Programación de Sistemas Distribuidos

Curso 2024/2025

Práctica 2

Diseño e implementación del juego conecta-4 en modo multijugador distribuido utilizando Servicios Web

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 conecta-4 utilizando Web Services. 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 las mismas que las utilizadas en la práctica 1, salvo que en este caso no se gestionará la información relativa a las tres últimas partidas.

En la comunicación entre cliente y servidor podemos definir tres elementos que deberán transmitirse entre estas partes: un código, un mensaje y un tablero. A diferencia de la práctica 1, donde información se enviaba por separado, ahora podremos enviar todos los datos en una única estructura. Los valores de los códigos y la definición de las estructuras se proporcionan en el fichero `conecta4.h`. El siguiente cuadro muestra una parte de este fichero.

```
//gsoap conecta4ns service name: conecta4
//gsoap conecta4ns service style: rpc
//gsoap conecta4ns service location: http://localhost:10000
//gsoap conecta4ns service encoding: encoded
//gsoap conecta4ns service namespace: urn:conecta4ns
. . .

/** Players */
typedef enum players {player1, player2} conecta4ns__tPlayer;

/** Result for moves */
typedef enum moves {OK_move, fullColumn_move} conecta4ns__tMove;

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

/** Message used to send the player's name and messages sent from server */
typedef struct tMessage{
       int __size;
       xsd__string msg;
}conecta4ns__tMessage;

/** Response from the server */
typedef struct tBlock{
       unsigned int code;
       conecta4ns__tMessage msgStruct;
       int __size;
       xsd__string board;
}conecta4ns__tBlock;

int conecta4ns__register (conecta4ns__tMessage playerName, int *code);

int conecta4ns__getStatus (conecta4ns__tMessage playerName, int gameId,
                           conecta4ns__tBlock* status);
```

La primera parte del fichero establece las directivas gSOAP. En ellas se configura el nombre del servicio (conecta4), el prefijo utilizado (`conecta4ns`) y el espacio de nombres (`conecta4ns`).

El turno del jugador actual se representa con el enumerado `conecta4ns__tPlayer`.

Las constantes y los tipos de datos enumerados para representar la ficha de cada jugador, el turno, y posibles errores, también se han definido en este fichero.

En esta práctica se van a utilizar *arrays* dinámicos de caracteres para representar los nombres y los mensajes. En concreto, se ha definido para ello el tipo `xsd__string`, el cual es, básicamente, un puntero a carácter. Así, para enviar las cadenas de texto entre cliente y servidor, será necesario utilizar la estructura `conecta4ns__tMessage`, la cual contiene el mensaje a enviar (`msg`) y la longitud del mismo, en número de caracteres (`size`). El importante remarcar que, dado que se utilizan punteros para representar las cadenas de caracteres, será necesario reservar memoria de forma dinámica.

La estructura `conecta4ns__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 un tablero (`board`) junto con su tamaño (`size`) en número de caracteres.

Seguidamente se muestran los servicios ofrecidos. El primero, llamado **`conecta4ns__register`**, registra un usuario y recibe como parámetro de entrada el nombre de un usuario para registrarlo en una partida. Como parámetro de salida se devuelve el código con el resultado del registro, el cual puede representar el ID de la partida – si el registro se ha llevado a cabo correctamente – o un código de error indicando el motivo que ha impedido realizar el registro. En el caso de que no haya hueco para este nuevo jugador, se devolverá el código `ERROR_SERVER_FULL`. Si el nombre del jugador **ya existe en la partida** donde se ha encontrado un hueco, se devuelve el código `ERROR_PLAYER_REPEATED`. En esta práctica no vamos a considerar el nombre del usuario como identificador unívoco, es decir, **pueden existir** dos (o más) usuarios con el mismo nombre, siempre que sea en partidas distintas.

El segundo servicio, llamado **`conecta4ns__getStatus`**, obtiene el estado de la partida en el parámetro de salida, el cual incluye el código, el mensaje con la descripción del estado y el tablero. Se utilizarán los mismos códigos empleados en la práctica 1 para indicar el estado de la partida, esto es, `TURN_MOVE` y `TURN_WAIT` para indicar si al jugador le toca insertar ficha o esperar, respectivamente, y `GAMEOVER_WIN`, `GAMEOVER_DRAW`, y `GAMEOVER_LOSE`, para indicar si el jugador ha ganado, empatado o perdido, respectivamente. Como parámetros de entrada se deben proporcionar el nombre del jugador y el ID de la partida.

El último servicio – insertChip – deberá implementar la lógica necesaria para insertar una ficha en la columna `playerMove` del tablero. Como parámetros de entrada se recibirán el ID de la partida y el nombre del jugador, y como parámetro de salida, almacenará el estado de la partida una vez realizado el movimiento.

El fichero `game.h` contiene las cabeceras de los subprogramas encargados de la lógica del juego. Además, este fichero contiene una descripción detallada de los parámetros de entrada y salida de cada subprograma. Seguidamente, se describen a continuación:

```
void initBoard (tBoard board);
```

Inicializa un tablero. Debe ser invocado antes de comenzar una partida.

```
tMove checkMove (tBoard board, unsigned int column);
```

Comprueba el movimiento de un jugador. El movimiento será el número `[0-BOARD_WIDTH)` de la columna (`column`) donde el jugador intenta insertar la ficha.

```
void insertChip (tBoard board, tPlayer player, unsigned int column);
```

Introduce una ficha en la columna indicada por el jugador, actualizando el tablero.

```
int checkWinner (tBoard board, tPlayer player);
```

Comprueba si el jugador actual es el ganador. Este subprograma deberá invocarse después de haber realizado el movimiento.

```
int isBoardFull (tBoard board);
```

Comprueba si el tablero está lleno, esto es, ya no es posible realizar más movimientos.

La parte cliente no necesita manipular (ni debe) el tablero. Una vez reciba la información del servidor, podrá utilizar el siguiente subprograma para mostrarlo por pantalla:

```
void printBoard (xsd__string board, xsd__string message);
```

Imprime por pantalla el estado de la partida, contenido en la estructura `status`.

```
void allocClearBlock (struct soap *soap, conecta4ns__tBlock* block);
```

Reserva memoria para un bloque que contiene el estado del juego (estructura `conecta4ns__tBlock`). El primer parámetro indica el contexto *soap* asociado a la reserva de 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 conecta4.h
```

Implementación del cliente

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

- Registrar al jugador en el sistema.
- Solicitar el estado de la partida.
- Realizar un movimiento insertando una ficha en el tablero.

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 (no acabe el juego)
      Pedir estado del juego
      Imprimir estado del juego
      Mientras (jugador tenga el turno)
            Leer movimiento
            Insertar ficha
            Si (fin de juego)
                  Fin de juego
            Si (no es fin de juego)
                  Imprimir tablero
```

Implementación del servidor

En esta parte se pide atender las peticiones en el servidor de forma concurrente. 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];
```

Dado que cada partida tendrá dos jugadores, y se podrán gestionar varias partidas a la vez, este array se definirá como una variable global, permitiendo el acceso de varios *threads* ejecutados en la aplicación. Estos *threads* se crearán cuando un cliente solicite un servicio del servidor, ejecutando el servicio indicado como parámetro.

Con el fin de proporcionar un acceso controlado a este array, será necesario utilizar tanto variables de tipo cerrojo (*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.

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 en el servicio para registrar a los jugadores. Si no establecemos un acceso exclusivo a determinados campos como, por ejemplo, el estado de la partida, es posible que el sistema llegue a estar en un estado inconsistente.

Una partida puede tener varios estados (`tGameStatus`): `gameEmpty`, `gameWaitingPlayer`, o `gameReady`, que representan una partida vacía, una partida donde se han 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.

Cuando se registran los dos jugadores, dará comienzo la partida. El jugador que empieza la misma se calcula aleatoriamente. Para simplificar la implementación del servidor, es posible que un jugador registrado comience la partida aunque no se haya registrado su rival, siempre y cuando tenga el turno para jugar. Así, si un jugador se registra correctamente, y tiene turno para insertar una ficha, podrá realizar el primer movimiento, quedando seguidamente bloqueado hasta que su rival realice un movimiento correcto.

Uno de los puntos más sensibles del servidor para establecer la sincronización de los procesos está en el servicio **`conecta4ns__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 realizar una operación `pthread_cond_wait`.

```
mientras ((playerName == nameJ1 && no turno J1) ||
          (playerName == nameJ2 && no turno J2))
            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.

Es importante tener en cuenta cómo desbloqueamos a los procesos. Por ejemplo, si hay un proceso bloqueado esperando su turno, podemos hacer uso de la llamada a `pthread_cond_signal`. De esta forma, desbloquearemos al proceso bloqueado, permitiendo continuar la ejecución al proceso que tenga el turno.

Cuando un jugador inserta una ficha en el tablero, es posible que este movimiento provoque el fin de la partida, bien porque haya ganado, bien porque se haya llenado el tablero de fichas y no exista ganador (empate). En este caso, es posible devolver el código de fin de partida en la estructura `conecta4ns__tBlock`, además del tablero. De esta forma, este jugador no tendrá que volver a invocar el servicio **`conecta4ns__getStatus`** para comprobar si la partida ha finalizado.

Para facilitar la copia de parámetros a la estructura `conecta4ns__tBlock`, se proporciona el subprograma **`copyGameStatusStructure`** en el servidor:

```
void copyGameStatusStructure (conecta4ns__tBlock* status,
                              char* message,
                              char* board,
                              int newCode);
```

donde `status` es la estructura donde se copiarán los datos, `message` es el mensaje con información sobre el estado de la partida, `board` es el tablero, y `newCode` el código que indica el estado de la partida.

Queda a responsabilidad del/de la alumno/a 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.

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_Prac2_WS.zip`.

Para desarrollar este apartado se deberán modificar, **únicamente,** los ficheros `server.h`, `server.c`, `client.h` y `client.c`. Adicionalmente, se permite agregar más subprogramas en los ficheros `client.h` y `server.h`, aunque no es necesario para poder implementar completamente la práctica.

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.

Entrega de la práctica

Para entregar esta práctica se habilitará un entregador en la página de la asignatura del Campus Virtual. La fecha límite para entregarla será el **día 5 de Noviembre de 2024 a las 18:00 horas.**

No se permitirá la entrega de prácticas fuera del plazo establecido.

La entrega se realizará mediante un **único fichero con extensión .zip**, el cual deberá incluir los ficheros necesarios para ejecutar y compilar la aplicación pedida, además del fichero `nombres.txt`, que contendrá el nombre y apellidos de los integrantes del grupo.

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