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

IIC2333 — Sistemas Operativos y Redes — 2/2018

Proyecto 2

Jueves 15 de Noviembre de 2018

Fecha de Entrega: 2 de Diciembre de 2018 a las 23:59hrs
Fecha de corrección: Semana 3-7 Diciembre de 2018 de forma presencial
Composición: grupos de n personas, donde $3 \leq n \leq 4$

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: *Tic-Tac-Toe*

Germy[1] se pasa todas las noches en vela jugando a su juego favorito: el Tic-Tac-Toe (o Gato, como lo llamaremos de ahora en adelante). 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, le piden 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. Es importante notar que este *Chat* se activa una vez que se ha iniciado 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.
Se deben tomar en cuenta las siguientes consideraciones:

1. Un juego considera un conjunto de 3 partidas de Gato distintas.
2. Se gana un juego considerando el mejor de 3 partidas.
3. Se entiende como una partida el resultado del juego correspondiente a un tablero de Gato.
4. El servidor elige aleatoriamente quién comienza cada una de las partidas.

Funcionamiento del *Chat*

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

---

[1] (͡° ͜ʖ ͡°)
[2] ಠ_ಥ
[3] ʘ‿ʘ

Funcionamiento del sistema de *Logs*

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

Cada paquete que se envía o que se recibe debe ser registrado por el *Log*. Este registro debe imprimirse en consola informando al usuario el nombre de los paquetes enviados y recibidos. 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, junto con 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 el Gato. 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 logística 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. 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 logística del juego (controlar si los jugadores realizan un movimiento valido, 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 binarios que siguen un patrón estándar. Específicamente, un mensaje en este protocolo contiene:

- `MessageType ID` (8 bits): Corresponde al tipo de paquete que se está enviando.
- `Payload Size` (8 bits): Corresponde al tamaño en *bytes* de la información (*Payload*) que se va a enviar.
- `Payload` (variable): Corresponde la información propiamente tal. Si en algunos paquetes no sale definido qué tipo de *Payload* enviar, o si el paquete en sí no necesita enviar *Payload*, el *Payload* debe tener valor *0*.

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 a cliente este paquete para preguntarle el *nickname* (nombre) del cliente que se acaba de conectar.

4. *Return Nickname*: Cliente responde a servidor este paquete con el *nickname* del cliente.

5. *Opponent Found*: Servidor envía a cliente este paquete indicando que se encontró otro cliente para comenzar el juego. El *Payload* de este paquete es el *nickname* del contrincante.

6. *Game Start*: Servidor envía a cliente ña indicación de que comenzó correctamente el juego.

7. *Start Round*: Servidor envía a cliente indicación de que comenzó una nueva ronda del juego. El *Payload* de este paquete es de 8 *bits* y representa el número del *round*.

8. *Scores*: Servidor envía a cliente el *score* actual de los jugadores. El *Payload* será de 16 *bits*: los primeros 8 *bits* son el puntaje del jugador al que le llega el paquete, y los 8 restantes son el puntaje del contrincante.

9. *Who's First*: Servidor envía a cliente en el *Payload* un 1 o un 2. Si es 1, el cliente es el que juega primero. Si es 2, él espera la jugada del otro cliente. 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. El jugador que parte es el que ocupará las cruces.

10. *BoardState*: Servidor envía el estado actual del tablero al cliente. El *Payload* constará de 9 números de 8 *bits*, donde cada número representa un espacio en el tablero. Si el número es 0, significa que el espacio correspondiente está vacío. Si el número es 1, significa que hay una cruz en el espacio. Si el número es 2, significa que hay un círculo en el espacio **(revisar sección formato de envío)**.

11. *BoardUpdate*: Cliente envía al servidor el tablero actualizado con la jugada realizada.

12. *Ask New Game*: Si un jugador alcanzó las 3 partidas ganadas se debe preguntar a ambos participantes si es que desean jugar un nuevo juego. En caso contrario, se debe pasar a terminar el juego.

13. *Answer New Game*: El cliente responde con este paquete al servidor un *boolean* de 8 *bits* indicando si es que el cliente quiere una nueva partida o no.

14. *Error Board*: El servidor envía este paquete al cliente informando que la jugada recibida no es correcta. Justo después de este *package*, el servidor envía nuevamente el paquete *Board State* al cliente.

15. *Ok Board*: El servidor envía este paquete informando que el movimiento del cliente fue exitoso.

16. *End Round*: El servidor envía este paquete informando que se terminó la ronda actual. El *Payload* de este paquete es de 8 *bits* y representa el número del *round* finalizado.

17. *Round Winner/Loser*: El servidor envía este paquete informando del ganador y perdedor del *round*. El *Payload* se compondrá de la siguiente manera: los 8 primeros *bits* representarán el número del *round* actual; los 8 *bits* siguientes representarán un *boolean* indicando empate (0 si no hay empate y 1 si es que lo hay); si es que no hay empate, los últimos 8 *bits* representarán el `ID` del cliente ganador.

18. *End game*: Servidor indica el fin del juego para ambos jugadores.

19. *Game Winner/Loser*: Luego de terminar el juego, el servidor debe enviar el estado final del juego indicando quién gano. El *Payload* será un número de 8 *bits* indicando el `ID` del cliente ganador.

20. *Disconnect*: Servidor envía señal de desconexión a ambos clientes.

21. *Error Not Implemented*: Servidor debe enviar este error a ambos clientes en caso de recibir un paquete con `ID` desconocido o no implementado.

22. *Send Message*: El cliente le envía este paquete al servidor con el mensaje que desea enviar al otro cliente.

23. *Receive Message*: El servidor envía este paquete con el mensaje de un cliente al otro.

Tablas de identificadores

MessageType	**ID**
Start Connection	1
Connection Established	2
Ask Nickname	3
Return Nickname	4
Oponent found	5
Game Start	6
Start Round	7
Scores	8
Who's First	9
BoardState	10
BoardUpdate	11
Ask New Game	12
Answer New Game	13
Error Board	14
Ok Board	15
End Round	16
Round Winner/Loser	17
End Game	18
Game Winner/Loser	19
Disconnect	20
Error Not Implemented	21
Send Message	22
Receive Message	23
Bonus *Image*	64

Cuadro 1: Identificadores de mensajes.

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 debe 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:

X 1	2	O 3
4	X 5	6
O 7	8	O 9

Los números del tablero indican la posición de cada casilla. Usando la definición y el `ID` de *BoardState*, sabemos que el paquete contendrá:

- `MessageType ID`: 10
- `Payload Size`: 9
- `Payload`:

- Casilla 1: 1
- Casilla 2: 0
- Casilla 3: 2
- Casilla 4: 0
- Casilla 5: 1
- Casilla 6: 0
- Casilla 7: 2
- Casilla 8: 0
- Casilla 9: 2

Por lo tanto, el *Payload* tendrá el siguiente formato:

$$102010202$$

Luego, cada **número** presentado anteriormente tiene que ser pasado a su representación binaria:

$$\underbrace{1}_{00000001}\ \underbrace{0}_{00000000}\ \underbrace{2}_{00000010}\ \underbrace{0}_{00000000}\ \underbrace{1}_{00000001}\ \underbrace{0}_{00000000}\ \underbrace{2}_{00000010}\ \underbrace{0}_{00000000}\ \underbrace{2}_{00000010}$$

Finalmente, el servidor arma el paquete a enviar al cliente, asignándole el `ID` *10* y con *Payload Size* de *9 bytes* (ya que son 9 números, cada uno de 1 bit):

$$\underbrace{00001010}_{\textit{ID: 10}}\ \underbrace{00001001}_{\textit{Payload Size: 9}}\ \underbrace{000000010000000000000010000000000000001000000000000010000000000000010}_{\textit{Payload: 102010202}}$$

El cliente, al recibir este paquete, primero interpretará el primer *byte* para saber qué paquete recibió. Luego, interpretará el segundo *byte*, que es el *Payload Size* para saber el tamaño del *Payload*. Y, por último, interpetará el *Payload*. Sería incorrecto que el cliente interprete el *Payload* como un solo número (si seguimos el ejemplo, el cliente interpretaría *18447307027958071298*), sino que tiene que ser capaz de dividir los bits de tal forma de interpretar correctamente el mensaje. 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.

Evaluación

A diferencia de otras tareas, la evaluación de esta tarea se realizará de forma **presencial** y en conjunto con otros grupos. El día de la evaluación se conectaran dos grupos a la misma red LAN configurada por el equipo docente y procederán a probar las funcionalidades de su programa en modalidad de demo. Es por esto que es fundamental que todo el protocolo sea implementado exactamente como lo descrito arriba y, en caso de no tener funcionalidades, indicarlo adecuadamente y no dejando que el programa termine abruptamente por algún tipo de interrupción.

README y Formalidades

Deberá incluir un archivo `README` que indique quiénes son los autores del proyecto (**con sus respectivos números de alumno**), cuáles fueron las principales decisiones de diseño para construir el 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 usted deberá utilizar el repositorio de equipo creado con GitHub Classroom. **Solo debe incluir código fuente** necesario para compilar su tarea, además del `README` y un `Makefile`[4]. **NO debe incluir archivos binarios (será penalizado)**. Se revisará el contenido del repositorio el día 2 de Diciembre de 2018 a las 23:59hrs.

- Puede ser realizada en forma individual, o en grupos de $3 \leq n \leq 4$ personas. En cualquier caso, recuerde indicar en el `README` los autores de la tarea con sus respectivos números de alumno.

Evaluación de Funcionalidades

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.
- 20 % Logística de movimientos.
- 30 % Correcto envío y recepción de mensajes.

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

- 20 % 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

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.

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*)**.

Preguntas

Cualquier duda preguntar a través del foro.