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

Dune

Ejercicio 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 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. 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

Descripción

Dune el jugador puede construir edificios que le permitirán desbloquear nuevas unidades y tecnologías y aumentar la producción.

Estos edificios no pueden ser construidos en cualquier lado, cada edificio ocupa cierto tamaño por lo que debe haber suficiente espacio libre sobre las rocas para construirlo.

En este ejercicio se implementará la parte del protocolo que le permitirá al jugador (el cliente) decirle al servidor que edificio y en donde se quiere construir.

El servidor por su lado verificará las dimensiones del edificio y su locación. Tras la verificación le responderá al cliente si se pudo o no realizar la construcción.

Nota: la restricción de que los edificios deben construirse a lo sumo a 5 bloques de distancia de otro edificio no será implementada en este ejercicio.

Tampoco es requerido implementar la demolición y asumiremos que todo el escenario está constituido de roca donde se puede construir libremente.

Protocolo

El cliente enviará el siguiente mensaje:

```
<edificio> <posicion-x> <posicion-y>
```

Donde:

`<edificio>` es un número **sin signo de 1 byte** y representa al edificio a construir. Para este ejercicio estos

son Trampas de Aire (número 1), Cuartel (número 2) y Silo (número 3).

`<posicion-x>` y `<posicion-y>` son dos números **sin signo de 2 bytes** cada uno en **big endian** y representan el lugar de la construcción del extremo superior izquierdo.

Por ejemplo, un Cuartel tiene dimensiones 2x3. Esto es, ocupa 2 bloques en el eje horizontal (x) y 3 bloques en el eje vertical (y).

Si un jugador decide construir un Cuartel en la posición (x=10, y=20), el Cuartel terminará ocupando los bloques (10 20), (11, 20), (10 21), (11, 21), (10 22) y (11, 22).

Nótese como entonces la posición (10, 20) representa la posición del extremo superior izquierdo. O sea, la construcción se realiza desde esa posición hacia la derecha y hacia abajo.

El servidor por su lado responderá con:

```
<status>
```

Donde:

`<status>` es un número **sin signo de 1 byte** que representa el resultado de la construcción. Este número puede ser Construcción Exitosa (0) o Lugar Insuficiente (1).

En el caso de Lugar Insuficiente, el servidor enviará un mensaje adicional:

```
<cnt> [<posicion-x> <posicion-y> ...]
```

Donde:

`<cnt>` es un número **sin signo de 1 byte** que cuenta cuántos bloques o celdas **no** están disponibles
`<posicion-x>` y `<posicion-y>` son números **sin signo de 2 bytes en big endian** de las posiciones (x, y) de los bloques que están ocupados. Esta lista de posiciones debe estar **ordenada** de menor a mayor por la coordenada **y** primero y por la coordenada **x** en segundo lugar.

Formato de Línea de Comandos y Archivos

El servidor recibirá por línea de comandos el puerto o servicio donde escuchará una **única conexión** y las dimensiones del mapa o escenario que supondrá que está completamente cubierto de roca. O sea que inicialmente se puede construir en cualquier lado.

Por su parte, el cliente leerá de un archivo de texto que construcciones va a querer realizar.

El formato del archivo es por linea, donde cada linea tiene 3 campos separados por 1 o más espacios en blanco:

```
<edificio> <posicion-x> <posicion-y>
```

Donde <edificio>, <posicion-x> y <posicion-y> son números en **texto**, no en binario y representan el edificio a construir y el dónde.

Por ejemplo:

```
1 10 20
1 15 25
1       22      44
2   10      20
```

Se podrá suponer que el archivo no contiene errores: los números son válidos, no hay ni más ni menos campos por línea 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 y esperará la respuesta. Respuesta que el cliente imprimirá con los siguientes mensajes

```
Construcción Exitosa\n
```

si se recibió un <status> con valor 0.

```
Lugar Insuficiente. Posiciones bloqueadas: (<posicion-x>,<posicion-y>)...\n
```

si se recibió un <status> con valor 1. Las posiciones deben ser impresas en el **mismo orden** en el que se recibieron del servidor.

Por ejemplo:

```
Lugar Insuficiente. Posiciones bloqueadas: (10,20) (11,20) (10,21)\n
```

Nótese que cada línea termina en un salto de línea (\n) incluida la última línea impresa por el cliente.

Tras procesar cada mensaje del cliente, el servidor deberá imprimir por salida estándar el estado actual del escenario representando con puntos (.) los lugares disponibles para la construcción y con una letra (T, C, S) los lugares ya construidos. Entre cada impresión de un escenario deberá haber una línea vacía.

Códigos de Retorno

Si hay algún problema con los argumentos pasados (puerto invalido, archivo inexistente) se debe retornar un código de 1.

De lo contrario retornar 0.

Ejemplo de Ejecución

Lanzamos el servidor de la siguiente manera:

```
$ ./server 8080 20 6
```

El servidor escuchará en el puerto TCP 8080 y emulará un escenario de 20 bloques (o celdas) de ancho (eje x) y 6 de alto (eje y).

Supongamos el siguiente archivo `const.txt`

```
1 1 2
2 2 3
2 5 3
```

Ahora corremos el cliente:

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

El cliente se conectara por TCP a la IP/hostname `localhost`, puerto/servicio `http-alt`. Nótese que el servicio `http-alt` es el puerto 8080.

Luego, por cada línea del archivo el cliente enviará un mensaje al servidor.

Por ejemplo, para la primer línea "`1 1 2`", que representa un *"construir una Trampa de Aire en la posición x=1 y=2"*, el cliente enviará el siguiente mensaje:

```
01 0001 0002
```

Nótese que el cliente envía 5 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 que se quiere crear una Trampa de Aire de 3x3 en la posición x=1, y=2.

Como las posiciones (1,2), (2,2), (3,2), (1,3), (2,3), (3,3), (1,4), (2,4) y (3,4) están libres, el servidor responde con un mensaje de éxito:

```
00
```

El servidor luego imprimirá el estado del escenario hasta ese momento:

```
..................
..................
.TTT..............
.TTT..............
.TTT..............
..................
```

Nótese que las coordenadas son 0-based.

El cliente recibe el mensaje de éxito e imprime

```
Construccion Exitosa\n
```

Luego el cliente enviará el segundo pedido de construcción, *"construir un Cuartel en x=2, y=3"*:

```
02 0002 0003
```

El servidor recibe este mensaje pero detecta que no todas las posiciones están libres ya que algunas colisionan con la construcción previa.

Entonces envía el siguiente mensaje

```
01 04 0002 0003 0003 0003 0002 0004 0003 0004
```

Nótese el orden de las posiciones, ordenadas por la coordenada y primero y por la coordenada x en segundo lugar.

El servidor imprime:

```
..................
..................
.TTT..............
.TTT..............
.TTT..............
..................
```

El cliente entonces recibe el mensaje e imprime

```
Lugar Insuficiente. Posiciones bloqueadas: (2,3) (3,3) (2,4) (3,4)\n
```

El cliente luego envía el siguiente mensaje

```
02 0005 0003
```

El servidor recibe este mensaje, verifica que todo está bien y envía:

```
00
```

E imprime:

```
..................
..................
.TTT..............
.TTT.CC...........
.TTT.CC...........
.....CC...........
```

El cliente imprime entonces

```
Construccion Exitosa\n
```

Como ya no hay más líneas a procesar, el cliente cierra la conexión TCP y finaliza.

El servidor detecta el cierre por parte del cliente y también finaliza.

Recomendaciones

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

1. Tener siempre a mano las páginas de manual, y al abrirlas leerlas con mucha atención. Sobre todo para las funciones que no hayas usado nunca: no cuesta nada escribir '*man send*' o '*man recv*' en la consola y leer *bien* qué dice cada una sobre los valores del último parámetro o sobre los valores de retorno.
2. **¡Repasar los temas de la clase!** Los videos, las diapositivas, los handouts, las guías, los ejemplos, los tutoriales.
3. **¡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.
   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.
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. **Documentá** a medida que desarrolles. No gastes tiempo en documentar funciones triviales, documenta las funciones o métodos y las clases más importantes. En el informe también **escribí claro**, en impersonal, y evitando opiniones.
6. No te compliques la vida con diseños complejos. **Cuanto más fácil sea tu diseño, mejor**.

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.
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.