Taller de Programación I (75.42)
Facultad de Ingeniería
Universidad de Buenos Aires

Duck Game

Ejercicio / Prueba de Concepto 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 y sin usar AI. 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

En este trabajo haremos una *prueba de concepto* del TP enfocándonos en el uso de *threads*.

Puede que parte o incluso el total del código resultante de esta PoC ***no*** te sirva para el desarrollo final de tu juego. – Y está bien.

En una PoC el objetivo es familiarizarse con la tecnología, en este caso, con los threads. Familiarizarse ***antes*** de embarcarse a desarrollar el TP real te servirá para tomar mejores decisiones.

Descripción

El servidor iniciará una única partida que contiene 4 *Red Item Boxes* (cajas de recompensa). Los jugadores podrán unirse y recoger los premios que estas contienen. Cuando un jugador recoge un premio, la caja desaparece, y reaparece luego de algunos segundos.

Cada vez que se recoge o reaparece una caja, el servidor deberá enviarle un mensaje a **todos** los clientes conectados indicando el evento sucedido.

Ante estos eventos, **tanto los clientes como el servidor** imprimirán, según el caso:

- `<player-name> picked up a <reward-name>`
- `A new box has appeared`

Acciones del cliente

Al iniciar al cliente, este imprimirá por stdout:

- `What is your name?`

El cliente, por su lado, escribe su nombre en una linea.

Luego de leer el nombre, cada cliente deberá leer de entrada estándar que acciones va a realizar. Estas son:

- `Pickup <box-id>`: el cliente deberá enviar un mensaje al servidor indicando su intención de recoger la caja con id `<box-id>`.
- `Read <n>`: el cliente espera a recibir `<n>` mensajes del servidor, imprimiendolos a medida que llegan.
- `Exit`: el cliente debe finalizar.

En una implementación real el cliente estaría enviando y recibiendo mensajes de forma asincrónica y concurrentemente pero para esta PoC vamos a simplificar el diseño: el cliente debe tener un **único thread** (el main).

Ante cada mensaje que el cliente envía, este **no** espera respuesta. Es **solo** cuando ejecuta `Read <n>` que espera por exactamente `<n>` mensajes desde el servidor. Por cada uno de estos `<n>` mensajes imprimirá el evento correspondiente, tal como fue mencionado anteriormente.

Requerimientos del servidor

La lógica del juego **deberá** ejecutarse en **un único thread** que corra un **gameloop**. Este hilo correrá el siguiente loop **5 veces cada segundo**.

1. Leer **todos los comandos pendientes** de los clientes y **ejecutarlos** (recoger las cajas).
   Es importante remarcar que **el gameloop no debe bloquearse.** Para esto, los clientes enviarán sus comandos al gameloop a través de una **única queue del gameloop.**

2. Simular una iteración en el *mundo del juego*. En este paso las cajas pueden reaparecer según su estado. En la siguiente tabla se describe el comportamiento de cada caja.

ID de caja	Contenido	Tiempo de reaparición luego de ser recogida	Cantidad de iteraciones en el *mundo del juego* hasta reaparecer
1	Bazooka	3 seg	15
2	Chainsaw	1 seg	5
3	Death ray	4 seg	20
4	Shotgun	2 seg	10

3. Enviar los mensajes a los clientes. Cada vez que una caja aparece, o es recogida por un jugador, se

deben enviar los mensajes a **todos los clientes**, e imprimir por pantalla lo sucedido. Es un **broadcast**! Este paso puede ejecutarse dentro de los pasos 1 y 2, o luego de haber ejecutado los comandos y simulado la iteración.
Es posible que uno o varios jugadores no puedan recibir el mensaje exactamente en ese momento y por ende `skt.send` se **bloqueara del lado del servidor**. Pero el **gameloop no puede bloquearse!** Es por esto que **cada cliente** en el servidor deberá tener una **queue** de mensajes a enviar a través del socket.

4. Sleep. Dado que queremos que hayan 5 loops por cada segundo, se pedirá que **sólamente aquí y en ningún otro lugar del código** agreguen un **sleep.**

   Nota: En este trabajo, por simplificación, haremos un **sleep de 200 milisegundos** luego de cada loop. Esto no es correcto si buscamos exactamente 5 loops por segundo (como querremos en el trabajo final), dado que los pasos 1, 2 y 3 también consumen tiempo real.

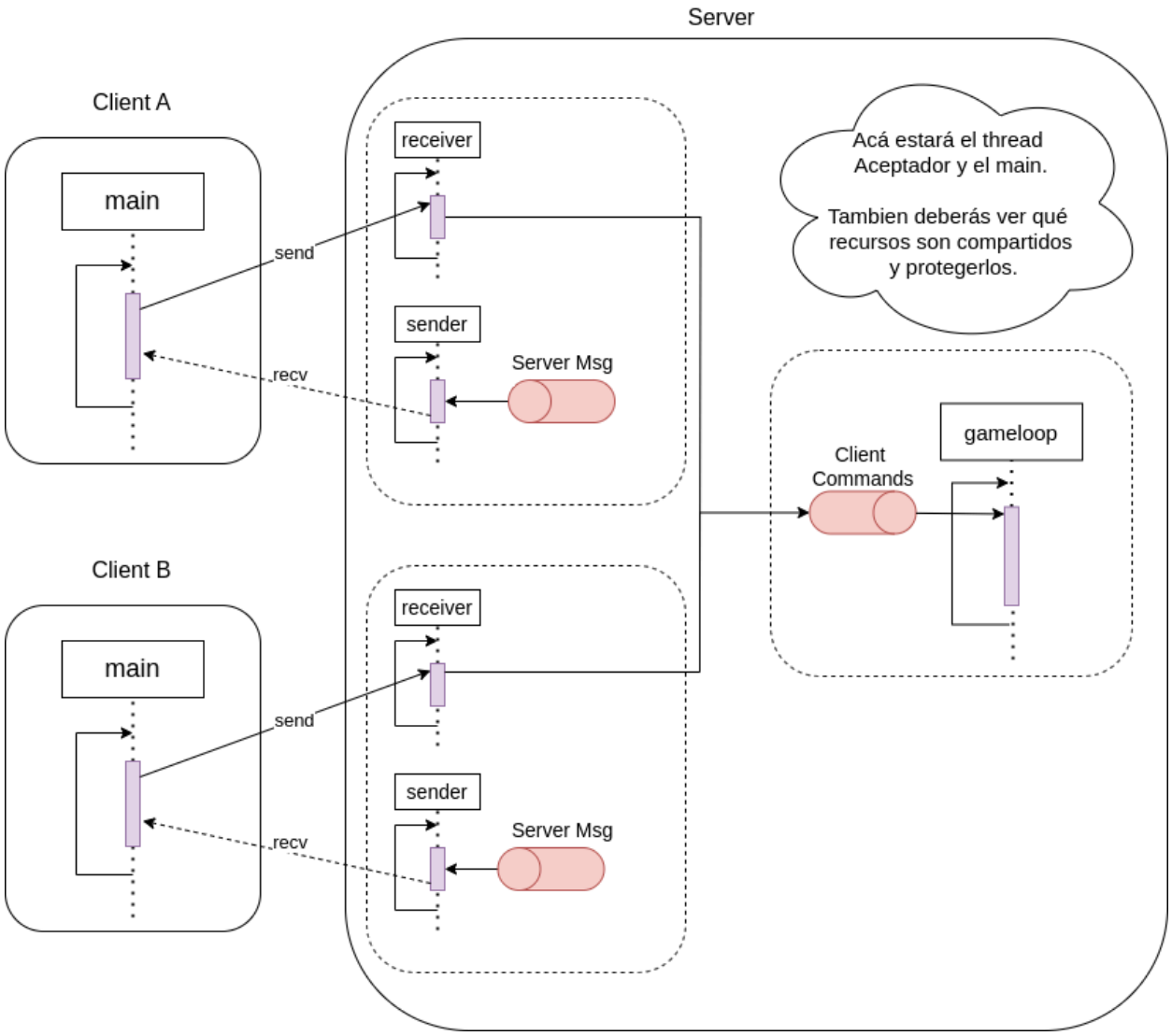
Además, el servidor **deberá** tener **2 threads por cada cliente conectado**: un thread será el encargado de recibir por socket los mensajes del cliente y el otro de enviarlos hacia el cliente.
El thread recibidor agregará comandos **a la única queue del gameloop** y el thread enviador leerá mensajes de su **queue** para enviar al cliente**.**

Estas queues deberán ser thread-safe para prevenir las RC entre el gameloop y los hilos recibidor y enviador respectivamente. Es decisión del alumno qué tipo de queue usar (blocking/nonblocking, bounded/unbounded) y deberá **justificarlo**.

El servidor puede tener threads adicionales como el thread Aceptador y el main.

Estas queues no nos previenen de todas las RC: como también se permite que los clientes se unan y retiren de la partida en distintos tiempos, hay que agregar o remover la queue de cada cliente de una ***“lista de queues”*** (o ***"del mapping de queues"***) y como esta está compartida entre threads deberá protegerse con un **monitor**. Recordá que el gameloop recorrerá esta lista para hacer un **broadcast**.

El servidor **debe** estar leyendo de la entrada estándar a la espera de **leer** la letra `q` que le indica que debe finalizar cerrando todos los sockets, queues y joinenando todos los threads sin enviar ningún mensaje adicional ni imprimir por salida estándar.



Protocolo

El cliente podrá enviar un único mensaje:

- `0x03 <player-name> <box-id>` donde `0x03` es un byte con el número literal `0x03`, `<player-name>` es un entero de dos bytes en big endian con el largo del nombre en ASCII, seguido con el nombre del jugador, y `<box-id>` es un byte con el id de la caja a recoger (`0x01`, `0x02`, `0x03` o `0x04`).

  Nota: En un juego real, tal como el del trabajo final, enviar el nombre o el id del jugador desde el cliente es una práctica **muy insegura.** Un cliente malicioso que conoce el protocolo podría hacerse pasar por otro.

El servidor podrá enviar dos mensajes:

- `0x06 0x04 <player-name> <reward-id>`, donde el `<reward-id>` es un byte con el id de la recompensa. Este método indica que un jugador recogió el premio de una caja.

reward-id	reward-name
0x10	Bazooka
0x11	Chainsaw
0x12	Death ray
0x13	Shotgun

- `0x06 0x05` para indicar que una caja reapareció.

Formato de Línea de Comandos

```
./client <hostname o IP> <servicename o puerto>

./server <servicename o puerto>
```

Códigos de Retorno

Tanto el cliente como el servidor deberán retornar 1 si hay algún problema con los argumentos del programa o 0 en otro caso.

Ejemplo de Ejecución

Lanzamos el servidor:

```
./server 8080
```

Y lanzamos el cliente A:

```
./client 127.0.0.1 8080
```

En este punto, el cliente ya se conectó al servidor.

El cliente A imprime por stdout:

- `What is your name?`

Enviamos `Client A` en el cliente A.

Luego enviamos el comando `Pickup 1` en el cliente A. En la salida estándar del servidor vemos el mensaje:

```
Client A picked up a Bazooka
```

En la salida estándar del cliente A no vemos ese mensaje. Si ahora le damos el comando `Read 1` al cliente A, ahí se imprimirá dicho mensaje:

```
Client A picked up a Bazooka
```

Supongamos que un segundo cliente B se conecta al servidor, ingresa su nombre, `Client B,` y le damos el comando `Read 1`. Tanto en el servidor como en el cliente B imprimirán el siguiente mensaje una vez que pasen 3 segundos luego de que el cliente A haya recogido su Bazooka.

```
A new box has appeared
```

Si ahora le damos `Pickup 3` tanto desde el cliente A como desde el cliente B, ambos *competirán* por la recompensa de la caja 2. Si se procesa antes el comando del cliente B, el servidor imprimirá:

```
Client B picked up a Death ray
```

Y luego de 4 segundos aproximadamente,

```
A new box has appeared
```

Ahora si le damos `Read 3` desde el cliente A, se imprimirá:

```
A new box has appeared
Client B picked up a Death ray
A new box has appeared
```

Si le damos el comando `Exit` tanto al cliente A como al B ambos terminan.
Si escribimos en la entrada estándar del servidor la letra `q,` este finaliza.

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**.
9. **Usa las tools!** Corre *cppcheck* y *valgrind* a menudo para cazar los errores rápido y usa algun **debugger** para resolverlos (GDB u otro, el que más te guste, lo importante es que **uses** un debugger)

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.