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

# *Counter Strike 2D*

## *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* y cumplimiento de la **totalidad** del enunciado 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), interbloqueo en el acceso a recursos (deadlocks y livelocks) y threads daemon (detach).<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).*

# Indice

# Introducción

Este ejercicio consistirá en programar el clásico juego Ta-Te-Ti, **pero** de manera distribuida: Habrá un servidor con la lógica del juego, y los clientes podrán crear partidas o unirse a partidas existentes para desafiar a otros jugadores. Se deberán soportar múltiples partidas al mismo tiempo.

# Descripción

## Reglas del Juego

El TaTeTi se juega entre dos jugadores, por turnos, y en un tablero de 3 filas y 3 columnas. Cada jugador es dueño de un símbolo: una X o una O, y en cada turno colocará su símbolo en alguna casilla desocupada del tablero, ocupando la misma. Si alguno de los jugadores logra colocar tres símbolos propios alineados, gana. Si todas las casillas son ocupadas y ninguno de los jugadores logró alinear tres símbolos, se declara un empate.

*Ayuda: Empezar por una clase que represente una partida de Ta-Te-Ti con una buena interfaz y testearla.*

## Partidas Online

El servidor del juego deberá soportar múltiples partidas al mismo tiempo, que los jugadores podrán crear durante la ejecución del programa. Cuando un jugador crea una partida, es automáticamente sumado a la misma y puede empezar a jugar.
Otra manera de empezar a jugar una partida es uniéndose a una partida existente. Siempre jugará primero quien haya creado la partida.

*Ayuda: Tener un hilo "aceptador" en el servidor, más un hilo "manejador" para cada cliente. Notar que el lugar donde se guardan las partidas es compartido entre todos los "manejadores", y necesitará un Monitor acorde. También habrá partidas que van a ser compartidas entre los dos jugadores. Para evitar contención habrá que usar un Monitor acorde.*

## Interfaz de Usuario

Los jugadores verán el tablero **representado en texto**, con el siguiente formato:

```
    1 . 2 . 3 .
  +---+---+---+
1 | X |   |   |
  +---+---+---+
2 |   | O |   |
  +---+---+---+
3 |   |   |   |
  +---+---+---+
```

En este tablero el jugador 1, que creó la partida y siempre usa las letras O, jugó en la casilla central, y el jugador 2, que se unió a la partida y siempre usa las letras X, jugó en la casilla superior izquierda.

*Ayuda: No copy-pastear del PDF, a veces incluye caracteres invisibles indeseables. Sobrecargar el operator<< de ostream con el modificador friend y revisar que se imprima correctamente el tablero. Puede que tengas algún salto de línea de diferencia con las pruebas subidas al sistema de corrección, tratá de hacer alguna entrega con tiempo para poder adaptar el formato.*

### Dinámica de las Partidas

Cada vez que sea el **turno** de un jugador, el servidor le enviará un mensaje con la representación del tablero en modo **texto**. El jugador enviará su siguiente jugada, y el servidor le contestará cuando vuelva a ser su turno (con la representación en texto del tablero).

*Ayuda: Notar que el "manejador" de cada cliente deberá **esperar a que sea el turno** del jugador que está atendiendo para entonces enviarle la representación del tablero. ¿Qué mecanismo de sincronización tenemos para esperar a que algo suceda? ¿Dónde tiene que estar la lógica que usa ese mecanismo?*

Una vez que la partida haya finalizado, el servidor le contestará a ambos participantes con alguno de los siguientes mensajes:

- "Felicitaciones! Ganaste!\n", si el jugador ha ganado la partida.
- "Has perdido. Segui intentando!\n", si el jugador ha perdido la partida.
- "La partida ha terminado en empate\n" si el tablero se ha llenado y no hay un ganador.

Cuando el cliente detecta que ha recibido alguno de esos tres mensajes, deberá cerrarse ordenadamente.

## Formato de Línea de Comandos

Como en un ejercicio anterior, en este tendremos dos procesos, "client" y "server". El servidor se ejecutará de la siguiente manera:

```
./server <port>
```

Y el cliente de esta otra manera:

```
./client <host> <port>
```

Ninguno de los dos procesos usará archivos adicionales para entradas o salidas.

## Códigos de Retorno

Tanto cliente como servidor retornarán un 0 en caso de que la ejecución haya sido exitosa, o un 1 si hubo un error en los parámetros.

Se permite al alumno decidir sobre el uso de otros códigos de retorno para otros tipos de error que detecte (inputs inválidos, por ejemplo). Sin embargo, se asegura que los casos de prueba no forzarán ese tipo de errores, sino que usarán "caminos felices".

# Entrada y Salida Estándar

### Server

El servidor escuchará conexiones entrantes en el puerto especificado por línea de comandos hasta **leer un caracter 'q' de entrada estándar**. Luego de leído dicho caracter, se deberá cerrar el socket "aceptador" para desbloquear el *accept()*, y esperar a que se termine de atender a todos los clientes que ya fueron aceptados.

*Ayuda: Leer la página de manual de accept para entender qué pasa cuando alguien cierra el socket mientras estamos bloqueados en accept(), y manejar el caso de manera conveniente.*

El servidor no utilizará la **salida estándar** ni la **salida estándar de error**.

### Client

El cliente recibirá los **comandos** por **entrada estándar**, y mostrará las respuestas del servidor por **salida estándar**.

El cliente no utilizará la **salida estándar de error**.

# Comandos del Cliente

El cliente leerá **comandos** de la entrada estándar, uno por cada línea ingresada. Estos comandos podrán ser:

- **crear <nombre-de-partida>**: Que servirá para enviar una orden al servidor de crear una partida con el nombre especificado.
- **listar**: Para saber qué partidas hay disponibles para unirse.
- **unirse <nombre-de-partida>**: Se utilizará para enviar un pedido de unirse a la partida especificada.
- **jugar <columna> <fila>**: Servirá para hacer una jugada en la posición indicada. Tanto la columna como la fila tendrán valores 1, 2, o 3.

# Protocolo de Comunicación

El cliente enviará de la siguiente manera el mensaje **crear:**

1. Un byte fijo 0x6E
2. Dos bytes con el largo del nombre de la partida, en Big Endian
3. El nombre de la partida

Por ejemplo, un mensaje para crear una partida con el nombre "Partida", se enviaría esta tira de bytes (la representamos en hexa)

```
6E 00 07 50 61 72 74 69 64 61
```

El cliente enviará de la siguiente manera el mensaje **unirse**:

1. Un byte fijo 0x6A
2. Dos bytes con el largo del nombre de la partida, en Big Endian
3. El nombre de la partida

Por ejemplo, un mensaje para unirse a una partida con el nombre “123”, se enviaría esta tira de bytes (la representamos en hexa)

```
6A 00 03 31 32 33
```

El cliente enviará solamente el byte **0x6C** para **listar** las partidas.

El cliente usará la siguiente estructura para el mensaje **jugar**:

1. Un byte fijo 0x70
2. 4 bits con la columna
3. 4 bits con la fila

Por ejemplo, para hacer una jugada en la posición central-inferior, se enviará esta tira de bytes:

```
70 12
```

*Ayuda: Usar el operador OR lógico para “armar” el segundo byte. Notar que el comando por consola y la representación como string del tablero tienen columnas y filas con índices empezando desde 1, pero en este protocolo los índices empiezan desde cero.*

**El servidor enviará todos los mensajes al cliente pre-concatenando el largo, en 2 bytes, y en big endian. Todos los textos que envía el servidor terminan en un salto de línea (‘\n’).**
El servidor contestará al mensaje **crear** con una representación vacía del tablero:

```
    1 . 2 . 3 .
  +---+---+---+
1 |   |   |   |
  +---+---+---+
2 |   |   |   |
  +---+---+---+
3 |   |   |   |
  +---+---+---+
```

El servidor contestará al mensaje **unirse** con el estado del tablero, **cuando sea el turno del jugador** (por ende, cuando el jugador creador ya ha realizado su primera jugada):

```
    1 . 2 . 3 .
  +---+---+---+
1 | O |   |   |
  +---+---+---+
2 |   |   |   |
  +---+---+---+
3 |   |   |   |
  +---+---+---+
```

El servidor contestará al mensaje **listar** con la lista de las partidas, ordenadas “casi alfabéticamente”:

```
Partidas:
 - Partida1
 - Partida2
 - partida1
 - partida2
```

*Ayuda: el orden no es puramente alfabético, sino que es el orden respecto al operator<() de std::string. La STL provee un diccionario ordenado. Notar que “Partida2” empieza con “P” mayúscula, que está antes en la tabla ascii que la “p” minúscula, y por eso “Partida2” está antes que “partida1”.*

El servidor contestará al mensaje **jugar** con una representación del tablero **cuando sea el turno del jugador**, y concatenará el mensaje de fin de partida si corresponde:

```
    1 . 2 . 3 .
  +---+---+---+
1 | O | X |   |
  +---+---+---+
2 | O | X |   |
  +---+---+---+
3 | O |   |   |
  +---+---+---+
Felicitaciones! Ganaste!
```

*Notar que este último es UN SOLO mensaje, y por ende el largo que se deberá enviar por el socket será la suma del largo del string que representa el tablero, y el largo del mensaje de finalización de la partida.*

# Ejemplo de Ejecución

Vamos a ver un ejemplo completo de cómo debería comportarse nuestro sistema.

Primero, vamos a ejecutar el servidor en el puerto 7777:

```
./server 7777
```

Luego, vamos a ejecutar un cliente 1, de manera que se conecte a ese servidor:

```
./client localhost 7777
```

Y desde otra consola vamos a ejecutar otro cliente 2, de manera que se conecte al mismo servidor:

```
./client localhost 7777
```

El usuario del cliente 1 va a crear una partida, ingresando el siguiente comando:

```
crear Partida
```

Entonces, el proceso del cliente 1 enviará por su socket la siguiente tira de bytes:

```
6E 00 07 50 61 72 74 69 64 61
```

Como la partida no existe, la misma será creada en el servidor, que le responderá al cliente 1 un mensaje con su tablero vacío:

```
    1 . 2 . 3 .
  +---+---+---+
1 |   |   |   |
  +---+---+---+
2 |   |   |   |
  +---+---+---+
3 |   |   |   |
  +---+---+---+
```

El cliente 1 mostrará este mensaje por salida estándar.

Desde su consola, el usuario del cliente 2 envía un comando para ver qué partidas tiene disponibles:

```
listar
```

Entonces el cliente le envía el siguiente mensaje al servidor:

```
6C
```

El servidor recibe su mensaje, y le devuelve una lista de las partidas creadas:

```
Partidas:
 - Partida
```

El cliente 2 mostrará este mensaje por su salida estándar, y entonces el usuario decidirá unirse a la única partida disponible, ingresando el nombre de esa partida:

```
unirse Partida
```

Luego, el cliente 2 enviará el siguiente mensaje por su socket:

```
6A 00 07 50 61 72 74 69 64 61
```

El servidor lo unirá a la partida, pero todavía no le contestará ya que no es su turno.
En ese momento, el usuario del cliente 1 decide jugar en la posición central, e ingresa:

```
jugar 2 2
```

Por lo tanto, el cliente 1 enviará la siguiente tira de bytes por su socket:

```
70 11
```

En ese momento, el servidor tomará la jugada del cliente 1, pero no le contestará ya que no es su turno. A quien sí le contestará será al cliente 2, con el estado del tablero:

```
    1 . 2 . 3 .
  +---+---+---+
1 |   |   |   |
  +---+---+---+
2 |   | O |   |
  +---+---+---+
3 |   |   |   |
  +---+---+---+
```

Entonces el usuario del cliente 2 decidirá ocupar la casilla superior izquierda:

```
jugar 1 1
```

Enviará la jugada por su socket:

```
70 00
```

En ese momento, el servidor le contestará al cliente 1:

```
    1 . 2 . 3 .
  +---+---+---+
1 | X |   |   |
  +---+---+---+
2 |   | O |   |
  +---+---+---+
3 |   |   |   |
  +---+---+---+
```

Y así sucesivamente, hasta que algún jugador gane, o se llene el tablero, digamos que se realizan las siguientes jugadas (omitiendo respuestas):

```
jugar 1 3
jugar 2 1
```

En ese momento, el cliente 1 recibirá su chance de ganar:

```
    1 . 2 . 3 .
  +---+---+---+
1 | X | X |   |
  +---+---+---+
2 |   | O |   |
  +---+---+---+
3 | O |   |   |
  +---+---+---+
```

Y entonces jugará en (3,1), le enviará el mensaje al servidor, y el servidor le contestará a ambos jugadores. Al cliente 1 le dirá que ganó:

```
    1 . 2 . 3 .
  +---+---+---+
1 | X | X | O |
```

```
  +---+---+---+
2 |   | O |   |
  +---+---+---+
3 | O |   |   |
  +---+---+---+
Felicitaciones! Ganaste!
```

Y al cliente 2 le dirá que perdió:

```
    1 . 2 . 3 .
  +---+---+---+
1 | X | X | O |
  +---+---+---+
2 |   | O |   |
  +---+---+---+
3 | O |   |   |
  +---+---+---+
Has perdido. Segui intentando!
```

# 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.
4. Seguir las *ayudas*.