Tema3Sistemas-basados-en-paso-de...

beatrizmartin05

Sistemas Concurrentes y Distribuidos

2º Grado en Ingeniería Informática

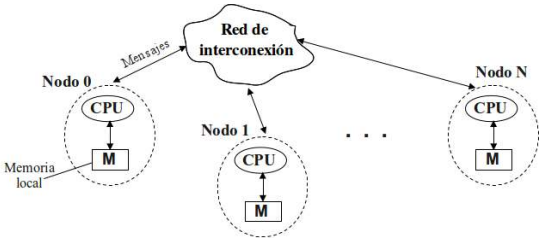
Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación
Universidad de Granada

SISTEMAS BASADOS EN PASO DE MENSAJES

1. MECANISMOS BÁSICOS EN SISTEMAS BASADOS EN PASO DE MENSAJES

1.1. Introducción

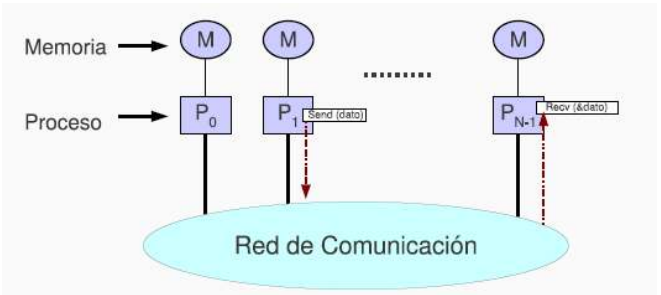
Sistemas Distribuidos: es un conjunto de procesos (en uno o varios ordenadores) que no comparten memoria, pero se transmiten datos a través de una red:



- 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.

1.2. Vista lógica arquitectura y modelo de ejecución

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



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.

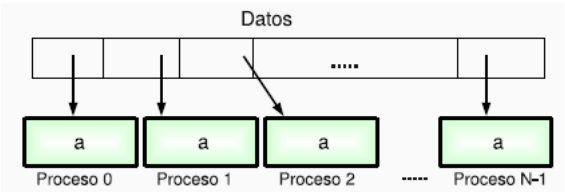
Diseñar un código diferente para cada proceso puede ser complejo. Una solución es el estilo **SPMD** (**Single Program Multiple Data**):

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

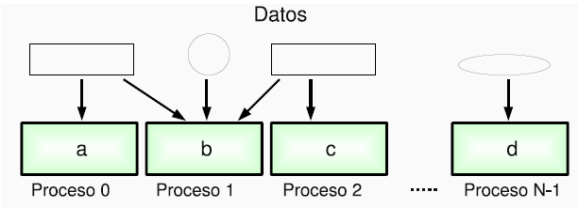
Estructura de un programa de paso de mensajes. MPMD.
Otra opción es usar el estilo **MPMD** (**Multiple Program Multiple Data**):

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

SPDM:

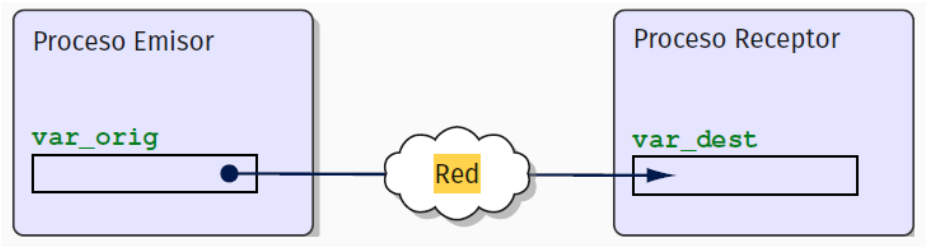


MPMD:



1.3. Primitivas básicas de paso de mensajes

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



1. Se leen de una variable del proceso emisor (var_orig).
2. Se tranfieren a través de alguna red de interconexión.
3. Se escriben en una variable del proceso receptor (var_dest).

El proceso emisor realiza el envío invocando a send, y el proceso receptor realiza la recepción invocando a receive.
Sintaxis:

- **send**(variable_origen, identificador_destino)
- **receive**(variable_destino, identificador_proceso_origen)

El receptor nombra explícitamente al emisor. El emisor puede nombrar un receptor (o bien otros tipos de entidad de destino).

Esquemas de identificación de la comunicación

¿Cómo identifica el emisor al receptor del mensaje y viceversa?
Existen dos posibilidades:

1. **Denominación directa estática**

El emisor identifica explícitamente al receptor y viceversa. Para la identificación se usan identificadores de procesos.

```
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.

Inconvenientes

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

2. Denominación indirecta

Los mensajes se depositan en almacenes intermedios que son accesibles desde todos los procesos (buzones).

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

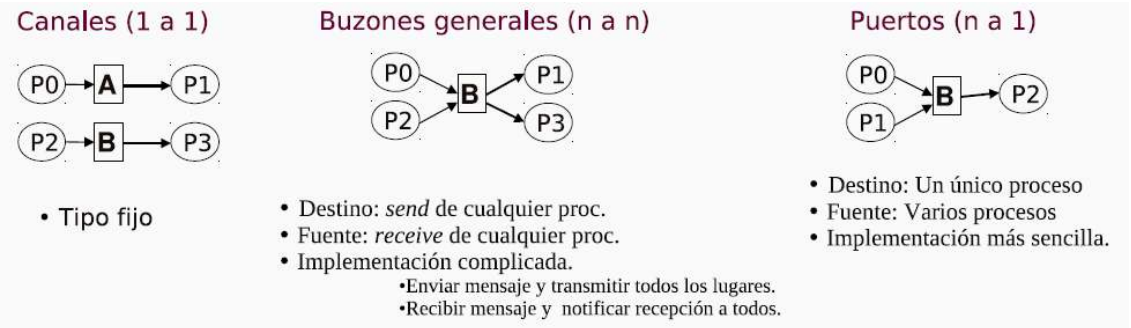
```
process P0 ;
   var var_orig : integer ;
begin
   var_orig := Produce();
   send( var_orig, buzon );
end
```

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

Ventajas:

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

Existen tres **tipos de buzones**:



Declaración estática versus dinámica

Los identificadores de proceso suelen ser valores enteros, cada uno de ellos está unívocamente asociado a un proceso del programa concurrente.

Se pueden gestionar:

- **Estáticamente**: en el código fuente se asocian explícitamente los números enteros a los procesos (es muy eficiente en tiempo pero los cambios en la estructura del programa requieren adaptar el código fuente y recompilarlo).
- **Dinámicamente**: el identificador numérico de cada proceso se calcula en tiempo de ejecución cuando es necesario (es menos eficiente en tiempo pero el código puede seguir siendo válido aunque cambie el número de procesos de cada tipo).

Secuencia de operaciones en el lado del emisor

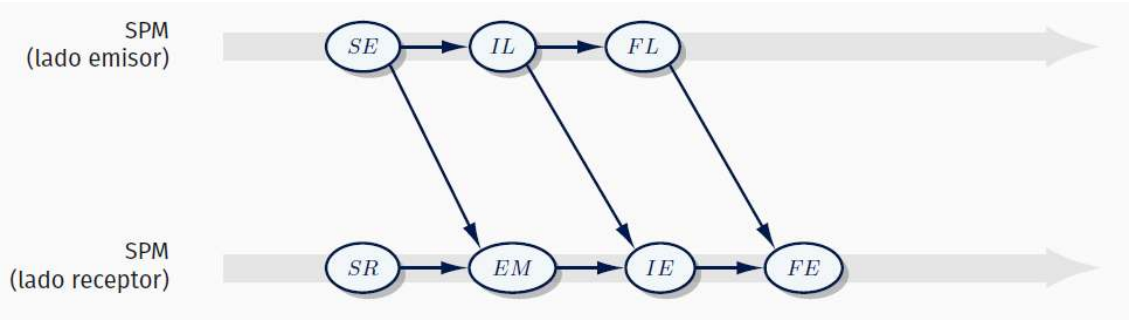
1. Fin del registro de la **solicitud de envío (SE)**: después de iniciada la llamada a send, el SPM registra los identificadores de ambos procesos, y la dirección y tamaño de la variable de origen.
2. **Inicio de lectura (IL)**: el SPM lee el primer byte de todos los que forman el valor de la variable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

Secuencia de operaciones en el lado del receptor

1. Fin del registro de la **solicitud de recepción (SR)**: después de iniciado receive, el SPM registra los idents. 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, los instantes del SPM en ambos lados están relacionados por el siguiente grafo de sincronización:



- Para un mensaje cualquiera, transcurrirá un determinado intervalo de tiempo finito entre la solicitud de envío (SE) y el fin de la escritura (FE).
- Durante ese intervalo de tiempo, se dice que el correspondiente mensaje es un mensaje en tránsito.

El almacén temporal de los mensajes en tránsito

En un programa concurrente, el SPM necesitará usar memoria para almacenar datos y metadatos de todos los mensajes en tránsito en un momento dado.
A esa memoria la llamamos **almacén temporal de datos**.
La cantidad de memoria necesaria para el almacén dependerá de diversos detalles (tamaño y número de los mensajes en tránsito, velocidad de la transmisión de datos…).
Dicha memoria puede estar ubicada en el nodo emisor y/o en el receptor y/o en nodos intermedios de la red, si los hay.

Seguridad de las operaciones de paso de mensajes

Las operaciones podrían ser **no 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.
  - **Recepción insegura**: ocurre cuando se puede acceder a var_dest entre SR y FE.

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:
  1. Envío y recepción síncronos.
  2. Envío asíncrono seguro.

1/6

Este número es indicativo del riesgo del producto, siendo 1/6 indicativo de menor riesgo y 6/6 de mayor riesgo.

ING BANK NV se encuentra adherido al Sistema de Garantía de Depósitos Holandés con una garantía de hasta 100.000 euros por depositante. Consulta más información en ing.es

Que te den **10 € para gastar** es una fantasía. ING lo hace realidad.

Abre la **Cuenta NoCuenta** con el código **WUOLAH10**, haz tu primer pago y llévate 10 €.

Quiero el cash

Consulta condiciones **aquí**

do your thing

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.
- Existen sentencias adicionales para comprobar el estado de la operación.

Operaciones síncronas

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

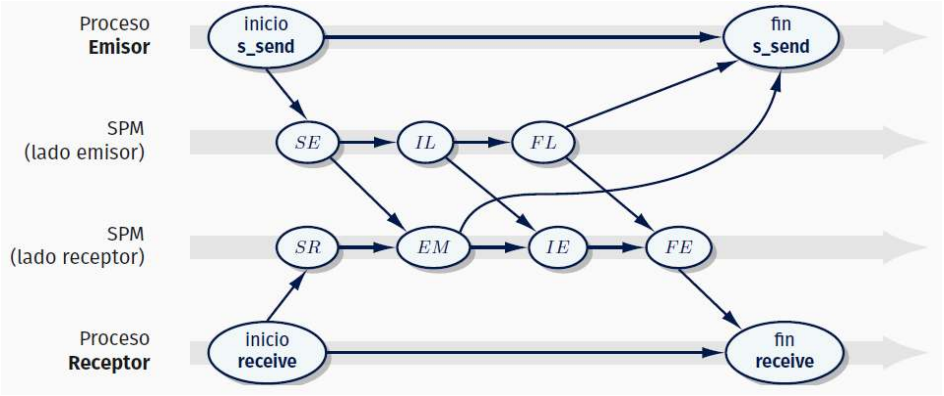
Realiza el envío de los datos y espera bloqueado hasta que los datos hayan terminado de leerse en el emisor y se haya iniciado y emparejado un receive en el receptor.

```
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.

El grafo de sincronización para s_send con receive es este:

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



Las operaciones síncronas exige una **cita entre emisor y receptor**: La operación s_send no devuelve el control hasta que se alcance el receive correspondiente en el receptor. Ejemplo → comunicación telefónica y chat.

Las **desventajas** de las operaciones síncronas:

1. Fácil de implementar, pero poco flexible.
2. Sobrecarga por espera ociosa: adecuado sólo cuando send/receive se inician aprox. al mismo tiempo.
3. Interbloqueo: es necesario alternar llamadas en intercambios.

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 );
```

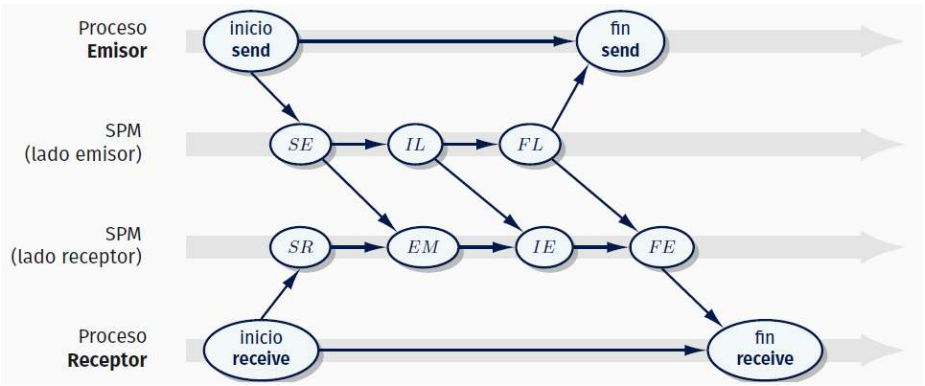
Envío asíncrono seguro

```
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.

El grafo de sincronización para send con receive es este:

- 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.



Las **ventajas** del envío asíncrono seguro:

1. El uso de send lleva, en general, a menores tiempos de espera bloqueada.
2. Usar send es generalmente más eficiente en tiempo y preferible cuando el emisor no tiene que esperar la recepción.

Las **desventajas** del envío asíncrono seguro:

1. Hay que tener en cuenta que send requiere memoria para almacenamiento temporal, el cual, puede crecer mucho o incluso indefinidamente.

2. 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.

Operaciones inseguras

Las operaciones seguras son menos eficientes en tiempo(por las esperas bloqueadas) y en memoria(por el almacenamiento temporal).
Su alternativa son las operaciones de inicio de envío o recepción.

- Devuelven el control antes de que sea seguro modificar/acceder a los datos.
- Deben existir sentencias de chequeo de estado (indican si los datos pueden alterarse o acceder a ellos sin comprometer la seguridad).

Paso de mensajes asíncrono inseguro:

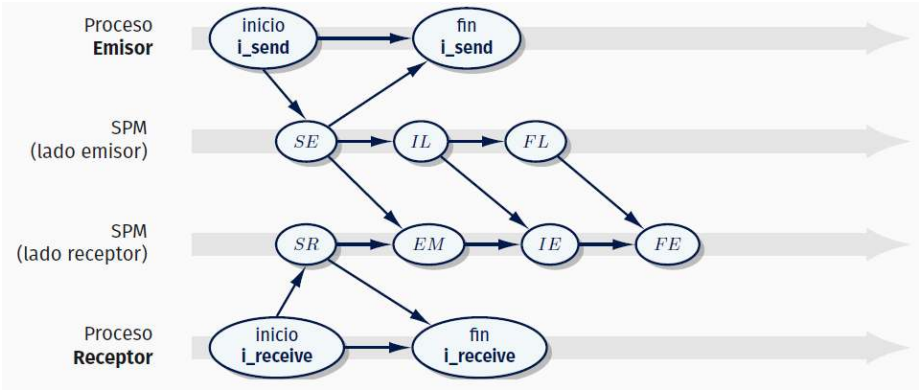
```
i_send( variable_orig , id_proc_receptor , var_resguardo ) ;
```

Indica al SPM que comience una operación de envío al receptor. Var_resguardo permite consultar después el estado del envío.

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

Indica al SPM que inicie una recepción de un mensaje del emisor. Var_resguardo permite consultar después el estado de la recepción.

El grafo de sincronización 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 recepción.

Esperar el final de operaciones asíncronas:
Cuando un proceso hace i_send o i_receive puede continuar trabajando hasta el momento en el que deba esperar a que termine la operación asíncrona, se disponen de estos dos procedimientos:

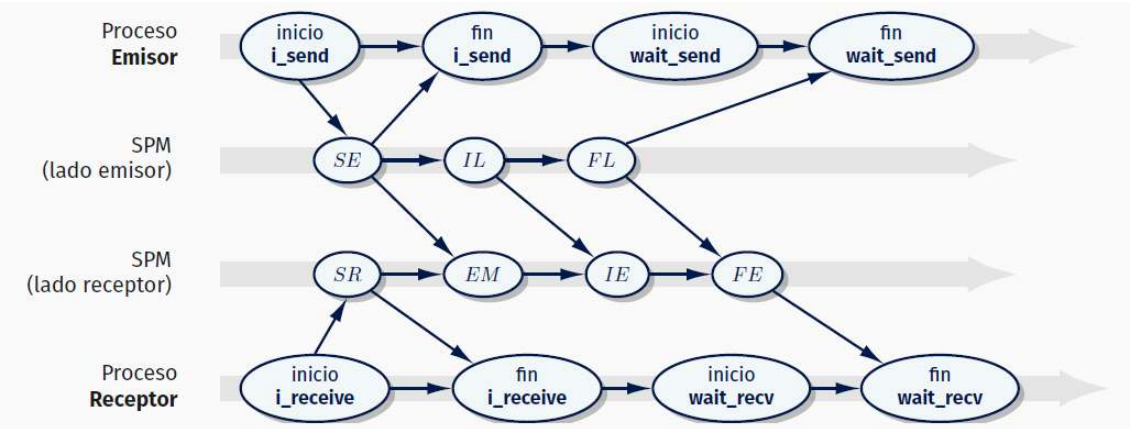
```
wait_send( var_resguardo ) ;
```

Se invoca por el proceso emisor, que se 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).

Por tanto, cuando se usan las operaciones asíncronas, el inicio y fin de las operaciones de espera con wait_send y wait_recv debe seguir a las correspondientes operaciones de envío o recepción:



Utilidad: estas operaciones permiten a los procesos emisor y receptor hacer trabajo útil concurrentemente con la operación de envío o recepción.

Ejemplo: Productor-Consumidor con paso asíncrono inseguro

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

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

Se pueden usar dos funciones de comprobación de estado de una transmisión de un mensaje. No suponen bloqueo, sólo una consulta:

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 (si no, devuelve false).

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, (si no, devuelve false).

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:

<u>Espera bloqueada de múltiples procesos</u>

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 cuál 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 qué emisor queremos

esperar recibir en primer lugar, y este emisor no coincide necesariamente con el emisor del primer mensaje que realmente podría recibirse.

1.4. Espera selectiva

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:

Productor-consumidor distribuido

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

- Produce y Consume pueden tardar tiempos distintos.
- Si usaramos 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.

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):

Síntesis general y comportamiento

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

- Cada bloque que comienza en when se llama alternativa.
- El texto entre when y do se denomina guarda de dicha alternativa, e 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

Una guarda es una **condición lógica** que determina si una acción asociada puede ser ejecutada en un momento dado.

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

- La expresión lógica puede omitirse. En ese caso, la sintaxis sería:

que es equivalente a:

- La sentencia receive puede no aparecer, siendo la sintaxis:

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:

1. La condición de la guarda se evalúa en ese momento a true.
2. 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 (match) con el receive.

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

1. La condición de la guarda se evalúa en ese momento a true.
2. Tiene una sentencia de entrada 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

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
- 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.

Una vez se ha intentando seleccionar la guarda:

- Si **no se ha podido**, no se hace nada 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.
     A continuación 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
```

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:

```

```

Es equivalente a:

```

```

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:

2. PARADIGMA DE INTERACCIÓN DE PROCESOS EN PROGRAMAS DISTRIBUIDOS

2.1. Introducción
Un paradigma de interacción define un esquema de interacción entre procesos y una estructura de control que aparece en múltiples programas.
Veremos los siguientes paradigmas de interacción:

1. Maestro-Esclavo.
2. Iteración síncrona.
3. Encauzamiento (pipelining).
4. Cliente-Servidor.

2.2. 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 subtareas (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

Procesos Maestro y Esclavo

2.3. 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.

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

Transformación iterativa: vista general

Cada columna representa un proceso, cada fila una etapa de cálculo

Transformación iterativa: vista de cada bloque

Cada celda de cada bloque en una etapa se calcula a partir de tres celdas de un bloque (o dos bloques adyacentes) de la etapa anterior:

Proceso coordinador

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

2.4. 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.

3. MECANISMOS DE ALTO NIVEL EN SISTEMAS DISTRIBUIDOS

3.1. Introducción

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 de ejecución de procedimientos y funciones por el cual un proceso llamador hace una llamada a procedimiento:

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.

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).

3.2. 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 (ej. 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 (ej. consulta en base de datos).

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:

1/6

Este número es indicativo del riesgo del producto, siendo 1/6 indicativo de menor riesgo y 6/6 de mayor riesgo.

ING BANK NV se encuentra adherido al Sistema de Garantía de Depósitos Holandés con una garantía de hasta 100.000 euros por depositante. Consulta más información en ing.es

Que te den **10 € para gastar** es una fantasía. ING lo hace realidad.

Abre la **Cuenta NoCuenta** con el código **WUOLAH10**, haz tu primer pago y llévate 10 €.

Quiero el cash

Consulta condiciones **aquí**

do your thing

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.

3.3. Llamada a procedimiento (RPC)

Mecanismo de comunicación entre procesos que sigue el esquema cliente-servidor y que permite realizar las comunicaciones como llamadas a procedimientos convencionales (locales).

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

- 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. Este proceso se suele denominar marshalling o serialización.
    3. El representante del 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.
- 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 (local) al procedimiento 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 (programa cliente).

Representación de datos en la RPC

- En un sistema distribuido, los nodos pueden tener diferente hardware y/o sistema operativo, 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 en la RPC

- **Por valor** (mucho más sencillo): 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 finalizar el procedimiento (se convierte en copia de valor-resultado).

3.4. Java Remote Method Invocation (RMI)

- **Interfaz remota**: especifica los métodos del objeto remoto que están accesibles para los demás objetos, así como las excepciones derivadas.
- **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 la invocación de un método de un objeto local.

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 utiliza 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. En algunos lenguajes de programación ya no se requiere el uso del skeleton.

El stub y el skeleton permiten al programador ignorar los detalles de la comunicación y empaquetamiento de datos.

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.

3.5. Servicios web

En la actualidad, gran parte de la comunicación en Internet ocurre vía los llamados servicios web

Características

- 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.e., SOAP), pero lo más frecuente es usar el método REST, que se caracteriza por:
    1. Los clientes solicitan un recurso o documento especificando su URL.
    2. El servidor responde enviando el recurso en su versión actual o notificando un error.
    3. 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.

Llamadas y sincronización

Las peticiones de recursos o documentos pueden hacerse desde:

- Una **aplicación cualquiera** ejecutándose en el cliente.
- Un **programa Javascript** ejecutándose en un navegador en el nodo cliente (lo más frecuente).

Las peticiones pueden gestionarse de forma:

- **Síncrona**: el proceso cliente espera bloqueado la llegada de las respuestas. Se considera no aceptable en aplicaciones web interactivas.
- **Asíncrona**: el proceso cliente envía la petición y continúa: 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.