Programación Concurrente 2016

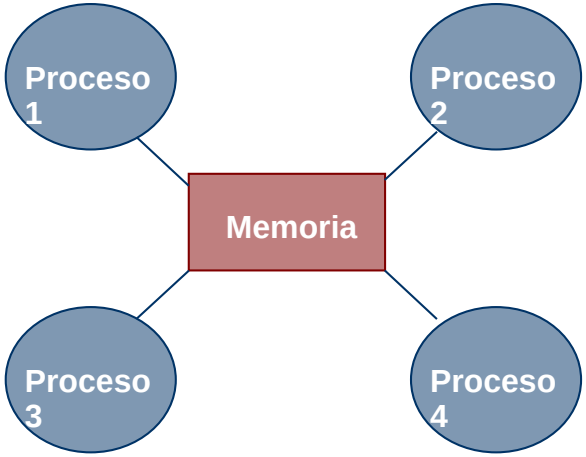
Clase 7

Facultad de Informática
UNLP

Programación Distribuida. Motivación

Sincronización por lectura y escritura de *variables compartidas* ⇒ programas concurrentes ejecutados sobre hardware con ***acceso a memoria compartida.***

Arq. de memoria distribuida ⇒
procesadores + memo local + red de comunicaciones **+ *mecanismo de comunicación / sincronización*** ⇒ ***intercambio de mensajes***



Programación Distribuida. Conceptos básicos

Programa distribuido ⇒ **programa concurrente comunicado por mensajes**
Supone la ejecución sobre una arq. de memoria distribuida, aunque puedan ejecutarse sobre una de memoria compartida (o híbrida).

Primitivas de pasaje de mensajes**: interfaz con el sistema de comunicaciones**

⇒ **semáforos + datos + sincronización**

⇒ ***Los procesos comparten canales*** (físicos o lógicos).

Qué abstraen?

Programación Distribuida

Los canales **son lo único** que comparten los procesos

⇒ Variables locales a un proceso (“cuidador”)
⇒ La EM no requiere mecanismo especial
⇒ Los procesos interactúan comunicándose
⇒ Accedidos por primitivas de envío y recepción

Variantes para los canales

- Mailbox, input port, link
- Uni o bidireccionales
- Sincrónicos o asincrónicos

Programación Distribuida

Combinaciones $\Rightarrow$ mecanismos “equivalentes” funcionalmente:

- PMA
- PMS
- RPC
- Rendezvous

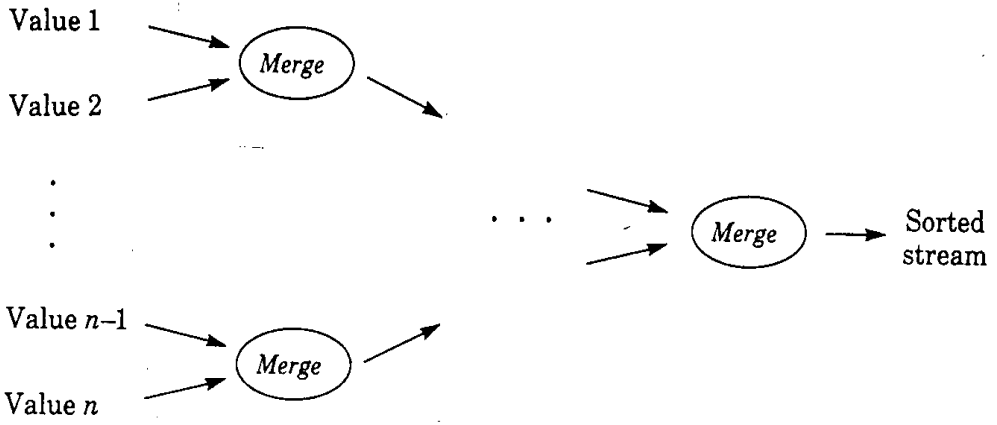
La sincronización de la comunicación interproceso depende del patrón de interacción:

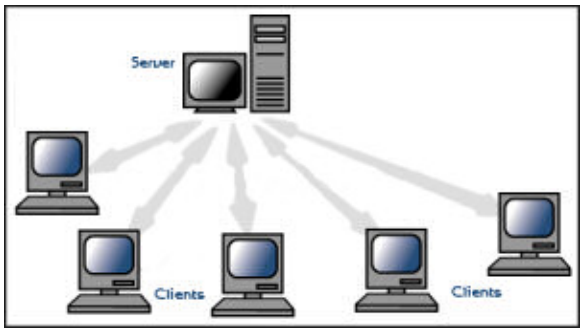
- productores y consumidores
- clientes y servidores
- peers

Cada mecanismo es más adecuado para determinados patrones

Clases básicas de procesos. Algoritmos clásicos.

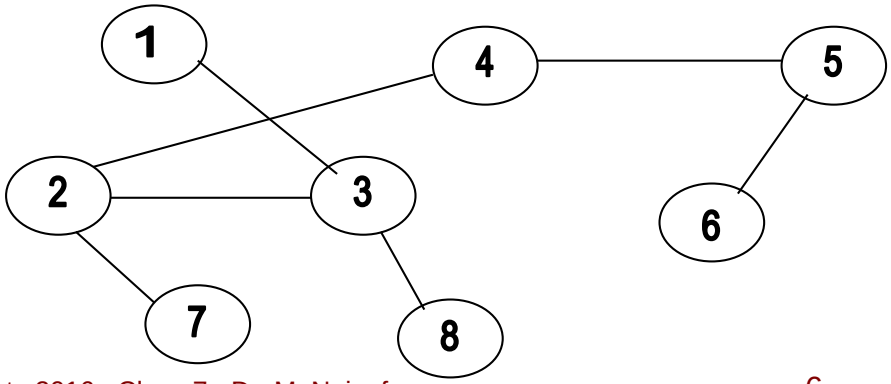
Productores/consumidores (Filtros, Pipes).
Ejemplo: sorting network





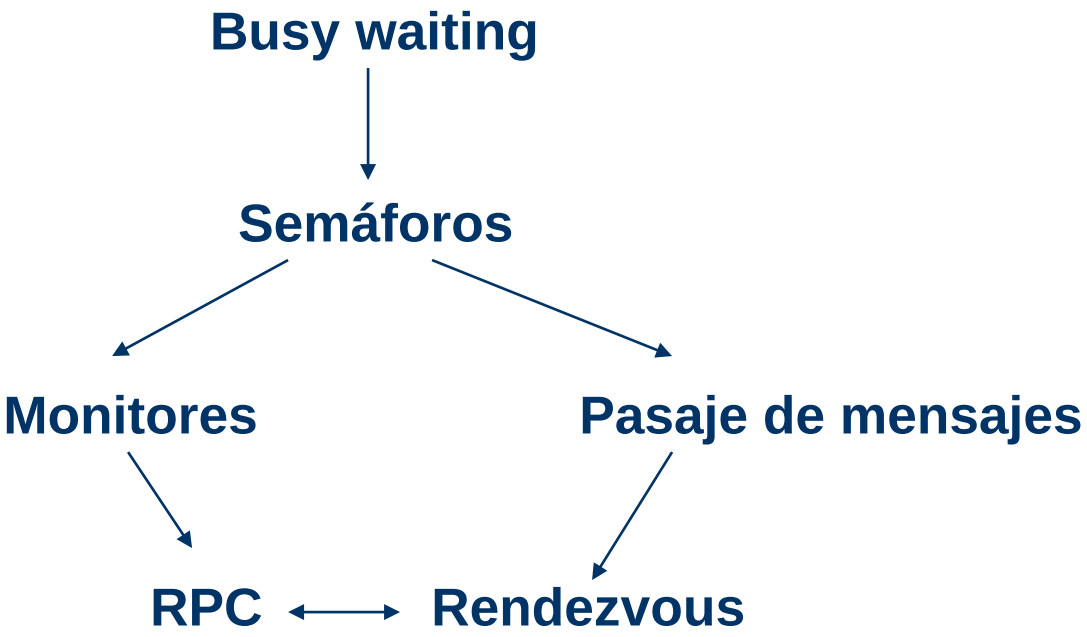
Clientes y Servidores.
Ejemplos: alocación de recursos, file servers, scheduling

Peers.
Ejemplos: probe/echo, heartbeat, semáforos distribuidos, servers replicados



Relación entre mecanismos de sincronización

Semáforos ⇒ mejora respecto de busy waiting;
Monitores ⇒ combinan EM implícita y señalización explícita
PM ⇒ extiende semáforos con datos;
RPC y rendezvous ⇒ combinan la interfase procedural de monitores con PM implícito.



Paradigmas para la interacción entre procesos

3 esquemas básicos de interacción e/ procesos: productor / consumidor, cliente / servidor e interacción entre pares.

Se pueden combinar de muchas maneras, dando lugar a **paradigmas** o modelos de interacción entre procesos.

Paradigma 1: servidores replicados
Los servidores manejan (mediante múltiples instancias) recursos compartidos tales como dispositivos o archivos.

Paradigma 2: algoritmos heartbeat
Los procesos periódicamente deben intercambiar información con mecanismos tipo send/receive.

Paradigma 3: algoritmos pipeline
La información recorre una serie de procesos utilizando alguna forma de receive/send.

Paradigmas para la interacción entre procesos

Paradigma 4: probes (send) y echoes (receive)

La interacción entre los procesos permite recorrer grafos o árboles (o estructuras dinámicas) diseminando y juntando información.

Paradigma 5: algoritmos broadcast

Permiten alcanzar una información global en una arquitectura distribuida. Sirven para toma de decisiones descentralizadas.

Paradigma 6: token passing

En muchos casos la arquitectura distribuida recibe una información global a través del viaje de tokens de control o datos.
También permite la toma de decisiones distribuidas.

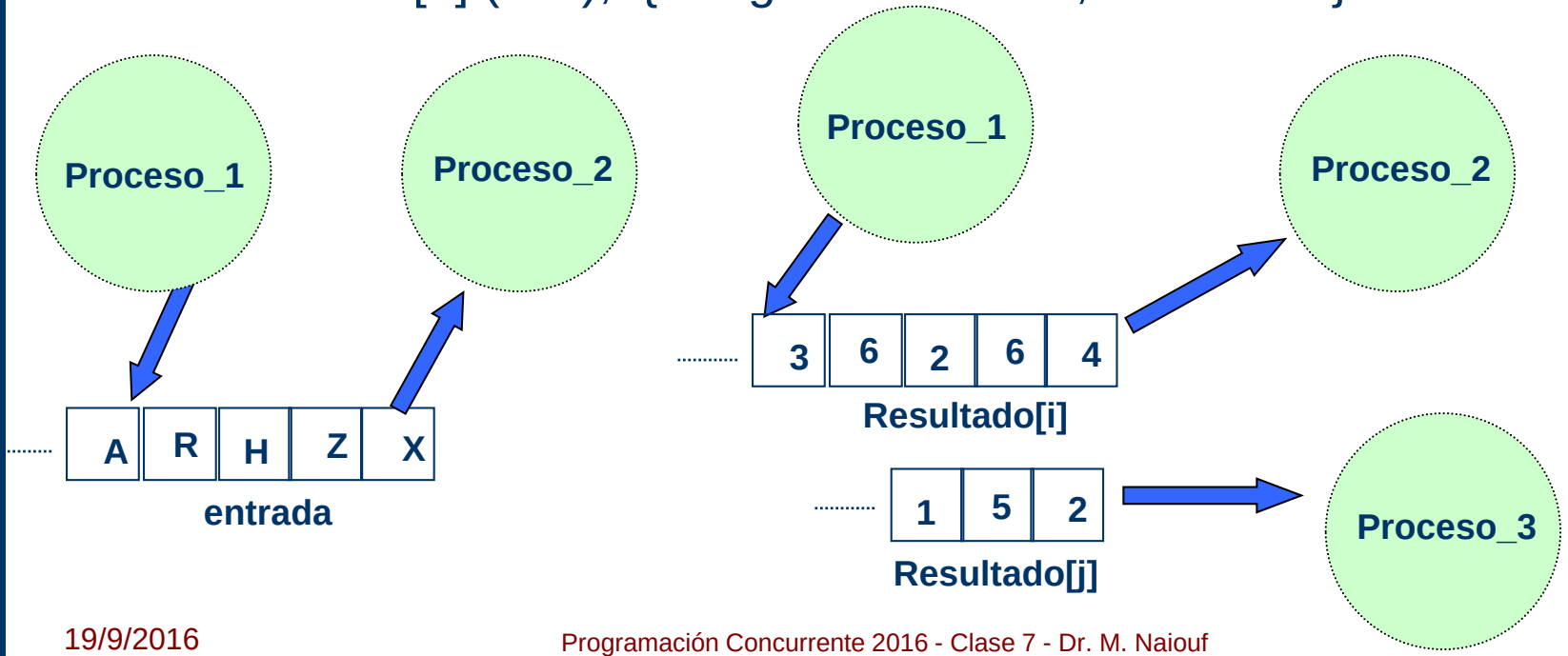
Paradigma 7: manager/workers

Implementación distribuida del modelo de *bag of tasks*.

Pasaje de Mensajes Asincrónicos. Canales

PMA ⇒ canales = colas de mensajes enviados y aún no recibidos:
chan *ch*(*id*$_1$: *tipo*$_1$, ... , *id*$_n$: *tipo*$_n$)

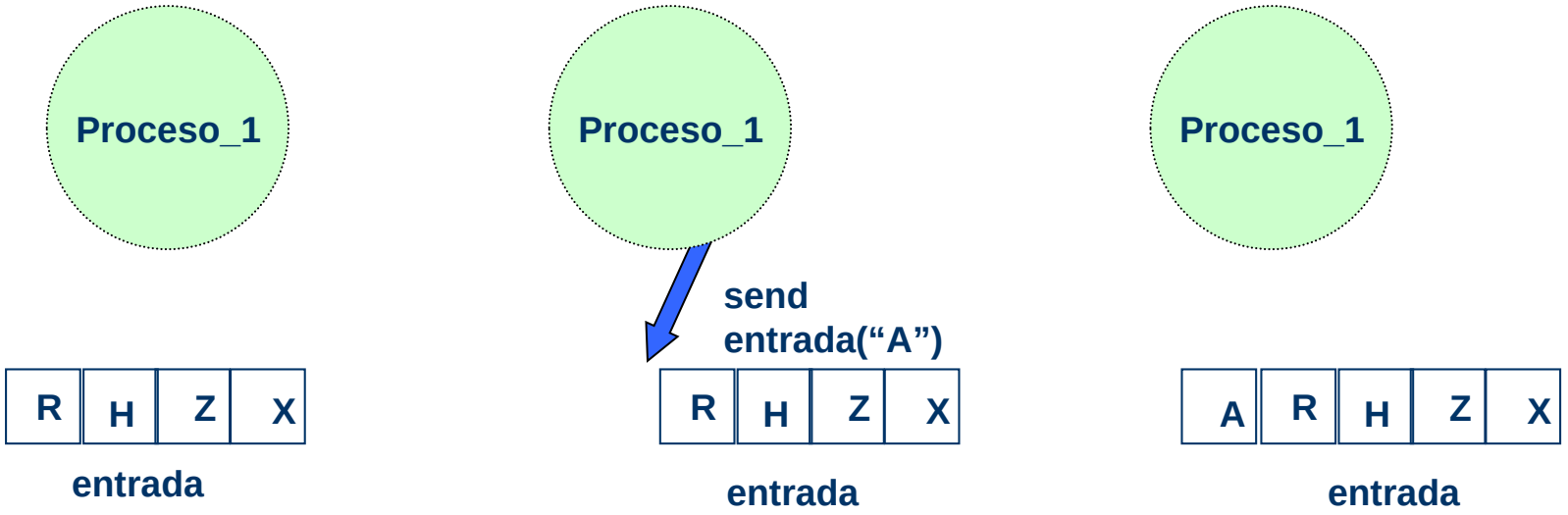
chan entrada(char);
chan acceso_disco(INT cilindro, INT bloque, INT cant, CHAR* buffer)**;**
chan resultado[n] (INT); {arreglo de canales, de 0 a n-1}



Pasaje de Mensajes Asincrónicos. Operación Send

Un proceso agrega un mensaje al final de la cola (“ilimitada”) de un canal ejecutando un **send**, que no bloquea al emisor

send *ch*(*expr*$_1$, ... , *expr*$_n$);

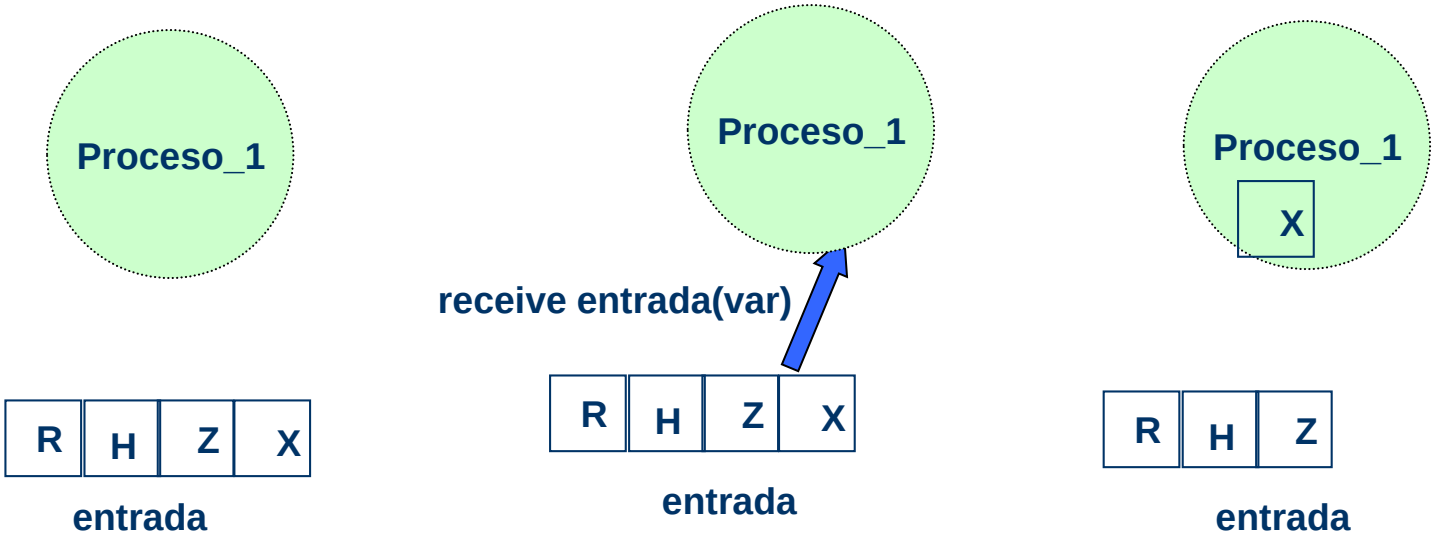


Pasaje de Mensajes Asincrónicos. Operación Receive

Un proceso recibe un msg desde un canal con **receive**, que demora al receptor hasta que en el canal haya al menos un msg; luego toma el primero y lo almacena en variables locales

receive ch(var_1, ... , var_n);

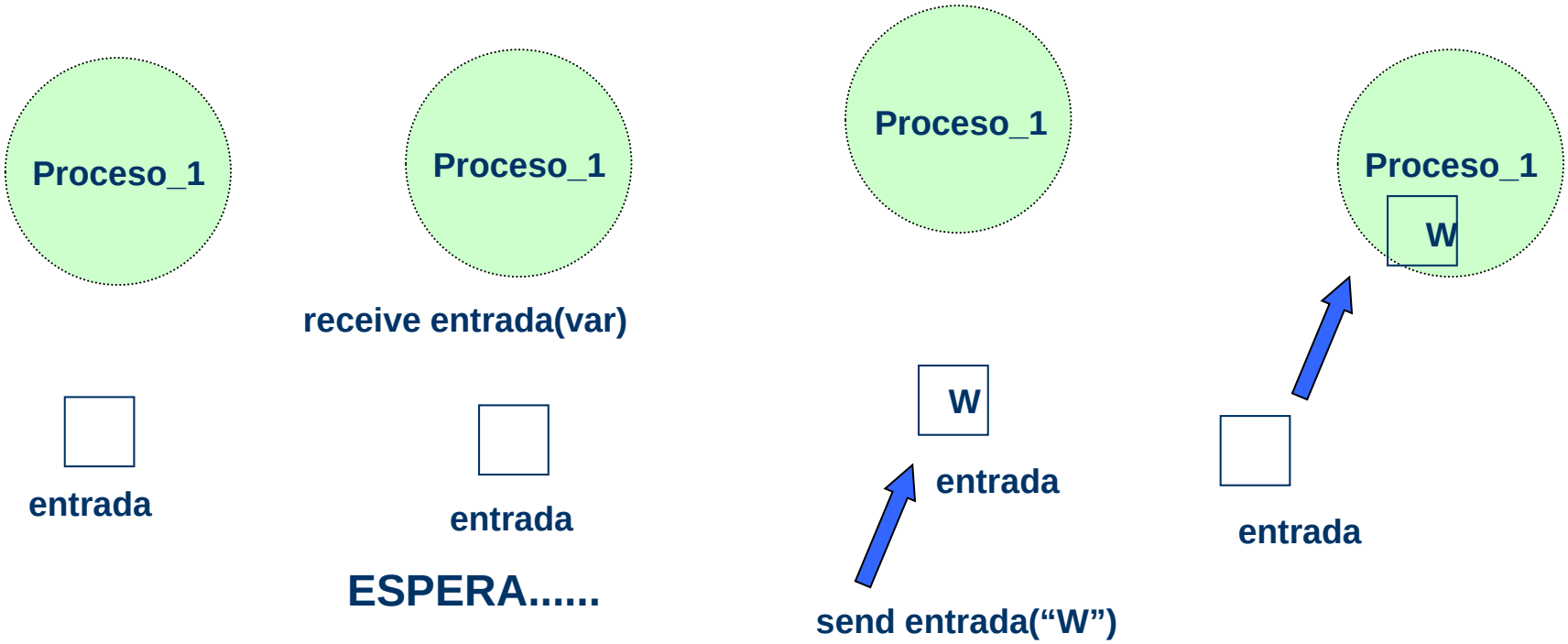
Las variables del receive deben tener los mismos tipos que la declaración del canal ch.



Pasaje de Mensajes Asincrónicos. Operación Receive

Receive es una primitiva bloqueante, ya que produce un delay. Semántica: el proceso NO hace nada hasta recibir un msg en la cola correspondiente al canal ch; NO es necesario hacer polling

⇒ **Canal ≡ semáforo + datos**



Pasaje de Mensajes Asincrónicos

Acceso a los contenidos de c/ canal: atómico y respeta orden FIFO. En principio los canales son ilimitados, aunque las implementaciones reales tendrán un tamaño de buffer asignado.

Se supone que los mensajes NO se pierden ni modifican y que todo mensaje enviado en algún momento puede ser “leído”.

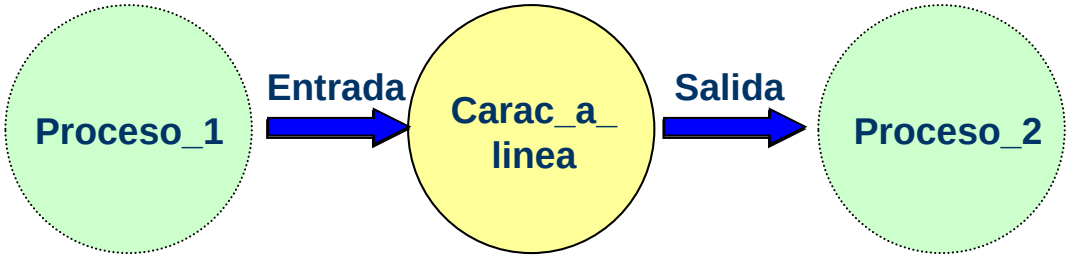
empty(*ch*) determina si la cola de un canal está vacía

Util cuando el proceso puede hacer trabajo productivo mientras espera un mensaje, **pero debe usarse con cuidado.**

La evaluación de empty podría ser true, y sin embargo existir un mensaje al momento de que el proceso reanuda la ejecución.

O podría ser false, y no haber más mensajes cuando sigue ejecutando (si no es el único en recibir por ese canal)

Pasaje de Mensajes Asincrónicos. Ejemplo de proceso *filtro*.




```
chan entrada(char), salida(char [CantMax]);   # Canales globales
Process Carac_a_Linea { char linea [CantMax], int i := 0;
        WHILE true {
                receive entrada (linea[i]);
                WHILE linea[i] ≠ CR and i < CantMax {
                        i := i + 1;
                        receive entrada (linea[i]);  }
                linea [i] := EOL;
                send salida(linea);
                i := 0 }
    }
```

Pasaje de Mensajes Asincrónicos. Ejemplo.

En el ejemplo anterior los canales *entrada* y *salida* son declarados ***globales*** a los procesos, ya que pueden ser compartidos.

Cualquier proceso puede enviar o recibir por alguno de los canales declarados. En este caso suelen denominarse **mailboxes**.

En algunos casos un canal tiene un solo receptor y muchos emisores (***input port***)***.***

Si el canal tiene un único emisor y un único receptor se lo denomina ***link*****:** provee un “camino” entre el emisor y sus receptores.

Pasaje de Mensajes Asincrónicos. Filtros: Red de Ordenación

Filtro: proceso que recibe mensajes de uno o más canales de E/ y envía mensajes a uno o más canales de S/. La salida de un filtro es función de su estado inicial y de los valores recibidos.

Esta función del filtro puede especificarse por un predicado que relacione los valores de los mensajes de S/ con los de E/.

Problema: ordenar una lista de *N* números de modo ascendente
Podemos pensar en un filtro *Sort* con un canal de E/ (*N* números desordenados) y un canal de S/ (*N* números ordenados).

Process *Sort* {
** *receive* todos los números del canal *entrada*;**
** ordenar los números;**
** *send* de los números ordenados por el canal *OUTPUT*;**
** }**

Pasaje de Mensajes Asincrónicos. Filtros: Red de Ordenación

Problema: cómo determina *Sort* que recibió todos los números??
- conoce *N*
- envía *N* como el primer elemento a recibir por el canal *entrada*
- cierra la lista de *N* números con un valor especial o "centinela".

Solución más eficiente que la "secuencial" ⇒ *red de pequeños procesos* que ejecutan en paralelo e interactúan para "armar" la salida ordenada (*merge network*).

Idea: mezclar repetidamente y en paralelo dos listas ordenadas de *N1* elementos cada una en una lista ordenada de *2*N1* elementos.

Con PMA, pensamos en 2 canales de E/ por cada canal de S/.
Un carácter especial EOS cerrará cada lista parcial ordenada.

Pasaje de Mensajes Asincrónicos. Filtros: Red de Ordenación

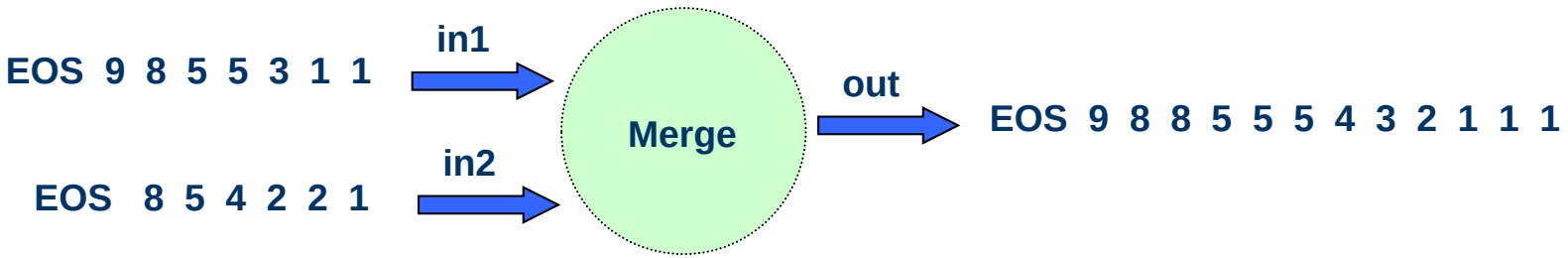
La red es construida con filtros *Merge*

C/ *Merge* recibe valores de dos streams de E/ ordenados, *in1* e *in2*, y produce un stream de salida ordenado, *out.*

Los streams terminan en EOS, y *Merge* agrega EOS al final

Cómo implemento *Merge ?*

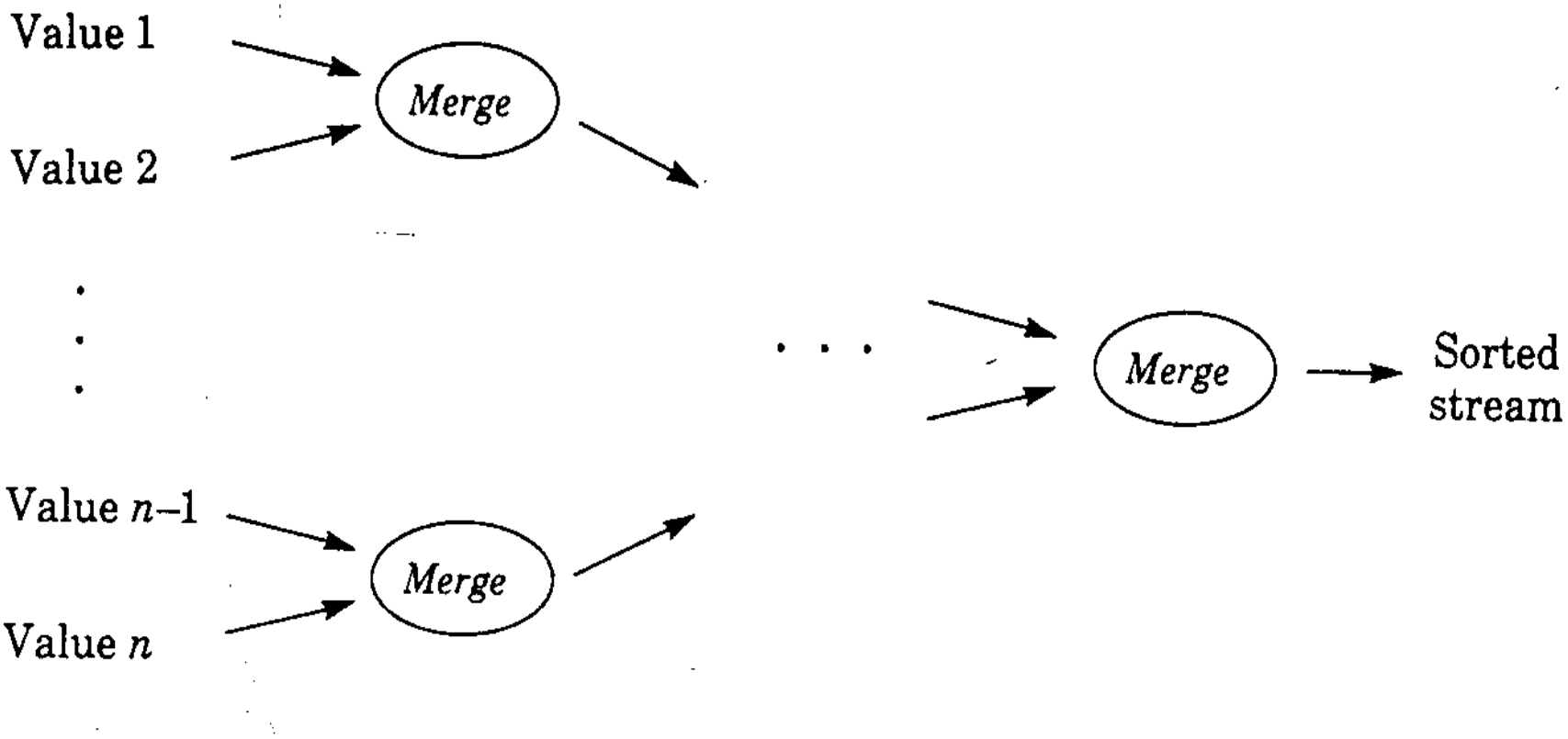
Comparar repetidamente los próximos dos valores recibidos desde *in1* e *in2* y enviar el menor a *out*



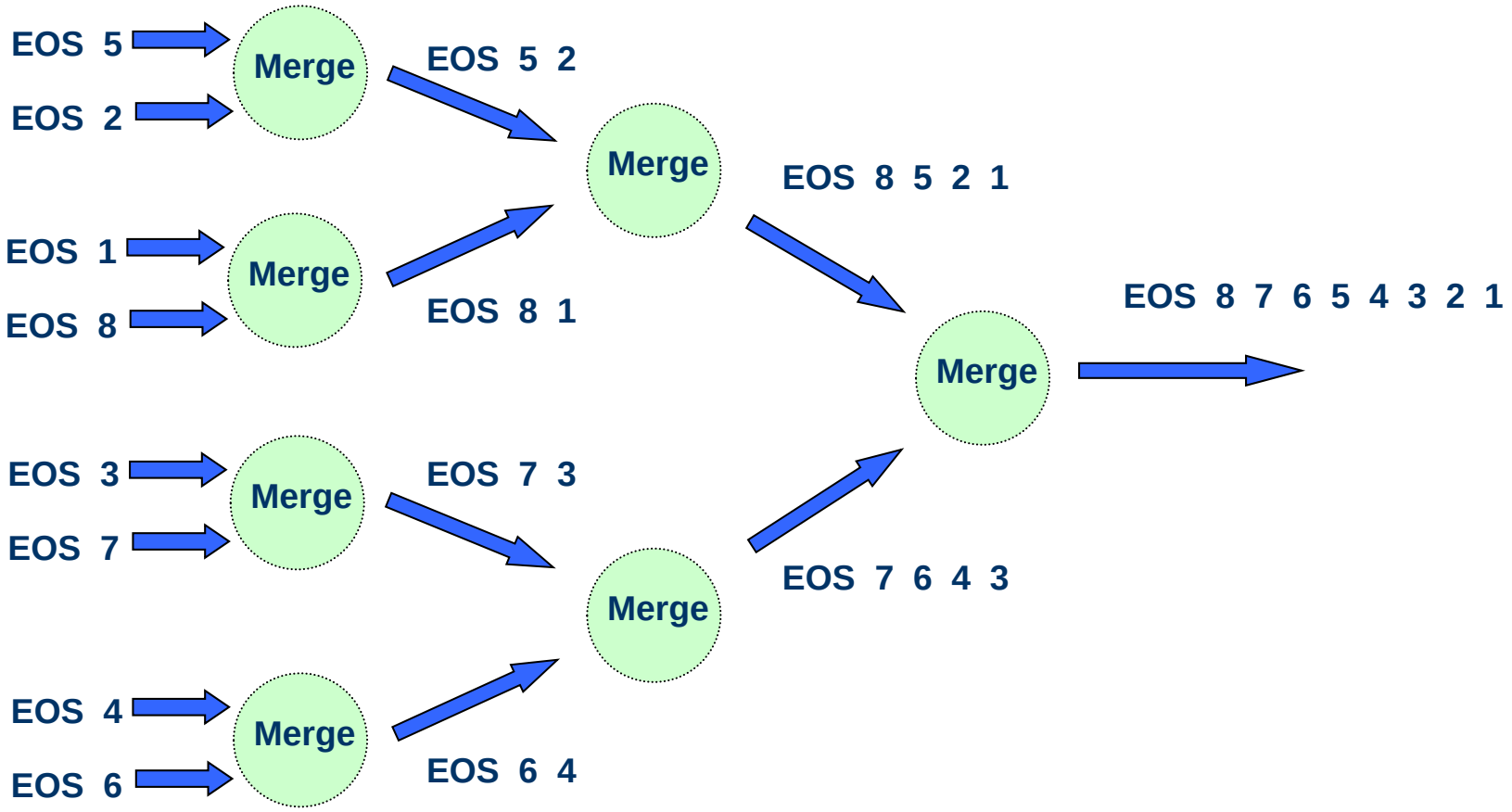
Pasaje de Mensajes Asincrónicos. Filtros: Red de Ordenación

```
chan in1(int), in2(int), out(int);

Process Merge {
        INT v1, v2;
        receive in1(v1); receive in2(v2);
        WHILE (v1 ≠ EOS) and (v2 ≠ EOS) {
          IF v1 ≤ v2 {send out(v1); receive in1(v1); }
                ELSE {send out(v2); receive in2(v2); }   #(v2 < v1)
          }

        # Consumir el resto de los datos
        IF (v1 == EOS)
                WHILE (v2 ≠ EOS) {send out(v2); receive in2(v2);}
          ELSE          # (v2 == EOS)
                WHILE (v1 ≠ EOS) {send out(v1); receive in1(v1);}

        #Agregar el centinela al final
        send out (EOS) ;
   }
```

Pasaje de Mensajes Asincrónicos. Filtros: Red de Ordenación



Pasaje de Mensajes Asincrónicos. Filtros: Red de Ordenación



Pasaje de Mensajes Asincrónicos. Filtros: Red de Ordenación

n-1 procesos; el ancho de la red es $\log_2 n$.

Canales de E/ y S/ compartidos

Puede programarse usando:
- ***static naming*** (arreglo global de canales, y c/ instancia de Merge recibe desde 2 elementos del arreglo y envía a otro $\Rightarrow$ embeber el árbol en un arreglo)

- ***dynamic naming*** (canales globales, parametrizar los procesos, y darle a c/ proceso 3 canales al crearlo; todos los Merge son idénticos, pero se necesita un coordinador)

Los filtros podemos conectarlos de distintas maneras. Solo se necesita que la S/ de uno cumpla las suposiciones de E/ del otro

$\Rightarrow$ pueden reemplazarse si se mantienen los comportamientos de E/ y S/

PMA. Clientes y Servidores. Ejemplo: Monitores Activos

Servidor: proceso que maneja pedidos (“requests”) de otros procesos **clientes**. Cómo implementamos C-S con AMP?

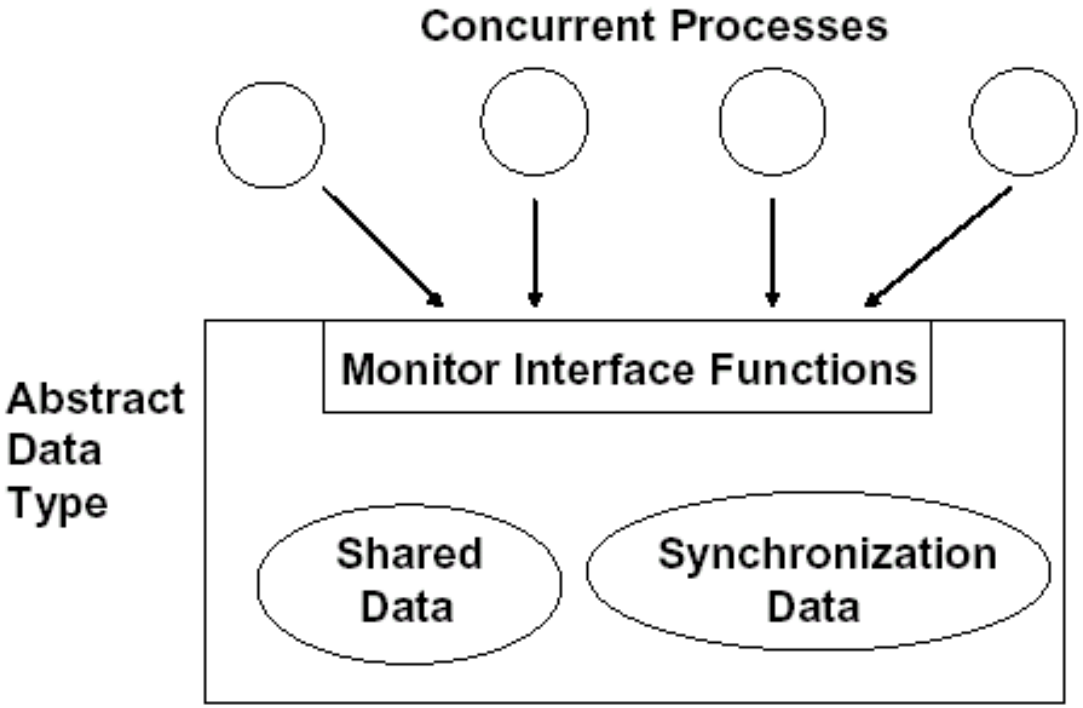
Analizamos cómo convertir monitores en servidores y cómo implementar manejadores de recursos compartidos.

Dualidad entre monitores y PM: c/u de ellos puede simular al otro.

Monitor $\Rightarrow$ manejador de recurso. Encapsula variables permanentes que registran el estado, y provee un conjunto de procedures

Los simularemos usando procesos servidores y MP, como procesos activos en lugar de como conjuntos pasivos de procedures

Recordar: Monitores



Recordar: Monitores

EM **⇒ implícita asegurando que los procedures en el mismo monitor no ejecutan concurrentemente**

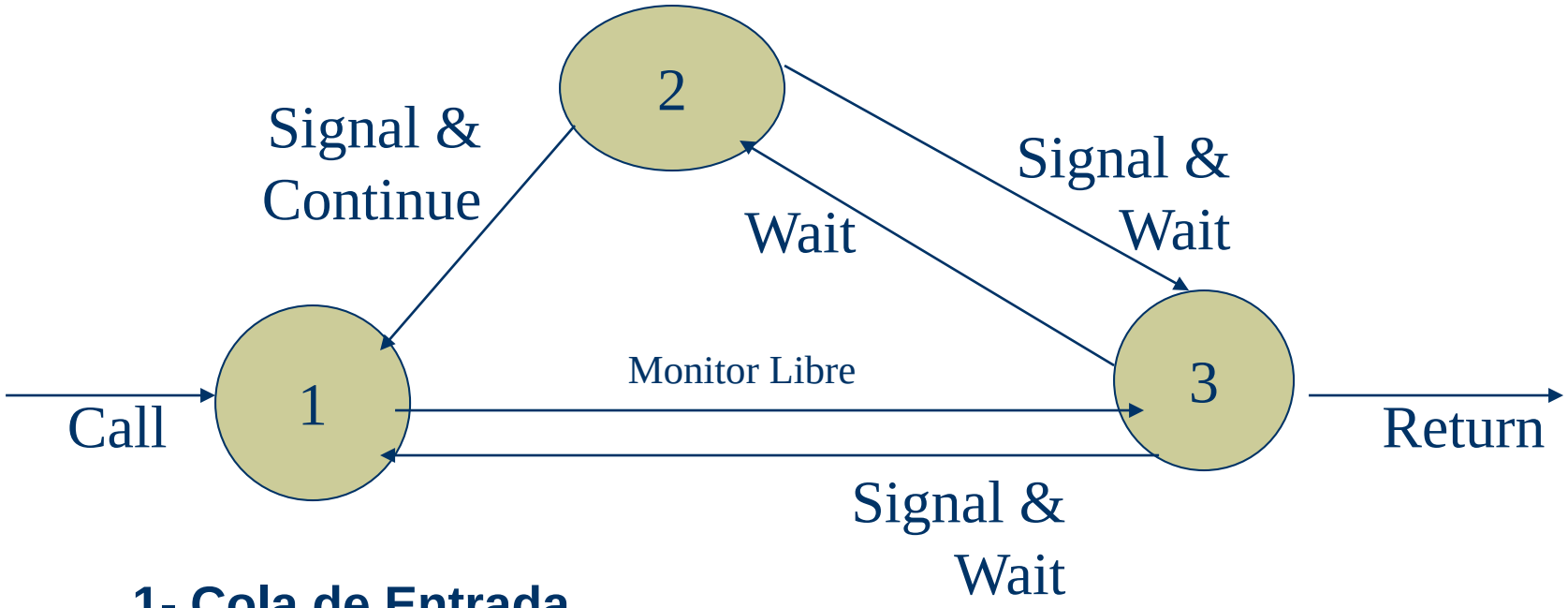
SxC **⇒ explícita con variables condición**

Programa Concurrente ⇒ procesos activos y monitores pasivos
Dos procesos interactúan invocando procedures de un monitor.

```
monitor NombreMonitor {
  declaraciones de variables permanentes;
  código de inicialización
  procedure op1 (par. formales1) {
        cuerpo de op1  }
  .....
  procedure opn (par. formalesn) {
        cuerpo de opn  }
  }
        call NombreMonitor.opi (argumentos)
```

Recordar: Monitores. Sincronización

Signal and continue vs. Signal and Wait



1- Cola de Entrada

2- Cola por Variable Condición

3- Ejecutando en el Monitor

PMA. Clientes y Servidores. Ejemplo: Monitores Activos

ENTONCES…

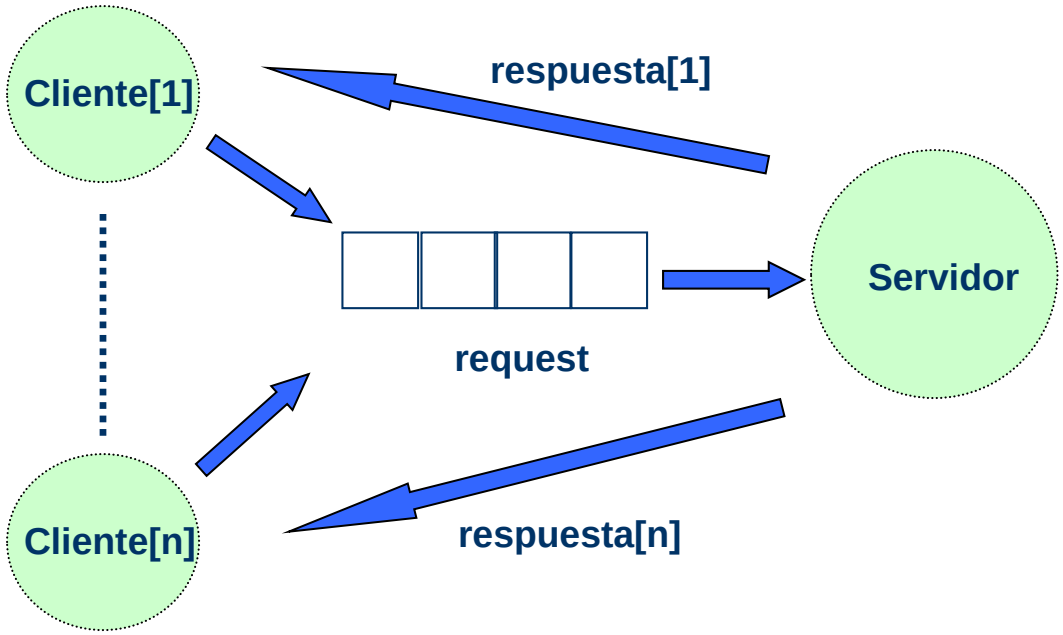
- Un Servidor es un proceso que maneja pedidos (“requests”) de otros procesos clientes. Veremos cómo implementar C-S con AMP.

- Un proceso Cliente que envía un mensaje a un canal de request general, luego recibe el resultado desde un canal de reply propio (por qué?)

- En un sistema distribuido, lo natural es que el proceso Servidor resida en un procesador físico y M procesos cliente residan en otros N procesadores (N<= M)

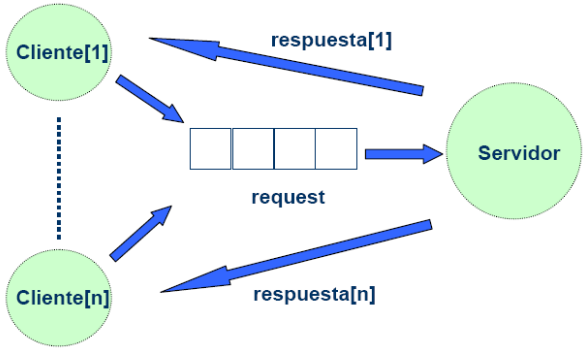
PMA. Clientes y Servidores. Monitores Activos – 1 operación

Para simular *Mname*, usamos un proceso server *Servidor.*
Las variables permanentes serán variables locales de *Servidor.*
Llamado: Un proceso *cliente* envía un mensaje a un canal de *request*
Luego recibe el resultado por un canal de *respuesta* propio



PMA. Clientes y Servidores. Monitores Activos – 1 operación

chan request (INT IdCliente, tipos de los valores de entrada);
chan respuesta[n] (tipos de los resultados);




```
Process Servidor {
        INT IdCliente;
        declaración de variables permanentes;
        código de inicialización;
        WHILE (true) {   receive request (IdCliente, valores de entrada);
                         cuerpo de la operación op;
                         send respuesta[IdCliente](resultados);     }

Process Cliente [i = 1 to n] {
        send request(i, argumentos)        # "llama" a op
        receive respuesta[i] (resultados)  # espera la respuesta     }
```

PMA. Clientes y Servidores. Monitores Activos – Múltiples operaciones

Podemos generalizar esta solución de C-S con una única operación para considerar múltiples operaciones.

El IF del *Servidor* será un CASE a ≠ clases de operaciones.

El cuerpo de c/ operación toma datos de un canal de E/ en args y los devuelve *al cliente adecuado* en resultados.

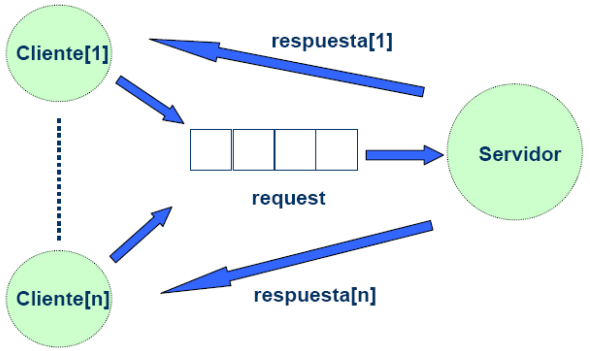
type clase_op = enum(op_1, ..., op_n);
type tipo_arg = union(arg_1 : $tipoAr_1$, ..., arg_n : $tipoAr_n$);
type tipo_result = union(res_1 : $tipoRe_1$, ..., res_n : $tipoRe_n$);

chan request(INT IdCliente, clase_op, tipo_arg);
chan respuesta[n](tipo_result);

Process Servidor .....
Process Cliente [i = 1 to n] .....

PMA. Clientes y Servidores. Monitores Activos – Múltiples operaciones

```
Process Servidor {
        INT IdCliente; clase_op oper; tipo_arg args;
        tipo_result resultados;      #otras declaraciones de variables
        código de inicialización;
        WHILE ( true) {
            receive request(IdCliente, oper, args);
            IF ( oper == op1 ) { cuerpo de op1;}
            .....
            ELSE IF ( oper == opn ) { cuerpo de opn;}
            send respuesta[IdCliente](resultados);
            }
        }

Process Cliente [i = 1 to n] {
        tipo_arg mis_args;
        tipo_result  mis_resultados;
        # poner los valores de los argumentos en mis_args;
        send request(i, opk, mis_args);                  # "llama" a opk
        receive respuesta[i] (mis_resultados);           # espera la respuesta
        }
```



PMA. Clientes y Servidores. Monitores Activos – Múltiples operaciones y variables condición

Caso general: monitor con múltiples operaciones y con ***SxC***.

Hasta ahora el Monitor no requería variables condición ⇒ ***Servidor no requería demorar la atención de un pedido de servicio.***

Para los clientes, la situación es transparente ⇒ ***cambia el servidor.***

Consideramos un caso específico de manejo de múltiples unidades de un recurso (ej: bloques de memoria, impresoras).

Los clientes “adquieren” y devuelven unidades del recurso.
Las unidades libres se insertan en un “conjunto” sobre el que se harán las operaciones de INSERTAR y REMOVER.

El número de unidades disponibles es lo que “controla” nuestra variable de sincronización por condición.

PMA. Clientes y Servidores. Monitores Activos – Múltiples operaciones y variables condición

```
Monitor Alocador_Recurso {
    INT disponible = MAXUNIDADES;
    SET unidades = valores iniciales;
    COND libre;    # TRUE cuando hay recursos
    procedure adquirir( INT  Id )  {
            if (disponible == 0)
                    wait(libre)
            else
                    disponible = disponible - 1;
            remove(unidades, id);
     }
    procedure liberar( INT id ) {
            insert(unidades, id);
            if (empty(libre))
                    disponible := disponible + 1
            else
                    signal(libre);
     }
 }
```

PMA. Clientes y Servidores. Monitores Activos – Múltiples operaciones y variables condición

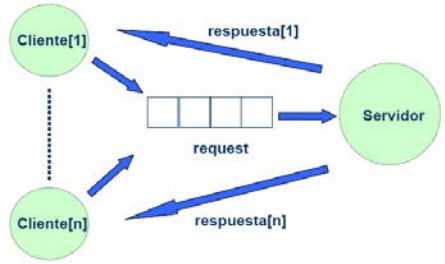
Dos operaciones.

Diferencia: si no hay unidades disponibles, el servidor no puede esperar cuando sirve un pedido: debe salvarlo y diferir la respuesta

Cuando una unidad es liberada, atiende un pedido salvado, si hay, enviando la unidad

```
type clase_op = enum(adquirir, liberar);
chan request(INT IdCliente, clase_op oper, INT id_unidad );
chan respuesta[n] (INT id_unidad);

Process Alocador {
    INT disponible = MAXUNIDADES;
    SET unidades = valor inicial disponible;
    QUEUE pendientes;    # Inicialmente vacía
    # declaración de otras variables;
            ..... sigue
```

PMA. Clientes y Servidores. Monitores Activos – Múltiples operaciones y variables condición




```
WHILE (true) {
        receive request(IdCliente, oper, id_unidad);
        IF (oper == adquirir) {
            IF (disponible > 0) {      # puede atender el  pedido ahora
                  disponible = disponible - 1;
                  remove(unitdades, id_unidad);
                  send respuesta[IdCliente](id_unidad);          }
             ELSE  # recordar el pedido
                  insert(pendientes, IdCliente);
           }

           ELSE {  # significa que el pedido es un liberar
              IF empty(pendientes)   { #devuelve id_unidad a unidades
                  disponible= disponible + 1; insert(unidades, id_unidad);  }
                ELSE  {   # darle id_unidad a un cliente esperando
                  remove(pendientes, IdCliente);
                  send respuesta[IdCliente](id_unidad);          }
               }
        }
}
```

PMA. Clientes y Servidores. Monitores Activos – Múltiples operaciones y variables condición

```
Process Cliente[i = 1 to n]  {
        INT id_unidad;
        send request(i, adquirir, 0)    # "llama" a request
        receive respuesta[i](id_unidad);
        # usa el recurso id_unidad, y luego lo libera;
        send request(i, liberar, id_unidad);
        .....       }
```

El monitor y el Servidor muestran la dualidad entre monitores y MP: hay una correspondencia directa entre los mecanismos de ambos.
La eficiencia de monitores o PM depende de la arq. física de soporte.

Con MC conviene la invocación a procedimientos y la operación sobre variables condición.

Con arquitecturas físicamente distribuidas tienden a ser más eficientes los mecanismos de PM.

Dualidad entre Monitores y Pasaje de Mensajes

Programas con Monitores	***Programas basados en PM***
Variables permanentes	Variables locales del servidor
Identificadores de procedures	Canal *request* y tipos de operación
Llamado a procedure	*send request*( )*; receive respuesta*
Entry del monitor	*receive request*( )
Retorno del procedure	*send respuesta*( )
Sentencia *wait*	Salvar pedido pendiente
Sentencia *signal*	Recuperar/ procesar pedido pendiente
Cuerpos de los procedure	Sentencias del “case” de acuerdo a la clase de operación.

PMA. Clientes y Servidores. File Servers / Continuidad Conversacional

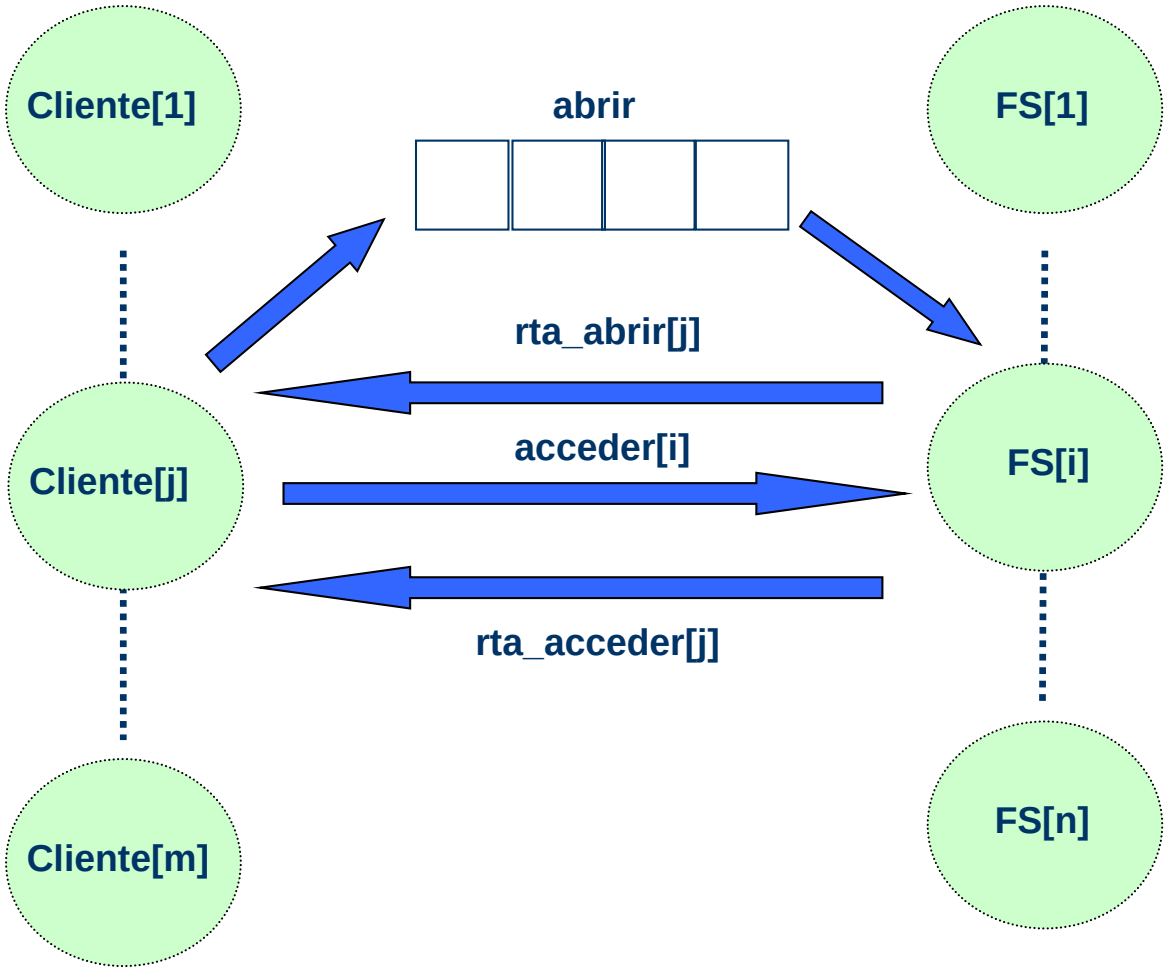
Procesos “cliente” que acceden a archivos externos almacenados en disco.

Deben hacer OPEN; si el archivo se puede abrir hacen una serie de pedidos de READ o WRITE y luego cierran el archivo (CLOSE)

Si hay N archivos, consideramos 1 File Server por archivo.
Los procesos servidores son idénticos, y cualquiera de ellos que esté libre puede atender un requerimiento de OPEN.

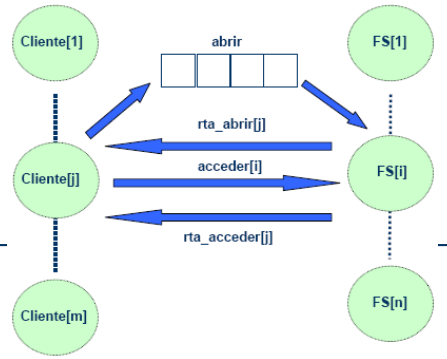
Todos los clientes pueden pedir OPEN por un canal global (qué argumentos son necesarios) y recibirán respuesta de un servidor dado por un canal propio. Por qué??

PMA. Clientes y Servidores. File Servers / Continuidad Conversacional



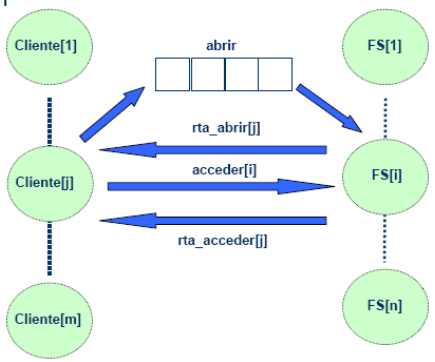
PMA. Clientes y Servidores. File Servers / Continuidad Conversacional

```
type oper = enum(LEER,ESCRIBIR,CERRAR)
chan abrir (STRING nombre_ar; INT id_cliente);
chan acceder[n] (INT oper, otros tipos de argumentos);
chan rta_abrir[m](INT id_servidor);        # Nro. Servidor o Cod. Error
chan rta_acceder[m] (tipos resultado);    # datos, flags de error

Cliente [j = 1 to m] {
        INT id_servidor; otras declaraciones;
        send abrir ("Prueba.doc", j);          # trata de abrir el archivo
        receive rta_abrir[j] (id_servidor);  # Nro. de servidor
        send acceder[id_servidor](argumentos de pedido);
        receive rta_acceder[j](resultados);
      .....
}
```



PMA. Clientes y Servidores. File Servers / Continuidad Conversacional

```
Process File_Server[i = 1 to n] {
        STRING nombre_ar; INT id_cliente;
        oper k; BOOL seguir = false; otras declaraciones;
        WHILE (true) {
                receive abrir(nombre_ar, id_cliente);
                abre archivo nombre_ar ; Si tiene éxito entonces:
                send rta_abrir[id_cliente](i); seguir = true;
                WHILE seguir {
                        receive acceder[i](k,argumentos);
                        IF (k == LEER)
                                procesa pedido de lectura;
                        ELSE IF (k == ESCRIBIR)
                                procesa pedido de escritura;
                        ELSE   # k== CERRAR
                                { procesa el cierre; seguir = false;  }
                        send rta_acceder[id_cliente] (resultados);
                }
        }
}
```



PMA. Clientes y Servidores. File Servers / Continuidad Conversacional

Este ejemplo de interacción entre clientes y servidores se denomina ***continuidad conversacional*** (del ABRIR al CERRAR).

abrir es un canal compartido por el que cualquier FS puede recibir.
Si c/ canal puede tener 1 solo receptor, necesito un alocador de archivos separado, que recibiría pedidos de ABRIR y alocaría un FS libre a 1 cliente; los FS necesitarían decirle al alocador cuándo están libres.

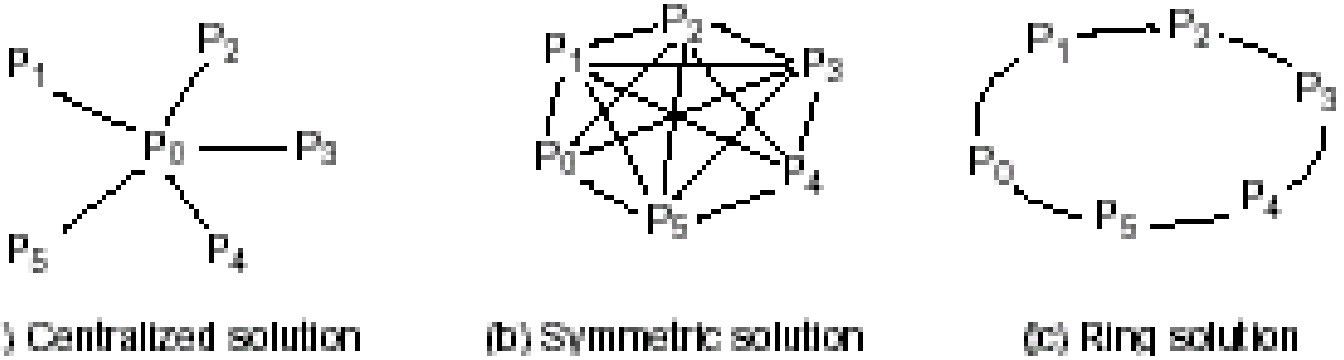
Si el lenguaje soporta creación dinámica de procesos y canales, el número *n* de FS puede adaptarse a los requerimientos del sistema.
Otro esquema sería un file server por disco (interfase más compleja)

Otra solución: Sun Netwok File System: ABRIR $\Rightarrow$ adquirir un descriptor completo del file. Las operaciones sucesivas son RPC trasmitiendo el descriptor (ventajas y desventajas)

PMA. Pares (peers) interactuantes: Intercambio de valores

Ejemplo donde los procesadores están conectados por tres modelos de arquitectura: ***centralizado, simétrico y en anillo circular***.

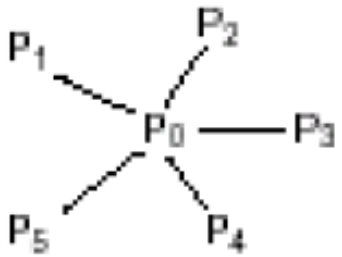
Problema: c/ proceso tiene un dato local **V** y los N procesos deben saber cuál es el menor y cuál el mayor de los valores



La arq. centralizada es apta para una solución en que todos envían su dato local **V** al procesador central, éste ordena los N datos y reenvía la información del mayor y menor a todos los procesos.
$\Rightarrow$ 2(N-1) mensajes.
Si p[0] dispone de una primitiva broadcast se reduce a N mensajes.

PMA. Pares (peers) interactuantes: Intercambio de valores (Solución centralizada)

```
chan vaores(INT), resultados[n] (INT minimo, INT maximo);
Process P[0]  {    #Proceso coordinador. v ya está inicializado.
    INT v; INT nuevo, minimo = v, máximo = v;
    FOR [i=1 to n-1]  {
            receive valores (nuevo);
            IF (nuevo < minimo)
                minimo = nuevo;
            IF (nuevo > maximo)
                maximo = nuevo;
    }
    FOR [i=1 to n-1]
            send resultados [i] (minimo, maximo);
}
Process P[i=1 to n-1]  {    # Proceso cliente. v ya está inicializado.
    INT v; INT minimo, máximo;
    send valores (v);
    receive resultados [i] (minimo, maximo);
}
```



PMA. Pares (peers) interactuantes: Intercambio de valores (Solución simétrica)

En la arquitectura simétrica o “full conected” hay un canal entre cada par de procesos. Todos los procesos ejecutan el mismo algoritmo.

Cada proceso trasmite su dato local v a los n-1 restantes procesos. Luego recibe y procesa los n-1 datos que le faltan, de modo que ***en paralelo*** toda la arquitectura está calculando el mínimo y el máximo y toda la arquitectura tiene acceso a los n datos.

Ejemplo de solución *SPMD*: cada proceso ejecuta el mismo programa pero trabaja sobre datos distintos

$\Rightarrow$ **n(n-1) mensajes**
Si disponemos de una primitiva de broadcast, serán nuevamente n mensajes.

PMA. Pares (peers) interactuantes: Intercambio de valores (Solución simétrica)

```
Chan valores[n] (INT);

Process P[i=0 to n-1]  {              # Todos los procesos idénticos
   INT v;                             # asumimos que v fue inicializado
   INT nuevo, minimo = v, máximo=v; # estado inicial,

   FOR [k=0 to n-1  st  k <> i ]   # envío del dato local
          send valores[k] (v);

   FOR [k=0 to n-1  st  k <> i ] { # recibo y proceso los datos remotos
          receive valores[k] (nuevo);
          IF (nuevo < minimo)
             minimo = nuevo;
          IF (nuevo > maximo)
             maximo = nuevo;
      }
  }
```

PMA. Pares (peers) interactuantes: Intercambio de valores (Solución en anillo circular)

Un tercer modo de organizar la solución es tener un anillo donde P[i] recibe mensajes de P[i-1] y envía mensajes a P[i+1].

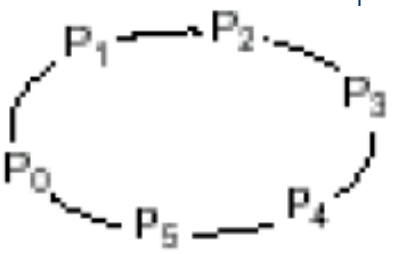
P[n-1] tiene como sucesor a P[0]

Esquema de 2 etapas. En la 1ra c/ proceso recibe dos valores y los compara con su valor local, trasmitiendo un máximo local y un mínimo local a su sucesor. En la 2da etapa todos deben recibir la circulación del máximo y el mínimo global.

P[0] deberá ser algo diferente para "arrancar" el procesamiento.
Se requerirán 2 (n-1) mensajes.
Notar que si bien el número de mensajes es lineal (igual que en la centralizada) los tiempos pueden ser muy diferentes.
Por qué???

PMA. Pares (peers) interactuantes: Intercambio de valores (Solución en anillo circular)

```
chan valores[n] (INT minimo, INT maximo);
Process P[0]  {    # Proceso que inicia los intercambios.
   INT v; INT minimo = v, máximo=v;
   # Enviar v a P[1]
   send valores[1] (minimo, maximo);
   # Recibir los valores minimo y maximo globales de P[n-1] y pasarlos
   receive valores[0] (minimo, maximo);
   send valores[1] (minimo, maximo);   }

Process P[i = 1 to n-1]  {    # Procesos del anillo.
   INT v; INT minimo, máximo;
   # Recibe los valores minimo y maximo hasta P[i-1]
   receive valores[i] (minimo, maximo);
   IF (v < minimo) minimo = v;
   IF (v > maximo) maximo = v;
   # Enviar el minimo y maximo al proceso i+1
   send valores[i+1 MOD n] (minimo, maximo);
   # Esperar el minimo y maximo global
   receive valores[i] (minimo, maximo);
   IF (i < n-1) send valores[i+1] (minimo, maximo);   }
```



PMA. Pares (peers) interactuantes: Intercambio de valores. Comentarios sobre las soluciones

Simétrica es la más corta y sencilla de programar, pero usa el mayor número de mensajes (si no hay broadcast)

Pueden transmitirse en paralelo si la red soporta transmisiones concurrentes, pero el overhead de comunicación acota el speedup.

Centralizada y ***anillo*** usan n° lineal de mensajes, pero tienen distintos patrones de comunicación que llevan a distinta performance

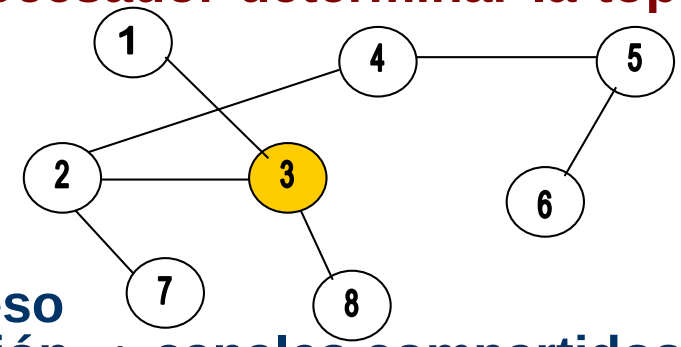
En ***centralizada***, los msgs al coordinador se envían casi al mismo tiempo ⇒ sólo el 1er *receive* del coordinador demora mucho

En ***anillo***, todos los procesos son *productores* y *consumidores*. El último tiene que esperar a que todos los otros (uno por vez) reciban un msg, hacer poco cómputo, y enviar su resultado.
Los msg circulan 2 veces completas por el anillo ⇒ Solución inherentemente lineal y lenta para este problema, pero puede funcionar si cada proceso tiene mucho cómputo.

PMA. Algoritmo Heartbeat para el cálculo de la topología de una red

Procesadores conectados por canales bidireccionales
C/ uno se comunica sólo con sus vecinos y conoce esos links

Cómo puede *cada* procesador determinar la topología *completa* de la red?

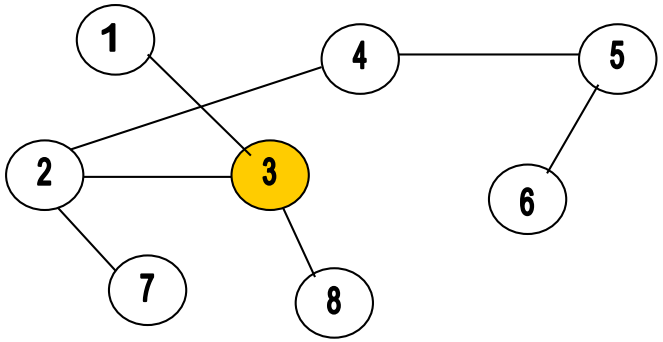


Modelización:
Procesador ⇒ proceso
Links de comunicación ⇒ canales compartidos.

Soluciones posibles:
- los procesos tienen acceso a MC
- distribuida: los vecinos interactúan para intercambiar info local

Algoritmo Heartbeat **→ se expande, enviando información; luego se contrae, incorporando nueva información**

PMA. Algoritmo Heartbeat para el cálculo de la topología de una red



Solución con Variables Compartidas

Procesos *Nodo*[p:1..n]

Vecinos de *p*:

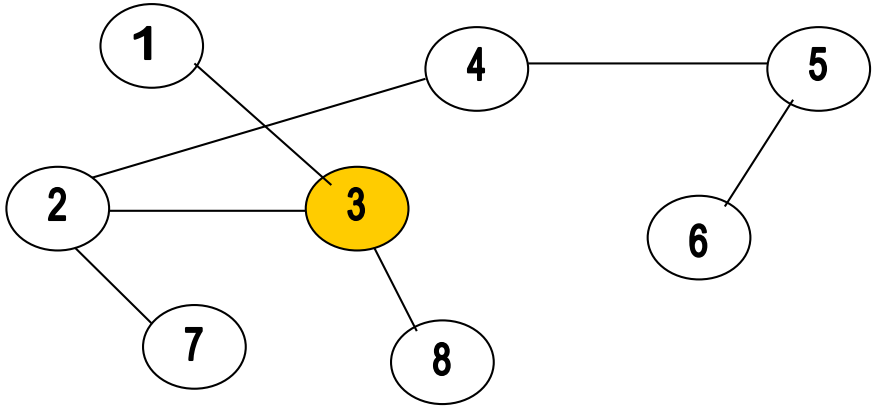
vecinos[1:n], t.q *vecinos*[q] true en *Nodo*[p] si *q* vecino de *p*

Problema: computar *top* (matriz de adyacencia), donde *top*[p,q] true si *p* y *q* son vecinos

```
var top[1:n,1:n] : bool := ([n*n] false)
Process Nodo[p:1..n]  { bool vecinos[1:n];
        # vecinos[q] es true si q es un vecino de Nodo[p]
        for [q = 1 to n st vecinos[q] ]  top[p,q] = true;
        { top[p,1:n] = vecinos[1:n] }
```

PMA. Algoritmo Heartbeat para el cálculo de la topología de una red

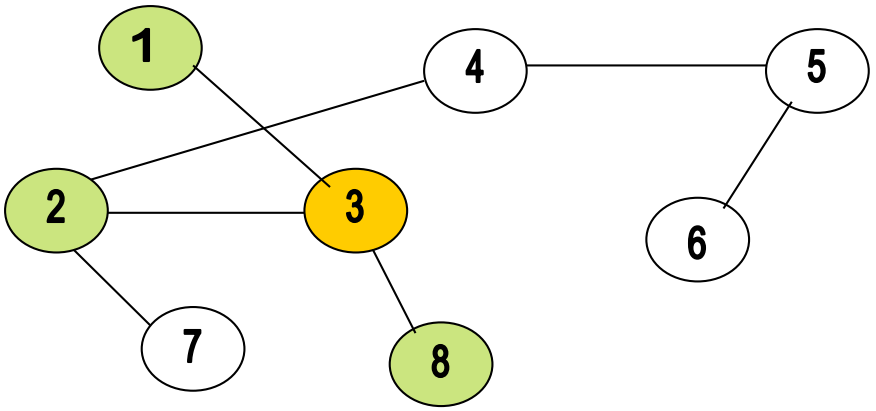
Solución Distribuida



Inicialmente en el nodo 3:

	1	2	3	4	5	6	7	8
1								
2								
3	T	T	T					T
4								
5								
6								
7								
8								

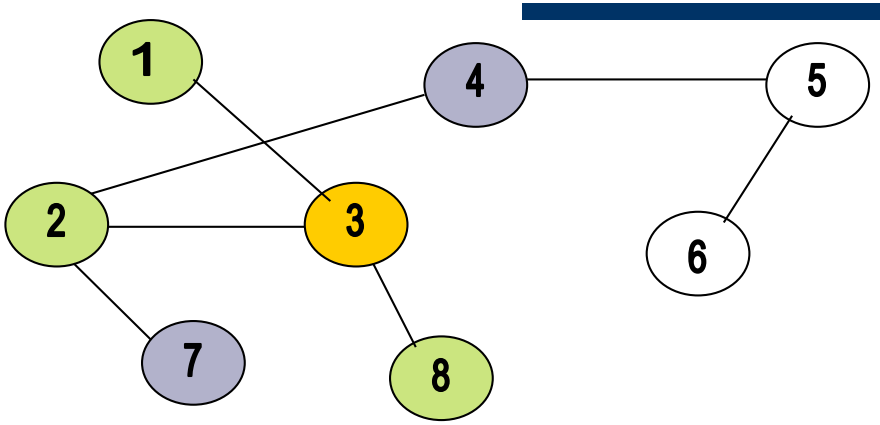
PMA. Algoritmo Heartbeat para el cálculo de la topología de una red



Después de una ronda, en nodo 3:

	1	2	3	4	5	6	7	8
1	T		T					
2		T	T	T			T	
3	T	T	T					T
4								
5								
6								
7								
8			T					T

PMA. Algoritmo Heartbeat para el cálculo de la topología de una red



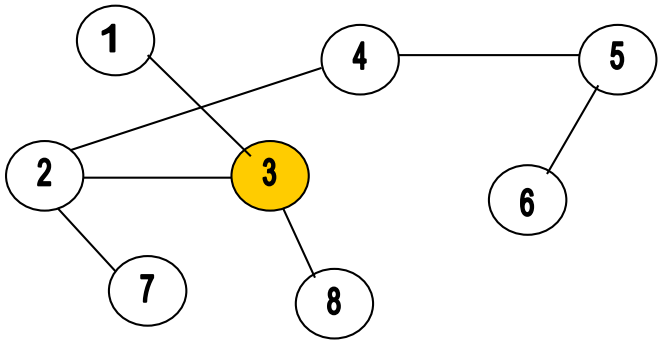
Después de dos rondas, en nodo 3:

	1	2	3	4	5	6	7	8
1	T		T					
2		T	T	T			T	
3	T	T	T					T
4		T		T	T			
5								
6								
7		T					T	
8			T					T

PMA. Algoritmo Heartbeat para el cálculo de la topología de una red

Después de *r* rondas lo siguiente es true ∀ *p*:
RONDA: (∀ q: 1 ≤ q ≤ n: (dist(p,q) ≤ r ⇒ top[q,*] lleno)
C/ nodo debe ejecutar un n° de rondas p/ conocer la top completa
Si el *diámetro* D de la red es conocido ⇒

```
chan topologia[1:n]([1:n,1:n] bool)      # un canal privado por nodo
Process Nodo[p:1..n] { var vecinos[1:n] : bool
  bool top[1:n,1:n] = ([n*n] false)
  top[p,1..n] = vecinos     # inicialmente vecinos[q] true si q es vecino de Nodo[p]
  int r = 0;   bool nuevatop[1:n,1:n];
  while (r < D)  {
     # envía conocimiento local a sus vecinos
     for [q = 1 to n st vecinos[q] ] send topologia[q](top);
     # recibe las topologías y hace or con su top
     for [q = 1 to n st vecinos[q] ] {
           receive topologia[p](nuevatop)
           top = top or nuevatop    }
    r = r + 1
    }
  }
```



PMA. Algoritmo Heartbeat para el cálculo de la topología de una red

- **rara vez se conoce el valor de D**

- **excesivo intercambio de mensajes ⇒ los procesos cercanos al "centro" conocen la topología más pronto y no aprenden nada nuevo en los intercambios**

- **el tema de la terminación ⇒ local o distribuida?**

Cómo se pueden solucionar estos problemas?

PMA. Algoritmo Heartbeat para el cálculo de la topología de una red

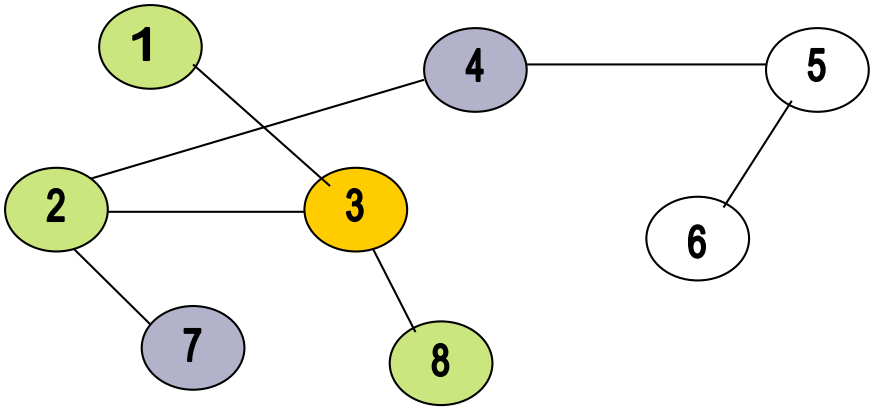
Después de *r* rondas, *p* conoce la topología a distancia *r* de él.

Para cada nodo *q* dentro de la distancia *r* de *p*, los vecinos de *q* estarán almacenados en la fila *q* de *top*

$\Rightarrow$ *p* ejecutó las rondas suficientes tan pronto como cada fila de *top* tiene algún valor true

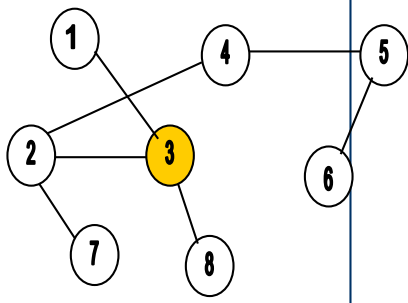
Luego necesita ejecutar una última ronda para intercambiar la topología con sus vecinos

No siempre la terminación se puede determinar localmente



PMA. Algoritmo Heartbeat para el cálculo de la topología de una red

```
chan topologia[1:n](emisor : int; listo : bool; top : [1:n,1:n] bool)
Process Nodo[p:1..n] { bool vecinos[1:n];
        # inicialmente vecinos[q] true si q es vecino de Nodo[p]
        bool activo[1:n] = vecinos              # vecinos aún activos
        bool top[1:n,1:n] = ([n*n]false)        # vecinos conocidos
        int r = 0; bool listo = false;
        int emisor; bool qlisto; bool nuevatop[1:n,1:n];
        top[p,1..n] = vecinos;          # llena la fila para los vecinos
        while (not listo)  {
          # envía conocimiento local de la topología a sus vecinos
          for [q = 1 to n st activo[q] ] send topologia[q](p,false,top);
          # recibe las topologías y hace or con su top
          for [q = 1 to n st activo[q] ]  {
                receive topologia[p](emisor,qlisto,nuevatop);
                top = top or nuevatop;
                if (qlisto) activo[emisor] = false;   }
        if (todas las filas de top tiene 1 entry true) listo=true;
        r := r + 1     }
        # envía topología a todos sus vecinos aún activos
        for [q = 1 to n st activo[q] ] send topologia[q](p,listo,top);
        # recibe un mensaje de cada uno para limpiar el canal
        for [q=1 to n st activo[q]] receive topologia[p](emisor,d,nuevatop)  }
```



Extensión de lenguajes secuenciales con bibliotecas específicas

Una técnica muy utilizada es el desarrollo de bibliotecas de funciones que permiten comunicar/sincronizar procesos, no dependientes de un lenguaje de programación determinado.

Las soluciones basadas en bibliotecas pueden ser menos eficientes que los lenguajes "reales" de pgmción concurrente, aunque permiten "agregarse" al código secuencial con bajo costo de desarrollo.

Las arquitecturas distribuidas han potenciado las soluciones basadas en PVM o MPI, que son básicamente bibliotecas de comunicaciones.
Un esquema anterior (y original) es el de LINDA.

Los pgms MPI usan un estilo SPMD. C/ proceso ejecuta una copia del mismo programa, y puede tomar distintas acciones de acuerdo a su "identidad". Las instancias interactúan llamando a funciones MPI, que soportan comunicación proceso-a-proceso y grupales

La biblioteca MPI. Un ejemplo simple

Dos procesos intercambian valores (14 y 25). Solución empleando MPI:

```
 # include  <mpi.h>
main (INT argc, CHAR *argv [ ] )  {
        INT myid, otherid, size;
        INT length=1, tag=1;
        INT myvalue, othervalue;
        MPI_status  status;
        MPI_Init (&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        MPI_Comm_Rank (MPI_COMM_WORLD, &myid);
        IF (myid == 0)   {
             otherid = 1; myvalue=14;
        } ELSE {
        otherid=0; myvalue=25;
        }
       MPI_send (&myvalue, length, MPI_INT, otherid, tag,MPI_COMM_WORLD);
       MPI_recv (&othervalue, length, MPI_INT, MPI_any_source, tag,
                     MPI_COMM_WORLD, &status);
       printf ("process %d received a %d\n", myid, othervalue);
       MPI_Finalize ( );   }
```

La biblioteca MPI

MPI_Init inicializa la biblioteca MPI y obtiene una copia de los comandos pasados al programa. La variable MPI_COMM_WORLD es inicializada con el conjunto de procesos que se arrancan.
MPI_COMM_Size determina el número de procesos arrancados (2)
MPI_COMM_Rank determina la id del proceso (0 a size-1)
MPI_Finalize Termina este proceso y libera la bilioteca MPI.

MPI_Send envía un mensaje a otro proceso. Los argumentos son: el buffer que contiene el mensaje, el nro de elementos a enviar, el tipo de datos del mensaje, la identidad del proceso destino, un tag del usuario para identificar el tipo de mensaje y la variable MPI_COMM_WORLD.

MPI_Recv recibe un mensaje de otro proceso. Los argumentos son: el buffer en el cual poner el mensaje, el número de elementos del mensaje, el tipo de datos del mensaje, la identidad del proceso que envía o “no importa”= MPI_any_source, un tag del usuario para identificar el tipo de mensaje, el grupo de comunicación y el status de retorno.