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

ComboBreaker

Ejercicio / Prueba de Concepto N° 1
Sockets

Objetivos	● Modularización del código en clases Socket, Protocolo, Cliente y Servidor entre otras. ● Correcto uso de recursos (memoria dinámica y archivos) y uso de RAII y la librería estándar STL de C++. ● Encapsulación y manejo de Sockets en C++
Entregas	● **Entrega obligatoria:** clase 4. ● **Entrega con correcciones:** clase 6.
Cuestionarios	● Sockets - Recap - Networking
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. ● Correcta **encapsulación** en clases, ofreciendo una interfaz que **oculte** los detalles de implementación (por ejemplo que **no** haya un *get_fd()* que exponga el *file descriptor* del Socket).. ● Código **ordenado**, separado en archivos .cpp y .h, con **métodos y clases cortas** y con la **documentación** pertinente. ● Empleo de memoria dinámica (*heap*) justificada. Siempre que se pueda **hacer uso del *stack*,** hacerlo antes que el del *heap*. Dejar el *heap* sólo para reservas grandes o cuyo valor sólo se conoce en *runtime* (o sea, es dinámico). Por ejemplo hacer un *malloc(4)* está mal, seguramente un *char buf[4]* en el *stack* era suficiente. ● Acceso a información de archivos de forma ordenada y moderada.

***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 en otras materias (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 *sockets*.

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 sockets. Familiarizarse ***antes*** de embarcarse a desarrollar el TP real te servirá para tomar mejores decisiones.

Descripción

El juego del Jazz Jack Rabbit cuenta con una serie de acciones especiales que realizan los personajes para golpear a los enemigos. En este ejercicio se implementará el protocolo que le permitirá al jugador (cliente) decirle al servidor los comandos necesarios para que los conejos utilicen sus poderes.
El servidor, por su lado, recibirá el input, y en función de la secuencia de acciones que el cliente quiere realizar, aplicará distintas acciones. Estas acciones variarán en función del orden de los inputs enviados.
El servidor además tendrá un contador de la cantidad de acciones recibidas.

> **Nota:** en esta PoC el servidor atiende a **un solo cliente y nada más**. En la PoC de *threads* aprenderás cómo atender a múltiples clientes en paralelo.
>
> **Podes usar el socket provisto por la cátedra** (usa el último commit): https://github.com/eldipa/sockets-en-cpp . Solo no te olvides de **citar** en el readme la fuente y su licencia.

Protocolo

El cliente enviará las acciones a realizar en **mensajes de 1 byte sin signo**.

<acción>

Las acciones a realizar pueden ser las siguientes:

Accion	***Valor***
NOP	0x00
JUMP	0x01
RIGHT	0x02
LEFT	0x03
DUCK	0x04
HIT	0x05

El servidor recibirá instrucciones hasta que llegue un No operation (**NOP**). Cuando recibe este comando, el servidor resolverá el stream de inputs enviado por el cliente. Luego de procesarlo, retorna un string con las acciones realizadas. Este string lo envía de la siguiente forma:

- 2 **bytes sin signo en formato big endian**, indicando el largo del string.
- la tira de bytes del string, sin contar el terminador de palabra (‘\0’).

<largo del payload> <payload>

Formato de Línea de Comandos

El servidor recibirá por línea de comandos el puerto o servicio donde escuchará una **única conexión**.

./server <puerto>

Por su parte, el cliente recibirá la ip o host y el puerto o servicio.

./client <hostname> <servicio>

Leerá de entrada estándar las instrucciones que va a querer realizar.

Por entrada estándar el cliente recibirá líneas de texto, donde cada línea tiene N palabras separados por **1 o mas espacios en blanco**, y cada palabra será una de las acciones mencionadas en la tabla anterior.

<acción 1> <acción 2> <acción 3> … <acción N>

El cliente leerá de una línea a la vez. Cada línea **representa un movimiento completo**.

NOTA: La única acción que **NUNCA** se encontrará en el archivo será la acción **NOP**. Esta operación se infiere y se envía al finalizar la serialización de cada movimiento.

Un ejemplo de secuencia de inputs que puede leer el cliente de su entrada estándar es el siguiente.

JUMP JUMP HIT DUCK LEFT

Se podrá suponer que el archivo no contiene errores: las acciones son válidas y no habrá caracteres incorrectos. Notar que entre cada argumento puede haber una cantidad arbitraria de **espacios**.

Por cada línea el cliente enviará el pedido al servidor, terminando esta con un **NOP** y luego esperará la respuesta. La respuesta recibida por el servidor será impresa por salida estándar, junto con un salto de línea.

En el ejemplo anterior, el resultado del movimiento será:

```
UPPERCUT DUCK LEFT
```

En una única ejecución del cliente, se podrán mandar múltiples líneas, donde cada una podrá contener cualquier cantidad de combos y acciones individuales.

Códigos de Retorno

Tanto el cliente como el servidor retornarán 0 en caso de éxito o 1 en caso de error (argumentos inválidos, archivos inexistentes o algún error de sockets).

Ejemplo de Ejecución

Lanzamos el servidor de la siguiente manera:

```
$ ./server 8080
```

El servidor escuchará en el puerto TCP 8080.

Ahora corremos el cliente:

```
$ ./client localhost http-alt
```

El cliente se conectará por TCP a la IP/hostname localhost, puerto/servicio http-alt.
Nótese que el servicio http-alt es el puerto 8080.
Luego, el cliente leerá una línea por entrada estándar y serializará en comandos de 1 byte.
Cuando termina de serializar la línea, envía un byte de finalización de movimiento

```
JUMP          JUMP
JUMP                  JUMP    HIT DUCK   LEFT
```

El cliente leerá la primer linea, que será “JUMP JUMP”. Esto se serializa en el siguiente mensaje:

```
01 01 00
```

Nótese que el cliente envía 3 bytes en total pero para facilitarte la lectura he escrito estos bytes en hexadecimal separado por espacios. Esto es a solo los efectos de escribir esto en este documento.

El servidor recibe el mensaje y verifica si hay algun combo presente. Procesa los movimientos y responde con los resultados:

```
0009 4a554d50 20 4a554d50
```

Es decir, envía un header de **2 bytes big endian sin signo**, que representa el largo del mensaje (9 caracteres en hexadecimal). Seguido a este header envía el string de 9 caracteres “JUMP JUMP” es formato ascii.

El cliente lo recibe e imprime

```
JUMP JUMP
```

Nota: el servidor **NO ENVIA** ni el salto de línea final, ni el ‘\0’ de todo string en C/C++

El cliente vuelve a leer una nueva línea, que contiene un único comando **JUMP JUMP HIT DUCK LEFT**. Lo serializa y lo envía al servidor.

```
01 01 05 04 03 00
```

El servidor contesta con el siguiente mensaje

```
0012 5550504552435554 20 4455434b 20 4c454654
```

y el cliente lo imprime en pantalla

```
UPPERCUT DUCK LEFT
```

Una vez finalizado de parsear el archivo, el cliente realiza un **shutdown** del socket, y libera los recursos ordenadamente. Ese shutdown hará que el servidor se destrabe de su receive, liberando los recursos ordenadamente. Antes de finalizar, el servidor imprimirá la cantidad de movimientos ejecutados por salida estándar:

```
Actions: 5
```

Combos disponibles:

Secuencia	Resultado
JUMP JUMP HIT	UPPERCUT
HIT DUCK LEFT	SIDEKICK
LEFT RIGHT JUMP HIT	HIGHKICK

Recomendaciones

Los siguientes lineamientos son claves para acelerar el proceso de desarrollo sin pérdida de calidad:

1. **¡Repasar los temas de la clase!** Los videos, las diapositivas, los handouts, las guías, los ejemplos, los tutoriales, los recaps. **Todo**. ***Muchas soluciones y ayudas están ahí.***
2. **¡Programar por bloques!** No programes todo el TP y esperes que funcione. Debuggear un TP completo es más difícil que probar y debuggear sus partes por separado. No te compliques la vida con diseños complejos. **Cuanto más fácil sea tu diseño, mejor**.
   Dividir el TP en bloques, codearlos, testearlos por separada y luego ir construyendo hacia arriba.
   ¡Si programas así, podés hasta hacerles tests fáciles antes de entregar!. Teniendo cada bloque, es fácil armar un bloque de más alto nivel que los use. La **separación en clases** es crucial.
3. **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)

Usa la librería estándar de C++ y las clases que te damos en la cátedra. Cuando más puedas reutilizar código oficial mejor: menos bugs, menos tiempo invertido.

4. **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**.

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** (salvo código dado por la cátedra). Para este trabajo no es necesario el uso de excepciones (que se verán en trabajos posteriores).
3. Todo socket utilizado en este TP debe ser **bloqueante** (es el comportamiento por defecto).
4. Deberá haber una clase **Socket** tanto el socket aceptador como el socket usado para la comunicación. Si lo preferís podes separar dichos conceptos en 2 clases.
5. Deberá haber una clase **Protocolo** que encapsule la serialización y deserialización de los mensajes entre el cliente y el servidor. Si lo preferís podes separar la parte del protocolo que necesita el cliente de la del servidor en 2 clases pero asegurate que el código en común no esté duplicado.
    - La idea es que ni el cliente ni el servidor tengan que armar los mensajes *"a mano"* sino que le delegan esa tarea a la(s) clase(s) **Protocolo**.