Sistemas Distribuidos

Práctica 3: Sistema de mensajes con soporte para depuración

Autor: Javier Celaya, modificada por Rafael Tolosana

Resumen

Como habéis visto en clase, es complicado determinar el estado global de una aplicación o sistema distribuido en un momento arbitrario. Esto dificulta enormemente la tarea de depurar aplicaciones distribuidas. En esta práctica, programaréis un middleware orientado a mensajes que os permita, durante el proceso de depuración, decidir de manera interactiva los estados a través de los que va pasando vuestra aplicación. Sobre él, debéis implementar una serie de procesos y una conversación entre ellos en la que se aprecie cómo la ordenación de los eventos puede determinar el resultado de la aplicación.

Estas prácticas incluyen redactar una memoria y escribir código fuente. El texto de la memoria debe ser original, pero el código puede basarse en otro ya existente, siempre con el consentimiento del autor y citando la fuente. Por ejemplo, el código que encontréis en Internet puede ser utilizado siempre que citéis el nombre del autor y la URL de dónde lo habéis sacado. Sin embargo, las aportaciones personales siempre serán mejor valoradas. No citar la fuente se considera copia, y copiar supone un cero en la nota de prácticas.

Notas sobre esta práctica

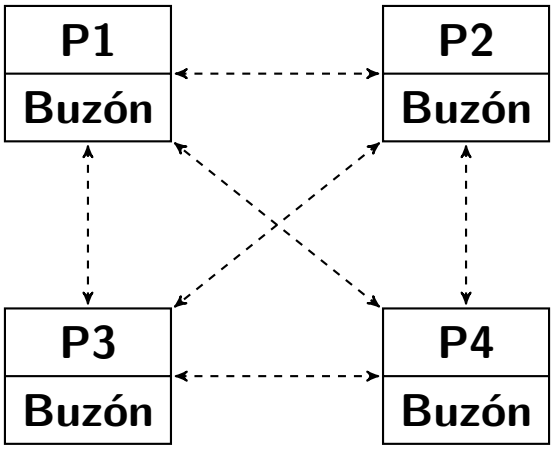
- Al crear una clase que implementa la interfaz `Serializable`, Eclipse sugiere incluir un campo llamado `serialVersionUID`. Decidle que genere el valor por defecto, en esta práctica es suficiente. Para más información, consultad la documentación de la interfaz `Serializable`.

1. Un sistema de mensajes en Java

En esta práctica vais a diseñar un mecanismo de comunicación entre los procesos de una aplicación distribuida con dos funciones principales:

1. Que implemente el paradigma de sistema de mensajes punto a punto.
2. Que os permita determinar la *linearización* de una ejecución de la aplicación, decidiendo en cada momento cuál es el siguiente evento.

Recordaréis que el paradigma de sistema de mensajes punto a punto permite el envío de mensajes entre cada par de procesos, de forma que este envío está *desacoplado*. Es decir, el receptor no tiene por qué estar esperando la llegada de un mensaje cuando ésta se produce. En su lugar, los mensajes se almacenan en un *buzón* hasta que el receptor decida consultarlo. En el siguiente diagrama podéis ver una supuesta aplicación con cuatro procesos, cada uno con su buzón. Todos los procesos tienen asignado un identificador, desde 1 hasta el número de procesos de la aplicación. Además, conocen a los demás participantes en la aplicación, y se pueden enviar mensajes entre todos ellos.



1.1. Envío de objetos por red

Vuestro sistema de mensajes va a permitir enviar casi cualquier objeto de un proceso a otro. La única condición es que ese objeto sea *serializable*. En la práctica anterior sobre RMI hablamos de la serialización, la capacidad que tiene Java de convertir un objeto en una ristra de bytes y viceversa. Para que un objeto sea serializable, tiene que cumplir dos condiciones: implementar la interfaz `Serializable` [1] y que todos sus campos sean a su vez objetos serializables. Todos los tipos básicos y (en general) los tipos de datos estándar de Java son serializables (vectores, listas, maps, etc...). Así, construir clases serializables es muy sencillo. Podéis encontrar más información sobre serialización en [2].

Una vez que hemos definido una clase de objeto serializable, para enviar una instancia por red os harán falta también las clases `ObjectInputStream` y `ObjectOutputStream` del

paquete `java.io`. Suponiendo que la clase `MiMensaje` es serializable, enviar o recibir una instancia por sockets es tan fácil como:

```
MiMensaje m;
...
ObjectInputStream ois = new ObjectInputStream(socket.getInputStream());
m = (MiMensaje) ois.readObject();
ois.close();
...
ObjectOutputStream oos = new ObjectOutputStream(socket.getOutputStream());
oos.writeObject(m);
oos.close();
```

Para vuestro sistema de mensajes, vamos a utilizar la clase `Envelope`:

```
public class Envelope implements Serializable {
    private int source;
    private int destination;
    private Serializable payload;

    public Envelope(int s, int d, Serializable p) {
        source = s;
        destination = d;
        payload = p;
    }

    public int getSource() { return source; }
    public int getDestination() { return destination; }
    public Serializable getPayload() { return payload; }
}
```

Una instancia de la clase `Envelope` encapsula un objeto serializable cualquiera e información acerca del remitente y destinatario del mensaje. Ese "sobre" es lo que los procesos se enviarán entre sí, para poder identificar fácilmente el origen de cada mensaje.

1.2. Buzón de mensajes

El buzón de un sistema de mensajes punto a punto es lo que permite desacoplar el envío de la recepción de mensajes. No es más que una cola de mensajes recibidos pero no leídos. Es decir, el sistema de mensajes debe recibir los mensajes según vayan llegando, y guardarlos en el buzón para cuando el receptor lo consulte. Esto se puede hacer con un hilo que se encarga de escuchar conexiones entrantes. Cada vez que llega un mensaje nuevo, lo añade al final de la cola. Cuando el hilo principal examine el buzón, le devolverá el primer mensaje de la cola.

Asignad un tamaño máximo al buzón: otro de los problemas de los sistemas distribuidos es que los recursos son limitados. Si se recibe un mensaje y el buzón está lleno, será descartado.

2. Objetivo de la práctica

En esta práctica debéis:

1. Programar el sistema de mensajes descrito en la sección 1 con los detalles que se dan en la sección 3, a continuación.
2. Programar una aplicación de ejemplo que utilice el sistema de mensajes.
3. Hacer una sesión de depuración de esa aplicación en la que obtengáis ejecuciones entrelazadas diferentes, según se explica en la sección 4.

De nuevo, como resultado de la práctica entregaréis un fichero JAR que se llamará `SSDDp3.jar`. El fichero JAR debe contener tres cosas:

1. una memoria (4 páginas) con los nombres de los dos componentes de la pareja, en la que resumiréis vuestro diseño del sistema, explicando el funcionamiento de cada componente y sus interacciones. **Poned especial atención en explicar cómo habéis implementado el buzón y cómo gestionáis las conexiones TCP entre los procesos.** Ceñíos a lo que se os pide, sed claros y concisos. Esa memoria debe incluir también una traza de las pruebas que creáis necesarias para validar su correcto funcionamiento, abarcando todos los casos que se os ocurran (envío de mensajes entre cada par de procesos, procesos en distintos ordenadores, etc...). Finalmente, debéis describir la aplicación de ejemplo que habéis diseñado y la sesión de depuración, tal como se indica en la sección 4. No os extendáis más de cuatro páginas en total.
2. todo el código fuente que hayáis desarrollado, según el estándar de codificación de Java que ya habéis utilizado en prácticas anteriores. Comentadlo convenientemente, si fuera necesario. Poned atención a la organización del código, la modularidad, la legibilidad, la encapsulación, etc... Y mucho cuidado con el acceso concurrente al buzón.
3. el fichero JAR debe contener el bytecode compilado. De esta manera, por un lado, se debe poder utilizar el JAR para desarrollar cualquier aplicación distribuida con vuestro sistema de mensajes. Además, tiene que ser posible invocar la ejecución de un proceso de la aplicación de ejemplo con la orden:

```
java -jar SSDDp3.jar [-d] num_proc fichero_red
```

La opción "-d" activa la aparición de mensajes de depuración en la salida estándar. El parámetro "num_proc" indica el identificador que tendrá ese proceso dentro del sistema de mensajes. El parámetro "fichero_red" es un fichero que describe la localización de cada proceso. Encontraréis los detalles en la sección 3.

3. Implementación del sistema de mensajes

Cada proceso que quiera utilizar vuestro sistema de mensajes creará una instancia de la clase `MessageSystem`, en el paquete `ssdd.ms`:

```
package ssdd.ms;

public class MessageSystem {
    public MessageSystem(int source, String networkFile, boolean debug)
        throws FileNotFoundException { ... }
    public void send(int dst, Serializable message) { ... }
    public Envelope receive() { ... }
    public void stopMailbox() { ... }
    ...
}
```

El constructor recibe tres parámetros, los mismos que le debéis pasar al JAR cuando ejecutéis vuestra aplicación de ejemplo más adelante. El primer parámetro es el identificador del proceso actual dentro del sistema de mensajes. El segundo es un fichero de red que contiene la dirección de los participantes del sistema; su estructura se describe en la siguiente sección. Si ese fichero no existe, se produce una excepción de clase `FileNotFoundException`. El tercer parámetro es una bandera que activa la aparición de mensajes de depuración cada vez que se produzca un evento de envío o recepción. Estos mensajes deben indicar el remitente y destinatario, y el contenido. Por eso es útil que todos los objetos que enviéis tengan adecuadamente implementado el método `toString`.

Los siguientes dos métodos se encargan del envío y la recepción. El método `send` envía un objeto serializable a cualquier destinatario del sistema. Si la bandera de depuración está activada, lo primero que debe hacer ese método es indicar lo que se va a enviar. Luego construirá un `Envelope` y lo enviará por red. El método `receive` se encarga de extraer el siguiente mensaje del buzón. Una vez hecho, si la bandera de depuración está activada, mostrará el contenido de ese mensaje.

Finalmente, el método `stopMailbox` detiene la ejecución del hilo que gestiona el buzón.

Es importante dar un nombre de paquete y una interfaz consistentes, para que cualquiera pueda utiliza vuestro sistema de mensajes en sus aplicaciones. Por eso, la clase `Envelope` también debe estar en el paquete `ssdd.ms`.

Ejercicio 1. Completad las partes que aparecen con puntos suspensivos en la clase `MessageSystem`. Programad las clases que creáis necesarias para implementar el buzón.

3.1. Fichero de red

El constructor de la clase `MessageSystem` tiene dos parámetros, un entero y una cadena. El primer parámetro define el identificador del proceso actual. El segundo parámetro es un fichero que describe dónde localizar a los demás procesos, y tiene la siguiente pinta:

```
maquina1:puerto1
maquina2:puerto2
...
```

Es decir, la linea n contiene una pareja de valores, con el nombre (o la IP) de la maquina donde se está ejecutando el proceso n y el puerto en el que escucha mensajes entrantes. Así, cada proceso puede saber a dónde dirigirse al enviar un mensaje. Además, de la línea que corresponde al proceso en ejecución, el objeto `MessageSystem` puede extraer en qué puerto se tiene que poner a escuchar. Le tendréis que pasar el mismo fichero a todos los procesos, para que todos se conozcan entre sí.

4. Depuración de una aplicación de ejemplo

Observad la siguiente aplicación que utiliza el sistema de mensajes, en la que dos procesos, A y B, quieren escribir en una variable almacenada en un tercero, C:

```
class MessageValue implements Serializable {
    private int value;
    public MessageValue(int v) { value = v; }
    public int getValue() { return value; }
    public String toString() { return "Asignar valor " + value; }
}

class A {
    public static void main(String args[]) {
        ...
        MessageSystem ms = new MessageSystem(1, networkFile, debug);
        ms.send(3, new MessageValue(1));
        ms.stopMailbox();
    }
}

class B {
    public static void main(String args[]) {
        ...
        MessageSystem ms = new MessageSystem(2, networkFile, debug);
        ms.send(3, new MessageValue(2));
        ms.stopMailbox();
    }
}
```

```
class C {
    public static void main(String args[]) {
        ...
        MessageSystem ms = new MessageSystem(3, networkFile, debug);
        int valor;
        Envelope e;
        e = ms.receive();
        valor = ((MessageValue)e.getPayload()).getValue();
        e = ms.receive();
        valor = ((MessageValue)e.getPayload()).getValue();
        System.out.println("El valor almacenado finalmente es " + valor);
        ms.stopMailbox();
    }
}
```

¿Podemos saber a priori cuál será el resultado del programa? Según el funcionamiento de esta aplicación, A y B envían cada uno su valor a C. Como son eventos independientes, es igual en qué orden ocurran. No así la recepción por parte de C: el último valor que reciba será el que finalmente permanezca en la variable. Ejecuciones entrelazadas diferentes producen resultados distintos. Normalmente, este no sería el comportamiento deseable de una aplicación. La depuración con este sistema de mensajes os permite ver un error de diseño como ese.

Además de la bandera de depuración, que nos muestra los mensajes que se envían y reciben, debemos poder controlar el orden en que esto ocurre. Eso lo haremos con la depuración de Eclipse. Vais a colocar un *breakpoint* justo antes de enviar un mensaje (pero después de mostrar su contenido en pantalla) y justo antes de recibirlo.

Ejercicio 2. Depurad la aplicación de ejemplo con Eclipse y comprobad que sale un resultado distinto con ejecuciones entrelazadas diferentes. No es necesario que hagáis ningún comentario en la memoria. El código está colgado en la web de la asignatura.

Ejercicio 3. Diseñad un concepto de aplicación distribuida que utilice vuestra implementación de un sistema de mensajes. En esa aplicación debe darse la situación anterior: ejecuciones entrelazadas distintas pueden conllevar resultados diferentes. Realizad una traza de depuración en la que encontréis el problema y explicadla en la memoria.

5. Realización y Evaluación (Rúbrica)

La realización de las prácticas es por parejas, pero los dos componentes de la pareja deberán entregarla de forma individual. En general estos son los criterios de evaluación:

- Deben entregarse todos los programas, se valorará de forma negativa que falte algún programa / alguna funcionalidad.

- Todos los programas deben compilar correctamente, se valorará de forma muy negativa que no compile algún programa.
- Todos los programas 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 moodle2. Además de lo especificado en el manual de estilo, cada fichero fuente deberá comenzar con la siguiente cabecera:

```
/*
 * AUTOR: nombre y apellidos
 * NIA: n'umero de identificaci'on del alumno
 * FICHERO: nombre del fichero
 * TIEMPO: tiempo en horas de codificaci'on
 * DESCRIPCI'ON: breve descripci'on del contenido del fichero
 */
```

5.1. Rúbrica

Con el objetivo de que, tanto los profesores como los estudiantes de esta asignatura por igual, puedan tener unos criterios de evaluación objetivos y justos, se propone la siguiente rúbrica en el Cuadro 1. Los valores de las celdas son los valores mínimos que hay que alcanzar para conseguir la calificación correspondiente y tienen el siguiente significado:

- A+ (excelente). En el caso de software, conoce y utiliza de forma autónoma y correcta las herramientas, instrumentos y aplicativos software necesarios para el desarrollo de la práctica. Plantea *correctamente* el problema a partir del enunciado propuesto e identifica las opciones para su resolución. Aplica el método de resolución adecuado e identifica la corrección de la solución, *sin errores*. En el caso de la memoria, se valorará una estructura y una presentación adecuadas, la corrección del lenguaje así como el contenido explica de forma precisa los conceptos involucrados en la práctica. En el caso del código, este se ajusta exactamente a las guías de estilo propuestas.
- A (bueno). En el caso de software, conoce y utiliza de forma autónoma y correcta las herramientas, instrumentos y aplicativos software necesarios para el desarrollo de la práctica. Plantea *correctamente* el problema a partir del enunciado propuesto e identifica las opciones para su resolución. Aplica el método de resolución adecuado e identifica la corrección de la solución, *con ciertos errores* no graves. Por ejemplo, algunos pequeños casos (marginales) no se contemplan o no funionan correctamente. En el caso del código, este se ajusta *casi* exactamente a las guías de estilo propuestas.
- B (suficiente). En el caso de software, conoce y utiliza de forma autónoma y correcta las herramientas, instrumentos y aplicativos software necesarios para el desarrollo de

la práctica. No plantea correctamente el problema a partir del enunciado propuesto y/o no identifica las opciones para su resolución. No aplica el método de resolución adecuado y / o identifica la corrección de la solución, pero *con errores*. En el caso de la memoria, bien la estructura y / o la presentación son mejorables, el lenguaje presenta deficiencias y / o el contenido no explica de forma precisa los conceptos importantes involucrados en la práctica. En el caso del código, este se ajusta a las guías de estilo propuestas, pero es mejorable.

- B- (suficiente, con deficiencias). En el caso de software, conoce y utiliza de forma autónoma y correcta las herramientas, instrumentos y aplicativos software necesarios para el desarrollo de la práctica. No plantea correctamente el problema a partir del enunciado propuesto y/o no identifica las opciones para su resolución. No se aplica el método de resolución adecuado y/o se identifica la corrección de la solución, pero *con errores* de cierta gravedad y/o sin proporcionar una solución completa. En el caso de la memoria, bien la estructura y / o la presentación son *manifiestamente* mejorables, el lenguaje presenta *serias* deficiencias y / o el contenido no explica de forma precisa los conceptos importantes involucrados en la práctica. En el caso del código, hay que mejorarlo para que se ajuste a las guías de estilo propuestas.

- C (deficiente). El software no compila o presenta errores graves. La memoria no presenta una estructura coherente y/o el lenguaje utilizado es pobre y/o contiene errores gramaticales y/o ortográficos. En el caso del código, este no se ajusta exactamente a las guías de estilo propuestas.

Calificación	Sistema	Traza	Código	Memoria
10	A+	A+	A+	A+
9	A+	A+	A	A
8	A	A	A	A
7	A	A	B	B
6	B	B	B	B
5	B-	B-	B-	B-
suspenso	1 C			

Cuadro 1: Detalle de la rúbrica: los valores denotan valores mínimos que al menos se deben alcanzar para obtener la calificación correspondiente

6. Entrega y Defensa

Deberéis entregar el fichero JAR a través de moodle2 en la actividad habilitada a tal efecto, no más tarde del jueves anterior al comienzo de la siguiente práctica, esto es, el *19 de noviembre de 2015* para los grupos A y el *26 de noviembre de 2015* para los grupos B.

Los días 20 y 27 de noviembre durante la sesión de prácticas se realizará una defensa "in situ" de la práctica.

Referencias

[1] *Documentación oficial de la interfaz Serializable* `http://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html`

[2] *Tutorial sobre serialización de objetos en Java* `http://chuwiki.chuidiang.org/index.php?title=Serializaci%C3%B3n_de_objetos_en_java`