Prácticas Concurrencia y Distribución (16/17)

Arno Formella, Francisco Rodríguez Martínez,
Javier Rodeiro Iglesias, David Olivieri

8 de mayo de 2017

Introducción

Según la guía docente las prácticas se evaluan con los siguientes aspectos:

(P3) Informes: Elaboración de informes (según una guia) que recogen los principales desarrollos y resultados obtenidos por el/la estudante. Partes de los informes se elaboraran en pequeños grupos.

(P4) Pruebas prácticas: Demonstración de los desarrollos e implementaciones de las tareas de programación y experimentos de estudio.

(P5) Resolución de problemas: Elaboración de algoritmos y análises con cierto nivel de formalismo para comprobar la corrección y estudios de rendimiento.

(P6) Presentaciones: Breves presentaciones orales con medios audiovisuales de desarrollo y resultados obtenidos por el/la estudiante.

Eso se traduce en la realidad de este curso a:

(P3) Informes

Informes de realización de las prácticas a entregar en pequeños grupos de 1 o 2 personas con posibles variaciones/sugerencias como puedan surgir durante las clases presenciales por la tutoría del profesor.

Un informe deberá constar de y cumplir con:

- Una identificación clara de sus autores (nombres completos y DNIs).
- Una sección para cada apartado y subapartado que se deba incluir en el informe, siguiendo la misma numeración que la de los apartados de los anuncios de las prácticas, si procede.
- Un informe no debe superar las tres (4) páginas.
- Los informes se entregan a través de la plataforma FaiTIC/TEMA en los plazos que ahí se indican.

(P4) Pruebas prácticas

Respecto a las tareas de las prácticas:

- Se debe entregar un archivo comprimido con todos los ficheros del código fuente que fueron elaborados para realizar las prácticas con su documentación embebida como se genera con una herramienta adecuada (como por ejemplo `doxygen`).
- Dichas entregas se realizan semanalmente como está indicado en los boletines.
- El profesor mirará durante las prácticas los desarrollos realizados y evalua las competencias adqueridas en el diálogo con los estudiantes.

(P5) Resolución de problemas

Se distribuye problemas que se pueden resolver en el plazo fijado con la entrega del informe correspondiente.

(P6) Presentaciones

Se ofrece la posiblidad de realizar pequeñas presentaciones sobre el contenido y los resultados de las actividades previamente acordados con el profesor en clases prácticas (10 min) y se evaluan las competencias correspondientes.

Evaluación y competencias

Los informes se evaluarán y computarán dentro del apartado de realización de informes y memorias de prácticas de la materia. Para la evaluación del informe se tendrán en cuenta la corrección del código elaborado en cada apartado, su claridad, idoneidad, eficiencia y buen diseño, control de errores y excepciones, finalización correcta, comentarios, modularidad; y las respuestas a las preguntas planteadas y mediciones realizadas (completitud y corrección de las respuestas, número e idoneidad de los ejemplos aclaratorios, número y adecuación de las pruebas realizadas, realización de las cuestiones opcionales, número de máquinas utilizadas para hacer pruebas si es el caso, adecuación y completitud de gráficas y datos presentados, conclusiones obtenidas etc.).

Las prácticas de este curso están organizados en 5 actividades principales que están a su vez divididos en apartados que se deben realizar **semanalmente** en grupos de **una** o **dos** personas con entregas **individuales** al final de las horas de clase práctica.

Las entregas se realizan mediante la herramienta FaiTIC que estará configurada para permitir la subida de ficheros durante la segunda mitad de la clase práctica. Una entrega sirve al mismo tiempo como **testigo de asistencia** a clases prácticas. No obstante el profesor de prácticas puede usar otras medidas adicionales para monitorizar dicha asistencia.

Las entregas consisten en la subida de un **único** fichero simple o de tipo archivo (`.zip`, `.rar`, `.tgz`, etc.). Dichos ficheros **siempre** tendrán nombres que se forman de la siguiente manera:

`Apellido1_Apellido2_Nombre_Grupo_Fecha.Extensión`

donde el `Apellido1`, `Apellido2`, y `Nombre` se entienden como tales, el `Grupo` es uno de `CDI1` hasta `CDI5`, la fecha se indica en formato mes y día (MMDD), es decir, el 30 de enero será: 0130, y la extensión según tipo de fichero. Es decir, si Fran sube un fichero java el lunes, 30 de enero de 2017, este fichero se llamaría:

`Rodríguez_Martínez_Francisco_Javier_CD3_0130.java`

Ficheros con nombres que no cumplen con esta nomenclatura serán simplemente **ignorados**.

1. Actividad: Introducción a la concurrencia en Java

Objetivos: Adquirir conocimientos básicos sobre la forma como está implementada la concurrencia en Java, los métodos básicos para crear hilos, uso de esperas programadas, medición de tiempo de ejecución, sincronización simple.

Metodología: Esta actividad (que sigue con más tareas en las siguientes semanas) se realizará en horas de prácticas presenciales y en horas no presenciales. Al tratarse de la primera parte de esta actividad, además de las tareas descritas, se llevarán a cabo tareas de familiarización con las herramientas, la documentación de Java y métodos de trabajo.

Material adicional: Son de especial interés los siguientes enlaces

- {`http://docs.oracle.com/javase/7/docs/`}
- {`http://docs.oracle.com/javase/8/docs/`}
- {`http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html`}
- {`http://www.stack.nl/~dimitri/doxygen/index.html`}
- {`http://en.wikipedia.org/wiki/Markdown`}

Requisito general a todos los programas en todas las actividades es: siempre termina el programa con un mensaje como *Program of exercise X has terminated*, es decir, todos los componentes del programa concurrente terminan correctamente su ejecución.

Las preguntas que aparecen intercalados en los anunciados tienen como objetivos: animar a la reflexión y al auto-aprendizaje, servir como ejemplos de posibles preguntas en la fase de evaluación (examen), fundamentan la base para los breves informes que se entregan para las actividades (más adelante hacia final de cada actividad).

1.1. Introducción a los hilos en Java

1. Examina las dos formas que provee Java para crear un hilo: la clase `Thread` y la interfaz `Runnable`.

2. Utiliza ambas formas para crear un programa que cree y ejecute un hilo que imprima en pantalla un mensaje como *Hello world, I'm a java thread*.

   ¿Hay alguna diferencia de funcionamiento entre ambas formas? ¿A nivel de diseño, cuál te parece preferible, y por qué?

   Es decir, realiza dos programas: `Thread.java` y `Runnable.java`.

3. Modifica tu programa para que el hilo tarde aproximadamente un segundo en mostrar el mensaje. ¿Qué método usarás para ello?

   Es decir, realiza un programa: `Segundo.java`.

4. Cuando arrancas un hilo desde el programa principal, ¿cuántos hilos hay activos? ¿Qué métodos y herramientas tienes disponibles para averiguarlo?

   Saca una lista de los hilos activos por pantalla.

   Es decir, realiza un programa: `Activos.java`.

1.2. Creación de múltiples hilos

1. Escribe un programa en Java que mediante parámetros de línea de comando reciba cuantos hilos se debe crear. El programa creará y ejecutará el número de hilos indicado. Cada hilo debe imprimir en pantalla un mensaje como *Hello, I'm thread number X*, y después de uno o varios segundos (puedes usar un segundo argumento via línea de comando), un mensaje como *Bye, this was thread number X*, siendo *X* el valor de un contador que se va incrementando con el número de hilos creados. Experimenta con diferentes argumentos, incluso yiendo a valores grandes. ¿Las salidas del programa reflejan lo que has esperado?

   Es decir, realiza un programa: `Hilos.java`.

1.3. Sincronización completa de hilos

1. Como seguramente has observado en los programas anteriores, sin una sinconización explicita al final del programa no se consigue que el último mensaje del programa sea el mensaje final del hilo principal.

   Aumenta tu código para que el hilo principal se sincroniza con los hilos lanzados una vez que ellos hayan terminado su método `run` (pista: método `join()`).

   ¿Hay otras posibilidades para este tipo de sincronización?

1.4. Medición de tiempo de ejecución

1. Queremos medir el tiempo que tardan en ejecutarse todos los hilos que se crean, es decir, medir el tiempo de ejecución (en tiempo real o tiempo de reloj en la pared) de todo el programa.

   Aumenta tu(s) programa(s) anterior(es) para que muestre(n) este tiempo. Busca y encuentra métodos para tal fin.

   ¿Cómo debemos hacer para asegurarnos que la medición del tiempo de ejecución es fiable? es decir, que realmente se mide el tiempo desde que se arranca el primer hilo hasta que todos los hilos hayan finalizado. ¿Qué errores de medida de tiempo pueden ocurrir, si no nos aseguramos de que todos los hilos han acabado?

2. Realiza un diagrama que visualiza las *vidas* de todos los hilos de tu programa, es decir, desde el punto de *arranque* hasta el punto de *finalización*.

   Clasifica y describe las fases que observas.

3. Trata de modificar el programa para poder distinguir entre *tiempo de creación de hilos*, *tiempo de ejecución de los hilos* y *tiempo de sincronización final de los hilos*.

   ¿Es fácil distinguir los tiempos de las diferentes fases? ¿Qué consecuencias tiene esto a la hora de diseñar y evaluar soluciones concurrentes a un problema?

4. Modifica tu programa multi-hilo para que los hilos o bien muestren algo por pantalla o bien que hagan unas operaciones matemáticas suficientemente complejas basicamente internas de la CPU. Distingue el modo de ejecución mediante un parámetro en línea de comando. La ejecución del programa por prueba debe durar unos segundos para obtener mediciones interpretables, es decir, realiza bucles con un número de iteraciones suficiente.

   Pruébalo con un número creciente de hilos para conseguir finalmente una función de tiempo de ejecución en relación al número de hilos trabajadores (y modo de ejecución). Genera los diagramas correspondientes.

   ¡Es interesante que antes de realizar las mediciones intentes predecir los resultados por lo menos cualitativamente y averiguar si tu predicción coincide con los resultados medidos!

   ¡Interpreta y razona detenidamente sobre los gráficos que obtienes!

   ¡Es interesante realizar las mediciones en diferentes entornos, respecto a hardware, sistema operativo, y máquina virtual de Java!

2. Actividad: Concurrencia simple

Objectivos: Adquirir conocimientos básicos como distribuir el trabajo por realizar entre diferentes hilos para mejorar el rendimiento del sistema.

En esta práctica realizamos simples operaciones sobre imágenes en tonos de gris. Un ejemplo, como leer y escribir tales imágenes se encuentra en la página web acompañante del curso ({`http://formella.webs.uvigo.es/doc/cdg16/Gray.java`}). Se puede generar documentación simple con "doxygen" y el fichero de configuración (`http://formella.webs.uvigo.es/doc/cdg16/Doxyfile`).

2.1. Uso de hilos para operaciones sobre arrays bidimensionales

1. Escribe un programa que lee una imagen en tonos de gris en un array bidimensional y que escriba dicho array de nuevo como imagen (con otro nombre en el sistema de ficheros).

2. Implementa un algoritmo de binarización, es decir, que convierte la imagen de tonos de gris en una imagen blanco y negro tomando la decisión con un simple umbral (parámetro de línea de comando del programa).

3. Implementa tu algoritmo de binarización para que el trabajo se divide en tareas iguales entre cierto número de hilos (parámetro de línea de comando del programa).

4. Mide con imágenes de diferentes tamaños y diferente número de hilos el tiempo de cálculo. Realiza los diagramas/tablas correspondientes.

2.2. Distribución de trabajo entre hilos.

1. Escribe un programa concurrente que aplica un filtro para suavizar una imagen en tonos de gris, es decir, para obtener el valor de un píxel en posición (i, j) en la imagen resultante J se aplica

$$J(i,j) = \frac{1}{(2f+1)^2} \sum_{k=-f}^{f} \sum_{l=-f}^{f} I(i+k, j+l)$$

   donde I es la imagen original y f es el tamaño del filtro. Nota: si $I(i+k, j+l)$ resulta en un píxel fuera de la imagen, refleja las coordinadas en el borde, es decir, asume, por ejemplo, $I(-2,-1) = I(2,1)$, e igual a lo largo de los demás bordes.

   El tamaño del filtro f y el número de hilos participantes se debe pasar por línea de comando.

   Realiza diagramas de tiempo de ejecución para diferentes tamaños de filtros y creciente número de hilos.

2. Experimente con diferentes estratégicas para distribuir los píxeles por tratar entre los hilos:

   - por filas,
   - por columnas,
   - por bandas horizontales,
   - por bandas verticales.

   ¿Observas diferencias en el tiempo de ejecución?

3. Combina tu programa de filtrado con el programa de binarización del apartado anterior, es decir, consigue que tu programa lea una imagen tonos de gris, que lo filtre con un filtro de cierto tamaño, que lo binarice con cierto umbral.

2.3. Operación de reducción

1. Aumento tu programa del apartado anterior para que cuente los píxeles en negro de forma concurrente. Realiza las mediciones de tiempo oportunas.

3. Exclusión mutua para operaciones simples.

1. En clase de teoría vimos un algoritmo simple de multiplicación basado en sumación iterativa.

   Implementa el algoritmo de tal manera que se puede ejecutar de forma concurrente con cierto número de hilos participantes.

   Usa diferentes técnicas para garantizar la ejecución de las operaciones críticas con exclusión mútua (por ejemplo: `AtomicInteger`, `LongAdder`, y métodos sincronizados).

   Realiza las mediciones correspondientes con un números creciente de hilos para generar diagramas de tiempo de ejecución comparables en si (asegúrate tener suficiente carga de trabajo para que las mediciones se realizan adecuadamente).

   Describe tus observaciones.

2. ¿Estás seguro que tu algoritmo funciona correctamente?

   Haz pruebas con $p = 1$ y q relativamente grande, por ejemplo 50.000.000 y un número de hilos moderado, alrededor de 6.

   Modifica tu código para que el resultado sea correcto en todos los casos.

3. Modifica el programa de tal manera que los hilos no comparten la variable q, es decir, que cada hilo antemano sepa cuantas veces tiene que sumar (por ejemplo, el hilo principal le pasa a cada hilo el valor correspondiente durante la construcción).

   Mide de nuevo el tiempo de ejecución con las diferentes versiones de sincronización. ¿Es más rápido?

4. Modifica el programa de tal manera que tampoco el resultado r esté compartido.

   Mide de nuevo el tiempo de ejecución ya que ahora no hace falta ninguna sincronización cuando se modifica q o r. ¿Es más rápido?

4. Exclusión mutua sobre un objeto totalmente sincronizado

En esta actividad queremos implementar sincronización entre hilos. Usamos la metáfora de una biblioteca (el programa principal) con un libro (el objeto sincronizado) y varios lectores (los hilos participantes).

Considera los siguientes requisitos:

- Se pasa al programa 3 parámetros: número de lectores, número de páginas del libro, tiempo de lectura por página.
- El programa principal crea el libro con p páginas y un tiempo t (milisegundos ya que usaremos `sleep()`) de lectura por página.
- Los hilos lectores tienen un número de identificación, un tiempo de espera entre lecturas, y una referencia al libro por leer en común.
- El programa principal crea todos los lectores, lanza su actividad, y sincroniza con todos los hilos creados (bucle de `joins`).
- El objeto libro es totalmente sincronizado e internamente cuenta las páginas ya leídas. El libro tiene un método que indica, si quedan páginas por leer.
- El método de lectura de una página del libro debe imprimir en pantalla qué hilo está leyendo qué página (por ejemplo: implementa un `Read(int idHilo)`).

Una vez elaborado la estructura principal del programa, considera las tareas siguientes:

1. Implementa en el método `run()` de los hilos lectores un bucle que lee el libro página por página (llamando al método `Read()` de la clase `Book`) y aposteriori espera un tiempo de $t/(2n)$ (donde t es el tiempo de lectura usado en el método `Read()` y n es el número de hilos lectores participantes).

   El bucle termina cuando el libro indica que ya se han leído todas sus páginas.

   Lanza el programa por ejemplo con 10 hilos, 200 páginas y 20 milisegundos de tiempo de lectura por página.

   Una persona novata en el tema argumenta lo siguiente:

   Dado que el primer hilo que se lanza está leyendo la primera página del libro durante bastante tiempo, todos los demás hilos se ponen a la espera (`Read()` sincronizada) de leer la suya. Cuando le toca al segundo lector, lee otra vez tan lento que el primero ya está de nuevo a la espera de leer su siguiente página. Y así seguidamente con todos los lectores.

   Entonces el progama tiene obviamente el siguiente comportamiento: en el orden como el hilo principal finalmente lanza los hilos, éstos leen siempre en el mismo orden cada uno una página. Por eso al final del progama cada hilo habrá leído el mismo número de páginas y se han leído exactamente todas las páginas del libro.

   ¿Qué opinas sobre este ingenuo? Lo confirmas? Haz las pruebas! Abajo unas pistas como aumentar el programa.

2. Aumenta tu método `run()` de los lectores para que cada uno cuente el número de páginas que ha leído y que imprima este número con el mensaje de terminación.

3. Implementa la espera con el `sleep()` de tal manera que se llama a la función solamente si el tiempo de espera es más grande que zero. Lanza entonces el programa también con un valor zero de tiempo de lectura ($t = 0$).

4. Intercambia en el bucle del método `run()` las líneas tal que primero se espera y luego se lee la página del libro. Observa que pasa al final del programa!

Seguimos con la actividad de la metáfora de la lectura concurrente de un libro. Añadimos los siguientes requisitos a los ya existentes:

- Se garantiza que se lee cada página exactamente una vez.
- Los lectores leen las páginas en turnos, es decir, lector 1 lee página 1, lector 2 lee página 2, ..., lector n lee página n, y se empieza con lector 1 otra vez hasta que se haya acabado el libro.
- El libro gestiona internamente a cual de los lectores toca el turno de leer una página.

Aprovechando de la estructura principal del programa de los apartados anteriores, considera las tareas siguientes:

5. Modifica el método `Read()` de la clase `Book` para que devuelva un booleano verdadero si se ha podido leer una página, sino, es decir, que no haya más páginas, que devuelva falso.

6. Modifica el correspondiente recuento de las páginas leídas en el método `run()` de los hilos.

7. Aumenta la clase del libro para gestionar los turnos, por ejemplo, que sepa cuantos lectores hay, que haya un método `MyTurn(id)` que devuelve si el lector con la `id` debe leer, y que haya un método `NextTurn()` que prepara el turno para el siguiente lector.

8. Usa en el bucle del método `run()` de los lectores los métodos `wait()` and `notify()` respectivamente `notifyAll()` de tal manera que un hilo vuelve a esperar si no es su turno cuando le hayan despertado.

9. Realiza un gráfico de los tiempos de ejecución manteniendo el número de páginas por leer fijo pero incrementando el número de lectores. Para tal fin, no realices ninguna llamada a ningún `sleep()` en el programa.

Seguimos con la actividad de la metáfora de la lectura concurrente de un libro.

Con los experimentos (y el gráfico) de la semana anterior se ha observado que la lectura de un libro con cierto número de páginas realizada con un número creciente de lectores cada vez consume más tiempo si se usa tanto el método `notify()` como el método `notifyAll()`.

Obviamente este comportamiento se observa porque se despiertan muchos lectores a los cuales todavía no les toca su turno. Debe ser posible modificar el programa de tal manera que se despierta con un solo `notify()` exactamente el lector correspondiente a la página por leer. De esta mánera el tiempo de ejecución del programa debe ser más o menos constante independientemente del número de lectores participantes.

Entonces:

10. Modifica tu programa para que ya no contenga ningún `sleep` y ninguna salida a la consola (menos las dos de comienzo y terminación del hilo principal), mejor dependiendo de una variable booleana para que se pueda activar/desactivar este comportamiento mediante parámetro por línea de comando.

11. Genera de nuevo el gŕafico del tiempo de ejecución manteniendo el número de páginas muy alto (por ejemplo 500.000) y aumentado el número de lectores hasta 1000 (por ejemplo usando la secuencia 3 5 10 30 50 100 300 500 1000).

12. Modifica el programa para conseguir el objetivo que un `notify()` despierta exactamente el siguiente hilo por actuar. Activa las salidas del programa y verifica que sigue la lectura correctamente según los requisitos establecidos.

13. Suprime de nuevo las salidas y esperas y genera el gŕafico del tiempo de ejecución manteniendo el número de páginas muy alto (por ejemplo 500.000) y aumentado el número de lectores hasta 1000 (por ejemplo usando la secuencia 3 5 10 30 50 100 300 500 1000).

    ¿Consigues la mejora en tiempo de ejecución que se puede esperar?

Seguimos con la actividad de la metáfora de la lectura concurrente de un libro.

14. Seguramente has notado que realizar la tarea de tal manera que un lector despierta con `notify()` exactamente al siguiente lector que debe leer a continuación y que debe estar en su `wait()` provoca una posible condición de carrera, es decir, que se realiza tal `notify()` antes de que el otro hilo haya entrado en su `wait()`.

    ¿Cómo puedes resolver este problema?

    Inténtalo de las siguientes dos maneras:

    - Usa el estado del monitor de Java (`getState()`) como condición para estar seguro que el hilo ya haya entrado en espera antes de efectuar el `notify()` correspondiente. (Este método funcionará pero el manual de Java dice que no se debe usar este método para el fin de sincronizar hilos.)

- Implementa tu propia condición que puedes chequear, es decir, el hilo que está a punto de ejecutar su `wait()` pone la condición de forma adecuada, el hilo que ejecuta el `notify()` espera de forma adecuada hasta que se haya cumplido la condición. Observa que la condición debe estar implementada en el mismo objeto que usas para tu `notify/wait`-pareja para que esté protegida por los bloques sincronizados correspondientes. (Observerás que necesitas un bucle de espera activa del hilo que debe notificar.)

15. ¿Funcionan las versiones de tu programa también para el caso de un solo hilo? (Observa que en los requisitos este caso nunca fue explícitamente excluido). ¿Qué solución propones?

5. Una aplicación simple distribuida

Retomamos la actividad 2 donde implementamos concurrencia simple para trabajar con una imagen. En vez de lanzar hilos en la misma máquina queremos usar o bien otros procesos en el mismo ordenador o bien otros procesos en otro(s) ordenador(es). Los requisitos con más detalle son:

1. Se implementa un sistema cliente-servidor.
2. El servidor trabaja con una estructura de datos para las tareas. Las tareas se distribuyen a los clientes cuanto estos realizan peticiones.
3. Con los resultados recibidos como respuestas de los clientes el servidor compone la imagen resultante.
4. Para cada cliente que se conecta al servidor se usa en el servidor un hilo para realizar la gestion, es decir, para mandar la tarea y esperar la respuesta.
5. Se transmite un objeto desde el servidor al cliente para indicar los parámetros para calcular (básicamente la región de la imagen). Se devuelve un objeto desde el cliente al servidor con el resultado.

Con estos requisitos transforma el programa concurrente de la actividad 2 que realizaba la binarización en un programa distribuido con la siguiente ayuda:

1. El servidor se preocupa de la lectura y escritura de la imagen (igual como ya realizado en la actividad 2).
2. En vez de crear los hilos trabajadores en un bucle fijo, se espera para que se conecten los clientes y se crea para cada cliente el hilo que se dedica a la comunicación.
3. La comunicación entre servidor y clientes se realiza con `sockets` y flujos sobre estas conexiones tipo `ObjectOutputStream` y `ObjectInputStream`.
4. El objeto que se transmite al cliente contiene los parámetros de la región sobre la cual este cliente debe actuar, es decir, coordinadas, zona de la imagen, y umbral.
5. El objeto que se transmite al servidor contiene en la región los valores del resultado de la operación que el hilo de comunicación copia a la imágen de resultado.
6. Para realizar la sincronización al final, es decir, que el servidor determina que todos los hilos trabajadores respectivamente clientes, hayan terminado se usa un `CountDownLatch` de forma adecuada. (Recomendación: usa el `CountDownLatch` primero en una modificación de la activadad 2 para sustituir el uso de `join`.)

Seguimos con la aplicación cliente-servidor. Hasta ahora se ha pedido solamente la realización de un solo *trabajo* (región de cómputo) al cliente una vez conectado. Como extensión se pide:

7. El cliente trabaja en bucle, es decir, sigue pidiendo sobre la misma conexión nuevos trabajos hasta que el servidor indica que no haya más trabajo (por ejemplo, mediante un objeto con parámetros de la región especiales).

8. Añade también en el cliente una espera artificial configurable (ya que nuestro cómputo es simple) para simular un trabajo más extenso del cliente. Observa el comportamiento de tu programa. ¿Sigue funcionando correctamente?

Seguimos con la aplicación cliente-servidor. Como último aspecto para tratar nos concentramos en el problema de fiabilidad del cliente y la terminación del programa. Como extensión se pide:

9. Implementa la espera artificial por trozo de trabajo en el cliente con un valor aleatorio adecuado.

10. Implementa que el servidor remanda los trabajos a otros clientes que pidan más trabajo en caso que no haya recibido resultados para alguna tarea. Si varios clientes devuelven un resultado o bien descarta los a mayores o bien sobreescribe el primero (no importa ya que debe ser lo mismo).

11. Cuando el servidor haya recibido todos los resultados del trabajo distribuido, puede terminar. Asumiendo que no haya fallos, ¿cómo implementarías que los clientes que todavía trabajan (obviamente por duplicidad de encargo) terminan ordenadamente (y no por ruptura de comunicación)?

    No se pide una implementación, pero un razonamiento sobre un diseño.