Administración de reservaciones online para restaurantes

Ejercicio N° 2

Objetivos	• Diseño y construcción de sistemas con procesamiento concurrente • 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
Instancias de Entrega	**Entrega 1:** clase 6 (14/4/2015). **Entrega 2:** clase 8 (28/04/2015).
Temas de Repaso	• POSIX Threads • Funciones para el manejo de Sockets • Uso básico de clases en C++
Criterios de Evaluación	• Criterios de ejercicios anteriores • Ausencia de secuencias concurrentes que permitan interbloqueo • Ausencia de condiciones de carrera en el acceso a recursos • Buen uso de Mutex, Condition Variables y Monitores para el acceso a recursos compartidos • Eficiencia del protocolo de comunicaciones definido • Control de paquetes completos en el envío y recepción por Sockets

Índice

Introducción

El objetivo del trabajo práctico es realizar una aplicación cliente-servidor para administrar las reservaciones de un restaurante.
El sistema permitirá a los clientes consultar la oferta de fecha, hora y cantidad de lugares disponibles en el restaurante, realizar reservas para una determinada cantidad de personas y cancelar reservaciones realizadas con anterioridad.
La información referente al restaurante (cantidad de lugares disponibles, fechas y horas, reservas realizadas) estará contenida en un archivo.

Se pide realizar dos programas: servidor y cliente. El servidor será el encargado de administrar la información del restaurante mientras que el cliente consultará los datos al servidor y efectuará reservaciones y/o cancelaciones.

Detalle

Formato archivo del restaurante

Los datos de cada turno a ofrecer para reserva se separa por un caracter de nueva línea '\n'.
El formato utilizado es el siguiente:
`<fecha-hora> |<disponibilidad>|[|<id_cliente>|<cant_comensales>|...|<id_cliente>|<cant_comensales>|...|<id_cliente>|<cant_comensales>]\n`

* `fecha-hora`: tiene el siguiente formato: DD/MM-HH:MM
* `disponibilidad`: es numérico y no supera los 3 dígitos.
* `id_cliente`: es el dni del cliente
* `cant_comensales`: es el número de comensales que reserva el cliente

Servidor

Debe recibir como argumento de entrada el puerto por el que escuchará conexiones entrantes y el nombre de un archivo de texto del que se cargarán los datos existentes sobre la disponibilidad de reservas del restaurante y se guardarán nuevamente al finalizar su ejecución.

Ejecución del servidor: `./servidor <puerto> <nombre del archivo del restaurante>`

Al iniciar el programa, se debe leer el archivo con la información del restaurante. A cada turno de reservación disponible leído del archivo se le deberá asignar un ID numérico (número natural), válido para la sesión actual. De no existir el archivo, el programa debe crearlo.
Con el objetivos de simplificar el desarrollo del trabajo práctico, no se plantean restricciones especiales

sobre el almacenamiento de los datos de restaurantes ni sobre los órdenes de ejecución en búsquedas.
Se permite el uso de contenedores dinámicos de la Standard Template Library (STL) pero se aclara que no son necesarios y es posible resolver el sistema con simples arreglos dinámicos de C++.
El servidor debe recibir conexiones simultáneas y atender los pedidos de los distintos clientes y cerrarse al ingresar por entrada estándar el caracter 'q'.
El servidor retorna 0 siempre.

El programa enviará información respondiendo a las peticiones realizadas por los clientes. Todas las peticiones deben tener una respuesta. Las posibles respuestas del servidor son:

```
ok
error
<listado de reservas realizadas por un cliente>
<listado de disponibilidad de reservas en el restaurante>
```

Para enviar el listado de restaurantes se utilizará el siguiente formato:

```
<id_item_sesion>\t<fecha-hora>\t(<disponibilidad>)
```

Para enviar el listado de reservas se utilizará el siguiente formato:

```
<id_cliente>\t<fecha-hora>\t<cant_comensales>
```

id_item_sesion: es el identificador del restaurante asignado al obtener los datos del archivo. Este valor deberá ser formateado para ocupar 4 caracteres. (Ej 0001)

Si el servidor recibe un comando que no reconoce o que está mal formado, envía al cliente el mensaje “error”.
Si no hay datos para satisfacer el pedido del cliente, también envía el mensaje “error”.
El mensaje “ok” será enviado en respuesta a una actualización exitosa (reserva o cancelación).

Cliente

Debe recibir como argumento de entrada la IP del servidor y el puerto en el que el mismo escucha conexiones.

Ejecución del cliente: `./cliente <ip del servidor>:<puerto del servidor>`

El cliente se conectará al servidor, y podrá realizar las siguientes acciones:

`L`
Pedir el listado de fechas, hora y disponibilidad para realizar las reservas. (id, fecha-hora y disponibilidad)

`R <id_cliente> <id_item_sesion> <cant_comensales>`
Reservar para el dia y hora que tiene el id `id_item_sesion` la cantidad `cant_comensales`

`C <id_cliente>`
Consultar las reservas realizadas

`B <id_cliente> <id_item_sesion>`
Cancela la reservación realizada para la fecha y hora que tiene el id `id_item_sesion`

`S`
Finaliza la ejecución del cliente con valor de retorno 0.

El cliente deberá imprimir por pantalla todas las respuestas recibidas del servidor.
En caso de detectar un cierre de conexión por parte del servidor, el cliente deberá finalizar con valor de retorno 1.

Protocolo a utilizar entre el cliente y el servidor

La comunicación entre el cliente y el servidor se llevará a cabo utilizando una cadena de caracteres finalizada con el caracter de nueva línea '\n'.

El servidor enviará información respondiendo a las peticiones realizadas y el cliente debe imprimirlas por salida estándar.
`<data>\n`

Ej: En el caso de responder con el mensaje “error”:
`error\n`

El cliente, en cambio, enviará información que el servidor deberá interpretar, por lo tanto enviará el comando y los parámetros que requiera, separando todos los campos con pipes '|', finalizando con el caracter de nueva línea '\n'.

`<comando>[|<parametro_1>|..|<parametro_n>]\n`

Ej: En el caso de realizar una reserva con ID 0001: `R 30111222 0001 4`
`R|30111222|1|4\n`

Existe un caso particular que es el listado: como el protocolo finaliza con un caracter de nueva línea, no pueden enviarse caracteres de nueva línea dentro del mensaje, por lo cual, la información de cada turno será separada por el byte 0x03 ((char)3), y finalmente al terminar de armar el listado, se debe agregar el caracter de nueva línea, como indica el protocolo.

Para poder imprimir en pantalla un item de disponibilidad de reserva por línea, es necesario que del lado del cliente se parsee el listado reemplazando los caracteres 0x03 por el caracter de nueva linea. Ejemplo:

El servidor envía el listado de esta forma:

`0001\t05/04-22:30\t(23)0x030002\t06/04-22:00\t(10)0x030003\t07/04-20:30\t(1)0x030004\t20/04-21:30\t(6)0x030005\t23/04-21:00\t(47)0x030006\t25/04-22:00\t(5)0x030007\t14/05-21:15\t(8)0x03\n`

Y el cliente, parsea el mensaje reemplazando los 0x03 por \n:

```
0001\t05/04-22:30\t(23)\n0002\t06/04-22:00\t(10)\n0003\t07/04-20:30\t(1)\n0004
\t20/04-21:30\t(6)\n0005\t23/04-21:00\t(47)\n0006\t25/04-22:00\t(5)\n0007\t14/
05-21:15\t(8)\n\n
```

Luego, envía a la salida estándar el listado:

```
0001  05/04-22:30     (23)
0002  06/04-22:00     (10)
0003  07/04-20:30     (1)
0004  20/04-21:30     (6)
0005  23/04-21:00     (47)
0006  25/04-22:00     (5)
0007  14/05-21:15     (8)
```

Ejemplo de archivo del restaurante con disponibilidades y reservas realizadas:

```
05/04-22:30|(23)||30111222|2
06/04-22:00|(10)|
07/04-20:30|(1)||30111222|2|23555666|20|23555999|4|33225566|3
20/04-21:30|(6)|
23/04-21:00|(47)|
25/04-22:00|(5)|
14/05-21:15|(8)|
```

Ejemplos de Ejecución

Servidor

```
# ./servidor 4321 restaurante.txt
```

Indica al servidor que debe escuchar conexiones entrantes en el puerto 4321, debe cargar los datos contenidos en el archivo restaurante.txt y luego, al leer “q” por stdin, escribir los datos actualizados en el mismo archivo y terminar su ejecución.

Cliente

```
# ./cliente 127.0.0.1:4321
```

Indica al cliente que debe conectarse con el servidor que se encuentra escuchando en la dirección ip 127.0.0.1, en el puerto 4321.

Suponiendo que el servidor cuenta con los datos listados en el ejemplo de archivo del restaurante:
(en rojo lo ingresado por entrada estándar en el cliente)
(en negro la petición del cliente, formateada para que pueda interpretarla el servidor)
(en azul las respuestas del servidor que el cliente envía a la salida estándar)

- **Petición 1**

  L

  L

  El servidor debe retornar el listado, con el formato anteriormente explicado:

  ```
  0001  05/04-22:30      (23)
  0002  06/04-22:00      (10)
  0003  07/04-20:30      (1)
  0004  20/04-21:30      (6)
  0005  23/04-21:00      (47)
  0006  25/04-22:00      (5)
  0007  14/05-21:15      (8)
  ```

- **Petición 2**

  R 23333333 0001 2

  R|23333333|1|2

  El cliente reservación para dos personas para el 4 de abril a las 22:30, el servidor debe actualizar los datos y enviar una respuesta

  ok

- **Petición 3**

  R 23333333 0007 4

  R|23333333|7|4

  El cliente realiza otra reservación para 4 personas el 14 de mayo, el servidor debe actualizar los datos y enviar una respuesta. Si se hace más de una reserva con el mismo id de cliente para la misma fecha, se deberán sumar los comensales de la nueva reserva a la anterior y no generar una nueva.

  ok

- **Petición 4**

  R 23333333 0009 3

  R|23333333|9|3

  El cliente quiere realizar una reservación para una opción que no existe, el servidor debe retornar el mensaje “error”

  error

- **Petición 5**

  R 23333334 0003 2

  R|23333334|3|2

  El cliente quiere realizar una reservación para un afecha en la que no hay disponibilidad para esa cantidad de comensales, el servidor debe retornar el mensaje “error”

  error

- **Petición 6**

  C 23333333

  C|23333333

El cliente consulta las reservas realizadas, el servidor debe enviar los datos consultados, si no hay reservas con ese id de cliente, se debe retornar el mensaje “error”

```
23333333   0001  05/04-22:30     2
23333333   0007  14/05-21:15     4
```

- **Petición 7**

  ```
  B 33225566 0003
  ```

  ```
  B|33225566|3
  ```

  El cliente cancela la reservación para el 7 de abril a las 22:30, el servidor debe enviar como respuesta “ok” si esa reservación existía y se canceló correctamente, si no hay reservas para esa fecha con ese id de cliente, se debe retornar el mensaje “error”

  ```
  ok
  ```

- **Petición 8**

  ```
  L
  ```

  ```
  L
  ```

  El servidor debe retornar el listado de la disponibilidad para reservar:

  ```
  0001  05/04-22:30     (21)
  0002  06/04-22:00     (10)
  0003  07/04-20:30     (4)
  0004  20/04-21:30     (6)
  0005  23/04-21:00     (47)
  0006  25/04-22:00     (5)
  0007  14/05-21:15     (4)
  ```

- **Petición 9**

  ```
  S
  ```

  ```
  S
  ```

  Cierra la aplicación.

Estado final del archivo del servidor

Finalmente cuando se cierra el servidor, al leer “q” por entrada estándar, se deben grabar los datos actualizados en el archivo:

```
05/04-22:30|(21)||30111222|2|23333333|2
06/04-22:00|(10)|
07/04-20:30|(4)||30111222|2|23555666|20|23555999|4
20/04-21:30|(6)|
23/04-21:00|(47)|
25/04-22:00|(5)|
14/05-21:15|(4)||23333333|4
```

Restricciones

- Los sockets a utilizar deben ser bloqueantes y trabajar bajo protocolo TCP.
- Los threads y mutexes a utilizar deben ser de la implementación de hilos de POSIX, conocida como pthreads [1].
- El empleo de sockets debe ser abstraído en una o más clases según las recomendaciones indicadas por la cátedra.
- El empleo de threads y mutexes también debe ser abstraído según las recomendaciones de la cátedra.
- Programar usando ISO C++ 98

Sugerencias para la Resolución del Trabajo

Este suele ser el trabajo individual que más tiempo demanda de los alumnos, no por la complejidad, sino porque introduce temas nuevos como threads y sockets. Es común que el alumno encare el trabajo intentando incorporar todos los elementos desde el principio y, al no tener los conceptos claros, resulta mucho más difícil resolverlo.
A continuación se detalla una serie de recomendaciones para organizar la resolución del trabajo y optimizar los tiempos necesarios:

1. **Implementación y Prueba de Sockets:**
   La mejor forma de implementar sockets y threads en este trabajo es haciéndolo de manera iterativa e incremental. En esta etapa debemos olvidar los threads, mutex y el modelo de negocio en sí.
   Lo primero que tenemos que hacer es crear las clases necesarias para manejar los sockets, ver toda la funcionalidad que se necesita exclusivamente en el servidor, la que se necesita para los clientes y la que es común a ambas.
   Una vez que tenemos las clases de sockets armadas, creamos pequeños programas para hacer uso de estos sockets: un programa cliente y otro servidor. Los programas deben ser simples, sin más complejidad que un chat, donde un server que escucha sobre cierto puerto y un solo cliente que se conecta y envía un mensaje (un simple string) para que el servidor le responda con otro mensaje.

2. **Primera Versión del Modelo de Negocio:**
   En este punto podemos agregar una versión simple del protocolo a utilizar. Por ejemplo, leer caracteres hasta que reciban un ***n**. Esta lógica de negocio debería implementarse en nuevas clases, para así evitar el acoplamiento del protocolo con las clases de sockets. De esta forma es posible reutilizar los sockets para cualquier otro propósito (por ejemplo, manejar comunicaciones en el trabajo final) sin tener que modificarlas al usar un protocolo diferente.

3. **Implementación y Prueba de Threads:**
   Una vez resueltas la comunicación y una primer versión del modelo, necesitamos que el server atienda más de un cliente a la vez, para ello agregaremos manejo de concurrencia con threads.
   Como en el caso de sockets, siempre es recomendable construir las clases necesarias para encapsular el concepto y hacer un sistema muy simple para probarlas. Esta aplicación posee necesariamente un único ejecutable y debe contar como mínimo con el lanzamiento de varios hilos, la operación de los hilos sobre algún recurso compartido (modificación de una variable en el heap o la

escritura por consola), el bloqueo de la sección crítica y la espera de que todos los threads finalizan exitosamente. Este último paso se conoce como ***join*** y es tan importante como un buen manejo de concurrencia y acceso a recursos.

4. **Inclusión de Threads a la Primera Versión del Modelo de Negocio:**

Para incorporar correctamente los threads al trabajo es necesario preguntarse:

- ¿Qué tareas tiene que atender de manera simultánea un servidor?
- ¿Los clientes necesitan threads?
- ¿Qué recursos son accedidos concurrentemente?
- ¿Cuáles son las secciones críticas para el acceso a dichos recursos?
- ¿En qué circunstancias finaliza el programa principal y cómo espera las hilos que lanzó?

Con esto en mente, agregamos threads a nuestra primer versión del modelo de negocio y nos concentramos en verificar que varios clientes pueden acceder simultáneamente al servidor. A su vez, aseguramos la integridad de los datos a los que acceden los clientes concurrentemente mediante la incorporación de mutex.

5. **Mejoras en el Modelo de Negocio:**

En este punto nuestros programas se conectan entre sí y soportan varios clientes. Es momento de mejorar el modelo de negocio y finalizar el trabajo.
Se mejoran entonces los aspectos relacionados con el negocio que nos plantea el enunciado sin que sea necesario modificar las librerías de soporte (clases utilizadas para threads y sockets) ya que confiamos en su correcto funcionamiento. Entre las mejoras necesarias, se puede mencionar:

- Extensiones al protocolo de comunicación para cumplir con las restricciones del enunciado
- Definición de los comandos cliente faltantes para ejecutar en el servidor
- Inicialización de datos en el servidor y almacenamiento del nuevo estado al finalizar
- Validaciones y manejos de error

Para esta tarea es muy importante contar con un conjunto de pruebas sobre los requerimientos del enunciado, incluyendo archivos de ejemplo con distintas secuencias. Es el momento indicado para descargar los casos de test del servidor y ejecutarlos localmente.

En cada una de las etapas es indispensable hacer uso de Valgrind, tanto para el cliente como para el servidor. De esa forma podemos asegurar los recursos fueron liberados correctamente y evitar la acumulación de errores para futuras etapas.

Al encarar el trabajo práctico de esta manera, no sólo se aseguran concluirlo en menos tiempo, sino que también disminuyen el acoplamiento entre módulos [2].

Referencias

[1] http://en.wikipedia.org/wiki/POSIX_Thread
[2] http://es.wikipedia.org/wiki/GRASP