

Teoría de Lenguajes

Teoría de la Programación

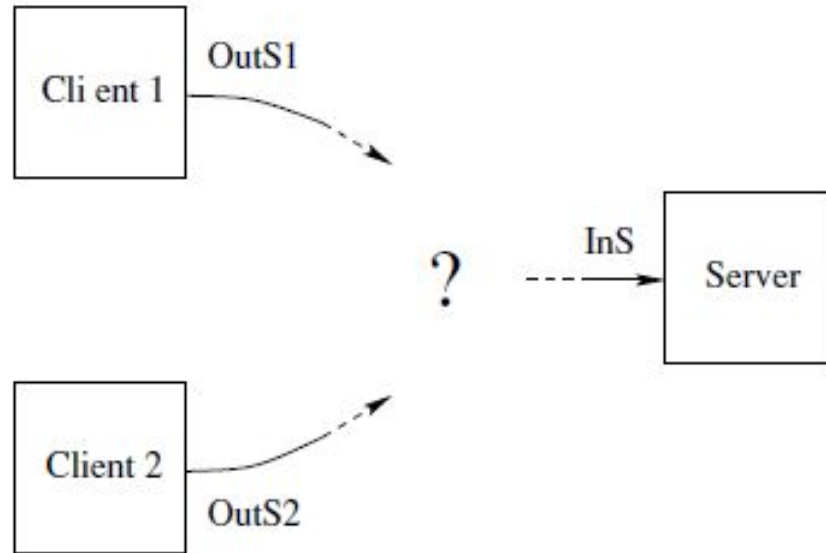
Clase 5: Message passing concurrency

Problemas

Sobre el productor consumidor previamente visto:

- ¿podemos poner varios productores?

Modelo cliente servidor



Canales de comunicación asincrónica

Async channels

Puerto (TDA)

- Canal asincrónico
- Tiene un stream asociado
- Stream object -> Port object

Sintaxis

Tenemos dos nuevos statements válidos

```
{NewPort <xs> <p>}
```

Port creation

```
{Send <p> <m>}
```

Message sending

Sintaxis

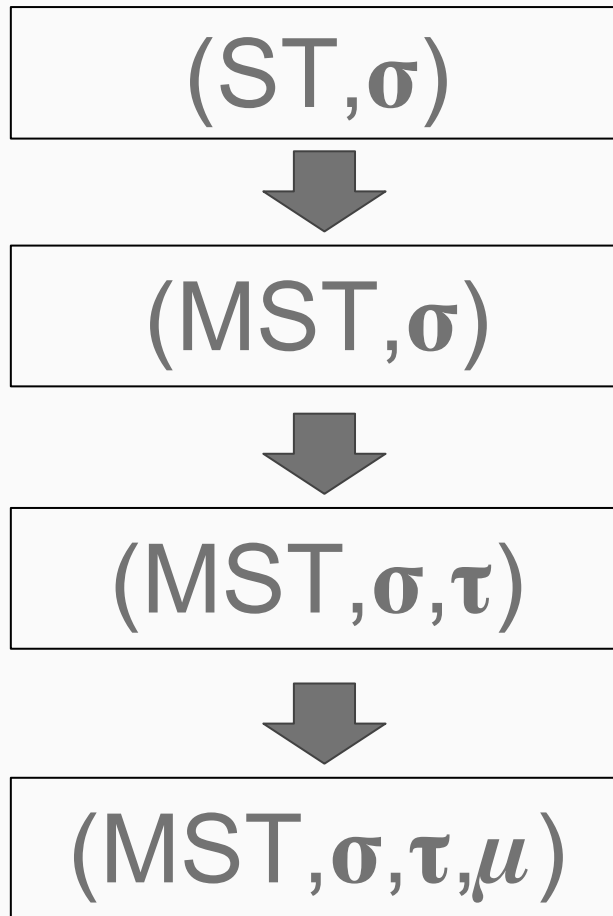
```
local Puerto S Enviar Recibir in
  proc {Enviar N Puerto Limit}
    if N < Limit then {Send Puerto N}{Enviar N+1 Puerto Limit}
    End
  end
  proc {Recibir S}
    case S of H|T then {Browse H} {Recibir T} end
  end
  {NewPort S Puerto}
  thread {Recibir S} end
  thread {Enviar 1 Puerto 10} end
end
```

Semántica

Mutable store

μ

Aparece un nuevo store, mutable store (μ)



Semántica - Port creation

En el tope tenemos el siguiente semantic statement

$(\{NewPort\ <x> \<y>\}, E)$

1. Se crea un puerto con nombre n
2. Bind $E(\<y>)$ y n en el store
3. Se agrega el par $(E(\<y>) : E(\<x>))$ al store μ

Si llegan a existir problemas con el bind del paso 2 se hace un raise con error

Semántica - Message sending

En el tope tenemos el siguiente semantic statement

$(\{Send \langle x \rangle \langle y \rangle\}, E)$

1. Se activa si $E(\langle x \rangle)$ está determinado
 - a. Si $E(\langle x \rangle)$ no está ligada al nombre de un puerto, se levanta un error.
 - b. Si existe el par $(E(\langle x \rangle) : z)$ en el store μ
 - i. Se crea un z' en el store SIGMA
 - ii. Se actualiza en el store μ el par con $(E(\langle x \rangle) : z')$
 - iii. Se hace un bind de z a una lista nueva $E(\langle y \rangle)|z'$
2. Si no está determinado se suspende

Semántica - Memoria

Se cambia el manejo de memoria por la inclusión del mutable store

Una variable y sera alcanzable, si existe en el store un par $(x:y)$ y x es alcanzable

Si existe el par $(x:y)$ y x es inalcanzable, puede borrarse el par

Port Objects

Definimos dos nueva abstracciones

NewPortObject : TDA Puerto con estado interno

NewPortObject2: TDA Puerto sin estado interno

NewPortObject - Con estado interno

```
fun {NewPortObject Init Fun}  
  local MsgLoop Sin in  
    MsgLoop = proc {$ S1 State}  
      case S1 of Msg|S2 then  
        {MsgLoop S2 {Fun Msg State}}  
      end  
    end  
    thread {MsgLoop Sin Init} end  
    {NewPort Sin}  
  end  
end
```

NewPortObject2 - Sin estado interno

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

Simple Message Protocols

Actor model

Hewitt's Actor Model (Hewitt, 1977)

Actor model: Define los cálculos como intercambio de mensajes entre actores (basado en SmallTalk)

Un actor tiene las siguientes propiedades:

- Sociable: Envía mensajes

- Reactivo: Solo actúa cuando recibe un mensaje - *message driven*

- Identificable: Tiene un nombre, una dirección

- Comportamiento: Frente al mensaje se comporta de una manera determinada

- Estado: Pueden tener estado interno

Agents

Los agentes son una extensión del modelo de actor.

Se basan en CAS (Complex adaptive system)

Lo que hacen los agentes es más complejo. El comportamiento se va adaptando.

Herramientas conocidas: JADE, Repast

RMI (Remote Method Invocation)

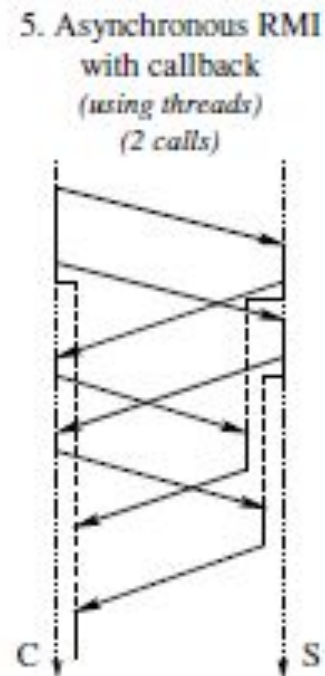
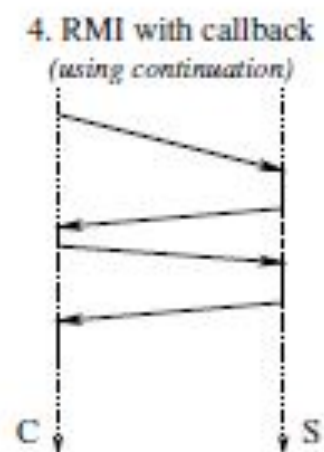
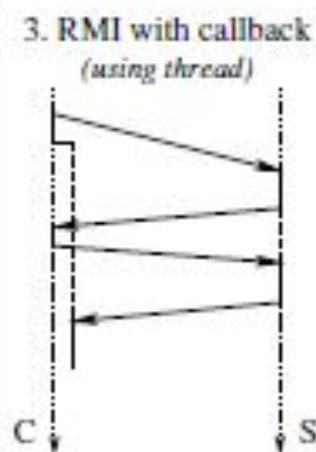
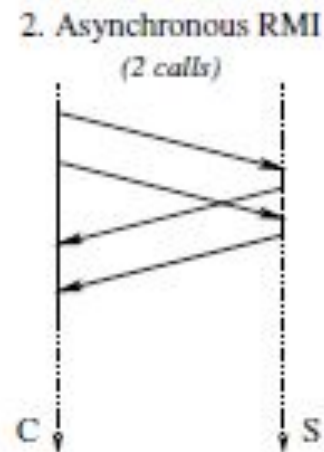
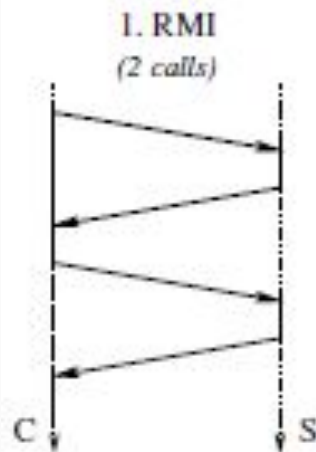
Invocación de métodos remota a través de mensajes

Existen varias implementaciones (ligado a la implementación de puertos)

- Sync / Async
- Server secuencial o concurrente
- Con o sin callbacks

RPC vs RMI

Protocolos de intercambio de mensajes basados en RMI



idle
suspended
active

Thread states

Analicemos el siguiente código - Caso 1

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

Analicemos el siguiente código - Caso 1

```
ClienteProc = proc {$ Msg}
  case Msg of
    work(Y)
  then Y1 Y2 in
    {Send Server calc(4 Y1)}
    {Send Server calc(5 Y2)}
    Y = Y1+Y2
  end
end
Cliente = {NewPortObject2 ClienteProc}
```

```
{Send Cliente work(R)}
{Browse R}
```

Caso 2 - Hagamos el cliente sincrónico!

```
ClienteProc = proc {$ Msg}
  case Msg of
    work(Y)
  then Y1 Y2 in
    {Send Server calc(4 Y1)}
    {Wait Y1}
    {Send Server calc(5 Y2)}
    {Wait Y2}
    Y = Y1+Y2
  end
end
```

```
{Send Cliente work(R)}
{Browse R}
```

Caso 3 - Hagamos el server no secuencial!

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

Caso 4 - RMI con callbacks + threads (Server)

```
ServerProc = proc {$ Msg}
  case Msg of
    calc(X Client) then
      local Y in
        Y =X*X+1
        {Send Client res(Y)}
      end
    end
  end
end
Server = {NewPortObject2 ServerProc}
```


Caso 4 - RMI con callbacks + threads (Cliente)

```
ClienteProc = proc {$ Msg}
  case Msg of
    work(Y) then
      {Send Server calc(Y Cliente)}
    [] res(R) then
      {Browse R}
    end
  end
end
```

```
Cliente = {NewPortObject2 ClienteProc}
{Send Cliente work(10)}
```

Caso 5 - RMI con callbacks intermedios + threads (Server)

```
ServerProc = proc {$ Msg}
  case Msg of
    calc(X Y Client) then
      local D in
        {Send Client part(D)}
        Y =X*X+D
      end
    end
  end
end
Server = {NewPortObject2 ServerProc}
```

Caso 5 - RMI con callbacks intermedios + threads (Cliente)

```
ClienteProc = proc {$ Msg}
  case Msg of
    work(Y) then
      {Send Server calc(4 Y Cliente)}
    [] part(D) then
      D = 10
    end
  end
end
Cliente = {NewPortObject2 ClienteProc}
```

```
{Send Cliente work(R)}
{Browse R}
```

Caso 6 - RMI con callbacks intermedios que continua + threads (Cliente)

```
ClienteProc = proc {$ Msg}
  case Msg of
    work(Y) then
      local Y1 Y2 in
        {Send Server calc(4 Y1 Cliente)}
        thread Y = Y1 + 1 end
      end
    [] part(D) then D = 10 end
  end
Cliente = {NewPortObject2 ClienteProc}
```

```
{Send Cliente work(R)}
{Browse R}
```

Analizando el caso 5

El cliente podría hacer el llamado tanto en forma sincrónica como asincrónica.

¿Por qué el cliente usa threads después del Send?

Caso 7 - RMI con callbacks intermedios + record continuation (Server)

```
ServerProc = proc {$ Msg}
  case Msg of
    calc(X Client Cont) then
      local D X1 in
        {Send Client part(D)}
        X1 =X*X+D
        {Send Client Cont#X1}
      end
    end
  end
end
Server = {NewPortObject2 ServerProc}
```

Caso 7 - RMI con callbacks intermedios + record continuation (Cliente)

```
ClienteProc = proc {$ Msg}
  case Msg of
    work(Y) then
      {Send Server calc(4 Cliente cont(Y))}
      [] cont(Y)#Z then
        Y = Z+1
      [] part(D) then
        D = 100
      end
    end
  end
  end
  Cliente = {NewPortObject2 ClienteProc}
```

```
{Send Cliente work(R)}
{Browse R}
```

Bibliografía

- **Concepts, Techniques, and Models of Computer Programming - Capítulo 5**, Peter Van Roy and Seif Haridi
- **Extras:**
 - **An Introduction to MultiAgent Systems**, Wooldridge
 - **Principles of Concurrent and Distributed Programming**, M. Ben-Ari
 - **Agent-based modeling and simulation**, North and Macal
 - **Managing Business Complexity, discovering Strategic Solutions with Agent-Based Modeling and Simulation**, North and Macal