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

IIC2333 — Sistemas Operativos y Redes — 1/2019

Proyecto 2

Lunes 27 de Mayo de 2019

Fecha de Entrega: Lunes 10 de Junio de 2019 a las 14:00 hrs
Composición: grupos de n personas, donde $3 \leq n \leq 4$
Fecha de ayudantía: Viernes 31-Mayo-2019 a las 10:00

Objetivo

Esta tarea requiere que usted implemente un protocolo de comunicación entre un servidor y uno o más clientes para coordinar un juego en línea. La tarea debe ser programada usando la API `POSIX` de *sockets* en el lenguaje C.

Descripción: *Damas Españolas*

Germy[1] se pasa todas las noches en vela jugando a su juego favorito: las Damas. Sin embargo, últimamente Germy[2] se ha sentido un poco triste, ya que ya no encuentra divertido jugar solo. En respuesta a esto, el grupo docente del curso ha decidido ayudar a Germy[3] para que vuelva a ser feliz. Por esto, les pide a los alumnos de Sistemas Operativos y **Redes** que desarrollen una plataforma en la que sea posible jugar con otra persona a través de una red LAN.

Además del juego, se pide implementar un *Chat* entre los contrincantes de manera que puedan discutir **amablemente** acerca del estado de la partida y felicitarse por las buenas jugadas. Es importante notar que este *Chat* se activa una vez que ha comenzado el juego.

Para evitar la trampa y verificar que las acciones se desarrollan en la secuencia correcta, se pide implementar un *Log* que registrará todas las acciones importantes antes y durante el juego. Las acciones que deberán ser registradas se detallan más adelante.

Este juego tendrá las siguientes reglas (algunas han sido modificadas respecto a las oficiales):

- Un juego consiste en jugar 1 partida. Luego de comunicar al ganador, se debe preguntar a los jugadores si quieren jugar otra partida. Si alguna respuesta es negativa, se debe terminar el juego.
- El tablero posee 64 casillas.
- Cada jugador dispone de 12 fichas de un mismo color situadas en las **casillas blancas** más próximas a éste. El objetivo del juego es capturar las fichas del oponente o acorralarlas para que no puedan realizar movimientos. Comienza el jugador con las fichas blancas.
- Se juega por turnos. En cada turno el jugador mueve una de sus fichas. Las fichas se mueven (cuando no capturan) una posición adelante en diagonal hacia una casilla blanca vacía.
- No existe la obligación de captura.

---

[1] (͡° ͜ʖ ͡°)

[2] ಠ_ಥ

[3] ʘ‿ʘ Más información sobre *kaomojis* en http://kaomoji.ru/

- Un jugador puede realizar capturas consecutivas.
- Si una ficha llega hasta el lado contrario del tablero, se convierte en dama. La dama se mueve también una posición en diagonal, pero hacia adelante y hacia atrás.
- Una partida finaliza cuando se da una de las siguientes situaciones:
  - Un jugador se queda sin piezas sobre el tablero, por lo tanto ha perdido.
  - Un jugador no puede mover cuando llega su turno, puesto que todas las fichas que le quedan están bloqueadas. Este jugador pierde.
  - Un jugador decide retirarse de la partida, o se produce una desconexión por parte de uno de los jugadores.
  - La partida también puede finalizar en empate si ambos jugadores quedan con un número igual y muy reducido de fichas y por muchos movimientos que se hagan no se resolvería la partida. Para efectos prácticos, se determinará automáticamente un empate cuando se efectúen 10 movimientos por cada jugador, y que el número de piezas de cada uno no haya variado durante dichos movimientos.

Funcionamiento del *Chat*

Cada vez que se inicia un nuevo turno de algún jugador se debe dar la opción de enviar un mensaje al otro cliente. El cliente podrá enviar solamente un mensaje en su turno. Se asumirá que el mensaje no será superior a 255 caracteres (o *bytes*).

Funcionamiento del sistema de *Logs*

Para mantener registro y poder saber qué acciones se han realizado desde que el servidor comenzó a funcionar es necesario generar una funcionalidad de *Log*. Todos los eventos deben ser registrados en conjunto con su respectivo *timestamp* en que ocurrieron (ver gettimeofday).

Cada paquete que se envía o se recibe debe ser registrado por el *Log*. En el caso de que esté presente el *flag* del *log* al momento de la ejecución del programa, el *log* deberá guardarse en un archivo de texto *log.txt*, en el mismo directorio del código, y deberá registrar el *timestamp* correspondiente, nombre del paquete y contenido recibido/enviado.

Parte I - Cliente de juego

Esta parte consiste en construir un cliente de juego para las Damas. El cliente debe contener algún tipo de interfaz (en terminal) que permita visualizar el tablero, el puntaje actual y los mensajes del *Chat* recibidos. No es necesario que todo se muestre al mismo tiempo, se espera que diseñe una interfaz tipo menú que permita navegar por todos los elementos relevantes del cliente. El cliente se comporta como un *dumb-client*, es decir, depende totalmente de la conexión con el servidor para el procesamiento de la lógica del juego.

Importante: Se debe implementar el protocolo estándar de esta tarea de tal modo que su cliente funcione comunicándose tanto con el servidor elaborado por usted como con uno elaborado por sus compañeros. La corrección de esta tarea va a incluir el conectarse con otras tareas. Si el servidor se desconecta repentinamente, el cliente debe ser capaz de controlar dicha conexión e informar al jugador sobre la desconexión.

Parte II - Servidor de juego

Para esta parte de la tarea, deberá implementar un servidor de juego que ofrezca la mediación de comunicación entre los clientes. El servidor es el encargado de procesar toda la lógica del juego (controlar si los jugadores realizan un movimiento válido, ver quién ganó, etc.) y enviársela correctamente a los clientes.

Debe implementar el protocolo estándar de esta tarea de tal modo que sea posible que, tanto clientes elaborados por

usted como por compañeros, puedan jugar sin inconvenientes. Si algún cliente interrumpe su conexión repentinamente, el servidor debe ser capaz de controlar dicha desconexión e informar al otro cliente conectado que ganó.

Protocolo

El protocolo que utilizará este juego considera una red mensajes que siguen un patrón estándar. Específicamente, un mensaje en este protocolo contiene:

- `MessageType ID` (1 byte): Corresponde al tipo de paquete que se está enviando.
- `Payload Size` (1 byte): Corresponde al tamaño en *bytes* de la información (*Payload*) que se va a enviar.
- `Payload` (variable): Corresponde a la información propiamente tal. Si algunos paquetes no necesitan enviar esta información, el *Payload Size* será 0 y el *Payload* no existirá.

Un mensaje tendrá distintos significados de acuerdo a su `ID`. El contenido y uso del paquete se determina de acuerdo a la siguiente lista:

1. *Start Connection*: Cliente envía este paquete al servidor para indicar el comienzo de la conexión.
2. *Connection Established*: Servidor responde con este paquete luego de recibir el *Start Connection* del cliente.
3. *Ask Nickname*: Servidor envía al cliente que se acaba de conectar este paquete para preguntarle el *nickname* (nombre).
4. *Return Nickname*: Cliente responde al servidor este paquete con su *nickname*.
5. *Opponent Found*: Servidor envía al cliente este paquete indicando que se encontró otro cliente para comenzar el enfrentamiento. El *Payload* de este paquete es el *nickname* del contrincante.
6. *Start Game*: Servidor envía al cliente la indicación de que comenzó correctamente la partida.
7. *Scores*: Servidor envía al cliente el *score* actual de los jugadores. El *Payload* será de 2 *bytes*: el primeros corresponde al puntaje del jugador al que le llega el paquete, y el otro es el puntaje del contrincante.
8. *Who's First*: Servidor envía al cliente en el *Payload* un número 1 o 2. Si se recibe un 1, este cliente es el que comienza jugando con las fichas blancas. Si recibe un 2, él espera la jugada del otro jugador. Este número será el `ID` de cada cliente que se usará en otros paquetes. La forma en que el servidor determina quién comienza es de manera aleatoria.
9. *BoardState*: Servidor envía el estado actual del tablero al cliente. El *Payload* constará de 64 caracteres de 1 *byte*, donde cada uno representa una casilla en el tablero. El significado de los caracteres es explicado más adelante. El tablero será codificado fila tras fila utilizando la vista del jugador de Blancas. La decodificación dependerá del jugador al que le llegue este paquete.
10. *Send Move*: Cliente envía al servidor la jugada realizada. El *Payload* se compone de 4 *bytes*, donde los 2 primeros indican las coordenadas de la casilla de la pieza a mover y los 2 restantes indican la casilla de destino de la pieza. Las coordenadas se especifican según la columna y la fila, como por ejemplo **C4**.
11. *Error Move*: El servidor envía este paquete al cliente informando que la jugada recibida no es correcta. Justo después de este paquete, el servidor envía nuevamente el paquete *Board State* al cliente.
12. *Ok Move*: El servidor envía este paquete informando que el movimiento del cliente fue exitoso.
13. *End Game*: El servidor envía este paquete informando que se terminó la partida.

14. *Game Winner/Loser*: El servidor envía este paquete informando del ganador y perdedor de la partida. El *Payload* consiste en 1 byte que indica el `ID` del cliente ganador, o 0 si hubo empate.

15. *Ask New Game*: Si un jugador ganó la partida, el servidor debe preguntarle a ambos clientes si es que desean jugar otra. En caso contrario, se debe pasar a terminar la conexión.

16. *Answer New Game*: El cliente responde al servidor, mediante un paquete cuyo *Payload* es un *boolean* de 1 *byte* indicando si es que desea jugar una nueva partida o no.

17. *Disconnect*: Servidor envía señal de desconexión a ambos clientes. Este paquete también lo puede enviar el cliente al servidor, indicando que el jugador desea desconectarse del juego.

18. *Error Bad Package*: Servidor debe enviar este error al cliente en caso de recibir un paquete con `ID` desconocido, no implementado o mal construido.

19. *Send Message*: El cliente le envía este paquete al servidor con el mensaje que desea enviarle a su rival.

20. *Spread Message*: El servidor envía este paquete al receptor del mensaje, junto con el mensaje correspondiente.

Tablas de identificadores

MessageType	**ID**
Start Connection	1
Connection Established	2
Ask Nickname	3
Return Nickname	4
Oponent found	5
Start Game	6
Scores	7
Who's First	8
BoardState	9
Send Move	10
Error Move	11
Ok Move	12
End Game	13
Game Winner/Loser	14
Ask New Game	15
Answer New Game	16
Disconnect	17
Error Bad Package	18
Send Message	19
Spread Message	20
Bonus *Image*	64

Cuadro 1: Identificadores de mensajes.

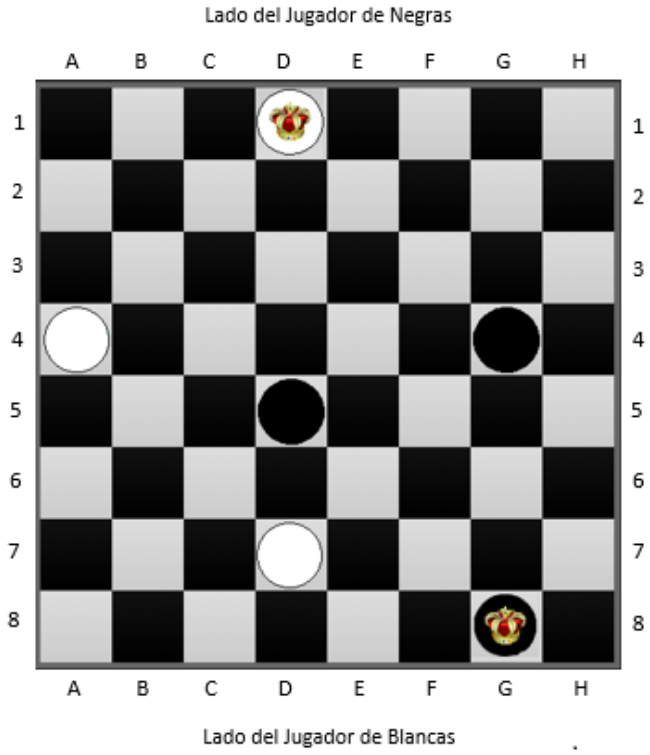
Caracter	Significado
’b’	Casilla blanca vacía
’n’	Casilla negra vacía
’o’	Peón blanco
’O’	Dama blanca
’x’	Peón negro
’X’	Dama negra

Cuadro 2: Identificadores de piezas y espacios del tablero

Nota: Tanto su cliente como su servidor no se pueden caer por recibir paquetes mal formados o de funciones que no implementen. En caso que no implementen alguna función, al menos deben ser capaz de manejar la recepción del paquete y tomar alguna acción. Las caídas de su programa debido a mal manejo originarán descuentos.

Formato de envío

Veamos un ejemplo: Supongamos que en una partida en curso, el servidor quiere enviarle el paquete *BoardState* a los clientes, para mostrarles el estado actual del siguiente tablero:



La información que contendrá el *Payload* de este paquete indicará qué tipo de ficha hay en cada casilla del tablero. De no haber una ficha, se indicará el color de la celda. De esta forma, para el tablero anteriormente ilustrado, el paquete *BoardState* tendría un largo total de 66 bytes y estaría compuesto de la siguiente forma (los bytes han sido interpretados como enteros o caracteres, según corresponda):

- `MessageType ID`: 9

- `Payload Size`: 64
- `Payload`:

$$\underbrace{nbnOnbnb}_{\text{Primera fila}}\underbrace{bnbnbnbn}_{\text{Segunda fila}}\underbrace{nbnbnbnb}_{\text{Tercera fila}}\underbrace{onbnbnxn}_{\text{Cuarta fila}}\underbrace{nbnxnbnb}_{\text{Quinta fila}}\underbrace{bnbnbnbn}_{\text{Sexta fila}}\underbrace{nbnonbnb}_{\text{Séptima fila}}\underbrace{bnbnbnXn}_{\text{Octava fila}}$$

El cliente, al recibir este paquete, primero interpretará el primer *byte* como entero para saber qué mensaje del protocolo recibió. Luego, interpretará el segundo *byte* como entero, que es el *Payload Size* para saber cuántos *bytes* le faltan por leer. Y, por último, interpretará el *Payload* como **un arreglo de caracteres**. Este procedimiento es equivalente a los otros paquetes del protocolo, y es de suma importancia que se respete, ya que de esa forma se asegurará que puedan interactuar distintos programas.

Formato de Ejecución

Tanto el servidor como el cliente deben ejecutarse de la siguiente manera:

```
$ ./server -i <ip_address> -p <tcp-port> -l
$ ./client -i <ip_address> -p <tcp-port> -l
```

Donde:

- `-i <ip-address>` es la dirección IP que va a ocupar el servidor para iniciar, o la dirección IP en la cual el cliente se va a conectar.
- `-p <tcp-port>` es el puerto TCP donde se establecerán las conexiones.
- `-l` *flag* opcional que indicará, si es que está presente, que se guarde el *log* en un archivo *log.txt*.

Para que el cliente se conecte correctamente a la IP y puerto, es necesario que primero se inicie el servidor. Por ejemplo:

```
$ ./server -i 127.0.0.1 -p 8888
```

El servidor iniciará la conexión con esos parámetros. Luego, el cliente tendrá que usar el mismo IP y puerto para establecer la conexión:

```
$ ./client -i 127.0.0.1 -p 8888
```

Bonus: Envío de Imágenes (+5 décimas)

Se define un paquete especial para mandar imágenes desde el servidor a los clientes, llamado *Image*. Como cada paquete está definido en base a un tamaño máximo de 255 *bytes* de *Payload*, enviar una imagen de ese tamaño sería muy pequeña. Por lo tanto, la imagen tendrá que enviarse a través de varios *Payloads*. La estructura de este nuevo paquete es la siguiente:

- `MessageType ID` (8 bits): `ID` del paquete (que es 64).
- `Total Payloads` (8 bits): La cantidad de *Payloads* totales usados para enviar la imagen.
- `Current Payload` (8 bits): `ID` del *Payload* que se está enviando actualmente.
- `Payload Size` (8 bits): Tamaño en *bytes* de la información de la imagen.
- `Payload` (variable): Información de la imagen enviada.

El cliente, al recibir el primer paquete de este tipo, tendrá que almacenar el *Payload* en una variable y esperar que el servidor le envíe todos los *Payloads* pendientes. Cuando el `Total Payloads` sea igual al `Current Payload`, signifca que el servidor envió el total de la imagen. Por lo tanto, el cliente va a poder almacenar toda la información recolectada en un archivo *image.jpg* en el mismo directorio que el código.

README y Formalidades

Deberá incluir un archivo `README` que indique quiénes son los autores del proyecto (**con sus respectivos números de alumno**), instrucciones para ejecutar el programa, 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 considere necesarias para facilitar la corrección de su tarea. Se sugiere utilizar formato **markdown**.

Para entregar el proyecto, si el grupo es el mismo que el proyecto 1 (SO), pueden utilizar el mismo repositorio. En el caso de no haber ocupado GitHub Classroom para su repositorio, o el grupo no es el mismo, se debe asegurar que los ayudantes de Redes estén compartidos en el nuevo repositorio de su entrega (usuarios de GitHub `lukassr` y `nivek0o0`). **Solo debe incluir código fuente** necesario para compilar su tarea, además del `README` y un `Makefile`[4]. Con respecto a los archivos correspondientes a la entrega del Proyecto 1 (SO), estos deben estar en una *branch* aparte, para así dejar en *master* solo la entrega del proyecto de Redes. **NO debe incluir archivos binarios (será penalizado)**. Se revisará el contenido del repositorio el día Lunes 10 de Junio de 2019 a las 14:00 hrs.

Este proyecto **debe** ser programado en C. No se aceptarán desarrollos en otros lenguajes de programación.

El no respeto de las formalidades o un código extremadamente desordenado podría originar descuentos adicionales. Se recomienda modularizar, utilizar funciones y ocupar nombres de variables explicativos. En el caso de no entregar en la carpeta especificada la tarea **no** se corregirá.

Evaluación de Funcionalidades

La corrección y evaluación de este proyecto se hará conectando el programa con una pauta previamente programada. Por lo tanto, se recomienda fuertemente que **antes de la fecha de entrega** se intente conectar la aplicación con otras. Así, se podrán descartar detectar con antelación posibles errores de protocolo.

Cada funcionalidad se evaluará en una escala de logro de 4 valores, ponderado según la siguiente información:

- 10 % Inicio y término conexión.
- 5 % *Nickname* de jugadores.
- 15 % Logística de movimientos.
- 40 % Implementación del protocolo (envío y recepción de paquetes).
- 15 % Implementación del *Log*.
- 10 % *README* y formalidades. Esto incluye cumplir las normas de la sección formalidades.
- 5 %. Manejo de memoria. *Valgrind* reporta en su código 0 *leaks* y 0 errores de memoria, considerando que los programas funcionen correctamente.
- **Bonus:** +5 décimas

Descuentos

- Descuento de 0.5 puntos por subir **archivos binarios** (programas compilados).
- Descuento de 1 punto si la entrega **no tiene un** `Makefile`**, no compila o no funciona** (*segmentation fault*).
- Descuento de 1 punto si la entrega (*branch master*) **tiene archivos del Proyecto 1 (SO)**.

[4] Puede incluir, además, imágenes de ejemplo si es que el grupo desarrolla el bonus

Política de atraso

Se puede hacer entrega de la tarea con un máximo de 4 días de atraso. La fórmula a seguir es la siguiente:

$$N_{P_2} = 1{,}0 + \sum_i p_i + b$$

$$N_{P_2}^{\text{Atraso}} = \text{mín}(N_{P_2}, 7{,}0 - 0{,}75 \cdot d + b)$$

Siendo N_{P_2} la nota obtenida, p_i el puntaje obtenido del ítem i, b el puntaje asignado a través del bonus, d la cantidad de días de atraso y $\text{mín}(x, y)$ la función que retorna el valor mas pequeño entre x e y.

Coevaluación

La nota final de este proyecto estará ponderada según una coevaluación, que se realizará luego del día Lunes 10 de Junio de 2019 a las 14:00 hrs. Los criterios y ponderaciones de la coevaluación serán comunicados por parte del Cuerpo Docente a través de los medios oficiales del curso.

Preguntas

Cualquier duda preguntar a través del foro.