PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN

IIC2333 — Sistemas Operativos y Redes — 1/2021

Proyecto 2

3 de Junio, 2021

Fecha de Entrega: 17 de Junio, 2021 hasta las 20:59
Fecha de ayudantía: 4 de Junio, 2021
Composición: grupos de 5 personas

Índice

1. Objetivos

Este proyecto requiere que como grupo implementen un protocolo de comunicación entre un servidor y múltiples clientes para coordinar un juego en línea. El proyecto debe ser programado en el **lenguaje C**. La comunicación entre las partes debe hacerse través de la funcionalidad de *sockets* de la API **POSIX**.

Requisitos fundamentales:

1. Los clientes de este juego deberán comportarse como *dumb-clients*. Esto significa que actúan únicamente de intermediarios entre el usuario y el servidor. Este último es quien se encargará de computar **toda** la lógica.
2. Debe existir obligatoriamente una interfaz por consola. En otras palabras, todos los efectos de las funcionalidades del juego deben verse reflejados en ella.

Nota: No se asignará el puntaje correspondiente para cada funcionalidad que no cumpla con estas restricciones.

2. Descripción: *Monster Hunter: Ruz*

Durante la pandemia el profesor Ruz no ha podido tener mucho contacto con sus amigos de la infancia, es por eso que decide crear un juego multijugador para poder pasar el rato con ellos, lamentablemente el tiempo del profesor es consumido por los deberes de la casa y su hijo, por eso le pide a sus alumnos expertos en Redes que creen el juego **Monster Hunter: Ruz** por él.

El juego en sí consiste de un sistema de peleas tipo ***turn-based RPG*** de hasta 4 jugadores, es decir, hasta 4 clientes conectados a un servidor central. Cada jugador controla a un personaje y en conjunto y por turnos, usan sus habilidades para combatir contra un monstruo.

2.1. Conexión inicial e inicio del juego (9 puntos)

- **Conexión inicial** - El servidor en ejecución deberá esperar a que los clientes o jugadores se conecten para empezar el juego. El servidor debe estar preparado para mantener hasta 4 jugadores conectados de forma simultánea. Cuando un cliente se conecte al servidor se le debe dar la bienvenida. El primer jugador en conectarse será designado como el **líder del grupo**.
- **Lobby** - Al conectarse, todos los jugadores ingresan a un lobby inicial en donde se les preguntará por su **nombre** y **clase** con la que deseen jugar (se puede repetir). El **líder del grupo** debe ser notificado sobre los nuevos jugadores que ingresen mostrando su nombre y clase elegida.
- **Inicio del juego** - El **líder del grupo** será el único con la opción de iniciar el juego. Al iniciar el juego se le preguntará contra qué monstruo desean combatir, pudiendo también optar por combatir contra un monstruo aleatorio [1]. Si alguno de los jugadores conectados todavía no ha ingresado su nombre y clase al intentar iniciar el juego, esto último no ocurrirá y se le notificará al **líder del grupo** que los jugadores no están listos.

2.2. Sistema de pelea por turnos (10 puntos)

Monster Hunter: Ruz consiste en combates de jugadores contra un monstruo controlado por el servidor. El combate se lleva a cabo a través de **rondas**, y dentro de cada una, cada participante juega 1 **turno**. Al iniciar el juego, el **líder del grupo** es el primero en jugar. Las **rondas** se desarrollan de la siguiente manera:

1. El **líder del grupo** realiza su turno: se despliega una interfaz en donde puede **observar la situación actualizada de los participantes e interactuar con un menú para elegir qué hacer**: elegir 1 habilidad y objetivo (un jugador aliado o el monstruo) para llevar a cabo su efecto, o bien rendirse.
2. El resto de los jugadores realizan su turno, uno a la vez y de la misma forma que el **líder del grupo**. Queda a criterio de ustedes determinar el orden de los turnos entre los jugadores.
3. El monstruo realiza su turno: al igual que los jugadores, elije 1 habilidad y objetivo y se lleva a cabo su efecto. El monstruo es controlado completamente por el servidor.

Cuando un jugador o el monstruo tenga puntos de vida igual a 0, inmediatamente es **derrotado** y no puede seguir participando en el combate. En el caso de un jugador, este **sigue conectado en el juego**.

En **ningún momento** un jugador o monstruo puede tener una cantidad de puntos de vida mayor a la cantidad inicial o menor a 0.

El juego continua de esta forma hasta que **el monstruo sea derrotado o no queden jugadores participando**, ya sea porque **todos fueron derrotados** o **se rindieron**. Un jugador puede **rendirse en cualquiera de sus turnos**. Al hacerlo, el jugador inmediatamente deja de participar en el combate. Queda a criterio de ustedes decidir si un jugador que se rinde queda como espectador o se desconecta completamente.

Cuando un combate termine **se le debe preguntar a los jugadores si quieren seguir jugando o no**. Los jugadores que opten por seguir jugando eligen nuevamente su clase y el **líder del grupo** elige contra que monstruo combatir. A los jugadores que opten por no seguir jugando se les debe desplegar un mensaje de despedida y son desconectados del juego. Si el **líder del grupo** opta por no seguir jugando, se debe elegir uno nuevo de forma aleatoria.

2.3. Clases (9 puntos)

Para poder batallar contra los monstruos del juego el profesor les pide implementar las distintas clases, las cuales tendrán habilidades y estadísticas únicas, las clases son las siguientes:

[1] Se puede utilizar la función rand() para generar aleatoridad

2.3.1. Cazador

- **Vida Inicial:** 5000 puntos de vida

Habilidades:

- **Estocada**: Realiza una estocada al monstruo, infligiéndole 1000 puntos de daño y dejando un sangrado que le quita 500 puntos de vida cada ronda, este efecto puede acumularse hasta 3 veces durante una batalla (un monstruo no puede tener mas de 3 sangrados).
- **Corte Cruzado**: Realiza un corte al monstruo infligiendo 3000 puntos de daño.
- **Distraer**: Distrae al monstruo, haciendo que este ataque al último Cazador en usar esta habilidad durante el próximo turno.

2.3.2. Médico

- **Vida Inicial:** 3000 puntos de vida

Habilidades:

- **Curar**: Eliges a un jugador para curarlo. El jugador elegido recupera 2000 puntos de vida.
- **Destello Regenerador**: Invocas un rayo de luz para infligir un número entero aleatorio entre 750 y 2000 de daño, ambos incluidos. Un jugador elegido al azar recupera la mitad del daño hecho redondeado hacia arriba.
- **Descarga Vital**: Liberas todo tu dolor para infligir 2 veces la diferencia entre tu vida máxima y tu vida actual como daño.

2.3.3. Hacker

- **Vida Inicial:** 2500 puntos de vida

Habilidades:

- **Inyección SQL**: Inyecta código y duplica el ataque de otro jugador por 2 turnos.
- **Ataque DDOS**: Realiza muchas peticiones de información al monstruo infligiendo 1500 puntos de daño.
- **Fuerza Bruta**: hace 0 daño. Si el jugador ocupa esta habilidad 3 veces (no necesariamente consecutivas) contra el monstruo hace 10000 de daño la última vez.

2.4. Monstruos (14 puntos)

Los amigos del profesor son personas exigentes que desean jugar un juego que tenga una gran cantidad de monstruos, por eso se deberán crear distintos tipos de monstruos.

Los monstruos tienen diferentes habilidades, la utilización de cada una de estas tiene una probabilidad de ocurrencia, esta se describe mas abajo para cada monstruo. La elección de dañar a un objetivo es completamente aleatoria, es decir, cada jugador tiene la misma probabilidad de ser atacado por el monstruo (a menos de que se diga lo contrario).

2.4.1. Great JagRuz

Tendrá las siguientes características:

- **Vida Inicial:** 10,000 puntos de vida.

Habilidades:

- **Ruzgar** (50 %): El JagRuz ocupa sus poderosas garRaz para atacar a un enemigo 1000 puntos de daño.
- **Coletazo** (50 %): El JagRuz golpea a todos los objetivos con su cola 500 puntos de daño.

2.4.2. Ruzalos

Tendrá las siguientes características:

- **Vida Inicial:** $20{,}000$ puntos de vida.

Habilidades:

- **Salto** (40 %): El Ruzalos salta sobre su objetivo, cayendo con gran fuerza sobre el, este recibe 1500 puntos de daño. Ruzalos no puede usar esta habilidad 2 o más veces seguidas, es decir, si usa esta habilidad, en su próximo turno no puede usarla.
- **Espina Venenosa** (60 %): El Ruzalos golpea al objetivo con la espina en su cola intoxicándolo, recibiendo 400 puntos de daño cada turno (dura 3 turnos este efecto). Si el objetivo ya está intoxicado, inflije 500 puntos de daño.

2.4.3. Ruiz, el Gemelo Malvado del Profesor Ruz

Tendrá las siguientes características:

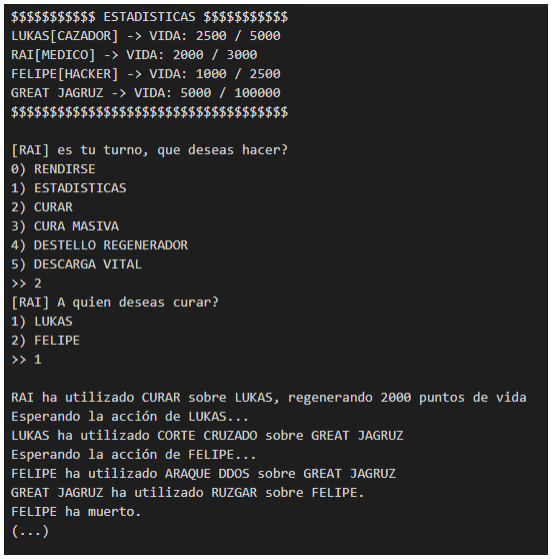
- **Vida Inicial:** $25{,}000$ puntos de vida.

Habilidades:

- **Caso de Copia** (40 %): Ruiz rompe el Código de Honor y copia de forma aleatoria una de las habilidades de algún jugador y la usa contra alguno de los jugadores causando los efectos asociados. Si la habilidad normalmente involucra ayudar a un jugador aliado, Ruiz recibe el efecto sobre si mismo.
- **Reprobatron-9000** (20 %): Ruiz **reprueba** instantáneamente a un jugador. El jugador queda con estado ***reprobado*** hasta el fin del siguiente turno de Ruiz. Un jugador con estado ***reprobado*** está desmoralizado y recibe $50\,\%$ de daño extra de todas las fuentes e inflige $50\,\%$ menos de puntos de vida con sus habilidades, ya sea al curar o dañar.
- **sudo rm -rf** (40 %): Ruiz borra todas las rondas ocurridas hasta ahora para infligir daño a todos los jugadores. Hace $100\cdot$(número de rondas desde el inicio del combate hasta ahora, sin considerar usos anteriores de esta habilidad) de daño a todos los jugadores.

2.5. Interfaz por consola (8 puntos)

Se espera que para este proyecto desarrollen una **interfaz por consola** para que los jugadores puedan interactuar con el juego. Se debe mostrar de forma clara menús, lo que ocurre en cada turno de los combates, mensajes de bienvenidas y otros elementos que estimen necesarios.



Nota: Esto es solo un ejemplo. Ustedes son libres de decidir cómo van a mostrar lo que ocurre durante el juego y toda la información que crean necesaria, siempre y cuando sea de forma **clara y ordenada**. Se debe dejar por escrito la forma de **utilizar su programa** en su archivo `README.md`.

3. Implementación

3.1. Cliente

El cliente debe tener algún tipo de interfaz en consola que permita visualizar el estado del juego, resultados de los turnos de cada participante de un combate, mensajes del servidor e ingresar los ***inputs*** necesarios. Siéntase libres de diseñar esto como quieran, siempre que se entienda el flujo del juego y se cumplan con los requisitos mínimos de este.

Otro punto importante, que fue mencionado anteriormente, es que el cliente debe comportarse como un ***dumb-client***, es decir, que solamente actúa de intermediario entre el usuario y el servidor y no es un participante activo de la lógica del programa.

3.2. Servidor

El servidor que ustedes implementarán debe mediar la comunicación entre los clientes. El servidor es el encargado de procesar **toda** la lógica del juego (recibir y procesar *inputs* de usuario, mantener el estado de cada participante de un combate, gestionar cada turno, controlar monstruos, evaluar si un combate ha terminado, etc.).

3.3. Protocolo de comunicación (7 puntos)

Todos los mensajes enviados, tanto de parte del servidor, como de parte del cliente, deberán seguir el siguiente formato:

- `ID` (1 *byte*): Indica el tipo de paquete.
- `PayloadSize` (1 *byte*): Corresponde al tamaño en *bytes* del `Payload` (entre 0 y 255).
- `Payload` (`PayloadSize` *bytes*): Es el mensaje propiamente tal. En caso de que no se requiera, el `PayloadSize` será `0` y el `Payload` estará vacío.

Tienen total libertad para implementar los paquetes que estimen necesarios. No obstante, deberán documentarlos todos en su `README.md`, explicitando el `ID` y el formato de `Payload`, junto con una breve descripción de cada uno.

A modo de ejemplo, consideren que queremos enviar el mensaje `Hola` con el `ID` 5. Si serializamos el `Payload` según **ASCII**, su tamaño correspondería a cuatro *bytes*. El paquete completo se vería así:

```
00000101 00000100 01001000 01101111 01101100 01100001
   5        4         H        o        l        a
```

3.4. Ejecución (3 puntos)

Los clientes y el servidor deberán ejecutarse de la siguiente manera, respectivamente:

```
$ ./server -i <ip_address> -p <tcp_port>
$ ./client -i <ip_address> -p <tcp_port>
```

Donde `<ip_address>` corresponde a la **dirección IP** del servidor (en formato *human-readable*, por ejemplo, `172.16.254.1`) y `<tcp_port>` al puerto TCP a través del cual el servidor recibirá nuevas conexiones.

El proyecto solo será corregido si cumple con esta modalidad de ejecución.

4. Bonus (¡hasta 15 décimas!)

4.1. Controlar Monstruo (+3 décimas)

Este bonus consiste en permitirle a un jugador controlar a un monstruo. Al crear la partida, el **líder del grupo** debe elegir si es que se podrá controlar o no al monstruo. Si se acepta, el máximo de jugadores del lobby sube a 5 y se debe agregar la posibilidad de elegir ser el monstruo al momento de elegir una clase y se debe poder elegir que monstruo será. Solo puede existir a lo más un jugador controlando al monstruo. Si nadie elije ser el monstruo antes de empezar el juego, este continua de forma normal (hasta 4 jugadores contra la computadora).

4.2. Recompensas (*Loot*) (+7 décimas)

En la raíz del servidor existirá un directorio con nombre `loot/`. Dentro de este directorio habrán imágenes que representarán las recompensas posibles por acabar exitosamente con un monstruo. Cuando un monstruo es vencido en combate, todos los jugadores que lograron sobrevivir recibirán de 3 a 5 piezas de **loot** elegidas de forma aleatoria en donde una pieza es representada por una imagen. No es necesario que todas las piezas conseguidas sean distintas entre sí.

El **loot** conseguido deberá guardarse en un directorio ubicado en la raíz del cliente llamada `loot_conseguido/`. Esta carpeta debe ser **generada durante la ejecución del programa** para que este bonus sea considerado.

Para lograr el envío de imágenes se definirá un paquete especial para enviar imágenes desde el servidor a los clientes llamado *Image*. Como los paquetes de envío de datos entre el servidor y los clientes tienen un `Payload` de un tamaño máximo de 255 *bytes*, una imagen deberá ser enviada a través de varios segmentos. La estructura de este nuevo paquete es el siguiente:

- `ID` (1 *byte*): Indica el tipo de paquete.
- `TotalPackages` (1 *byte*): Corresponde al número total de paquetes necesarios para enviar la imagen.
- `CurrentPackage` (1 *byte*): Es el índice del paquete que se está enviando (se parte desde 1).
- `PayloadSize` (1 *byte*): Corresponde al tamaño en *bytes* del `Payload` que se está enviando (entre 0 y 255).

- `Payload` (`PayloadSize` *bytes*): Es la información de la imagen propiamente tal.

Por ejemplo, supongamos que la imagen que quiero enviar pesa 2552 *bytes*. Corresponde enviar un total de 11 paquetes[2] ($255 \cdot 10 + 2 \cdot 1$). Además, supongamos que los últimos 2 *bytes* son `0x0A` y `0x9F`. Si el el `ID` del envío de imágenes es 64, entonces el último paquete en enviarse se vería así:

```
1000000 00001011 00001011 00000010 00001010 10011111
```

El cliente, al recibir el primer paquete de este tipo, tendrá que almacenar su información en un *buffer* y esperar a que el servidor le envíe todos los *payloads* pendientes. Que el `TotalPackages` sea igual al `CurrentPackage` significa que el servidor ha terminado de enviarle los datos de la imagen. Por lo tanto, el cliente va a poder guardar toda la información recibida en un solo archivo.

4.3. *SSH Tunneling* (+5 décimas)

Su programa debe poder funcionar de forma completamente distribuida. Esto es, con el servidor ejecutando en `iic2333.ing.puc.cl`, y clientes ejecutándose en lugares distintos (como sus domicilios, por ejemplo). El servidor `iic2333.ing.puc.cl` posee abierto solamente los puertos 22 y 80.

Este desafío deberá ser resuelto haciendo uso de **SSH Tunneling** para poder conectarse con un puerto local de `iic2333.ing.puc.cl`, un mecanismo cuyo funcionamiento deberán investigar por cuenta propia. Se recomienda revisar **este link** como punto de partida.

5. *README* y formalidades

5.1. Grupos

Está permitido que los grupos cambien respecto al proyecto anterior.

5.2. Entrega

Su proyecto debe encontrarse en un directorio con nombre `P2` dentro del directorio de uno de los integrantes del grupo en el servidor del curso. Además, deberán incluir:

- Un archivo `README.md` que indique quiénes son los autores del proyecto (**con sus respectivos números de alumno**), instrucciones para ejecutar y usar el programa, descripción de los paquetes utilizados en la comunicación entre cliente y servidor, cuáles fueron las principales decisiones de diseño para construir el programa, cuáles son las principales funciones del programa, qué supuestos adicionales ocuparon, y cualquier información que consideren necesaria para facilitar la corrección de su tarea. Se recomienda usar formato **Markdown**.
- Uno o dos archivos `Makefile` que compilen su programa en dos ejecutables llamados `server` y `client`, correspondientes al servidor y al cliente, respectivamente.

5.3. Otras consideraciones

- Este proyecto **debe** ser programado usando el lenguaje C. No se aceptarán desarrollos en otros lenguajes de programación.
- No respetar las formalidades o un código extremadamente desordenado podría originar descuentos adicionales. Se recomienda modularizar, utilizar funciones y ocupar nombres de variables explicativos.

[2] Ojo que comienza con el paquete de `ID = 1`

6. Descuentos (¡hasta 20 décimas!)

- 5 décimas por subir archivos binarios (programas compilados).
- 5 décimas por no incluir el/los `Makefiles`, o bien incluirlos y que no funcionen.
- 5 décimas por tener *memory leaks*. (Se recomienda fuertemente utilizar **Valgrind**).
- 5 décimas por la presencia archivos correspondientes a entregas del curso pasadas en la mismo directorio.

7. Nota final y atraso

La nota final del proyecto entregado a tiempo se calcula de la siguiente manera:

$$N = 1 + \frac{\sum_i p_i}{10} + b - d$$

Se puede hacer entrega del proyecto con un máximo de cuatro días de atraso. La fórmula a seguir es la siguiente:

$$N_{atraso} = \text{mín}(N; 7 + b - d - 0{,}75 \cdot a)$$

Siendo $\boldsymbol{d}$ el descuento total, $\boldsymbol{p_i}$ el puntaje obtenido en el ítem $\boldsymbol{i}$, $\boldsymbol{b}$ el bonus total, $\boldsymbol{a}$ la cantidad de días de atraso y $\textbf{mín}(\boldsymbol{x}; \boldsymbol{y})$ la función que retorna el valor mas pequeño entre $\boldsymbol{x}$ e $\boldsymbol{y}$.