Programación de Redes y Servicios
Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación
Universidad de Zaragoza, curso 2019/2020

Práctica 5: **Paso de Mensajes Asíncrono en Java**

Objetivos y requisitos

Objetivos

- Entender las particularidades de la programación concurrente mediante paso de mensajes.
- Resolver y programar problemas de complejidad media mediante el paradigma de paso de mensajes.

Requisitos

- Compilador y máquina virtual de Java (JVM) Java Platform Standard Edition 7[1]
- `Eclipse`, como entorno de desarrollo[2].

Contexto de la práctica

En las prácticas anteriores desarrollasteis un componente que permite que dos procesos (sean concurrentes o distribuidos) se comuniquen mediante el paso de mensajes asíncrono (véase la Figura 1). En esencia, el componente proporciona dos métodos –`send` y `receive`– que permiten enviar y recibir mensajes, respectivamente. La semántica de estas dos operaciones es asíncrona, de manera que la operación `send` nunca es bloqueante (un proceso P al invocar `send` envía el mensaje al destinatario y P continúa su ejecución); mientras que la operación `receive` comprueba si hay mensajes en el buzón, y en caso de no haberlos deja al proceso invocante a la espera de recibir al menos un mensaje. En esta práctica vamos a utilizar dicho componente para el desarrollo de programas concurrentes

[1] Disponible para descarga en `http://www.oracle.com/technetwork/java/javase/downloads/index.html`.
[2] Disponible para descarga en `http://www.eclipse.org/downloads/`

```
public class MessageSystem {

      private int pid;
      private ArrayList<InetSocketAddress> peers= new ArrayList<InetSocketAddress>();
      private MailBox mailbox;
      private Sending sending;
      boolean tcp=true;

      public MessageSystem(int source, String usersFile, boolean TCP) throws
FileNotFoundException { … }
      public void send(int dst, Serializable message) { … }
      public Envelope receive() { … }
      public void stopMailbox() { … }
      private InetSocketAddress loadPeerAddresses(String usersFile) throws
FileNotFoundException { … }
…
```

Figura 1: Componente en Java para el *Paso de Mensajes Asíncrono*.

(y distribuidos) mediante el paradigma de paso de mensajes (asíncrono). Por tanto, vamos a considerar el componente como una abstracción sobre la que elaboraremos construcciones más complejas.

1. El problema de las costureras

Como primer problema en esta práctica, vamos a implementar el problema de las costureras de la práctica 2 que ya conocéis, pero esta vez lo resolveremos mediante el paso de mensajes.

Ejercicio 1.

Un taller de costura, que se dedica a confeccionar jerséis, ocupa a tres personas. Una persona está **continuamente** fabricando mangas, que deposita en un cesto de capacidad limitada. Cuando el cesto está lleno de mangas, la costurera deja de coser hasta que haya hueco libre. Otra persona está **continuamente** fabricando los cuerpos de los jerséis, que también deposita en su correspondiente cesta de capacidad limitada. Ambas personas no fabrican nuevas mangas o cuerpos si su cesta está llena, esperando a que haya algún hueco para continuar trabajando.
Finalmente, existe una tercera costurera, encargada de ensamblar mangas y cuerpos de jerséis. Para cada jersey, necesita dos mangas (de la cesta de mangas) y un cuerpo (de la cesta de cuerpos). Esta persona no fabricará un jersey si le faltan las piezas necesarias para hacerlo.

Puntuación: 4 puntos

Este ejercicio puede entregarse. Diseña un programa en Java (concurrente o distribuido) de manera que se cumplan las condiciones de sincronización anteriores, mediante paso de mensajes y utilizando vuestro `MessageSystem`*. Implementa los códigos fuente necesarios y guárdalos en un directorio (carpeta) de nombre* `p05e01`*. Recuerda: en este directorio guarda únicamente los códigos fuente (ficheros con extensión* `.java`*), no los ficheros objeto (extensión* `.class`*).*

2. Arquitecturas de Sistemas Distribuidos

En la práctica, los sistemas distribuidos se construyen atendiendo a requisitos funcionales (*qué debe de realizar el sistema*) y requisitos no funcionales (*cómo debe realizarlo y cuáles son sus prestaciones o coste*). En los sistemas distribuidos actuales, es muy habitual que los requisitos no funcionales se materialicen en un acuerdo (o contrato formal entre el cliente y la empresa que proporciona el servicio) denominado *acuerdo del nivel del servicio* (*Service Level Agreement*, SLA, en inglés). Desde un punto de vista cuantitativo, el SLA se especifica en términos de métricas o indicadores como por ejemplo el tiempo de ejecución total, el *throughput* (número de tareas ejecutadas por unidad de tiempo), el consumo energético máximo o el coste económico.

Durante la ejecución (*runtime*, en inglés), los indicadores del SLA forman parte de un conjunto de variables denominadas *calidad del servicio* (*Quality of Service*, QoS, en inglés). En este ejercicio vamos a hacer uso de un único indicador: el tiempo de ejecución. Además, vamos a estudiar la escalabilidad de los sistemas distribuidos y cómo las distintas arquitecturas software que hemos estudiado pueden ayudar a mantener el QoS. En esencia, el QoS depende de 3 factores:

- de la carga de trabajo (es decir, las tareas que hay que procesar);
- de los recursos computacionales disponibles (CPU, red de comunicación y almacenamiento); y
- de cómo se distribuyen las tareas entre los recursos computacionales.

Para este ejercicio, considera el esquema para un servidor que calcula números perfectos mostrando en el Código 3. Un *número perfecto* es un número natural que es igual a la suma de sus divisores propios positivos. Por ejemplo, el número 6 es el primer número perfecto, dado que sus divisores propios son 1, 2 y 3 ($1 + 2 + 3 = 6$). En este problema, puesto que es fundamentalmente de cálculo, el tiempo de transmisión por la red y el almacenamiento en disco son despreciables frente al tiempo de cómputo. Vamos a considerar distintos escenarios de carga de trabajo y vamos a utilizar al menos 2 computadores conectados mediante una red de comunicación para realizar distribuir las tareas entre las máquinas.

En primer lugar, vamos a evaluar el hardware que tenéis disponible para ejecutar el programa del Código 3. Para ello, realiza el test de evaluación proporcionado por la clase `Test` (Código 1)[3]. Como salida por pantalla, este programa muestra la siguiente información:

- el número de núcleos del procesador de tu computadora (*unidades de hardware* que van a permitir la ejecución de tareas en paralelo);
- el tiempo de ejecución para ejecutar la tarea $\mathcal{T}$, que encuentra los números perfectos en el intervalo entero $\{1, 10000\}$; y

[3]Para que funcionen los códigos proporcionados en este ejercicio necesitarás compilar previamente el código fuente `Interval.java`, proporcionado como fichero auxiliar de la práctica.

- el número de tareas máximo que la máquina puede calcular por unidad de tiempo.

Con esa información, podemos determinar el número de núcleos que necesitamos para garantizar una cierta escalabilidad en el sistema, de manera que si ejecutamos la tarea $\mathcal{T}$ (que encuentra los números perfectos en el intervalo $\{1, 10000\}$) y el sistema tarda un tiempo t_1 en proporcionar una respuesta, si ejecutamos 10 tareas $\mathcal{T}$, el sistema debería proporcionar las respuestas aproximadamente en el mismo tiempo t_1, sin ningún retraso.

Ejercicio 2.

Considera para este ejercicio **4 escenarios distintos**, donde en cada escenario se ejecuta siempre la tarea $\mathcal{T}$, pero con un número de tareas $\mathcal{T}$ y frecuencia periódica con la que se realizan las peticiones diferente para cada escenario. Las frecuencias para cada escenario son las siguientes:

- *Escenario 1*: 1 tarea $\mathcal{T}$ cada 2 segundos.
- *Escenario 2*: 2 tareas $\mathcal{T}$ cada 2 segundos.
- *Escenario 3*: 6 tareas $\mathcal{T}$ cada 2 segundos.
- *Escenario 4*: 8 tareas $\mathcal{T}$ cada 2 segundos.

Puntuación: 3 puntos

Lo que se pide en este ejercicio es lo siguiente:

- *A partir del esquema del código fuente del cliente (véase el Código 2), modifica el código adecuadamente para generar los cuatro escenarios anteriores y utilizar el* `MessageSystem` *para comunicarse con un servidor.*
- *Para cada escenario, proponer razonadamente la arquitectura más simple que garantiza el QoS en cada escenario. Se puede ordenar las arquitecturas de mayor a menor simplicidad de esta manera: (i) cliente / servidor secuencial (más simple), (ii) cliente / servidor concurrente, (iii) máster / worker (menos simple). La explicación debe encontrarse en la cabecera del servidor o en el máster. Hay que diseñar el sistema de manera que* **el tiempo de ejecución de cada petición en cada escenario sea parecido al tiempo que se obtenía al realizar el test inicial**.

En este ejercicio es obligatorio utilizar vuestro `MessageSystem` *y realizar invocaciones a la función* `getPerfectNumbers` *para resolver el problema (es decir, no se permite guardar resultados previamente calculados). Por último, observa empíricamente en cada escenario que no se viola el QoS.* **NOTA***: puedes utilizar el número de máquinas que estimes oportuno. Implementa los códigos fuente necesarios y guárdalos en un directorio (carpeta) de nombre* `p05e02`*. Recuerda: en este directorio guarda únicamente los códigos fuente (ficheros con extensión* `.java`*), no los ficheros objeto (extensión* `.class`*).*

Código 1: Código fuente del fichero `Test.java`.

```
import java.util.List;

public class Test{

    public static void main(String args[]){
        Interval intervalo = new Interval(1, 10000);
        System.out.println("Puede realizar " + Runtime.getRuntime
            ().availableProcessors() + " tareas a la vez");
        PerfectNumbersServer perfecto= new PerfectNumbersServer();
        long ini = System.currentTimeMillis();
        List<Long> perfectos = perfecto.getPerfectNumbers(
            intervalo);
        long end = System.currentTimeMillis();
        System.out.println("Tiempo de ejecución por tarea: " + (
            end - ini));
        for(int i = 0; i < perfectos.size(); i++){
            System.out.println(" N perfecto:" + perfectos.get(i));
        }

        System.out.println("El num tareas por unidad tiempo (
            throughput) de esta maquina es:\n  "
                + Runtime.getRuntime().availableProcessors() + "
                    tareas cada "  + (end - ini) + " ms"
                + " o " + (Runtime.getRuntime().
                    availableProcessors() * 1000.0) / (end - ini) +
                     " tareas / s");
    }
}
```

Código 2: Código fuente (incompleto) del fichero `ClientePerfectos.java`.

```
public class ClientePerfectos {

    public static void main(String args[]){

        long delay = ¿?;
        int numeroDeTareas = ¿?;
        while(true){
            long t_ini = System.currentTimeMillis();
            for(int i = 0; i < numeroDeTareas; i++)
                List<Long> perfectos = ¿? //  invoca al servidor /
                    master con el MessageSystem

            ¿? receive(); // recibir los resultados usando el
                MessageSystem, completar
            long t_end = System.currentTimeMillis();
            System.out.println("Tiempo de ejecución de la(s) tarea
                (s): " + (t_end - t_ini));
            Thread.sleep(delay);
        }
    }
}
```

Código 3: Código fuente (incompleto) del fichero `PerfectNumbersServer.java`.

```
import java.util.ArrayList;
import java.util.List;

public class PerfectNumbersServer {

    public List<Long> getPerfectNumbers(Interval interval) {
        ArrayList<Long> perfectNumbersList =
                new ArrayList<Long>();

        // Calculamos el conjunto de perfectos en el intervalo
        for (long i = interval.getLowerBound();
                    i <= interval.getUpperBound(); i++) {
            int suma = 0;
            for (int j = 1; j <= i - 1; j++) {
                if (i % j == 0) // ¿es j un divisor de i?
                    suma = suma + j;
            }
            if (suma == i){ //  i es Perfecto
                perfectNumbersList.add(i);
            }
        }
        return perfectNumbersList;
    }

    public static void main(String args[]){
        while(true){
            // Por completar
        }
    }
}
```

3. Algoritmo de Ricart-Agrawala

Ejercicio 3.

Como último ejercicio, se solicita traducir el algoritmo de Ricart-Agrawala a Java (desde el algoritmo original en ALGOL), tal cual está descrito en [1].

Puntuación: 3 puntos

Este ejercicio puede entregarse. Implementa el algoritmo pedido en Java y entrega el código fuente del mismo, guárdalo en un directorio (carpeta) de nombre `p05e03`. *Recuerda: en este directorio guarda únicamente los códigos fuente (ficheros con extensión* `.java`*), no los ficheros objeto (extensión* `.class`*).*

4. Evaluación

La realización de las prácticas es por parejas. Las normas de la evaluación y entrega son las siguientes:

- Todos los programas entregados deben compilar correctamente, se valorará de forma muy negativa que no compile algún programa.
- Todos los programas entregados deben funcionar correctamente como se especifica en el problema.
- Todos los programas tienen que seguir el manual de estilo de Java propuesto por Oracle, disponible en el anillo digital docente (un 20 % de la nota estará en función de este requisito). Además de lo especificado en el manual de estilo, cada fichero fuente deberá comenzar con la siguiente cabecera:

```
/*
 * AUTOR: nombre y apellidos del autor de la práctica
 * NIA: número de identificación del alumno
 * FICHERO: nombre del fichero
 * TIEMPO: tiempo (en horas) empleado en la implementación
 * DESCRIPCIÓN: breve descripción del contenido del fichero
 */
```

Resumen de la puntuación de los ejercicios solicitados en esta práctica

Ejercicio	Puntuación
Ejercicio 1	4 puntos.
Ejercicio 2	3 puntos.
Ejercicio 3	3 puntos.

5. Instrucciones de entrega y Defensa

- Solo uno de los miembros de la pareja de prácticas realizará la entrega.
- Se entregará un único fichero comprimido `.zip`, que contendrá los códigos fuentes de todos los ejercicios solicitados (ficheros con extensión `.java`, **NO los ficheros con extensión .class**).
- Los códigos fuentes estarán en sus respectivas carpetas, indicadas en el enunciado de cada carpeta.
- La entrega se realizará a través Moodle (`http://moodle2.unizar.es`), se utilizará para ello una tarea correspondiente a la entrega de la práctica habilitada a tal efecto.
- **Fecha de entrega:**
  - Grupo `prs2` (921): 31 de mayo de 2020.

- Grupos `prs4` y `prs5` (921): 31 de mayo de 2020.
- Grupo `prs1` (921): 31 de mayo de 2020.
- Grupos `prs1`, `prs2` y `prs3` (931): 31 de mayo de 2020.

Las fechas de entrega coinciden con el fin del curso académico presencial (antes de la situación sobrevenida por la COVID-19). **No se permite la entrega tardía**.

Referencias

[1] Glenn Ricart and Ashok K Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, 1981.