My Little Middleware

Ejercicio N°3

Objetivos	● Diseño y construcción de sistemas con acceso distribuido ● Encapsulación de Threads y Sockets en Clases ● Definición de protocolos de comunicación ● Protección de los recursos compartidos ● Uso de buenas prácticas de programación en C++
Instancias de Entrega	**Entrega 1:** clase 7 (26/10/2021). **Entrega 2:** clase 9 (09/11/2021).
Temas de Repaso	● Definición de clases en C++ ● Contenedores de STL ● Excepciones / RAII ● Move Semantics ● Sockets ● Threads
Criterios de Evaluación	● Criterios de ejercicios anteriores ● Eficiencia del protocolo de comunicaciones definido ● Control de paquetes completos en el envío y recepción por Sockets ● Atención de varios clientes de forma simultánea ● Eliminación de clientes desconectados de forma controlada

***El trabajo es personal**: debe ser de autoría completamente tuya. Cualquier forma de **plagio es inaceptable**: copia de otros trabajos, copias de ejemplos de internet o copias de tus trabajos anteriores (self-plagiarism).*
Si usas material de la cátedra deberás dejar en claro la fuente y dar crédito al autor (a la materia).

Índice

Introducción

El enunciado de este ejercicio va a ser leído en papel, o bien utilizando diversos hardwares, y va a ser resuelto en varios de ellos utilizando bibliotecas de sockets y syscalls que a más “bajo nivel” están implementadas de tal manera que se adaptan a cada uno de estos hardwares. Sin embargo, el código que se escribe y ejecuta en uno de ellos, puede ser también compilado y ejecutado en los demás. Esto es posible gracias a la existencia de los **sistemas operativos**, a los **lenguajes portables**, y a que las mencionadas bibliotecas cumplen determinados contratos (a.k.a. APIs) que sirven como una **capa de abstracción** entre nuestros programas y los mencionados hardwares heterogéneos.

Uno de los conceptos que introdujimos en esta materia son los sockets, que son la piedra fundamental a la hora de implementar sistemas distribuidos (como el cliente-servidor) en los que varios procesos (usualmente en distintas computadoras) intentan resolver una problemática común: persiguen el mismo objetivo. También

sabemos que los sockets TCP nos brindan el control de la transmisión de flujos de bytes, pero no tienen el concepto de “mensaje”, sino que nosotros necesitamos crear un protocolo de comunicación de más alto nivel. Para organizar el pensamiento de los programadores, se inventó el concepto de **middleware**, que es una **capa de abstracción** entre la lógica del programa distribuido y cada uno de nuestros procesos/máquinas. Los middlewares tienen distintos tipos, entre los que se destacan aquellos que son orientados a mensajes (*Message Oriented Middleware, o MOM*).

Los protocolos de colas de mensajes son ampliamente utilizados para implementar sistemas distribuidos de gran escala, por su simplicidad de uso e interoperabilidad. Protocolos como AMQP, MQTT o SOMTP se vuelven indispensables a la hora de implementar un sistema distribuido de un tamaño real, y las implementaciones comerciales de este tipo de productos manejan tolerancia a fallos y features muy avanzados, de manera de simplificar la complejidad del resto del sistema.

En este ejercicio implementaremos una versión muy simplificada de este tipo de middleware, que nos permitirá crear y utilizar colas de mensajes para poder implementar un sistema distribuido particular, pero que no contemplará ningún protocolo estándar de mensajería, ni tampoco tolerancia a fallos o redundancia como los MOM comerciales.

El “sistema distribuido particular” será la entrada de nuestro sistema, y ejecutará operaciones muy simples, como agregar o sacar cosas de una cola.

Descripción

Tendremos que implementar un sistema cliente-servidor, en el que el servidor será el “dueño” de las colas de mensajes, de manera que sean accesibles desde todos los clientes. Por su parte, el cliente deberá leer por entrada estándar una serie de comandos, y enviarlos al servidor respetando un determinado protocolo que se describe más adelante.

Los comandos a implementar son los siguientes:

Definir una Cola de Mensajes

```
define <nombre-de-la-cola>
```

El cliente va a usar este comando cuando necesite una nueva cola de mensajes, y va a utilizar el objeto resultante para interactuar con el middleware. Por su parte, el middleware va a crear la cola si ésta no existe, y no le devolverá nada al cliente. Una cola puede ser definida por varios clientes, y todos podrán tanto agregar como retirar mensajes. Una vez que un mensaje fue consumido por un cliente, no podrá ser visto por los demás.

Ayuda: como las colas van a ser accedidas y declaradas desde distintos clientes, el servidor deberá tener algún mecanismo de sincronización para el acceso al lugar donde se guardan las colas. No dejes de plantear un Monitor para acceder a ese recurso!

Agregar un Mensaje a una Cola

```
push <nombre-de-la-cola> <mensaje-como-string>
```

Como se puede notar, las colas contendrán solamente strings. El cliente enviará un mensaje al middleware, para que lo agregue a la cola asociada con el objeto.

Ayuda: nuevamente, las colas van a ser accedidas de forma concurrente desde los distintos clientes, pero cuando ya obtuvimos la cola a usar no hace falta proteger todo el contenedor sino solamente la cola. Fijate dónde poner cada mutex para evitar una contención excesiva.

Consumir un Mensaje de una Cola

```
pop <nombre-de-la-cola>
```

Es análogo al anterior, y va a **imprimir el mensaje por salida estándar**. Por supuesto, si un cliente agrega un mensaje a una cola, los demás clientes que la hayan definido tienen que poder consumirlo. **Notar** que si un cliente consume un mensaje de la cola, los demás ya no lo podrán consumir. Si la cola no tiene mensajes para consumir, el cliente quedará bloqueado en una llamada a **recv**. Para más detalles de esto, ver la sección del protocolo.

Salir

```
exit
```

El único uso de este comando es avisar al cliente que no recibirá más comandos. Será el indicador para dejar de leer comandos y cerrar el programa ordenadamente. Este comando no implica enviarle un mensaje al servidor.

Formato de Línea de Comandos

Como indica la sección anterior, este ejercicio se compone de dos ejecutables: “client” y “server”. El servidor se ejecuta de la siguiente manera:

```
./server <service>
```

Y el cliente de esta otra manera:

```
./client <host> <service>
```

Códigos de Retorno

Ambos procesos deberán retornar uno de los siguientes valores:

- 0 si se pudo completar la ejecución sin problemas.
- 1 si se recibieron parámetros incorrectos.
- 2 si se atrapa alguna excepción que no corresponda a los parámetros de la línea de comandos.

Entrada y Salida Estándar

El cliente la utilizará para leer los comandos del usuario. El formato de cada comando se describe en la sección *Descripción*.

El servidor, por su parte, leerá de entrada estándar para recibir el comando de cierre del sistema, representado por un caracter ‘q’. Cuando el servidor lee la letra ‘q’ debe dejar de aceptar conexiones entrantes, esperar a que se termine de atender a los clientes que ya se están atendiendo, y por último cerrar ordenadamente.

Ayuda: Para dejar de aceptar conexiones entrantes, podés usar shutdown sobre el socket que está aceptando.

Protocolo

Los comandos van a ser identificados con un caracter, y van a enviar un string por cada parámetro, precedidos por su largo en dos bytes big endian.

Ayuda: Una clase Protocolo que haga exactamente lo que dice esta sección es fácil de programar y te permite evitar problemas, tanto de acoplamiento como bugs difíciles de encontrar cuando tengas el resto del TP hecho.

Veamos cada comando:

Definir una Cola de Mensajes

El caracter identificador será una ‘d’ (ASCII 0x64). Entonces pasemos a un ejemplo para explicar el formato completo. Si el usuario del cliente escribe esto:

```
define UnaCola
```

El cliente deberá escribir en el socket un caracter ‘d’, que será el identificador del comando:

```
64
```

Luego el largo del mensaje, en 2 bytes, en big endian (7):

```
64 00 07
```

Y luego el nombre de la cola (en 7 bytes):

```
64 00 07 55 6e 61 43 6f 6c 61
```

Este mensaje **no tiene** respuesta por parte del servidor.

Agregar un Mensaje a una Cola

El caracter identificador será una ‘u’ (ASCII 0x75). Es muy similar al anterior, pero agrega un nuevo string, que es el mensaje a agregar. Digamos, siguiendo el ejemplo, que el mismo usuario agrega un mensaje a la cola que definió.

```
push UnaCola UnMensaje
```

El cliente deberá escribir en el socket un caracter ‘u’, que será el identificador del comando:

```
75
```

Luego el largo del mensaje, en 2 bytes, en big endian (7):

```
75 00 07
```

Y luego el nombre de la cola (en 7 bytes):

```
75 00 07 55 6e 61 43 6f 6c 61
```

Luego, el largo del mensaje a enviar, en 2 bytes, en big endian (un 9):

```
75 00 07 55 6e 61 43 6f 6c 61 00 09
```

Y por último, los 9 bytes que representan al mensaje:

```
75 00 07 55 6e 61 43 6f 6c 61 00 09 55 6e 4d 65 6e 73 61 6a 65
```

Este mensaje **no tiene** respuesta por parte del servidor.

Consumir un Mensaje de una Cola

El caracter identificador será una ‘o’ (ASCII 0x6f). Digamos que otro usuario necesita consumir un mensaje de la misma cola, entonces escribirá en la entrada estándar de su cliente.

```
pop UnaCola
```

El cliente deberá escribir en el socket un caracter ‘o’, que será el identificador del comando:

```
6f
```

Luego el largo del mensaje, en 2 bytes, en big endian (7):

```
6f 00 07
```

Y luego el nombre de la cola (en 7 bytes):

```
6f 00 07 55 6e 61 43 6f 6c 61
```

El servidor, entonces, contestará con un string que representa el resultado del pop. En este caso de ejemplo, el servidor enviará el largo del mensaje

```
00 09
```

Y luego, los 9 bytes que representan al mensaje:

```
00 09 55 6e 4d 65 6e 73 61 6a 65
```

Salir

Si un usuario quiere dejar de usar su cliente, deberá escribir:

```
exit
```

En este caso el cliente no enviará nada al servidor sino que se cerrará ordenadamente, y el servidor se enterará que el cliente terminó porque notará que cerró el socket.

Ayuda: ¿Qué función retorna 0 cuando se cierra el socket del otro lado?

Ejemplos de Ejecución

Ejemplo 1

Retomemos el ejemplo con el que explicamos el protocolo, pero ahora desde las consolas.

Lo primero que tendremos que hacer será levantar un servidor. Usemos el puerto 7777:

```
./server 7777
```

Una vez que el server esté levantado, podemos correr dos clientes en paralelo (a diferencia del TP1, en este el server debe ser capaz de atenderlos al mismo tiempo). Para identificarlos, vamos a escribir lo relacionado con el primero en azul, y el segundo en rojo.

Levantamos el primero:

```
./client localhost 7777
```

Y el segundo:

```
./client localhost 7777
```

Ahora que ambos clientes están levantados, definamos la cola “UnaCola” en ambos. Entonces en la entrada estándar del cliente azul escribimos:

```
define UnaCola
```

Entonces el socket del rojo verá pasar estos caracteres:

```
64 00 07 55 6e 61 43 6f 6c 61
```

Ahora lo mismo en la del rojo:

```
define UnaCola
```

Entonces el socket del rojo verá pasar estos caracteres:

```
64 00 07 55 6e 61 43 6f 6c 61
```

Ayuda: Para validar esto, te podés ayudar con tests, con netcat o con tiburoncin! El Sercom usará este último.

Ahora, usaremos el cliente azul para agregar un mensaje a una cola:

```
push UnaCola UnMensaje
```

Entonces, el cliente azul le enviará estos bytes al servidor:

```
75 00 07 55 6e 61 43 6f 6c 61 00 09 55 6e 4d 65 6e 73 61 6a 65
```

Mientras tanto, el cliente rojo querrá consumir el mensaje. Entonces escribirá:

```
pop UnaCola
```

Y enviará esto por su socket:

```
6f 00 07 55 6e 61 43 6f 6c 61
```

En este caso, como es un pop, el server le contestará con el mensaje:

```
00 09 55 6e 4d 65 6e 73 61 6a 65
```

Y el cliente imprimirá ese mensaje por su salida estándar, seguido de un salto de líneas:

```
UnMensaje
```

Nota y ayuda: ¡Si el mensaje del pop le llega al servidor antes que el push, la salida debe ser la misma! Esto lo podés lograr usando colas bloqueantes en el server (en este caso vale reutilizar la cola que usaste en el TP2).

Ejemplo 2

En este caso no se repasa un ejemplo completo, sino que se muestra cómo levantar el ejemplo anterior con tiburoncin, porque hay que prestar atención a los puertos.

Primero, levantamos el server, usaremos el puerto 7777.

```
./server 7777
```

Ahora, vamos a necesitar un tiburoncin para que haga de man-in-the-middle entre el cliente azul y el server:

```
tiburoncin -A 7778 -B 7777
```

En este punto, el tiburoncin azul ya hizo un bind sobre el puerto 7778, entonces si levantamos otro tiburoncin con el mismo parámetro A, vamos a caer en un error (Address Already in Use), entonces para evitar ese error, levantamos otro tiburoncin en el puerto 7779.

```
tiburoncin -A 7779 -B 7777
```

Y ahora levantamos nuestros clientes:

```
./client localhost 7778
```

Y el rojo:

```
./client localhost 7779
```

A partir de aquí, podemos seguir con el ejemplo anterior con visibilidad de qué mensajes están mandando los sockets.

Restricciones

La siguiente es una lista de restricciones técnicas exigidas por el cliente:

1. El sistema debe desarrollarse en C++ (C++11) con el estándar POSIX 2008.
2. Está prohibido el uso de variables globales.
3. Todo socket utilizado en este TP debe ser **bloqueante** (es el comportamiento por defecto).
4. El acceso a los recursos compartidos por varios threads debe estar encapsulado en monitores siguiendo los lineamientos que se vieron en clase.
5. Ningún método de la solución puede tener más de 10 líneas. Si creés que es más prolijo un método particular con más de 10 líneas, debe estar justificado en la documentación del método.

Aclaraciones

1. Debido a que este TP levanta múltiples clientes concurrentemente, no se provee un archivo by-example como en los anteriores.
2. ¡Esto no es un Message Oriented Middleware completo! Sin embargo, podés usarlo como punto de partida para investigar más. ¡En la orientación de distribuidos se usan mucho!

Referencias

Podés leer wikipedia para saber un poco más sobre las capacidades de un MOM completo:
https://en.wikipedia.org/wiki/Message-oriented_middleware