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

IIC2333 — Sistemas Operativos y Redes — 2/2019

Proyecto 2

Viernes 11 de Octubre de 2019

Fecha de Corrección Presencial: Miércoles 30-Octubre-2019 a las 08:30 hrs
Composición: grupos de n personas, donde $3 \leq n \leq 4$
Fecha de ayudantía: Viernes 11-Octubre-2019 a las 08:30

Objetivo

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

Descripción: *Dobble*

Germy[1] ha traído el nuevo juego sensación al DCC; ¡se trata de *Dobble*! Este divertido juego de cartas ha generado gran adicción (y horas de procastinación) entre el cuerpo docente del DCC.

Sin embargo, últimamente Germy[2] se ha sentido un poco triste, ya que le gustaría continuar jugando con sus amigos desde la comodidad de su casa. 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.

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.

Reglas del Juego

Este juego tendrá las siguientes reglas:

- Una partida consiste en jugar 5 rondas. Luego de comunicar el ganador de la partida, se debe preguntar a los jugadores si quieren jugar otra partida. Si alguna respuesta es negativa, se debe terminar el juego.
- Una ronda consiste en adivinar la palabra repetida en 2 tarjetas, donde cada tarjeta contiene 10 palabras distintas y solo 1 de ellas es la correcta. Ambos jugadores reciben el mismo par de tarjetas y el puntaje de cada ronda dependerá del número de intentos según la siguiente tabla:

Adivinó al	Puntaje
Primer intento	5
Segundo intento	3
Tercer intento	1

Cuadro 1: Tabla de puntajes.

[1] (͡° ͜ʖ ͡°)
[2] ಠ_ಥ
[3] ʘ‿ʘ Más información sobre *kaomojis* en `http://kaomoji.ru/`

- Cada jugador dispone de 3 intentos para adivinar la palabra repetida. Luego de la tercera equivocación, el cliente no recibirá un nuevo *Send Cards*, por lo que el humano no podrá ingresar nuevas palabras. El servidor enviará el paquete *Round Winner/Loser* cuando ambos clientes hayan adivinado, ambos se hayan quedado sin intentos o una combinación de esos estados.

Detalles de Implementación

- Cada tarjeta será una grilla cuadrada de 20x20. Las 10 palabras de una tarjeta serán colocadas fila por medio, empezando desde la primera fila. La posición dentro de la fila será dictada por el servidor (explicada más adelante).
- Las palabras siempre serán manejadas y desplegadas en mayúsculas, aunque usted no debe preocuparse de esto ya que las palabras vienen bien formateadas en el archivo **.txt** que lee el servidor (y que será entregado).
- Los espacios de la grilla que no contengan palabras deben ser llenados con algún caracter que no dificulte la lectura de las palabras, como un guión.
- Las tarjetas deben mostrarse de forma horizontal, es decir, una al lado de la otra, para que quepan en una terminal y faciliten visualmente el juego.
- El servidor debe construir 2 conjuntos de palabras (uno por tarjeta) para cada ronda. Para ello dispone de una lista de 500 palabras en un archivo `CSV`. Los pasos que debería seguir son:
  1. Elegir al azar 19 palabras distintas.
  2. De estas, elegir al azar 1 que será la repetida. Así, se pueden formar los 2 conjuntos.
  3. La palabra repetida debe estar en distinta posición en los conjuntos, para dificultar que los jugadores la encuentren.
  4. Luego, hay que escoger al azar la posición que tendrá cada palabra dentro de la fila. Es importante que esta posición garantice que la palabra no se salga de la tarjeta. La posición corresponde al índice donde empieza la palabra, por lo que su mínimo valor es 0 y máximo 19.

Funcionamiento del sistema de *Logs*

Para mantener registro y poder saber qué acciones se han realizado desde que el servidor y el cliente comenzaron 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).

En el caso de que esté presente el *flag* del *log* al momento de la ejecución del programa, se deberá registrar cada paquete que se envía o se recibe en el cliente y en el servidor. El *log* debe ser escrito en un archivo de texto *log.txt* en el mismo directorio del código, y cada línea deberá registrar el *timestamp* correspondiente, si el paquete es IN/OUT, el nombre del paquete y su contenido incluyendo ID, Size y Payload.

Parte I - Cliente de juego

Esta parte consiste en construir un cliente del juego *Dobble*. El cliente debe tener algún tipo de interfaz en terminal que permita visualizar las cartas, enviar la respuesta, ver el puntaje actual o abandonar la partida (*desconectarse*). No es necesario que todo se muestre al mismo tiempo, se espera que diseñe una interfaz tipo menú navegable que permita realizar las funcionalidades antes mencionadas. El cliente se debe comportar 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 pueda comunicarse con el servidor elaborado por usted, el de sus compañeros o de los ayudantes. Esto es importante porque la corrección

incluye el poder conectarse con otras tareas. Si el servidor se desconecta repentinamente, el cliente debe ser capaz de controlar dicho evento e informar al jugador sobre la desconexión.

Parte II - Servidor de juego

Para esta parte de la tarea deberá implementar un servidor del juego que ofrezca la mediación de comunicación entre los clientes. El servidor es el encargado de procesar toda la lógica del juego (escoger las palabras, generar las tarjetas, enviarlas a los clientes y recibir las respuestas, evaluar quién ganó, entre otras funcionalidades).

Importante: 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 con su servidor. 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 de comunicación

El protocolo que utilizará este juego considera un conjunto de mensajes que sigue un patrón estándar. Específicamente, un mensaje en este protocolo contiene:

- `MessageType ID` (1 byte, *integer*): Indica el tipo de paquete que se está enviando.
- `Payload Size` (1 byte, *unsigned integer*): Corresponde al tamaño en *bytes* de la información (*Payload*) que se está enviando. Dado que solo se tiene 1 byte, éste será interpretado como entero sin signo, por lo que su máximo valor será de 255.
- `Payload` (PayloadSize, *chars*): 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 que desea jugar una partida.
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 su *nickname*.
4. *Return Nickname*: Cliente responde al servidor este paquete con su *nickname*. Se aconseja incluir el caracter terminal para facilitar la impresión del *string*.
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. *Send IDs*: Servidor envía al cliente 1 *byte* con su identificador personal, el que será utilizado en otros mensajes para identificar al cliente. Este ID debe ser numérico, por ejemplo, 1 ó 2, (**no puede ser 0**).
7. *Start Game*: Servidor envía al cliente la indicación de que comenzó correctamente la partida. El *Payload* corresponde a 1 byte con el número de la partida.
8. *Scores*: Servidor envía al cliente el *score* actual de los jugadores. El *Payload* será de 2 *bytes*: el primero corresponde al puntaje del jugador al que le llega el paquete, y el otro es el puntaje del contrincante.
9. *Send Cards*: Servidor envía el contenido de las 2 tarjetas al cliente. El *Payload* consiste en **20 secuencias** consecutivas de bytes, donde cada secuencia tiene:

- 1 byte para el largo de la palabra;
- la palabra, un byte por letra; y
- 1 byte para la posición en la fila donde debe ir la palabra.

Por ejemplo, si las palabras fueran **GATO**, **PEZ** y **PERRO**, el contenido del paquete completo sería:

[9, 18, 4, 'G', 'A', 'T', 'O', 0, 3, 'P', 'E', 'Z', 17, 5, 'P', 'E', 'R', 'R', 'O', 9]

Donde el ID del paquete es 9, el largo del *Payload* es 18 y la palabra **GATO** debería mostrarse a la izquierda de la tarjeta (posición 0), **PEZ** a la derecha (posición 17) y **PERRO** desde la mitad (posición 9).

10. *Send Word*: Cliente envía al servidor su respuesta a la ronda. El *Payload* se compone de tantos *bytes* como letras tenga la palabra.

11. *Response Word*: El servidor envía este paquete al cliente informando de la correctitud de la palabra enviada y el número de intentos restantes. El primer byte del *Payload* será un 1 si la palabra es correcta o un 0 si no lo es. El segundo byte corresponde a los intentos restantes y varía entre 2 y 0. Si la respuesta fue incorrecta, el servidor envía nuevamente el paquete *Send Cards* al cliente.

12. *Round Winner/Loser*: El servidor envía este paquete informando del ganador y perdedor de la ronda. El *Payload* consiste en 1 byte que indica el `ID` del cliente ganador o un 0 si es que hubo empate de puntaje.

13. *End Game*: El servidor envía este paquete informando que se terminó la partida. El *Payload* corresponde a 1 byte con el número de 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.

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* (1 o 0) 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.

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

Tabla Resumen de Identificadores

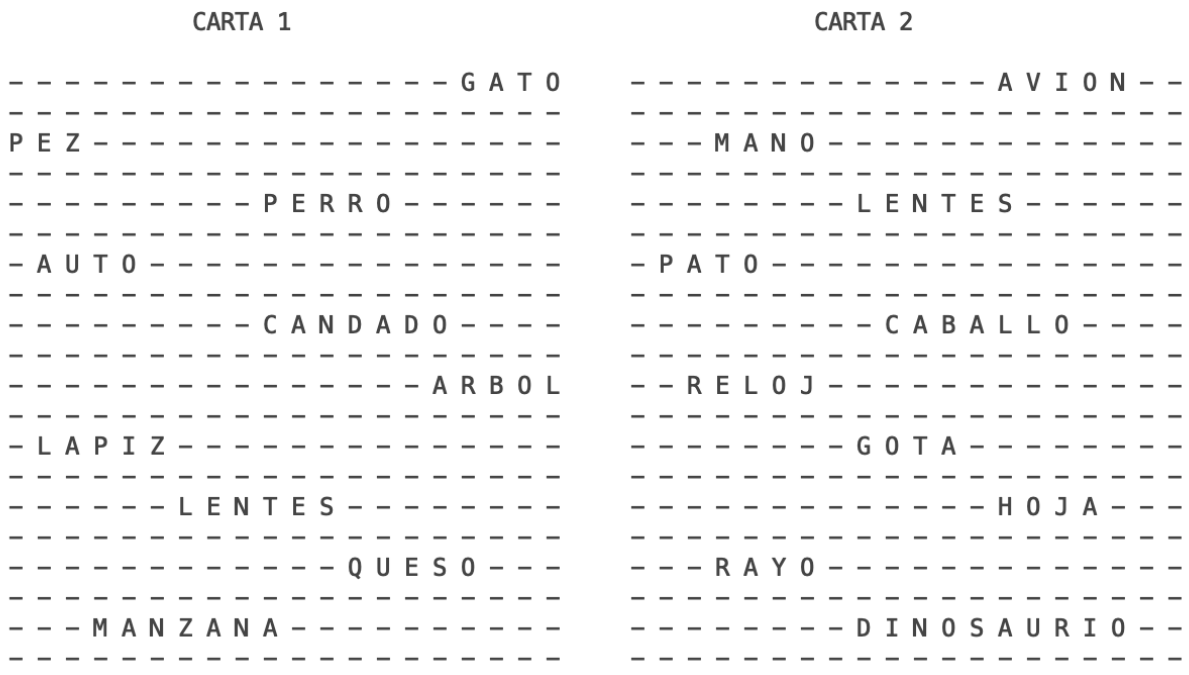
MessageType	**ID**
Start Connection	1
Connection Established	2
Ask Nickname	3
Return Nickname	4
Opponent Found	5
Send IDs	6
Start Game	7
Scores	8
Send Cards	9
Send Word	10
Response Word	11
Round Winner/Loser	12
End Game	13
Game Winner/Loser	14
Ask New Game	15
Answer New Game	16
Disconnect	17
Error Bad Package	20
Bonus *Image*	64

Cuadro 2: 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 deben ser capaces de manejar la recepción del paquete y hacer nada. Las caídas de su programa debido a mal manejo originarán descuentos.

Ejemplo Formato de Envío

Supongamos que en una partida en curso, el servidor quiere enviarle el paquete *SendCards* a los clientes, para mostrarles la pareja actual de cartas con las que deben jugar:



La información que contendrá el *Payload* de este paquete indicará cuales son las palabras que se deben asignar a cada carta y en que posición deben ser colocadas. Cada palabra debe presentarse en filas, dejando una fila libre intermedia entre dos palabras. Por ejemplo, supongamos que la información para posicionar nuestra primera palabra es **4GATO16**, en este caso, deberíamos ubicar la palabra en la primera fila de la grilla, partiendo desde la posición 16 (considerando que se ha partido del índice 0). Las posiciones que no contienen palabras deben ser rellenadas con un guión '-'. De esta forma, para el tablero anteriormente ilustrado, el paquete *SendCards* tendría un largo total de 146 bytes y estaría compuesto de la siguiente forma (los bytes han sido interpretados como enteros o caracteres, según corresponda):

- `MessageType ID`: 9
- `Payload Size`: 144
- `Payload`:

$\underbrace{4GATO16}_{\text{Primera fila}}\ \underbrace{3PEZ0}_{\text{Tercera fila}}\ \underbrace{5PERRO9}_{\text{Quinta fila}}\ \underbrace{4AUTO1}_{\text{Séptima fila}}\ \underbrace{7CANDADO9}_{\text{Novena fila}}\ \underbrace{5ARBOL15}_{\text{Undécima fila}}\ \underbrace{5LAPIZ2}_{\text{Decimotercera fila}}\ \underbrace{6LENTES6}_{\text{Decimoquinta fila}}\ \underbrace{5QUESO12}_{\text{Decimoséptima fila}}\ \underbrace{7MANZANA3}_{\text{Decimonovena fila}}$

$\underbrace{5AVION13}_{\text{Primera fila}}\ \underbrace{4MANO3}_{\text{Tercera fila}}\ \underbrace{6LENTES8}_{\text{Quinta fila}}\ \underbrace{4PATO1}_{\text{Séptima fila}}\ \underbrace{7CABALLO9}_{\text{Novena fila}}\ \underbrace{5RELOJ2}_{\text{Undécima fila}}\ \underbrace{4GOTA8}_{\text{Decimotercera fila}}\ \underbrace{4HOJA13}_{\text{Decimoquinta fila}}\ \underbrace{4RAYO3}_{\text{Decimoséptima fila}}\ \underbrace{10DINOSAURIO8}_{\text{Decimonovena fila}}$

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. Por último, interpretará el *Payload* como **un arreglo de caracteres y enteros**. 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 el servidor escuchará por nuevas conexiones.
- `-l` *flag* opcional que indicará, si es que está presente, que se guarde el *log* en un archivo *log.txt*.

Los parámetros no son posicionales, por lo que pueden venir en distinto orden. La utilidad del *flag* es indicar que justo después viene un determinado parámetro.

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

```
$ ./server -p 3000 -l -i 127.0.0.1
```

El servidor iniciará la conexión con esos parámetros (127.0.0.1 es equivalente a *localhost*). Luego, el cliente tendrá que usar el mismo IP y puerto para establecer la conexión:

```
$ ./client -l -i 127.0.0.1 -p 3000
```

Bonus 1: 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*, las imágenes que se podrían enviar de ese tamaño serían muy pequeñas. Por lo tanto, la imagen tendrá que enviarse a través de varios **Segmentos**. La estructura de este nuevo paquete es la siguiente:

- `MessageType ID` (1 byte): `ID` del paquete (que es 64).
- `Total Payloads` (1 byte): La cantidad total de *Payloads* que serán enviados.
- `Current Payload` (1 byte): N° del *Payload* que se está enviando actualmente (se parte desde 1).

- `Payload Size` (1 byte): 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 un **buffer** 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.png* en el mismo directorio que el código.

Las imágenes se almacenarán en un directorio llamado `assets` en la raíz del servidor. Los nombres de las imágenes serán `winner.png` y `loser.png` y deberán ser envidas correctamente al cliente ganador y perdedor.

Bonus 2: Conexión entre más de 2 clientes (+3 décimas, solo si nota final $\geq 5,5$)

Inicialmente el juego se definió entre 2 jugadores, pero la naturaleza del juego permite extenderlo a múltiples clientes alterando ligeramente la implementación, sin modificaciones en la lógica del juego.

Por esa razón, si usted logra que puedan jugar más de 2 clientes, hasta un tope de 10, obtendrá esta bonificación en su nota final. Es importante que se mantenga la interactividad durante el juego de cada uno de los múltiples clientes.

README y Formalidades

Los grupos

Está permitido que los grupos cambien respecto al proyecto anterior, por lo que **todos deberán inscribir** sus grupos en el siguiente Formulario. Basta 1 respuesta por grupo.

La Entrega

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**. Además, deberá subir su proyecto completo a una carpeta P2 en su servidor, para dejar un respaldo de su trabajo en caso de correcciones posteriores. **Solo debe incluir código fuente** necesario para compilar su tarea, además del `README` y un `Makefile`. **NO debe incluir archivos binarios (será penalizado)**.

La Corrección

La corrección se realizará de forma **presencial** durante la mañana del día de la entrega, en hora y lugar a informar.

Otros

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 consistirá en 2 casos de uso:

- Un servidor y cliente del grupo jugando con un cliente de los ayudantes.
- Un servidor y cliente de los ayudantes jugando con un cliente del grupo.

Por lo tanto, se recomienda fuertemente que **antes de la corrección** se prueben esto casos de uso para detectar con antelación posibles errores de protocolo.

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

- 5 % Inicio y término conexión.
- 5 % *Nickname* de jugadores.
- 15 % Logística del juego.
- 5 % Sistema de puntaje.
- 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*).

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:

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

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

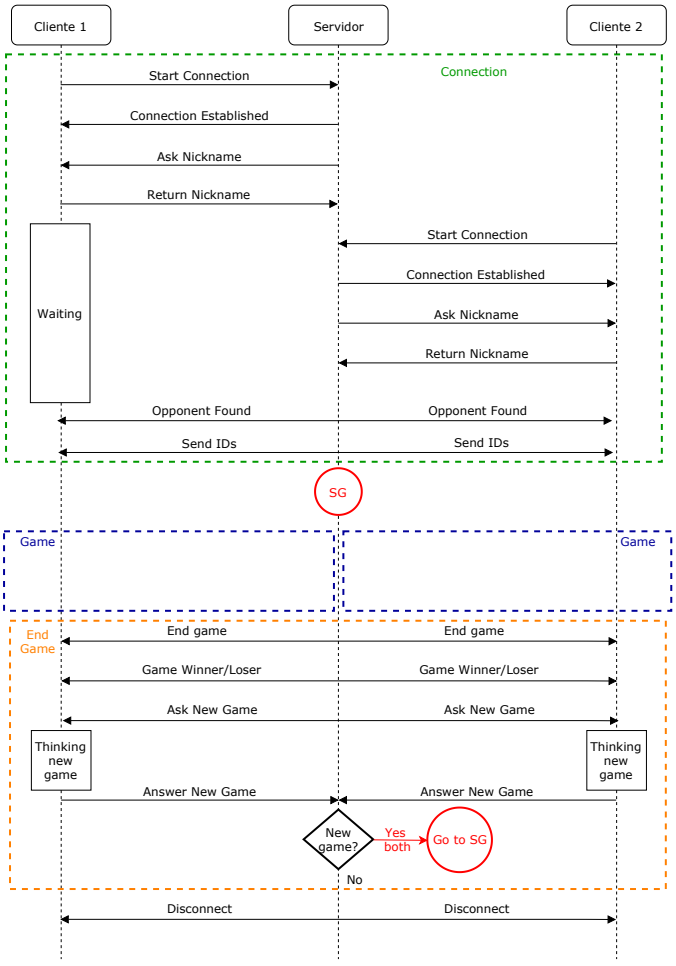
Siendo N_{P_2} la nota obtenida, p_i el puntaje obtenido del ítem i, b el puntaje asignado a través del bonus, d la cantidad de días de atraso y $\text{mín}(x, y)$ la función que retorna el valor mas pequeño entre x e y.

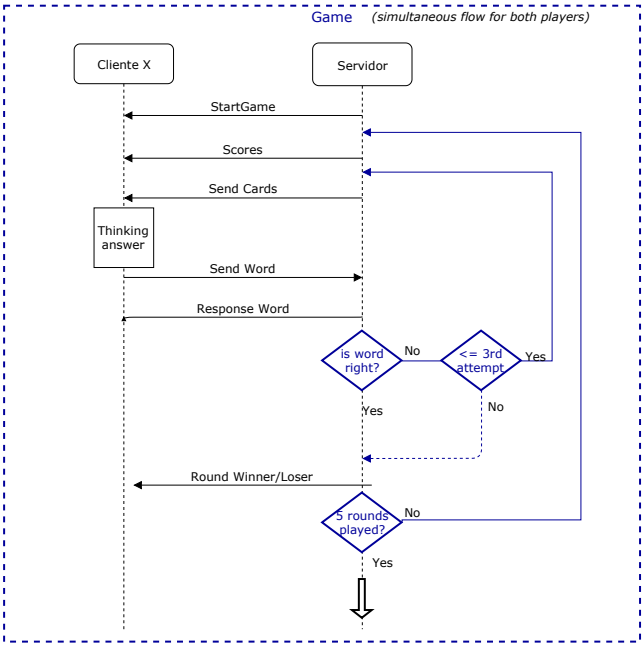
Coevaluación

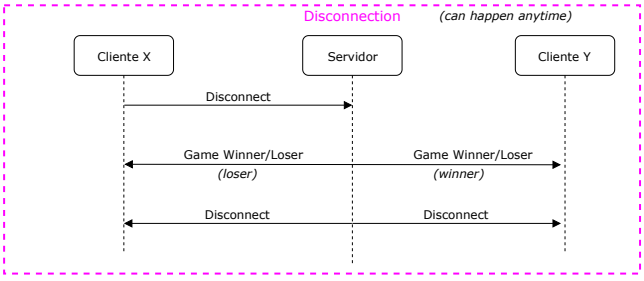
La nota final de este proyecto estará ponderada según una coevaluación, que se realizará luego de la corrección presencial del día Miércoles 30-Octubre-2019 a las 08:30 hrs. Los criterios y ponderaciones de la coevaluación serán comunicados por parte del Cuerpo Docente a través de los medios oficiales del curso.

Preguntas

Cualquier duda preguntar a través del foro.


Cliente 1
Servidor
Cliente 2
Connection
Start Connection
Connection Established
Ask Nickname
Return Nickname
Waiting
Start Connection
Connection Established
Ask Nickname
Return Nickname
Opponent Found
Opponent Found
Send IDs
Send IDs
SG
Game
Game
End Game
End game
End game
Game Winner/Loser
Game Winner/Loser
Ask New Game
Ask New Game
Thinking new game
Thinking new game
Answer New Game
Answer New Game
New game?
Yes both
Go to SG
No
Disconnect
Disconnect


Game
(simultaneous flow for both players)
Cliente X
Servidor
StartGame
Scores
Send Cards
Thinking answer
Send Word
Response Word
is word right?
No
<= 3rd attempt
Yes
Yes
No
Round Winner/Loser
5 rounds played?
No
Yes


Disconnection
(can happen anytime)
Cliente X
Servidor
Cliente Y
Disconnect
Game Winner/Loser
(loser)
Game Winner/Loser
(winner)
Disconnect
Disconnect

Text