Sistemas Distribuidos

Práctica 4: Sala virtual de chat

Autor: Javier Celaya, modificada por Víctor Medel

Resumen

A patir del sistema de mensajes de la práctica anterior, se va a implementar una aplicación de sala virtual de chat distribuida. Cada proceso tendrá una típica ventana de conversación, con un campo de texto y una lista de mensajes recibidos. La comunicación será *multicast*, es decir, los mensajes que envíe cada proceso les llegarán a todos los demás. Haciendo uso del sistema de mensajes implementado anteriormente, se deberá implementar un sistema de envío *multicast* con relojes lógicos donde los destinatarios sean capaces de dar el mismo orden a todos los mensajes que reciban, para que la conversación sea consistente.

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; 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

- Cuidado con la concurrencia y la interfaz de usuario. Cuando se llama al método `ActionListener.actionPerformed()`, la interfaz se queda bloqueada hasta que vuelve. Por esa razón, lo que se haga en ese método tiene que ser una tarea corta. Si se va a hacer algo que lleve un cierto tiempo, como comunicación por red (aunque sólo sea envío de mensajes), es mejor lanzarlo en un hilo aparte.

- Se pueden realizar pequeñas modificaciones al código de las prácticas anteriores si es necesario, siempre que no se cambie la semántica de las operaciones principales. Por ejemplo, se puede añadir un método al sistema de mensajes que devuelva cuántos procesos tiene la aplicación, pero no se puede dejar de usar un buzón para la recepción.

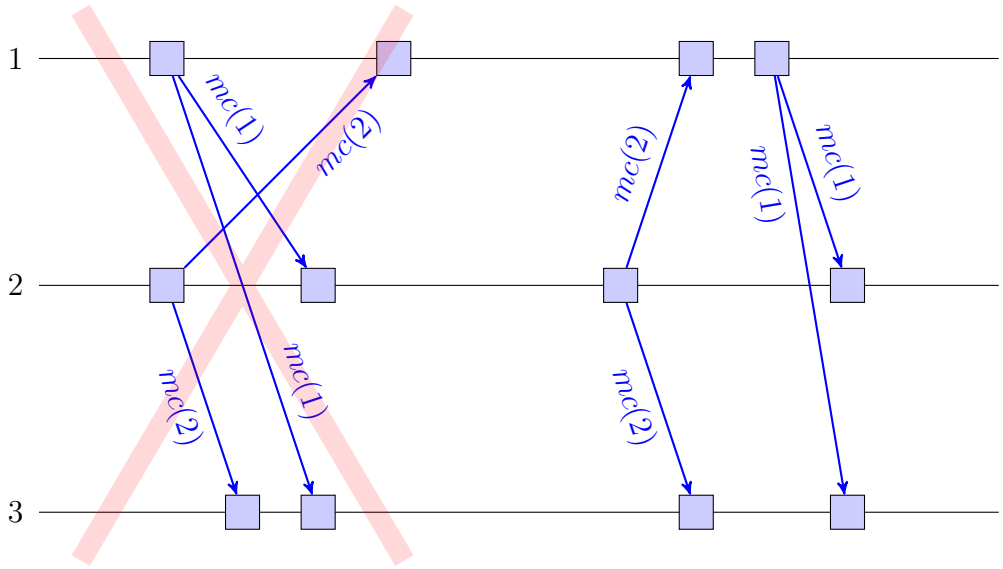

Figura 1: Ejemplo de orden total en mensajes *multicast*.

1. Multicast con ordenación total

Se va a implementar una sencilla aplicación de sala virtual de chat. Esta aplicación permite a un grupo de usuarios enviarse mensajes de texto entre sí. Lo que uno escribe, aparece inmediatamente en las pantallas de todos los participantes. Será requisito indispensable que todos los mensajes de texto se muestren en el mismo orden a todos los usuarios. Para eso, primero se va a implementar un protocolo de comunicación *multicast con ordenación total*, utilizando el sistema de mensajes de la anterior práctica. Este protocolo permite enviar mensajes *multicast* (cada mensaje llega todos los demás procesos), asegurando que todos llegarán a todo el mundo en el mismo orden. En la figura 1 se observa una situación no deseada, en la que los mensajes *multicast* del proceso 1 y 2 no llegan en el mismo orden a todo el mundo, y una situación correcta, en la que llegan en el mismo orden a todos.

Tal y como se ha visto en la parte teórica de la asignatura, un método para ordenar los eventos de una aplicación distribuida es utilizar relojes lógicos. Se estampilla cada mensaje con un valor que nos permite decidir si debe aparecer antes o después de otro. Sin embargo, existe un problema fundamental: la ordenación se conoce *a posteriori*; es decir, cuando ha terminado la aplicación, se pueden recolectar todos los eventos y listarlos en el orden correcto. Cuando un proceso recibe un mensaje, **no tiene forma de saber si luego le llegará un mensaje con una estampilla anterior.** Por lo tanto, los relojes lógicos no son mecanismo suficiente para la aplicación de chat.

Una solución centralizada podría ser usar un secuenciador, que nos de un número de secuencia incremental y globalmente único para cada mensaje. Un proceso no podría enviar su mensaje hasta que no le hubieran llegado todos los mensajes con un número de secuencia menor. En distribuido hay que pensar otro enfoque. Una posible solución

es forzar que cuando un proceso envíe un mensaje *multicast*, éste sea recibido por todos antes de que otro proceso envíe el siguiente mensaje. Es decir, hay que acceder al resto de procesos de forma exclusiva: *exclusión mutua distribuida*. Usando un algoritmo de exclusión mutua distribuida como los vistos en clase, un proceso solicitará entrar en la sección crítica cuando quiera enviar un mensaje *multicast*. Este mensaje sólo se enviará de forma efectiva cuando obtenga el acceso a la sección crítica.

2. Objetivo de la práctica

Los objetivos de esta práctica son:

1. Implementar relojes lógicos de Lamport sobre el sistema de mensajes de la práctica anterior.
2. Usar los relojes para implementar el algoritmo de exclusión mutua distribuido de Ricart y Agrawala.
3. Usar ese algoritmo para enviar mensajes *multicast* con ordenación total.
4. Usar los mensajes *multicast* en una aplicación de chat distribuida.

El fichero JAR que se debe entregar se llamará `NIP1_NIP2_SSDDp4.jar`, donde "NIP1" y "NIP2" es el número de identificación en la Universidad de cada uno de los componentes de la pareja de prácticas. El fichero JAR debe contener tres cosas: Primero, una memoria, con los nombres de los dos componentes de la pareja, en la se explicará vuestro diseño del sistema, analizando el funcionamiento de cada componente y sus interacciones. **Poned especial atención en explicar cómo se han implementado los relojes y el algoritmo de Ricart y Agrawala.** La memoria debe ceñirse a las especificaciones, valorándose positivamente la claridad y concisión. Esa memoria debe incluir también una traza de las pruebas que se consideren necesarias para validar su correcto funcionamiento, abarcando el mayor número de casos posibles (distinto número de participantes, distintas linearizaciones, etc...).

En segundo lugar, debe contener todo el código fuente que se haya desarrollado, comentado convenientemente si fuera necesario. Debe prestarse atención a la organización del código, la modularidad, la legibilidad, la encapsulación, etc...

Por último, el fichero JAR debe contener el bytecode compilado. De esta manera, tiene que ser posible invocar la ejecución de una ventana de chat con la orden:

```
java -jar NIP1_NIP2_SSDDp4.jar [-d] num_proc fichero_red
```

Como en la práctica anterior, la opción "-d" activa la depuración, el parámetro "num_proc" indica el identificador de ese proceso y el parámetro "fichero_red" es el fichero que describe la localización de cada proceso.

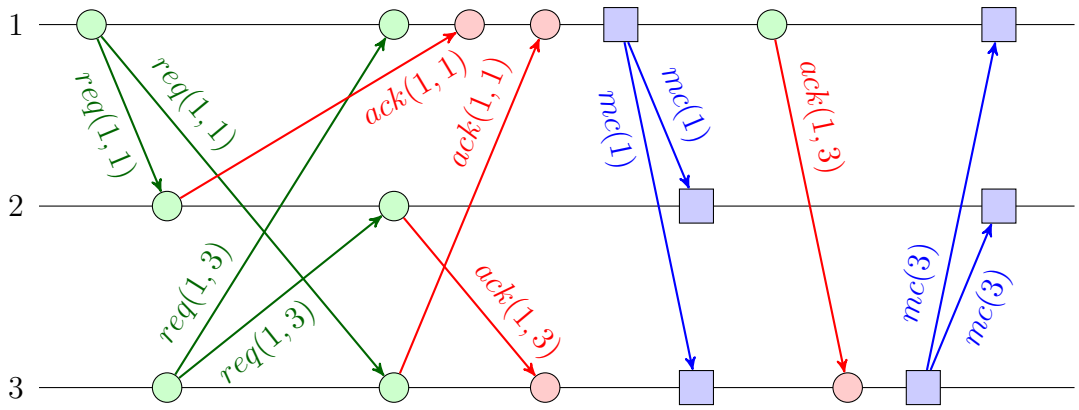

Figura 2: Ejemplo del algoritmo de Ricart y Agrawala con mensajes *multicast*.

3. Relojes lógicos de Lamport

Los relojes lógicos de Lamport, o relojes lógicos escalares, permiten establecer una relación de orden total entre los eventos de una aplicación distribuida, independientemente del instante físico en el que se produjeron. A menudo, es más interesante establecer ese orden lógico que poder determinar el orden físico. En esta práctica se va a añadir un servicio de tiempo lógico al sistema de mensajes de la práctica 3. La clase `MessageSystem` deberá incluir una estampilla para controlar el tiempo local, mientras que cada `Envelope` deberá incluir una estampilla que indique en qué momento se envió.

Ejercicio 1. Se deben modificar las clases `MessageSystem` y `Envelope` para que estampillen cada mensaje según las reglas de los relojes lógicos de Lamport. Se pueden añadir las clases, atributos y métodos que se consideren necesarios. Además, los mensajes de depuración deberán ir precedidos de la estampilla correspondiente al evento de emisión o recepción que se está produciendo.

4. Multicast con ordenación total distribuida

Una vez implementados los relojes lógicos (y probados), se van a usar para el algoritmo de exclusión mutua distribuido de Ricart y Agrawala. El objetivo es usar la sección crítica para enviar los mensajes *multicast*, así que se debe extender la funcionalidad del sistema de mensajes para que permita enviar un mensaje *multicast*. Es interesante ver el envío de un mensaje *multicast* como un único evento. De esta manera, el reloj local sólo se incrementa una vez, y todos los mensajes serán estampillados con la misma marca temporal.

Ejercicio 2. Se debe implementar el método `void sendMulticast(Serializable message)` en la clase `MessageSystem` que envíe un mismo mensaje al resto de procesos. Todos ellos irán estampillados con la misma marca temporal.

En la figura 2 se puede ver un resumen de cómo se orquestaría el envío simultáneo de dos mensajes *multicast*, utilizando la sección crítica del algoritmo de Ricart y Agrawala.

Se debe implementar este algoritmo, tal y como se ha explicado en la parte teórica de la asignatura. La única diferencia reside en utilizar las estampillas asignadas por el sistema de mensajes como número de secuencia. Como resultado, se debe completar la clase `TotalOrderMulticast`, dentro del paquete `ssdd.ms`:

```
package ssdd.ms;

public class TotalOrderMulticast {
    private MessageSystem msystem;
    ...
    public TotalOrderMulticast(MessageSystem ms) { msystem = ms; }
    public void sendMulticast(Serializable message) { ... }
    public Envelope receiveMulticast() {
        while (true) {
            Envelope e = msystem.receive();
            if (...) {
                ...
            } else {
                return e;
            }
        }
    }
}
```

Esta clase tiene dos métodos importantes. El método `sendMulticast()` inicia el protocolo de Ricart y Agrawala enviando una petición para entrar en la sección crítica al resto de procesos. Es decir, **no envía el mensaje *multicast***, propiamente dicho; eso ocurrirá cuando se reciban todas las confirmaciones del resto de procesos. Este método puede ser llamado desde un hilo distinto, así que debe estar convenientemente protegido. Hay que tener en cuenta que si el usuario es rápido, podría pedir enviar un nuevo mensaje antes de que el anterior se haya terminado de enviar...

El método `receiveMulticast()` devuelve el siguiente mensaje *multicast* que se reciba. El bucle interno está ahí porque la llamada a `msystem.receive()` puede devolver tres cosas:

- Una petición de entrada en la sección crítica, a la que se puede responder con una confirmación.
- Una confirmación de otro proceso, que permitirá entrar en la sección crítica y enviar el mensaje que se pasó como parámetro del método `sendMulticast()`.
- Un mensaje *multicast*, que será devuelto.

El método `receiveMulticast()` debe, por tanto, atender a todos los mensajes de control del algoritmo de Ricart y Agrawala hasta que se reciba un mensaje *multicast*. Como el

método `sendMulticast()` no envía directamente el mensaje *multicast*, interesa que el método `receiveMulticast()` devuelva también los mensajes *multicast* propios cuando dan paso a la sección crítica y se envían al resto de procesos. Así, se pueden tratar en el mismo orden. La idea es que una aplicación que utiliza la clase `TotalOrderMulticast` esté contínuamente llamando al método `receiveMulticast()` mientras no tenga otra cosa que hacer, para que atienda las peticiones de entrada en la sección crítica de otros procesos.

Ejercicio 3. Se debe completar la clase `TotalOrderMulticast`. Para ello, se deben crear las clases, atributos y métodos que se consideren necesarios, por ejemplo para representar los mensajes de petición y confirmación de entrada en la sección crítica. Para comprobar su funcionamiento, se debe implementar una aplicación de prueba con cuatro procesos, donde dos de ellos envían un mensaje *multicast* simultáneamente. Se debe ejecutar una prueba con el flag de depuración activo y se mostrará en una traza que todos los eventos de recepción de mensajes *multicast* ocurren en el mismo orden en todos los procesos. Como el sistema *multicast* da servicio a una aplicación de chat, se creará un tipo de mensaje que contenga una cadena de texto.

5. Interfaz gráfica

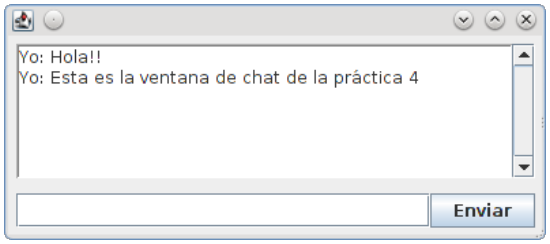

Figura 3: Ventana de chat para la aplicación.

Una vez que se envíen mensajes *multicast* ordenados, se debe terminar de implementar la aplicación de chat. En esta aplicación, habrá un usuario al cargo de cada proceso. A los usuarios se les mostrará una sencilla interfaz gráfica, consistente en un campo de texto y una lista de mensajes, como la de la figura 3. Cada vez que un usuario pulsa el botón enviar, se envía un mensaje a todos los demás procesos con el contenido del cuadro de texto. Al recibir un mensaje de otro proceso, éste se muestra en la lista de mensajes en el orden correcto de la conversación. El código de este interfaz se suministra a través de la web de la asignatura. Tiene la siguiente interfaz:

```
public class ChatDialog extends JFrame {
    public ChatDialog(final ActionListener l) { ... }
    public String text() { ... }
    public void addMessage(final String text) { ... }
}
```

El constructor de la clase `ChatDialog` crea la ventana con la interfaz gráfica y la muestra en pantalla. Tiene como parámetro un objeto `ActionListener` cuyo método `actionPerformed()` será llamado cuando se pulse el botón de envío o la tecla intro. La idea es que ese método se encargue de enviar el contenido del campo de texto como mensaje *multicast*. El método `text` recupera el contenido del campo de texto; después, borra el campo para que el usuario pueda escribir el siguiente mensaje. Finalmente, el método `addMessage()` añade una linea al final de la lista de mensajes.

El siguiente programa muestra cómo se usa la clase `ChatDialog`. Simplemente añade al final de la lista cada mensaje que se escriba en el campo de texto.

```
import java.awt.event.*;
import javax.swing.JFrame;

public class Ejemplo {
    private ChatDialog v;
    public Ejemplo() {
        v = new ChatDialog(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                String m = v.text();
                if (!m.isEmpty())
                    v.addMessage("Yo: " + m);
            }
        });
        v.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
    public static void main(String[] args) {
        Ejemplo e = new Ejemplo();
    }
}
```

Ejercicio 4. Se debe terminar la aplicación de chat incorporando la interfaz gráfica. El método `actionPerformed()` debe enviar un mensaje *multicast* al resto de participantes con el contenido del campo de texto. Al recibir un mensaje *multicast*, propio o ajeno, se añade al final de la lista.

6. 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
 */
```

6.1. 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	Solución del Problema	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

7. Entrega y Defensa

Se debe 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 *3 de Diciembre de 2015* para los grupos A y el *10 de Diciembre de 2015* para los grupos B.

Los días 4 de Diciembre y 11 de Diciembre, durante la sesión de prácticas, se realizará una defensa “in situ” de la práctica.