FAQ – v.6
Ejercicio 5 – Preguntas frecuentes

josé a. mañas

8.4.2016

1 ¿Qué hay que hacer?

Un router que recibe paquetes de varios emisores (senders) y los entrega a varios receptores (receivers). Emisores y receptores son threads concurrentes.

El router tiene una capacidad máxima para encolar paquetes recibidos mientras esperan a ser entregados. Esta capacidad de marca en el constructor.

Cuando el router recibe un paquete y no le caben más, desecha el más antiguo y de menor prioridad.

Cuando el router tiene que entregar un paquete y tiene varios encolados, entrega el más antiguo y de más prioridad. Si no tiene paquetes, devuelve null.

1.1 ¿Cómo se programa? Estructura de datos

El router tiene una capacidad máxima M: es capaz de almacenar hasta M paquetes.

La primera opción que tiene que decidir es la estructura de datos que va a emplear para almacenar paquetes. Podemos usar un array de M posiciones

```
Packet[] queue = new Packet[M];
```

o

```
List<Packet> queue = new ArrayList<Packet>();
```

Funcionalmente es lo mismo; pero a partir de esta decisión hay que hacer un programa con arrays o con listas.

Use la estructura de datos que le resulte más cómoda. Dependiendo de sus habilidades, el uso de una lista es más sencillo y se presta menos a errores.

1.2 ¿Cómo se programa? Posición en la cola

La segunda decisión está relacionada con cuándo ordenamos los paquete que entran (método send()).

Supongamos que llega esta secuencia de paquetes

```
packetM1, packetB1, packetA1, packetM2
```

Opción 1.

Cuando llega un nuevo paquete, lo metemos en la cola al final. Paso a paso:

{ packetM1 }
{ packetM1 packetB1 }
{ packetM1 packetB1 packetA1 }
{ packetM1 packetB1 packetA1 packetM2 }

Cuando hay que sacar un paquete (receive()), hay que seleccionar cuál es el primero:

{ packetM1 packetB1 packetA1 packetM2 } → return packetA1
{ packetM1 packetB1 packetM2 } → return packetM1
{ packetB1 packetM2 } → return packetM2
{ packetB1 } → return packetB1
{ } → return null

Opción 2.

Cuando llega un nuevo paquete, lo metemos en la cola ya preparado en el orden de salida. Paso a paso:

{ packetM1 }
{ packetM1 packetB1 }
{ packetA1 packetM1 packetB1 }
{ packetA1 packetM1 packetM2 packetB1 }

Cuando hay que sacar un paquete (receive()), simplemente se saca el primero:

{ packetA1 packetM1 packetM2 packetB1 } → return packetA1
{ packetM1 packetM2 packetB1 } → return packetM1
{ packetM2 packetB1 } → return packetM2
{ packetB1 } → return packetB1
{ } → return null

Entre las 2 opciones, elija la que le parezca más cómoda. Sin miedo: son igual de difíciles.

1.3 ¿Cómo se programa? Pérdida de paquetes cuando no caben

Estén los paquetes almacenados como estén, cuando llega uno nuevo y no cabe, hay que recorrer toda la cola y apuntar cuál es el paquete más antiguo y de menor prioridad. Al acabar el recorrido, se elimina el detectado. Esta eliminación operación parece más fácil con una List<Packet>.

1.4 ¿Una cola o tres colas?

Sea array o lista, puede usarse una cola para todos los paquetes, o también se puede usar 3 colas, una para paquetes de ALTA prioridad, otra para los de MEDIA y otra para los de BAJA. Tiene que ir con cuidado de que en ningún momento haya más paquetes en total de los que admite el router.

Esta opción tiene algo de 'truco de programación' ya que un router real con limitación de memoria usará una única cola para todos los paquetes.

1.5 ¿Puede ser que desechemos el paquete nuevo que llega?

Sí.

Por ejemplo, si tenemos la cola llena de paquetes de alta prioridad y llega un paquete de baja prioridad, el que se desecha es el que llega.

Como 'truco' de programación, puede primero meterlo en la cola (que puede crecer 1 posición por encima del máximo) y si después de meterlo se da cuenta de que se ha pasado, "pedir perdón" y sacar el menos importante. Esta forma de programar simplifica el código porque hay menos casos especiales.

1.6 ¿Cómo se comparan prioridades?

Use

```
int compareTo(Priority)
```

Ver

http://www.dit.upm.es/~pepe/libros/vademecum/topics/99.html

2 Estado

El estado del router es la cola de paquetes pendientes de entregar.

Tanto emisores como receptores alteran el estado del router.

Es necesario proteger el estado como una zona de exclusión mutua.

3 ¿Cómo se establece una zona de acceso exclusivo?

Lo mejor es usar métodos sincronizados.

Aunque también podría recurrir a otros mecanismos como

- cerrojos (java.util.concurrent.locks.Lock)
- semáforos (java.util.concurrent.Semaphore)
- zonas protegidas por sysnchronized(Object) { … }

3.1 ¿Hay que usar sincronización condicional; o sea, wait()?

No.

4 Pruebas

Todo sistema debe ser probado antes de entregarlo.

Un fallo en las pruebas indica que hay un error que debe ser reparado antes de entregarlo.

Pasar todas las pruebas no implica que el sistema carezca de errores: simplemente es que no los hemos encontrado.

Un buen conjunto de pruebas reduce la probabilidad de que el sistema aún contenga errores.

4.1 Unitarias de corrección

Use JUnit para validar los métodos send() y receive().

JUnit ejecuta los tests en orden aleatorio; pero no concurrente. Se puede usar para probar la corrección de los métodos sin concurrencia.

Hay que validar

- que los paquetes salen según el criterio de primero los de mayor prioridad y, dentro de la misma prioridad, primero los más antiguos (FIFO – First In First Out)
- que se desechan los paquetes que no caben, primero el de menor prioridad y, dentro de la misma prioridad, primero el más antiguo

4.1.1 Ejemplo

```
public class TsRouterTest {
    private final Packet packetA1 = new Packet(ALTA, 1);
    private final Packet packetA2 = new Packet(ALTA, 2);
    private final Packet packetA3 = new Packet(ALTA, 3);
    private final Packet packetM1 = new Packet(MEDIA, 1);
    private final Packet packetM2 = new Packet(MEDIA, 2);
    private final Packet packetM3 = new Packet(MEDIA, 3);
    private final Packet packetB1 = new Packet(BAJA, 1);
    private final Packet packetB2 = new Packet(BAJA, 2);
    private final Packet packetB3 = new Packet(BAJA, 3);

    @Test
    public void ejemplo() {
        Packet[] seq_send = new Packet[]{
                packetM1, packetB1, packetA1, packetM2,};
        Packet[] seq_rec = new Packet[]{
                packetA1, packetM1, packetM2, packetB1,};
        TsRouter router = new TsRouter(5);
        for (Packet packet : seq_send)
            router.send(packet);
        for (Packet packet : seq_rec)
            assertSame(packet, router.get());
        assertNull(router.get());
    }
```

4.2 De concurrencia

Haga pruebas de humo (*smoke test*) con

- 1 TsRouter
- 5 Sender
- 5 Receiver

propiedades que deben satisfacerse
corrección (correctness) los paqutes salen o son desechados según los criterios apuntados **seguridad (safety)** el estado no se corrompe; un estado corrupto genera excepciones **vivacidad (liveness)** entran y salen paquetes; el sistema no se queda congelado **equidad (fairness)** todos los emisores mandan; todos los receptores reciben

Las pruebas de corrección se hacen con JUnit.

Las pruebas de equidad se hacen con Log

- cada vez que un emisor logra enviar un paquete, llame a Log.sending(id)
- cada vez que un receptor recibe un paquete (¡= null), llame a Log.receiving(id)
- Log se encarga de imprimir cada 3s una estadística de cuántos paquetes ha enviado cada emisor y cuántos ha recibido cada receptor. Deben ser tasas equilibradas.

Ejemplo de Log

```
senders:   {0=126, 1=115, 2=119, 3=128, 4=115}
receivers: {0=125, 1=104, 2=116, 3=116, 4=117}

senders:   {0=242, 1=233, 2=241, 3=243, 4=234}
receivers: {0=239, 1=225, 2=228, 3=232, 4=237}

senders:   {0=366, 1=348, 2=367, 3=363, 4=361}
receivers: {0=352, 1=349, 2=342, 3=353, 4=347}

senders:   {0=488, 1=471, 2=489, 3=486, 4=482}
receivers: {0=465, 1=469, 2=459, 3=463, 4=469}
```

5 Entrega

- Sender.java
- Receiver.java
- TsRouter.java
- TsRouterTest.java
- TsRouterSmokeTest.java
- … otras clases que haya usado