Programación de Sistemas Distribuidos

Curso 2020/2021

Práctica 1

Diseño e implementación de un sistema de juegos on-line

Esta práctica consiste en el diseño e implementación de un sistema de juegos on-line. Concretamente, es necesario desarrollar tanto la parte cliente, como el servidor, para permitir partidas remotas del juego *Black Jack*.

Al juego *Black Jack* se juega con una baraja de cartas francesa, es decir, la que está formada por 52 cartas de 4 palos: corazones, tréboles, picas y diamantes. Para esta práctica vamos a desarrollar una **versión simplificada** de este juego. Recordad que lo importante no es el propio juego en sí, sino la comunicación entre las aplicaciones cliente (jugadores) y el servidor.

En esta versión del juego deben participar dos jugadores:, a los que llamaremos *jugA* y *jugB*. Inicialmente, tanto *jugA* como *jugB* parten con una cantidad establecida de fichas. Por turnos (suponemos que comienza *jugA*) deben realizar una apuesta (en fichas). Seguidamente, se reparten 2 cartas a cada jugador. Las cartas del 1 (o as) al 10, tienen el mismo valor que su número. Las figuras valen 10 puntos. El objetivo es obtener la puntuación más cercana a 21, sin pasarse. Una vez repartidas las dos cartas iniciales, se juega por turnos. El jugador "activo" (en este caso, *jugA*) podrá elegir obtener carta o plantarse. Si el jugador se pasa de 21 puntos, finaliza su turno. Una vez finalice el turno del jugador activo, el turno pasa al otro jugador. Cuando finaliza el turno de los dos jugadores, se comprueban los puntos obtenidos. Si ambos jugadores se pasan de 21 o consiguen la misma puntuación, se considera empate. Si únicamente un jugador se pasa de 21, pierde y gana el otro jugador. En otro caso, gana el jugador que obtenga una puntuación más próxima a 21. Una vez finalice el juego actual, se barajan las cartas y comienza una nueva mano, cambiando los turnos, de forma que en la mano actual comenzará *jugB*. El juego se da por finalizado cuando uno de los jugadores se queda sin fichas. En este caso, el jugador que tenga las fichas ganará la partida.

Para el desarrollo de esta práctica **NO será necesario implementar la lógica del juego**. El objetivo de la misma se centra en la parte encargada de las comunicaciones entre los clientes (jugadores) y el servidor. Por ello, se proporciona un código fuente inicial con la lógica del juego y funcionalidades auxiliares que facilitan el desarrollo de la práctica.

1. Estructura general del sistema

La estructura general del sistema a desarrollar cuenta con dos partes: la parte cliente y la parte servidor. En esta práctica, ambas partes se desarrollarán en **C utilizando sockets**.

La parte cliente será el programa que ejecuten los jugadores. Esta parte **no se encarga, en ningún momento, de controlar la lógica del juego**. Su función es mostrar por pantalla el estado del juego actual, esto es, el estado de la partida. Además, recogerá las acciones introducidas por el jugador. La parte servidor controlará la lógica del juego (p.e. controlar que un jugador se pasa de 21 puntos). Además, se encargará de controlar los turnos de los jugadores involucrados en la partida.

Para que pueda dar comienzo una partida, será necesario que 2 jugadores estén conectados al servidor. De lo contrario, la partida no podrá comenzar. El siguiente esquema muestra la arquitectura del juego.

En la comunicación entre cliente y servidor podemos definir 3 elementos que deberán transmitirse entre estas partes.

- **Número (unsigned int)**: Este número puede representar un código, los puntos del jugador o su *stack* (número de fichas). Todos estos elementos en la práctica se representan con un `unsigned int`. Los códigos se utilizan para que tanto los jugadores como el servidor puedan conocer el estado en el que se desarrolla la partida. Por ejemplo, un jugador - al recibir un código - puede saber si es su turno para tomar una acción, esperar a que realice una acción o conocer quién ha ganado al finalizar la partida. Estos códigos están definidos en el fichero `utils.h` y se describirán en detalle posteriormente.

- **Mensaje**: Cadena de texto que contiene mensajes sobre el estado de la partida. Este mensaje se enviará en dos partes. La primera consiste en un número de 4 bytes (`unsigned int`) que contiene la longitud (en número de caracteres) del mensaje a enviar. La segunda parte consiste en el propio mensaje (`tString`), el cual está formado por una secuencia de caracteres. Los mensajes se utilizan para enviar los nombres desde la parte cliente al servidor.

- **Deck**: Mazo de cartas. Se utiliza para conocer la jugada de los jugadores. De esta forma, el jugador activo puede conocer su puntuación durante la partida y jugador rival puede saber qué puntuación ha obtenido su oponente. Un d*eck* se representa con la siguiente estructura: un array de 52 cartas - representadas con números `unsigned int` - y un número entero que indica el número de cartas existentes en el *deck*. Además, se utiliza como baraja de juego, donde se obtendrán las cartas que se reparten a los jugadores.

```
typedef struct{
	unsigned int cards [DECK_SIZE];
	unsigned int numCards;
}tDeck;
```

1.1 Estructura de una partida

La estructura de una partida se define a continuación. Consideramos que *jugA* representa el jugador que inició la comunicación con el servidor en primer lugar y *jugB* al jugador que inició la comunicación con el servidor en segundo lugar.

1. El programa servidor deberá esperar la conexión de dos clientes (jugadores). Una vez los jugadores realicen la conexión con el servidor, dará comienzo una partida.
2. El servidor recibirá el nombre de *jugA* y, seguidamente, el nombre de *jugB*.
3. El servidor inicializará una sesión para que de comienzo la partida. Para ello, se puede utilizar la función `initSession`. Adicionalmente, se puede hacer uso de la función `printSession` para depurar el programa en el servidor.
4. Mientras no finalice la partida:
    a. El servidor enviará, a *jugA*, el código `TURN_BET` y su *stack*.
    b. El jugador *jugA* introducirá por teclado una apuesta, que se enviará al servidor. Para ello, puede utilizar la función `readBet`. Seguidamente, esperará la respuesta del servidor, que podrán ser los códigos:
        i. `TURN_BET_OK`, si la apuesta es correcta.
        ii. `TURN_BET` si la apuesta no es correcta. Con lo cual, se repetirá el punto 4.b
    c. Seguidamente, se repetirá el paso 4.b para *jugB*.

d. En este punto, donde ambos jugadores ya han recibido el código `TURN_BET_OK`, comienza la jugada por parte de los jugadores. Inicialmente, el jugador activo – en este caso, *jugA* – realizará las acciones, mientras que el jugador pasivo – en este caso, *jugB* – podrá ver la jugada de *jugA*. Así, el servidor enviará:
    i. A *jugA*, `TURN_PLAY`, sus puntos de la jugada actual y su *deck*.
    ii. A *jugB*, `TURN_PLAY_WAIT`, los puntos de *jugA* y el *deck* actual de *jugA*.
    iii. La opción tomada por *jugA* (*stand* o *hit*) se envía al servidor, de forma que el jugador envía `TURN_PLAY_STAND` si decide "plantarse" y no pedir más cartas, o `TURN_PLAY_HIT` si desea pedir una carta nueva. Para leer esta acción se puede hacer uso de la función `readOption`.
        1. En caso de pedir una nueva carta, el servidor enviará un código, los puntos y el nuevo *deck* con la carta nueva. El código puede ser `TURN_PLAY,` si el jugador tiene opción de seguir jugando, o `TURN_PLAY_OUT,` si el jugador ha superado los 21 puntos. Esta misma información debe enviarse a *jugB* de la misma forma que se hizo en el punto 4.d.ii, mientras *jugA* tenga posibilidad de seguir jugando.
        2. Si *jugA* se planta - por lo cual ha enviado el código `TURN_PLAY_STAND` al servidor, o recibe el código `TURN_PLAY_OUT`, el servidor enviará:
            a. El código `TURN_PLAY_WAIT` a *jugA* para indicar que ahora pasa a ser un jugador pasivo, de forma que se comportará a partir de este momento como tal.
            b. El código `TURN_PLAY_RIVAL_DONE` a *jugB* para indicar que su rival ha finalizado.
    iv. Una vez *jugB* tenga el turno, se realizarán los mismos pasos del punto 4.d, de forma que ahora *jugB* pasa a ser el jugador activo y *jugA*, el jugador pasivo.
    v. En este punto, se actualizarán las fichas de cada jugador. Se puede hacer uso de la función updateStacks.
    vi. Para acabar la mano, se debe comprobar si hay algún ganador:
        1. Si el *stack* de *jugA* es 0, se manda a este jugador el código `TURN_GAME_LOSE`. A *jugB* se manda el código `TURN_GAME_WIN`. El juego finaliza y se deben cerrar los sockets correspondientes, tanto en el cliente como en el servidor.
        2. Si el *stack* de *jugB* es 0, se manda a este jugador el código `TURN_GAME_LOSE`. A *jugA* se manda el código `TURN_GAME_WIN`. El juego finaliza y se deben cerrar los sockets correspondientes, tanto en el cliente como en el servidor.
        3. En otro caso, se cambia el turno de los jugadores utilizando la función `getNextPlayer` y se repite el punto 4.

Consideraciones de implementación:

- Los *stacks,* apuestas, puntos y códigos se representan con `unsigned int`, tanto en el cliente como en el servidor.
- Una apuesta se considera correcta si es menor que el *stack* del jugador que realiza la apuesta y menor que la apuesta máxima permitida (constante `MAX_BET`). No se puede realizar esta comprobación en el cliente.

1.2 Funciones y estructuras de datos para la lógica del juego

El fichero `game.h` contiene las cabeceras de las funciones encargadas de la lógica del juego. Además, este fichero contiene una descripción detallada de los parámetros de entrada y salida de cada función. Seguidamente, se describen estas funciones:

```
void initDeck (tDeck *deck);
```

Inicializa un *deck* (Mazo). Básicamente, "coloca" todas las cartas en su posición. Debe usarse, antes de jugar cada mano, para inicializar el *gameDeck*.

```
void clearDeck (tDeck *deck);
```

Vacía un *deck*. Debe utilizarse, antes de jugar cada mano, en los decks de los jugadores.

```
void printSession (tSession *session);
```

Imprime por pantalla la información de una sesión. Puede utilizarse para depurar el comportamiento del servidor.

```
void initSession (tSession *session);
```

Inicializa una sesión. Debe utilizarse al principio de la partida.

```
unsigned int getRandomCard (tDeck* deck);
```

Obtiene una carta aleatoriamente de un *deck*. Se Utiliza cuando un jugador pide una carta (*hit*). El *deck* se actualiza, quitando la carta obtenida, la cual es devuelta por la función como parámetro de retorno.

```
unsigned int calculatePoints (tDeck *deck);
```

Calcula los puntos de un *deck*. Puede utilizarse para calcular los puntos de un jugador durante la partida.

```
void updateStacks (tSession *session);
```

Actualiza los *stacks* de ambos jugadores. Debe llamarse cuando ambos jugadores han realizado sus jugadas, al final de cada mano.

Las constantes `MAX_BET`, `INITIAL_STACK` y `GOAL_GAME` representan la apuesta máxima permitida, el *stack* inicial de cada jugador al iniciar la partida y la puntuación máxima permitida en una mano para poder ganarla, respectivamente.

La estructura donde se almacenará la información de la partida se muestra a continuación:

```
typedef struct{

	// Data for player 1
	tString player1Name;
	tDeck player1Deck;
	unsigned int player1Stack;
	unsigned int player1Bet;

	// Data for player 2
	tString player2Name;
	tDeck player2Deck;
	unsigned int player2Stack;
	unsigned int player2Bet;

	// Deck for the current game
	tDeck gameDeck;
}tSession;
```

2. Implementación del servidor

El servidor se ejecutará recibiendo como parámetro, únicamente, el puerto donde realizará la escucha de las conexiones de los clientes. Seguidamente, establecerá conexión con dos clientes (jugadores), asignando un socket a cada una de estas conexiones. A partir de ese momento, el servidor ya podrá comunicarse con los dos jugadores.

El fichero `serverGame.h` contendrá las cabeceras de las funciones que deberán implementarse en el fichero `serverGame.c`. Generalmente, el objetivo de estas funciones es simplificar el código de la parte servidor. Por ejemplo, funciones que se encargan de enviar/recibir un mensaje a/del jugador. Puesto que los mensajes se transmiten muy a menudo con los jugadores, estas funciones ayudan a reducir el código fuente y aumentan la claridad del programa.

El servidor deberá ser capaz de mantener varias partidas simultáneamente. Para ello, será necesario el uso de *threads*. La estructura `tThreadArgs`, definida en el fichero `serverGame.h`, puede utilizarse para gestionar la comunicación de cada partida.

3. Implementación del cliente

La parte cliente se ejecuta recibiendo dos parámetros: la IP del servidor y el puerto donde éste permanece escuchando conexiones de los jugadores.

Cada cliente realizará una única conexión con el servidor, la cual se deberá cerrar una vez finalice la partida. Además, el cliente no controlará en ningún momento el estado de la partida. De esta forma, la lógica del juego se gestionará completamente en el servidor. Por ejemplo, aspectos tales como comprobar si una apuesta es correcta, o si el jugador se ha pasado de 21 puntos, se controlarán en el servidor.

El fichero `clientGame.h` contendrá las cabeceras de las funciones utilizadas por la parte cliente. El fichero `clientGame.c` deberá contener la implementación de estas funciones. Por ejemplo, las funciones `readBet` *y* `readOption` se proporcionan ya implementadas, las cuales se encargan de leer la apuesta y la acción del usuario, respectivamente.

4. Ficheros a entregar

Los ficheros necesarios para la realización de esta práctica se encuentran en el fichero `PSD_Prac1_Sockets.zip`. Para desarrollar la práctica se deberán modificar, **únicamente,** los ficheros `clientGame.h` , `clientGame.c`, `serverGame.h` y `serverGame.c`. Se deberá entregar, además, un fichero llamado `autores.txt` que contenga el nombre completo de los integrantes del grupo.

La entrega de esta práctica se llevará a cabo mediante un **único fichero comprimido** en formato `zip`. Es importante matizar que la práctica entregada debe contener los ficheros necesarios para realizar la compilación, tanto del cliente como del servidor.

En caso de que cualquiera de las partes entregadas no compile, se tendrá en cuenta la penalización correspondiente.

NO se permite incluir nuevos ficheros para el desarrollo de este apartado.

5. Plazo de entrega

La práctica debe entregarse a través del Campus Virtual **antes del día 12 de Noviembre de 2020, a las 18:00 horas**.

No se recogerá ninguna práctica que no haya sido enviada a través de la plataforma indicada o esté entregada fuera del plazo indicado.