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

Worms

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 inicializara una única partida que estará en espera para que los jugadores se unan a ella. Esto es lo que se conoce como el ***lobby*** de la partida.

Los jugadores unidos a la partida pueden enviar y recibir mensajes a modo de un **chat** simplificado mientras esperan en el lobby.

Cada vez que un jugador se una o se retire del lobby, el servidor deberá enviarle un mensaje a **todos** los clientes conectados con la cantidad total de jugadores en el lobby.

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

- `Jugadores <N>, esperando al resto de tus amigos...`

Donde `<N>` es la cantidad de jugadores unidos esperando en el lobby.

Además, cada jugador podrá enviar mensajes de chat: el servidor deberá obtenerlos y enviarlos a **todos** los clientes conectados, sean jugadores o espectadores. Es un *broadcast*. Tanto el servidor como los clientes

deberán imprimirlos.

En caso de que un cliente se desconecte, el servidor debe actualizar el contador de jugadores y notificar el cambio a todos los jugadores conectados.

Requerimientos del servidor

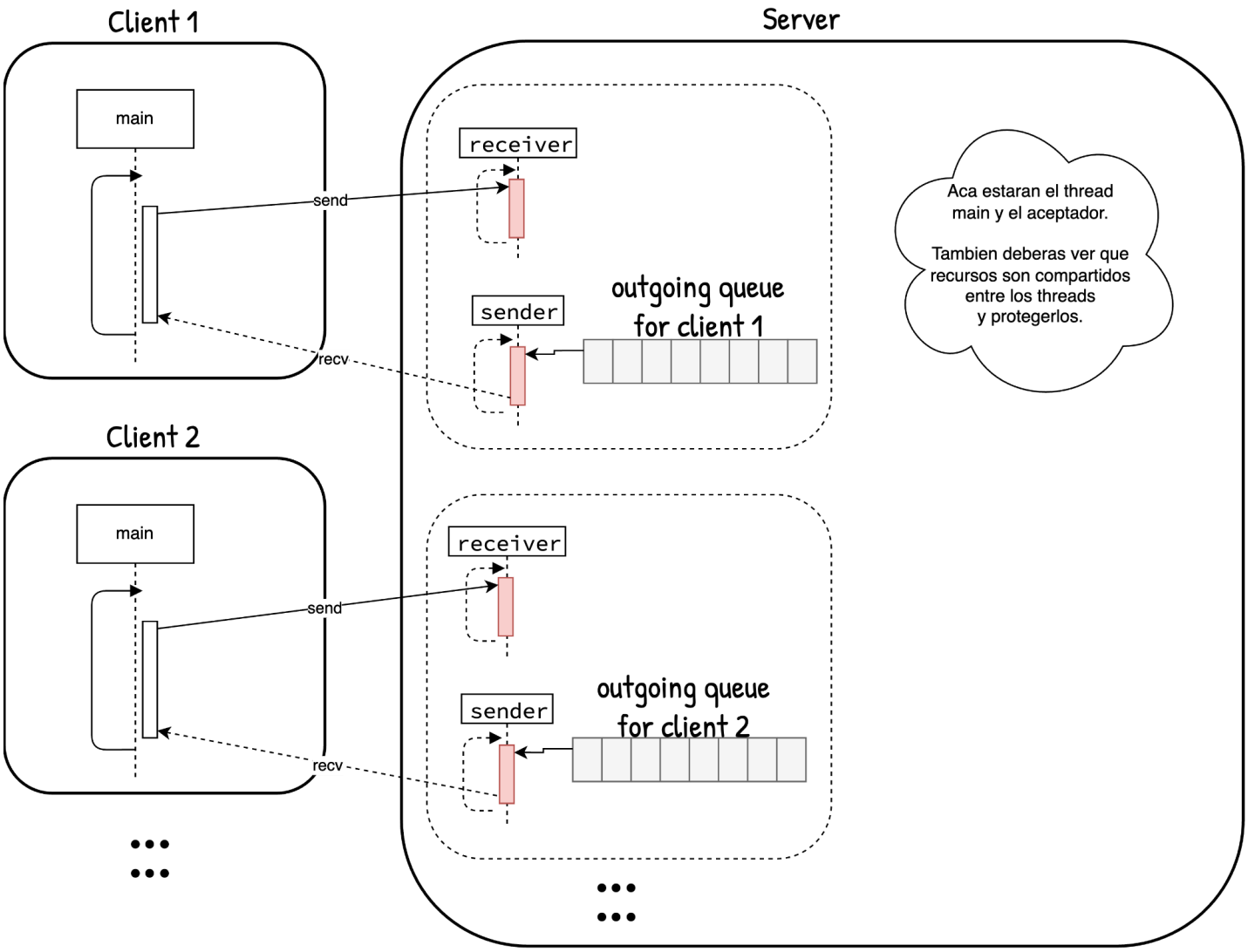
El servidor **deberá** tener **2 threads por cada cliente conectado**: un thread será el encargado de recibir por socket los mensajes del cliente y el otro de enviarlos hacia el cliente (el servidor puede tener threads adicionales como el thread Aceptador/Lobby y el main).

Dado que los mensajes de chat de un jugador deben ser enviados al resto de los clientes y que el servidor debe también poder enviar mensajes a todos los clientes, esto impone una race condition sobre los sockets (múltiples threads quieren enviar usando el mismo socket). Por ello el servidor **debe** tener **una thread-safe queue por cada thread del cliente que se dedique a enviar** mensajes hacia el cliente (lo que serían las **outgoing** queues).

El uso de estas thread-safe queues previene las RCs sobre los sockets. Es decisión del alumno qué tipo de queue usar (blocking/nonblocking, bounded/unbounded) y deberá **justificarlo**.

Las RCs sobre los sockets no son las únicas: como también se permite que los clientes se unan y retiren del lobby en distintos tiempos, hay que agregar o remover la queue de cada cliente de ***"la lista de queues"*** (o ***"del mapping de queues"***) del lobby y como esta está compartida entre threads deberá protegerse con un **monitor** (recordá que habrá múltiples threads recorriendo esa lista para hacer un **broadcast**). Notar que según el diseño que elijas puedes tener más o menos monitores.

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.



Acciones del cliente

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

- `Chat <chatmsg>`: el cliente envía el mensaje de chat. `<chatmsg>` es un texto arbitrario que incluye todo hasta el fin de la línea. El texto **puede** tener espacios en el medio por lo que tal vez quieras ayudarte combinando el operador `>>` con `getline`.
  Nota: entre `Chat` y `<chatmsg>` habrá 1 o más espacios en blanco que deben ser *ignorados*, el mensaje a enviar comienza desde el primer caracter no-blanco hasta el fin de la línea *sin incluirla*.
- `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 de vuelta desde el servidor. Estos pueden ser:

- `Jugadores <N>, esperando al resto de tus amigos...`
- `<chatmsg>`

Protocolo

El cliente podrá enviar un único mensaje:

- `0x05 <chatmsg length> <chatmsg>` donde `0x05` es un byte con el número literal `0x05`, `<chatmsg length>` son 2 bytes sin signo en big endian con la longitud del mensaje y `<chatmsg>` es el mensaje sin el salto de línea sin el `` `\0` `` al final.

El servidor podrá enviar:

- `0x06 <player cnt>` donde `0x06` es un byte con el número literal `0x06` y `<player cnt>` es un campo de 1 byte sin signo con la cantidad de jugadores actualmente unidos a la partida y esperando en el lobby.

- `0x09 <chatmsg length> <chatmsg>` es similar al mensaje que envían los clientes: es el mensaje de chat que un cliente envió al servidor y este lo está reenviando a todos los jugadores.

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 la salida estándar del servidor vemos el mensaje:

```
Jugadores 1, esperando al resto de tus amigos...
```

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

```
Jugadores 1, esperando al resto de tus amigos...
```

Supongamos un segundo cliente B que se conecta al servidor y al que le damos el comando `Chat foo`.

Tanto el cliente A como el B no muestran ningún mensaje pero el servidor muestra:

```
Jugadores 2, esperando al resto de tus amigos...
foo
```

Si al cliente A le damos el comando `Read 1` muestra:

```
Jugadores 2, esperando al resto de tus amigos...
```

Si al cliente B le damos el comando `Read 2` muestra:

```
Jugadores 2, esperando al resto de tus amigos...
foo
```

Si le damos el comando `Exit` tanto al cliente A como al B ambos terminan. El servidor muestra:

```
Jugadores 1, esperando al resto de tus amigos...
Jugadores 0, esperando al resto de tus amigos...
```

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.