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

Counter Strike 2D

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	● 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. ● Separación del protocolo de la capa de aplicación. ● 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
- Descripción
- Formato de Línea de Comandos
- Códigos de Retorno
- Entrada y Salida Estándar
- Ejemplos de Ejecución
  - Ejemplo 1
  - Ejemplo N
- Restricciones
- Referencias

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

Previo a las rondas de juego en Counter Strike 2D, los jugadores cuentan con determinado tiempo en la etapa de preparación para comprar el armamento necesario. En esta PoC se desarrollará una simplificación de esta fase del juego. El servidor aceptará la conexión de un único cliente, quién solicitará la compra de armas y de balas.

Acciones del cliente y requerimientos del servidor

Una vez establecida la conexión cliente-servidor, el cliente enviará su nombre de usuario al servidor. El servidor imprimirá por salida estándar:

- `<username> has arrived!`

y enviará un mensaje al cliente indicando el tipo de protocolo (ver Protocolo) que se utilizará a partir de este punto en la comunicación. Además, enviará el equipamiento y saldo inicial del cliente, quién lo imprime por salida estándar:

- `money: $500 | knife: equipped | primary: not_equipped | secondary: glock, 30`

El servidor se quedará esperando por peticiones del cliente, quien lee de entrada estándar las ordenes de compra. Según si el cliente quiere comprar un arma o balas para un arma, leerá:

- `buy <weapon>`
- `ammo <weapon> <count>`

La instrucción *buy* permitirá al cliente comprar su arma primara, la cual puede ser: *awp, ak-47* o *m3*. La compra de un arma incluye un pack inicial de 30 balas. La instrucción *ammo*, acepta también el arma *glock*.

Por simplificación, todas las armas tienen el mismo valor ($100) y todas las balas también ($10). Si la compra no puede concretarse, el servidor imprimirá, según el caso:

- `Not enough money to buy weapon`
- `Not enough money to buy ammo`

En todos los casos, el servidor enviará el equipamiento y saldo al cliente.

El proceso de compra se repite hasta que el cliente lee

- `exit`

de entrada estándar.

Protocolo

El cliente enviará su username con el siguiente formato:

- `0x01 <length> <username>` donde `0x01` es un byte con el número literal `0x01`, `<length>` es un entero de dos bytes en big endian con el largo del username en ASCII, seguido con el username.

El servidor enviará el tipo de protocolo a utilizar con el siguiente formato.

- `0x06 <tipo-de-protocolo>` donde `0x06` es un byte con el número literal `0x06` y `<tipo-de-protocolo>` es un byte con el número literal `0x07` si el protocolo a utilizar es **binario**, y `0x08` si el protocolo a utilizar es **de texto**.

Protocolo binario

Si el tipo de protocolo es **binario**, los mensajes respetarán el siguiente formato.

El cliente enviará los siguientes mensajes.
Para la compra de un arma:

- `0x02 <weapon-code>` donde `0x02` es un byte con el número literal `0x02`, y `<weapon-code>` es un byte con el número indicado en la tabla de conversión de arma - código de arma (en la siguiente página).

Para la compra de balas:

- `0x03 <type-weapon> <count>` donde `<type-weapon>` es `0x01` si la compra es referida al arma primaria o `0x02` si es referida al arma secundaria. `<count>` es un entero de dos bytes en big endian con la cantidad de balas a comprar.

Por su parte, el servidor enviará el equipamiento y saldo con el siguiente formato:

- `0x07 <money> <knife> <primary-weapon-code> <primary-weapon-ammo> <secondary-weapon-code> <secondary-weapon-ammo>` siendo:

    1. `<money>` un entero de dos bytes en big endian.
    2. `<knife>` el número literal `0x01` indicando que tiene el cuchillo equipado.
    3. `<primary-weapon-code>` un byte con el código del arma primaria.
    4. `<primary-weapon-ammo>` dos bytes en big endian con la cantidad de balas para el arma primaria. Si no tiene arma primaria equipada, envía dos bytes con el literal `0x00`.
    5. `<secondary-weapon-code>` el código del arma secundaria (siempre *glock*).
    6. `<secondary-weapon-ammo>` la cantidad de balas para el arma secundaria.

Protocolo de texto

Si el tipo de protocolo es **de texto**, los mensajes respetarán el siguiente formato.

El cliente enviará los siguientes mensajes.
Para la compra de un arma:

- “`buy.weapon:<weapon>\n`” donde `<weapon>` es el nombre del arma a comprar.

Para la compra de balas:

- “`buy.ammo.<weapon-type>:<count>\n`” donde `<weapon-type>` es `primary` si busca comprar municiones para el arma primaria, y `secondary` para la secundaria.

Por su parte, el servidor enviará el equipamiento y saldo con el siguiente formato:

- “`equipment.money:<money>\n`”
- “`equipment.knife:true\n`”
- “`equipment.primary:<weapon>,<ammo>\n`”
- “`equipment.secondary:<weapon>,<ammo>\n`”

Arma	Código de arma
none	0x00
glock	0x01
ak-47	0x02
m3	0x03
awp	0x04

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.

Podes implementar **tu** propio socket o reescribir algunos métodos de la clase socket provista para practicar, pero hacelo cuanto tengas el TP ya andando así tenes *peace of mind*.

Recordar que tenes que tener el código de parsing, impresión y lógica del juego **separado** del código de armado de protocolo así como tener separado este del código del socket. No tengas métodos que parseen, armen un mensaje y lo envíen por socket todo en un solo lugar, así como no tengas la lógica del juego llamando directamente a `std::cout`.

En esta PoC te parecerá un overkill pero cuando te enfrentes al TP Final el *"separar"* te permitirá organizarte mejor en tu equipo, trabajar en paralelo y testear por separado. Es tu ***única arma para mitigar la complejidad y el gran tamaño del TP.***

Nota: los archivos son **archivos de texto**, están **libres de errores** y cumplen el formato especificado.

Recomendación: puede que te interese repasar algún **container de la STL** de C++ como `std::vector` y `std::map`.

Recomendación: usar el `operador >>` de `std::fstream` para la lectura de la entrada estándar y de los archivos del servidor por que simplifica enormemente el parsing. Puede que también quieras ver los métodos `getc` y `read` de `std::fstream` para leer ciertas partes de los archivos.

Que sea C++ quien parsee por vos.

Formato de Línea de Comandos

```
./server <puerto> <tipo-de-protocolo>
```

siendo <tipo-de-protocolo> “text” o “binary”.

```
./client <hostname> <servicio> <username>
```

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

Nota: Marcamos con **(a)** el ejemplo utilizando protocolo **binario**, y con **(b)** el mismo ejemplo utilizando el protocolo **de texto**.

Lanzamos el servidor:

```
(a)   ./server 12345 binary
(b)   ./server 12345 text
```

Y lanzamos el cliente A:

```
./client 127.0.0.1 12345 mate
```

En este punto, el cliente ya se conectó al servidor.

El cliente envía al servidor su nombre de usuario:

```
01 | 00 04 | 6d 61 74 65
```

```
El server imprime:

- mate has arrived!

Además el server indica que el protocolo a utilizar  enviando:

- (a) 06 | 07
- (b) 06 | 08
```

Seguido del saldo y equipamiento inicial:

```
- (a) 07 | 01 f4 | 01 | 00 | 00 00 | 01 | 00 1e
- (b)
    - equipment.money:500
    - equipment.knife:true
    - equipment.primary:none,0
    - equipment.secondary:glock,30
```

El cliente imprime:

```
- money: $500 | knife: equipped | primary: not_equipped | secondary: glock, 30
```

Luego el cliente lee de stdin:

```
- ammo glock 10000
```

Por lo que envía al server:

```
- (a) 03 | 02 | 27 10
```

```
(b) buy.ammo.secondary:10000
```

Dado que el cliente no tiene el saldo suficiente para comprar esa cantidad de balas, el server imprime:

- `Not enough money to buy ammo`

Y vuelve a enviar el equipamiento, sin cambios en el mismo:

- `(a) 07 | 01 f4 | 01 | 00 | 00 00 | 01 | 00 1e`
- `(b)`
  - `equipment.money:500`
  - `equipment.knife:true`
  - `equipment.primary:none,0`
  - `equipment.secondary:glock,30`

El cliente imprime nuevamente:

- `money: $500 | knife: equipped | primary: not_equipped | secondary: glock, 30`

y luego lee de stdin:

- `buy awp`

Por lo que envía

- `(a) 02 | 04`
- `(b) buy.weapon:awp`

El servidor procesa el mensaje y envía el equipamiento y saldo actualizado:

- `(a) 07 | 01 90 | 01 | 04 | 00 1e | 01 | 00 1e`
- `(b)`
  - `equipment.money:400`
  - `equipment.knife:true`
  - `equipment.primary:awp,30`
  - `equipment.secondary:glock,30`

El cliente imprime:

- `money: $400 | knife: equipped | primary: awp, 30 | secondary: glock, 30`

Luego el cliente lee de stdin:

- `exit`

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**.