifo_name)").

SOCKETS

1- Hacer make sockets.

2- Ejecutar "hostname" para averiguar el nombre de host de la computadora.

3- Ejecutar "./database.bin (hostname):(db_port)"

4- Ejecutar "./server.bin (hostname):(sv_port) (hostname):(db_port)"

5- Ejecutar n clients en otras terminales pasando como argumento el socket del servidor ("./client.bin (hostname):(sv_port)").

NOTA: si se prefiere usar launcher.sh solo basta con haber compilado con la implementación de IPC deseada (make [sockets | fifo]) y ejecutar ./launcher.sh arg1 arg2 donde arg1 y arg2 dependen de la implementación de IPC. El launcher ejecutará ./database.bin arg2 & y ./server.bin arg1 arg2 &