Programación concurrente y distribuida
Paso de mensajes

Camilo Rueda [1]

[1] Universidad Javeriana-Cali

PUJ 2008

Arquitecturas Cliente/servidor

- El servidor provee algún servicio
  - recibe un mensaje
  - contesta el mensaje
  - ejemplo: web server, mail server, etc.
- Los clientes conocen la dirección del servidor, solicitan el servicio mediante mensajes
- El servidor y los clientes corren independientemente

Arquitecturas de par a par (peer to peer)

- Similar a Cliente/Servidor
  - cada cliente es también un servidor
  - se comunican enviándose mensajes
- Cliente, Servidor, Par, son agentes
- En el curso se llaman: objetosPuerto

Propiedades

- Agentes

tienen identidad:	dirección de correo
reciben mensajes:	buzón
procesan mensajes:	buzón ordenado
contestan mensajes:	carta de respuesta pre-direccionada

- Cuestión: cómo representarlos en un modelo de programación

Envío de Mensajes

Mensaje:	estructura de datos
Dirección:	puerto
Buzón:	stream de mensajes
Respuesta:	variable dataflow en el mensaje

Agentes

- Los objetos puerto son agentes
- Cada agente se define por la manera en que contesta a los mensajes
- cada agente corre en un hilo propio

Propiedades

Envío asíncrono

$$P = \{Newport\ S\}$$
thread $\ldots\ \{Send\ P\ M\} \ldots$ **end**
thread $\ldots\ \{Procese\ S\} \ldots$ **end**

- Asíncrono: (1) continua inmediatamente después de enviar
- El enviador no sabe cuándo se procesa el mensaje
  - El mensaje se procesa eventualmente

Respuesta asíncrona

- El enviador envía un mensaje que contiene una variable “dataflow” para recibir la respuesta
  - No espera hasta recibirlo
  - No espera por una respuesta cuando envía
- Espera por respuesta: Solamente si la respuesta se requiere
- Ayuda a evitar latencia
  - El enviador continua su proceso
  - El receptor puede que ya haya enviado el mensaje

Envío síncrono

- A veces se requiere mayor sincronización
  - El enviador quiere sincronizarse con el receptor cuando se reciba el mensaje
  - Esto se conoce como handshake rendezvous
- Puede usarse también para enviar una respuesta
  - El enviador no espera a que la respuesta se calcule o,
  - El enviador espera hasta que la respuesta se calcule.

Envío síncrono

```
proc{SyncSend P M}
 Ack in {Send P M#Ack}
 {Wait Ack}
end
proc {Procese Ms}
 case Ms of M#Ack then
    Ack = ok ...
 end
end
```

Envío Asíncrono

- Lo síncrono se vuelve asíncrono mediante hilos

```
proc{ASyncSend P M}
   thread {SyncSend P M} end
end
```

Mensajes

- Son aspectos importantes de los agentes
- Son valores de primera clase: pueden calcularse, chequearse, manipularse, guardarse
- Pueden contener cualquier estructura de datos, incluyendo procedimientos
- Los mensajes de primera clase son muy expresivos
  - mensajes recibidos pueden ir a un log
  - el agente reenvía estampillando el mensaje en el tiempo

Protocolos Simples

- RMI síncrono
- RMI asíncrono
- RMI con “retorno de llamada” (“callback”) mediante hilos Comunicación de doble vía entre servidor y cliente.
- RMI callback usando registros de continuaciones
- RMI callback usando procedimientos como continuaciones

RMI síncrono

```
proc {ServerProc Msg}
  case Msg of calc(X Y) then Y = X * X + 3,0 * (X + 1,0) end
end
Server = {NewPortObject2 ServerProc}
```

```
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

El cliente no necesita esperar el proceso del servidor

```
proc {ServerProc Msg}
  case Msg of calc(X Y) then Y = X * X + 3,0 * (X + 1,0) end
end
Server = {NewPortObject2 ServerProc}
```

```
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
```

RMI con hilos “callback”

El servidor debe llamar al cliente para completar su servicio

```
proc {ServerProc Msg}
  case Msg of calc(X ?Y Cliente) then X1 D in
    {SendCliente delta(D)}
    X1 = X + D   Y = X1 * X1 + 3,0 * (X1 + 1,0)
    end
end
Server = {NewPortObject2 ServerProc}
```

```
proc {ClientProc Msg}
  case Msg of work(?Z) then Y in
     {Send Server calc(10,0 Y Cliente)}
       Z = Y + 110,0
     [] delta(D) then D = 1,0 end
end
```

RMI con hilos “callback” (2)

Problemas con la solución anterior:
deadlock!.
Solución:
El cliente no espera la respuesta.

```
proc {ClientProc Msg}
  case Msg of work(?Z) then Y in
     {Send Server calc(10,0 Y Cliente)}
     thread Z = Y + 110,0 end
     [] delta(D) then D = 1,0 end
end
```

Cómo sabe el cliente que el servidor ya calculó?
local Z in {Send Cliente work(Z)} {Wait Z} end

RMI con procedimientos de continuación

```
proc {ClientProc Msg}
  case Msg of work(?Z) then
     C = proc {$ Y} Z = Y + 110,0 end
     in
     {Send Server calc(10,0 Cliente cont(C))}
     [] cont(C)#Y then {C Y}
     [] delta(D) then D = 1,0
     end
end
Cliente = {NewPortObject2 ClientProc}
```

Ejercicio1: programar el servidor. Ejercicio2: hacer que el servidor actualice un log del cliente

Manejo de excepciones

```
proc {ServerProc Msg}
  case Msg of sqrt(X Y E)) then
    try
        Y = {Sqrt X} E = ok
    catch Exc then E = excepcion(Exc) end
   end
end
Servidor = {NewPortObject2 ServerProc}
```

```
{Send Servidor sqrt(X Y E)}
case E of excepcion(Exc) then raise Exc end end
```

Combinación de protocolos

Un ejemplo: RMI asíncrono con “callback”

```
proc {ServerProc Msg}
  case Msg of calc(X ?Y Cliente) then X1 D in
    {Send Cliente delta(D)}
    thread X1 = X + D
        Y = X1 * X1 + 3,0 * (X1 + 1,0) end
  end
end
```

Combinación de protocolos(2)

El cliente

```
proc {ClientProc Msg}
  case Msg of work(?Y) then Y1 Y2 in
     {Send Server calc(10,0 Y1) Cliente}
     {Send Server calc(20,0 Y2) Cliente}
      thread Y = Y1 + Y2 end
     []delta(?D) then
         D = 1,0
      end
end
```

Es necesario el *thread*? y en el servidor?

doble “call back”: servidor

```
proc {ServerProc Msg}
  case Msg of calc(X ?Y Cliente) then X1 D in
    {Send Cliente delta(D)}
    thread X1 = X + D
        Y = X1 * X1 + 3,0 * (X1 + 1,0) end
    [] paramservidor(?S) then S = 0,1 end
  end
end
```

Doble “call back”

```
proc {ClientProc Msg}
  case Msg of work(Z) then Y in
     {Send Server calc(10,0 Y Cliente)}
      thread Z = Y + 3 end
     []delta(?D) then S in
     {Send Servidor paramservidor(S)}
     thread D = 1,0 + S end
      end
end
```

Es necesario el *thread*?

Trabajo 1

Se debe modelar una intersección de una calle de doble vía con una de una sola vía.

- La calle de una vía va en el sentido este-oeste.
- Hay dos semáforos: uno colgante y uno de poste. El de poste controla el giro hacia el oeste de los carros que vienen del norte. El colgante controla todo lo demás.
- Hay un controlador de los semáforos. Se encarga del tiempo de cambio de luces en ellos. La flecha verde de giro siempre dura menos. El controlador puede hacer que los semáforos queden con luz amarilla intermitente.
- La calle de doble vía tiene dos carriles en la dirección Norte-sur y uno en la dirección sur-norte. Uno de los carriles Norte-sur es para los que cruzan (o siguen derecho!).

Trabajo 1 (cont)

- Un carro puede pasarse al carril de cruce, sin hay espacio.
- Los carros no necesariamente andan todos a la misma velocidad
- Cada carro, luz de un semáforo y controlador, es un objeto puerto independiente.
- Puede suponer que la intersección con sus calles es una grilla de $m \times n$ espacios.
- En cada espacio cabe un carro. Cada espacio es un objeto puerto.
- Cuando hay luz amarilla intermitente, todos pasan como pueden (cuidando de no estrellarse!).

Trabajo 1 (cont)

- Hay carros que entran de vez en cuando a la grilla (y que salen de ella). Esto es asíncrono.
- Debe modelarse todo el sistema con máxima concurrencia (siempre mediante paso de mensajes).
- Debe incluir una interfaz gráfica para visualizar la operación del sistema.
- Interfaces más realistas tienen mayor puntaje.