Programación de Sistemas Distribuidos

Curso 2025/2026

Práctica 1

Diseño e implementación del juego Black Jack en un sistema distribuido

Implementación con sockets

Esta práctica consiste en el diseño e implementación del juego *Black Jack* en un entorno distribuido. Para ello es necesario desarrollar, tanto la parte cliente, como el servidor, de forma que puedan llevarse a cabo partidas remotas multijugador. 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 (*hearts*), tréboles (*clubs*), picas (*spades*) y diamantes (*diamonds*). 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 predefinida de fichas. Por turnos (suponemos que comienza *jugA*) deben realizar una apuesta (en fichas). Seguidamente, se reparten dos cartas a cada jugador. Las cartas del 1 (A) al 10 (Ten), tienen el mismo valor que su número. Todas las figuras valen 10 puntos (J's, Q's y K's). 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 pedir carta (*hit*) o plantarse (*stand*). 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, el servidor comprueba quién ha ganado y se reparten los puntos como corresponda. Si ambos jugadores se pasan de 21 o consiguen la misma puntuación, se considera empate y se devuelven las fichas apostadas a cada jugador. Si únicamente un jugador se pasa de 21, pierde, y gana el otro jugador. El jugador que gana obtiene todas las fichas en juego. Una vez finalice el juego actual, se barajan las cartas y comienza una nueva mano cambiando los turnos, de forma que – en este caso - 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.

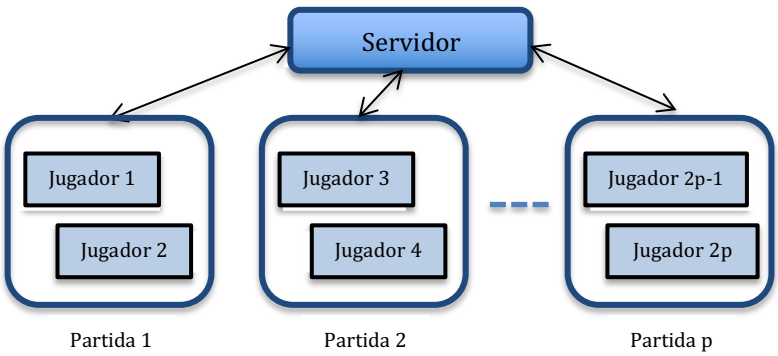
Para el desarrollo de esta práctica **NO será necesario implementar la lógica del juego**. El objetivo de la misma se centra en la parte encargada de las comunicaciones entre los clientes (jugadores) y el servidor. Por ello, se proporciona un código fuente inicial con la lógica del juego y subprogramas 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 de controlar la lógica del juego**. Su función es mostrar por pantalla el estado del juego actual, es decir, el estado de la partida. Además, recogerá las acciones introducidas por el jugador. La parte servidor controlará la lógica del juego (p.ej. controlar si 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 dos jugadores estén conectados al servidor. De lo contrario, la partida no podrá comenzar. Una vez se conecten dos jugadores, dará comienzo la partida y el servidor seguirá escuchando nuevas peticiones, de forma que los dos siguientes jugadores comenzarán una nueva partida, y así, sucesivamente. El siguiente esquema muestra la arquitectura del juego.



En la comunicación entre cliente y servidor podemos definir tres 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 clientes 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 llevar a cabo una acción, esperar a que realice una acción el jugador rival, o conocer quién ha ganado al finalizar la partida. Estos códigos están definidos en el fichero `game.h` y se describirán en detalle posteriormente.

- **Mensaje**: Esencialmente es una cadena de caracteres. 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, entre otras cosas. De esta forma, el jugador activo puede conocer su puntuación durante la partida, y el 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. Su definición se encuentra en el fichero `game.h`.

```
/** Length for tString */
#define STRING_LENGTH 128

/** Type for names, messages and this kind of variables */
typedef char tString [STRING_LENGTH];

/** Structure that represents a deck */
typedef struct{
        unsigned int cards [DECK_SIZE];
        unsigned int numCards;
}tDeck;
```

Las cartas de la baraja se representarán con los números enteros entre el 0 y el 51. Dado que en la bajara existen 4 palos, esto es, 13 cartas de cada palo, utilizaremos los números del 0 al 12 para representar los tréboles (*clubs*), del 13 al 25 para representar las picas (*spades*), del 26 al 38 para representar los diamantes (*diamonds*) y del 39 al 51 para representar los corazones (*hearts*). Además, estos números representan las cartas en orden creciente, es decir, empezando por el as (A) y terminando por el rey (King). Así, el número 0 representa AC (as de tréboles), el número 1 representa 2C (dos de tréboles), el número 12 representa KC (rey de tréboles), y el número 15 representa 3S (tres de picas).

Para calcular el valor y palo de una carta dado su número, se pueden utilizar las funciones – que ya se facilitan implementadas – **`suitToChar`** y **`cardNumberToChar`**.

1.1 Estructura de una partida

La estructura de una partida se define a continuación. Suponemos que se han conectado al servidor dos jugadores de forma satisfactoria (sin errores): *jugA* y *jugB*. Además, 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 dé comienzo la partida. Para ello, se puede utilizar el subprograma `initSession`. Adicionalmente, se puede hacer uso del subprograma `printSession` para depurar el programa en el servidor como se considere oportuno.
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 acción de *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 – ya que 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.
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 o igual que el *stack* del jugador que realiza la apuesta, menor o igual que la apuesta máxima permitida (constante `MAX_BET`), y consta, al menos, de una ficha. No se puede realizar esta comprobación en el cliente.

1.2 Subprogramas y estructuras de datos para la lógica del juego

El fichero `game.h` contiene las cabeceras de algunos de los subprogramas encargados de la lógica del juego, los cuales pueden utilizarse tanto en la parte cliente, como en la parte servidor. Además, este fichero contiene una descripción detallada de los parámetros de entrada y salida de cada subprograma. El siguiente código muestra una porción del fichero donde se definen varias constantes. Las constantes con prefijo `TURN` representan los códigos intercambiados entre cliente y servidor para sincronizar las distintas acciones de una partida, tales como realizar jugada, esperar a que termine el turno del jugador rival, o comunicar el ganador/perdedor de la partida, entre otras. La constante `MAX_BET` representa la apuesta máxima permitida, en fichas. Las constantes `DECK_SIZE` y `SUIT_SIZE` representan, respectivamente, el número de cartas totales y el número de cartas relativas a un mismo palo. Finalmente, podéis utilizar las constantes `TRUE` y `FALSE` como literales *boolean* (verdadero y falso).

```
#define TURN_XXX …

/** Maximum bet */
#define MAX_BET 5

/** Deck's size */
#define DECK_SIZE 52

/** Number of suits in the deck */
#define SUIT_SIZE 13

/** True value */
#define TRUE 1

/** False value */
#define FALSE 0
```

Seguidamente, se describen los subprogramas auxiliares, ya implementados en el código facilitado:

`void showError` `(const char *msg);`

Muestra por pantalla el error detectado y detiene la ejecución del programa.

`void showCode` `(unsigned int code);`

Muestra el texto descriptivo del código `code`.

`char suitToChar` `(unsigned int number);`

Función que calcula el palo (*suit*) de una carta a partir del número que la representa.

`char cardNumberToChar` `(unsigned int number);`

Función que calcula el valor de una carta a partir del número que la representa.

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

Subprograma que imprime un mazo de cartas (*deck*) utilizando la representación numérica para mostrar las cartas.

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

Subprograma que muestra un mazo de cartas (*deck*) en modo gráfico ASCII.

`unsigned int min` `(unsigned int a, unsigned int b);`

Devuelve el menor de los números `a` y `b`.

2. Implementación del servidor

El fichero `serverGame.h` deberá contener las cabeceras de los subprogramas incluidos en la parte servidor. Por ejemplo, subprogramas que se encargan de enviar/recibir un mensaje a/del jugador. Puesto que los mensajes se transmiten muy a menudo entre los jugadores, estos subprogramas ayudan a reducir el código fuente y aumentan la claridad del programa.

La siguiente porción del fichero `serverGame.h` muestra las constantes definidas. Las dos primeras, `SERVER_DEBUG` y `DEBUG_PRINT_GAMEDECK`, se utilizan para depurar la parte servidor, de forma que la primera se utiliza para mostrar mensajes generales que representen el estado de la ejecución del programa, mientras que la segunda se utiliza para mostrar el mazo de cartas en cada acción de las partidas.

Las constantes `INITIAL_STACK` y `GOAL_GAME` hacen referencia a las fichas con las que empieza cada jugador y la puntuación máxima permitida en una mano para poder ganarla, respectivamente. El valor de cada figura (10 puntos) se define en la constante `FIGURE_VALUE`, y para representar una carta que ya no se puede utilizar en la partida utilizaremos la constante `UNSET_CARD`.

```
/** Debug server? */
#define SERVER_DEBUG 1

/** Shows the game deck? */
#define DEBUG_PRINT_GAMEDECK 1

/** Initial stack for each player */
#define INITIAL_STACK 5

/** Number of points to win the game */
#define GOAL_GAME 21

/** Value of a figure */
#define FIGURE_VALUE 10

/** Code to represents an empty card (in the deck) */
#define UNSET_CARD 100
```

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. La estructura `tSession` representa el estado de una partida entre dos jugadores. Se recomienda comenzar la implementación del servidor para que gestione una única partida utilizando esta estructura. Una vez el programa cumpla con la funcionalidad pedida, y se depuren los errores, modificad el programa para que, mediante el uso de *threads*, el servidor pueda gestionar varias partidas de forma simultánea. Dentro de la estructura `tSession` vemos la información de cada jugador (su nombre, el *deck* actual, su *stack* y su apuesta). Además, esta estructura contiene `gameDeck`, que representa el mazo de la partida. Las cartas se obtendrán de `gameDeck` y se pasarán al *deck* del jugador que corresponda.

El enumerado `tPlayer` puede utilizarse para representar el jugador que tiene el turno en la partida. Si se emplea correctamente, se puede reducir considerablemente el código en el servidor.

```
/** Sockets of a game used by a thread in the server */
typedef struct threadArgs{
	int socketPlayer1;
	int socketPlayer2;
} tThreadArgs;

/** Session represents a game between 2 players */
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;

/** Players in one session */
typedef enum {player1, player2} tPlayer;
```

Además, se proporcionan algunos subprogramas ya implementados para facilitar la implementación del servidor. Estos subprogramas se describen a continuación:

`tPlayer getNextPlayer` `(tPlayer currentPlayer);`

Calcula el jugador que tiene el turno cuando finaliza la jugada `currentPlayer`.

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

Inicializa un *deck*. Básicamente, coloca todas las cartas en su posición. Debe usarse antes de jugar cada mano para inicializar `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 calculatePoints` `(tDeck *deck);`

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

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

Obtiene una carta aleatoriamente de `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.

El servidor se ejecutará indicando, como parámetro, únicamente el puerto donde realizará la escucha de las conexiones de los clientes.

3. Implementación del cliente

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 de forma que no supere el número máximo de fichas permitido, o si el jugador se ha pasado de 21 puntos, se controlarán en el servidor.

El fichero `clientGame.h` contiene las cabeceras de los subprogramas utilizados en la parte cliente y la constante `DEBUG_CLIENT` para representar el modo de depuración. El fichero `clientGame.c` deberá contener la implementación de estos subprogramas.

Seguidamente se detallan los subprogramas que se proporcionan ya implementados.

`unsigned int readBet` `();`

Realiza la lectura de la apuesta del jugador, introducida por teclado.

`unsigned int readOption` `();`

Realiza la lectura de la acción del jugador, introducida por teclado.

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

4. Ficheros a entregar

Los ficheros necesarios para la realización de esta práctica se encuentran en el fichero `PSD_Sockets_Prac1_BlackJack.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 7 de octubre de 2025, a las 18:00 horas**. La defensa de la práctica se realizará en la clase de laboratorio del mismo día.

No se recogerá ninguna práctica que no haya sido enviada a través del Campus Virtual o esté entregada fuera del plazo `indicado`**.**

Se van a perseguir las copias y plagios de prácticas, aplicando con rigor la normativa vigente.