Lección 10: Coordinación mediante espacios de tuplas

- Introducción
- El modelo de coordinación Linda
- Ejemplos
- Acceso a un espacio de tuplas en el mundo real
- Aproximación conceptual a los monitores
- Ejercicios

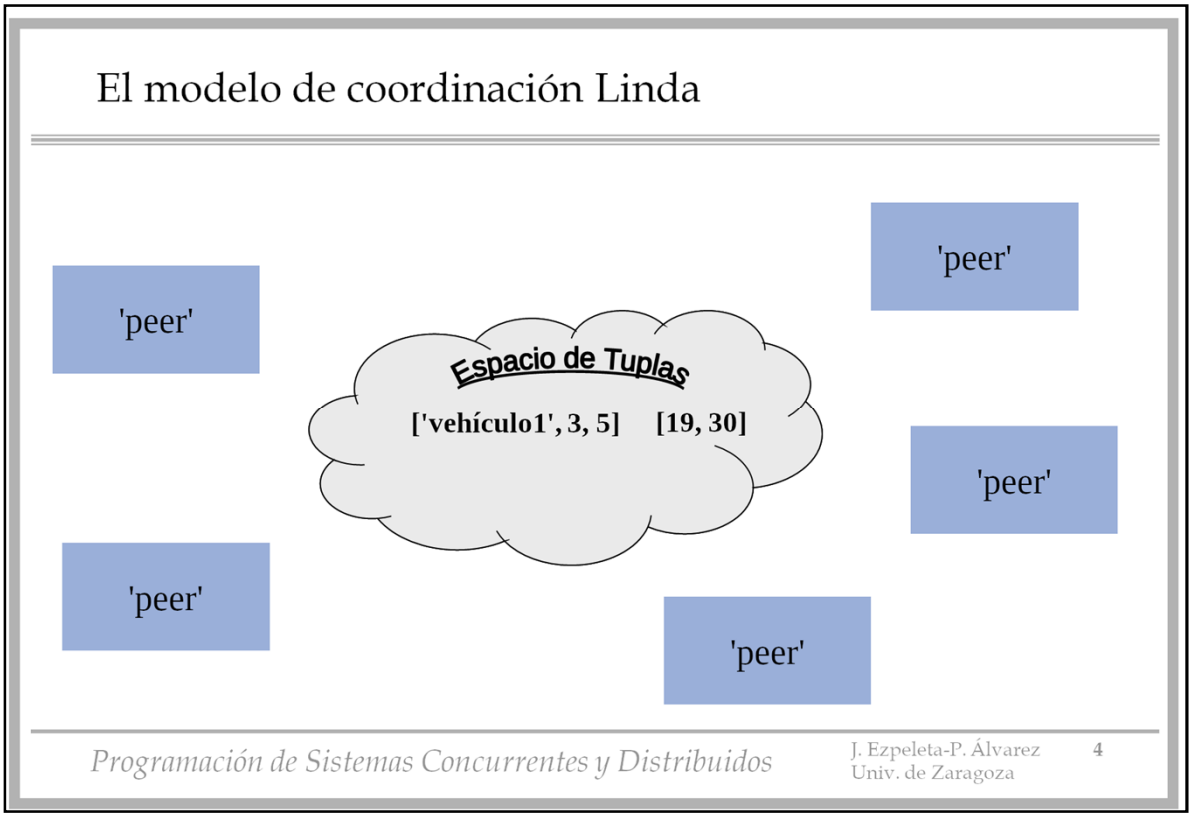
Introducción

- Comunicación síncrona implica fuerte acoplamiento
  - tiempo y espacio
  - ambos procesos han de existir
  - la estructura de red ha de ser conocida por los procesos
- Modelo de coordinación Linda
  - desacoplar procesos
  - almacenamiento persistente de datos

El modelo de coordinación Linda

- David Gelernter, 1985
  Generative communication in Linda
  ACM Transactions on Programming Languages and Systems
  Volume 7 , Issue 1 (January 1985), pp. 80-112
- **Linda** es un lenguaje/sistema de coordinación
- Basado en un espacio "lógico" de memoria compartida: el ***espacio de tuplas***
- Los procesos se coordinan:
  - introduciendo tuplas en el espacio
  - retirando, *de manera selectiva*, tuplas del espacio
  - consultando la existencia de tuplas en el espacio
- ¿Qué son tuplas?
  - versión general: una especie de listas anidadas (árboles)
    - listas con elementos que pueden ser listas
  - versión original: una especie de listas planas

JavaSpaces
GigaSpaces
TSpaces
JXTASpaces
XMLSpaces
C-Linda

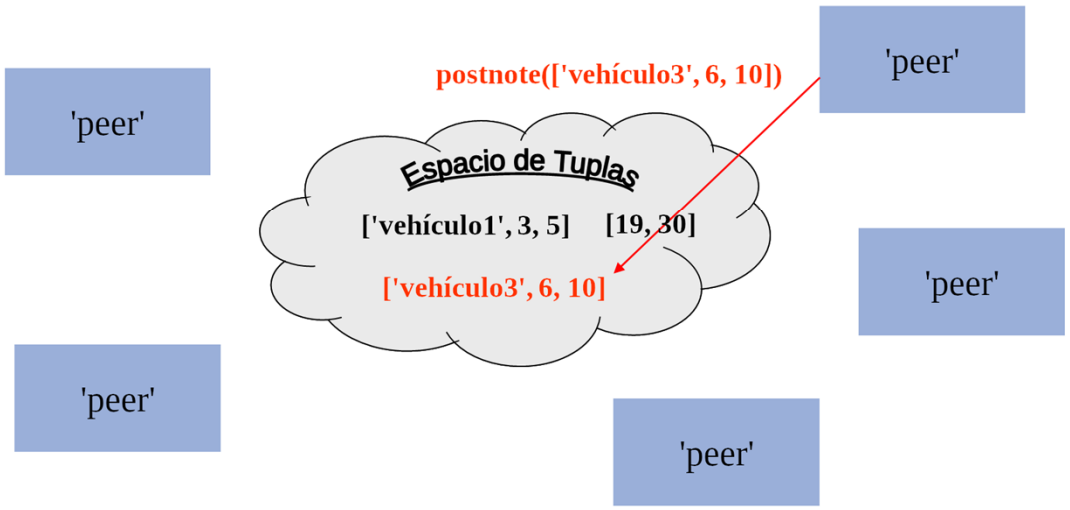


Conceptos: (1) tupla y (2) espacio de tuplas.

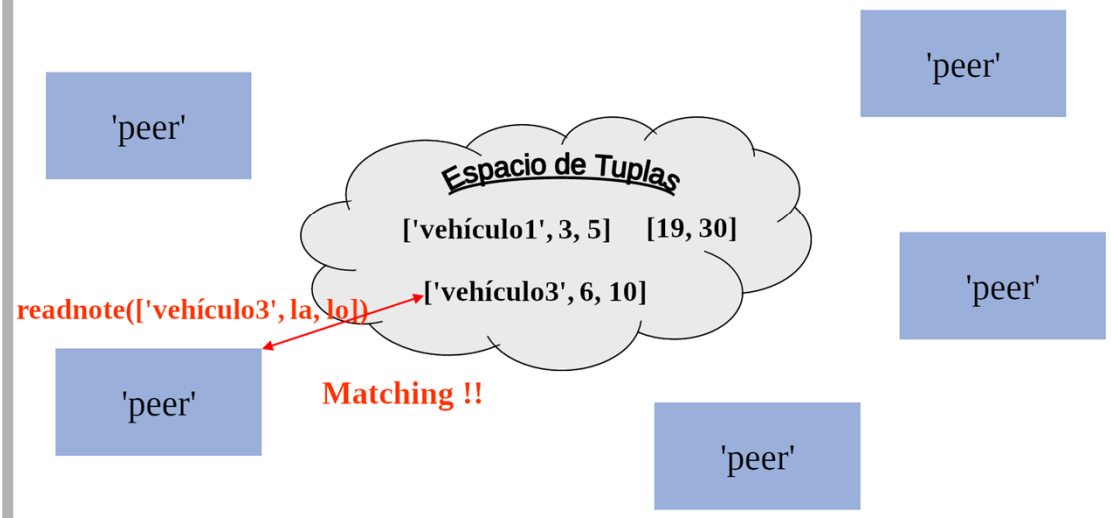
La recuperación de tuplas del espacio es por medio de un mecanismo de busqueda asociativa (busqueda por contenido, en vez de por identificadores o por claves)

Tres operaciones que facilitan la interaccion entre procesos: write/read/take

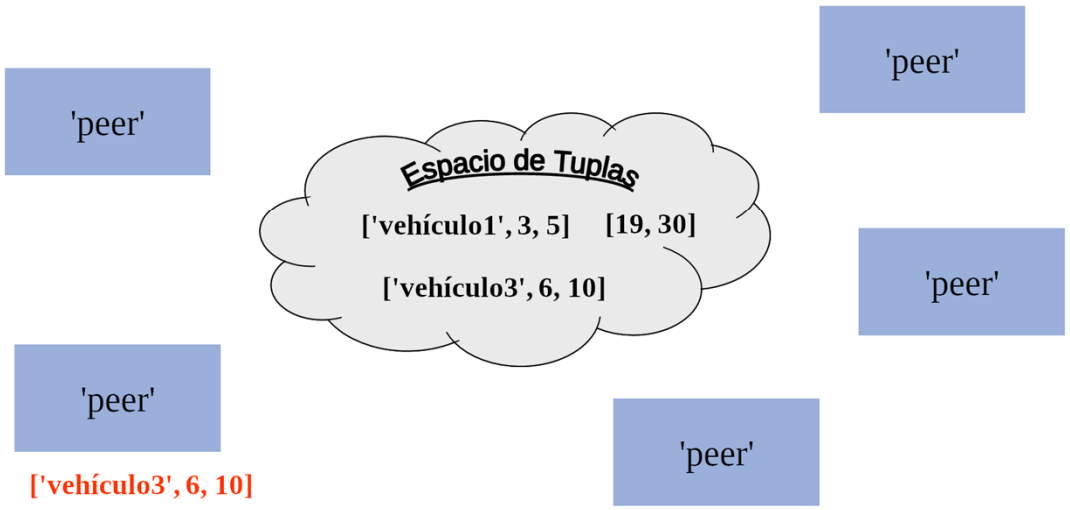
El modelo de coordinación Linda



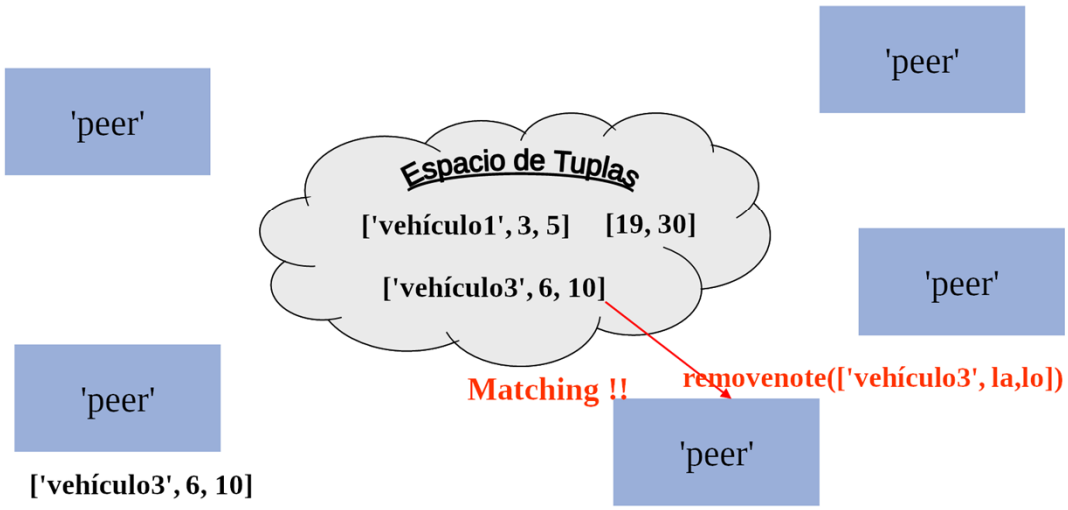
El modelo de coordinación Linda



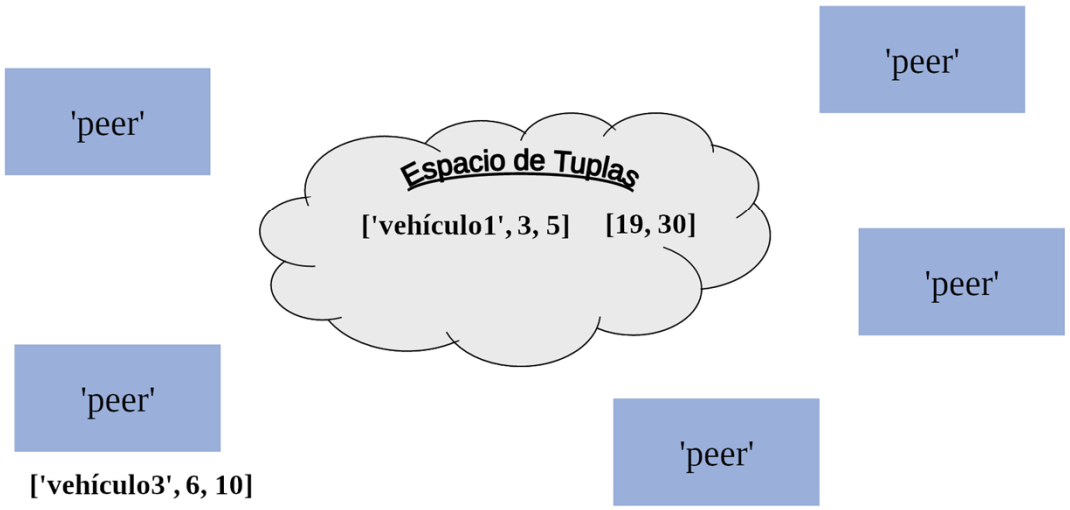
El modelo de coordinación Linda



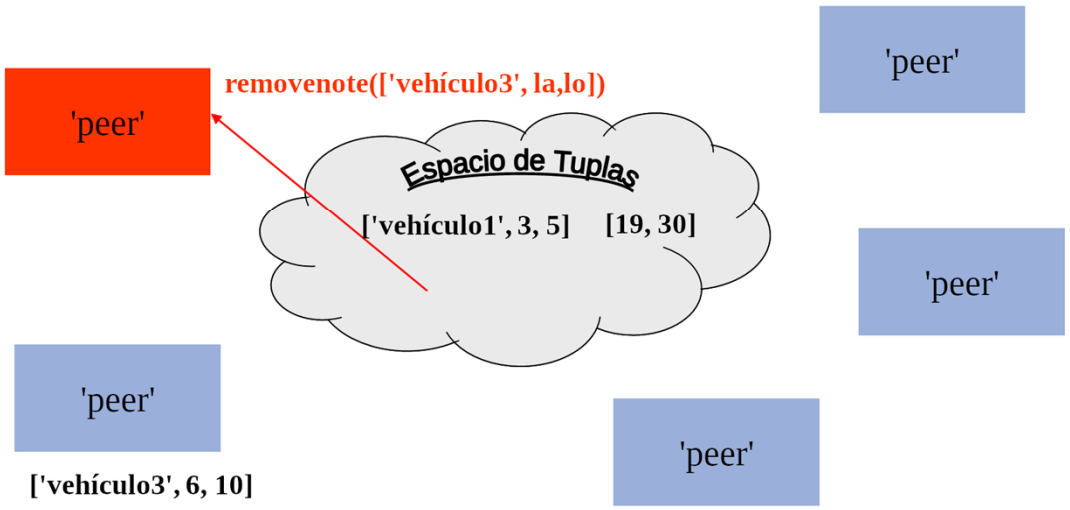
El modelo de coordinación Linda



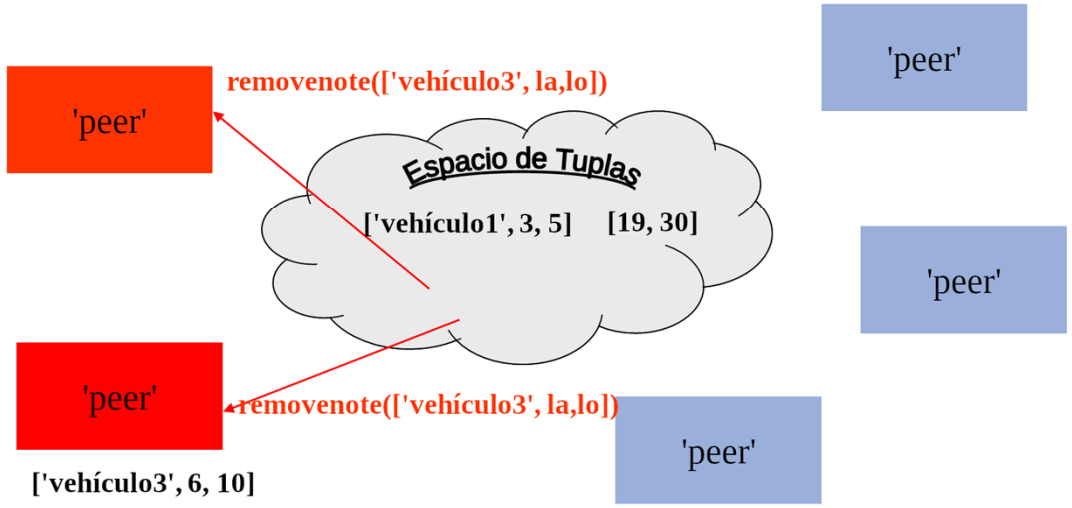
El modelo de coordinación Linda



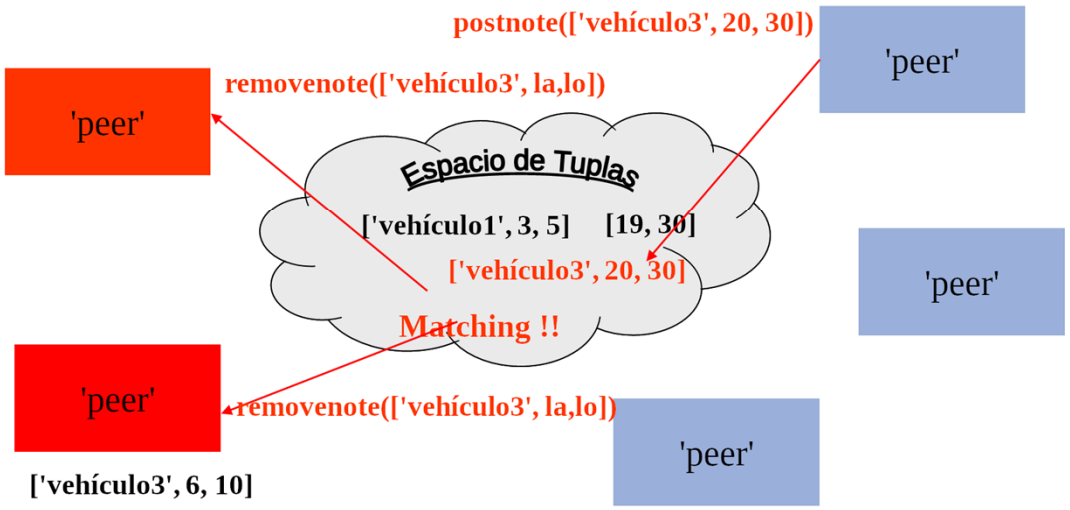
El modelo de coordinación Linda



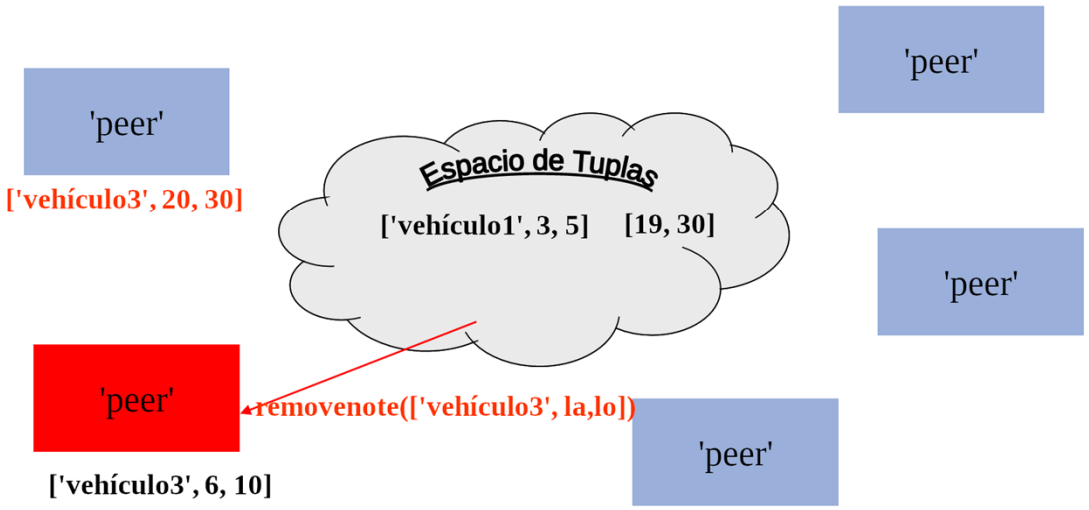
El modelo de coordinación Linda



El modelo de coordinación Linda



El modelo de coordinación Linda



El modelo de coordinación Linda

- Habitualmente se definen tres operaciones sobre un espacio de tuplas:
  - **postnote([exp1,exp2,...])** — out, write
    - t es una tupla; el proceso deposita la tupla y sigue (**no bloqueante**)
  - **removenote([v_v1,v_v2,...])** — in, take
    - **v_v*** es un valor, una variable o un parámetro formal
    - el proceso se **bloquea** hasta que el espacio de tuplas le asigna una que corresponda al patrón
    - la tupla es quitada del espacio de tuplas
  - **readnote([v_v1,v_v2,...])** — read
    - versión del "removenote" en que la tupla que le es asignada no se elimina del espacio de tuplas
    - puede haber varios read simultáneos

¡atómicas!

Ejemplo: Un semáforo

- Implementación de las operaciones de semáforo

```
semaphore s := 0
semaphore s2 := 3
```

```
process P
   wait(s2)
   wait(s2)
   wait(s)
```

```
process Q
   wait(s2)
   signal(s)
```

Ejemplo: Un semáforo

- Implementación de las operaciones de semáforo

```
semaphore s := 0
semaphore s2 := 3
```

```
process P
   wait(s2)
   wait(s2)
   wait(s)
```

```
process Q
   wait(s2)
   signal(s)
```

```
process init
   for i:=1..3
      postnote(['s2'])
```

```
process P
   removenote(['s2'])
   removenote(['s2'])
   removenote(['s'])
```

```
process Q
   removenote(['s2'])
   postnote(['s'])
```

Ejemplo: Un semáforo

- Implementación de las operaciones de semáforo

```
semaphore s := 0
semaphore s2 := 3
```

```
process P
   wait(s2)
   wait(s2)
   wait(s)
```

```
process Q
   wait(s2)
   signal(s)
```

```
process init
   for i:=1..3
      postnote(['s2'])
   postnote(['P'])
   postnote(['Q'])
```

```
process P
   removenote(['P'])
   removenote(['s2'])
   removenote(['s2'])
   removenote(['s'])
```

```
process Q
   removenote(['Q'])
   removenote(['s2'])
   postnote(['s'])
```

Ejemplo: Un semáforo

- Implementación de las operaciones de semáforo

```
semaphore s := 0
semaphore s2 := 3
```

```
process P
   wait(s2)
   wait(s2)
   wait(s)
```

```
process Q
   wait(s2)
   signal(s)
```

```
process init
   for i:=1..3
      postnote(['s2'])
   postnote(['init'])
```

```
process P
   readnote(['init'])
   removenote(['s2'])
   removenote(['s2'])
   removenote(['s'])
```

```
process Q
   readnote(['init'])
   removenote(['s2'])
   postnote(['s'])
```

Primeros ejercicios

- Ejercicio 1: Resolver el problema de la "2-exclusión mutua": no más de procesos en la sección crítica simultáneamente

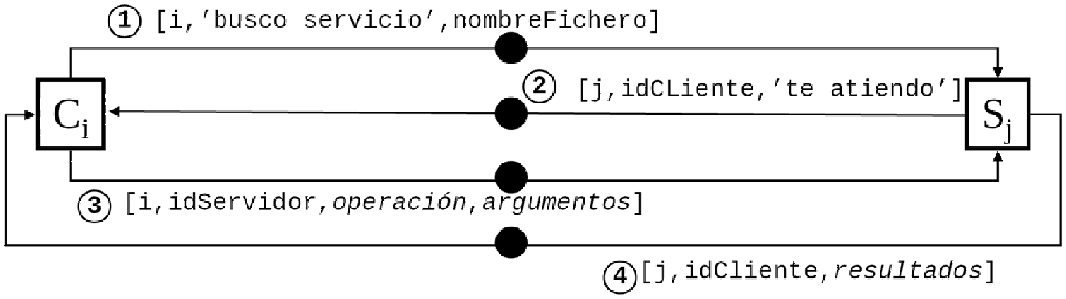
```
process P(i:1..N)::
   loop

      //SC

   end
```

- Ejercicio 2: resolver el mismo problema, pero usando únicamente una tupla

Ejemplo: Servidor de ficheros

- Es un ejemplo de arquitectura cliente-servidor
- Sistema que puede mantener hasta n ficheros abiertos simultáneamente
- Compuesto por:
  - n servidores
  - m clientes
- Tal que:
  - cliente pide atención por parte de un servidor (cualquiera) para acceder a un fichero
  - un servidor libre puede atender las peticiones de un cliente
  - las operaciones sobre el fichero son pedidas por el cliente y atendidas por el servidor

Ejemplo: Servidor de ficheros



Ejemplo: Servidor de ficheros

```
process cliente(i:1..m)
   integer idServidor
   string nombreFichero := ... --el que sea
  ...
  postnote([i,'busco servicio',nombreFichero])
  removenote([idServidor,i,'te atiendo'])
     --esquema [from,to,...]
  while me_da_la_gana
     postnote([i,idServidor,operación,argumentos])
     removenote([idServidor=,i=,resultados])
     ...
```

Ejemplo: Servidor de ficheros

```
process servidorDeFicheros(j:1..n)
    string nomFich
    integer idCliente

   loop
     removenote([idCliente,'busco servicio',nomFich])
     postnote([j,idCliente,'te atiendo'])
     while me_da_la_gana
        removenote([idCliente=,j=,operacion,argumentos])
        --procesar operación y obtener resultados
        postnote([j,idCliente,resultados])
```

Ejercicios

- **Ejercicio** : Dar una solución al problema de los filósofos utilizando Linda como medio para la sincronización
  - versión 1: cada proceso coge sus propios tenedores
  - versión 2: hay un proceso servidor de tenedores, que concede los dos simultáneamente

7.2 Problem: the wavefront computation of a matrix

N. Carriero and D. Gelernter
*"**How to write parallel programs. A first course**"*
The MIT Press, 1990

Given a function $h(x, y)$, we need to compute a matrix H such that the value in the ith row, jth column of H—that is, the value of $H[i, j]$ (in standard programming language notation) or $H_{i,j}$ (in mathematical notation)—is the function h applied to i and j, in other words $h(i, j)$.

The value of $h(i, j)$ depends on three other values: the values of $h(i-1, j)$, of $h(i, j-1)$ and of $h(i-1, j-1)$. To start off with, we're given the value of $h(0, j)$ for all values of j, and of $h(i, 0)$ for all values of i. These two sets of values are simply the *two strings that we want to compare.* In other words, we can understand the computation as follows: draw a two-dimensional matrix. Write one of the two comparands across the top and write the other down the left side. Now fill in the matrix; to fill in each element, you need to check the element on top, the element to the left, and the element on top and to the left.

Ejercicios

- **Ejercicio**: Dos tipos de procesos, A y B pueden entrar y salir de una habitación común. Un proceso A no puede salir de la habitación hasta que haya coincidido en ella, simultáneamente, con dos B, mientras que un B no puede salir hasta que haya coincido con al menos un A. Se pide desarrollar el controlador de acceso a la habitación, así como el código de los procesos de tipo A y B. La comunicación de los procesos con el controlador debe hacerse mediante Linda.

Ejercicios

- **Ejercicio**: Nuestro sistema informático dispone de dos impresoras, I1 e I2, parecidas pero no idénticas. Éstas son usadas por tres tipos de procesos cliente. Los del primer tipo requieren la impresora I1, los del segundo la I2, mientras que los del tercero cualquiera de las dos. Usando Linda como medio de coordinación, desarrollar el código de los tres tipos de impresoras y del controlador de las impresoras. La solución debe ser equitativa, asumiendo que un proceso que toma una impresora la libera.

Acceso a un espacio de tuplas en el mundo real

```
<tupleDescription>
<tuple>
  <string>esto</string>
  <tuple>1</tuple>
  <string>prueba</string>
</tuple>
</tupleDescription>
```

```
import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import javax.xml.namespace.QName;

public class client {
public static void main(String [] args) {
   try {
       String endpoint = "http://luna1.cps.unizar.es:8080/axis/services/RLindaWS";
       Service  service = new Service();
       Call call = (Call) service.createCall();
       call.setTargetEndpointAddress( new java.net.URL(endpoint) );
       call.setOperationName(new QName("http://soapinterop.org/", "RLindaOUT"));
       String value = (String) call.invoke(new Object[] {"[\"esto\",[1],\"prueba\"]"});
       System.out.println(value);
   } catch (Exception e) {
       System.err.println(e.toString());
   }
}
}
```

```
postnote(['esto',[1],'prueba'])
```

Aproximación conceptual a los monitores

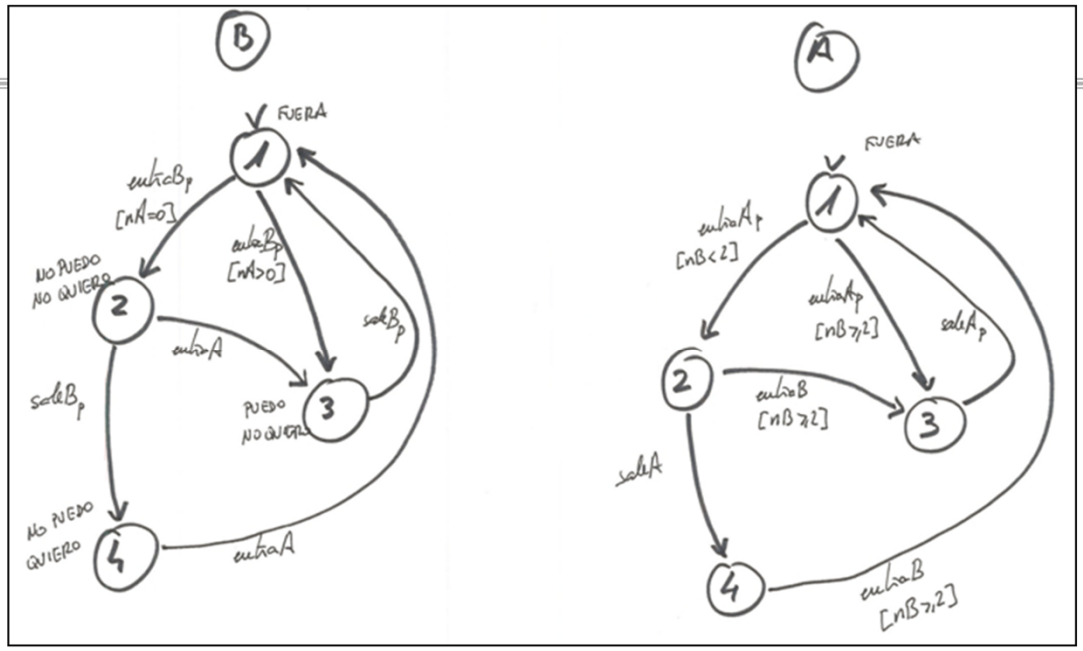
- Es fácil implementar mediante el espacio de tuplas una gestión de la concurrencia análoga a la hecha con monitores

```
variables permanentes
declaración proc.
llamadas a procs.
entrada a proc.
terminación proc.
wait
signal
cuerpo proc.
```

monitor

```
variables locales servidor
disponibilidad de servicio
envío tupla solicitud de servicio
“removenote” de la solicitud
“postnote” de la respuesta
encolar peticiones como pendiente
procesa petición pendiente en cola
“Switch” sobre solicitudes
```

proceso servidor


B
FUERA
entraB_p
[nA=0]
NO PUEDO
NO QUIERO
2
entraB_p
[nA>0]
sale B_p
entra A
sale B_p
PUEDO
NO QUIERO
3
NO PUEDO
QUIERO
4
entra A
A
FUERA
entraA_p
[nB<2]
entraA_p
[nB>=2]
sale A_p
2
entra B
[nB>=2]
3
sale A
4
entra B
[nB>=2]

Ejercicio 3 (2.50 ptos.)

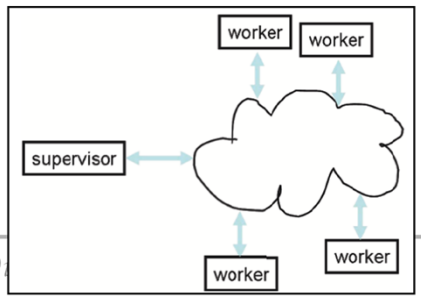
Un jardín tiene dos puertas de acceso. Cada puerta tiene instalado un torno responsable de gestionar las entradas y salidas del jardín. Debido a su reducido tamaño no puede haber más de 50 personas simultáneamente visitándolo. Inicialmente, hay 100 personas interesadas en realizar la visita. Cada vez que una persona quiere acceder al jardín debe comprar un ticket de entrada. En el ticket se le especifica la puerta concreta por la que debe entrar y salir del jardín.

Todas las personas se comportan de la misma manera, repitiendo la siguiente secuencia de acciones durante un número aleatorio de veces: primero, compran un ticket; segundo, solicitan a la puerta asignada permiso para entrar en el jardín y esperan a que ésta les dé acceso; tercero, visitan el jardín durante un cierto tiempo; y, finalmente, solicitan salir y esperan a que el torno les autorice la salida. Por otro lado, el funcionamiento de los tornos es muy simple: reciben las solicitudes de los usuarios (que ya tienen su ticket correspondiente) y les dan los permisos tanto de entrada como de salida.

Se pide implementar el sistema concurrente anteriormente descrito. El sistema consta de tres tipos de procesos: el proceso *expendedor de tickets*, los dos procesos *puerta*, y los 100 procesos *persona*. La comunicación y coordinación de los procesos debe ser programada utilizando Linda.

La figura 1 representa de manera esquemática el patrón de diseño *supervisor-worker*. Este esquema propone una organización sencilla en la que un proceso supervisor suministra tareas a los trabajadores a través de un medio común para el intercambio de información. Los procesos trabajadores van tomando las solicitudes, obtienen los resultados y los envía al supervisor, quedando a la espera de nuevas tareas encargadas por el proceso supervisor, o hasta que éste le indique que puede terminar. El supervisor toma los resultados y actúa en función a cuál sea su objetivo.

Utilizando Linda como medio de comunicación, se pide resolver el siguiente problema. Se trata de ordenar 100 ficheros de datos, de manera que un proceso supervisor suministra los nombres de los ficheros, y cuatro procesos trabajadores van tomando las órdenes depositadas por el supervisor y ordenan el fichero cuyo nombre se les ha suministrado. Cada trabajador va iterando ese proceso hasta que recibe del supervisor la orden de terminar. Dependiendo del entorno de ejecución, es posible que distintos procesos trabajador tengan distintas velocidades.



```
process supervisor
    constant natural N := 100
    string array[1..N] ficheros := (1..N, ...)
             --Nombres de ficheros a ordenar. Ya inicializada

    --completar de acuerdo al enunciado del ejercicio
end process

process worker(i:1..4)::
    operation ordena_fichero(string nom_fich)
       --Pre: el fichero en <nom_fich> existe
       --Post: el fichero en <nom_fich> está ordenado
       --Com: Asumimos que está implementada

          ...
    end operation

    --completar de acuerdo al enunciado del ejercicio
end process
```