Trabajo Practico 2: `pthreads`
"BattleSystem"

Sistemas Operativos - Segundo Cuatrimestre de 2013

Límite de entrega: Lunes 17 de Junio a las 23:59

1. Introducción

Con el furor de los juegos simples y la aparición de ARPANet, los docentes de esta materia decidieron intentar combatir la situación económica actual con su granito de arena.

Se diseñó un nuevo juego multijugador basado en la clásica batalla naval. La idea principal se basa en el clásico *todos contra todos*. Se eliminó el sistema de turnos y se estableció un tiempo de penalización aleatorio como "el tiempo que tarda un misil en llegar al barco enemigo". Cuando el juego comienza, los jugadores eligen un enemigo y apuntan un misil a alguna de las casillas que no han sido bombardeadas previamente. Al disparar, el servidor establece en forma estimada, el tiempo que tarda el misil en llegar a destino. Durante ese tiempo, el jugador no puede realizar otros disparos. El resto del mecanismo del juego es similar al de la batalla naval.

El último mes un consultor fue reclutado para analizar la difícil situación financiera de la empresa y los problemas que su implementación acarreaba. Tras recibir una escandalosa suma en concepto de honorarios, dicho analista le hizo saber a los docentes que, "Dicho mal y pronto, su negocio está en las últimas. El juego de PC por internet *sha fue* y lo que se viene ahora es la *ueb tu point ou*[1] y las aplicaciones para *esmar fon*[2]".

Desesperado, el ayudante de segunda que dijo que iba a trabajar para ayudar con la causa, huyó a Perú. El resto de los docentes decidió lanzar al mercado una versión *web-social-online-multijugador-twitter-facebook* del juego.

Rápidamente, los docentes de la cátedra hicieron uso de sus contactos para conseguir inversores suficientemente ingenuos, financiar el proyecto, y contrataron a un grupo de programadores *monoproceso-teístas*. Sin embargo, durante el transcurso del desarrollo se hizo cada vez más notorio que era necesario adoptar las bondades del multiprocesamiento para que el juego pudiera hacerse realidad y soportar la tremenda cantidad de usuarios que lo juegan mientras viajan en colectivo.

Ante tal atentado contra su fe, los *mono-programadores* abandonaron el proyecto, y a los docentes no les quedó otra que recurrir a ustedes para que mejoren su servidor de modo de que permita grandes cantidades de jugadores a la vez. Así fue como ustedes se comprometieron a entregar un prototipo del servidor que permita **múltiples clientes jugando simultáneamente** sobre un mismo modelo.

2. Las reglas del juego

Según la especificación (ampliamente informal) que nos entregaron, el objetivo del juego es hundir la mayor cantidad de barcos enemigos sin que sean hundidos todos los propios. Los barcos

[1] Web 2.0
[2] smartphones

se colocan sobre un tablero antes de comenzar el juego. No pueden colocarse en diagonal y no pueden tocarse entre si. Hay barcos de tamaño 1, 2, 3, 4 y 5. Según la configuración del servidor, el tablero puede cambiar de tamaño (aunque siempre es cuadrado) y la cantidad de barcos puede variar.

Al comenzar un juego, los jugadores comienzan a tirar misiles para intentar hundir los barcos enemigos. Para ello seleccionan un enemigo de la lista, eligen las coordenadas en el tablero y disparan. Transcurrido el tiempo que el servidor estima que tarda en llegar el misil a destino, el jugador observa el resultado y puede volver a tirar. Si un jugador tiene todos sus barcos hundidos, no participa más del juego y debe esperar a un nuevo juego.

El juego termina cuando hay un sólo jugador con barcos flotando (o ninguno) y gana el que más puntaje obtiene basado en una regla secreta implementada en el modelo.

3. La implementación

3.1. Arquitectura del sistema

La aplicación esta programada en HTML + Javascript y utiliza *websockets* para comunicarse con el servidor. Esto permite que sea versátil y pueda utilizarse en todo tipo de plataformas[3].

Por tratarse de una aplicación que funciona por internet, el sistema necesita atender las peticiones que realizán los *browsers* de los usuarios. El responsable de atender a estos navegadores es el servidor de *backend* del sistema. Esta comunicación utiliza un dialecto particular sobre websockets que se traducen a sockets TCP utilizando la aplicación *websockify*. Este dialecto afortunadamente ya fue programado por los desarrolladores *monoproceso-teístas*. Un único servidor de *backend* puede atender a varios *browsers* que deseen conectarse para jugar.

3.2. Comunicación cliente/servidor

Al iniciar el servidor de *backend*, se especifica el puerto en el que el servidor espera que se conecten los jugadores, la cantidad de jugadores que se espera, la cantidad de barcos que debe ubicar cada jugador, el tamaño del tablero y la suma de los tamaños de los barcos para cada jugador.

3.3. Protocolo de la comunicación entre *browsers* y *backend*

El protocolo de comunicación entre ambas partes fue definido por los docentes y son mensajes enviados como strings en formato JSON delimitados por el carácter |. Todos los mensajes son de la forma

```
{ "Name": "Comando","Data": "Datos" }
```

Donde `Comando` puede ser uno de los siguientes:

- `Subscribe`
- `Setup`
- `Ok`
- `Error`
- `Start`
- `Player_Info`

[3]posta que anda en android: `http://caniuse.com/websockets`

- `Accept`
- `Decline`
- `Event`
- `Finish`
- `Scores`
- `Get_Info`
- `Shoot`
- `Get_Update`
- `Unsubscribe`
- `Nop`
- `Get_Scores`

3.3.1. Comunicación asincrónica

En varias ocasiones el *backend* puede enviar *eventos* a los jugadores. Estos eventos indican que se han producido cambios en el estado del juego y es el mecanismo por el cual son notificados los participantes. Cuando se envía una respuesta del *backend* a un jugador (*browser*), el servidor verifica si hay eventos pendientes y los envía a continuación de la respuesta. Para que la comunicación sea fluida, los *browsers* envían periódicamente un comando `Nop` que no requiere respuesta. Si hay eventos pendiente, esto desencadenará el envío de dichos eventos a los respectivos jugadores.

3.3.2. Configuración

Cuando un nuevo usuario desea comenzar a jugar, se produce la siguiente comunicación:

1. El *browser* establece la conexión TCP con el *backend*.
2. El *browser* registra al jugador enviando `Subscribe` y como dato, el nombre del jugador. Ej: `{"Name":"Subscribe","Data":{"Name":"Diego Armando"}}`
3. El *backend* le responde `Subscribe` indicando como datos el tamaño del tablero, la suma de los tamaños de los botes y el Id del jugador.

   Ej: `{"Name":"Subscribe","Data":{"Boardsize":12,"Boatssize":19,"Id":0}}`
4. A continuación, el cliente está listo para empezar a ubicar los botes y se pasa a la fase de *Pre-Batalla*.

3.3.3. Pre-Batalla

Durante esta fase del juego, el usuario ubica sus botes en el tablero y una vez aceptados por el servidor, espera que todos los jugadores estén listos para pasar a la fase de *Batalla*.

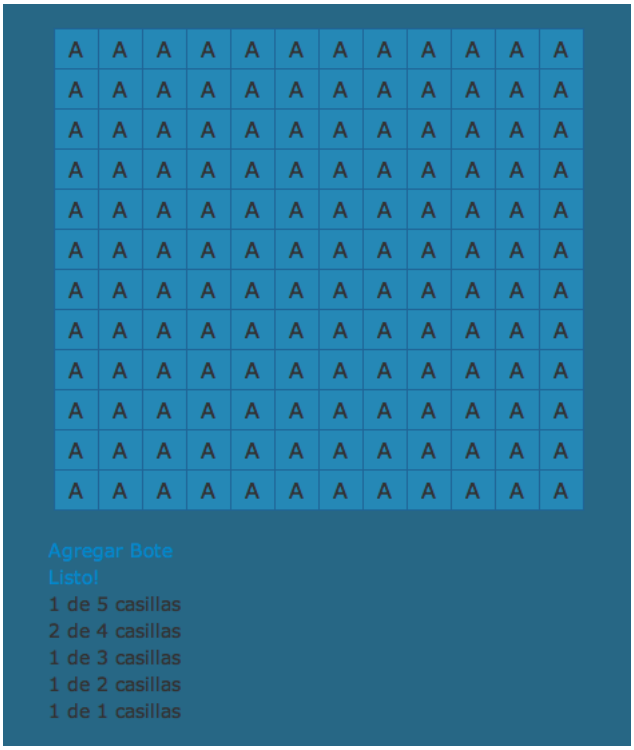

Figura 1: Pantalla Pre-Batalla sin barcos ubicados

Mediante *clicks* con el mouse en las casillas, el usuario puede seleccionar las casillas que compondrán un bote. Estas se irán pintando de color verde y se identificará con un número que corresponde al número de barco. Una vez seleccionadas todas las casillas deseadas para el barco, debe presionar *Agregar Bote*. Este cambiará a color naranja y la lista de barcos pendientes de ubicar se actualizará. Repitiendo este proceso, el usuario deberá ubicar todos los barcos hasta que la lista indique que no quedan más barcos por ubicar.

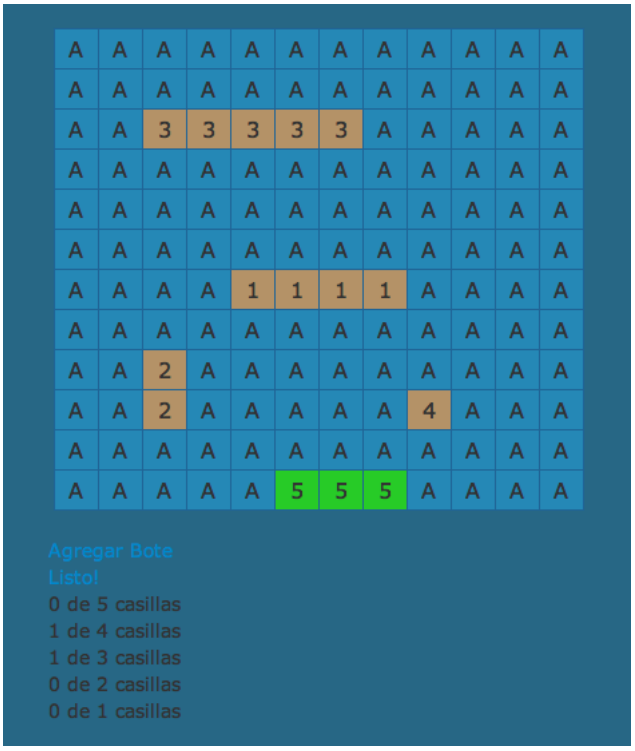

Figura 2: Pantalla Pre-Batalla con algunos barcos ubicados

Si se desea reubicar un barco, basta con realizar *click* en el barco deseado y confirmar la acción.

Una vez ubicados todos los barcos debe presionarse *Listo*. Si el servidor acepta la configuración (están todos los barcos, no hay dos barcos adyacentes, los barcos no se ubican en diagonal), se procede a esperar que todos los jugadores completen esta etapa y se pasa a la fase de *Batalla*.

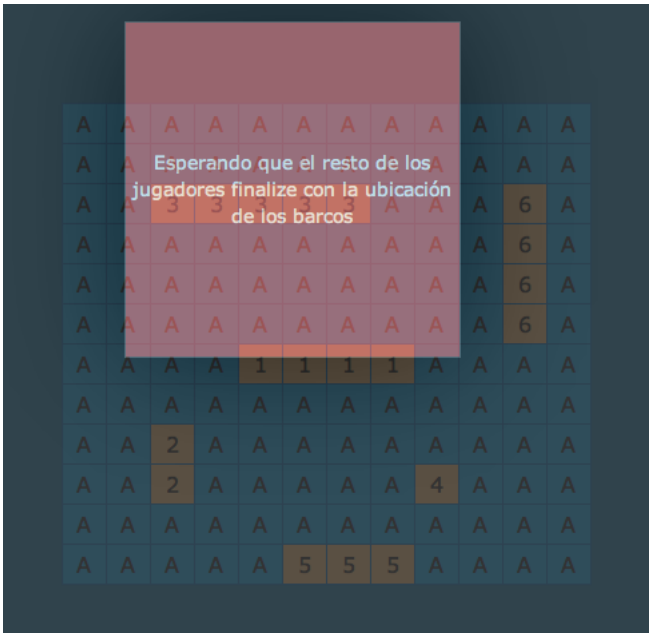

Figura 3: Pantalla Pre-Batalla a la espera del resto de los jugadores

Resumiendo, las comunicaciones que ocurren entre el *browser* y el *backend* son las siguientes:

1. El *browser* envía `Setup` con el Id de jugador y las coordenadas de los barcos como datos.
   Ej: `{"Name":"Setup","Data":{"s_id":1,"boats": [[{"x":6,"y":2 }, {"x":6,"y":3}, {"x":6,"y":4}, {"x":6,"y":5}, {"x":6,"y":6}], [{"x":10,"y":4}], [{"x":9,"y":8}, {"x":10,"y":8}, {"x":11,"y":8}], [{"x":2,"y":8}, {"x":2,"y":9}], [{"x":4,"y":1}, {"x":4,"y":2}, {"x":4,"y":3}, {"x":4,"y":4}], [{"x":7,"y":0}, {"x":8,"y":0}, {"x":9,"y":0}, {"x":10,"y":0}]]}}`

2. El *backend* procesa los datos y responde `Setup` si acepta la ubicación de los barcos.
   Ej: `{"Name":"Setup","Data":"Ok"}`
   Si no acepta la ubicación, devuelve un error.
   Ej: `{"Name": "Error","Data": { "Msg": "Posiciones invalidas"}}`

3. Si el último jugador ha realizado satisfactoriamente la ubicación de los botes, el servidor envía a todos los jugadores el evento `Start` con la lista de jugadores.
   Ej: `{"Name": "Start","Data": [{"Id":0,"Name": "Diego Armando"},{"Id":1,"Name": "Claudio Paul"}]}`

3.3.4. Batalla

Durante esta fase del juego, los participantes comienzan a enviar misiles a los tableros de los enemigos con el objetivo de hundir los barcos. A la izquierda de la pantalla, estos podrán elegir el jugador al que desean tirar. En el centro, se muestra el estado del tablero del jugador seleccionado. A la derecha, el estado actual del tablero del jugador. Debajo del tablero del jugador, la lista de eventos.

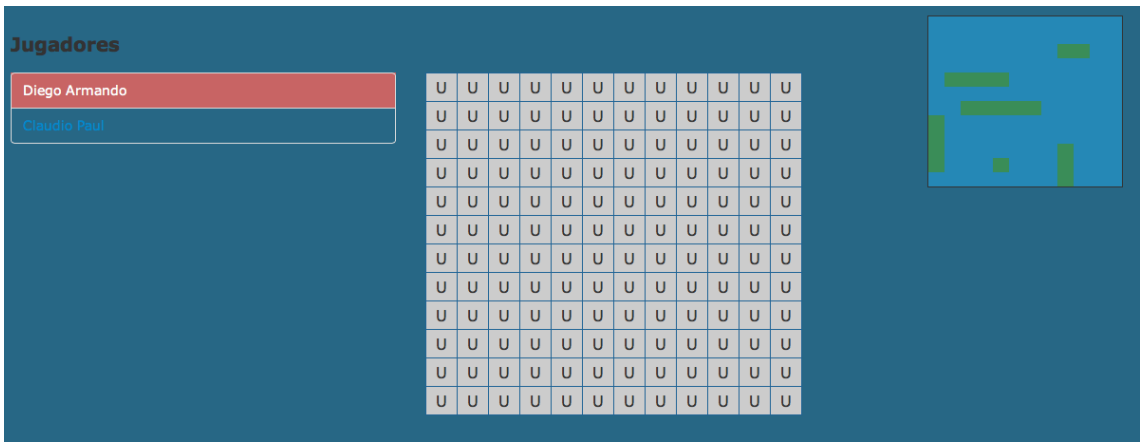

Figura 4: Pantalla al comienzo del juego

Tablero del enemigo:

- En color gris y con la letra U se observan las casillas cuyo estado es *Unknown* o desconocido. Es decir, que no han sido bombardeadas.
- En color rojo y con la letra T se observan las casillas que han sido bombardeadas y cuyo estado es *Tocado*. Si la letra es H, entonces ese bote está en estado *Hundido*.
- En color azul y con la letra A se observan las casillas que han sido bombardeadas y el resultado ha sido *Agua*

Tablero del jugador: Este tablero cumple la única funcionalidad de indicar al jugador cuál es su estado actual con respecto al juego.

- Si la casilla es de color azul, entonces ahí hay agua.
- Si la casilla es de color verde, entonces ahí hay un barco.
- Si la casilla es de color amarillo, un misil ha sido apuntado en esa dirección.
- Si la casilla es de color rojo, un misil ha detonado en esa ubicación, tocando o hundiendo un barco.
- Si la casilla es de color celeste, un misil ha detonado, pero la casilla contenía agua.

Lista de eventos:

- **Outgoing** Disparos realizados por el jugador.
  - Color verde: Un misil ha detonado tocando o hundiendo un bote.
  - Color negro: Un misil ha detonado en el agua.
- **Incoming** Disparos realizados al jugador:
  - Color amarillo: Un misil ha sido apuntado.
  - Color rojo: Un misil ha detonado tocando o hundiendo un bote.
  - Color negro: Un misil ha detonado en el agua.

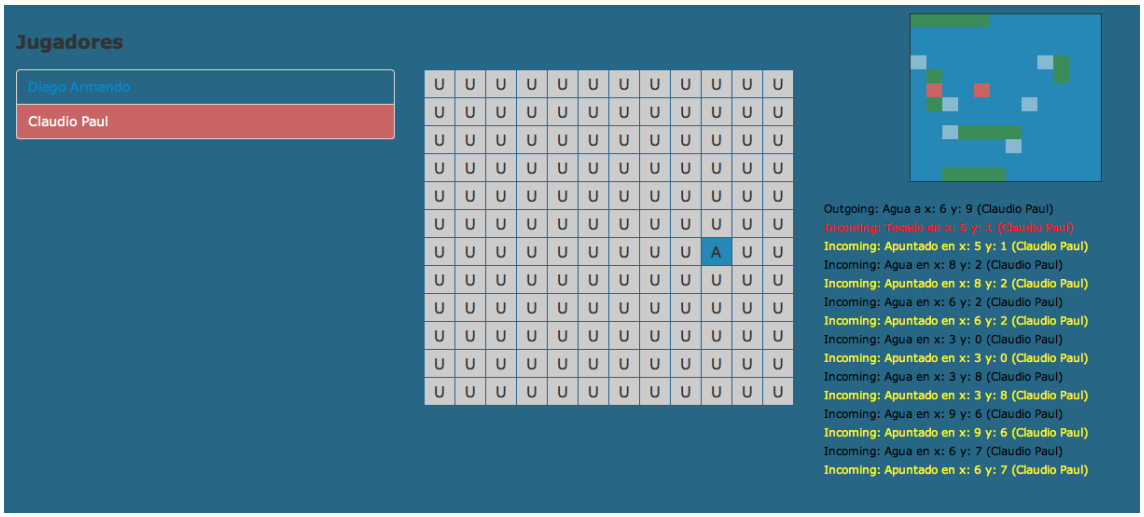

Figura 5: Diego Armando siendo bombardeado por Claudio Paul

Los mensajes enviados para concretar un disparo se describen a continuación:

1. El *browser* envía `Get_Info` con el Id de los dos jugadores (fuente y destino) como datos. Ej: `{"Name":"Get_Info","Data":{"t_id":0,"s_id":1}}`

2. El *backend* envía `Player_Info` con la información del tablero del jugador seleccionado.
Ej: { "Name": "Player_Info","Data": {"t_id": 0,"board":[
["U","U","U","U","U","U","U","U","U","U","U","U"],
["U","U","U","U","U","U","U","U","U","U","U","U"],
["U","U","U","U","U","U","U","U","U","U","U","U"],
["U","U","U","U","U","U","U","U","U","U","U","U"],
["U","U","U","U","U","U","U","U","U","U","U","U"],
["U","U","U","U","U","U","U","U","U","U","U","U"],
["U","U","U","U","U","U","U","U","U","U","U","U"],
["U","U","U","U","U","U","U","U","U","U","U","U"],
["U","U","U","U","U","U","U","U","U","U","U","U"],
["U","U","U","U","U","U","U","U","U","U","U","U"],
["U","U","U","U","U","U","U","U","U","U","U","U"],
["U","U","U","U","U","U","U","U","U","U","U","U"]]}}

3. El jugador selecciona una casilla y el *browser* envía el mensaje `Shoot` con las coordenadas y los datos de los jugadores. El dato *status* indica con la letra S que es un evento de tipo *Shoot* (tiro):
Ej:
{"Name":"Shoot","Data":{"t_id":0,"coord":{"x":"5","y":"4"},"status":"S","s_id":1}}

4. El *backend* puede aceptar o rechazar el tiro basado en el estado del juego en el momento que llega el mensaje. Si este es rechazado, devuelve `Decline` y un mensaje de error como dato.
Ej:{ "Name": "Decline","Data": {"Msg": "Juego no comezado"}}

   Si, en cambio, el tiro es aceptado, devuelve el tiempo estimado de arriba (*ETA*) en milisegundos. Ej: { "Name": "Accept","Data": {"eta": "2207"}}

   Al aceptar el tiro, este envía un evento al jugador destino para indicar que un misil ha sido apuntado a una coordenada de su tablero. El dato *status* indica con la letra I que es un evento de tipo *Incoming*. Ej:
{ "Name": "Event","Data": {"t_id": 0,"s_id": 1,"coords": { "x":5,"y":4},"status":
"I"}

5. El *browser* se bloquea durante *ETA* milisegundos y luego envía `Get_Update` con los datos del jugador actual, para verificar que ocurrió con su tiro.
Ej: {"Name":"Get_Update","Data":{"s_id":1}}

6. El *backend* procesa el tiro del jugador y genera dos eventos iguales y los envía a ambos jugadores para informar el resultado del tiro. El dato *status* indica con la letra H que es un evento de tipo *Hundido*. Si la letra es T el evento es de tipo *Tocado*. La letra A indica *Agua*.
Ej: { "Name": "Event","Data": {"t_id": 0,"s_id": 1,"coords": { "x":5,"y":4},"status":
"H"}}

3.4. Implementación

3.4.1. `elemento.h` y `elemento.cpp`

Abstrae la implementación de un elemento ubicado en una casilla del tablero. Representa tanto a los barcos como al agua. Posee los siguientes atributos:

- `tamanio`: Tamaño del elemento. En el caso de los barcos, representa el tamaño del barco. En el caso de agua, la cantidad de casillas que poseen agua.
- `toques`: Cantidad de toques que se le han dado a este elemento.

- `id_jugador`: Es el ID del jugador que realizó el primer toque a este elemento.
- `id`: Es el ID del elemento en el juego.
- `mismo_jugador`: Una variable *booleana* que es verdadera cuando todos los toques los realizó el mismo jugador.

Las funciones aplicables sobre elementos de esta clase son:

- `tocar(int s_id)`: El jugador pasado por parámetro tira un misil sobre este elemento. Devuelve el resultado. Por defecto, el resultado devuelto es *Agua*.
- `es_mismo_jugador(int s_id)`: Devuelve un valor *booleano* indicando si el jugador pasado por parámetro es el jugador que realizó todos los toques a este elemento.
- `hundido()`: Devuelve `true` si y sólo si el elemento está hundido.
- `dameResultado()`: Devuelve el resultado del último tiro sobre el elemento.
- `dameId()`: Devuelve el Id del elemento.

3.4.2. `embarcacion.h` y `embarcacion.cpp`

Es una subclase de `Elemento` que modifica el comportamiento de los métodos `tocar` y `dameResultado` para ser consistentes con la funcionalidad de los barcos: devuelven *Tocado* en vez de *Agua*.

3.4.3. `casilla.h` y `casilla.cpp`

Modelan una casilla en el tablero. Una casilla está libre y contiene un `elemento`. Primero es apuntada y luego es tocada.

Posee los siguientes atributos:

- `estado`: Es el estado de la casilla. Puede ser *Libre*, *Apuntado* o *Tocado*.
- `id_jugador`: Luego de ser apuntada o tocada, representa el ID del jugador que lo hizo.
- `elemento`: El elemento que está en esa casilla. Inicialmente está vacía hasta que se ubica algún elemento, ya sea agua o un bote.

Las funciones aplicables sobre elementos de esta clase son:

- `apuntar(int s_id)`: El jugador pasado por parámetro apunta un misil sobre este elemento. Devuelve *Aceptado* o *Denegado* basado en el estado de la casilla: sólo las casillas libres pueden ser apuntadas.
- `tocar(int s_id)`: El jugador pasado por parámetro tira un misil sobre esta casilla. Devuelve el resultado.
- `mismo_jugador(int s_id)`: Devuelve un valor *booleano* indicando si el jugador pasado por parámetro es el jugador que realizó todos los toques al elemento en esta casilla.
- `ubicar(Elemento * elemento)`: Ubica el elemento en la casilla.
- `dameId()`: Devuelve el Id de la casilla.
- `dameEstado()`: Devuelve el estado de la casilla.
- `dameResultado()`: Devuelve el resultado del último tiro sobre la casilla.

3.4.4. `tablero.h` y `tablero.cpp`

Modela un tablero, con las casillas y los barcos. Inicialmente todas las casillas tienen el mismo elemento que representa el agua.

Posee los siguientes atributos:

- `tamanio`: Valor de tipo entero que representa el tamaño del tablero (en casillas).
- `casillas`: Lista de casillas que componen el tablero.
- `barcos`: Lista de embarcaciones que se encuentran en el tablero.
- `totalbarcos`: Suma de los tamaños de los barcos (parámetro del server).
- `ocupado_barcos`: Espacio ocupado por barcos en el tablero.
- `cantidad_barcos`: Cantidad de barcos en el tablero.
- `agua`: Elemento que representa al agua en el tablero.

Las funciones aplicables sobre elementos de esta clase son:

- `apuntar(int s_id, int x, int y)`: El jugador pasado por parámetro apunta a la casilla indicada por las coordenadas `x` e `y`.
- `tocar(int s_id, int x, int y)`: El jugador pasado por parámetro toca la casilla indicada por las coordenadas `x` e `y`.
- `mismo_jugador(int s_id, int x, int y)`: Devuelve un valor *booleano* indicando si el jugador pasado por parámetro es el jugador que realizó todos los toques al elemento en la casilla indicada por las coordenadas `x` e `y`.
- `ubicar(int * xs, int * ys, int tamanio)`: `xs` e `ys` son dos listas de tamaño `tamanio`. Estas representan las coordenadas de un barco de ese tamaño. Esta función, de ser posible, ubica al barco. En caso de que no sea posible, devuelve un error.
- `completo()`: Devuelve un valor *booleano* indicando si el tablero tiene ubicados todos los barcos necesarios para comenzar el juego.

3.4.5. `jugador.h` y `jugador.cpp`

Modelan un jugador, con su nombre, su tablero y su puntaje.

Posee los siguientes atributos:

- `nombre`: El nombre del jugador.
- `tablero`: El tablero del jugador.
- `puntaje`: Representa al puntaje que tiene el jugador en ese momento. Es un valor que se computa basado en la cantidad de toques y hundimientos realizados y las características los mismos.
- `cantidad_barcos_flotando`: Es el número de barcos que el jugador tiene flotando (que no han sido hundidos). Al llegar este valor a 0, el jugador ha perdido y no puede continuar disparando.

Las funciones aplicables sobre elementos de esta clase son:

- `apuntar(int s_id, int x, int y)`: El jugador pasado por parámetro apunta a la casilla indicada por las coordenadas `x` e `y`.
- `tocar(int s_id, int x, int y)`: El jugador pasado por parámetro toca la casilla indicada por las coordenadas `x` e `y`.
- `ubicar(int * xs, int * ys, int tamanio)`: `xs` e `ys` son dos listas de tamaño `tamanio`. Estas representan las coordenadas de un barco de ese tamaño. Esta función, de ser posible, ubica al barco. En caso de que no sea posible, devuelve un error.
- `agregar_puntaje(int mas)`: Suma el valor indicado por parámetro al puntaje del jugador.
- `dame_puntaje()`: Devuelve el valor del puntaje.
- `listo()`: Devuelve un valor *booleano* indicando si el jugador está listo para pasar a la fase de batalla.
- `reiniciar()`: Reinicializa las variables internas del jugador al estado inicial.
- `esta_vivo()`: Devuelve un valor *booleano* indicando si el jugador tiene al menos un barco flotando.
- `dame_nombre()`: Devuelve el nombre del jugador.
- `quitar_barcos()`: Quita todos los barcos del tablero del jugador.

3.4.6. `modelo.h` y `modelo.cpp`

Esta clase representa el modelo principal del juego.

Posee dos estructuras auxiliares `tiro_t` y `evento_t` que representan los estados de los tiros realizados por los jugadores y los eventos generados.

Posee los siguientes atributos:

- `cantidad_jugadores`: Cantidad de jugadores inscriptos en el juego.
- `jugadores`: Lista de `jugadores`.
- `tiros`: Lista de estructuras `tiro_t` que representa el estado del tiro de cada jugador. Los jugadores no pueden realizar más de un tiro en un instante de tiempo, por lo que existe un sólo elemento de este tipo por jugador.
- `eventos`: Arreglo de colas de eventos asociada a cada jugador. Hay una cola por cada jugador.
- `jugando`: *Booleano* que representa el estado del juego. Es `true` cuando el juego está en la fase de *Batalla*.

Las funciones aplicables sobre elementos de esta clase son:

- `agregarJugador(std::string nombre)`: Agrega un nuevo jugador al modelo.
- `ubicar(int t_id, int * xs, int * ys, int tamanio)` Ubica el barco en el tablero del jugador `t_id` (ver 3.4.5 para más detalles),
- `borrar_barcos(int t_id)`: Borra los barcos del jugador `t_id`.
- `empezar()`: Pasa a la fase de *Batalla*. Genera un evento de tipo *Start* para cada jugador activo y lo encola en su respectiva cola.

- `reiniciar()`: Reinicia los puntajes y los jugadores, vuelve a la etapa de *Pre-Batalla*.
- `quitar_jugador(int s_id)`: Elimina al jugador `s_id` del juego.
- `apuntar(int s_id, int t_id, int x, int y, int *eta)`: Modela la acción de apuntar a la coordenada (`x`, `y`) del jugador `t_id`. El jugador que realiza la acción es `s_id`. Si la acción es aceptada, genera un evento de tipo *Incoming* para el jugador `t_id` y lo encola en su cola de eventos.
- `tocar(int s_id, int t_id, int x, int y, int *eta)`: Modela la acción de tocar la coordenada (`x`, `y`) del jugador `t_id`. El jugador que realiza la acción es `s_id`. Si la acción es aceptada, genera dos eventos: uno de tipo *Incoming* para el jugador `t_id` y otro de tipo *outgoing* para el jugador `s_id`. Los encola en las respectivas colas de eventos.
- `dame_eta(int s_id)`: Devuelve el *ETA* del último tiro realizado por el jugador `s_id`.
- `actualizar_jugador(int t_id)`: Actualiza el estado del tiro del jugador `t_id`. Si ya transcurrió el *ETA*, realiza la acción `tocar`. Devuelve el evento generado por el toque.
- `hayEventos(int s_id)`: Devuelve la cantidad de eventos pendientes de enviar en la cola del jugador `s_id`.
- `dameEvento(int s_id)`: Devuelve el evento más antiguo en la cola de eventos del jugador `s_id`.

3.4.7. `globales.h` y `globales.cpp`

Contiene variables globales (por simplicidad) y las traducciones (a cadenas de caracteres) de los códigos de error, estado y resultados de las operaciones para facilitar su interpretación.

3.4.8. `jsonificador.h` y `jsonificador.cpp`

Esta clase es la encargada de traducir las respuestas de los comandos a cadenas de caracteres en formato JSON. Tiene acceso a las partes privadas del modelo y sabe cómo interpretar cada uno de los elementos.

Las funciones aplicables sobre elementos de esta clase son:

- `subscribe_resp(int id)`: Genera el mensaje de respuesta al comando `Subscribe` para el jugador dado por parámetro. No accede al modelo para generarla.
- `setup_resp()`: Genera el mensaje de respuesta al comando `Setup`. No accede al modelo para generarla.
- `error(int err)`: Genera un mensaje de error.
- `start(void)`: Genera el mensaje de respuesta al comando `Start`. Accede al modelo iterando por todos los jugadores para obtener sus nombres.
- `player_info(int id)`: Genera un mensaje con la información de un jugador. Es la respuesta al comando `Get_Info`. Accede al modelo, pero sólo al jugador `id`.
- `shoot_resp(int resp, int eta)`: Genera el mensaje de aceptación o rechazo al comando `Shoot`. No accede al modelo para generarla.
- `update(evento_t * event)`: Genera el mensaje para el evento pasado por parámetro. No accede al modelo.

- `finish(void)`: Genera un mensaje de finalización de juego. No es utilizada actualmente dado que los programadores no implementaron esa funcionalidad.
- `scores(void)`: Genera el mensaje de respuesta al comando de control `Get_Scores`. Accede a todos los jugadores del modelo.

3.4.9. `decodificador.h` y `decodificador.cpp`

Esta clase puede interpretar una cadena de caracteres en formato JSON y ejecutar el comando descodificado. Al finalizar la ejecución, utiliza el `jsonificador` descripto en 3.4.8 para devolver las respuestas en el formato correspondiente.

Posee los siguientes atributos:

- `modelo`: El modelo en el que se van a ejecutar las acciones.
- `reader`: Un objeto de una clase correspondiente a la libreria *JsonCPP* que permite transformar un string en objetos de tipo *Json::Value*.
- `root`: El objeto raíz del tipo *Json::Value* que contiene todos los datos del ultimo mensaje interpretado.
- `jsonificador`: El *jsonificador* que permite transformar las respuestas al formato especificado.

Las funciones aplicables sobre elementos de esta clase son:

- `decodificar(const char * mensaje)`: Interpreta un mensaje, lo ejecuta y devuelve la respuesta.
- `dameIdJugador(const char * mensaje)`: Dado un mensaje, intenta extraer el ID del jugador asociado a esa respuesta. No accede al modelo.
- `encodeEvent(int s_id)`: Devuelve una cadena de caracteres que representa el evento más antiguo encolado en la cola del jugador `s_id`. Accede al modelo utilizando la función `dameEvento` descripta en 3.4.6

Las funciones auxiliares (definidas como privadas) de esta clase son:

- `subscribe()`: Ejecuta el comando `Subscribe`. Accede al modelo y a la función `agregarJugador` descripta en 3.4.6. Devuelve la respuesta correspondiente.
- `setup()`: Ejecuta el comando `Setup`. Accede al modelo, intentando ubicar los barcos con la función `ubicar` descripta en 3.4.6. Si ocurre algún error, quita todos los barcos y devuelve el error.
- `get_info()`: Ejecuta el comando `Get_Info`. No accede al modelo directamente, pero utiliza la función `player_info` del *jsonificador* para ello, que sí accede al modelo.
- `shoot()`: Ejecuta la función `apuntar` descripta en 3.4.6 y utiliza el *jsonificador* para devolver la respuesta.
- `start()`: Ejecuta la función `start` descripta en 3.4.6.
- `get_update()`: Verifica el estado del tiro y lo ejecuta. Accede al modelo mediante la función `actualizar_jugador` y utiliza el *jsonificador* para codificar el evento resultante.
- `unsubscribe()`: Ejecuta el comando `Unsubscribe` accediendo al modelo mediante la función `quitarJugador`.

- `scores()` Ejecuta el comando `Get_Scores`. No accede al modelo directamente, pero utiliza la función `scores` del *jsonificador* que sí lo hace.

3.4.10. `server.c`

Es la implementación del ciclo principal del servidor. Crea un modelo basado en los parámetros de entrada, espera conexiones, recibe los mensajes y utiliza un decodificador para interpretar los mensajes y ejecutarlos en el modelo. Una vez obtenida las respuestas, las envía a los jugadores.

Posee dos funciones auxiliares importantes:

- `void atender_jugador(int i)`: Dado un mensaje del jugador `i`, ejecuta el comando pedido y envía la respuesta y los eventos pendientes para este jugador.
- `void void atender_controlador()`: Dado un mensaje del controlador, ejecuta el comando y envía la respuesta.

3.4.11. Funcionalidad no implementada (aún)

Para llevar a cabo un torneo de este juego, es necesario contar con un *controlador*. El controlador es el encargado de enviar comandos de control y monitoreo al *backend* que permitan reiniciar el juego, finalizarlo, obtener los resultados y puntajes, etc.

Los programadores *monoproceso-teístas* no pudieron realizar la funcionalidad pedida: debía existir un puerto especial (12346) que ejecute los comandos de control. Llegaron a implementar la función `atender_controlador()` en el archivo `server.c` pero no pudieron asociarla al puerto pedido. Y su incapacidad no termina aquí: sólo implementaron el comando de control `Get_Scores`.

3.5. Utilización del código provisto

El código que dejaron tras de sí los programadores anteriores está disponible para descargar en la página de la materia.

Para utilizarlo, se deben realizar los siguientes pasos en el servidor:

1. Compilar el servidor de *backend*: `make`
2. Iniciar el servidor de *backend*: `./backend-mono/backend 12345 2 12 19` (puerto, jugadores, tamaño del tablero, suma del tamaño de los barcos)
3. Opcional: iniciar un servidor HTTP para dar acceso a los archivos de juego[4]:
   `cd game && python -m SimpleHTTPServer`

Y en el cliente:

1. Ejecutar *websockify*[5] para que redirija el puerto 5555 local al puerto 12345 del servidor.
   `cd game/websockify && ./run 5555 quentin.narnia:12345`[6]
2. Si se realizó el ítem 3 en el servidor, acceder con un browser a `http://quentin.narnia:8000`. Si no se realizó, abrir el archivo `index.html` con el navegador.

[4] Se debe ejecutar en la carpeta donde esta el archivo `index.html`
[5] `https://github.com/kanaka/websockify`
[6] `quentin.narnia` es el nombre/IP del servidor

Para iniciar múltiples clientes, alcanza con abrir nuevas ventanas o pestañas del navegador y realizar nuevamente el paso 2 del cliente. Si se quiere inicializar un cliente en otra PC, debe realizar el paso 1 en esa PC también.

3.6. Herramientas

3.6.1. netcat

Esta poderosa herramienta permite simular un cliente sin la necesidad de utilizar la interfaz gráfica mediante el uso del navegador web.

Para conectarse, basta con ejecutar `nc 127.0.0.1 12345` en una terminal, donde `127.0.0.1` es la dirección IP del servidor y `12345` el puerto.

Una vez establecida la conexión, basta con enviar los comandos en forma de cadena de caracteres tal como se indica en 3.2.

Ejemplo:

```
quentin:backend-mono fraimondo$ nc 127.0.0.1 12345
{"Name":"Subscribe","Data":{"Name":"Diego Armando"}}
{"Name":"Subscribe","Data":{"Boardsize":12,"Boatssize":19,"Id":1}}|
{"Name":"Setup","Data":{"s_id":1,"boats": [[{"x":6,"y":2 }, {"x":6,"y":3},  {"x":6,"y":4},
{"x":6,"y":5}, {"x":6,"y":6}], [{"x":10,"y":4}], [{"x":9,"y":8}, {"x":10,"y":8}, {"x":11,"y":8}],
[{"x":2,"y":8}, {"x":2,"y":9}], [{"x":4,"y":1}, {"x":4,"y":2}, {"x":4,"y":3}, {"x":4,"y":4}],
[{"x":7,"y":0}, {"x":8,"y":0}, {"x":9,"y":0}, {"x":10,"y":0}]]}}
{"Name":"Setup","Data":"Ok"}|
```

En el ejemplo, se envía el comando `Subscribe` y se obtiene la respuesta. Luego, se envía el comando `Setup` donde también se observa una respuesta satisfactoria.

Cabe destacar que mucha de la lógica de validación de la correctitud de los comandos y los órdenes de los mismos se realiza en la interfaz gráfica, por lo que la utilización incorrecta de los comandos puede resultar en errores severos y la terminación por falla de segmentación del servidor.

4. Entregable

Deberán entregar el trabajo antes del Lunes 17 de Junio a las 23:59 mediante el sistema de entrega `http://so.exp.dc.uba.ar/`

El trabajo consta de dos apartados:

1. En primer lugar, deberán implementar un *Read-Write Lock* **libre de inanición** utilizando únicamente semáforos POSIX y respetando la interfaz provista en los archivos `backend-multi/RWLock.h` y `backend-multi/RWLock.cpp`. Esta implementación deberá tener *tests* que permitan verificar la correctitud de la implementación.
2. En segundo lugar, deberán implementar el servidor de *backend multithreaded* inspirándose en el código provisto y lo desarrollado en el punto anterior.

En la entrega se deberán adjuntar únicamente:

- El documento del informe (en PDF).
- El código fuente **completo y con Makefiles** del servidor de *backend* y los tests del *Read-Write Lock*.

El informe deberá ser de caracter **breve** e incluir el seudocódigo de los algoritmos que se ejecutan en el servidor de *backend* frente a cada petición de un cliente, poniendo énfasis en las primitivas de sincronización al estilo del primer parcial de la materia. Si fuera necesario, puede ser buena idea incluir una explicación del funcionamiento del servidor en lenguaje natural. Cualquier decisión de diseño que hayan tomado deberá ser incluida aquí.

La implementación que realicen del servidor de *backend* debe estar libre de condiciones de carrera y presentar la funcionalidad descripta arriba a cada uno de los clientes. A su vez, debe:

- Permitir que múltiples clientes se conecten al *backend* de forma **simultánea**.
- Permitir que un *controlador* se conecte al *backend* en cualquier momento sin importar el estado del juego y pueda obtener los puntajes de los jugadores con el comando `Get_Scores`.
- Permitir que todos los jugadores realicen disparos sobre jugadores distintos de forma **simultánea**.
- Permitir que varios jugadores consulten el estado del tablero de un mismo jugador de forma **simultánea**.