



UNIVERSIDAD
DE GRANADA

Sistemas Concurrentes y Distribuidos:

Tema 3. Sistemas basados en paso de mensajes.

Carlos Ureña / Jose M. Mantas / Pedro Villar

2017-18

Grado en Ingeniería Informática / Grado en Ingeniería Informática y Matemáticas.

Dpt. Lenguajes y Sistemas Informáticos

ETSI Informática y de Telecomunicación

Universidad de Granada

Tema 3. Sistemas basados en paso de mensajes.

Índice.

1. Mecanismos básicos en sistemas basados en paso de mensajes
2. Paradigmas de interacción de procesos en programas distribuidos
3. Mecanismos de alto nivel en sistemas distribuidos

Sección 1.

Mecanismos básicos en sistemas basados en paso de mensajes.

- 1.1. Introducción
- 1.2. Vista lógica arquitectura y modelo de ejecución
- 1.3. Primitivas básicas de paso de mensajes
- 1.4. Espera selectiva

Sistemas Concurrentes y Distribuidos., curso 2017-18.

Tema 3. Sistemas basados en paso de mensajes.

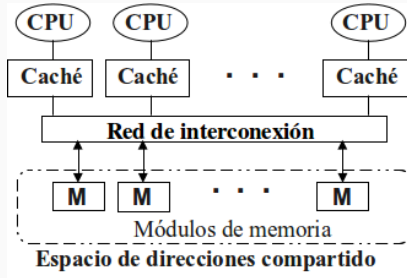
Sección 1. Mecanismos básicos en sistemas basados en paso de mensajes

Subsección 1.1.

Introducción.

Memoria compartida vs. Sistema Distribuido (1)

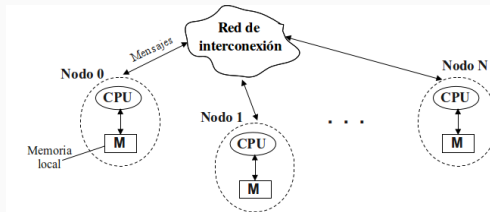
En el capítulo anterior hemos visto los sistemas multiprocesador con memoria compartida:



- ▶ Más fácil programación (variables compartidas): se usan mecanismos como cerrojos, semáforos y monitores.
- ▶ Implementación más costosa y escalabilidad hardware limitada: el acceso a memoria común supone un cuello de botella.

Memoria compartida vs. Sistema Distribuido (2)

Sistemas Distribuidos: es un conjunto de procesos (en uno o varios ordenadores) que no comparten memoria, pero se transmiten datos a través de una red:



- ▶ Facilita la distribución de datos y recursos.
- ▶ Soluciona el problema de la escalabilidad y el elevado coste.
- ▶ Presenta una mayor dificultad de programación: no hay direcciones de memoria comunes y mecanismos como los monitores son inviables.

Necesidad de una notación de progr. distribuida

- ▶ Lenguajes tradicionales (memoria común)
 - ▶ **Asignación:** cambio del estado interno de la máquina.
 - ▶ **Estructuración:** secuencia, repetición, alternación, procedimientos, etc.
- ▶ Extra añadido: **Envío / Recepción** \implies Afectan al entorno externo.
 - ▶ Son tan importantes como la asignación.
 - ▶ Permiten comunicar procesos que se ejecutan en paralelo.
- ▶ Paso de mensajes:
 - ▶ **Abstracción:** oculta hardware (red de interconexión).
 - ▶ **Portabilidad:** se puede implementar eficientemente en cualquier arquitectura (multiprocesador o plataforma distribuida).
 - ▶ No requiere mecanismos para asegurar la exclusión mutua.

Sistemas Concurrentes y Distribuidos., curso 2017-18.

Tema 3. Sistemas basados en paso de mensajes.

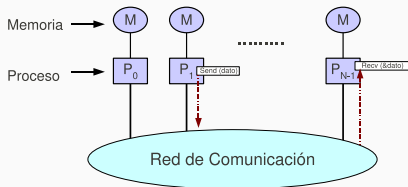
Sección 1. Mecanismos básicos en sistemas basados en paso de mensajes

Subsección 1.2.

Vista lógica arquitectura y modelo de ejecución.

Vista logica de la arquitectura

Existen N procesos, cada uno con su espacio de direcciones propio (memoria). Los procesos se comunican mediante **envío y recepción de mensajes**.

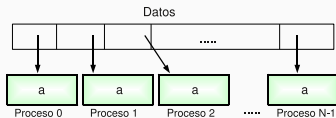


- ▶ En un mismo procesador pueden residir físicamente varios procesos.
- ▶ Por motivos de eficiencia, frecuentemente se adopta la política de alojar un único proceso en cada procesador disponible.
- ▶ **Cada interacción requiere cooperación entre 2 procesos:** el propietario de los datos (emisor) debe intervenir aunque no haya conexión lógica con el evento tratado en el receptor.

Estructura de un programa de paso de mensajes. SPMD

Diseñar un código diferente para cada proceso puede ser complejo. Una solución es el **estilo SPMD (Single Program Multiple Data)**:

- ▶ Todos los procesos ejecutan el mismo código fuente.
- ▶ Cada proceso puede procesar datos distintos y/o ejecutar flujos de control distintos.

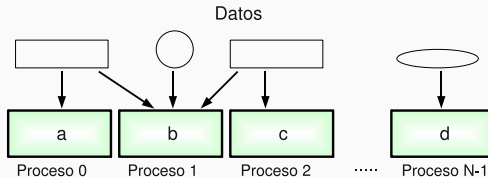


```
process Proceso[ n_proc : 0..1 ];  
  var dato : integer ;  
begin  
  if n_proc == 0 then begin {si soy 0}  
    dato := Produce();  
    send( dato, Proceso[1]);  
  end else begin {si soy 1}  
    receive( dato, Proceso[0] );  
    Consume( dato );  
  end  
end
```

Estructura de un programa de paso de mensajes. MPMD

Otra opción es usar el **estilo MPMD (Multiple Program Multiple Data)**:

- ▶ Cada proceso ejecuta el mismo o diferentes programas de un conjunto de ficheros ejecutables.
- ▶ Los diferentes procesos pueden usar datos diferentes.



```
process ProcesoA ;  
  var var_orig : integer ;  
begin  
  var_orig := Produce();  
  send( var_orig, ProcesoB );  
end
```

```
process ProcesoB ;  
  var var_dest : integer ;  
begin  
  receive( var_dest, ProcesoA );  
  Consume( var_dest );  
end
```

Sistemas Concurrentes y Distribuidos., curso 2017-18.

Tema 3. Sistemas basados en paso de mensajes.

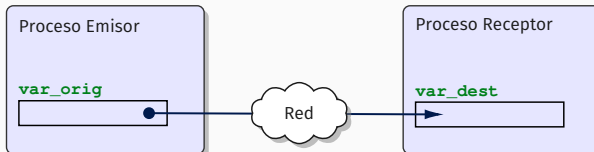
Sección 1. Mecanismos básicos en sistemas basados en paso de mensajes

Subsección 1.3.

Primitivas básicas de paso de mensajes.

Transferencia de mensajes

El paso de un mensaje entre dos procesos constituye una transferencia de una secuencia finita de bytes:



- ▶ Se leen de una variable del **proceso emisor** (`var_orig`).
- ▶ Se transfieren a través de alguna red de interconexión.
- ▶ Se escriben en una variable del **proceso receptor** (`var_dest`).
- ▶ Implica **sincronización**: los bytes acaban de recibirse **después** de iniciado el envío.
- ▶ Ambas variables son del mismo tipo.
- ▶ El efecto neto final es equivalente a una hipotética asignación (`var_dest := var_orig`), si se pudiera hacer (no se puede).

Primitivas básicas de paso de mensajes

El proceso emisor realiza el envío invocando a **send**, y el proceso receptor realiza la recepción invocando a **receive**. La sintaxis es como sigue:

- ▶ **send**(*variable_origen*, *identificador_proceso_destino*)
- ▶ **receive**(*variable_destino*, *identificador_proceso_origen*)

Cada proceso nombra explícitamente al otro, indicando su nombre de proceso como identificador. Pej., aquí vamos la transferencia de un valor entero:

```
process P1 ; { Emisor (produce)}  
  var var_orig : integer ;  
begin  
  var_orig := Produce();  
  send( var_orig, P2 );  
end
```

```
process P2 ; { Receptor (consume)}  
  var var_dest : integer ;  
begin  
  receive( var_dest, P1 );  
  Consume( var_dest );  
end
```

Esquemas de identificación de la comunicación

¿Cómo identifica el emisor al receptor del mensaje y viceversa?

Existen dos posibilidades:

Denominación directa estática

- ▶ El emisor identifica explícitamente al receptor y viceversa.
- ▶ Para la identificación se usan **identificadores de procesos**. Cada identificador es un valor (típicamente un entero) biunivocamente asociado a un proceso en tiempo de compilación (es decir: estáticamente).

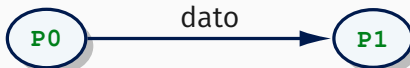
Denominación indirecta

- ▶ Los mensajes se depositan en almacenes intermedios que son accesibles desde todos los procesos (buzones).
- ▶ El emisor nombra a un buzón al que envía. El receptor nombra un buzón desde el que quiere recibir.

Denominación directa estática

```
process P0 ;  
  var var_orig : integer ;  
begin  
  var_orig := Produce();  
  send( var_orig, P1 );  
end
```

```
process P1 ;  
  var var_dest : integer ;  
begin  
  receive( var_dest, P0 );  
  Consume( var_dest );  
end
```



Ventajas

- ▶ No hay retardo para establecer la identificación (los símbolos **P0** y **P1** se traducen en dos enteros en una implementación)

Inconvenientes

- ▶ Cambios en la identificación requieren recompilar el código.
- ▶ Sólo permite comunicación en pares (1-1).

Denominación directa con identificación asimétrica

Existen otras posibilidades llamada **esquemas asimétricos**: el emisor identifica al receptor, pero **el receptor no indica un emisor específico**.

- ▶ El receptor inicia una recepción, pero indica que acepta recibir el mensaje de cualquier otro posible proceso emisor.
- ▶ Una vez recibido el mensaje, el receptor puede conocer que proceso ha sido el emisor, de dos formas, ya que el identificador del emisor
 - ▶ puede formar parte de los metadatos del mensaje.
 - ▶ puede ser un parámetro de salida de **receive**.
- ▶ También es posible especificar que el emisor debe pertenecer a un subconjunto de todos los posibles emisores (se deben definir previamente los subconjuntos de forma coordinada).

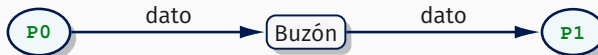
Denominación indirecta

En la denominación indirecta, el emisor y el receptor identifican un buzón o canal intermedio a través del cual se transmiten los mensajes.

```
var buzón : channel of integer; { es accesible por ambos procesos }
```

```
process P1 ;  
  var var_orig : integer ;  
begin  
  var_orig := Produce();  
  send( var_orig, buzón );  
end
```

```
process P2 ;  
  var var_dest : integer ;  
begin  
  receive( var_dest, buzón );  
  Consume( var_dest );  
end
```

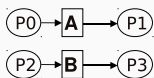


- El uso de buzones da mayor **flexibilidad**: permite comunicaciones entre múltiples receptores y emisores.

Denominación indirecta (2)

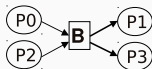
Existen tres tipos de buzones: canales (uno a uno), puertos (muchos a uno) y buzones generales (muchos a muchos):

Canales (1 a 1)



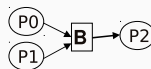
- Tipo fijo

Buzones generales (n a n)



- Destino: *send* de cualquier proc.
- Fuente: *receive* de cualquier proc.
- Implementación complicada.
 - Enviar mensaje y transmitir todos los lugares.
 - Recibir mensaje y notificar recepción a todos.

Puertos (n a 1)



- Destino: Un único proceso
- Fuente: Varios procesos
- Implementación más sencilla.

- Un mensaje enviado a un buzón general permanece en dicho buzón hasta que haya sido leído por todos los procesos receptores potenciales (cada envío es un *broadcast*).

Declaración estática versus dinámica

Los identificadores de proceso suelen ser valores enteros, cada uno de ellos está biunivocamente asociado a un proceso del programa concurrente. Se pueden gestionar:

- ▶ **Estáticamente:** en el código fuente se asocian explícitamente los números enteros a los procesos:
 - ▶ Ventaja: es muy eficiente en tiempo.
 - ▶ Inconveniente: cambios en la estructura del programa (número de procesos de cada tipo) requieren adaptar el código fuente y recompilarlo.
- ▶ **Dinámicamente:** el identificador numérico de cada proceso se calcula en tiempo de ejecución cuando es necesario:
 - ▶ Desventaja: es menos eficiente en tiempo, más complejo.
 - ▶ Ventaja: el código puede seguir siendo válido aunque cambie el número de procesos de cada tipo (no hay que recompilar).

Comportamiento de las operaciones de paso de mensajes

Consideramos esta transmisión de un mensaje

```
process Emisor ;  
  var var_orig : integer := 100 ;  
begin  
  send( var_orig, Receptor ) ;  
  var_orig := 0 ;  
end
```

```
process Receptor ;  
  var var_dest : integer := 1 ;  
begin  
  receive( var_dest, Emisor ) ;  
  imprime( var_dest );  
end
```

- ▶ El comportamiento esperado del **send** es que el valor recibido en **var_dest** será el que tenía **var_orig** (100) justo antes de **send**.
- ▶ Si se garantiza esto, se habla de comportamiento **seguro** (se habla de que el programa de paso de mensajes es seguro).
- ▶ Si pudiera imprimirse 0 o 1 en lugar de 100, el comportamiento sería **no seguro**. Se considera incorrecto, aunque existen situaciones en las que puede interesar usar operaciones que no garantizan dicha seguridad.

Instantes críticos en el lado del emisor

Para poder transmitir un mensaje el **sistema de paso de mensajes (SPM)**, **en el lado del emisor**, debe dar estos pasos:



1. Fin del registro de la **solicitud de envío (SE)**: después de iniciada la llamada a **send**, el SPM registra los identificadores de ambos procesos, y la dirección y tamaño de la variable de origen.
2. **Inicio de lectura (IL)**: el SPM lee el primer byte de todos los que forman el valor de la variable de origen.
3. **Fin de lectura (FL)**: el SPM lee el último byte de todos los que forman el valor de la variable de origen.

Instantes críticos en el lado del receptor

En el **lado del receptor**, el SPM debe dar estos pasos:



1. Fin del registro de la **solicitud de recepción** (*SR*): después de iniciado **receive**, el SPM registra los ident. de procesos y la dirección y tamaño de la variable de destino.
2. Fin del **emparejamiento** (*EM*): el SPM espera hasta que se haya registrado una solicitud de envío que case con la de recepción anterior. Entonces se emparejan ambas.
3. **Inicio de escritura** (*IE*): el SPM escribe en la variable de destino el primer byte recibido.
4. **Fin de escritura** (*FE*): el SPM escribe en la variable de destino el último byte recibido.

Sincronización en el SPM para la transmisión

La transmisión de un mensaje supone sincronización:

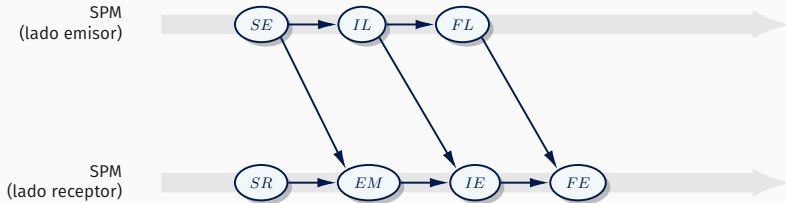
- ▶ El emparejamiento solo puede completarse después de registrada la solicitud de envío es decir, $SE < EM$
- ▶ Antes de que se escriba el primer byte en el receptor, se debe haber comenzado ya la lectura en el emisor, por tanto $IL < IE$.
- ▶ Antes de que se acaben de escribir los datos en el receptor, se deben haber acabado de leer en el emisor, es decir $FL < FE$.
- ▶ Por transitividad, se cumple $IL < FE$.

Sin embargo, no hay orden entre estas acciones:

- ▶ No hay orden predefinido entre SE y SR .
- ▶ No hay orden predefinido entre EM y IL (ni FL).

Grafo dependencia del SPM. Almacenamiento temporal.

Por tanto, los instantes del SPM en ambos lados están relacionados por el siguiente grafo de dependencia:



Se puede iniciar la lectura (*IL*) antes de que ocurra el emparejamiento (*EM*). Si esto se hace:

- ▶ El SPM deberá almacenar temporalmente algunos o todos los bytes de la variable origen en alguna zona de memoria (en el lado del emisor).
- ▶ Esa zona se llama **almacén temporal** de los datos.

Mensajes en tránsito. Memoria.

Por la hipótesis de progreso finito, el intervalo de tiempo entre la solicitud de envío y el fin de la escritura tiene una duración no predecible. Entre *SE* y *FE*, se dice que el **mensaje está en tránsito**.

- ▶ El SPM necesita usar memoria temporal para todos los mensajes en tránsito que esté gestionando en un momento dado.
- ▶ La cantidad de memoria necesaria dependerá de diversos detalles (tamaño y número de los mensajes en tránsito, velocidad de la transmisión de datos, políticas de envío de mensajes, etc...)
- ▶ Dicha memoria puede estar ubicada en el nodo emisor y/o en el receptor y/o en nodos intermedios, si los hay.
- ▶ En un momento dado, el SPM puede detectar que no tiene suficiente memoria para almacenamiento temporal de datos durante el envío (debe retrasar la lectura en el emisor hasta asegurarse que hay memoria para enviar los datos)

Seguridad de las operaciones de paso de mensajes

Las operaciones podrían **no ser seguras**: el valor que el emisor pretendía enviar podría no ser el mismo que el receptor recibe:

- ▶ **Operación de envío-recepción segura**: se puede garantizar a priori que el valor de **var_orig** antes del envío (antes de *SE*) coincidirá con el valor de **var_dest** tras la recepción (después de *FE*)
- ▶ **Operación de envío-recepción insegura**: en dos casos:
 - ▶ **Envío inseguro**: ocurre cuando es posible modificar el valor de **var_orig** entre *SE* y *FL* (podría enviarse un valor distinto del valor en *SE*).
 - ▶ **Recepción insegura**: ocurre cuando se puede acceder a **var_dest** entre *SR* y *FE*, si se lee antes de recibirlo totalmente o se modifica después de haberse ya recibido parcialmente.

Tipos de operaciones de paso de mensajes

Operaciones seguras

- ▶ Devuelven el control cuando se garantiza la seguridad: **send** no espera a la recepción, **receive** sí espera.
- ▶ Existen dos mecanismos de paso de mensajes seguro:
 - ▶ Envío y recepción síncronos.
 - ▶ Envío asíncrono seguro

Operaciones inseguras

- ▶ Devuelven el control inmediatamente tras hacerse la solicitud de envío o recepción, sin garantizar la seguridad.
- ▶ El programador debe asegurar que no se alteran las variables mientras el mensaje está en tránsito.
- ▶ Existen sentencias adicionales para comprobar el estado de la operación.

Operaciones síncronas. Comportamiento.

```
s_send( variable_origen , ident_proceso_receptor ) ;
```

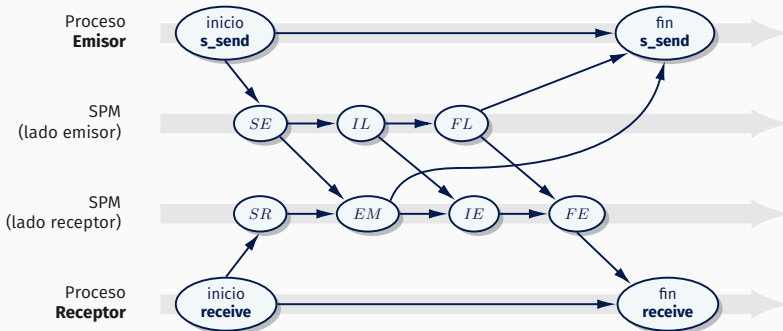
- Realiza el envío de los datos y espera bloqueado hasta que los datos hayan terminado de leerse en el emisor y se haya iniciado y emparejado un **receive** en el receptor.
- Es decir: **s_send** no termina antes de que ocurran *FL* y *EM*.

```
receive( variable_destino , ident_proceso_emisor ) ;
```

- Espera bloqueado hasta que el emisor emita un mensaje con destino al proceso receptor (si no lo había hecho ya), y hasta que hayan terminado de escribirse los datos en la zona de memoria designada en la variable de destino.
- Es decir: **receive** no termina antes de que ocurra *FE*.

Operaciones síncronas: grafo de dependencia

Si usamos **s_send** con **receive** hay estas dependencias:



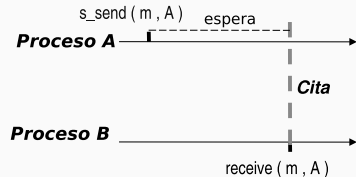
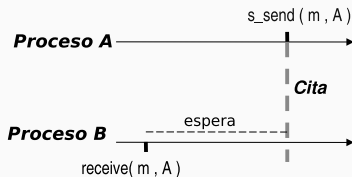
Sincronización entre emisor y receptor:

- El fin de **receive** ocurre después del inicio de **s_send**.
- El fin de **s_send** ocurre después del inicio de **receive**.

Operaciones síncronas. Cita.

Exige **cita** entre emisor y receptor: La operación **s_send** no devuelve el control hasta que el **receive** correspondiente sea alcanzado en el receptor

- ▶ El intercambio de mensaje constituye un punto de sincronización entre emisor y receptor.
- ▶ El emisor podrá hacer aserciones acerca del estado del receptor.
- ▶ Análogo: comunicación telefónica y chat.




Operaciones síncronas. Desventajas.

- ▶ Fácil de implementar pero poco flexible.
- ▶ **Sobrecarga por espera ociosa:** adecuado sólo cuando send/receive se inician aprox. mismo tiempo.
- ▶ **Interbloqueo:** es necesario alternar llamadas en intercambios (código menos legible).

Ejemplo de Interbloqueo

```
{ Proceso P1 }  
s_send( enviado1, P2 );  
receive( recibido1, P2 );
```

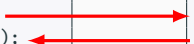
```
{ Proceso P2 }  
s_send( enviado2, P1 );  
receive( recibido2, P1 );
```



Corrección

```
{ Proceso P1 }  
s_send( enviado1, P2 );  
receive( recibido1, P2 );
```

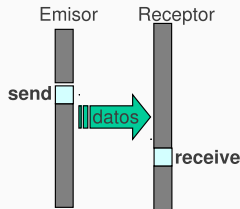
```
{ Proceso P2 }  
receive( recibido2, P1 );  
s_send( enviado2, P1 );
```



Envío asíncrono seguro

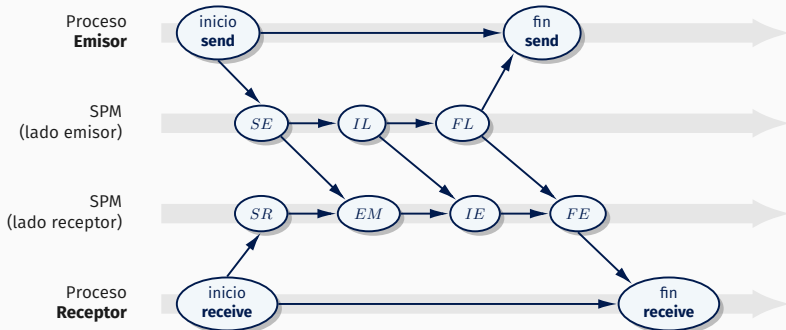
```
send( variable , id_proceso_receptor ) ;
```

- ▶ Inicia el envío de los datos designados y espera bloqueado hasta que hayan terminado de copiarse todos los datos de **variable** a un lugar seguro. Tras la copia de los datos designados, devuelve el control sin que tengan que haberse recibido los datos en el receptor.
- ▶ Por tanto, devuelve el control después de *FL*.
- ▶ Se suele usar junto con la recepción síncrona (**receive**).



Envío asíncrono seguro: grafo de dependencia

Si usamos **send** con **receive** hay estas dependencias:



- El fin de **send** no depende de la actividad del receptor. Puede ocurrir antes, durante o después de la recepción.
- Al igual que ocurre con **s_send**, el fin de **receive** ocurre después del inicio de **send**.

Envío asíncrono seguro: valoración

Ventaja:

- ▶ El uso de **send** lleva en general a menores tiempos de espera bloqueada que **s_send**, ya que no es necesario esperar el emparejamiento.
- ▶ Por tanto, usar **send** es generalmente más eficiente en tiempo y preferible cuando el emisor no tiene que esperar la recepción.

Posible inconveniente:

- ▶ Hay que tener en cuenta que **send** requiere memoria para almacenamiento temporal, el cual, en algunos casos, **puede crecer mucho o incluso indefinidamente**
- ▶ El SPM puede tener que retrasar el inicio de la lectura (IL) en el lado del emisor, cuando detecta que no dispone de memoria suficiente para copiar los bytes y no se ha producido aún el emparejamiento con ningún receptor.

Ejemplo de memoria temporal creciente.

```
Process Productor;  
  var var_orig :  $T$  ;  
begin  
  for i:= 1 to  $N$  do begin  
    var_orig := Produce() ;  
    send( var_orig, Consumidor ) ;  
  end  
end
```

```
Process Consumidor;  
  var var_dest :  $T$  ;  
begin  
  for i:= 1 to  $N$  do begin  
    receive( var_dest, Productor ) ;  
    Consume( var_dest );  
  end  
end
```

Supongamos que generalmente **Produce** tarda menos que **Consume**, y que ocurren algunas de estos dos condiciones:

- ▶ El tamaño de cada variable de tipo T es grande.
- ▶ El valor de N es grande.

Entonces, la cantidad de memoria necesaria para almacenamiento temporal puede crecer mucho, y el SPM puede acabar usando un comportamiento síncrono en **send**.

Interbloqueo con send/receive

Posibilidad de interbloqueo: no se debe olvidar que aunque se hagan los envíos con **send**, las llamadas a **receive** siguen siendo síncronas (o **bloqueantes**: suponen espera bloqueada hasta el fin de la escritura). En este ejemplo

```
process P1 ;  
  var a1, b1 : integer ;  
begin  
  a1 := .... ;  
  receive( b1, P1 );  
  send( a1, P1 );  
end
```

```
process P2 ;  
  var a2, b2 : integer ;  
begin  
  a2 := .... ;  
  receive( b2, P1 );  
  send( a2, P1 );  
end
```

se produce interbloqueo con seguridad, ya que cada proceso queda indefinidamente a la espera de recibir del otro proceso.

Operaciones inseguras

Operaciones seguras: son menos eficientes

- ▶ en tiempo, por las esperas bloqueadas (**s_send/receive**).
- ▶ en memoria, por el almacenamiento temporal (**send/receive**)

Alternativa: operaciones de **inicio de envío o recepción**:

- ▶ Las operaciones devuelven el control antes de que sea seguro modificar (en el caso del envío) o leer los datos (en el caso de la recepción).
- ▶ Deben existir **sentencias de chequeo de estado**: indican si los datos pueden alterarse o leerse sin comprometer la seguridad.
- ▶ Una vez iniciada la operación, el usuario puede realizar cualquier cómputo que no dependa de la finalización de la operación y, cuando sea necesario, chequeará su estado.

Paso de mensajes asíncrono inseguro. Operaciones

```
i_send( variable_orig , id_proc_receptor , var_resguardo ) ;
```

Indica al SPM que comience una operación de envío al receptor:

- ▶ Se registra la solicitud de envío (*SE*) y acaba.
- ▶ No espera a *FL* ni a ninguna acción del receptor.
- ▶ *var_resguardo* permite consultar después el estado del envío.

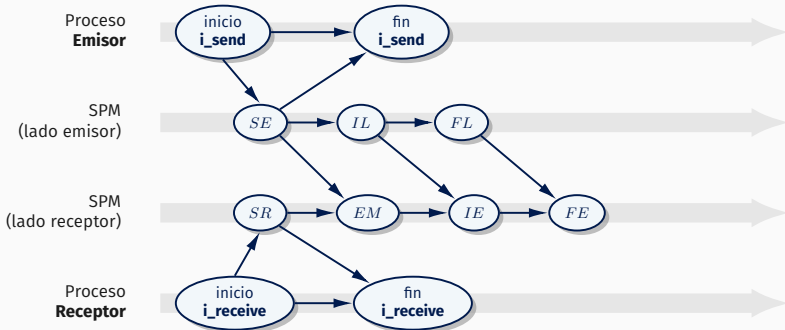
```
i_receive( variable_dest , id_proc_emisor , var_resguardo ) ;
```

Indica al SPM que se inicie una recepción de un msg. del emisor:

- ▶ Se registra la solicitud de recepción (*SR*) y acaba.
- ▶ No espera a *FE* ni a ninguna acción del emisor.
- ▶ *var_resguardo* permite consultar después el estado de la recepción.

Operaciones asíncronas: grafo de dependencia

El grafo de dependencia entre las acciones es este:



En general, las operaciones asíncronas tardan mucho menos que las síncronas, ya que simplemente suponen el registro de la solicitud de envío o de recepción.

Esperar el final de operaciones asíncronas

Cuando un proceso hace **i_send** o **i_receive** puede continuar trabajando hasta que llega un momento en el que debe esperar a que termine la operación. Se disponen de estos dos procedimientos:

```
wait_send( var_resguardo ) ;
```

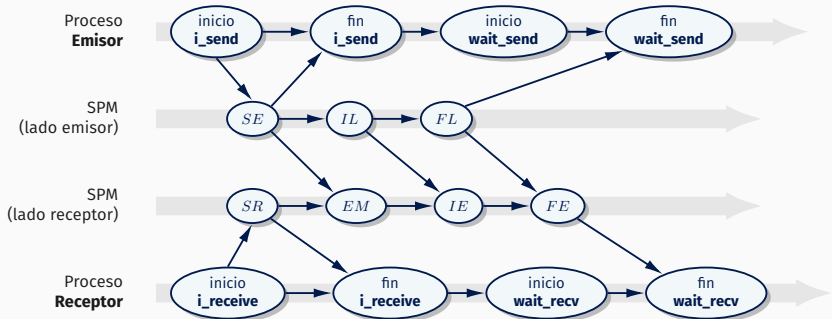
- ▶ Se invoca por el proceso emisor, y lo bloquea hasta que la operación de envío asociada a *var_resguardo* ha llegado al instante *FL* (hasta que es seguro volver a usar la variable de origen)

```
wait_recv( var_resguardo ) ;
```

- ▶ Se invoca por el proceso receptor, que queda bloqueado hasta que la operación de recepción asociada a *var_resguardo* ha llegado al instante *FE* (hasta que se han recibido los datos)

Operaciones asíncronas y espera: grafo de dependencia

Por tanto, cuando se usan las operaciones asíncronas, el inicio y final de las operaciones de espera sigue a las correspondientes operaciones de envío o recepción:



(**wait_send** no espera el emparejamiento, es seguro pero no síncrono)

Operaciones asíncronas: utilidad

Estas operaciones permiten a los procesos emisor y receptor hacer trabajo útil concurrentemente con la operación de envío o recepción

- ▶ **Mejora:** el tiempo de espera ociosa se puede emplear en computación (se aprovechan mejor las CPUs disponibles)
- ▶ **Coste:** reestructuración programa, mayor esfuerzo del programador.

```
process Emisor ;  
  var a : integer := 100 ;  
begin  
  i_send( a, Receptor, resg );  
  { trabajo útil: no escribe en a }  
  trabajo_util_emisor();  
  wait_send( resg );  
  a := 0 ;  
end
```

```
process Receptor ;  
  var b : integer ;  
begin  
  i_receive( b, Emisor, resg );  
  { trabajo útil: no accede a b }  
  trabajo_util_receptor();  
  wait_rcv( resg );  
  print( b );  
end
```

Ejemplo: Productor-Consumidor con paso asíncrono

Se logra simultanear transmisión, producción y consumición:

```
process Productor ;  
  var x, x_env : integer ;  
begin  
  x := Produce() ;  
  while true do begin  
    x_env := x ;  
    i_send( x_env, Consumidor, resg);  
    x := Produce() ;  
    wait_send( resg );  
  end  
end
```

```
process Consumidor ;  
  var y, y_rec : integer ;  
begin  
  i_receive( y_rec, Productor, resg);  
  while true do begin  
    wait_rcv( resg );  
    y := y_rec ;  
    i_receive( y_rec, Productor, resg);  
    Consume( y );  
  end  
end
```

Limitaciones:

- ▶ La duración del trabajo útil podría ser muy distinta de la de cada transmisión.
- ▶ Se descarta la posibilidad de esperar más de un posible emisor.

Consulta de estado de operaciones asíncronas

Para solventar los dos problemas descritos, se pueden usar dos funciones de comprobación de estado de una transmisión de un mensaje. No suponen bloqueo, solo una consulta:

```
test_send( var_resguardo ) ;
```

- Función lógica que se invoca por el emisor. Si el envío asociado a *var_resguardo* ha llegado al fin de la lectura (*FL*), devuelve **true**, sino devuelve **false**.

```
test_recv( var_resguardo ) ;
```

- Función lógica que se invoca por el receptor. Si el envío asociado a *var_resguardo* ha llegado al fin de la escritura (*FE*), devuelve **true**, sino devuelve **false**.

Paso asíncrono con espera ocupada posterior.

El trabajo útil de nuevo se puede simultánear con la transmisión del mensaje, pero en este caso dicho debe descomponerse en muchas tareas pequeñas.

```
process Emisor ;
  var a : integer := 100 ;
begin
  i_send( a, Receptor, resg );
  while not test_send( resg ) do begin
    {trabajo útil: no escribe en a}
    trabajo_util_emisor();
  end
  a := 0 ;
end

process Receptor ;
  var b : integer ;
begin
  i_receive( b, Emisor, resg );
  while not test_rcv( resg ) do begin
    {trabajo útil: no accede a b}
    trabajo_util_receptor();
  end
  print( b );
end
```

Si el trabajo útil es muy corto, puede convertirse en una espera ocupada que consume muchísima CPU inútilmente. Es complejo de ajustar bien.

Recepción simultánea de varios emisores.

En este caso se comprueba continuamente si se ha recibido un mensaje de uno cualquiera de dos emisores, y se espera (con **espera ocupada**) hasta que se han recibido de los dos:

```
process Emisor1 ;
  var a : integer:= 100;
begin
  send( a, Receptor);
end

process Emisor2 ;
  var b : integer:= 200;
begin
  send( b, Receptor);
end
```

```
process Receptor ;
  var b1, b2 : integer ;
      r1, r2 : boolean := false ;
begin
  i_receive( b1, Emisor1, resg1 );
  i_receive( b2, Emisor2, resg2 );
  while not r1 or not r2 do begin
    if not r1 and test_recv( resg1 ) then begin
      r1 := true ;
      print("recibido de 1 : ", b1 );
    end
    if not r2 and test_recv( resg2 ) then begin
      r2 := true ;
      print("recibido de 2 : ", b2 );
    end
  end
end
end
```

Espera bloqueada de múltiples procesos

Usando las operaciones **i_receive** junto con las de test, se usa **espera ocupada**, de forma que:

- ▶ Se espera un mensaje cualquiera proveniente de **varios emisores**.
- ▶ Tras recibir **el primero de los mensajes** se ejecuta una acción, independientemente de cual sea el emisor de ese mensaje.
- ▶ Entre la recepción del primer mensaje y la acción **el retraso es normalmente pequeño**.

Sin embargo, con las primitivas vistas, **no es posible** cumplir estos requisitos usando **espera bloqueada**:

- ▶ es inevitable seleccionar de antemano de que emisor queremos esperar recibir en primer lugar, y este emisor no coincide necesariamente con el emisor del primer mensaje que realmente podría recibirse.

Sistemas Concurrentes y Distribuidos., curso 2017-18.

Tema 3. Sistemas basados en paso de mensajes.

Sección 1. Mecanismos básicos en sistemas basados en paso de mensajes

Subsección 1.4.

Espera selectiva.

Espera bloqueada de múltiples procesos

La **espera selectiva** es una operación que permite **espera bloqueada de múltiples emisores**. Se usan las palabras clave **select** y **when**. El ejemplo que hemos visto antes puede implementarse de esta forma:

```
process Emisor1 ;  
  var a : integer:= 100;  
begin  
  send( a, Receptor);  
end  
  
process Emisor2 ;  
  var b : integer:= 200;  
begin  
  send( b, Receptor);  
end
```


```
process Receptor ;  
  var b1, b2 : integer ;  
      r1, r2 : boolean := false ;  
begin  
  while not r1 or not r2 do begin  
    select  
      when receive( b1, Emisor1 ) do  
        r1 := true ;  
        print("recibido de 1 : ", b1 );  
      when receive( b2, Emisor2 ) do  
        r2 := true ;  
        print("recibido de 2 : ", b2 );  
    end  
  
  end { while }  
end { process }
```

Productor-consumidor distribuido

Consideramos una solución muy simple, en la cual el productor usa **s_send**:

```
process Productor ;  
begin  
  while true do begin  
    v := Produce();  
    s_send( v, Consumidor );  
  end  
end
```

```
process Consumidor ;  
begin  
  while true do begin  
    receive( v, Productor);  
    Consume(v);  
  end  
end
```

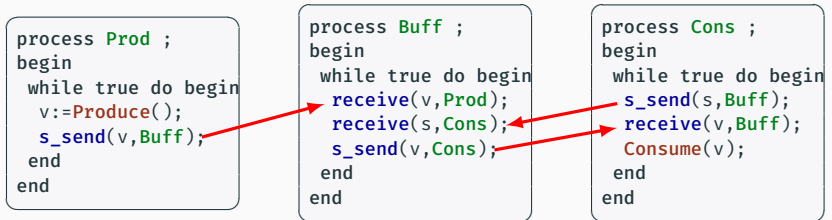


Produce y **Consume** pueden tardar tiempos distintos:

- ▶ Si usamos **send**, el SPM tendría que alojar memoria para almacenamiento temporal, y la cantidad de memoria podría crecer, quizás indefinidamente.
- ▶ **Problema:** Al usar **s_send** se pueden introducir esperas largas (hay bajo aprovechamiento de las CPUs disponibles).

Proceso intermedio

Para intentar reducir las esperas, usamos un proceso intermedio (**Buff**) que acepte peticiones del productor y el consumidor



Problema: el proceso intermedio se bloquea por turnos para esperar bien al emisor, bien al receptor, pero nunca a los dos a la vez. No estamos solucionando las esperas excesivas:

Espera selectiva y buffer FIFO intermedio

Para solucionar el problema descrito usamos **espera selectiva** en el proceso intermedio: ahora dicho proceso puede esperar a ambos procesos a la vez. Para reducir las esperas, usamos un vector de datos pendientes de leer (FIFO):

```
process Prod ;  
var v:integer;  
begin  
  while true do  
    begin  
      v:=Produce();  
      s_send(v, Buff);  
    end  
  end  
end
```

```
process Buff ;  
var esc, lec, cont : integer := 0 ;  
buf : array[0..tam-1] of integer ;  
begin  
  while true do  
    select  
      when cont < tam receive(v, Prod) do  
        buf[esc] := v ;  
        esc := (esc+1) mod tam ;  
        cont := cont+1 ;  
      when 0 < cont receive(s, Cons) do  
        s_send(buf[lec], Cons);  
        lec := (lec+1) mod tam ;  
        cont := cont-1 ;  
      end  
    end  
  end  
end
```

```
process Cons ;  
var v:integer;  
begin  
  while true do  
    begin  
      s_send(s, Buff);  
      receive(v, Buff);  
      Consume(v);  
    end  
  end  
end
```

Sintaxis general y comportamiento

En general, la **espera selectiva** con varias alternativas guardadas es una nueva sentencia compuesta, con la siguiente sintaxis:

```
select
  when condicion1 receive( variable1, proceso1 ) do
    sentencias1
  when condicion2 receive( variable2, proceso2 ) do
    sentencias2
  ...
  when condicionn receive( variablen, proceson ) do
    sentenciasn
end
```

- ▶ Cada bloque que comienza en **when** se llama una **alternativa**.
- ▶ El texto entre **when** y **do** se llama la **guarda** de dicha alternativa, incluye una expresión lógica (*condicion*_{*i*}).
- ▶ Cada **receive** nombra a otro proceso (*proceso*_{*i*}), y a una variable local (*variable*_{*i*}).

Sintaxis de las guardas.

En una guarda de una orden **select**:

- ▶ La expresión lógica puede omitirse, es ese caso la sintaxis sería:

```
when receive( mensaje, proceso ) do  
    sentencias
```

que es equivalente a:

```
when true receive( mensaje, proceso ) do  
    sentencias
```

- ▶ La sentencia **receive** puede no aparecer, la sintaxis es:

```
when condicion do  
    sentencias
```

decimos que esta es una guarda sin sentencia de entrada.

Guardas ejecutables. Evaluación de las guardas.

Una guarda es **ejecutable** en un momento de la ejecución de un proceso P cuando se dan estas dos condiciones:

- ▶ La condición de la guarda se evalúa en ese momento a **true**.
- ▶ Si tiene sentencia de entrada, entonces el proceso origen nombrado ya ha iniciado en ese momento una sentencia **send** (de cualquier tipo) con destino al proceso P , que casa con el **receive**.

Una guarda será **potencialmente ejecutable** si se dan estas dos condiciones:

- ▶ La condición de la guarda se evalúa a **true**.
- ▶ Tiene una sentencia de entrada, sentencia que nombra a un proceso que no ha iniciado aún un **send** hacia P .

Una guarda será **no ejecutable** en el resto de los casos, en los que forzosamente la condición de la guarda se evalúa a **false**.

Ejecución de select: selección de una alternativa

Para ejecutar **select**, al inicio se selecciona una alternativa:

- ▶ Si hay guardas ejecutables con sentencia de entrada: se selecciona aquella cuyo **send** se inició antes (esto garantiza a veces la equidad).
- ▶ Si hay guardas ejecutables, pero ninguna tiene una sentencia de entrada: se selecciona aleatoriamente una cualquiera.
- ▶ Si no hay ninguna guarda ejecutable, pero sí hay guardas potencialmente ejecutables: se espera (bloqueado) a que alguno de los procesos nombrados en esas guardas inicie un **send**, en ese momento acaba la espera y se selecciona la guarda correspondiente a ese proceso.
- ▶ Si no hay guardas ejecutables ni potencialmente ejecutables: no se selecciona ninguna guarda.

Ejecución de select: ejecución de una alternativa

Una vez se ha intentando seleccionar la guarda:

- ▶ Si no se ha podido, no se hace nada (no hay guardas ni ejecutables, ni potencialmente ejecutables) y finaliza la ejecución de **select**.
- ▶ Si se ha podido, se dan estos dos pasos en secuencia:
 1. Si esa guarda tiene sentencia de entrada, se ejecuta el **receive** (siempre habrá un **send** iniciado), y se recibe el mensaje.
 2. Se ejecuta la sentencia asociada a la alternativa.y después finaliza la ejecución del select.

Hay que tener en cuenta que **select** conlleva potencialmente esperas, y por tanto se pueden producir esperas indefinidas (interbloqueo).

Select con guardas indexadas

A veces es necesario replicar una alternativa. En estos casos se puede usar una sintaxis que evita reescribir el código muchas veces, con esta sintaxis:

```
for indice := inicial to final
  when condicion receive( mensaje, proceso ) do
    sentencias
```

Tanto la *condición*, como el *mensaje*, el *proceso* o la *sentencia* pueden contener referencias a la variable *indice* (usualmente es *i* o *j*). Es equivalente a:

```
when condicion receive( mensaje, proceso ) do
  sentencias { se sustituye indice por inicial }
when condicion receive( mensaje, proceso ) do
  sentencias { se sustituye indice por inicial + 1 }
...
when condicion receive( mensaje, proceso ) do
  sentencias { se sustituye indice por final }
```

Ejemplo de select con guardas indexadas

A modo de ejemplo, si `suma` es un vector de n enteros, y `fuente[0]`, `fuente[1]`, etc... son n procesos, entonces:

```
for i := 0 to n-1
  when suma[i] < 1000 receive( numero, fuente[i] ) do
    suma[i] := suma[i] + numero ;
```

Es equivalente a:

```
when suma[0] < 1000 receive( numero, fuente[0] ) do
  suma[0] := suma[0] + numero ;
when suma[1] < 1000 receive( numero, fuente[1] ) do
  suma[1] := suma[1] + numero ;
...
when suma[n-1] < 1000 receive( numero, fuente[n-1] ) do
  suma[n-1] := suma[n-1] + numero ;
```

En un **select** se pueden combinar una o varias alternativas indexadas con alternativas normales no indexadas.

Ejemplo de select

Suma los primeros números de cada proceso hasta llegar a 1000:

```
process Fuente[ i : 0..n-1 ] ;
    var numero : integer ;
begin
    while true do begin
        numero := .... ; s_send( numero, Cuenta );
    end
end
process Cuenta ;
var suma      : array[0..n-1] of integer := (0,0,...,0) ;
    continuar : boolean := true ;
    numero     : integer ;
begin
    while continuar do begin
        continuar := false ; { terminar cuando  $\forall i \text{ suma}[i] \geq 1000$  }
        select
            for i := 0 to n-1
            when suma[i] < 1000 receive( numero, Fuente[i] ) do
                suma[i] := suma[i]+numero ; { sumar }
                continuar := true ; { iterar de nuevo }
            end
        end
    end
end
```

Sección 2.

Paradigmas de interacción de procesos en programas distribuidos.

- 2.1. Introducción
- 2.2. Maestro-Esclavo
- 2.3. Iteración síncrona
- 2.4. Encauzamiento (pipelining)

Sistemas Concurrentes y Distribuidos., curso 2017-18.

Tema 3. Sistemas basados en paso de mensajes.

Sección 2. Paradigmas de interacción de procesos en programas distribuidos

Subsección 2.1.

Introducción.

Paradigma de interacción

Un **paradigma** de interacción define un esquema de interacción entre procesos y una estructura de control que aparece en múltiples programas.

- ▶ Unos pocos paradigmas de interacción se utilizan repetidamente para desarrollar muchos programas distribuidos.
- ▶ Veremos los siguientes paradigmas de interacción:
 1. Maestro-Eslavo.
 2. Iteración síncrona.
 3. Encauzamiento (pipelining).
 4. Cliente-Servidor.
- ▶ Se usan principalmente en programación paralela, excepto el ultimo que es más general (sistemas distribuidos) y se verá en el siguiente apartado del capítulo (Mecanismos de alto nivel para paso de mensajes).

Sistemas Concurrentes y Distribuidos., curso 2017-18.

Tema 3. Sistemas basados en paso de mensajes.

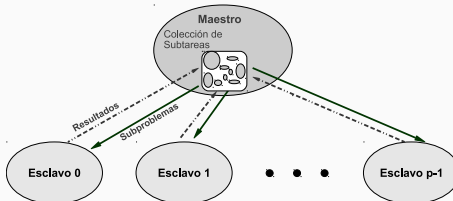
Sección 2. Paradigmas de interacción de procesos en programas distribuidos

Subsección 2.2.

Maestro-Esclavo.

Maestro-Esclavo

- ▶ En este patrón de interacción intervienen dos entidades: un proceso maestro y múltiples procesos esclavos.
- ▶ El **proceso maestro** descompone el problema en pequeñas subtarefas (que guarda en una colección), las distribuye entre los procesos esclavos y va recibiendo los resultados parciales de estos, de cara a producir el resultado final.
- ▶ Los **procesos esclavos** ejecutan un ciclo muy simple hasta que el maestro informa del final del cómputo: reciben un mensaje con la tarea, procesan la tarea y envían el resultado al maestro.

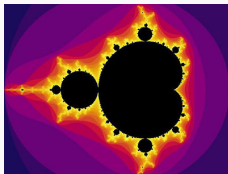


Ejemplo: Cálculo del Conjunto de Mandelbrot

- **Conjunto de Mandelbrot:** Conjunto de puntos c del plano complejo (dentro de un círculo de radio 2 centrado en el origen) que no excederán cierto límite cuando se calculan realizando la siguiente iteración (inicialmente $z = 0$ con $z = a + bi \in \mathbb{C}$):

Repetir $z_{k+1} := z_k^2 + c$ hasta $\|z\| > 2$ o $k > \text{límite}$

- Se asocia a cada pixel (con centro en el punto c) un color en función del número de iteraciones (k) necesarias para su cálculo.
- **Conjunto solución**= {pixels que agoten iteraciones límite dentro de un círculo de radio 2 centrado en el origen}.



Ejemplo: Cálculo del Conjunto de Mandelbrot

- ▶ **Paralelización sencilla:** Cada pixel se puede calcular sin ninguna información del resto
- ▶ **Primera aproximación:** asignar un número de pixels fijo a cada proceso esclavo y recibir resultados.
 - ▶ **Problema:** Algunos procesos esclavos tendrían más trabajo que otros (el número de iteraciones por pixel no es fijo) para cada pixel.
- ▶ **Segunda aproximación:**
 - ▶ El maestro tiene asociada una colección de filas de pixels.
 - ▶ Cuando los procesos esclavos están ociosos esperan recibir una fila de pixels.
 - ▶ Cuando no quedan más filas, el Maestro espera a que todos los procesos completen sus tareas pendientes e informa de la finalización del cálculo.
- ▶ Veremos una solución que usa envío asíncrono seguro y recepción síncrona.

Procesos Maestro y Esclavo

```
process Maestro ;
begin
  for i := 0 to num_esclavos-1 do
    send( fila, Esclavo[i] ) ;    { enviar trabajo a esclavo    }
  while queden filas sin colorear do
    select
      for j := 0 to  $n_e - 1$  when receive( colores, Esclavo[j] ) do
        if quedan filas en la bolsa
          then send( fila, Esclavo[j] )
          else send( fin, Esclavo[j] );
        visualiza(colores);
      end
    end
  end
end
```

```
process Esclavo[ i : 0..num_esclavos-1 ] ;
begin
  receive( mensaje, Maestro );
  while mensaje != fin do begin
    colores := calcula_colores(mensaje.fila) ;
    send (colores, Maestro );
    receive( mensaje, Maestro );
  end
end
```

Sistemas Concurrentes y Distribuidos., curso 2017-18.

Tema 3. Sistemas basados en paso de mensajes.

Sección 2. Paradigmas de interacción de procesos en programas distribuidos

Subsección 2.3. Iteración síncrona.

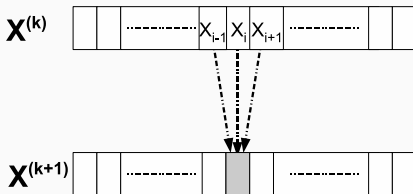
Iteración síncrona

- ▶ **Iteración:** En múltiples problemas numéricos, un cálculo se repite y cada vez se obtiene un resultado que se utiliza en el siguiente cálculo. El proceso se repite hasta obtener los resultados deseados.
- ▶ A menudo se pueden realizar los cálculos de cada iteración de forma concurrente.
- ▶ **Paradigma de iteración síncrona:**
 - ▶ En un bucle diversos procesos comienzan juntos en el inicio de cada iteración.
 - ▶ La siguiente iteración no puede comenzar hasta que todos los procesos hayan acabado la previa.
 - ▶ Los procesos suelen intercambiar información en cada iteración.

Ejemplo: Transformación iterativa de un vector (1)

Supongamos que debemos realizar M iteraciones de un cálculo que transforma un vector x de n reales:

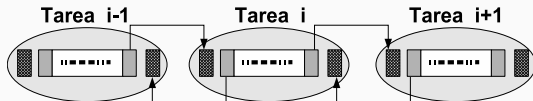
$$x_i^{(k+1)} = \frac{x_{i-1}^{(k)} - x_i^{(k)} + x_{i+1}^{(k)}}{2}, \quad i = 0, \dots, n-1, \quad k = 0, 1, \dots, N,$$
$$x_{-1}^{(k)} = x_{n-1}^{(k)}, \quad x_n^{(k)} = x_0^{(k)}.$$



Veremos una solución que usa envío asíncrono seguro y recepción síncrona.

Ejemplo: Transformación iterativa de un vector (2)

El esquema que se usará para implementar será este:



- ▶ El número de iteraciones es una constante predefinida m
- ▶ Se lanzan p procesos concurrentes.
- ▶ Cada proceso guarda una parte del vector completo, esa parte es un vector local con n/p entradas reales, indexadas de 0 a $n/p - 1$ (vector **bloque**) (asumimos que n es múltiplo de p)
- ▶ Cada proceso, al inicio de cada iteración, se comunica con sus dos vecinos las entradas primera y última de su bloque.
- ▶ Al inicio de cada proceso, se leen los valores iniciales de un vector compartido que se llama **valores** (con n entradas), al final se copian los resultados en dicho vector.

Ejemplo: Transformación iterativa de un vector (3)

El código de cada proceso puede quedar así:

```
process Tarea[ i : 0..p-1 ] ;  
  var bloque : array[0..n/p-1] of float ; { bloque de datos }  
begin  
  receive( bloque, Coordinador ) { recibe valores iniciales }  
  for k := 0 to m do begin { bucle que ejecuta las iteraciones }  
    { comunicacion de valores extremos con los vecinos }  
    send( bloque[0] , Tarea[i-1 mod p] ); { enviar primero a anterior }  
    send( bloque[n/p-1], Tarea[i+1 mod p] ); { enviar ultimo a siguiente }  
    receive( izquierda, Tarea[i-1 mod p] ); { recibir ultimo de anterior }  
    receive( derecha, Tarea[i+1 mod p] ); { recibir primero del sigueie. }  
    { calcular todas las entradas excepto la ultima }  
    for j := 0 to n/p-2 do begin  
      tmp := bloque[j] ;  
      bloque[j] := ( izquierda - bloque[j] + bloque[j+1] )/2;  
      izquierda := tmp ;  
    end  
    { calcular ultima entrada }  
    bloque[n/p-1] := ( izquierda - bloque[n/p-1] + derecha )/2;  
  end  
  send( bloque, Coordinador ); { envía resultados }  
end
```

Proceso coordinador

El proceso coordinador se encarga de repartir los bloques con los valores iniciales, y de recibir los bloques con los valores finales:

```
process Coordinador ;
var
  inicial    : array[0..n-1] of float ; { valores iniciales}
  resultado  : array[0..n-1] of float ; { valores resultado }
  bloque     : array[0..n/p-1] of float; { para envío o recepción }
begin
  for i :=0  to n-1 do
    inicial[i] := ..... ; { calcula valores iniciales }
  for j := 0 to p-1 do begin
    for k := 0 to n/p-1 do bloque[k] := inicial[j*n/p+k] ; {copia}
    send( bloque, Tarea[j] ); { enviar bloque a la tarea j }
  end
  for j := 0 to p-1 do begin
    receive( bloque, Tarea[j] ) ; { recibe bloque de tarea j }
    for k := 0 to n/p-1 do resultado[j*n/p+k] := bloque[k] ; {copia}
  end
  for i :=0  to n-1 do
    print("resultado[",i,"] == ",resultado[i]);
  end
end
```

Sistemas Concurrentes y Distribuidos., curso 2017-18.

Tema 3. Sistemas basados en paso de mensajes.

Sección 2. Paradigmas de interacción de procesos en programas distribuidos

Subsección 2.4.

Encauzamiento (pipelining).

Encauzamiento (pipelining)

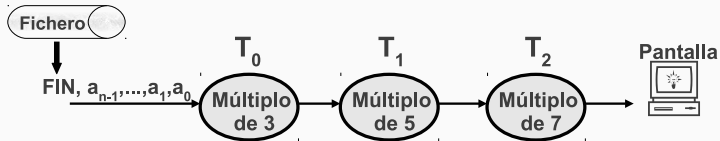
- ▶ El problema se divide en una serie de tareas que se han de completar una después de otra
- ▶ Cada tarea se ejecuta por un proceso separado.
- ▶ Los procesos se organizan en un cauce (pipeline) donde cada proceso se corresponde con una *etapa* del cauce y es responsable de una tarea particular.
- ▶ Cada etapa del cauce contribuirá al problema global y devuelve información que es necesaria para etapas posteriores del cauce.
- ▶ Patrón de comunicación muy simple ya que se establece un flujo de datos entre las tareas adyacentes en el cauce.



Encauzamiento: Ejemplo (1)

Cauce paralelo para filtrar una lista de enteros

- ▶ Dada una serie de m primos p_0, p_1, \dots, p_{m-1} y una lista de n enteros, $a_0, a_1, a_2, \dots, a_{n-1}$, encontrar aquellos números de la lista que son múltiplos de todos los m primos ($n \gg m$)
- ▶ El proceso **E**tapa[i] (con $i = 0, \dots, m - 1$) mantiene el primo p_i y chequea multiplicidad con p_i .
- ▶ Veremos una solución que usa operaciones síncronas.



Encauzamiento: Ejemplo (2)

```
process Etapa[ i : 0..m-1 ] ;

{ vector (local) con la lista de primos }
var primos      : array[0..m-1] of float := {  $p_0, p_1, p_2, \dots, p_{m-1}$  } ;
    izquierda : integer := 0 ; { variable recibida del proceso a la izq.}

begin
    while 0 <= izquierda do begin
        if i == 0 then                                { si es el primero: }
            leer( izquierda );                        {   obtiene siguiente entero }
        else                                           { si no es el primero }
            receive( izquierda, Etapa[i-1]); {   recibir del anterior }

        if izquierda mod primos[i] == 0 then begin { si es múltiplo }
            if i != m-1 then                          { si no es el último }
                s_send ( izquierda, Etapa[i+1]); {   enviar al sig. }
            else                                         { si es el último: }
                print( izquierda );                  {   imprimir núm. }
            end
        end
    end
end
```

Sección 3.

Mecanismos de alto nivel en sistemas distribuidos.

- 3.1. Introducción
- 3.2. El paradigma Cliente-Servidor
- 3.3. Llamada a Procedimiento (RPC)
- 3.4. Java Remote Method Invocation (RMI)
- 3.5. Servicios Web

Sistemas Concurrentes y Distribuidos., curso 2017-18.

Tema 3. Sistemas basados en paso de mensajes.

Sección 3. Mecanismos de alto nivel en sistemas distribuidos

Subsección 3.1.

Introducción.

Introducción. Llamada a procedimiento.

Los mecanismos vistos hasta ahora (varios tipos de envío/recepción espera selectiva, ...) presentan un bajo nivel de abstracción.

Veremos mecanismos de mayor nivel de abstracción:

- ▶ Llamada a procedimiento remoto (RPC)
- ▶ Invocación remota de métodos (RMI)

Ambos están basados en el método habitual por el cual un **proceso llamador** hace una *llamada a procedimiento*, como sigue:

1. El llamador indica el nombre del procedimiento y los valores de los parámetros.
2. El **proceso llamador ejecuta el código del procedimiento**.
3. Cuando el procedimiento termina, el llamador obtiene los resultados y continúa ejecutando el código tras la llamada

Llamada a procedimientos remotos

En el **modelo de invocación remota** o **llamada a procedimiento remoto** (RPC), se dan los mismos pasos, pero es otro proceso (el **proceso llamado**) el que ejecuta el código del procedimiento:

1. El llamador indica el nombre del procedimiento y los valores de los parámetros.
2. **El proceso llamador se queda bloqueado. El proceso llamado ejecuta el código del procedimiento.**
3. Cuando el procedimiento termina, el llamador obtiene los resultados y continúa ejecutando el código tras la llamada

RPC presenta estas características:

- ▶ El flujo de comunicación es bidireccional (petición-respuesta).
- ▶ Se permite que varios procesos invoquen un procedimiento gestionado por otro proceso (esquema muchos a uno).

Sistemas Concurrentes y Distribuidos., curso 2017-18.

Tema 3. Sistemas basados en paso de mensajes.

Sección 3. Mecanismos de alto nivel en sistemas distribuidos

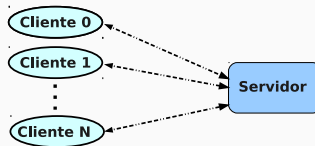
Subsección 3.2.

El paradigma Cliente-Servidor.

El paradigma Cliente-Servidor

Es el paradigma más frecuente en programación distribuida. Hay una relación asimétrica entre dos procesos: **cliente** y **servidor**.

- ▶ **Proceso servidor:** gestiona un recurso (por ejemplo, una base de datos) y ofrece un servicio a otros procesos (clientes) para permitir que puedan acceder al recurso. Puede estar ejecutándose durante un largo periodo de tiempo, pero no hace nada útil mientras espera peticiones de los clientes.
- ▶ **Proceso cliente:** necesita el servicio y envía un mensaje de petición al servidor solicitando algo asociado al servicio proporcionado por el servidor (p.e. una consulta sobre la base de datos).



El paradigma Cliente-Servidor

Es sencillo implementar esquemas de interacción cliente-servidor usando los mecanismos vistos. Para ello usamos en el servidor un **select** que acepta peticiones de cada uno de los clientes:

```
process Cliente[ i : 0.. $n-1$  ] ;
begin
  while true do begin
    s_send( petition, Servidor );
    receive( respuesta, Servidor );
  end
end

process Servidor ;
begin
  while true do
    select
      for i:= 0 to  $n-1$ 
      when condicion[i] receive( petition, Cliente[i] ) do
        respuesta := servicio( petition ) ;
        s_send( respuesta, Cliente[i] ),
      end
    end
  end
end
```

Problemas de la solución

No obstante, se plantean problemas de seguridad en esta solución:

- ▶ Si el servidor falla, el cliente se queda esperando una respuesta que nunca llegará.
- ▶ Si un cliente no invoca el `receive(respuesta, Servidor)` y el servidor realiza `s_send(respuesta, Cliente[j])` síncrono, el servidor quedará bloqueado

Para resolver estos problemas:

- ▶ El par (recepción de petición, envío de respuesta) se debe considerar como una única operación de comunicación bidireccional en el servidor y no como dos operaciones separadas.
- ▶ El mecanismo de **llamada a procedimiento remoto** (RPC) proporciona una solución en esta línea.

Sistemas Concurrentes y Distribuidos., curso 2017-18.

Tema 3. Sistemas basados en paso de mensajes.

Sección 3. Mecanismos de alto nivel en sistemas distribuidos

Subsección 3.3.

Llamada a Procedimiento (RPC).

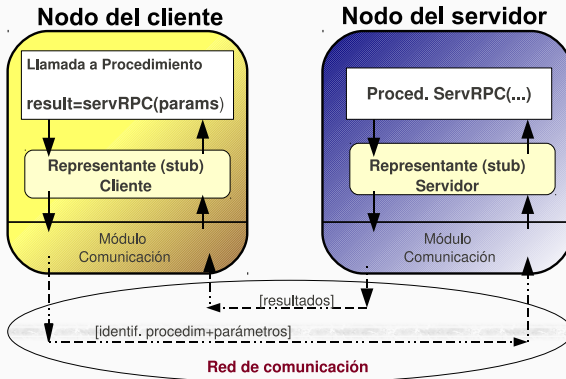
Introducción a RPC

- ▶ **Llamada a procedimiento remoto (Remote Procedure Call):**
Mecanismo de comunicación entre procesos que sigue el esquema cliente-servidor y que permite realizar las comunicaciones como llamadas a procedimientos convencionales (locales).
- ▶ **Diferencia ppal respecto a una llamada a procedimiento local:**
El programa que invoca el procedimiento (cliente) y el procedimiento invocado (que corre en un proceso servidor) pueden pertenecer a máquinas diferentes del sistema distribuido.



Esquema de interacción en una RPC

- ▶ **Representante o delegado (stub):** procedimiento local que gestiona la comunicación en el lado del cliente o del servidor.
- ▶ Los procesos cliente y servidor no se comunican directamente, sino a través de representantes.



LLamada RPC (1): inicio en cliente y envío de parámetros

1. En el nodo cliente se invoca un procedimiento remoto como si se tratara de una llamada a procedimiento local. Esta llamada se traduce en una llamada al *representante* del cliente.
2. El representante del cliente empaqueta todos los datos de la llamada (nombre del procedimiento y parámetros) usando un determinado formato para formar el cuerpo del mensaje a enviar (es muy usual utilizar el protocolo XDR, eXternal Data Representation). Este proceso se suele denominar **marshalling** o **serialización**.
3. El representante el cliente envía el mensaje con la petición de servicio al nodo servidor usando el módulo de comunicación del sistema operativo.
4. El programa del cliente se quedará bloqueado esperando la respuesta.

Llamada RPC (2): ejecución en servidor y envío de resultados

5. En el nodo servidor, el sistema operativo desbloquea al proceso servidor para que se haga cargo de la petición y el mensaje es pasado al representante del servidor.
6. El representante del servidor desempaqueta (*unmarshalling*) los datos del mensaje de petición (identificación del procedimiento y parámetros) y ejecuta una llamada al procedimiento local identificado usando los parámetros obtenidos del mensaje.
7. Una vez finalizada la llamada, el representante del servidor empaqueta los resultados en un mensaje y lo envía al cliente.
8. El sistema operativo del nodo cliente desbloquea al proceso que hizo la llamada para recibir el resultado que es pasado al representante del cliente.
9. El representante del cliente desempaqueta el mensaje y pasa los resultados al invocador del procedimiento.

Representación de datos y paso de parámetros en la RPC

Representación de los datos:

- ▶ En un sistema distribuido los nodos pueden tener diferente hardware y/o sistema operativo (sistema heterogéneo), utilizando diferentes formatos para representar los datos.
- ▶ En estos casos los mensajes se envían usando una representación intermedia y los representantes de cliente y servidor se encargan de las conversiones necesarias.

Paso de parámetros: los parámetros se pueden pasar:

- ▶ **Por valor:** en este caso basta con enviar al representante del servidor los datos aportados por el cliente.
- ▶ **Por referencia:** el objeto referenciado debe enviarse desde el proceso cliente hacia el servidor. Si puede ser modificado en el servidor, debe enviarse desde el servidor al cliente al final (se convierte en **copia de valor-resultado**).

Sistemas Concurrentes y Distribuidos., curso 2017-18.

Tema 3. Sistemas basados en paso de mensajes.

Sección 3. Mecanismos de alto nivel en sistemas distribuidos

Subsección 3.4.

Java Remote Method Invocation (RMI).

El modelo de objetos distribuidos

Invocación de métodos en programas orientados a objetos

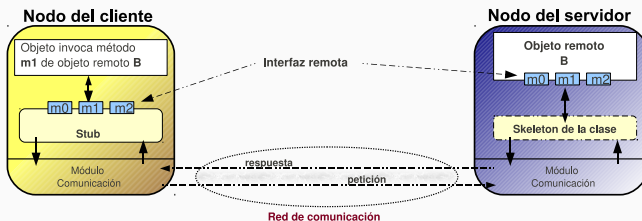
- ▶ Para invocar el método de un objeto hay que dar una referencia del objeto, el método concreto y los argumentos de la llamada.
- ▶ La interfaz de un objeto define sus métodos, argumentos, tipos de valores devueltos y excepciones.

Invocación de métodos remotos (RMI):

- ▶ En un entorno distribuido, un objeto podría invocar métodos de un objeto (**objeto remoto**), localizado en un nodo o proceso diferente del llamador, siguiendo el paradigma cliente-servidor (como ocurre en RPC).
- ▶ Para invocar métodos de un objeto remoto, el proceso llamador debe proporcionar el nombre del método y los parámetros (al igual que en RPC), pero además **debe de identificar el objeto remoto y el proceso o nodo en el que reside**.

Interfaz remota y representantes (1)

- **Interfaz remota:** especifica los métodos del objeto remoto que están accesibles para los demás objetos así como las excepciones derivadas (p.e., que el servidor tarde mucho en responder).
- **Remote Method Invocation (RMI):** acción de invocar un método de la interfaz remota de un objeto remoto. La invocación de un método en una interfaz remota sigue la misma sintaxis que un objeto local.



Interfaz remota y representantes (2)

El cliente y el servidor deben conocer ambos la interfaz de la clase del objeto remoto (los nombres y parámetros de los métodos invocables). Además:

- ▶ **En el cliente:** el proceso llamador usa un objeto llamado **stub**, que es responsable de implementar la comunicación con el servidor.
- ▶ **En el servidor:** se un objeto llamado **skeleton**, que es responsable de esperar la llamada, recibir los parámetros, invocar la implementación del método, obtener los resultados y enviarlos de vuelta.

El *stub* y el *skeleton* permiten al programador ignorar los detalles de la comunicación y empaquetamiento de datos (tanto en el lado del cliente como en la implementación en el servidor).

Referencias remotas

- ▶ Los **stubs** usan la definición de la interfaz remota.
- ▶ Los objetos remotos residen en el nodo servidor y son gestionados por el mismo. Los procesos clientes manejan **referencias remotas** a esos objetos:
 - ▶ Una referencia remota permite al cliente localizar el objeto remoto dentro del sistema distribuido. Para ello, la referencia remota incluye: la dirección IP del servidor, el puerto de escucha y un identificador de objeto.
- ▶ El contenido de la referencia remota no es directamente accesible, sino que es gestionado por el stub y por el enlazador.
- ▶ **Enlazador**: servicio de un sistema distribuido que registra las asignaciones de nombres a referencias remotas.

Sistemas Concurrentes y Distribuidos., curso 2017-18.

Tema 3. Sistemas basados en paso de mensajes.

Sección 3. Mecanismos de alto nivel en sistemas distribuidos

Subsección 3.5.

Servicios Web.

Servicios Web: características

En la actualidad gran parte de la comunicación en Internet ocurre via los llamados **servicios web**

- ▶ Se usan los protocolos HTTP o HTTPS en la capa de aplicación, típicamente sobre TCP/IP en la capa de transporte.
- ▶ Se usa codificación de datos basada en XML o JSON.
- ▶ Es posible usar protocolos complejos (p.ej.SOAP), pero lo más frecuente es usar el método REST, que se caracteriza por:
 - ▶ Los clientes solicitan un recurso o documento especificando su URL
 - ▶ El servidor responde enviando el recurso en su versión actual o notificando un error.
 - ▶ Cada petición es independiente de otras: el servidor, una vez enviada la respuesta, no guarda información de estado de sesión o sobre el estado del cliente (REST es **stateless**)

Llamadas y sincronización

Las peticiones de recursos o documentos pueden hacerse desde

- ▶ una aplicación cualquiera ejecutándose en el cliente, aunque
- ▶ en particular, lo más frecuente es usar una programa Javascript ejecutándose en un navegador en el nodo cliente.

Las peticiones pueden gestionarse de forma

- ▶ **Síncrona:** el proceso cliente espera bloqueado la llegada de las respuesta. Se considera no aceptable en aplicaciones web interactivas (paraliza la interacción entre el usuario y la página web)
- ▶ **Asíncrona:** el proceso cliente envía la petición y continua. Cuando se recibe la respuesta, se ejecuta una función designada por el cliente al hacer la petición. La función recibe la respuesta como un parámetro.

Bibliografía del tema 2.

Para más información más detallada, ejercicios y bibliografía adicional, se puede consultar:

3.1. Mecanismos básicos en sistemas basados en paso de mensajes.

Palma (2003), capítulos 7,8,9. Almeida (2008), capítulo 3. Kumar (2003), capítulo 6.

3.2. Paradigmas de interacción de procesos en programas distribuidos

Andrews (2000), capítulo 9. Almeida (2008), capítulos 5,6.

3.3. Mecanismos de alto nivel en sistemas distribuidos. RPC y RMI.

Palma (2003), capítulo 10. Coulouris (2011), capítulo 5.

Fin de la presentación.