

Need for Speed

Ejercicio / Prueba de Concepto N° 1

Sockets

Objetivos	<ul style="list-style-type: none">• 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	<ul style="list-style-type: none">• Entrega obligatoria: clase 4.• Entrega con correcciones: clase 6.
Cuestionarios	<ul style="list-style-type: none">• Sockets - Recap - Networking
Criterios de Evaluación	<ul style="list-style-type: none">• Resolución completa (100%) de los cuestionarios <i>Recap</i>.• 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 <i>get_fd()</i> que exponga el <i>file descriptor</i> del Socket)..• Código ordenado, separado en archivos .cpp y .h, con métodos y clases cortas y con la documentación pertinente.• Empleo de memoria dinámica (<i>heap</i>) justificada. Siempre que se pueda hacer uso del stack, hacerlo antes que el del <i>heap</i>. Dejar el <i>heap</i> sólo para reservas grandes o cuyo valor sólo se conoce en <i>runtime</i> (o sea, es dinámico). Por ejemplo hacer un <i>malloc(4)</i> está mal, seguramente un <i>char buff[4]</i> en el <i>stack</i> 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).

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 arrancar las carreras en el juego Need for Speed, el jugador debe elegir el auto, y adicionalmente, comprarle mejoras. En esta PoC se desarrollará una simplificación de esta fase del juego. El servidor aceptará la conexión de un único cliente, quién solicitará la información de los autos disponibles, así como elegirá el suyo y le agregará mejoras, de así desearlo.

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.

Funcionalidad pedida

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

- Hello, <username>

Y enviará la información del saldo inicial del jugador, quién lo imprime por salida estándar:

- Initial balance: 10500

El servidor se quedará esperando por peticiones del cliente, quien lee de entrada estándar la acción que el jugador desea llevar a cabo. Las acciones disponibles son las siguientes:

- current_car
- car_market
- buy_car <car-name>

La acción current_car devuelve la información del coche actualmente elegido. Si no se tiene un auto elegido, el servidor devolverá el mensaje de error apropiado:

- No car bought

La acción car_market devuelve la información de todos los autos disponibles.

La acción buy_car <car> permite al usuario comprar el auto establecido en <car>. Si ya tenía un auto previo, se pisa con el nuevo comprado. Si la compra no puede concretarse por falta de saldo, el servidor no concretará la compra, y devolverá el mensaje de error apropiado:

- Insufficient funds

Acciones del cliente

El cliente define las acciones a ejecutar en un archivo.txt. Se debe enviar el nombre del archivo en la línea de ejecución del cliente.

A continuación, se incluye un ejemplo sencillo.

```
username Alice
get_market
buy_car FordFocus
get_current_car
```

El formato del archivo enviado será el comando en un solo string (sin espacios), seguido por un espacio y el parámetro a enviar.

Autos disponibles

Los autos disponibles se cargan al servidor con un archivo.txt, así como el saldo inicial del usuario. Primero se debe incluir el saldo del usuario con el string “`money`”, y cada auto con el string “`car`”. Se debe enviar el nombre del archivo en la línea de ejecución del servidor.

A continuación, se incluye un ejemplo sencillo.

```
money 15000
car ToyotaCorolla 2018 12000
car HondaCivic 2020 14000
car FordFocus 2017 11000
```

El formato del archivo enviado será el comando en un solo string (sin espacios), seguido por un espacio y el parámetro a enviar. La primera línea será con el comando `money` y las siguientes, los autos disponibles del market, comenzando por el comando `car`.

Protocolo

Tabla con Códigos de Acciones

Los códigos de acciones se detallan en la siguiente tabla. Por protocolo, **todos se envían en un byte con el valor literal**.

Acción	Descripción	Código
SEND_USERNAME	El cliente envía el username al servidor	0x01
SEND_INITIAL_MONEY	El servidor envía el balance inicial al cliente	0x02
GET_CURRENT_CAR	El servidor envía el auto actual al cliente	0x03
SEND_CURRENT_CAR	El cliente pide el auto actual al servidor	0x04
GET_MARKET_INFO	El cliente pide la información del mercado al servidor	0x05
SEND_MARKET_INFO	El servidor envía la información del mercado al cliente	0x06
BUY_CAR	El cliente pide comprar un auto al servidor	0x07

SEND_CAR_BOUGHT	El servidor envía el auto comprado al cliente	0x08
SEND_ERROR_MESSAGE	El servidor envía el error ocurrido al cliente	0x09

Recomendamos que se concentren todas estas constantes en un archivo en la carpeta common llamada constants.h.

- Los **códigos de acción** se enviarán en un byte con el valor literal.
- Las **longitudes** (de arrays o strings) y **años** se enviarán en dos bytes en big endian.
- Las **cantidades** (dinero, precio) se enviarán en cuatro bytes en big endian.

Cliente

El cliente enviará los siguientes mensajes.

Para enviar el username (username)

- SEND_USERNAME <length> <username>
 - SEND_USERNAME es un byte con el número literal 0x01,
 - <length> es la **longitud** del username
 - <username> es el username del usuario

Para obtener el auto actual (current_car)

- GET_CURRENT_CAR
 - GET_CURRENT_CAR es un byte con el número literal 0x04

Para ver los autos disponibles (car_market)

- GET_MARKET_INFO
 - GET_MARKET_INFO es un byte con el número literal 0x05

Para comprar un auto (buy_car <length> <car-name>)

- BUY_CAR <length> <car-name>
 - BUY_CAR es un byte con el número literal 0x07
 - <length> es la **longitud** del nombre
 - <car-name> es el nombre del auto

Servidor

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

- SEND_INITIAL_MONEY <money>
 - SEND_INITIAL_MONEY es un byte con el número literal 0x02
 - <money> es la **cantidad** de saldo inicial

Respuesta de obtener el auto actual (current_car):

- SEND_CURRENT_CAR <car-info>
 - SEND_CURRENT_CAR es un byte con el número literal 0x04
 - <car-info> string en formato de texto con la información del auto, donde se debe enviar <length name> <name> <year> <price>
 - <length name> es la **longitud** del nombre del auto
 - <name> el nombre del auto
 - <year> es el **año** del modelo de auto
 - <price> se almacena como float pero se envía como **cantidad** de dinero que cuesta el auto. Para pasarlo por protocolo, no está permitido enviar el float directamente (ya que diferentes arquitecturas interpretan el float de diferente manera). Se debe multiplicar por 100 el valor antes de pasarlo, y castearlo a `uint32_t` en big endian.

Respuesta de ver autos disponibles:

- SEND_MARKET_INFO <num-cars> [<car-info>]
 - SEND_MARKET_INFO es un byte con el número literal 0x06
 - <num-cars> es la **longitud** del vector de autos enviado
 - Para cada auto: <car-info> string en formato de texto con la información del auto, donde se debe enviar <length name> <name> <year> <price>
 - <length name> es la **longitud** del nombre del auto
 - <name> el nombre del auto
 - <year> es el **año** del modelo de auto
 - <price> se almacena como float pero se envía como **cantidad** de dinero que cuesta

Respuesta de comprar un auto:

- SEND_CAR_BOUGHT <car-info> <remaining-money>
 - SEND_CAR_BOUGHT es un byte con el número literal 0x08
 - <car-info> toda la info del auto recién comprado, mismo formato que en mensajes anteriores
 - <remaining-money> **cantidad** de dinero restante en el saldo del usuario

Respuesta en caso de error:

- SEND_ERROR_MESSAGE <lenght> <error-message>
 - SEND_ERROR_MESSAGE es un byte con el número literal 0x09
 - <lenght> es la **longitud** del mensaje
 - <error-message> el string del mensaje

Formato de Línea de Comandos

`./server <puerto> <nombre-archivo>`

Ejemplo: `./server 8080 server-basic-flow.txt`

```
./client <hostname> <servicio> <nombre-archivo>
```

Ejemplo `./client localhost 8080 client-basic-flow.txt`

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

Comenzando con el siguiente archivo de servidor

```
money 15000
car ToyotaCorolla 2018 12000
car HondaCivic 2020 14000
car FordFocus 2017 11000
```

Y el siguiente archivo de cliente

```
username Alice
get_current_car
get_market
buy_car FordFocus
get_current_car
buy_car ToyotaCorolla
```

La ejecución esperada es la siguiente:

1. El servidor primero levantará el mercado con todos los autos disponibles y el saldo inicial del cliente del archivo. Por pantalla, imprime

Server started

2. El cliente iniciará la conversación enviando por protocolo el siguiente mensaje:

```
0x01 5 Alice
```

3. El servidor recibirá el pedido y mostrará por pantalla un mensaje de bienvenida,

Hello, Alice

4. El servidor le enviará al cliente el saldo inicial del usuario por protocolo con el siguiente mensaje

```
0x02 15000
```

Y lo imprimirá por pantalla de la siguiente manera

Initial balance: 15000

5. El cliente recibirá el mensaje e imprimirá la información por pantalla:

Initial balance: 15000

6. El cliente enviará el comando `get_current_car` por protocolo con el siguiente mensaje
`0x04`

7. El servidor recibirá el pedido y enviará por protocolo la siguiente respuesta.
`0x09 13 No car bought`

Y tanto el cliente como el servidor imprimirán por pantalla lo siguiente

Error: No car bought

8. El cliente enviará el comando `get_market` por protocolo con el siguiente mensaje

`0x05`

9. El servidor recibirá el pedido y enviará por protocolo la siguiente respuesta

`0x06 3 13 ToyotaCorolla 2018 12000 10 HondaCivic 2020 14000
9 FordFocus 2017 11000`

E imprimirá por pantalla

3 cars sent

10. El cliente recibirá la respuesta e imprimirá el resultado por consola.

*ToyotaCorolla, year: 2018, price: 12000
HondaCivic, year: 2020, price: 14000
FordFocus year: 2017, price: 11000*

11. El cliente enviará el comando `buy_car` con el parámetro `FordFocus` por protocolo con el siguiente mensaje

`0x07 9 FordFocus`

12. El servidor recibirá el pedido y enviará por protocolo la siguiente respuesta

`0x08 9 FordFocus 2017 11000 4000`

E imprimirá por pantalla

New cars name: FordFocus --- remaining balance: 4000

13. El cliente recibirá el resultado e imprimirá un mensaje por pantalla:

*Car bought: FordFocus, year 2017, price 11000
Remaining balance: 4000*

14. El cliente enviará el comando `get_current_car` con el siguiente mensaje

`0x04`

15. El servidor recibirá el pedido y enviará por protocolo la siguiente respuesta

`0x04 9 FordFocus 2017 11000`

E imprimirá por pantalla

Car FordFocus 11000 2017 sent

16. El cliente recibirá la respuesta e imprimirá por pantalla la información del auto. Por ejemplo,

Current car: FordFocus, year: 2017, price: 11000

17. El cliente enviará el comando `buy_car` con el parámetro `ToyotaCorolla` por protocolo con el siguiente mensaje

`0x07 13 ToyotaCorolla`

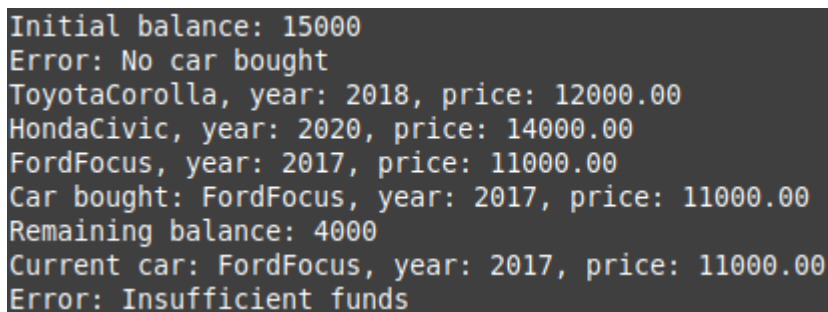
18. El servidor recibirá el pedido y enviará por protocolo la siguiente respuesta

`0x09 18 Insufficient funds`

Y tanto el cliente como el servidor imprimirán por pantalla

Error: Insufficient funds

Se adjunta una imagen de la consola del **cliente** para este caso:



```
Initial balance: 15000
Error: No car bought
ToyotaCorolla, year: 2018, price: 12000.00
HondaCivic, year: 2020, price: 14000.00
FordFocus, year: 2017, price: 11000.00
Car bought: FordFocus, year: 2017, price: 11000.00
Remaining balance: 4000
Current car: FordFocus, year: 2017, price: 11000.00
Error: Insufficient funds
```

Se adjunta una imagen de la consola del **servidor** para este caso:

```
Server started
Hello, Alice
Initial balance: 15000
Error: No car bought
3 cars sent
New cars name: FordFocus --- remaining balance: 4000
Car FordFocus 11000 2017 sent
Error: Insufficient funds
```

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.**
5. No implementes todo el cliente por un lado, todo el servidor por otro, y después conectes el cliente y servidor. Eso casi nunca funciona. Primero conectá el cliente y servidor con un caso simple, básico. Y sobre una base sólida, que funciona, construí el resto de las features de a una, sin avanzar hasta que lo anterior funcione.

Aclaraciones

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 hazlo 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.

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 podés 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 podés 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**.