# Lección 14: Programación concurrente mediante paso asíncrono de mensajes

- Operaciones sobre un canal
- Semántica formal
- Ejemplo de proceso filtro:
  - proceso “mezclar”
- El esquema cliente-servidor:
  - ejemplo: servidor de recursos compartidos
  - ejemplo: servidor de ficheros
- Un ejemplo “entre compañeros”:
  - extraer la estructura de una red
- Ejercicios

# Operaciones sobre un canal

- Un canal es un tipo abstracto de dato que ofrece operaciones para que los procesos puedan comunicar
- Declaración de canales:

```
nombreCanal: Canal(tipo)
```

- Operaciones sobre canales:

```
send nombreCanal(expresión, ... ,expresión)
receive nombreCanal(variable, ... ,variable)
empty(nombreCanal)
```

```
canEnt: Canal(real)
canSal: Canal(euros,pesetas:real; concepto:cadena)
```

# Operaciones sobre un canal

- Características:
  - canales ideales sin limitación de capacidad, sin duplicación ni pérdida de mensajes
    - implica que ***send*** es no bloqueante
  - operaciones de acceso a canales son *atómicas*
  - el efecto de ejecutar la operación *receive* sobre un canal suspende la ejecución del proceso receptor hasta que haya, al menos, un mensaje en el canal
  - para cada proceso, el orden de encolado es el mismo que el de envío
  - *empty* devuelve ***Verdad*** si y sólo si la cola de datos del canal está vacía (¡OjO!)
- Casos posibles:
  - **n/m** : buzón (*mailbox)*
  - **n/1** : puerto (*input port)*
  - **-1/m**: difusión *(broadcast)*
  - **-1/1** : enlace (*link)*

# Operaciones sobre un canal

- Ejemplo: exclusión mutua mediante canales

```
Vars buzon: Canal(entero)

Principio
   send buzon(1)
   P(1) || P(2) || ... || P(n)
Fin
```

```
P(i:1..n)::
  Vars p:Ent
  Mq True
    receive buzon(p)
    --seccion critica
    send buzon(p)
  FMq
```

# Semántica formal

- Notación. Sean $s_1$, $s_2$ dos secuencias de mensajes
  - **$s_1$+$s_2$**: concatenación
  - **$s_1$-$s_2$**: lo que queda de $s_1$ después de eliminar el prefijo $s_2$
  - **$s_1$ <= $s_2$**: ¿es $s_1$ un prefijo de $s_2$?
  - **pri($s_1$)**: primer elemento de $s_1$
- Algo de nomenclatura:
  - para poder afirmar algo sobre el estado de los canales usaremos las siguiente variables auxiliares
  - **env(c)**: secuencia de mensajes enviados al canal **c**
  - **rec(c)**: secuencia de mensajes recibidos desde el canal **c**
  - **env(c)-rec(c)**: secuencia de mensajes en el canal **c**

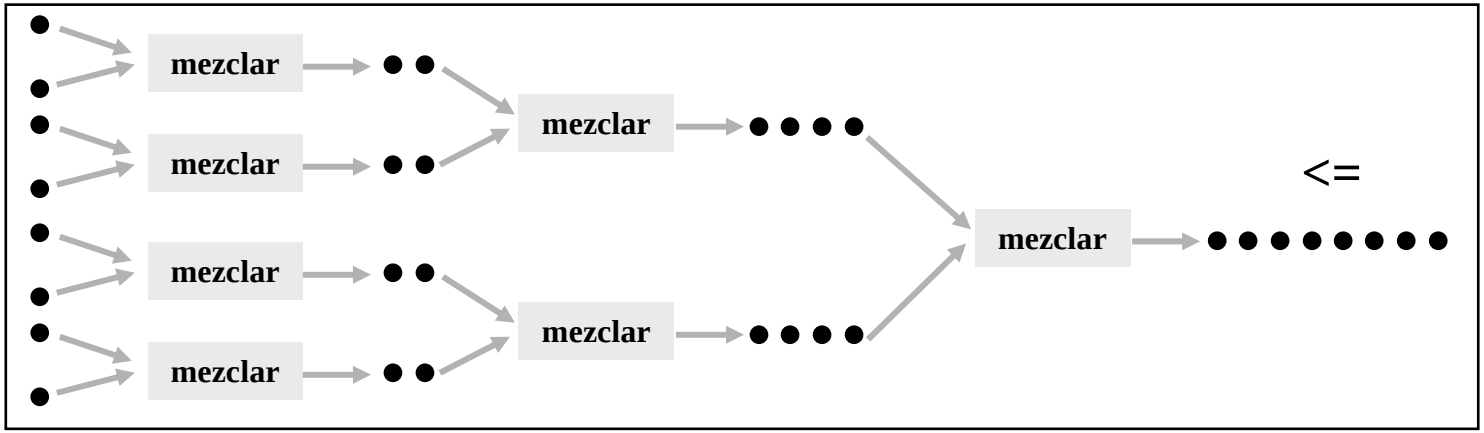
# Semántica formal

- **Axioma del canal:** $\texttt{rec(c)} \leq \texttt{env(c)}$
- **Axioma de *send*:** $\{R^{\texttt{env(c)+exprs}}_{\texttt{env(c)}}\}\ \texttt{send c(exprs)}\ \{R\}$
- **Regla del *receive*:**

$$\frac{Q \wedge (\texttt{env(c)-rec(c)} \neq \varnothing) \wedge \texttt{M=pri(env(c)-rec(c))} \rightarrow R^{\texttt{M,rec(c)+M}}_{\texttt{vars,rec(c)}}}{\{Q\}\ \texttt{receive c(vars)}\ \{R\}}$$

# Ejemplo: programación de filtros

- **Problema**: diseñar un programa distribuído que ordene un conjunto de datos
- Estrategia: mezcla de secuencias ordenadas
  - varios procesos "mezcladores" pasándose información a través de canales



# Ejemplo: programación de filtros

```
Vars entr1,entr2,salida: Canal(Ent)
mezclar::
  Vars  d1,d2: Ent
  receive entr1(d1);  receive entr2(d2)
  Mq d1<>FS ∨ d2<>FS
    Sel
      d1<>FS ∧ d2<>FS: Si d1<=d2 Ent send salida(d1)
                                 receive entr1(d1)
                       Si_No     send salida (d2)
                                 receive entr2(d2)
                       FSi
      d1<>FS ∧ d2=FS:  send salida (d1); receive entr1(d1)
      d1=FS ∧ d2<>FS:  send salida (d2); receive entr2(d2)
    Fsel
  FMq
  send salida(FS)
```

```
--FS: entero indicando
  fin de la secuencia
```

# Ejemplo: programación de filtros

```
with listas;

generic
   type dato is private;
package canalesAsinc is
   package listaDato is new listas(dato);

   protected type canalAsinc is
      procedure send(d: in dato); --no bloqueante
      entry receive(d: out dato);
   private
      losEl: listaDato.lista := listaDato.listaVacia;
                              --contenido del canal
   end canalAsinc;
end canalesAsinc;
```

# Ejemplo: programación de filtros

```
package body canalesAsinc is
   protected body canalAsinc is
      procedure send(d: in dato) is
      begin
         listaDato.anadeDch(losEl,d);
      end;

      entry receive(d: out dato) when listaDato.long(losEl)>0 is
      begin
         d := listaDato.observa(losEl,1);
         listaDato.eliminaIzq(losEl);
      end;
   end canalAsinc;
end canalesAsinc;
```

# Ejemplo: programación de filtros

- Algunos números:
  - **$n=2^k$** datos, **n-1** procesos, **$lg_2 n$** niveles
  - **2(n-1)+1** canales
  - nivel j contiene los procesos **$2^j, 2^j+1, ..., 2^j+2^j-1$**
  - proceso **i**: **2i** y **2i+1** como canales de entrada, i como salida

```
procedure mergeSort is
   package paqCanales is new canalesAsinc(integer); --de enteros
   FDS: constant integer := -1;    --Fin De Secuencia
   n: constant integer := 32;  --numero de datos en la sec.
   nProc: constant integer := n-1; --numero de procesos
   nNiveles: constant integer := Integer(Log(Float(n),2.0));
                              --niveles del arbol red (log_2(n))
   losCanales: array(1..2*(n-1)+1) of paqCanales.canalAsinc;
-------------------------------------------------------------------
   task type filtro(id: integer; cEnt1,cEnt2: integer; cSal: integer);
   type ptFiltro is ACCESS filtro;
...
end mergeSort;
```

# Ejemplo: programación de filtros

```
task body filtro is
  d1,d2: integer;
  ent1: paqCanales.canal renames losCanales(cEnt1);
  ent2: paqCanales.canal renames losCanales(cEnt2);
  sal:  paqCanales.canal renames losCanales(cSal);
begin
  ent1.receive(d1);ent2.receive(d2);
  while d1 /= FDS or d2 /= FDS loop
      if d1 /= FDS and d2 /= FDS then
        ...
      elsif d1 /= FDS and d2 = FDS then
        ...
      else
        ...
end filtro;
```

# Ejemplo: programación de filtros

```
      if d1 /= FDS and d2 /= FDS then
         if d1 <= d2 then
           sal.send(d1);
           if id = 1 then  --es el que da el res. definitivo
             put_line("----------------> " & INTEGER'IMAGE(d1));
           end if;
           ent1.receive(d1);
         else
           sal.send(d2);
           if id = 1 then  --es el que da el res. definitivo
             put_line("----------------> " & INTEGER'IMAGE(d2));
           end if;
           ent2.receive(d2);
         end if;
       elsif
       ...
```

# Ejemplo: programación de filtros

```
  task type inicializadorDeDatos;  --lanza los n datos
  task body inicializadorDeDatos is
  begin
    put("Secuencia de entrada: ");
    for j in reverse 1..n loop
      put(INTEGER'IMAGE(j) & " ");
      losCanales(2**nNiveles+n-j).send(j); --datos al reves
      losCanales(2**nNiveles+n-j).send(FDS); --FDS
    end loop;
    new_line;
  end;

  losFiltros: array(1..n-1) of ptFiltro;
  elIniciador: inicializadorDeDatos;
begin
  for i in 1..n-1 loop
    losFiltros(i) := new filtro(i,2*i,2*i+1,i);
  end loop;
end pruebaCanales;
```

# Ejemplo: programación de filtros

```
import java.util.*;
class CanalAsincrono<E> {

   private Queue<E> colaCanal = new LinkedList<E>();

   public synchronized void send(E d) {
      colaCanal.add(d);
      notifyAll();
   }

   public synchronized E receive() {
      while(colaCanal.isEmpty()) {
         try {
            wait();}
         catch (InterruptedException ignorar) {}
      }
      return colaCanal.remove();
   }
}
```

# Ejemplo: programación de filtros

```
class Filtro implements Runnable {
   public static final int FDS = -1;

   protected int id;
   protected MergeSort.CanalInt[] ent;
   protected MergeSort.CanalInt sal;

   protected Integer[] d = new Integer[2];

   Filtro( int id, MergeSort.CanalInt ent0,
           MergeSort.CanalInt ent1,
           MergeSort.CanalInt sal) {
      this.id  = id;
      this.ent = new MergeSort.CanalInt[]{ent0, ent1,};
      this.sal = sal;

      (new Thread(this)).start();
}
```

```
   protected void enviar(int i) {
      sal.send(d[i]);
      if (id == 0) { //Es el que da el resultado definitivo
         System.out.println("----------------> " + d[i]);
      }
      d[i] = ent[i].receive();
   }
   public void run() {
      d[0] = ent[0].receive();
      d[1] = ent[1].receive();
      while ((d[0] != FDS) || (d[1] != FDS)) {
         if ((d[0] != FDS) && (d[1] != FDS)) {
            if (d[0] <= d[1]) {
               enviar(0);
            }else{
               enviar(1);}
         }else if ((d[0] != FDS) && (d[1] == FDS)) {
            enviar(0);}
          else {enviar(1);}
      }
      sal.send(FDS);
   }
}
```

# Ejemplo: programación de filtros

```
class MergeSort {
   public static final int N = 32;

       //Renombramos por comodidad CanalAsincrono<Integer>
       //y permitimos al compilador y al intérprete trabajar con
       //un tipo no genérico
   public static class CanalInt extends CanalAsincrono<Integer> {}
       //Método principal
   public static void main(String[] args) {
      CanalInt[] canal = new CanalInt[2*N-1];

      for(int i = 0; i < canal.length; i++) {
         canal[i] = new CanalInt();} // inicializador de datos
      new InicializadorDatos(canal);

      for(int i = 0; i < N-1; i++) {
           //i-ésimo Filtro e inicia su ejecución
         new Filtro(i, canal[2*i+1],canal[2*(i+1)],canal[i]);
      }
   }
}
```

```
class InicializadorDatos implements Runnable {

   protected MergeSort.CanalInt[] canal;
   InicializadorDatos(MergeSort.CanalInt[] canal) {
      this.canal = canal;
      (new Thread(this)).start();
   }

   // Escribe la secuencia de números a ordenar en
   // los canales correspondientes

   public void run() {
      System.out.print("Secuencia de entrada:");
      for(int j = MergeSort.N; j > 0; j--) {
         System.out.print(" " + j);

         int idCanal = 2*MergeSort.N - j - 1;
         canal[idCanal].send(j);
         canal[idCanal].send(Filtro.FDS);
      }
      System.out.println();
   }
}
```

# Esquema cliente-servidor

- **Problema**: implementar un patrón cliente-servidor en el que el servidor puede atender peticiones $(op_1, ..., op_M)$
- Cada operación requiere un conjunto específico de argumentos $(at_1, ..., at_M)$
- Cada operación devuelve un conjunto específico de resultados $(rt_1, ..., rt_M)$

# Esquema cliente-servidor

```
Tipos  tipoOper = Enum(op1...opM)
       tipoArg = Union(arg1:at1...argM:atM)
       tipoRes = Union(res1:rt1...resM:rtM)
Vars   peticion: Canal(Ent,tipoOper,tipoArg)
       respuesta: Vector(1..n) De Canal(tipoResp)
Servidor::
  Vars indice:Ent;oper:tipoOper;args:tipoArg;res:tipoRes
   acciones de inicialización
   Mq Verdad
     receive peticion(indice,oper,args)
     Sel
       oper=op1: acciones de op1
       ...
       oper=opM: acciones de opM
     FSel
     send respuesta(indice)(res)
  FMq
```

# Esquema cliente-servidor

```
Cliente(i:1..n)::
  Vars elArg:tipoArg;elRes:tipoRes
  ...
  acciones de definición de elArg
  acciones previas
  send peticion(i,opI,elArg)    --envío petición servicio
  receive respuesta(i)(elRes)   --respuesta del servidor
  acciones posteriores
```

Modificar para que se puedan enviar varias peticiones seguidas

# Ejemplo: servidor de recursos compartidos

- **Problema**: implementar un servidor de recursos que atiende peticiones de sus clientes para adquirir y liberar recursos.
- Todos los recursos son idénticos, aunque cada uno tiene su propio identificador.
- Es preciso que las demandas se atiendan por orden de petición.

# Ejemplo: servidor de recursos compartidos

```
Tipos  tiposOp = Enum (TOMAR,DEJAR)
Vars   peticion: Canal(indice: Ent,tipoOp: tiposOp,
                       idRec: Ent)
       respuesta: Vector(1..n) De Canal(Ent)
Cliente(i:1..n)::
  Vars idRecurso: Ent
  ...
  acciones  previas
  send peticion(i,TOMAR,0)            --petición de recurso
  receive respuesta(i)(idRecurso)  --esperando respuesta
  ...                              --uso del recurso
  send peticion(i,DEJAR,idRecurso) --libera el recurso
  acciones  prosteriores
```

# Ejemplo: servidor de recursos compartidos

```
ServidorDeRecursos::
  Vars disponibles: Ent:=MAX
       recursos: Conjunto De Ent := {...}
       solicitados: Cola De Ent;
       indice,idRecurso: Ent; oper:tiposOp
  acciones de inicialización
  Mq Verdad
    receive peticion(indice,oper,idRecurso)
    Sel oper=TOMAR:                  --petición del recurso
      Si disponibles>0 Ent           --concede petición
        disponibles:=disponibles-1
        idRecurso:=tomar(recursos);  --uno del conjunto
        send respuesta(indice)(idRecurso)
      Si_No                  --no hay disponibles: a esperar
        insertarCola(solicitados,indice)
      Fsi
      ...
```

# Ejemplo: servidor de recursos compartidos

```
...
Mq Verdad
  receive peticion(indice,oper,idRecurso)
  Sel
     oper=TOMAR: ...
     oper=DEJAR:                --liberación de un recurso
        Si estaVacia(solicitados) Ent
           disponibles:=disponibles+1;
           añadir(recursos,idRecurso)
        Si_No
          indice:=extraerCola(solicitados);
          send respuesta(indice)(idRecurso)
        FSi
  FSel
FMq
```

**Mejoras:**
*** que un cliente no pueda "engañar" al servidor**
*** que se pueda pedir un lote de recursos**
*** que haya recursos de distintos tipos, y se pidan como multi-conjuntos**

# Ejemplo: recursos compartidos mediante un monitor

- Hay cierta dualidad entre monitores para controlar recursos y servidores

```
Monitor recursosCompartidos
  Vars disponibles: Ent:=MAX
       recursos: Conjunto De Ent := ...
       hayLibres: Cond

  Accion tomar(S idRecurso:Ent)
    Sel
      disponibles=0: wait(hayLibres)
      disponibles<>0: disponibles := disponibles-1
    FSi
    idRecurso := tomar(recursos);  --uno
  Fin
  ....
FMonitor
```

# Ejemplo: recursos compartidos mediante un monitor

```
Monitor recursosCompartidos
  ...

  Accion dejar(E idRecurso:Ent)
    recursos := recursos ∪ {idRecurso}
    Sel
      empty(hayLibres): disponibles := disponibles+1
      ¬empty(hayLibres): signal(hayLibres)
    FSel
  Fin
FMonitor
```

# Ejemplo: recursos compartidos mediante un monitor

- Similitudes entre monitores y servidores implementados mediante paso de mensajes

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

monitor

```
variables locales servidor
canal de petición y disc. petición
send petición + receive respuesta
receive petición
send respuesta
encolar peticiones pendientes
procesa petición pendiente
Sel...FSel sobre peticiones
```

tarea

```
ServidorDeRecursos::
  ......
    Sel oper=TOMAR:                --petición del recurso
      Si disponibles>0 Ent         --concede petición
        disponibles:=disponibles-1
        idRecurso:=tomar(recursos);  --uno del conjunto
        send respuesta(indice)(idRecurso)
      Si_No                  --no hay disponibles: a esperar
        insertarCola(solicitados,indice)
      Fsi
```

```
PMonitor
  ....
 Accion tomar(S idRecurso:Ent)
    Sel
      disponibles=0: wait(hayLibres)
      disponibles<>0: disponibles := disponibles-1
    FSi
    idRecurso := tomar(recursos);  --uno
  Fin
  ....
FMonitor
```

# n monitor

```
Sel
  oper=TOMAR: ...
  oper=DEJAR:                --liberación de un recurso
    Si estaVacia(solicitados) Ent
      disponibles:=disponibles+1;
      añadir(recursos,idRecurso)
    Si_No
      indice:=extraerCola(solicitados);
      send respuesta(indice)(idRecurso)
    FSi
FSel
```

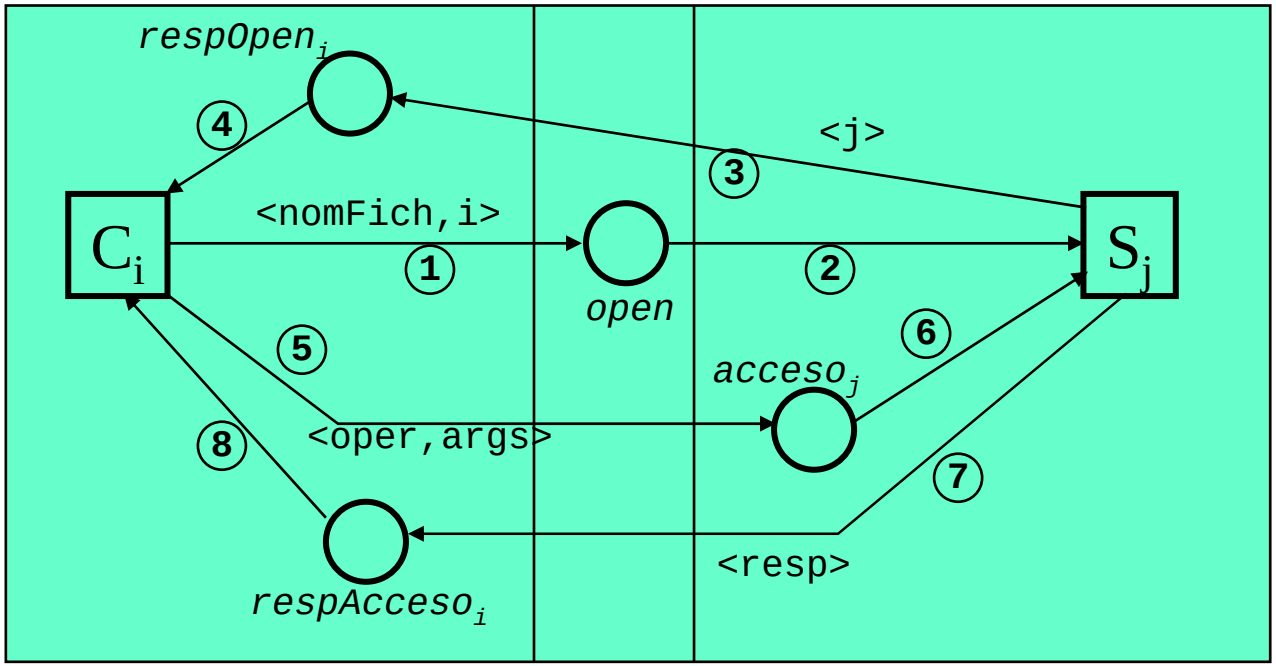
```
Monitor recursosCompartidos
  ...

  Accion dejar(E idRecurso:Ent)
    recursos := recursos ∪{idRecurso}
    Sel
      empty(hayLibres): disponibles := disponibles+1
      ¬empty(hayLibres): signal(hayLibres)
    FSel
  Fin
FMonitor
```

# Ejemplo: servidor de ficheros

- Otro 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

- Arquitectura del sistema:



# Ejemplo: servidor de ficheros

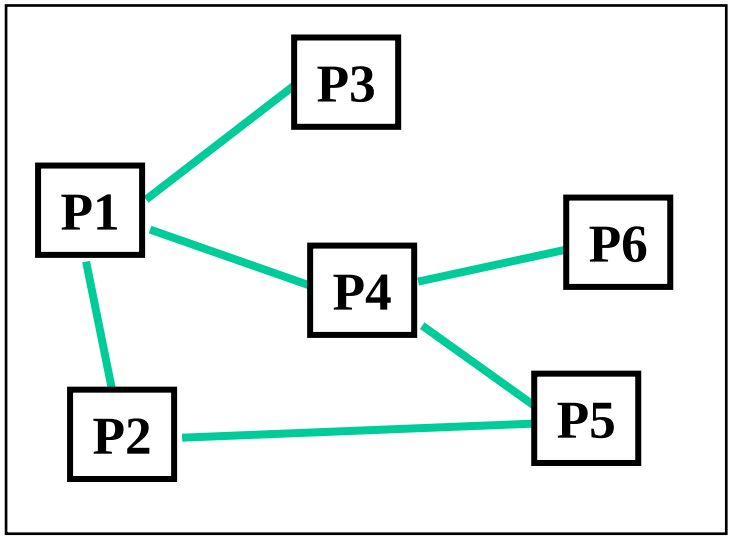
```
Tipos tipoOper = Enum(READ,WRITE,CLOSE)
      open: Canal(nombreF:Cadena,idCliente: Ent)
      acceso: Vector(1..n) De Canal(oper:tipoOp,...)
      respOpen: Vector(1..m) De Canal(Ent)
      respAcceso: Vector(1..m) De Canal(...)
             --datos, cód. de error, etc.
Cliente(i:1..m)::
  Vars idServidor: Ent
       resultados: tipoResult
  send open(nombreFichero,i)   --pide abrir fichero
  receive respOpen(i)(idServidor)  --espera respuesta
  ...                        --opera con el fichero
  Mq meDeLaGana
     send acceso(idServidor)(operación,argumentos)
     receive respAcceso(i)(resultados)
     ...
  FMq
```

# Ejemplo: servidor de ficheros

```
ServidorDeFicheros(j:1..n)::
  Vars nombreF: Cadena; idCliente: Ent
       oper: tipoOper; args: otrosTipos
       seguir: booleano := Falso
         --"buffer" local, direcciones de disco, etc.
  Mq Verdad
    receive open(nombreF,idCliente)
    send respOpen(idCliente)(j)
    seguir:=Verdad
    Mq seguir
      receive acceso(j)(oper,args)
      Sel oper=READ:  leer un dato
          oper=WRITE: escribir un dato
          oper=CLOSE: cerrar fichero; seguir:=Falso
      FSel
      send respAcceso(idCliente)(resultados)
    FMq
  FMq
```

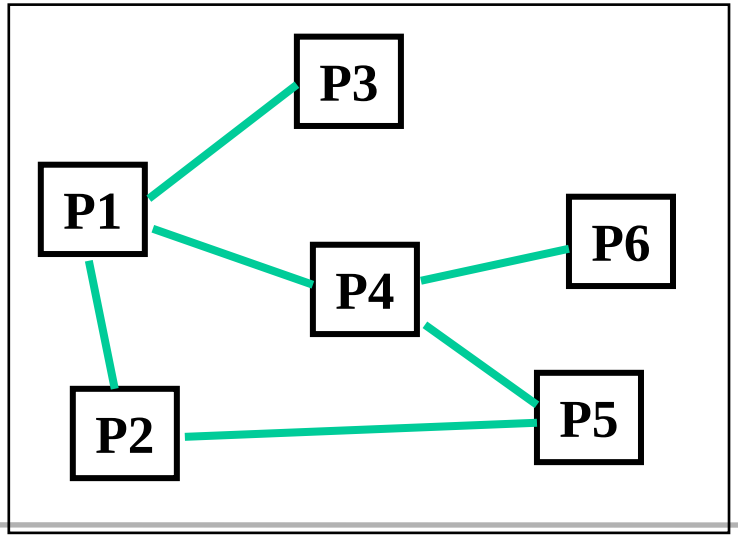
# Ejemplo: sacar la topología de una red

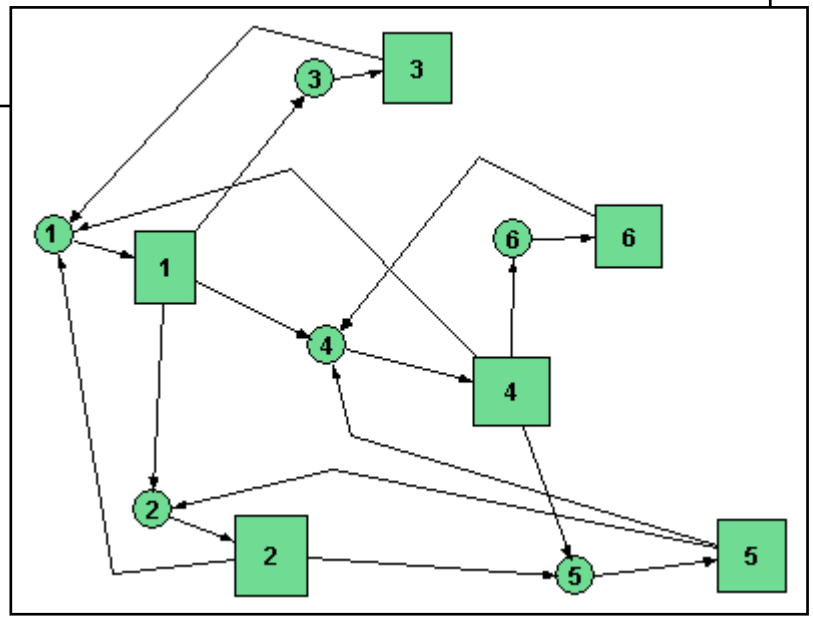
- **Problema**: implementar mediante paso asíncrono de mensajes el código de manera que cada nodo tenga una copia de la topología de la red
- Asumiendo que:
  - enlaces entre nodos son simétricos
  - cada uno sabe quiénes son sus vecinos
  - sabemos que el camino más largo tiene D enlaces



# Ejemplo: sacar la topología de una red

```
Constantes D := lo que sea  --long del camino más largo
Tipos grafo: Vector(1..n,1..n) De Booleano
      vecinos: Vector(1..n) De Booleano
Vars vectCanales: Vector(1..n) De canal(grafo)

Nodo[p:1..n,misVecinos: vecinos]
```



# Ejemplo: sacar la topología de una red

```
Nodo[p:1..n,misVecinos: vecinos]
   Vars miGrafo,otroGrafo: grafo := (others=> false)
        d: Entero
   miGrafo(p,*) := misVecinos(*)  --asumir correcto
   d := 1                  --inv: miGrafo tiene caminos
                           --     de long d desde p
   Mq d<D
      Pt q∈{1..n} t.q. misVecinos(q)
         send vectCanales(q)(miGrafo)
      FPt
      Pt q∈{1..n} t.q. misVecinos(q)
         receive vectCanales(p)(otroGrafo)
         miGrafo := miGrafo OR otroGrafo
      FPt
      d := d+1
   FMq
```

# Ejercicios

- **Ejercicio 1**: Implementar una solución al problema de los lectores/escritores mediante paso asíncrono de mensajes.
- **Ejercicio 2**: Implementar una solución al problema de los filósofos mediante paso asíncrono de mensajes.
- **Ejercicio 3**: Una cuenta de ahorros es compartida por varias personas. Cada persona puede depositar o retirar una cantidad positiva de dinero. El balance de la cuenta nunca puede ser negativo. Desarrollar un servidor que controle el acceso a la cuenta mediante paso asíncrono de mensajes, así como el código de un cliente.

# Ejercicios

- **Ejercicio 4**: Dos tipos de procesos, T1 y T2 pueden entrar y salir de una habitación común. Un proceso T1 no puede salir de la habitación hasta que haya coincido en ella, simultáneamente, con dos T2, mientras que un T2 no puede salir hasta que haya coincido con al menos un T1. Se pide desarrollar el controlador de acceso a la habitación, así como el código de los procesos de tipo T1 y T2. La comunicación de los procesos con el controlador debe hacerse mediante paso asíncrono de mensajes. Asumiendo que simpre hay procesos de ambos tipos queriendo entrar, comentar aspectos como posibilidades de bloqueo, de equidad, etc. en la solución propuesta.

# Ejercicios

- **Ejercicio 5**: 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 paso asíncrono de mensajes como medio de sincronización, 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.

# Ejercicios

- **Ejercicio 6**: El problema de la montaña rusa. Considerar un sistema con P posibles pasajeros y una montaña rusa, que puede llevar nP pasajeros (nP<P). Los pasajeros, viciosos ellos, están siempre esperando para montar en la montaña rusa y, cuando acaban, vuelven a querer montar. Por cicatería de los dueños, la montaña sólo se pone en marcha cuando tiene todas las plazas ocupadas. Desarrollar el código de un proceso pasajero y del controlador de la montaña. Usar paso asíncrono de mensajes como medio de sincronización.