Programación de Sistemas Distribuidos

Curso 2023/2024

Práctica 2

Diseño e implementación del juego Black Jack en un sistema distribuido

Implementación con Web Services

Esta práctica consiste en el diseño e implementación de un sistema de juegos on-line. Concretamente, se pretende desarrollar tanto la parte cliente, como el servidor, para permitir partidas remotas del juego Black Jack. En general, el juego a implementar será el mismo que el utilizado en la práctica 1, con la excepción de que la comunicación entre clientes y el servidor se llevará a cabo mediante Web Services (utilizando gSOAP con el lenguaje C) en lugar de sockets.

Las normas del juego serán similares a las que utilizamos en la práctica 1, aunque se adaptará la forma en la que se realiza la comunicación entre la aplicación que ejecutará el servidor y la que ejecutarán los usuarios. En la comunicación entre cliente y servidor podemos definir tres elementos que deberán transmitirse entre estas partes: un código, un mensaje y un mazo de cartas. A diferencia de la práctica 1, donde esta información se enviaba por separado, ahora podremos enviar todos los datos en una única estructura.

En esta versión de la práctica no se utilizan todos los códigos que se utilizaron en la versión anterior. Por ejemplo, en esta versión no hay un turno de apuestas, ya que por defecto el usuario siempre apuesta la misma cantidad de fichas (constante `DEFAULT_BET`). Además, el servidor deberá almacenar en un array la información de cada partida en curso, por lo cual deberemos gestionar si hay huecos para nuevas partidas, o si ya existe una partida cuando un jugador solicite realizar alguna acción.

Los valores de los códigos y la definición de las estructuras de datos se proporcionan en el fichero `blackJack.h`. El siguiente cuadro muestra una parte de este fichero:

```
//gsoap blackJackns service name: blackJack
//gsoap blackJackns service style: rpc
//gsoap blackJackns service location: http//localhost:10000
//gsoap blackJackns service encoding: encoded
//gsoap blackJackns service namespace: urn:blackJackns

/** A player is already registered with the same name */
#define ERROR_NAME_REPEATED -1

/** Server is full. No more games are allowed */
#define ERROR_SERVER_FULL -2

/** Player not found */
#define ERROR_PLAYER_NOT_FOUND -3

/** Action taken by the player to stand */
#define PLAYER_STAND 0

/** Action taken by the player to hit a card */
#define PLAYER_HIT_CARD 1

/** Play (player's turn) */
#define TURN_PLAY 2

/** Player must wait and see the rival's play */
#define TURN_WAIT 3

/** Player wins */
#define GAME_WIN 4

/** Player loses */
#define GAME_LOSE 5

/** Length for tString */
#define STRING_LENGTH 256
```

Las dos primeras constantes representan errores que se pueden producir cuando un usuario intenta registrarse en una partida, esto es, un jugador ya está registrado previamente en una partida en curso (`ERROR_NAME_REPEATED`), o el servidor está lleno y no se pueden crear más partidas (`ERROR_SERVER_FULL`).

Las constantes PLAYER_STAND y PLAYER_HIT_CARD representan, respectivamente, las acciones que puede realizar el jugador para plantarse o pedir carta. De forma similar, TURN_PLAY y TURN_WAIT indican cuándo un jugador tiene el turno para poder jugar la mano actual, o cuándo debe esperar a que el rival finalice la jugada. El servidor devolverá el código GAME_WIN cuando un jugador haya ganado la partida, y GAME_LOSE cuando haya perdido. La longitud máxima de los mensajes intercambiados entre clientes y servidor es de STRING_LENGTH caracteres.

Las estructuras de datos utilizadas se muestran a continuación:

```
/** Dynamic array of chars */
typedef char *xsd__string;

/** Structure for sending the player's name and messages from the server */
typedef struct tMessage{
       int __size;
       xsd__string msg;
}blackJackns__tMessage;

/** Structure that represents a deck */
typedef struct tDeck{
       int __size;
       unsigned int *cards;
}blackJackns__tDeck;

/** Response from the server */
typedef struct tBlock{
       int code;
       blackJackns__tMessage msgStruct;
       blackJackns__tDeck deck;
}blackJackns__tBlock;

int blackJackns__register (blackJackns__tMessage playerName, int* result);
```

El tipo xsd__string lo utilizaremos para representar un array de caracteres.

Los mensajes de texto que muestran el estado de la partida, así como los nombres de los jugadores, se podrán representar mediante el tipo xsd__string. Sin embargo, para enviar las cadenas de texto entre cliente y servidor será necesario utilizar la estructura blackJackns__tMessage, la cual contiene el mensaje a enviar (msg) y la longitud del mismo (__size), en número de caracteres.

De forma similar, utilizaremos la estructura blackJackns__tDeck para representar un mazo de cartas. Esencialmente, contiene un array de números enteros, donde cada entero representa una carta (cards), y un número entero que indica la cantidad de cartas existentes en el mazo (__size). La codificación de las cartas es la misma que la utilizada en la práctica 1.

La estructura blackJackns__tBlock contiene la información necesaria para comunicar el estado de la partida a un cliente. En particular, esta estructura contiene un código (code) para indicar el resultado de la acción realizada por el jugador, o su turno, un mensaje (msgStruct) con información textual sobre la partida, y el mazo de cartas de un jugador (deck), el cual puede representar el mazo del jugador que tiene el tuno, o el mazo del jugador rival.

Seguidamente se muestran los servicios ofrecidos. En este caso, el fichero ya contiene uno llamado blackJackns__register, el cual se encarga de registrar a un usuario. En esta práctica, consideramos el nombre del usuario como identificador unívoco, es decir, no pueden existir dos usuarios distintos con el mismo nombre. En el caso de que no exista un usuario ya registrado con el nombre indicado en playerName y exista un hueco en alguna de las partidas, se devolverá un número entero positivo con el identificador de la partida. Si el usuario ya está registrado, se devolverá el código ERROR_NAME_REPEATED, y si no hay hueco para este nuevo jugador, se devolverá ERROR_SERVER_FULL.

Para completar la aplicación pedida, es necesario desarrollar dos nuevos servicios: `getStatus` y `playerMove`. El primero devuelve al cliente el estado de la partida, mientras que el segundo permite al jugador que tiene el turno realizar una acción, que puede ser `PLAYER_STAND` o `PLAYER_HIT_CARD`. En estos servicios se deberá indicar, como parámetros de entrada, el nombre del jugador y el identificador de la partida, entre otros. Cuando el nombre introducido como parámetro (`playeName`) no esté registrado en el sistema, se devolverá el código `ERROR_PLAYER_NOT_FOUND`. En otro caso, se utilizarán los códigos `TURN_MOVE` y `TURN_WAIT` para indicar si al jugador le toca realizar un movimiento o esperar, respectivamente, y `GAME_WIN`, `GAME_LOSE`, para indicar si el jugador ha ganado, o perdido, respectivamente. Además, estos servicios deberán comprobar si `playerName` está registrado en la partida indicada. En caso de no estarlo, se devolverá el código `ERROR_PLAYER_NOT_FOUND` y el cliente finalizará su ejecución.

El fichero `game.h` contiene las cabeceras de los subprogramas auxiliares para, por ejemplo, imprimir un mazo por pantalla o reservar memoria. Además, este fichero contiene una descripción detallada de los parámetros de entrada y salida de cada subprograma, los cuales podrán ser invocados desde los programas cliente y servidor. Seguidamente, se describen a continuación:

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

Muestra el mensaje de error `msg` y finaliza la ejecución del programa.

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

Imprime textualmente el código (`code`) recibido como parámetro.

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

Imprime un mazo de cartas (`deck`) en modo textual. Cada carta se representa con dos caracteres.

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

Imprime un mazo de cartas (`deck`) en modo "gráfico" ☺

```
void printStatus (blackJackns__tBlock *status, int debug);
```

Imprime por pantalla el estado de la partida, contenido en la estructura `status`. El parámetro `debug` indica si se imprime también el código en modo textual.

```
void allocDeck (struct soap *soap, blackJackns__tDeck* deck);
```

Reserva memoria para un mazo de cartas (estructura `blackJackns__tDeck`). El primer parámetro indica el contexto *soap* asociado a la reserva de memoria.

```
void allocClearMessage (struct soap *soap, blackJackns__tMessage* msg);
```

Reserva memoria para un mensaje (estructura `blackJackns__tMessage`). El primer parámetro indica el contexto *soap* asociado a la reserva de memoria. Además, el mensaje se borra para eliminar cualquier residuo en esa parte de la memoria.

```
void allocClearBlock (struct soap *soap, blackJackns__tBlock* block);
```

Reserva memoria para un bloque que contiene el estado del juego (estructura `blackJackns__tBlock`). El primer parámetro indica el contexto *soap* asociado a la reserva de memoria. Este subprograma hace uso de los dos anteriores para reservar la memoria.

Generación del stub del cliente y el esqueleto del servidor

Para generar el *stub*, el *esqueleto*, los ficheros de cabecera y los ficheros encargados de realizar el *marshalling* y *unmarshalling* es necesario utilizar la herramienta `soapcpp2`.

```
$> soapcpp2 -b -c blackJack.h
```

Para compilar la práctica se proporciona un fichero `Makefile` que automatiza esta tarea.

Implementación del cliente

La aplicación cliente deberá establecer comunicación con el servidor para realizar las siguientes acciones:

- Registrar al jugador en el sistema (servicio `register`).
- Solicitar el estado de la partida (servicio `getStatus`).
- Realizar un movimiento para pedir carta o plantarse (servicio `playerMove`).

Dado que esta práctica se implementará utilizando Servicios Web, podremos enviar estructuras al invocar los servicios, lo cual resulta más cómodo que enviar a través de sockets cada dato de forma individual.

Una posible estructura para la aplicación cliente puede ser la siguiente:

```
Mientras (jugador no registrado)
       registrar(nombreJugador, &idPartida)
Mientras (no acabe el juego)
       getStatus (nombreJugador,idPartida, …);
       Imprimir estado del juego
       Mientras (jugador tiene el turno)
              jugada = Leer opción del jugador
              playerMove (nombreJugador,idPartida, jugada, …)
              Imprimir estado del juego
```

Para leer el movimiento a realizar en las jugadas se puede utilizar `readOption()`.

Implementación del servidor

En esta parte se pide atender las peticiones en el servidor de forma paralela. Para ello, cada petición realizada por los jugadores, y recibida por el servidor, conllevará la creación de un *thread* que procese esta petición como corresponda.

Se puede tomar, como punto de partida, la implementación del servidor *multi-thread* explicada en clase donde se implementa una calculadora básica.

El servidor almacena un array de estructuras `tGame` para gestionar las partidas:

```
tGame games[MAX_GAMES];
```

Los tipos de satos utilizados en el servidor se encuentran en el fichero `server.h`.

Dado que cada partida consta de dos jugadores, y se podrán gestionar `MAX_GAMES` partidas a la vez, este array se definirá como una variable global, permitiendo el acceso de los *threads*

ejecutados en la aplicación. Estos *threads* finalizarán cuando haya finalizado la ejecución del servicio solicitado por el cliente.

Con el fin de proporcionar un acceso controlado a este array, será necesario utilizar tanto *mutex* como *variables de condición*. El primer paso será diseñar cómo se realizarán los accesos a la/s región/es crítica/s. Seguidamente, definiremos en la estructura `tGame` tanto los *mutex* como las variables de condición necesarias. **Esta parte no está incluida en los ficheros proporcionados para empezar el desarrollo de la práctica**. Queda a responsabilidad del alumno el diseño e implementación del acceso a las zonas de memoria compartida (sección crítica). Es decir, se deja total libertad para utilizar estas estructuras como se considere oportuno, siempre y cuando se cumplan con los requisitos detallados en el enunciado.

Es muy importante realizar de forma adecuada los accesos a este array. Por ejemplo, el primer caso en el que hay que proteger el acceso concurrente de los *threads* es el servicio para registrar a los jugadores. Si no establecemos un acceso exclusivo a determinados campos como, por ejemplo, los nombres de los jugadores, es posible que el sistema pueda estar en un estado inconsistente.

Una partida puede tener varios estados, definidos por el tipo `tGameState`: `gameEmpty`, `gameWaitingPlayer`, o `gameReady`, que representan una partida vacía, una partida donde se ha registrado un jugador, y una partida con los dos jugadores registrados, respectivamente. Cuando dos jugadores intenten registrarse de forma simultánea, el servicio deberá hacer uso de los mecanismos descritos para acceder a la sección crítica de forma apropiada.

Una posible implementación podría consistir en definir un *mutex* que proteja el array `games` en todos los accesos, de forma que únicamente un *thread* tendrá acceso - a la vez - al array. Sin embargo, esto no sería una solución apropiada, ya que dejamos bloqueados a los procesos, aunque no necesitemos proteger campos -- como los nombres de los jugadores -- limitando de forma considerable la explotación del paralelismo.

Cuando se registran los dos jugadores, dará comienzo la partida. El jugador que empieza la misma se calcula aleatoriamente. Así, si un jugador tiene turno para realizar una acción en la mano actual, podrá realizarla, quedando bloqueado el jugador rival, en espera a que se realice una acción en el juego. Podemos suponer que si el nombre de un jugador está vacío en la estructura `tGame`, `strlen (nombreJugador) == 0`, existe ese hueco para que un jugador se registre en la partida.

Uno de los puntos más sensibles del servidor para establecer la sincronización de los procesos está en el servicio `getStatus`. En particular, este servicio deberá dejar bloqueado a un jugador cuando no sea su turno. Para ello, utilizaremos una variable de condición, de forma que si no es el turno del jugador que ha invocado el servicio, éste permanecerá bloqueado al invocar la llamada *pthread_cond_wait*. Para conocer si es el turno del jugador que invoca el servicio, deberemos acceder a los nombres de los jugadores (campos `player1Name` y `player2Name`) y al campo `currentPlayer` de la partida. Por ejemplo, así:

```
mientras ((playerName == nameJ1 && NO es turno de J1) ||
          (playerName == nameJ2 && NO es turno J2))
            pthread_cond_wait(…);
```

Es importante tener en cuenta cómo desbloqueamos a los procesos. Por ejemplo, si el jugador *jugA* está bloqueado esperando su turno, cuando el jugador *jugB* ha realizado una – o varias – acciones y pierde el turno, es necesario desbloquear a *jugA* para que éste pueda obtener el turno y continuar con el juego. Para ello, utilizaremos la llamada *pthread_cond_signal*.

Cuando un jugador realiza un movimiento en el juego (servicio `playerMove`), es posible que éste provoque el fin de la partida, bien porque haya ganado, bien porque se haya pasado y gane el rival. En este caso, es posible devolver el código de fin de partida en la estructura `blackJackns__tBlock`, además del mensaje correspondiente. De esta forma, este jugador no tendrá que volver a invocar el servicio `getStatus` para comprobar si la partida ha finalizado.

Para facilitar la copia de parámetros a la estructura `blackJackns__tBlock`, se proporciona el subprograma `copyGameStatusStructure` en el servidor:

```
void copyGameStatusStructure (blackJackns__tBlock* status,
                              char* message,
                              blackJackns__tDeck *newDeck,
                              int newCode);
```

donde *status* es la estructura donde se copiarán los datos, *message* es el mensaje con información sobre el estado de la partida, *newDeck* es el mazo del jugador, y *newCode* el código que indica el estado de la partida.

Además, en la parte del servidor se facilitan los siguientes subprogramas:

```
void initGame (tGame *game);
```

Inicializa una partida. Debe invocarse cuando haya finalizado una existente.

```
void initServerStructures (struct soap *soap);
```

Inicializa todas las partidas. Debe invocarse al iniciar el servidor.

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

Inicializa un mazo de cartas. Debe invocarse al principio de cada mano con el *deck* del juego.

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

Vacía un mazo de cartas. Debe invocarse al principio de cada mano con el *deck* de cada jugador.

```
tPlayer calculateNextPlayer (tPlayer currentPlayer);
```

Calcula el siguiente jugador en obtener el turno.

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

Obtiene una carta de un mazo de forma aleatoria.

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

Calcula los puntos actuales de un mazo de cartas (`deck`).

Consideraciones a tener en cuenta

Para desarrollar la práctica se aconseja tener en cuenta las siguientes consideraciones:

- Al inicio de cada servicio, es recomendable marcar con el carácter '\0' el final del nombre del jugador, es decir, `playerName.msg[playerName.__size] = 0;`
- Al realizar la búsqueda de un jugador en el array de partidas, tened cuidado a la hora de liberar el *mutex*, ya que es posible que si hacemos *unlock* después de actualizar el índice utilizado para acceder a cada posición del array, estemos liberando otro *mutex* distinto.
- Utilizad las funciones para reservar memoria comentadas anteriormente. (ver fichero `game.h`).
- Podéis utilizar el código del servidor paralelo visto en clase.

Ficheros a entregar

Esta práctica puede desarrollarse utilizando un código fuente inicial, el cual contiene parte de la implementación y los tipos de datos utilizados. Este código fuente se encuentra en el fichero `PSD_WebServices_Prac2_BlackJack.zip`. Para desarrollar la práctica se deberán modificar, **únicamente,** los ficheros `server.h, server.c, client.h y client.c.` Se deberá entregar, además, un fichero llamado `autores.txt` que contenga el nombre completo de los integrantes del grupo.

Es importante matizar que el fichero .zip entregado 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.

Plazo de entrega

La práctica debe entregarse a través del Campus Virtual **antes del día 16 de noviembre de 2023, a las 16:00 horas**. La defensa de la práctica se realizará en la clase de laboratorio del día **16 de noviembre de 2023**.

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.