Toty: multicast con orden total*

Federico C. Repond
frepond@unq.edu.ar

Esteban Dimitroff Hódi
esteban.dimitroff@unq.edu.ar

6 de junio de 2016

1. Introducción

La tarea es implementar un servicio de multicast con orden total usando un algoritmo distribuido. El algoritmo es el usado en el sistema ISIS y está basado en el pedido de propuestas a todos los nodos en un grupo.

2. La Arquitectura

Vamos a tener un conjunto de workers que se comunican entre si usando mensajes multicast. Cada worker va a tener acceso a un proceso de multicast que esconde la complejidad del sistema. Los procesos de multicast están conectados entre si y van a tener que acordar un orden en el que entregan los mensajes. Hay una clara distinción entre recibir un mensaje, que lo hace el proceso de multicast, y entregar un mensaje, que es cuando el worker ve el mensaje. Un proceso multicast va a recibir mensajes sin un orden específico pero solo los va a entregar a su worker en un orden total, esto es, todos los workers en el sistema van a recibir los mensajes en el mismo orden.

2.1. El Worker

Un worker en nuestro ejemplo es un proceso que en intervalos aleatorios desea enviar un mensaje multicast al grupo. El worker va a esperar hasta que ve su propio mensaje antes de enviar otro para prevenir el overflow de mensajes en el sistema.

El worker va a estar conectado a un proceso de gui que simplemente colorea la ventana. La ventana estará inicialmente en negro o en términos de RGB `{0, 0, 0}`. Este es también el estado inicial del worker. Cada mensaje entregado a un worker es un entero `N` en un intervalo, digamos entre 1 y 20. Un worker va a cambiar su estado sumando `N` al valor `R` (del RGB) y rotar los valores. Si

*Adaptado al español del material original de Johan Montelius (`https://people.kth.se/~johanmon/dse.html`)

el estado del worker is `{R, G, B}` y el worker recibe el mensaje `N`, entonces el nuevo estado es `{G, B, (R + N) rem 256}`.

El color del worker entonces va a cambiar a lo largo del tiempo y el orden de los mensajes por supuesto es importante. La secuencia 5, 12, 2 no va a crear el mismo color que la secuencia 12, 5, 2. Si todos los workers empiezan en el color negro y fallamos en la entrega de mensajes en el mismo orden los colores de los workers van a empezar a diferir.

Vamos a experimentar con diferentes implementaciones y para prepararnos para el futuro vamos a hacer la implementación un poco rebuscada. Para empezar proveemos algunos parámetros al worker para hacer los experimentos más fáciles de manejar. Podemos cambiar el módulo del proceso multicast y experimentar con distintos valores de los tiempos de sleep y jitter (fluctuación). El tiempo de sleep es el tiempo de cuanto deben esperar los workers hasta el envío del siguiente mensaje y el tiempo de jitter es un parámetro del proceso de multicast.

Cuando se arranca, el worker debe registrarse con un manager de grupo y va a recibir un estado inicial del proceso y el identificador de proceso de los otros miembros del grupo. Esto puede verse excesivo pero vamos a usarlo en los próximos ejercicios.

2.2. Multicast Básico

Cómo primer experimento deberíamos usar un proceso que implemente multicast básico. Al multicaster se le da un conjunto de procesos pares y cuando se le dice hacer multicast de un mensaje el mensaje es simplemente enviado a todos sus pares uno por uno.

Para hacer el experimento más interesante incluimos un parámetro jitter cuando el proceso arranca. El proceso va a esperar tantos mili-segundos antes de enviar el siguiente mensaje. Esto va a permitir a los mensajes intercalarse y posiblemente causar problemas a los workers.

2.3. Experimento

Experimentar con distintos valores de sleep y jitter y observar cuando los mensajes no son entregados en orden total.

Analizar que pasa si tenemos un sistema que se basa en una entrega con orden total pero esto no fue claramente establecido. ¿Si la congestión es baja y no tenemos retrasos en la red, cuánto tiempo tarda antes de que los mensajes se entreguen fuera de orden? ¿Cuán difícil es hacer debug del sistema y darnos cuenta que es lo que está mal?

3. Multicast con Orden Total

El proceso de multicast es por supuesto la parte más complicada. En esta sección vamos a ir a través del código pero cada uno va a tener que programar

una parte.

El procedimiento general tiene el siguiente estado:

- `Master`: el proceso al cual llegan los mensajes entregados
- `Next`: el siguiente valor a proponer
- `Nodes`: todos los nodos de la red
- `Cast`: un conjunto de referencias a los mensajes que han sido enviados pero aún no se le ha asignado un número de secuencia final.
- `Queue`: los mensajes recibidos pero aún no entregados
- `Jitter`: el parámetro jitter que introduce algún delay en la red

Los números de secuencia son representados por una tupla `{N, Id}`, donde `N` es un entero que es incrementado cada vez que hacemos una propuesta y `Id` es nuestro identificador de proceso.

El conjunto `Cast` es representado como una lista de tuplas `{Ref, L, Sofar}`, donde `L` es el número de propuestas que estamos esperando hasta el momento y `Sofar`, es la propuesta que recibimos hasta ahora.

La `Queue` es una lista ordenada de entradas que representan los mensajes que fueron recibidos pero para los cuales no existe consenso. La lista está ordenada basada en el número de secuencia propuesto o consensuado. Las entradas propuestas son entradas para las que hemos propuesto un número de secuencia. Si tenemos entradas con un número de secuencia acordado al comienzo de la cola estos pueden ser removidos y entregados al worker.

3.1. Enviando un Mensaje

Un mensaje send es una directiva de hacer multicast de un mensaje. Tenemos primero que consensuar en que orden se entregan los mensajes y en consecuencia enviar un pedido de propuestas a todos los pares.

El request debe ser enviado a todos los nodos con una referencia única. Esta referencia va a ser agregada también al conjunto `Cast` con información de cuantos nodos tienen que contestar. Estamos dejando "?" en los lugares del código dónde cada uno debe llenar los valores correctos.

```
{send, Msg} ->
    Ref = make_ref(),
    request(?, ?, ?, ?),
    Cast2 = cast(?, ?, ?),
    server(Master, Next, Nodes, Cast2, Queue, Jitter);
```

Notar que también estamos enviando un pedido a nosotros mismos. Vamos a manejar nuestra propia propuesta en la misma forma que las de los demás. Esto puede ser extraño pero hace el código más fácil.

3.2. Recibiendo un Mensaje

Cuando el proceso recibe un pedido debe responder un nuevo número de secuencia. También debe encolar el mensaje usando el número de secuencia como clave.

```
{request, From, Ref, Msg} ->
    From ! {proposal, ?, ?},
    Queue2 = insert(?, ?, ?, ?),
    Next2 = increment(?),
    server(Master, Next2, Nodes, Cast, Queue2, Jitter);
```

Incrementamos el valor para el siguiente número de secuencia, cualquier cosa que pase no debemos proponer un número igual o menor que el número de secuencia que ya hemos propuesto.

3.3. Recibiendo una Propuesta

Una propuesta es enviada como repuesta a un pedido que hemos enviado anteriormente. La propuesta contiene la referencia al mensaje y el número de secuencia propuesto. Si la propuesta es es la última que estamos esperando entonces hemos encontrado y consensuado el número de secuencia. Implementamos esto llamando a la función `proposal/3` que va a actualizar el conjunto y va a devolver `{agreed, Seq, Cast2}` si se alcanzó un acuerdo o simplemente la lista actualizada.

```
{proposal, Ref, Proposal} ->
    case proposal(?, ?, ?) of
        {agreed, Seq, Cast2} ->
            agree(?, ?, ?),
            server(Master, Next, Nodes, Cast2, Queue, Jitter);
        Cast2 ->
            server(Master, Next, Nodes, Cast2, Queue, Jitter)
    end;
```

Si tenemos consenso esto debe ser enviado a todos los nodos en la red. Esto es manejado por el procedimiento `agree/3`.

3.4. Consenso Finalmente!

Un mensaje de consenso contiene el número de secuencia consensuado de un mensaje particular. El mensaje que se encuentra en la cola debe ser actualizado y posiblemente movido la final (el número consensuado puede ser mayor que el número propuesto). Esto es manejado por la función `update/3`.

Debemos también incrementar nuestro siguiente número:

```
{agreed, Ref, Seq} ->
    Updated = update(?, ?, ?),
```

```
{Agreed, Queue2} = agreed(?, ?),
deliver(?, ?),
Next2 = increment(?, ?),
server(Master, Next2, Nodes, Cast, Queue2, Jitter);
```

Si el número de secuencia consensuado es mayor que el que tenemos debemos incrementar nuestro valor; debemos asegurarnos que nunca proponemos un valor que pueda ser menor que un valor consensuado.

Si nuestro primer mensaje en la cola tiene un número de secuencia acordado puede ser entregado. La función `agreed/2` debe remover los mensajes que pueden ser entregados y retornarlos en una lista. Estos mensajes peden entonces ser entregados usando el procedimiento `deliver/2`.

4. Experimentos

Probar usando el multicaster de orden total. ¿Mantiene los workers sincronizados? Tenemos muchos mensajes en el sistema, ¿cuántos mensajes podemos hacer multicast por segundo y cómo depende esto del número de workers?

Construir una red distribuida, ¿cuán grande puede ser antes de que empiece a fallar?