Dune

Ejercicio N°2 - Threads

Objetivos	● Implementación de un esquema cliente-servidor basado en threads. ● Encapsulación y manejo de Threads en C++ ● Comunicación entre los threads via Monitores y Queues.
Entregas	● **Entrega obligatoria:** clase 7. ● **Entrega con correcciones:** clase 9.
Cuestionarios	● Threads - Recap - Programación multithreading
Criterios de Evaluación	● Criterios de ejercicios anteriores. ● Resolución completa (100%) de los cuestionarios *Recap.* ● Cumplimiento de la **totalidad** del enunciado del ejercicio incluyendo el **protocolo** de comunicación y/o el **formato** de los archivos y salidas. ● Correcto uso de mecanismos de sincronización como **mutex**, **conditional variables** y colas bloqueantes (**queues**). Protección de los objetos compartidos en objetos **monitor**. ● Prohibido el uso de funciones de tipo *sleep()* como *hack* para sincronizar salvo **expresa** autorización en el enunciado. ● Ausencia de condiciones de carrera (race condition) e interbloqueo en el acceso a recursos (deadlocks y livelocks). ● Correcta encapsulación en objetos **RAII** de C++ con sus respectivos constructores y destructores, **movibles (move semantics) y no-copiables** (salvo excepciones justificadas y documentadas). ● Uso de **const** en la definición de métodos y parámetros. ● Uso de **excepciones** y manejo de errores.

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

Descripción

El juego Dune que implementaremos deberá soportar múltiples jugadores en múltiples partidas. En este ejercicio se implementará la parte en la que los jugadores (clientes) se conectarán al servidor para listar que partidas hay, crear una o unirse a una.

Protocolo

Definamos algunos componentes del protocolo.

`operation`: número de 1 byte sin signo que representa que acción se quiere hacer: unirse a una partida (01), listar partidas (02), crear partida (03)

`house`: número de 1 byte sin signo que representa la casa con la cual jugar: Harkonnen (00), Atreides (01), Ordos (02)

`game-name-len`: es un número de 2 bytes sin signo en big endian que representa la cantidad de bytes del nombre del juego o partida.

`game-name`: es dicho nombre, un string que no termina en null.

`ret`: número de 1 byte sin signo que representa el resultado de la operación: la operación fue un éxito (00), la operación falló (01).

`count`: count es un número de 2 bytes sin signo en big endian que indica cuántos elementos siguen a continuación. Es una forma de enviar una lista de elementos.

`current`: número de 1 byte sin signo que representa cuantos jugadores hay en la partida.

`required`: número de 1 byte sin signo que representa cuantos jugadores son necesarios para considerar a la partida completa.

Unirse a una partida

El jugador quiere unirse a una partida. Para ello indica a cual partida quiere unirse y con que casa quiere jugar.

```
operation house game-name-len game-name
```

El servidor verificará que la partida exista y que aun no este completa y responderá con:

```
ret
```

Listar partidas

El cliente puede querer ver que partidas hay actualmente enviando

```
operation
```

El servidor responde con

```
 count (current required game-name-len game-name)...
```

Nótese como (`current required game-name-len game-name`) se repiten tantas veces como partidas haya, cantidad que es indicada por `count`.

El listado debe estar ordenado por orden alfabético según los nombres de los juegos.

Crear una partida

El jugador crea una partida y se une a ella.

La idea es que el jugador pueda crear una partida indicando cuántos jugadores son necesarios (requeridos) y con qué casa él desea jugar.

```
operation house required game-name-len game-name
```

El servidor validará que la partida no exista (sea nueva) y responderá con:

```
ret
```

Simplificaciones

Queda afuera de esta versión del protocolo más datos que serán necesarios para la versión final del juego como por ejemplo que el servidor liste qué escenarios o mapas hay cargados en el servidor o que el jugador pueda crear una partida usando un escenario particular así como también la posibilidad de darse de baja de una partida esté o no ya comenzada.

La implementación deberá permitir que un mismo jugador se pueda registrar a una misma partida múltiples veces o que pueda unirse a varias partidas. Esto es con el fin de hacer los casos de prueba más sencillos.

Formato de Línea de Comandos

El cliente se ejecutará como:

```
./cliente <hostname> <servicename>
```

Y el servidor se ejecutará como:

```
./server <servicename>
```

Códigos de Retorno

Si hay algún problema con los argumentos pasados se debe retornar un código de 1.

De lo contrario retornar 0.

Entrada y Salida Estándar

La implementación del cliente será muy pequeña: qué acción realizará dependerá de los comandos leídos por entrada estándar.

El cliente deberá ir leyendo de a un comando a la vez. Tras leer un comando deberá ejecutarlo y luego podrá leer el siguiente comando. No se podrá leer todo de la entrada estándar en memoria para luego ir ejecutando todos los comandos.

Así, para *"simular"* a un jugador queriendo listar las partidas, se enviara por entrada estándar el siguiente

mensaje:

```
listar
```

El cliente deberá entonces preguntarle al servidor las partidas disponibles y luego imprimirá por salida estándar, en orden alfabético, las partidas actuales, sus jugadores actuales y los requeridos:

```
duelo 2/2
epic battle 21/50
tres amigos 1/3
```

Para crear una partida:

```
crear <house> <required> <game-name>
```

Por ejemplo:

```
crear Harkonnen 3 tres amigos
```

El cliente imprimirá una de las dos siguientes sentencias:

```
Creacion fallida
Creacion exitosa
```

Para unirse a una partida:

```
unirse <house> <game-name>
```

Por ejemplo:

```
unirse Harkonnen tres amigos
```

El cliente imprimirá una de las dos siguientes sentencias:

```
Union fallida
Union exitosa
```

El cliente deberá aceptar un comando adicional “`fin`” que le indicará que no habrán más comandos y que deberá finalizar.

Por su parte el servidor deberá imprimir por salida estándar el siguiente mensaje cada vez que una partida se complete:

```
Comenzando partida <game-name>...
```

Por ejemplo

```
Comenzando partida tres amigos...
```

Formato

Nótese que el nombre de la partida puede contener espacios sin embargo el parsing de las instrucciones es simple.

Cada instrucción está en su propia línea y la cantidad de palabras es conocida salvo por las últimas que son el nombre de la partida.

También está garantizado que las instrucciones estarán bien formadas, que no habrá nombres de casas inválidos, números negativos y esas cosas.

Finalización

Cuando el cliente encuentre el fin de la entrada estándar o bien el comando “`fin`” deberá cerrar la conexión con el servidor.

El servidor por su parte finalizará cuando lea una `'q'` de su entrada estándar.

Ejemplos de Ejecución

Imaginemos al cliente 1 recibiendo estos comandos:

```
crear Harkonnen 2 duelo
listar
```

El cliente debería enviar entonces al servidor el primer comando, `crear`:

```
030002000564756 56c6f
```

Quien responde con un código de éxito ya que no hubo problemas.

```
00
```

El cliente imprime entonces:

```
Creacion exitosa
```

Y luego envía el segundo mensaje, el pedido de listar las partidas:

```
02
```

Y el servidor responde con:

```
000101020005 6475656c6f
```

Nótese cómo `0001` indica que hay una solo partida (número de 2 bytes en big endian), `01 02` indican 1 jugador de 2 requeridos, `0005` indica que el nombre de la partida tiene 5 letras y `6475656c6f` es el nombre de la partida, *“duelo”* en este caso.

El cliente entonces imprime

```
duelo 1/2
```

Supongamos que el mismo cliente recibe más comandos:

```
unirse Atreides duelo
fin
```

Nótese que por simplificación vamos a permitir que un mismo cliente se pueda unir a una misma partida más de una vez.

Entonces el cliente envía:

```
010100056475656c6f
```

El servidor verifica que la unión es correcta y que además que completa la partida. Entonces el servidor imprimirá

```
Comenzando partida duelo...
```

Y le enviará al cliente

```
00
```

Quien por su parte imprimirá

```
Union exitosa
```

Luego, por recibir el comando “`fin`” el cliente finalizará.

Recomendaciones

Los siguientes lineamientos son claves para acelerar el proceso de desarrollo sin pérdida de calidad:

1. **Repasar las recomendaciones de los TPs pasados y repasar los temas de la clase.** Los videos,

las diapositivas, los handouts, las guías, los ejemplos, los tutoriales.

2. **Verificar** siempre con la **documentación** oficial cuando un objeto o método es *thread safe*. **No suponer.**
3. Hacer algún diagrama muy simple que muestre **cuales son los objetos compartidos** entre los threads y asegurarse que estén **protegidos** con un monitor o bien sean thread safe o **constantes**.
   Hay veces que la solución más simple es no tener un objeto compartido sino tener un objeto privado por cada hilo.
4. **Asegurate de determinar cuales son las critical sections.** Recordá que por que pongas mutex y locks por todos lados harás tu código thread safe. **¡Repasar los temas de la clase!**
5. **¡Programar por bloques!** No programes todo el TP y esperes que funcione. ¡Menos aún si es un programa multithreading!
   Dividir el TP en bloques, codearlos, testearlos por separado y luego ir construyendo hacia arriba. Solo al final agregar la parte de multithreading y tener siempre la posibilidad de "*deshabilitarlo*" (como algún parámetro para usar 1 solo hilo por ejemplo).
   ¡Debuggear un programa single-thread es mucho más fácil!
6. **Escribí código claro**, sin saltos en niveles de abstracción, y que puedas leer entendiendo qué está pasando. Si editás el código "*hasta que funciona*" y cuando funcionó lo dejás así, **buscá la explicación de por qué anduvo**.
7. **Usa RAII, move semantics y referencias.** Evita las copias a toda costa y punteros e instancia los objetos en stack. Las copias y los punteros no son malos, pero deberían ser la excepción y no la regla.
8. No te compliques la vida con diseños complejos. **Cuanto más fácil sea tu diseño, mejor**.

Restricciones

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

1. El sistema debe desarrollarse en C++17 con el estándar POSIX 2008.
2. Está prohibido el uso de **variables globales**, **funciones globales** y **goto**. Para el manejo de errores usar **excepciones** y no retornar códigos de error.
3. Todo socket utilizado en este TP debe ser **bloqueante** (es el comportamiento por defecto) y **no** puede usarse *sleep()* o similar para la sincronización de los threads salvo expresa autorización del enunciado.