Programación Concurrente y de Tiempo Real
Grado en Ingeniería Informática
Examen Final de Prácticas de la Asignatura
Febrero de 2012

1 Enunciados

1. (Cine) Un sistema de compra de entradas remoto permite a sus usuarios obtener la entrada para la película que desean ver sin pasar por taquilla. Se desea disponer de una arquitectura RMI que permita implantar tal sistema. Para ello, suponga las siguientes simplificaciones y especificaciones:

- Se gestionará un único cine que tiene 4 películas distintas en diferentes salas.
- Todas las salas tienen 100 butacas disponibles excepto una, que dispone de 200 butacas.
- Los clientes deben dirigirse al servidor indicando la película que desean ver, y la butaca que prefieren.
- El servidor contestará asignando la butaca preferida si está disponible, o notificando que no lo está. En este último caso, el cliente deberá volver a elegir una nueva opción.
- Toda la arquitectura del lado del servidor debe correr en el puerto 2001.
- Debe generar una políticica de seguridad sencilla en un fichero `cine.policy`, que restrinja las peticiones de acceso al puerto indicado.
- Documente su código, pues será sometido a `javadoc`.
- Prevea cualquier otra necesidad que considere oportuna. Si lo hace, justifique sus decisiones en un fichero `justif.txt`.
- El código que genere deberá ejecutarse con stub y skeleton, únicamente con stub, y sin ninguno de ellos, bajo generación dinámica de resguardos.

Escriba los ficheros que conforman su arquitectura en los ficheros `iCine.java, clientCine.java y serverCine.java`. Si necesita ficheros adicionales, añádalos bajo el formato de nombre siguiente: `cadCine.java`, donde `cad` será una descripción base del contenido del fichero. Por ejemplo `salaCine.java` si el fichero va a modelar una clase sala, etc.

2. (*Life*) El juego de la vida (Conway, 1977) es un mundo en el que se simulan estrategias evolutivas, modelado mediante un autómata celular bidimensional que teóricamente se implanta en un retículo infinito de casillas discretas llamadas células. Cada célula puede estar viva ó muerta. Inicialmente este mundo, que llamaremos *Life*, se siembra de forma aleatoria con una semilla compuesta por un número dado de células vivas. El tiempo transcurre de forma discreta y en cada instante del tiempo, el estado del mundo cambia de una generación a la siguiente, según las reglas siguientes:

- una célula viva con menos de dos vecinas vivas muere por aburrimiento.
- una célula viva con dos o tres vecinas vivas permanece en su estado.
- una célula viva con más de tres cecinas vivas muere por falta de alimento.
- una célula muerta con exactamente tres vecinas viva revive por reproducción.

Escriba un programa multihebrado en java para simular a *Life* de acuerdo a las siguientes especificaciones:

- Utilizará el concepto de vecindad de *Moore* (cada célula tiene 8 vecinas; las que la rodean). La alternativa, que no utilizaremos, es la vecindad de *Von-Neumann*: vecinas en los cuatro puntos cardinales.
- *Life* tendrá una topología de toro (rosquilla o donut), de dimensiones 10×10.
- Documente su código, pues será sometido a `javadoc`.
- Habrá diez *threads* en ejecución concurrente cada uno de los cuales calculará la siguiente generación para una fila de células. Estos *threads* debéran ser creados en el programa principal, y se tomarán obligatoriamente de un *pool* de *threads* creado a tal efecto.
- Deberá encapsular el toro de *Life* en un monitor java que ofrezca una interfaz de métodos con al menos los dos siguientes: `nextGen (fila)`, que permitirá que el hilo correspondiente calcule la siguiente generación de la fila que le corresponde y `status()`, que permitirá conocer cuántas celulas vivas hay en el toro en el momento de invocar el método. Cada hilo ciclará a través del método `run()` invocando primero a `netxGen` y luego mostrando en pantalla el estado actual del toro mediante `status`, con una salida similar a

  ```
  Soy el hilo m, mi prioridad es n y el estatus actual es:p células vivas
  ```

- Deberá iniciar el toro con 50 células vivas dispersas de forma aleatoria en la retícula. Cada hilo ciclará 100 veces y terminará. El hilo `main` deberá esperar la terminación de todos los hilos hijos un procotolo basado en la clase `CyclicBarrier`. (**siga leyendo en la otra cara**)

Ficheros a generar: como mínimo, dos; `Life.java` que contendrá el toro monitorizado y `pruebaLife.java` con el programa que crea los hilos y los activa. Si utiliza alguno más nómbrelo de esta forma: `cadLife.java`, donde `cad` aludirá al contenido del fichero. Ejemplo: si desea utilizar una clase para modelar las células, `celLife.java` contendría su código.

(Espacio para anotaciones de corrección)