

# Prácticas de laboratorio

## Problema de las hormigas

### (2 sesiones)

## Concurrencia y Sistemas Distribuidos

### Introducción

---

Esta práctica tiene como objetivo analizar y completar un programa concurrente en el que se aplican distintas condiciones de sincronización entre hilos, haciendo uso de algunas de las herramientas Java proporcionadas en la biblioteca *java.util.concurrent*. Cuando haya concluido sabrá:

- Utilizar de forma adecuada la herramienta *ReentrantLock*.
- Generar *Condiciones* asociadas a un determinado *ReentrantLock* y utilizarlas de forma apropiada.
- Utilizar barreras para la sincronización de los hilos.
- Aplicar soluciones para el problema de interbloqueos.

Esta práctica se realiza en grupo, donde cada **grupo debe estar formado por dos personas**.

La duración estimada de esta práctica es de dos semanas. Tenga en cuenta que necesitará destinar algo de tiempo de su trabajo personal para concluir la práctica. Utilice para ello los laboratorios docentes de la asignatura y las aulas informáticas del centro en horarios de libre acceso. Puede utilizar también su ordenador personal.

A lo largo de la práctica verá que hay una serie de ejercicios a realizar. Se recomienda resolverlos y anotar sus resultados para facilitar el estudio posterior del contenido de la práctica.

## El problema de las hormigas

---

Se dispone de un territorio en el que vive un conjunto de hormigas. El territorio se modela como una matriz rectangular  $N \times N$ , siendo  $N$  un valor a definir por el usuario (por defecto, vale 10). Por su parte, cada hormiga se implementa como un hilo Java, con una posición inicial aleatoria dentro del territorio. Cada hormiga se mueve libremente por el territorio, moviéndose en cada momento a una celda contigua a la que se encuentra (i.e., arriba, abajo, izquierda o derecha). En este ejemplo, cada hormiga realizará un total de 3 movimientos (steps). Y por defecto se lanzarán **15 hormigas**.

Como restricción a los movimientos de las hormigas, se dispone de la siguiente norma:

- En cada celda de la matriz puede haber como máximo **una sola hormiga**.

Se pretende resolver este problema utilizando el concepto de **monitor**. Para ello, el territorio actuará como un monitor, de modo que ofrezca métodos sincronizados para la actualización de los parámetros del territorio. Además, cuando una hormiga quiera desplazarse a una celda y ésta se encuentre ocupada, deberá suspenderse en una variable condición y quedará allí suspendida hasta que sea reactivada por otra hormiga.

Existen dos variantes para atacar este problema:

a) Definir una **única variable condición asociada a todo el territorio**, donde las hormigas se suspenden si no pueden desplazarse a la celda deseada.

b) Definir **una variable condición por cada celda** de la matriz territorio. De este modo, cada hormiga se suspende en la variable condición asociada a la celda a la que quiere desplazarse (y está aún ocupada).

Inicialmente, se ha implementado un monitor utilizando la sincronización básica que ofrece Java mediante el *lock intrínseco* asociado a la clase Object. De este modo, los métodos del territorio llevan la etiqueta **synchronized** (que garantiza las operaciones de cierre y apertura del lock intrínseco al entrar y salir del método, respectivamente). Asimismo, se dispone de una única variable condición (implícita) y se emplea la operación *wait()* para suspender a un hilo en la condición implícita asociada al lock; y la operación *notifyAll()* para reactivar a todos los hilos suspendidos en dicha condición implícita.

## Código proporcionado

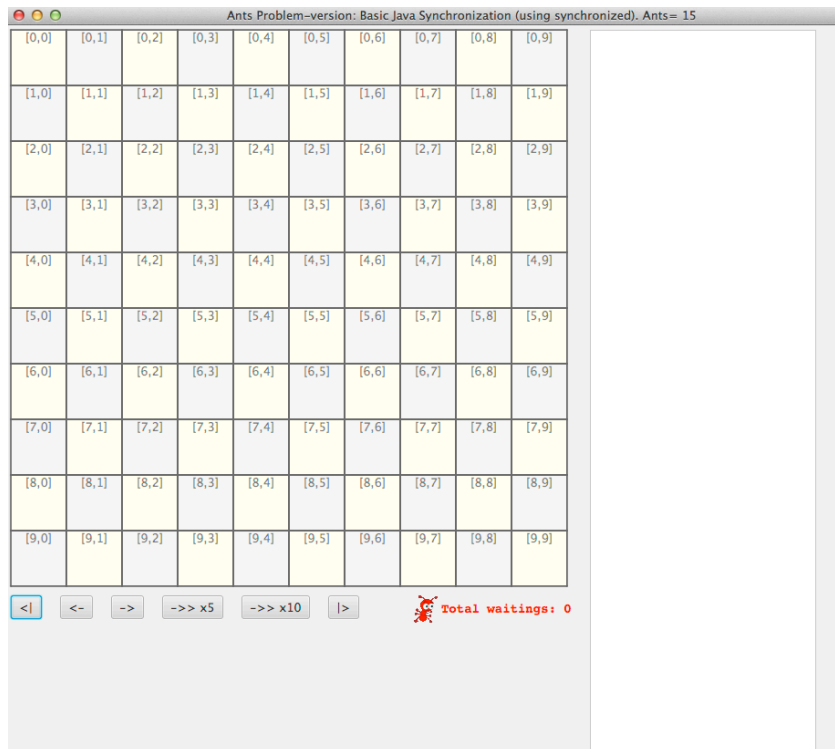
---

Puede descargar el código necesario para la práctica 3 desde el sitio Poliformat de la asignatura (carpeta "Recursos / Materiales para el laboratorio/ Práctica 3: Hormigas", fichero "TheAntsProblem.jar").

El código proporcionado soluciona los problemas de exclusión mutua y sincronización condicional, aunque puede presentar interbloqueos entre dos o más hormigas cuando se produce entre ellas una espera circular.







La clase *TheAntsProblem* contiene el método principal *main*, que podrá lanzar a ejecución sin argumentos. Desde un terminal, también puede ejecutar directamente la aplicación sin argumentos con la instrucción: `java -jar TheAntsProblem.jar`.

Al ejecutar la aplicación, se mostrará una pantalla similar a la siguiente figura:



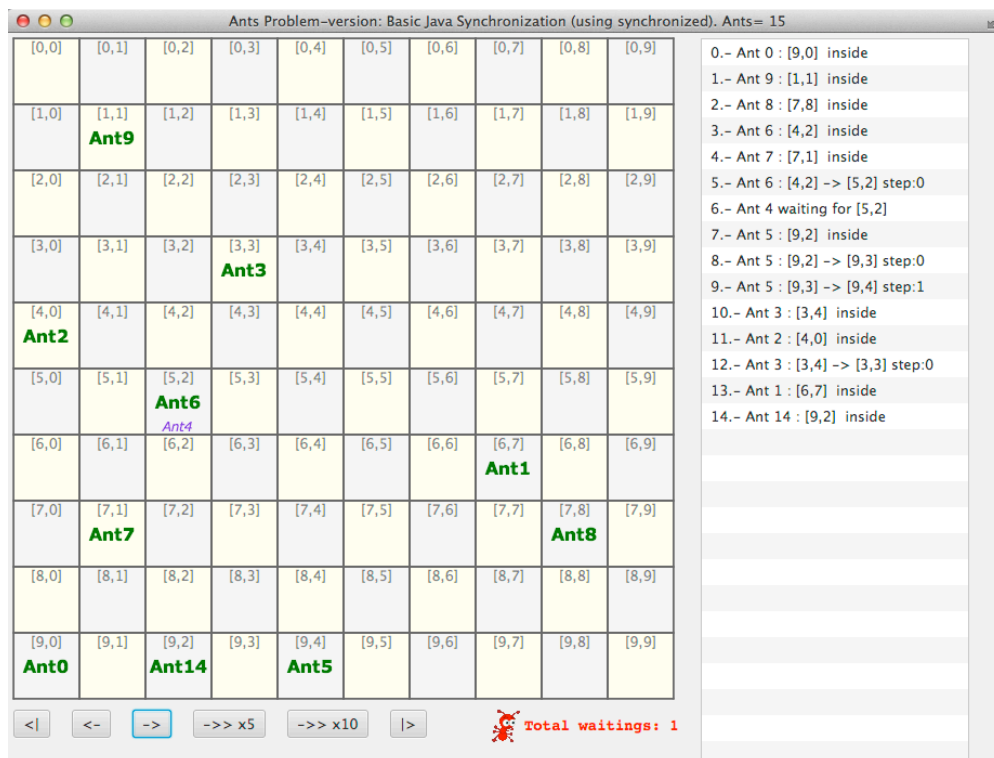
En la parte izquierda se muestra el territorio, con cada una de sus celdas. En la parte derecha (inicialmente en blanco) se irá mostrando la ejecución del problema.

Bajo el territorio disponemos una serie de botones que nos permitirán avanzar o retroceder en la ejecución del problema. Estos botones son:

-  Para ir al inicio de la ejecución.
-  Para retroceder un paso de la ejecución.
-  Para avanzar un paso de la ejecución.
-  Para avanzar 5 pasos de la ejecución.
-  Para avanzar 10 pasos de la ejecución.
-  Para ir al final de la ejecución.

Además, el mensaje "Total waitings" nos irá mostrando el total de esperas (*waitings*) que realizan las hormigas en cada paso de la ejecución. Recordemos que una hormiga tendrá que esperar si la celda a la que desea ir está ocupada.

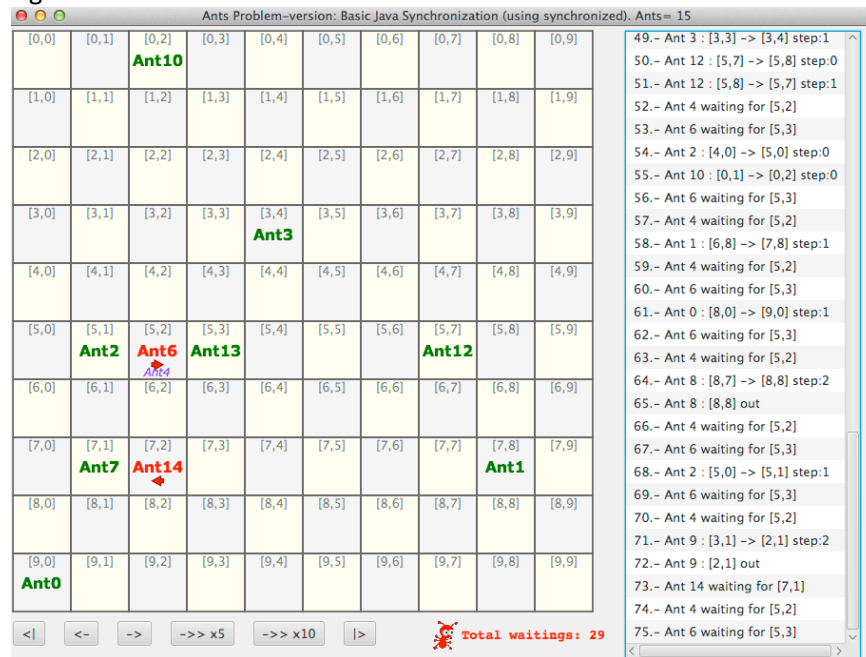
Cuando avanzamos en la ejecución, obtenemos una pantalla similar a la siguiente figura:



Como hemos comentado, en la parte derecha se van mostrando los pasos realizados en la ejecución. En concreto, en este ejemplo se han realizado ya 14 pasos.


- La línea **0.- Ant 0: [9,0] inside** indica que, en el paso 0, la hormiga 0 se ha situado en el territorio en la celda [9,0] y comienza su ejecución. Por cada hormiga, se tendrá una única línea de este tipo, con el mensaje "inside". Además, podemos ver que la hormiga **Ant0** se ha colocado correctamente en el territorio en la celda correspondiente (está de color **verde**).
- La línea **5.- Ant 6 [4,2] -> [5,2] step:0** indica que, en el paso 5, la hormiga 6 se ha desplazado de la celda [4,2] a la celda [5,2], en su movimiento (*step*) número 0. Como hemos comentado antes, cada hormiga realizará un total de 3 movimientos (desde *step 0* hasta *step 2*).
- La línea **6.- Ant 4 waiting for [5,2]** indica que la hormiga 4 está esperando a que la celda [5,2] quede libre para colocarse en ella. Esta hormiga todavía no ha sido colocada en el territorio, pero casualmente su casilla inicial está en estos momentos ocupada. Por tanto, debe esperar. Esta situación se ha representado en el territorio con la hormiga en color **morado** y un texto más pequeño.
  - Importante: la hormiga 4 no está todavía en el territorio (todavía no tenemos el mensaje "inside"), por lo que no se incumple con la restricción impuesta al territorio.

Al avanzar con la ejecución de este ejemplo, obtendremos una pantalla similar a la siguiente:



Podemos ver que la hormiga **Ant14** está remarcada en rojo en el territorio, pues está esperando a ir a la celda contigua (ocupada por otra hormiga). Lo mismo le ocurre a la hormiga **Ant6**.

- La línea **73.- Ant 14 waiting for [7,1]** indica que la hormiga 14 desea ir a la celda [7,1] pero dicha celda está ocupada. Se mostrará siempre en **rojo** el texto de una hormiga, situada ya en el territorio, que esté esperando que una determinada celda quede libre. Además, una flecha roja situada debajo de la hormiga nos indicará la celda concreta por la que está esperando.

Si avanzamos hasta el final de la ejecución del problema (por ejemplo, con el botón ) , podremos ver una pantalla similar a la siguiente:



Si toda la ejecución ha sido correcta, al final tendremos el mensaje de **"done"** y no habrá ya hormigas en el territorio, pues todas ellas han acabado sus movimientos y han salido del territorio. Por ejemplo:

- La línea **117.- Ant 4: [6,4] out** indica que la hormiga 4 ha terminado sus movimientos y libera la celda en la que estaba, saliendo así del territorio.

Además, en el mensaje "Total waitings" tendremos el número total de esperas (*waitings*) que han realizado las hormigas en este ejemplo. En este caso, 43 esperas.

No obstante, podría darse el caso de que dos o más hormigas se hubieran quedado bloqueadas entre sí con una espera circular, produciéndose un interbloqueo. En dicho caso, al llegar al final de la ejecución del problema obtendríamos un resultado similar al siguiente:



En este caso, veremos que al final no aparece el mensaje "done". Y que además en el territorio están las hormigas (en rojo) que han quedado interbloqueadas. Las flechas claramente nos muestran dicha situación.

## Análisis del código

El código proporcionado implementa distintas clases:

- Log, LogItem*: son "clases opacas", necesarias para el correcto funcionamiento de la aplicación, pero sin interés para el alumno. **No deben modificarse.**
- Ant*: clase que extiende de Thread y que implementa a la hormiga. En su método *run()*, la hormiga se sitúa en el territorio y realiza hasta 3 movimientos. En cada movimiento, se desplaza aleatoriamente a una celda contigua. Finalmente, la hormiga se elimina del territorio.
- TheAntsProblem*: es la **clase principal**. De esta clase interesa conocer cierto código del método *start*, en concreto:
  - La creación del territorio, de tamaño *tsize*. Esta variable es el primer argumento de la aplicación y permite valores de territorio de [2..12], siendo el valor por defecto 10.

```
int tsize=integer(arg,0,10,2,12);
// 1st arg: size of Territory (interval [2..12], default 10)
Territory t = new Territory(tsize,log);
```

- La creación y arranque de las hormigas. El número de hormigas se define en la variable *Nants*, que a su vez es el segundo argumento de la aplicación y permite valores de [2..30], siendo 15 el valor por defecto.

```
int Nants=integer(arg,1,15,2,30);
//2nd arg: number of ants (interval [2..30], default 15)
Ant[] ants = new Ant[Nants];
for (int i = 0; i < Nants; i++) {
    int x = (int) (rnd.nextDouble() * tsize); //X initial position
    int y = (int) (rnd.nextDouble() * tsize); //Y initial position
    ants[i] = new Ant(i, x, y, t, steps, barrier);
    ants[i].start();
}
```

- *Territory*: clase que actúa como un monitor y controla el acceso de las hormigas al territorio. Las actividades 1 y 2 indican qué debe actualizarse en esta clase.

## Actividad 0 (Sincronización básica en Java)

Lance varias ejecuciones del código proporcionado y complete la siguiente tabla, donde se indique para cada ejecución:

- el número total de esperas realizadas por las hormigas (*total waitings*)
- y la cantidad de hormigas que se han quedado al final bloqueadas.

Prueba	Total Waitings	Nº hormigas interbloqueadas
1	48	2
2	86	7
3	73	6
4	28	0
5	6	0
6	51	2

Indique, en promedio, cuántas esperas realizan en total las hormigas, cuando:

- a) La ejecución termina de forma adecuada, sin interbloqueos.

17

- b) Se produce un interbloqueo.

64 - 65

## Actividad 1 (Sincronización con ReentrantLocks en Java)

Implemente una nueva versión del problema, utilizando las herramientas **ReentrantLock** y **Condition** proporcionadas en la biblioteca *java.util.concurrent*, que aplique la variante de solución: "a) una **única variable condición asociada a todo el territorio**".

Para ello, sustituya en la clase **Territory** de forma apropiada el *lock intrínseco* asociado a la clase **Object** (y, por tanto, el uso de la etiqueta *synchronized*) por la utilización de un *ReentrantLock* con una única variable *Condition* asociada.

Además, en la clase **Territory** cambie la descripción del territorio por:

**String description="ReentrantLock with 1 Condition"**

Lance varias ejecuciones del nuevo código implementado y compare sus resultados con el código original, para condiciones similares del problema. Para ello, complete la tabla siguiente.

Prueba	Total Waitings	Nº hormigas interbloqueadas
1	74	2
2	34	2
3	102	6
4	95	6
5	104	8
6	52	5

Y calcule, en promedio, cuántas esperas realizan en total las hormigas, cuando:

a) La ejecución termina de forma adecuada, sin interbloqueos.

0

b) Se produce un interbloqueo.

76-77

**Cuestión 1.1:** Si utilizamos los mismos tamaños de territorio y número de hormigas, ¿podemos ver diferencias significativas entre utilizar el lock intrínseco (*synchronized*) o *ReentrantLock*? Por ejemplo, ¿se producen menos reactivaciones "innecesarias" de hilos?

No se pueden apreciar diferencias entre utilizar lock intrínseco o *ReentrantLock*. Se producen las mismas reactivaciones innecesarias de hilos ya que todas las hormigas esperan en una condición en ambos casos.



## Actividad 2 (Uso de varias variables Condition)

Implemente una nueva versión del problema, utilizando las herramientas **ReentrantLock** y **Condition** proporcionadas en la biblioteca *java.util.concurrent*, que aplique la variante de solución: "**b) una variable condición por cada celda de la matriz**".

Para ello, en la clase **Territory** utilice un único *ReentrantLock* asociado al territorio y cree tantas variables condición *Condition* como celdas tenga dicho territorio.

Además, en la clase **Territory** cambie la descripción del territorio por:

**String description="ReentrantLock with multiple Conditions"**

Lance varias ejecuciones del nuevo código implementado y compare sus resultados con el código de la Actividad 1, para condiciones similares del problema.

Para ello, complete la tabla siguiente:

Prueba	Total Waitings	Nº hormigas interbloqueadas
1	3	0
2	2	0
3	6	3
4	6	2
5	4	0
6	5	0

Y calcule, en promedio, cuántas esperas realizan en total las hormigas, cuando:

a) La ejecución termina de forma adecuada, sin interbloqueos.

3-4

b) Se produce un interbloqueo.

6

**Cuestión 2.1:** Si utilizamos los mismos tamaños de territorio y número de hormigas, ¿podemos ver diferencias significativas entre utilizar una única condición asociada a todo el territorio (Actividad 1) o bien una condición por cada celda (Actividad 2)? Por ejemplo, ¿se producen menos reactivaciones "innecesarias" de hilos?

Se pueden apreciar diferencias entre utilizar una condición para todo, a una por celda. Se producen menos reactivaciones innecesarias de hilos ya que las hormigas esperan en una condición a la celda a la que quieren entrar. (Podemos ver que el número de total waitings ha decrementado considerablemente debido a que antes se reactivaban todas las hormigas).

**Cuestión 2.2:** Si utilizamos una condición por cada celda, ¿podríamos obtener la siguiente secuencia de líneas por pantalla? ¿Qué representan cada una de las líneas con "waiting for"?

6.- Ant 4 : [9,4] inside  
7.- Ant 5 : [2,7] inside  
8.- Ant 4 waiting for [9,3]  
9.- Ant 6 : [7,9] inside  
10.- Ant 7 : [2,3] inside  
11.- Ant 7 : [2,3] -> [3,3] step:0  
12.- Ant 8 : [8,2] inside  
13.- Ant 9 : [3,6] inside  
14.- Ant 10 : [0,8] inside  
15.- Ant 11 : [2,1] inside  
16.- Ant 12 : [2,3] inside  
17.- Ant 4 waiting for [9,3]

18.- Ant 3 : [3,9] -> [2,9] step:0  
19.- Ant 4 waiting for [9,3]  
20.- Ant 6 : [7,9] -> [7,8] step:0  
21.- Ant 4 waiting for [9,3]  
22.- Ant 10 : [0,8] -> [1,8] step:0  
23.- Ant 4 waiting for [9,3]  
24.- Ant 1 : [6,0] -> [7,0] step:0  
25.- Ant 1 : [7,0] -> [7,1] step:1  
26.- Ant 14 : [9,3] -> [8,3] step:0  
27.- Ant 0 waiting for [3,3]  
28.- Ant 4 : [9,4] -> [9,3] step:0  
29.- Ant 1 : [7,1] -> [8,1] step:2  
30.- Ant 1 : [8,1] out  
31.- Ant 0 waiting for [3,3]

### Actividad 3 (Uso de barreras para la sincronización de hilos)

Implemente una nueva versión del problema, utilizando como base el código de la actividad anterior, para conseguir que todas las hormigas comiencen su ejecución a la vez. Para ello, utilice uno de los dos tipos de barrera (*CyclicBarrier* o *CountDownLatch*) que se ofrecen en la biblioteca *java.util.concurrent*.

En la clase *Territory* cambie la descripción del territorio por:

**String description="Using Barriers"**

**Cuestión 3.1:** ¿Qué tipo de barrera ha utilizado? ¿Qué ventajas le ofrece, frente al otro tipo de barrera?

Se ha utilizado *cyclic barrier*, permite reinicializar el valor N una vez este llegue a 0, además nos permite ejecutar código justo antes de que se abra la barrera.

**Cuestión 3.2:** Si utilizamos barreras conforme a lo indicado en esta actividad, ¿podríamos obtener la siguiente secuencia de líneas por pantalla? ¿Por qué?

```
0.- Ant 0 : [8,0] inside
1.- Ant 11 : [2,6] inside
2.- Ant 13 : [8,5] inside
3.- Ant 14 : [5,0] inside
4.- Ant 12 : [8,2] inside
5.- Ant 10 : [8,7] inside
6.- Ant 9 waiting for [5,0]
7.- Ant 12 : [8,2] -> [8,1] step:0
8.- Ant 7 : [0,8] inside
9.- Ant 6 : [3,1] inside
10.- Ant 14 : [5,0] -> [4,0] step:0
11.- Ant 5 : [7,2] inside
```

No, ya que ant 9 ya ha comenzado su ejecución al igual que ant 12, y aún faltan hormigas por entrar a la matriz.

WARNING: 

### Actividad 4 (Gestión de interbloqueos)

En todas las actividades anteriores podría presentarse en algún momento un problema de interbloqueo, si dos o más hormigas desean ocupar la misma celda. En esta actividad se pretende que el alumno proponga una solución para el problema de los interbloqueos entre las hormigas, de modo que se rompa alguna de las condiciones de Coffman, excepto la de exclusión mutua (i.e. las hormigas deben seguir respetando la restricción de no estar ambas en la misma celda del territorio).

Esta actividad se puede abordar de varias formas posibles. Aquí indicamos algunas guías de cómo abordarlo:

- Se podría hacer uso de métodos de tipo *wait(long timeout, int nanos)* o bien *await(long timeout, TimeUnit unit)* para esperar un máximo de tiempo.
- Se podría establecer un *lock* por cada celda y hacer uso de los métodos *tryLock()*, *tryLock(long timeout, TimeUnit unit)*, *isLocked()*... para determinar si una hormiga debe reconsiderar la celda a la que desea ir.

**Cuestión 4.1:** La solución adoptada, ¿qué condición o condiciones de Coffman rompe?

En caso de usar *tryLock* estaríamos rompiendo la condición de retención y espera, ya que al retener un el uso de una casilla, no se espera para que otra casilla este vacía, se opta por comprobar si está vacía y reconsiderar la celda a la que desea ir, de esta manera también se rompe la espera circular.

En caso de usar *wait/await* con timeouts estaríamos rompiendo la condición de retención y espera, ya que al retener un el uso de una casilla, se espera por un tiempo limitado para que otra casilla este vacía, de esta manera pasado el timeout se comprueba si está vacía y reconsiderar la celda a la que desea ir, de esta manera también se rompe la espera circular.