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

# *Jazz Jackrabbit 2*

## *Ejercicio / Prueba de Concepto N°2*
## *Threads*

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

Formato de Línea de Comandos
Códigos de Retorno
Ejemplo de Ejecución
Recomendaciones
Restricciones


# 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 5 enemigos. Los jugadores podrán unirse a la partida y matarlos. Cada enemigo, a su vez, revivirá siempre 3 segundos después de ser asesinado.

Cada vez que un enemigo revive o es asesinado, el servidor deberá enviarle un mensaje a **todos** los clientes conectados indicando el evento sucedido y el estado global de todos los enemigos.

Este mensaje será impreso **tanto por el servidor como por los clientes** y es, según el caso:

- Un enemigo ha muerto. <V> vivo(s) y <M> muerto(s).
- Un enemigo ha revivido. <V> vivo(s) y <M> muerto(s).

Para matar a un enemigo, los jugadores tendrán que enviar un mensaje al servidor indicando su intención. Siempre que haya algún enemigo vivo, este mensaje hará que uno muera.

## Acciones del cliente

Cada cliente deberá leer de entrada estándar que acciones va a realizar. Estas son:

- `Atacar`: el cliente deberá enviar un mensaje al servidor indicando su intención de atacar un enemigo.
- `Leer <n>`: el cliente espera a recibir `<n>` mensajes del servidor, imprimiendolos a medida que llegan.
- `Salir`: 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 `Leer <n>` que espera por exactamente `<n>` mensajes desde el servidor. Por cada uno de estos `<n>` mensajes imprimirá el evento (enemigo vivo / muerto) y el estado global de los enemigos, 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 los **comandos** de los clientes y **ejecutarlos** (matar a los enemigos).
   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*.
   Cada 15 iteraciones del gameloop, el enemigo revivirá.

3. Enviar los mensajes a los clientes. Cada vez que un enemigo es asesinado o revivido, 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

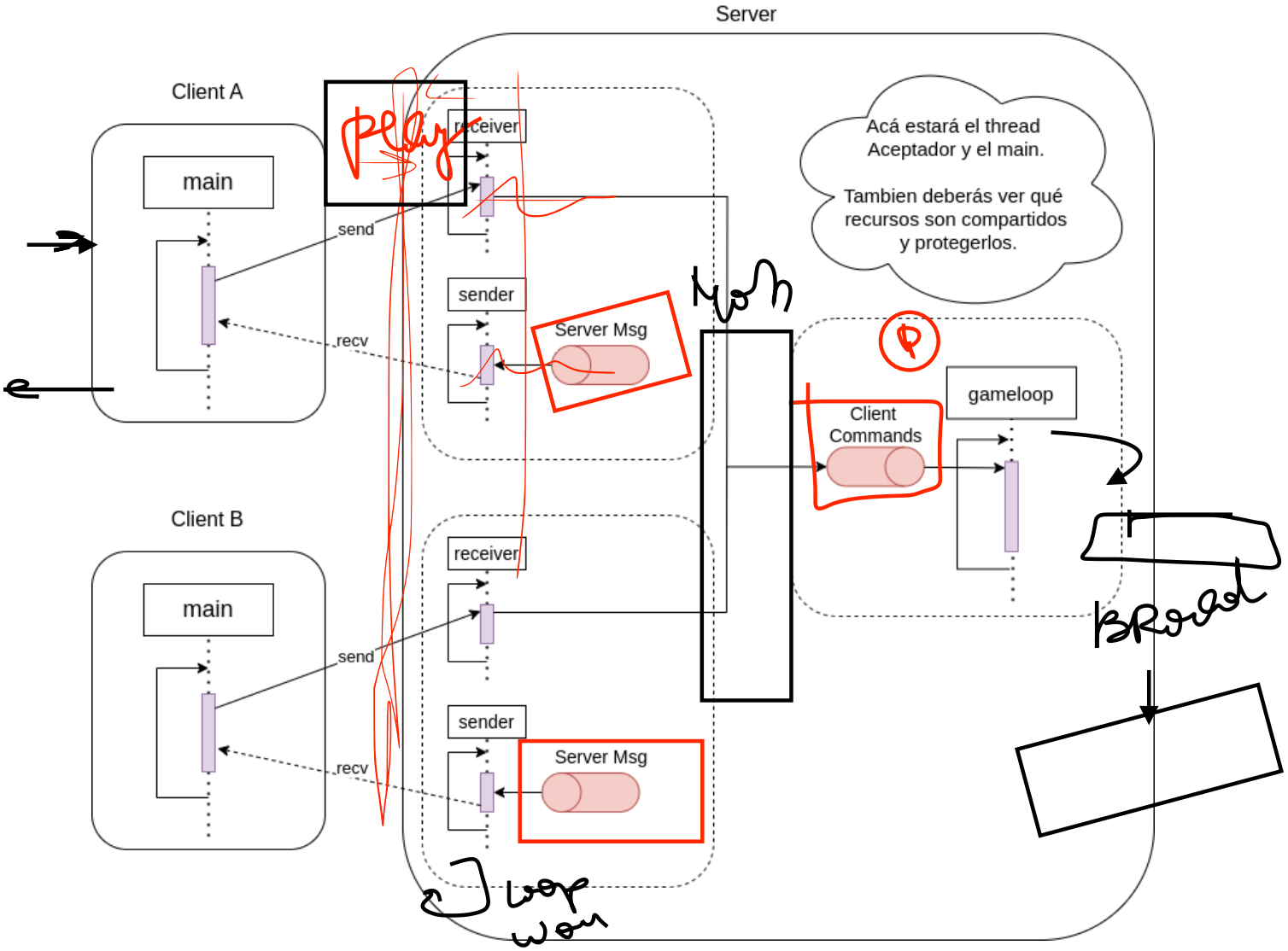
de su **queue** para enviar al cliente.

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` donde `0x03` es un byte con el número literal `0x03`. Este representará el ataque del cliente.

El servidor podrá enviar también un único mensaje:

- `0x06 <enemies alive cnt> <enemies dead cnt> <type event>` donde `0x06` es un byte con el número literal `0x06`, `<enemies alive cnt>` son 2 bytes sin signo en big endian con la cantidad de enemigos vivos, `<enemies dead cnt>` son 2 bytes sin signo en big endian con la cantidad de enemigos muertos y `<type event>` es un campo de un byte indicando el evento sucedido. Este puede valer:
  - `0x04` si un enemigo ha muerto.
  - `0x05` si un enemigo ha revivido.

# 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
```

Enviamos el comando `Atacar` en el cliente A. En la salida estándar del servidor vemos el mensaje:

```
Un enemigo ha muerto. 4 vivo(s) y 1 muerto(s).
```

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

```
Un enemigo ha muerto. 4 vivo(s) y 1 muerto(s).
```

Supongamos un segundo cliente B se conecta al servidor y le damos el comando `Leer 1`. Tanto en el servidor como en el cliente B imprimirán el siguiente mensaje una vez que pasen 3 segundos luego del anterior ataque del cliente A.

```
Un enemigo ha revivido. 5 vivo(s) y 0 muerto(s).
```

Si ahora le damos `Atacar` tanto desde el cliente A como desde el cliente B, el servidor imprimirá:

```
Un enemigo ha muerto. 4 vivo(s) y 1 muerto(s).
Un enemigo ha muerto. 3 vivo(s) y 2 muerto(s).
```

Y luego de 3 segundos aproximadamente,

```
Un enemigo ha revivido. 4 vivo(s) y 1 muerto(s).
Un enemigo ha revivido. 5 vivo(s) y 0 muerto(s).
```

Ahora si le damos `Leer 4` desde el cliente A, se imprimirá:

```
Un enemigo ha revivido. 5 vivo(s) y 0 muerto(s).
Un enemigo ha muerto. 4 vivo(s) y 1 muerto(s).
Un enemigo ha muerto. 3 vivo(s) y 2 muerto(s).
Un enemigo ha revivido. 4 vivo(s) y 1 muerto(s).
```

Si le damos el comando `Salir` 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.