Teoría de Lenguajes
Teoría de la Programación

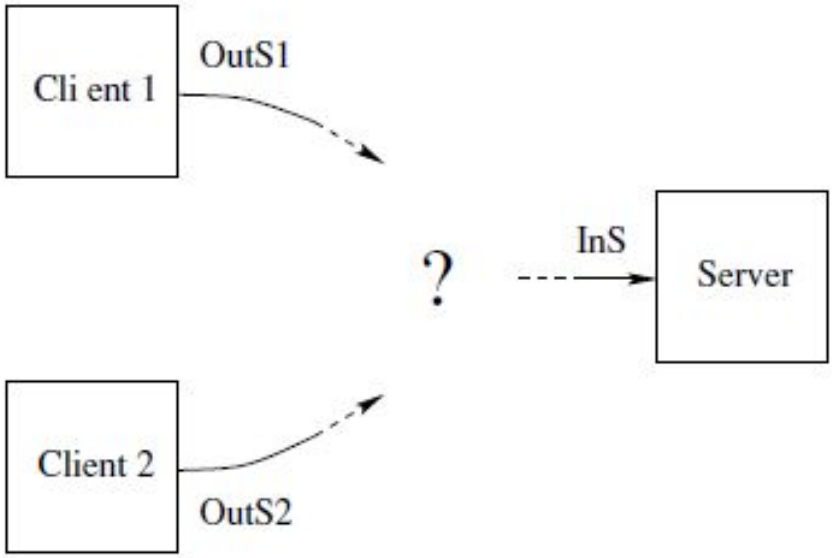
Clase 6: Message passing concurrency

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 <x_s> <p>}
```

Port creation

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

Message sending

Sintaxis

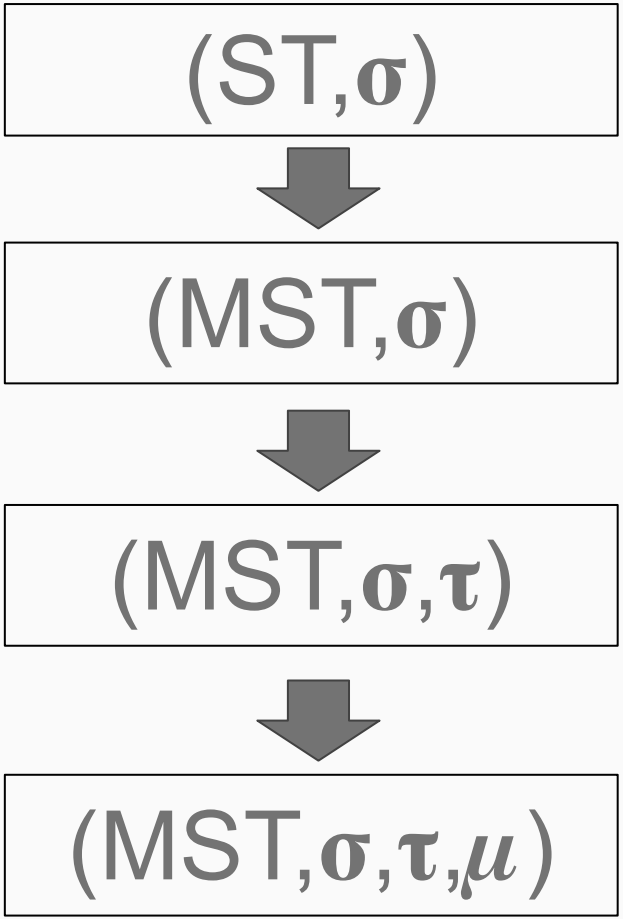
```
local Puerto S Enviar Recibir in
   proc {Enviar N Puerto Limit}
        if N < Limit then {Send Puerto N}{Enviar N+1 Puerto Limit}
        End
   end
   proc {Recibir S}
        case S of H|T then {Browse H} {Recibir T} end
   end
   {NewPort S Puerto}
   thread {Recibir S} end
   thread {Enviar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

1. Se activa si E(<x>) está determinado
   a. Si E(<x>) no está ligada al nombre de un puerto, se levanta un error.
   b. Si existe el par (E(<x>) : 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(<x>) : z')
      iii. Se hace un bind de z a una lista nueva E(<y>)|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

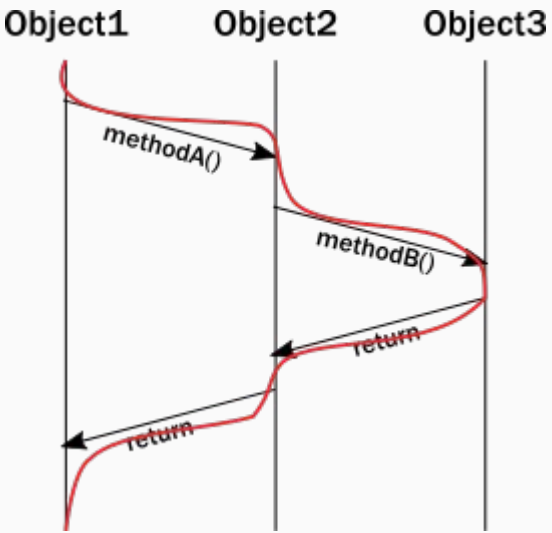
```
fun {NewPortObject Init Fun}
    local MsgLoop Sin in
        MsgLoop = proc {$ S1 State}
            case S1 of Msg|S2 then
                {MsgLoop S2 {Fun Msg State}}
            end
        end
        thread {MsgLoop Sin Init} end
        {NewPort Sin}
    end
end
```

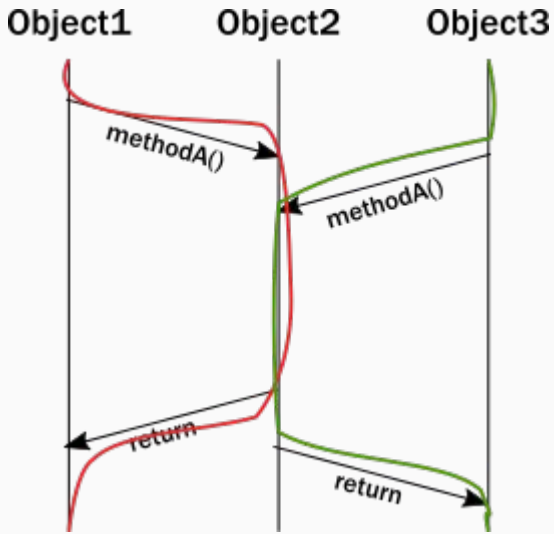
NewPortObject2 - Sin estado interno

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

Actores

Hilo de ejecución





El hijo de ejecución atraviesa todos los objetos involucrados. Si hay más de un hilo, el mismo código se ejecuta concurrentemente, generando posibles problemas con el estado interno

Actor model

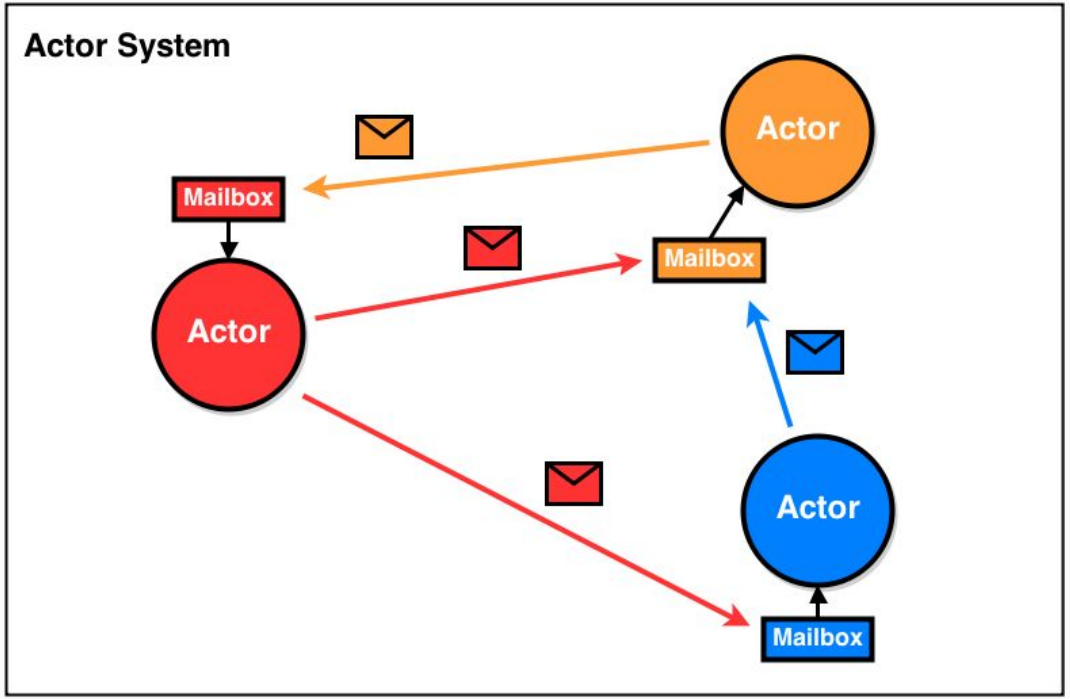
Hewitt's Actor Model (Hewitt, 1977)

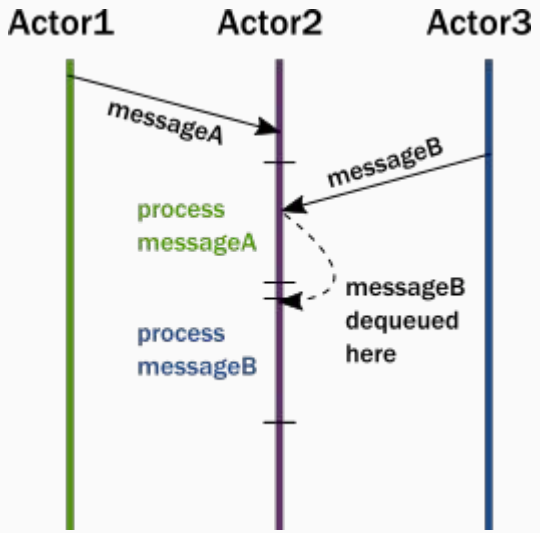
Actor model: Define los cómputos como intercambio de mensajes entre actores (basado en SmallTalk)

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

Actor model


Actor System
Actor
Mailbox
Mailbox
Actor
Actor
Mailbox


Actor1
Actor2
Actor3
messageA
messageB
process
messageA
messageB
dequeued
here
process
messageB

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

Simple Message Protocols

RMI (Remote Method Invocation)

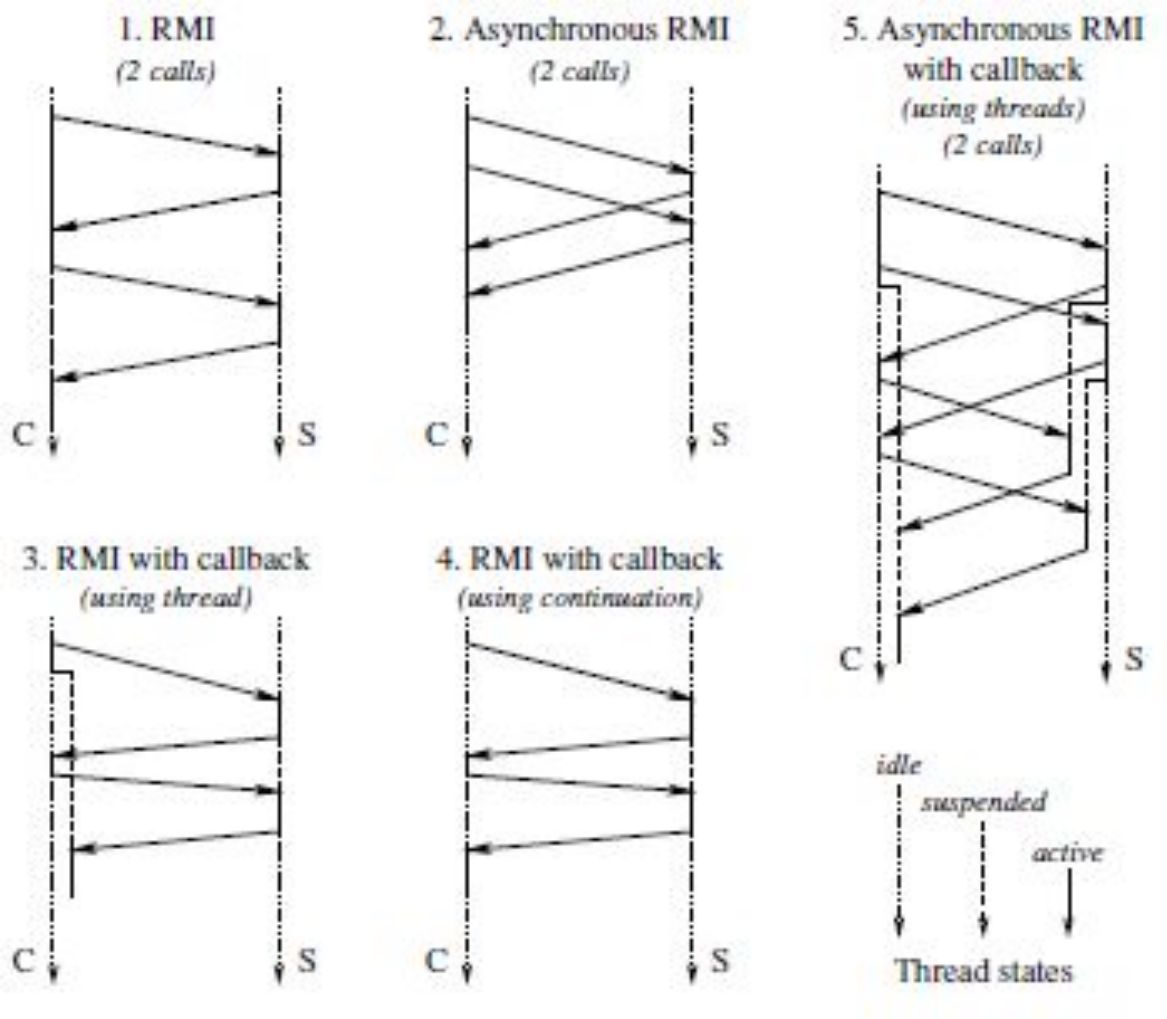
Invocación de métodos remota a través de mensajes

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



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
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
```

```
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
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
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
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
   Cliente = {NewPortObject2 ClienteProc}
```

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

Estado Explicito

Estado

¿Qué es el estado?

- Valores que contienen resultados intermedios

Puede ser:

- Implícito
- Explícito

Estado implícito (declarativo)

Estado compuesto por:

- Argumentos
- Variables libres

En el modelo declarativo el estado la clausura de los procedimientos. Esta dado por el entorno.

```
fun {SumList Xs S}
   case Xs of H|T then {SumList T H+S}
   else S end
end
```

Estado explícito

El estado excede al llamado del procedimiento

Celdas de memoria!

Estado explícito - abstracción de datos

Posibilidad de encapsular estados en tipos de datos:

- Interfaz mas sencilla
- Efectos secundarios

Estado explícito - abstracción de datos (ej)

```
local C in
   C = {NewCell 0}
   fun {SumList Xs S}
 C:=@C+1
 case Xs of H|T then {SumList T H+S}
 else S end
   end
   fun {SumCount} @C end
end
```

Sintaxis

Tenemos dos nuevos statements válidos

```
{NewCell <x> <c>}
```

Cell creation

```
{Exchange <c> <x> <m>}
```

Cell exchange

Sintaxis - Cell creation

{NewCell X C}

Crea una nueva celda C con contenido inicial X

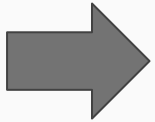
```
local C C2 C3in
   C = {NewCell 0}
   {NewCell 10 C2}
   C3 = {NewCell nil}
end
```

Sintaxis - Cell exchange

{Exchange C X Y}

Bind X con el valor actual de C y set de Y a nuevo valor de C

```
{Exchange C X Y}
```



```
X = @C
C:=Y
```

```
X=C:=Y
```

Sintaxis - Cell exchange

```
local C1 X Y Z in
   C1 = {NewCell 0}
   {Browse @C1}
   {Exchange C1 X 4}
   {Browse X#@C1}
   {Exchange C1 _ 8}
   {Exchange C1 Y @C1}
   {Browse Y#@C1}
   Z=C1:=10
   {Browse Z#@C1}
end
```

Semántica - Cell creation

En el tope tenemos el siguiente semantic statement
({NewCell <x> <y>}, E)

1. Se crea una celda 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 - Cell exchange

En el tope tenemos el siguiente semantic statement
({Exchange <x> <y> <z>}, E)

1. Se activa si E(<x>) está determinado
   a. Si E(<x>) no está ligada al nombre de una celda, se levanta un error.
   b. Si existe el par (E(<x>) : w) en el store μ
      i. Se actualiza en el store μ el par con (E(<x>) : E(<z>))
      ii. Se hace un bind de E(<y>) a w en el store σ
2. Si no está determinado se suspende

Sharing, aliasing & equality

Sharing

Dos identificadores se refieren a la misma celda. A veces conocido como aliasing.

Se cambia el valor de una celda, cambian ambos

Equality

Dos valores tienen la misma estructura y valores en todas sus partes.

Dos celdas son iguales no por su valor, sino porque son la misma celda

Modelo declarativo con estado

Es posible escribir un componente con estado, que desde afuera se comporte en forma declarativa.

Se puede utilizar también para **memoization**

Abstracción de datos

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