TP3: Entradas de cine

Santiago Alvarez Juliá, *Padrón Nro. 99522*
75.42 Taller de Programación
Facultad de Ingeniería, Universidad de Buenos Aires

Octubre 2018

Índice general

0.1. Introducción

En las siguientes secciones abordaré algunos temas claves para la resolución del programa Entradas de cine. Estos serán :

- Programación Orientada a Objetos (POO)
- Protocolo de comunicación
- Arquitectura servidor multi-cliente
- Excepciones

0.2. Temas Claves

0.2.1. Programación Orientada a Objetos

Excepto las respectivas funciones main del cliente y el servidor, el resto del programa fue diseñado en base a objetos que se relacionan entre si. A continuación haré una breve descripción de cada objeto de la aplicación.

- server_Thread: este objecto encapsula la clase std::thread (que representa un hilo de ejecución). Dentro del método start llamo al constructor de inicialización de std::thread que recibe como parámetros un puntero al método run de Thread y un puntero al objecto para el cual esta definido el método run. Dicho método run es virtual puro, por lo tanto Thread es una clase abstracta. Esto permite que cualquier objeto pueda correr en su propio hilo mientras herede de Thread e implemente el método run.
- server_Sala y server_Pelicula: ambos objetos contienen toda la información de la sala y de la película respectivamente en sus atributos. Ambos objetos solo implementan métodos getters.
- server_Funcion: al igual que los anteriores, contiene la información de la función en sus atributos. Además para representar los asientos de la función utilice un std::vector de std::vector de char, en el cual el primer std::vector representa la fila y el segundo la columna. Como las filas están alfabetizadas, server_Funcion tiene un método privado llamado fila_to_number() que utiliza el código ascii de los caracteres para convertir los caracteres a ints y poder acceder a una fila en el std::vector por medio de un índice. En este objeto se implementa la reserva de asientos y el estado de la función (si está agotada o no).

- server_Multi_Client_Acceptor: es un objeto que hereda de server_Thread, por lo que representa un hilo de ejecución y tiene su propia implementación del método run. En detalle en la sección Arquitectura servidor multi-cliente.
- server_ThreadServer: ídem server_Multi_Client_Acceptor. En particular se encarga de la comunicación entre el servidor y un cliente específico mediante un common_Socket.
- server_FuncionesProtected: este objecto encapsula al std::map que tiene como valores server_Funcion y un std::map con sus respectivos mutex protectores. En detalle en la sección Arquitectura servidor multi-cliente.
- server_main: lo primero que se hace es el parseo de los archivos de entrada. Para almacenar películas según el genero, idioma y edad utilizo un std::vector de std::multimap (1 multimap por cada clasificación). El multimap tiene como clave un std::string y como valor un server_Pelicula. Por lo tanto tengo cada película en los 3 multimap, lo cual para mí esta bien porque pienso al servidor como una base de datos donde importa la velocidad (el método equal_range de multimap es O(log n)) y además se podría decir que me devuelve solo los datos que consulté (realmente me devuelve un par de iteradores, por lo que podría acceder a datos que no pedí, pero da esa sensación).
- server_lock: es una encapsulación RAII de la toma y la liberación del recurso mutex.
- client_main: parsea el std::cin para obtener las funciones que quiere ejecutar el cliente y delega la implementación de éstas en client_Client.
- client_Client: recibe los datos que vienen desde client_main que le tiene que enviar al socket del servidor. Dependiendo de cada consulta, se llevan a cabo distintos send y receive del socket y también se imprimen por std::cout y std::cerr algunos resultados de dicha comunicación con el servidor.
- common_socket: es una encapsulación RAII del socket que implementa todas las funcionalidad necesarias (tanto para el cliente como para el servidor).
- common_SocketError y server_ArchivoEntradaError: ambas son excepciones que heredan de std::exception. En detalle en la sección Excepciones.

0.2.2. Protocolo de comunicación

Como los programas se comunican en su mayoría enviando strings (como por ejemplo el título de una película), en esa mayotía de casos envío paquetes con el formato:

— largo — payload —

Donde 'largo' es un entero de 4 Bytes en big-endian, y 'payload' es el texto a enviar, que tendrá el largo especificado en los Bytes que lo preceden.

Para los siguientes casos utilice otra manera de comunicarse entre el servido y el cliente, principalmente porque me parecía que podía enviarse menos bits de información de esta manera:

- comando del cliente: como los comando posibles son acotados, preferí mandar 1 solo byte que represente a cada funcion. Por ejemplo si el cliente le envía al servidor el caracter '1' al iniciar la comunicación esto significa que el cliente quiere conocer el título de las películas de cierto GENERO (diferentes caracteres representan distintos comandos). Para enviar al género en si (por ejemplo TERROR) utilizo el protocolo de — largo — payload —. Este protocolo es utilizado en las clases client_Client (al comienzo de todos sus método) y server_ThreadServer (al comienzo de su método run).

- validez operación (comandos genero, idioma o edad): existe la posibilidad de que el ususarion ingrese un genero, idioma o edad inexistentes en el servidor. Por lo tanto, justo después de recibir el genero, idioma o edad en si, el servidor envía un entero de 4 bytes (ahora me doy cuenta que podría ser como en el caso del comando del cliente que con 1 byte alcanza ya que hay 2 respuestas posibles) indicando la validez del comando.

- estado reserva asiento: (al igual que en validez operación me doy cuenta que con 1 byte alcanza ya que hay 2 respuestas posibles) el servidor para confirmar la reserva del asiento envía un entero de 4 bytes que indica si la reserva fue exitosa o hubo un error (el asiento había sido reservado anteriormente). Este protocolo es utilizado en el método reservar_asiento de client_Client.

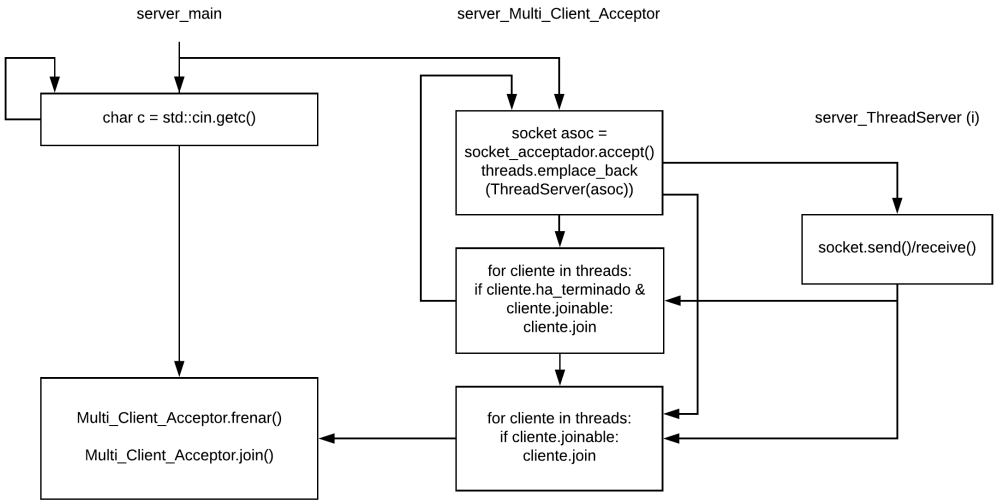
- cantidad filas y columnas (graficar asientos): cuando el cliente le pide los datos de los asientos de una función en particular al servidor, este le envía en enteros de 4 bytes la cantidad de filas y columnas que posee la sala de dicha función para facilitar el gráfico de los asientos. Este protocolo es utilizado en el método asientos_funcion de client_Client.

0.2.3. Arquitectura servidor multi-cliente

En este programa, hay ejecución de código en distintos hilos únicamente en la parte del servidor. En server_main se lanza el hilo server_Multi_Client_Acceptor cuya implementación del método run es un while que finaliza cuando el usuario del servidor ingresa 'q' por entrada estándar (existe otro while, esta vez en server_main que lee std::cin, que cuando sea el caso llama al método stop de server_Multi_Client_Acceptor). Cuando el usuario ingresa 'q', se dejan de escuchar conexiones nuevas (se cierra la conexión del socket aceptador), se termina de atender a los clientes que ya estÃ¡n conectados, y luego se cierra la ejecución de todo el programa del servidor ordenadamente. Dentro del while de server_Multi_Client_Acceptor el socket principal o aceptador acepta conexiones con nuevos clientes, una vez concretada dicha conexión se lanza el hilo server_ThreadServer que tiene como uno de sus parámetros el socket asociado al cliente (el devuelto por el accept del socket principal del servidor). Por lo tanto existe un std::vector de server_ThreadServer que representan distintos clientes haciendo sus respectivas consultas al servidor.

Cuando es creada una instancia de server_ThreadServer esta recibe como parámetros para construirse el std::vector de multimaps que contiene la información de las películas y una instancia de server_FuncionesProtected. Por lo tanto ambas estructuras son recursos compartidos que deben ser protegidos cuando se utilizan hilos (del lado del servidor, std::cout y std::cerr son utilizados únicamente en el main thread, por lo tanto no hay que protegerlos). Además las estructuras mencionadas anteriormente deben ser tratadas de distinta manera. El std::vector de multimaps, que tienen como clave un std::string y como valor server_Pelicula, son unicamente getters. Nunca son modificados dichos multimaps, es decir no se modifican ni las películas internamente ni pueden ser agregadas o borradas películas (excepto en el main thread antes de lanzar el resto de los hilos) y el método de multimap que utilizo para recorrelo es equal_range que por documentacion es thread safe. Por lo argumentado creo que no es necesario proteger este recurso compartido.

Diferente es la situación para la instancia de server_FuncionesProtected que al contener un map donde el valor es un objeto del tipo server_Funcion, debe ser protegida cada función por separado con un std::mutex. En este caso el std::vector de asientos es modificable durante la ejecución de hilos por lo que todos los metodos de la función que utilicen dicho atributo deben ser protegidos. En cada consulta relacionada con las funciones que el cliente tiene la posibilidad de hacer accede a los asientos, sea para saber si estan todos ocupados o para reservar un asiento. Entonces en todos los métodos



de server_ThreadServer que responden a las consultas de los clientes relacionadas con las funciones son lockeados los mutex de la función específica, permitiendo a otros hilos utilizar otras funciones.

0.2.4. Excepciones

- server_ArchivoEntradaError: esta excepción es lanzada cuando se encuentra un error en el parseo de los archivos de entrada. Estos errores suceden cuando se hace referencia en el archivo de funciones a una sala o a una película que no existen en sus respectivos archivos. El constructor recibe como parámetro un std::string que explica cual fue el error, el cual se obtiene mediante el método what en forma de const char* y luego es impreso en std::cerr como especifica el enunciado (en el main thread del servidor).

- common_SocketError: esta excepción es lanzada cuando se encuentra un error en el objeto common_socket. Puede lanzarse en los siguientes metodos: constructor, connect, bind_and_listen y accept. Los motivos por los que suceden estos errores son diversos. El constructor recibe como parámetro un std::string que explica cual fue el error, el cual se obtiene mediante el método what en forma de const char* (en este caso el error no es impreso en std::cerr, sino serÃa necesario proteger std::cerr ya que este error podría ser atrapado en un hilo distinto al main thread del servidor).