Modelos y Paradigmas de Programación
Paso de mensajes

Antal A. Buss

17 de abril de 2006

Concurrencia de paso de mensajes

En concurencia declarativa:

- Comunicación es síncrona
- En *objetos stream*, un solo proceso puede ser fuente de datos del stream
- El objeto debe poder conocer la fuente
- No se puede modelar comunicación cliente/servidor

Por qué no cliente/servidor declarativo?

Opciones propuestas:

- Cada cliente envía mensajes independientes al servidor (ver pasoMensajes1.oz clienteServidor_ingenuo1)

Por qué no cliente/servidor declarativo?

Opciones propuestas:

- Cada cliente envía mensajes independientes al servidor (ver pasoMensajes1.oz clienteServidor_ingenuo1)
  No funciona!

Por qué no cliente/servidor declarativo?

Opciones propuestas:

- Cada cliente envía mensajes independientes al servidor (ver pasoMensajes1.oz clienteServidor_ingenuo1)
  No funciona!
- Imponer sincronización fuerte entre los clientes (ver pasoMensajes1.oz clienteServidor_ingenuo2)

Por qué no cliente/servidor declarativo?

Opciones propuestas:

- Cada cliente envía mensajes independientes al servidor (ver pasoMensajes1.oz clienteServidor_ingenuo1)
  No funciona!
- Imponer sincronización fuerte entre los clientes (ver pasoMensajes1.oz clienteServidor_ingenuo2)
  Funciona, pero bajo fuertes restricciones

Conceptos

- Una idea nueva: canal de comunicación asíncrona i.e Los agentes no esperan por una respuesta
- El canal es un *puerto*:
  - Con un stream asociado objeto puerto=puerto+stream
  - El objeto puerto lee mensajes de su stream y envía mensajes a otros puertos
  - Cada objeto puerto se define mediante un procedimiento recursivo *declarativo*

Puerto

- *direccion* : $[S]$
  - Guarda el stream S en una dirección *única*
  - se registran los cambios en el stream a través del tiempo
- El puerto referencia siempre *la cola* del stream de mensajes
- Enviar un mensaje M al puerto, agrega M a la cola del stream

Enviar mensaje

- Un puerto $d : [S]$
- Enviar M a d:
  - Lee el stream guardado en la dirección d
  - crea una nueva variable S' en el store
  - Impone $S = M \mid S'$
  - Asigna a la dirección d el stream S'

El tipo abstracto puerto

- Creación de un puerto:
  $p = \{\textbf{Newport}\ Xs\}$
- Enviar mensaje
  $\{\textbf{Send}\ PM\}$

Semántica de *Newport*

El elemento semántico $(\{Newport < x >< y >\}, E)$

- Crea un nombre nuevo n (la dirección del puerto)
- Liga $E(< y >)$ y n en el store
- Si lo anterior tiene éxito, agrega el par $E(< y >) : E(< x >)$ al store mutable
- Si la ligadura falla, levanta una excepción de error.

Semántica de *Send*

El elemento $(\{Send\ < x >< y >\}, E)$

- Si la condición de activación es **true** (i.e $E(< x >)$ está determinado), entonces:
  - Si $E(< x >)$ no está ligado aL nombre de un puerto, levante excepción de falla
  - Si el store mutable contiene $E(< x >) : z$ entonces:
    - Crea una variable nueva z' en el store
    - Actualiza el store mutable con $E(< x >) : z'$
    - Crea una nueva lista $E(< y >) \mid z'$ y la liga con z en el store
- Si la condición de activación es **false**, suspender la ejecución

Objetos puerto

Combinación de uno o más puertos, con un stream. El objeto puerto:

- Crea los puertos (globales) y los stream (locales)
- Ejecuta un procedimiento recursivo sobre los streams

```
declare  P1 P2 ... Pn
local    S1 S2 ... Sn in
         {Newport S1 P1}
         {Newport S2 P2}
         ...
         {Newport Sn Pn}
         thread {PR  S1 S2 ... Sn} end
end
```

Objeto puerto con estado

```
fun   {NewPortObject Init Fun}
      proc {MsgLoop S1 State}
       case S1 of Msg|S2 then
           {MsgLoop S2 {Fun Msg State}}
          [] nil then skip end
      end
      Sin
in
      thread {MsgLoop Sin Init} end
      {Newport Sin}
end
```

Objeto puerto sin estado

```
fun   {NewPortObject2 Proc}
      Sin in
       thread for Msg in Sin do {Proc Msg} end end
       {NewPort Sin}
end
```

(ver puertos_clienteServidor en pasoMensajes1.oz)

Protocolos

- Objetos Puerto operan intercambiando mensajes de forma coordinada
- Un protocolo es una secuencia de mensajes entre una o más partes que puede entenderse "por encima" de la noción de mensaje.
- Protocolo simple: RMI
  - "método": Comportamiento del puerto cuando recibe un mensaje
  - El cliente envía una petición al servidor y espera a que el servidor envíe un reconocimiento.

Server

```
proc {ServerProc Msg}
   case Msg
   of calc(X Y) then
      {Delay 2000}
      Y=X*X+2.0*X+2.0
   end
end
Server={NewPortObject2 ServerProc}
```

RMI síncrono

```
proc {ClientProc Msg}
   case Msg
   of work(Y) then
      Y1 Y2 in
      {Send Server calc(10.0 Y1)} {Wait Y1}
      {Send Server calc(20.0 Y2)} {Wait Y2}
      Y=Y1+Y2
   end
end
Client={NewPortObject2 ClientProc}
{Browse {Send Client work($)}}
```

RMI asíncrono

```
proc {ClientProc Msg}
   case Msg
   of work(?Y) then
   Y1 Y2 in
      {Send Server calc(10.0 Y1)}
      {Send Server calc(20.0 Y2)}
      Y=Y1+Y2
   end
end
Client={NewPortObject2 ClientProc}
{Browse {Send Client work($)}}
```