

Tema 1: programación cliente/servidor

Práctica 1: Dominó C/S

Nombres: Fernando Mateus (NIUB: 16106425)
Alex Rodríguez (NIUB: 14987092)

Asignatura: Software Distribuido

Profesor: Eloi Puertas

Grupo de prácticas: B (jueves, de 17:00 a 19:00)

Fecha: 18 de marzo de 2014

Introducción

La primera práctica de Software Distribuido nos propone meternos de lleno en la **arquitectura cliente-servidor**, que básicamente se basa en distribuir una aplicación en dos partes muy diferenciadas dependiendo de su funcionalidad. En el caso que nos ocupa a nosotros, nuestro objetivo no era otro que desarrollar una versión virtual del clásico juego del **Dominó** en Java, usando programación multihilo y controlando a un nivel muy bajo la comunicación entre los clientes y el servidor.

El juego del Dominó es muy sencillo y se juega con fichas (no repetidas) que muestran dos números en sus extremos, cuyo valor se encuentra entre el 0 y el 6, ambos incluidos. Contamos con una serie de fichas en total (28), cada jugador roba un número determinado al comienzo (7) y el objetivo no es otro que, respetando los turnos, colocar de una en una alguna de las fichas de la mano en el tablero haciendo coincidir el número de sus extremos. Si no se puede tirar hay que robar ficha o bien hasta que exista un movimiento o bien hasta que ya no queden en el resto.

Finalmente, gana el jugador que antes vacíe su mano, aunque también existe un caso de empate (ningún jugador puede hacer ningún movimiento y no quedan fichas para robar en el resto), en el que gana aquel cuya mano tenga menos valor (sumatorio de los números de sus fichas).

En el caso de nuestra práctica sólo contemplamos un jugador humano (cliente), mientras que una inteligencia artificial con nulas intenciones de hacerse con la victoria (servidor) responde a sus jugadas. Las normas antes definidas son mantenidas en nuestra adaptación, aunque tampoco sería demasiado complicado hacer una versión multijugador que aceptara más clientes.

Diseño

A la hora de diseñar una aplicación cliente servidor, tenemos que tener en cuenta un concepto muy importante para definir qué parte ocupa cada rol: quién hace la solicitud y quién la responde. En este caso, el servidor es el que lleva las riendas del juego y el que crea la partida, controlando quién gana la partida (de forma que podría considerarse poco ética) y siendo el responsable de enviar la mano inicial al cliente y de brindarle las fichas necesarias cuando robe. El cliente (usuario), en cambio, sólo podrá comenzar a jugar una vez el servidor haya aceptado sus solicitudes, siendo únicamente capaz de jugar sus fichas y poco más.

A nivel de diseño diferenciamos varias secciones interesantes:

Threading

Tal y como hemos planteado la práctica, tendremos únicamente un servidor que aceptará la conexión de múltiples clientes, jugando cada uno a la vez su propia partida de forma individual, por lo que pensar en utilizar **programación multihilo** es una idea excelente.

El servidor cuenta con un `ServerSocket` que espera la solicitud de comenzar la partida por parte del cliente. Una vez aceptada, obtenemos de él un `DataSocket` que será el encargado de recibir/enviar los mensajes entre servidor y cliente, mientras que el `ServerSocket` vuelve a esperar a que algún cliente llame a su puerta. Dicho esto, es el momento de establecer la comunicación entre ambos elementos cuando generamos un nuevo *thread* (en nuestro caso, de la clase `PartidaDominoServer`) para que el servidor pueda gestionar paralelamente diferentes funciones. El lado del cliente, en cambio, no necesita ningún tipo de implementación multihilo ya que únicamente llevará a cabo la partida con el servidor.

Los threads son un arma de doble filo, ya que aunque nos permiten distribuir los procesos de forma eficiente y, en algunos casos, facilitar la programación, también nos abren las puertas a los problemas de concurrencia en el uso de variables globales, lo que nos obliga a usar algunos mecanismos de sincronización (semáforos, monitores...) para evitar desastres. Para bien o para mal, en nuestro contexto el único dato global que se encuentra en esa tesitura es el contador de clientes conectados a la vez, por lo que no hemos tenido demasiados problemas al respecto, utilizando la funcionalidad *synchronized* que Java nos ofrece de serie.

Timeouts

Al ser una aplicación cliente-servidor en la que de serie se da por hecho que cada una de las partes se encuentra en una máquina diferente, el correcto funcionamiento del programa dependerá totalmente de la conexión entre los sockets de ambas partes. Para más inri, usamos **operaciones de lectura y escritura bloqueantes**, por lo que es crucial que todo marche sobre ruedas para evitar que el programa se pueda quedar colgado esperando a un mensaje que nunca va a llegar, por poner un ejemplo. De ahí que el uso de timeouts para controlar dichas comunicaciones sea necesario.

Hemos establecido dos tiempos de espera dependiendo de la situación. Por un lado el tiempo de menú (único del servidor), que hace referencia al tiempo que se encuentra el cliente en la interfaz sin hacer nada, mientras que por otro lado tenemos el tiempo de lectura (presente en servidor y cliente, y más pequeño que el anterior), con el que controlaremos el tiempo que se tarda en recibir un mensaje concreto.

En el caso de que el timeout llegue a su fin, se producirá una excepción de diferente tipo (o bien por timeout o bien por desconexión del socket, **SocketException** e **InterruptedIOException**, respectivamente) que será tratada, siendo posteriormente cerrado el thread de forma limpia.

Modelo Vista Controlador (MVC)

El **Modelo Vista Controlador** es una arquitectura de programación con la que se pretende separar diferentes partes de un programa dependiendo de la función y el papel de las mismas, además de intentar que cada clase sea responsable de sí misma. A grandes rasgos, el **modelo** correspondería a la representación básica de los datos (en nuestro caso, clases), el **controlador** sería el encargado de gestionar los procesos internos y la **vista** llevaría a cabo la representación de la visualización del trabajo del antes citado, aunque dependiendo del contexto a veces es complicado situar la frontera entre las diferentes partes del paradigma.

En las tablas de la siguiente página explicamos a grandes rasgos las diferentes clases que componen cada uno de los proyectos, separando por las capas de modelo, vista y controlador.

Servidor	Clase	Información
Controlador	ComUtils	Clase encargada de la comunicación entre sockets.
	ControladorServidor	Clase encargada de usar ComUtils sabiamente para enviar/recibir los mensajes establecidos en el protocolo.
	IllegalActionException	Excepción que saltará cuando se reciba alguna acción ilegal.
	RechazarConexion	Pseudoservidor que enviará un mensaje a un Cliente cuando el servidor no pueda albergar más partidas, para desconectarse acto seguido.
	Rules	Contiene constantes e información importante de las partidas.
Modelo	Ficha	Ficha del Dominó.
	Histórico	Contiene el histórico de acciones de una partida.
	Tablero	Tablero de la partida, que tendrá las fichas en juego y mostrará los extremos para facilitar la tarea.
	Tirada	Clase que agrupa una una ficha, la posición del tablero en la que irá y la mano restante del servidor.
Vista	DominoesServer	Contiene el main del juego. Acepta las conexiones de cliente y crea threads por cada partida.
	PartidaDominoServer	Núcleo de las partidas. Usa activamente ControladorServidor para poder llevar a cabo el juego.

Explicación de las clases del Servidor.

Cliente	Clase	Información
Controlador	ComUtils	Clase encargada de la comunicación entre sockets.
	ControladorCliente	Clase encargada de usar ComUtils sabiamente para enviar/recibir los mensajes establecidos en el protocolo.
	IllegalActionException	Excepción que saltará cuando se reciba alguna acción ilegal.
	Rules	Contiene constantes e información importante de las partidas.
Modelo	Ficha	Ficha del Dominó.
	Histórico	Contiene el histórico de acciones de una partida.
	Tablero	Tablero de la partida, que tendrá las fichas en juego y mostrará los extremos para facilitar la tarea.
	FinDePartida	Mensaje que contiene la información de un final de partida recibido del servidor.
	TiradaCliente	Clase que agrupa una una ficha y la posición del tablero en la que irá.
	TiradaServidor	Clase que agrupa una una ficha, la posición del tablero en la que irá y la mano restante del servidor.
Vista	DominoesCliente	Contiene el main del juego. Se conecta con el Servidor y da comienzo la partida.
	PartidaDominoCliente	Núcleo de las partidas. Usa activamente ControladorCliente para poder llevar a cabo el juego.

Explicación de las clases del Cliente.

Excepciones y errores

Respecto al tratamiento de errores, hemos optado por la solución más drástica: cerrar conexión por lo sano entre el cliente y el servidor, terminando así la partida. La hemos considerado la mejor opción, no por falta de recursos/posibilidades para intentar arreglar la situación (mensajes mal enviados, mensajes mal recibidos, jugadas ilegales, errores externos...) sino por asunto de protocolo, donde este tipo de situaciones no se encuentra definido.

Teniendo en cuenta que la gracia de compartir estrictamente protocolo es que cualquier cliente y servidor puedan interactuar entre sí, si le añadimos alguna nueva faceta podemos echar a perder dicha posibilidad. Aun así, antes de cerrar la comunicación se intenta informar de qué error ha ocurrido al otro componente del tándem, una faceta que el protocolo sí contempla y engloba en cinco ramas diferentes.


Dicho esto, en los principales bloques de código en los que se lleva a cabo la partida encontramos que se capturan las excepciones (incluidas algunas creadas por nosotros mismos, como la **IllegalActionException**) de los niveles inferiores, de tal manera que siempre se intentará cerrar de forma limpia el *thread*.

Ejemplos de ejecución: Servidor

Para iniciar el servidor basta con ejecutar el archivo.jar indicando por parámetro el puerto por el cual se quiere escuchar con el siguiente comando:

java -jar servidor.jar <puerto>.

Con el servidor en marcha, quedará a la espera a que los clientes soliciten iniciar una partida.



```
ServerSocket preparado en el puerto 8056
Servidor: Esperando conexión de un cliente...
```

Servidor esperando la conexión de un cliente.

Ejemplos de ejecución: Cliente

Para iniciar una partida, hay que ejecutar el archivo.jar del cliente indicando la dirección del servidor y puerto en el cual espera conexión con el siguiente comando:

java -jar cliente.jar <ip> <puerto>.

```

-----Estado de partida-----
- Cantidad fichas servidor: 6
- Cantidad fichas cliente: 7
- Cantidad fichas tablero: 1
- Cantidad fichas resto: 14
- Histórico:

      Empieza servidor.          Servidor tira [6|6]

- Fichas en tablero:

      [6|6]

- Fichas cliente:

1) [1|5]      2) [4|5]      3) [2|4]      4) [1|4]
5) [0|3]      6) [2|3]      7) [1|6]

-----
Selecciona que ficha quieres tirar:

```

Cliente empezando una partida tras conectarse a un servidor.

Transcurso de una partida

Cliente

```

-----Estado de partida-----
- Cantidad fichas servidor: 2
- Cantidad fichas cliente: 3
- Cantidad fichas tablero: 9
- Cantidad fichas resto: 14
- Histórico:

      Empieza servidor.          Servidor tira [5|6]          Cliente tira [6|4] R.
      Servidor tira [4|1].        Cliente tira [1|1] R.        Servidor tira [1|2].
      Cliente tira [0|5] L.        Servidor tira [2|6].        Cliente tira [2|0] L.
      Servidor tira [2|2].

- Fichas en tablero:

      [2|2][2|0][0|5][5|6][6|4][4|1]
      -> [1|1][1|2][2|6]

- Fichas cliente:

1) [3|6]      2) [2|5]      3) [4|4]

-----
Selecciona que ficha quieres tirar:

```

Ejemplo de cómo transcurre una partida desde el punto de vista del cliente.

Servidor

```

--Estado de partida ID 1-----
tidad fichas servidor: 2
tidad fichas cliente: 3
tidad fichas tablero: 9
tidad fichas resto: 14
has en tablero: [[2|2], [2|0], [0|5], [5|6], [6|4], [4|1], [1|1], [1|2], [2|6]]
has servidor: [[0|6], [0|3]]
tórico:
ración 1:
  Empieza servidor.
ración 2:
  Cliente tira la ficha [6|4] R
  Servidor tira la ficha [4|1] R .
ración 3:
  Cliente tira la ficha [1|1] R
  Servidor tira la ficha [1|2] R .
ración 4:
  Cliente tira la ficha [0|5] L
  Servidor tira la ficha [2|6] L .
ración 5:
  Cliente tira la ficha [2|0] L
  Servidor tira la ficha [2|2] L .

```

Ejemplo de cómo transcurre una partida desde el punto de vista del servidor.

Cliente

```
-----Estado de partida-----
- Cantidad fichas servidor: 1
- Cantidad fichas cliente: 2
- Cantidad fichas tablero: 11
- Cantidad fichas resto: 14
- Histórico:

      Empieza servidor.
      Servidor tira [4|1].
      Cliente tira [0|5] L.
      Servidor tira [2|2].
      Servidor tira [3|0].

      Servidor tira [5|6]
      Cliente tira [1|1] R.
      Servidor tira [2|6].
      Cliente ficha ko.[2|5]

      Cliente tira [6|4] R.
      Servidor tira [1|2].
      Cliente tira [2|0] L.
      Cliente tira [6|3] R.

- Fichas en tablero:

      [2|2][2|0][0|5][5|6][6|4][4|1]
      -> [1|1][1|2][2|6][6|3][3|0]

- Fichas cliente:

1) [4|4]      2) [2|5]

-----
Selecciona que ficha quieres tirar:
```

Ejemplo de cómo transcurre una partida desde el punto de vista del cliente.

Servidor

```
-----Estado de partida ID 1-----
- Cantida fichas servidor: 1
- Cantida fichas cliente: 2
- Cantida fichas tablero: 11
- Cantida fichas resto: 14
- Fichas en tablero: [[2|2], [2|0], [0|5], [5|6], [6|4], [4|1], [1|1], [1|2], [2|6], [6|3], [3|0]]
- Fichas servidor: [[0|6]]
- Histórico:
  • Iteración 1:
    Empieza servidor.
  • Iteración 2:
    Cliente tira la ficha [6|4] R
    Servidor tira la ficha [4|1] R .
  • Iteración 3:
    Cliente tira la ficha [1|1] R
    Servidor tira la ficha [1|2] R .
  • Iteración 4:
    Cliente tira la ficha [0|5] L
    Servidor tira la ficha [2|6] L .
  • Iteración 5:
    Cliente tira la ficha [2|0] L
    Servidor tira la ficha [2|2] L .
  • Iteración 6:
    Cliente tira la ficha [6|3] R
    Servidor tira la ficha [3|0] R .
```

Ejemplo de cómo transcurre una partida desde el punto de vista del servidor.

Aunque el servidor es totalmente autónomo, hemos optado por mostrar el estado actual de la partida (todas tienen un identificador) en la cual el servidor realiza un movimiento. De esta forma se puede seguir de visualmente las diferentes partidas que se están llevando a cabo, lo que permite controlar el flujo de las partidas y detectar posibles fallos tanto de lógica como de protocolo.

Cliente

```
-----Estado de partida-----
- Cantidad fichas servidor: 1
- Cantidad fichas cliente: 1
- Cantidad fichas tablero: 12
- Cantidad fichas resto: 14
- Histórico:

      Empieza cliente.
      Cliente tira [5|6] R.
      Servidor tira [0|0].
      Cliente tira [3|6] R.
      Servidor tira [3|3].

      Cliente tira [5|5] R.
      Servidor tira [6|4].
      Cliente tira [0|2] R.
      Servidor tira [1|0].

      Servidor tira [0|5].
      Cliente tira [4|0] R.
      Servidor tira [2|3].
      Cliente tira [3|1] L.

- Fichas en tablero:

      [3|3][3|1][1|0][0|0][0|5][5|5]

      -> [5|6][6|4][4|0][0|2][2|3][3|6]

- Fichas cliente:

1) [1|5]

-----
Lamentablemente no puedes tirar, haz enter para pasar/robar :(
```

Ejemplo de cómo el cliente roba una ficha.

Servidor

```
En constructor char
-----Estado de partida ID 2-----
- Cantida fichas servidor: 3
- Cantida fichas cliente: 2
- Cantida fichas tablero: 13
- Cantida fichas resto: 10
- Fichas en tablero: [[4|4], [4|0], [0|6], [6|5], [5|5], [5|4], [4|3], [3|5], [5|0], [0|3], [3|3], [3|1], [1|1]]
- Fichas servidor: [[1|2], [2|2], [6|6]]
- Histórico:
- Iteración 1:
      Empieza servidor.
- Iteración 2:
      Cliente tira la ficha [5|4] R
      Servidor tira la ficha [4|3] R .
- Iteración 3:
      Cliente tira la ficha [3|5] R
      Servidor roba [6|5]
      Servidor tira la ficha [6|5] R .
- Iteración 4:
      Cliente tira la ficha [0|6] L
      Servidor tira la ficha [4|0] L .
- Iteración 5:
      Cliente tira la ficha [4|4] L
      Servidor roba [5|0]
      Servidor tira la ficha [5|0] L .
- Iteración 6:
      Cliente tira la ficha [0|3] R
      Servidor tira la ficha [3|3] R .
- Iteración 7:
      Cliente tira la ficha [3|1] R
      Servidor tira la ficha [1|1] R .
- Iteración 8:
      Cliente roba la ficha [2|4].
```

Ejemplo de cómo el cliente roba una ficha.

Cliente

```
-----Estado de partida-----
- Cantidad fichas servidor: 0
- Cantidad fichas cliente: 4
- Cantidad fichas tablero: 13
- Cantidad fichas resto: 11
- Histórico:

    Empieza servidor.
    Servidor tira [5|3].
    Cliente solicita robar.
    Cliente roba [4|6].
    Cliente tira [0|4] L.
    Servidor tira [1|0].
    Cliente tira [2|2] R.

    Servidor tira [6|6]
    Cliente solicita robar.
    Cliente roba [1|2].
    Cliente tira [4|6] L.
    Servidor tira [3|3].
    Cliente tira [4|1] L.
    Servidor tira [2|6].

    Cliente tira [6|5] R.
    Cliente roba [1|4].
    Cliente solicita robar.
    Servidor tira [4|4].
    Cliente tira [0|0] L.
    Servidor tira [3|2].

- Fichas en tablero:

    [4|1][1|0][0|0][0|4][4|4][4|6]

    -> [6|6][6|5][5|3][3|3][3|2][2|2]

    -> [2|6]

- Fichas cliente:

1) [5|5]      2) [0|2]      3) [2|5]      4) [1|2]
```

```
-----
-----FIN DE LA PARTIDA-----
--
--      No sabes jugar al d6mino!
--      Gana Servidor
--
-----
```

Servidor

```
-----Estado de partida ID 3-----
- Cantida fichas servidor: 0
- Cantida fichas cliente: 4
- Cantida fichas tablero: 13
- Cantida fichas resto: 11
- Fichas en tablero: [[4|1], [1|0], [0|0], [0|4], [4|4], [4|6], [6|6], [6|5], [5|3], [3|3], [3|2], [2|2], [2|6]]
- Fichas servidor: []
- Histórico:
- Iteración 1:
    Empieza servidor.
- Iteración 2:
    Cliente tira la ficha [6|5] R
    Servidor tira la ficha [5|3] R .
- Iteración 3:
    Cliente roba la ficha [1|4].
- Iteración 4:
    Cliente roba la ficha [1|2].
- Iteración 5:
    Cliente roba la ficha [4|6].
- Iteración 6:
    Cliente tira la ficha [4|6] L
    Servidor tira la ficha [4|4] L .
- Iteración 7:
    Cliente tira la ficha [0|4] L
    Servidor tira la ficha [3|3] L .
- Iteración 8:
    Cliente tira la ficha [0|0] L
    Servidor tira la ficha [1|0] L .
- Iteración 9:
    Cliente tira la ficha [4|1] L
    Servidor tira la ficha [3|2] L .
- Iteración 10:
    Cliente tira la ficha [2|2] R
    Servidor tira la ficha [2|6] R .
    Servidor gana la partida al vaciar su mano.

Final de la partida
Fin del thread 3.
```

Con tal de intentar conservar la integridad de la partida, el servidor comprueba que las tiradas que realiza el cliente sean legales, comprobando si la ficha está repetida o no. Siguiendo nuestra posición en cuanto a tratamiento de errores, hemos preparado el cliente para que no se puedan realizar acciones ilegales tales como enviar fichas que no están en mano del jugador o robar cuando él mismo puede tirar.

Resultados de la sesión de test

Errores nuestros detectados durante la sesión de test:

- No obligamos al cliente a lanzar su mejor tirada si es él el que empieza.
- No tenemos timeouts.
- Poco control de excepciones.

Teniendo en cuenta que la versión del cliente que usamos era bastante estricta a la hora de enviar fichas inexistentes o de hacer movimientos imposibles, nuestras pruebas en el resto de servidores fueron bastante limitadas. Sin duda, fue algo que se nos pasó totalmente por alto en su desarrollo que nos impidió poner a prueba a los servidores de los otros grupos. Para compensar, hemos creado un nuevo cliente que no da ningún problema a la hora de enviar fichas falsas (en todos los sentidos de la palabra) para descubrir los puntos débiles de un servidor que siga el mismo protocolo.

Servidor	Resultados con cliente del grupo 5
Grupo 1	No se recibía respuesta al mensaje inicial.
Grupo 2	Todo correcto.
Grupo 3	Se reciben mensajes no identificados.
Grupo 4	Todo correcto.
Grupo 5	-
Grupo 6	No termina la partida y se nos ofrece robar para seguir jugando.
Grupo 7	Todo correcto.
Grupo 8	Se reciben tiradas del servidor de forma aleatorias inválidas.
Grupo 9	Se reciben tiradas del servidor de forma aleatorias inválidas, que derivan en una NullPointerException.
Grupo 10	Todo correcto.
Grupo 11	Se reciben mensajes no identificados.
Grupo 12	Se encuentran fichas repetidas, no se controla el final de partida.

Resultados de nuestro cliente contra el resto de servidores.

Cliente	Resultados con servidor del grupo 5
Grupo 1	Problemas de protocolo.
Grupo 2	Todo correcto.
Grupo 3	Todo correcto.
Grupo 4	Todo correcto.
Grupo 5	-
Grupo 6	Todo correcto.
Grupo 7	Todo correcto.
Grupo 8	Todo correcto.
Grupo 9	Todo correcto.
Grupo 10	Todo correcto.
Grupo 11	Todo correcto.
Grupo 12	No se controla el recibir fichas repetidas.

Resultados de nuestro servidor contra el resto de clientes.