TEMA-3-SISTEMAS-BASADOS-EN-PASO-...

mrg23

Sistemas Concurrentes y Distribuidos

2º Grado en Ingeniería Informática

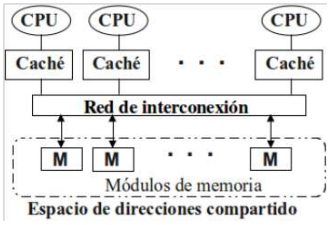
Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación
Universidad de Granada

TEMA 3: SISTEMAS BASADOS EN PASOS DE MENSAJES

1. MECANISMOS BÁSICOS EN SISTEMAS BASADOS EN PASO DE MENSAJES

a. INTRODUCCIÓN

MEMORIA COMPARTIDA



Para programar sistemas multiprocesador de memoria compartida:

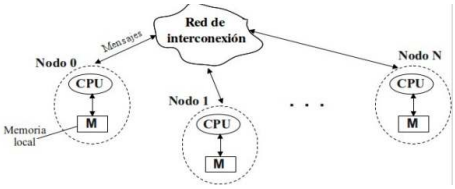
- **Más fácil programación** (variables compartidas): se usan mecanismos como cerrojos, semáforos y monitores.
- **Implementación más costosa y escalabilidad hardware limitada**.El acceso a memoria común supone un cuello de botella.

MEMORIA DISTRIBUIDA

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

- **Facilita la distribución** de datos y recursos.
- **Soluciona** el problema de la **escalabilidad** y **elevado coste**.
- **Mayor dificultad de programación**: no hay direcciones de memoria

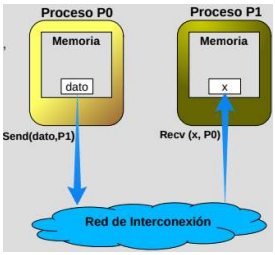
comunes y mecanismos como los monitores son inviables.



NECESIDAD DE UNA NOTACIÓN DE PROGRAMACIÓN DISTRIBUIDA

Lenguajes tradicionales (memoria común)

- Asignación: cambio estado interno máquina.
- Estructuración: secuencia, repetic., procedim., etc.



Extra añadido: Envío/Recepción ⇒ Afectan al entorno externo.

- Tan importantes como la asignación.
- Permiten comunicar procesos en ejecución paralela.

Paso de mensajes

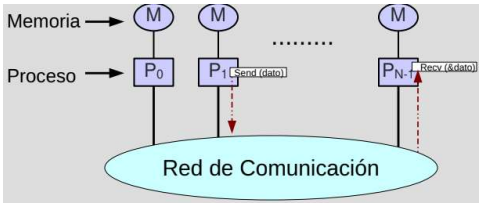
- Abstracción: oculta red de interconexión.
- Portabilidad: Implementable eficientemente en cualquier arquitectura (mem. Compartida o distribuida).
- No requiere mecanismos para asegurar EM.

b. 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.

→ En un procesador pueden residir físicamente varios procesos aunque por eficiencia, frecuentemente se aloja 1 proceso en cada procesador.

→ Interacción requiere cooperación entre 2 procesos: Propietario datos (emisor) debe intervenir aunque no haya conexión lógica con el evento tratado en Receptor.

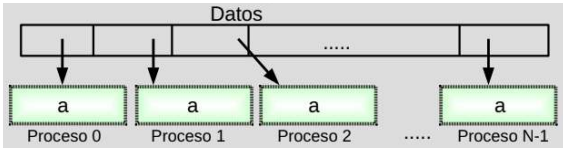


ESTRUCTURA DE UN PROGRAMA DE PASO DE MENSAJES. SPMD

Diseñar un código diferente para cada proceso → Complejo.

- Solución: Estilo SPMD (Single Program Multiple Data):
- Todos los procesos ejecutan el mismo código fuente.
- Cada proceso puede procesar datos distintos y/o ejecutar distintos flujos de control.

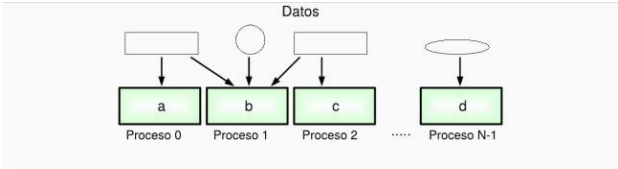
```
process Proceso[ n_proc : 0..1 ];
   var dato : integer ;
begin
   if n_proc == 0 then begin {si soy 0}
      dato := Produce();
      send( dato, Proceso[1]);
   end else begin {si soy 1}
      receive( dato, Proceso[0] );
      Consume( dato );
   end
end
```



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 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
```

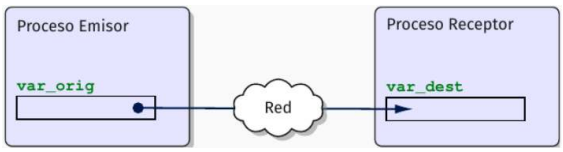
TRANSFERENCIA DE MENSAJES

Paso de un mensaje entre 2 procesos: transferencia de secuencia finita de bytes.

- Leídos de variable en Emisor (var_orig).
- Se transfieren a través de una red de interconexión.
- Se escriben en variable en Receptor (var_dest).

Sincronización: Bytes acaban de recibirse después de iniciar envío.

Efecto final: var_dest := var_orig (var_dest y var_orig son del mismo tipo).



c. PRIMITIVAS BÁSICAS DE PASO DE MENSAJES

Proceso emisor realiza envío invocando send, y Proc. receptor realiza recepción invocando receive.
Sintaxis:

- send (variable_origen, identificador_destino)
- receive(variable_destino, identificador_origen)

Ejemplo. Transferencia de un valor entero: cada proceso nombra explícitamente al otro, indicando nombre proceso como identificador.

```
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? Dos posibilidades:

Denominación directa estática

- Emisor identifica explícitamente al receptor y viceversa, mediante identificadores de procesos (típicamente enteros asociados a los procesos en tiempo de compilación).

Denominación indirecta

- Los mensajes se depositan en almacenes intermedios accesibles desde todos los procesos (buzones).
- Emisor nombra buzón donde envía. Receptor nombra buzón donde recibirá.

DE NOMINACIÓN DIRECTA ESTÁTICA

Ventajas

- Sin retardo para establecer identificación (P0 y P1 se traducen en enteros)

Inconvenientes

- Cambios en identificación ⇒ recompilar el código.
- Sólo comunicación 1-1.


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

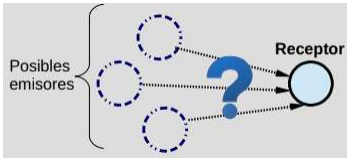
```
process P1 ;
  var var_dest : integer ;
begin
  receive( var_dest, P0 );
  Consume( var_dest );
end
```

DE NOMINACIÓN DIRECTA CON IDENTIFICACIÓN ASIMÉTRICA

Existen esquemas asimétricos: Emisor identifica al Receptor, pero Receptor no indica Emisor.

- Receptor indica que acepta recibir el mensaje de cualquier posible Emisor.

Receive (var_destino, ANY)



Posibilidades para conocer identificación del Emisor tras recibir mensaje:

- Identificador forma parte de los metadatos del mensaje.
- Identificador puede ser un parámetro de salida de receive.

Otra alternativa: Especificar que el emisor debe pertenecer a un subconjunto de todos los posibles.

DENOMINACIÓN DIRECTA

Emisor y el receptor identifican un buzón o canal intermedio a través del cual se transmiten los mensajes.

- Mayor flexibilidad: permite comunicaciones entre múltiples receptores y emisores.

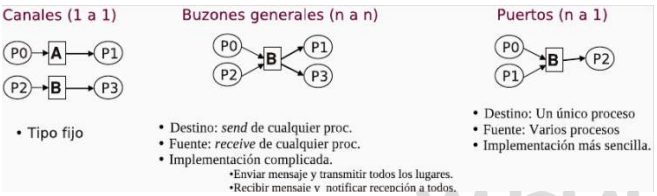

```
var buzon : channel of integer; { es accesible por ambos procesos }
```

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

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

Tres tipos de buzones: canales (uno a uno), puertos (muchos a uno) y buzones generales (muchos a muchos).

- Un mensaje enviado a un buzón general permanece en el buzón hasta que sea leído por todos los receptores potenciales (envío = difusión a todos).



DECLARACIÓN ESTÁTICA vs DINÁMICA

Los identificadores de proceso suelen ser valores enteros biunívocamente asociados a procesos del programa. Se pueden gestionar:

- Estáticamente: en código fuente se fija un entero a cada proceso.
  - – Ventaja: muy eficiente en tiempo.
  - – Inconveniente: Rigidez. cambios en la estructura del programa (num. procesos de cada tipo) requiere adaptar código fuente y recompilarlo.

- Dinámicamente: Identificadores de procesos se fijan en tiempo de ejecución.
  - – Inconveniente: menos eficiente en tiempo.
  - – Ventaja: Flexibilidad. Código sigue siendo válido aunque cambie la estructura (no hay que recompilar).

COMPORTAMIENTO DE LAS OPERACIONES DE PASO DE MENSAJES

```
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
```

Comportamiento Esperado: valor recibido en var_dest será el que se tenía var_orig (100) justo antes de invocar send.

- Se garantiza siempre ⇒ Comportamiento Seguro (programa de paso de mensajes seguro).
- Si pudiera imprimirse 0 ó 1 en lugar de 100 ⇒ Comportamiento NO Seguro.
  No deseable, aunque existen situaciones en las que puede interesar usar operaciones que no garantizan seguridad (usadas adecuadamente).

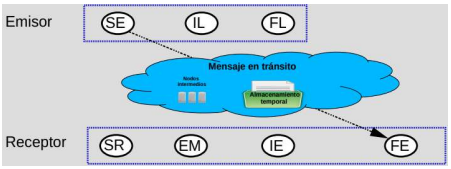
INSTANTES CRÍTICOS DE EMISOR Y RECEPTOR

El Sistema de Paso de mensajes (SPM) debe realizar una serie de pasos en emisor y receptor para transmitir el mensaje:

send(var_orig, Receptor)

receive(var_dest, Emisor)

INSTANTES EN EL EMISOR Y RECEPTOR



SEGURIDAD DE LAS OPERACIONES DE PASO DE MENSAJES

```
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
```

- **Operación de envío-recepción segura**: Se garantiza que el valor de var_orig antes del envío coincidirá con el valor de var_dest tras la recepción.
- **Operaciones inseguras**
  - – Envío inseguro: Es posible modificar el valor de var_orig entre SE y FL (podría enviarse un valor distinto del registrado en SE).
  - – Recepción insegura: Es posible acceder a var_dest entre SR y FE.

TIPOS DE OPERACIONES DE PASO DE MENSAJES

→ **Operaciones seguras**

- Devuelven el control cuando se garantiza la seguridad: send no espera recepción, receive sí espera.
- Dos mecanismos:
  - Envío y recepción síncronos.
  - Envío asíncrono seguro.

→ **Operaciones inseguras**

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

OPERACIONES SÍNCRONAS. COMPORTAMIENTO

s_send (variable_origen, ident_proceso_receptor);

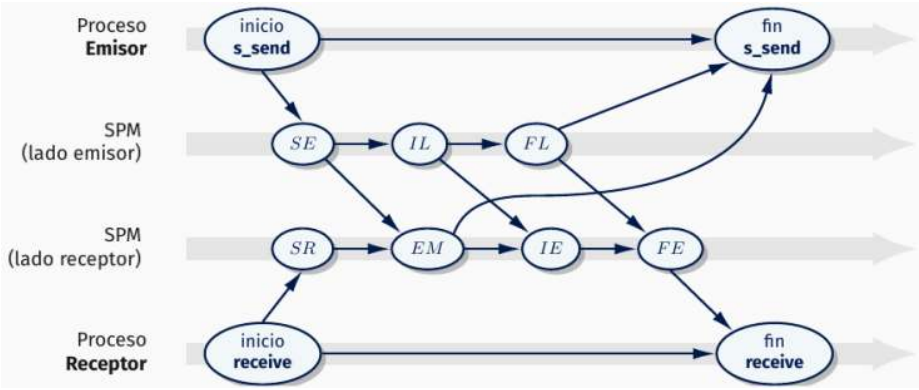
- Espera a que los datos se hayan leído en emisor y se produzca emparejamiento con receive en receptor.
- No termina antes de FL y EM. Posteriormente, se transferirán los datos.

receive(variable_destino , ident_proceso_emisor);

- Espera hasta que emisor emita mensaje hacia receptor y que terminen de escribirse los datos en variable de destino.
- No termina antes de que ocurra FE.

OPERACIONES SÍNCRONAS. GRAFO DE DEPENDENCIA

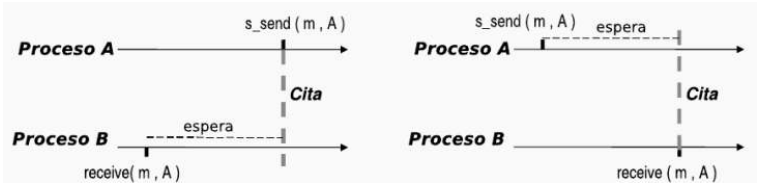
Uso de s_send en conjunción con receive.



OPERACIONES SÍNCRONAS. CITA

Cuando se usa s_send en conjunción con receive:

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



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 €.

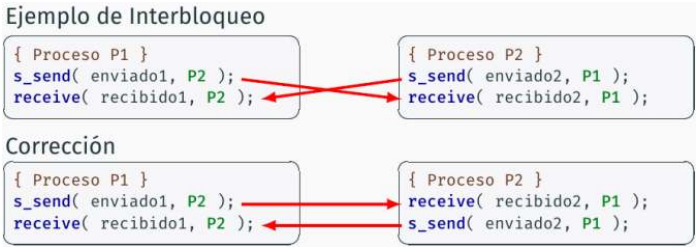
Quiero el cash

Consulta condiciones **aquí**

do your thing

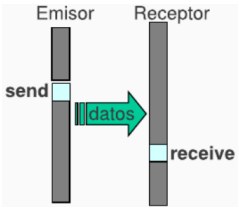
OPERACIONES SÍNCRONAS. DESVENTAJAS

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



ENVÍO ASÍNCRONO SEGURO

send (variable_origen , ident_proceso_receptor);



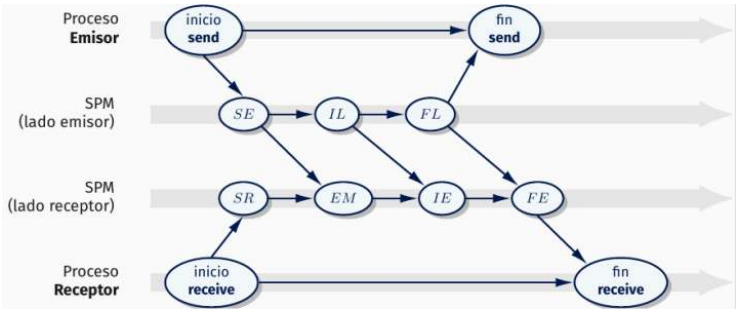
- Inicia envío datos y espera bloqueado hasta que se copien datos a lugar seguro. Tras copiar datos, devuelve control.
- Devuelve control después de FL.
- Se suele usar junto con receive.

Sincronización Emisor-Receptor:

- Fin send no depende actividad receptor. Puede ocurrir antes, durante o después recepción.
- Fin de receive ocurre después inicio send.

ENVÍO ASÍNCRONO SEGURO. GRAFO DE DEPENDENCIA

Uso de send en conjunción con receive.



ENVÍO ASÍNCRONO SEGURO. VALORACIÓN

Ventaja:

- Menores tiempos de espera bloqueada que s_send.
- Generalmente más eficiente en tiempo y preferible cuando emisor no tiene que esperar la recepción.

Posible inconveniente:

- send requiere memoria para almacenamiento temporal, que podría crecer mucho.
- SPM puede tener que retrasar IL en emisor, cuando detecta que no hay memoria suficiente para copia y no se ha producido aún emparejamiento.

ENVÍO ASÍNCRONO SEGURO. MEMORIA TEMPORAL CRECIENTE

Si Produce tarda menos que Consume, y ocurre:

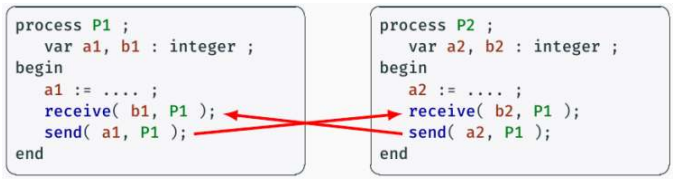
- Tamaño variable de tipo T es grande.
- Valor de N es grande.

Memoria para almacenamiento temporal puede agotarse ⇒ Comportamiento síncrono en send.

```
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
```

SITUACIÓN DE INTERBLOQUEO CON SEND/RECEIVE



OPERACIONES INSEGURAS

Operaciones Seguras: menos eficientes

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

Alternativa: Operaciones de inicio de envío o recepción:

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

PASO ASÍNCRONO INSEGURO. OPERACIONES

i_send (variable_origen , ident_proceso_receptor, var_resguardo) ;

Indica al SPM que comience una operación de envío:

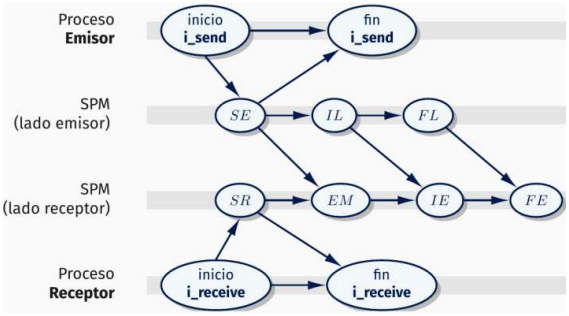
- Registra solicitud de envío (SE) y acaba.
- No espera a FL.
- var_resguardo permite consultar el estado después.

i_receive(variable_destino , ident_proceso_emisor, var_resguardo) ;

Indica al SPM que se inicie una recepción:

- Se registra solicitud de recepción (SR) y acaba.
- No espera a FE.
- var_resguardo permite consultar el estado después.

PASO ASÍNCRONO INSEGURO. GRAFO DEPENDENCIA



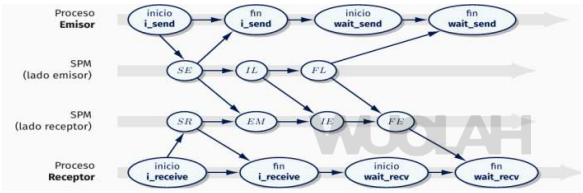
ESPERANDO HASTA SEGURIDAD EN i_send/ i_receive

wait_send (var_resguardo) ;

Bloquea emisor hasta que envío asociado a var_resguardo ha llegado a instante FL (es seguro volver a usar la variable origen).

wait_recv (var_resguardo) ;

Bloquea receptor hasta recepción asociada a var_resguardo ha llegado a FE (se han recibido los datos).



OPERACIONES ASÍNCRONAS. UTILIDAD

Permiten a procs. emisor y receptor hacer trabajo útil concurrentemente con el envio 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_recv( resg );
  print( b );
end
```

CHEQUEANDO SEGURIDAD EN i_send / i_receive

test_send (var_resguardo) ;

Función lógica que se invoca en emisor. Devuelve true si envío asociado a var_resguardo ha llegado a FL.

test_recv (var_resguardo) ;

Función lógica que se invoca en receptor. Devuelve true si recepción asociada a var_resguardo ha llegado a FE.

Ejemplo: Trabajo útil puede descomponerse en trozos.

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

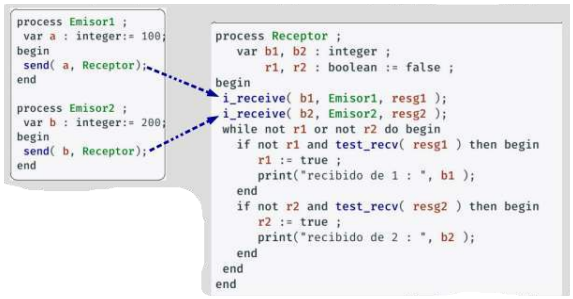
```
process Receptor ;
   var b : integer ;
begin
 i_receive( b, Emisor, resg );
 while not test_recv( resg ) do begin
   {trabajo útil: no accede a b}
   trabajo_util_receptor();
 end
 print( b );
end
```

RECEPCIÓN SIMULTÁNEA DE VARIOS EMISORES

Receptor comprueba continuamente si se ha recibido mensaje de uno cualquiera de 2 emisores, y espera (con espera ocupada) hasta que se ha recibido de ambos:

Limitaciones con las primitivas vistas:

- No es posible hacer esto usando espera bloqueada.
- Se debe seleccionar de qué emisor queremos esperar recibir primero (puede no coincidir con el del primer mensaje que llegue).



```
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
```

d. ESPERA SELECTIVA

Espera selectiva: Operación que permite espera bloqueada de múltiples emisores. Se usan palabras clave select y when.

Implementación del ejemplo visto anteriormente con espera selectiva:

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

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

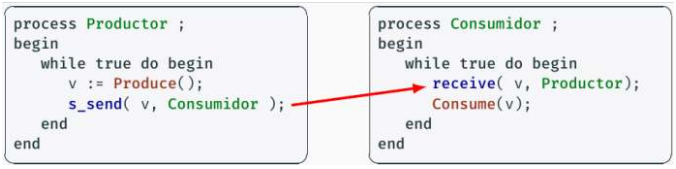
```
process Receptor ;
   var b1, b2 : integer ;
       r1, r2 : boolean := false ;
begin
 while not r1 or not r2 do begin

   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

Solución ingenua:



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

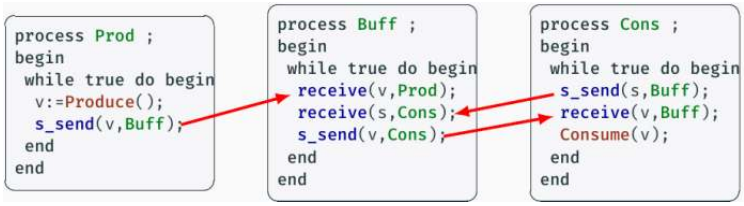
```
process Consumidor ;
begin
   while true do begin
      receive( v, Productor);
      Consume(v);
   end
end
```

Produce y Consume pueden tardar tiempos distintos:

- Si usáramos send, el SPM ⇒ memoria para almacenamiento temporal cuya cantidad podría crecer, quizás indefinidamente.
- Problema: Al usar s_send se pueden introducir esperas largas (bajo aprovechamiento de las CPUs disponibles).

PRODUCTOR-CONSUMIDOR CON PROCESO INTERMEDIO

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



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

```
process Buff ;
begin
 while true do begin
  receive(v,Prod);
  receive(s,Cons);
  s_send(v,Cons);
 end
end
```

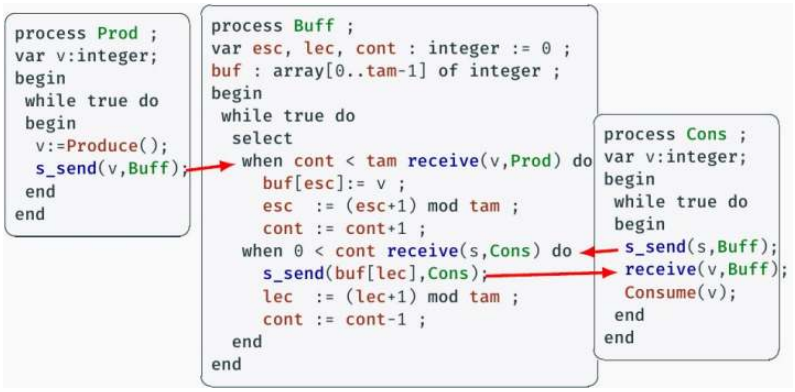
```
process Cons ;
begin
 while true do begin
  s_send(s,Buff);
  receive(v,Buff);
  Consume(v);
 end
end
```

Problema: Proceso intermedio se bloquea por turnos para esperar bien a emisor, bien a receptor, pero nunca a ambos simultáneamente. Persisten esperas excesivas.

ESPERA SELECTIVA Y BUFFER FIFO INTERMEDIO

Solución: usamos espera selectiva en proceso intermedio que puede esperar a ambos procesos a la vez.

- Para reducir esperas, usamos array de datos pendientes lectura (FIFO):

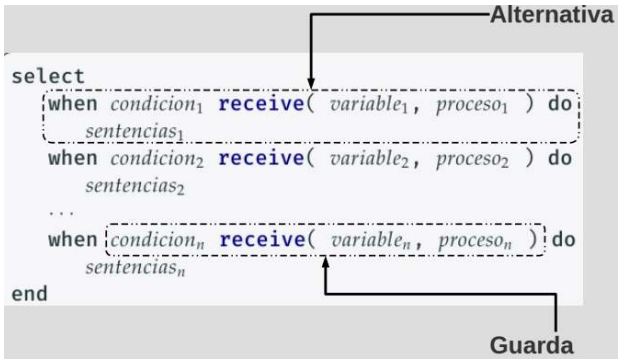


```
process Prod ;
var v:integer;
begin
 while true do
 begin
  v:=Produce();
  s_send(v,Buff);
 end
end
```

```
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
```

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

ESPERA SELECTIVA. SINTAXIS



SINTAXIS DE LAS GUARDAS. GUARDAS SIMPLIFICADAS

- La expresión lógica de una guarda puede omitirse:

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

- Guarda sin sentencia de Entrada: La sentencia receive también puede omitirse.

```
when condicion do
    sentencias
```

GUARDAS EJECUTABLES. EVALUACIÓN DE LAS GUARDAS

Guarda Ejecutable en proceso P cuando:
- Condición se evalúa a true.
- Si tiene receive, el emisor ya ha iniciado send hacia P, que casa con receive.

Guarda Potencialmente Ejecutable cuando:
- Condición se evalúa a true.
- Tiene Receive y nombra emisor que no ha iniciado send hacia P.

Guarda NO ejecutable: condición a false.

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

EJECUCIÓN SELECT. SELECCIÓN ALTERNATIVA

Se selecciona una alternativa entre aquellas con condición true:
- Hay guardas ejecutables con sentencia de entrada: se selecciona aquella cuyo send se inició antes (esto garantiza a veces la equidad).
- Solo guardas ejecutables, pero sin sentencia de entrada: selecciona aleatoriamente una cualquiera.
- Sin guardas ejecutables, pero sí potencialmente ejecutables: se espera (bloqueado) a que alguno de los procesos nombrados en esas guardas inicie send, en ese momento acaba la espera y selecciona la guarda con ese receive.

Sin guardas viables: no selecciona ninguna guarda.

```
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
```

EJECUCIÓN SELECT: EJECUCIÓN ALTERNATIVA

- Si no se ha podido seleccionar guarda, finaliza ejecución select (no hay guardas viables).
- Si se ha podido, 2 pasos en secuencia:

  1) Si guarda con sentencia entrada, se ejecuta receive (habrá send iniciado), y se recibe mensaje.
  2) Se ejecuta sentencia asociada alternativa y finaliza select.

  Select conlleva potencialmente esperas ⇒ Riesgo esperas indefinidas (interbloqueo).

SELECT CON GUARDAS INDEXADAS

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

Todos los componentes (*condicion, mensaje, proceso, sentencias*) pueden contener referencias a la variable índice.

Equivale 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 }
```

Ejemplos de Select con guardas indexadas

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

=

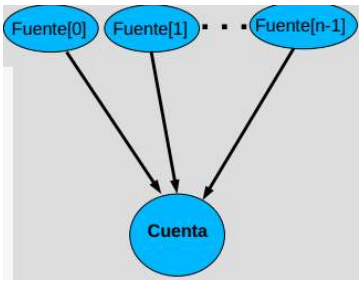
```
when suma[0] < 1000 receive( numero, fuente[0] ) do
   suma[0] := suma[0] + numero ;
when suma[1] < 1000 receive( numero, fuente[1] ) do
   suma[1] := suma[1] + numero ;
...
when suma[n-1] < 1000 receive( numero, fuente[n-1] ) do
   suma[n-1] := suma[n-1] + numero ;
```

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 Fuente 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 ∀i suma[i] ≥ 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
```



2. PARADIGMAS DE INTERACCIÓN DE PROCESOS EN PROGRAMAS DISTRIBUIDOS

a. INTRODUCCIÓN

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

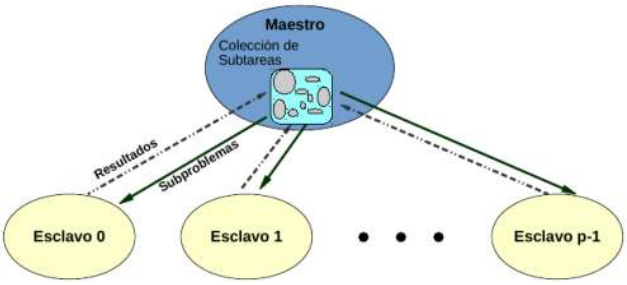
- Se utilizan repetidamente para desarrollar muchos programas distribuidos.
- Utilizaremos lo siguientes paradigmas de interacción:
  - Maestro-esclavo.
  - Iteración síncrona.
  - Segmentación (pipelining).
- Se usan principalmente en programación paralela.

b. MAESTRO-ESCLAVO

Intervienen dos tipos de procesos:

→ Proceso Maestro: Descompone el problema en subtareas, las distribuye entre los esclavos y va recibiendo resultados parciales hasta producir el resultado final.

→ Procesos esclavos: ejecutan iterativamente hasta que el maestro informa del final: (1) Recibir mensaje con otra tare, (2) Procesar tarea, (3) enviar resultado al maestro.



Ejemplo: Calculo del conjunto de Mandelbrot

Conjunto de Mandelbrot: *Conjunto de puntos c del plano complejo (dentro de un circulo de radio 2 centrado en el origen) que no excederán cierto límite cuando calculan realizand la iteración ($z_0 = 0$):*

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

- Color pixel *c* depende del número de iteraciones (*k*) requeridas.
- Conjunto solución = {pixels que agotan iteraciones límite dentro de un círculo de radio 2 centrado en el origen}.

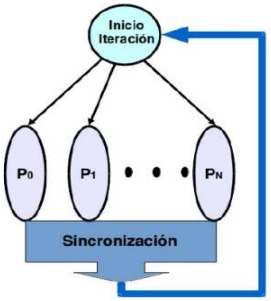
Paralelización sencilla: Cada pixel se puede calcular sin duda información del resto.

- Primera aproximación: asignar un número de pixels fijo a cada proceso esclavo y recibir resultados.
  - Problema: Algunos esclavos tendrían más trabajo que otros (numero de iteraciones por pixel variable).
- Segunda aproximación:
  - Maestro tiene una colección de filas de pixels.
  - Cuando esclavos están ociosos esperan recibir una fila.
  - Cuando no quedan más filas, Maestro espera la finalización de todos los esclavos e informa del final.
- Solución con envío asíncrono seguro y recepción síncrona.

```
process Maestro ;
begin
   for i := 0 to num_esclavos-1 do send( fila, Esclavo[i] );
   while queden filas sin colorear do
      select
         for j := 0 to n_t - 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
```

```
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
```

c. ITERACIÓN SÍNCRONA



Iteración: Un cálculo se repite y cada vez se obtiene un resultado que se utiliza en el siguiente cálculo.
A menudo, los cálculos de cada iteración se pueden realizar de forma concurrente.

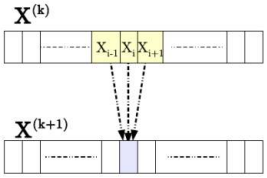
Paradigma de iteración síncrona:

- Diversos procesos comienzan juntos en el inicio de cada iteración.
- Sincronización: La siguiente iteración no puede comenzar hasta que todos hayan acabado la anterior.
- Los procesos suelen intercambiar información en cada iteración.

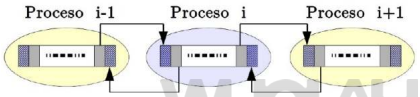
TRANSFORMACIÓN ITERATIVA DE UN VECTOR

Supongamos que debemos realizar *m* iteraciones de un cálculo que transforma un vector *x* de *n* reales.

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

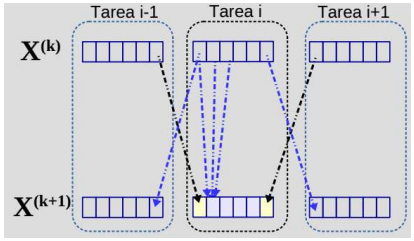


Solución distribuida que usa envío asíncrono seguro y recepción síncrona.



Patrón de comunicación:

- Se lanzan *p* procesos concurrentes.
- Vector repartido por bloques de *n/p* elementos consecutivos entre los *p* procesos.
- Cada proceso guarda su bloque en un vector local (*bloque)* con *n/p + 2* entradas (2 adicionales).
- Primera y última entrada del vector: almacena elementos recibidos de otros procesos.



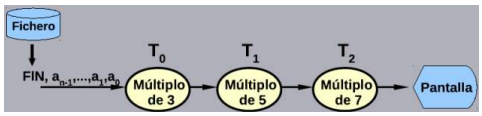

```
process Tarea[ i : 0..p-1 ] ;
   var bloque : array[0..n/p+1] of float ; { bloque local con dos celdas extra}
       float izquierda;
begin
   for k := 0 to M do begin { bucle que ejecuta las iteraciones }
      { comunicación de valores extremos con los vecinos }
      send( bloque[1]    ,Tarea[i-1 mod p] );
      send( bloque[n/p],Tarea[i+1 mod p] );
      receive( bloque[0], Tarea[i-1 mod p] );
      receive( bloque[n/p+1],   Tarea[i+1 mod p] );
      {Actualizar todas las entradas}
      for j := 1 to n/p do begin
          izquierda=bloque[j-1];
          bloque[j] := ( izquierda - bloque[j] + bloque[j+1] )/2;
      end
   end
end
```

d. ENCAUZAMIENTO (PIPELINING)

→ Problema se divide en una serie de tareas que se han completar en secuencia.
→ Cada tarea se ejecuta por un proceso separado.
→ Los procesos se originan 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 devuelve información necesaria para etapas posteriores.
→ Aplicación: Procesamiento en cadena de gran número de items de datos.

Ejemplo Cauce paralelo para filtrar una lista de enteros

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




```
process Etapa[ i : 0..m-1 ] ;
   var izquierda : integer := 0 ;
       { vector (replicado) con la lista de primos }
       primos : array[0..m-1] of float := { p0, p1, p2, ···, pm-1 } ;
begin
   while izquierda >= 0 do begin
      if i == 0 then
         leer( izquierda ); { obtiene siguiente entero }
      else
         receive( izquierda, Etapa[i-1]);
      if izquierda mod primos[i] == 0 then begin
         if i != m-1 then
            s_send ( izquierda, Etapa[i+1]);
         else
            imprime( izquierda );
      end
   end
end
```

3. MECANISMOS DE ALTO NIVEL EN SISTEMAS DISTRIBUIDOS

a. INTRODUCCIÓN

Los mecanismos vistos hasta ahora (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)

Están basados en el método habitual por el cual un proceso hace una llamada a procedimiento, como sigue:

1) Indica el nombre procedimiento y valores de parámetros.
2) Proceso ejecuta código del procedimiento.
3) Cuando procedimiento termina, proceso obtiene resultados y continúa tras la llamada.

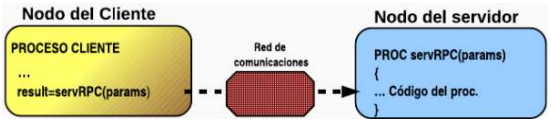
Llamada a procedimiento remoto

En el modelo de llamada a procedimiento remoto (RPC), es otro proceso (proceso llamado) el que ejecuta el código del procedimiento:

1) Llamador indica nombre de procedimiento y valores de parámetros.
2) Llamador queda bloqueado. Proceso llamado ejecuta código procedim.
3) Cuando procedimiento termina, llamador obtiene resultados y continúa.

Características RPC:

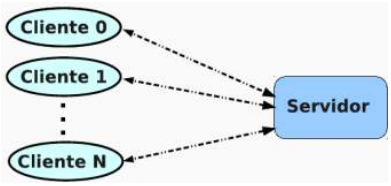
- Flujo de comunicación bidireccional (petición-respuesta).
- Varios procesos podrían invocar procedimiento gestionado por otro proceso (esquema muchos a uno).



b. EL PARADIGMA CLIENTE-SERVIDOR

Paradigma más frecuente en programación distribuida.

Relación asimétrica entre 2 procesos: cliente y servidor.



- Proceso servidor: gestiona recurso (p.e. base de datos) y ofrece servicio a otros procesos (clientes) para que puedan acceder al recurso. Puede estar ejecutándose continuamente, pero no hace nada útil mientras espera peticiones de clientes.
- Proceso cliente: envía un mensaje de petición al servidor solicitando un servicio proporcionado por el servidor (p.e. una consulta en base de datos).

Implementación de la interacción cliente-servidor usando los mecanismos vistos. Servidor con select que acepta peticiones de cada cliente:

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

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

Problemas de seguridad:

- Si el servidor falla, cliente queda esperando respuesta que nunca llegará.
- Si un cliente no invoca receive (respuesta,Servidor) y el servidor realiza envío síncrono, el servidor quedará bloqueado.

Solución: (recepción petición, envío respuesta) debe considerarse como única operación de comunicación. bidireccional en servidor (no 2 separadas).

El mecanismo de RPC proporciona solución en esta línea.

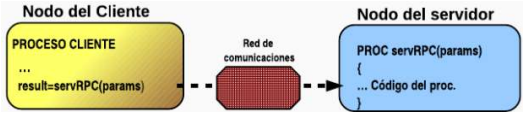
c. LLAMADA A PROCEDIMIENTO (RPC)

Llamada a procedimiento remoto (Remote Procedure Call)

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

Diferencia ppal respecto llamada local:

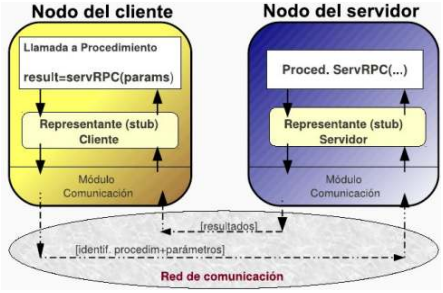
- Programa que invoca el procedimiento (cliente) y el procedimiento invocado (corre en proceso servidor) pueden pertenecer a máquinas diferentes del sistema distribuido.



ESQUEMA DE INTERACCIÓN EN RPC

Representante o delegado (stub): procedimiento local que gestiona la comunicación en el lado del cliente o del servidor.

- Procesos cliente y servidor no se comunican directamente, sino a través de representantes.



1) LLAMADA RPC: INICIO EN CLIENTE Y ENVÍO PARÁMETROS

a. En nodo cliente se invoca procedimiento remoto como si fuera local. Esta llamada se traduce a una llamada al representante del cliente.
b. Marshalling o Serialización: Representante cliente empaqueta datos llamada (nombre procedim. y parámetros) usando un determinado formato para formar el cuerpo del mensaje a enviar (p.e. el protocolo XDR, eXternal Data Representation).
c. Representante cliente envía mensaje con petición al nodo servidor usando módulo de comunicación sistema operativo.
d. Programa cliente queda bloqueado esperando respuesta.

2) LLAMADA RPC: EJEC. EN SERVIDOR Y ENVÍO RESULTADOS

a. El sistema operativo servidor desbloquea proceso servidor para que se haga cargo de la petición y mensaje es pasado al representante servidor.
b. Representante servidor desempaqueta datos mensaje (unmarshalling) (identificación procedimiento + parámetros) y ejecuta llamada al procedim, local usando parámetros obtenidos.
c. Finalizada la llamada, Representante servidor empaqueta resultados en un mensaje y lo envía al cliente.
d. Sistema operativo cliente desbloquea proceso invocador para recibir resultado, que es pasado a Representante cliente.
e. Representante cliente desempaqueta mensaje y pasa resultados al invocador.

REPRESENTACIÓN DE DATOS Y PASO DE PARÁMETROS

Representación de los datos

- Nodos pueden tener diferente hardware y/o sistema operativo (sistema heterogéneo) y usar diferentes formatos representac de datos.
- Solución: Mensajes se envían usando representación intermedia. Representantes de cliente y servidor efectúan conversiones necesarias.

Paso de parámetros

- Por valor: Se envía al representante servidor los datos aportados.
- Por referencia: el objeto referenciado debe enviarse al servidor.
    - Si puede ser modificado en servidor, debe enviarse de vuelta al cliente al final (copia de valor-resultado).

d. JAVA REMOTE METHOD INVOCATION (RMI)

Invocación de métodos en programas orientados a objetos

- Se debe aportar: referencia del objeto + método concreto + argumentos.
- Interfaz Objeto: define métodos, argumentos, tipos de valores devueltos y excepciones.

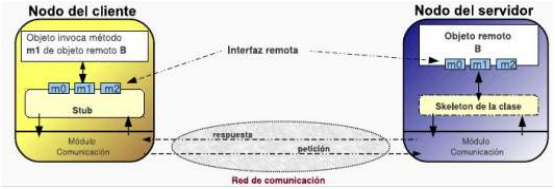
Invocación de métodos remotos (RMI)

- En entornos distribuidos, un objeto podría invocar métodos de otro objeto (remoto), localizado en un nodo o proceso diferente del llamador, siguiendo paradigma cliente-servidor (como RPC).
- Para invocar métodos de un objeto remoto, llamador debe:
    - Proporcionar: nombre método + parámetros
    - Identificar objeto remoto y proceso/nodo donde reside.

INTERFAZ REMOTA Y REPRESENTANTES

Interfaz remota: especifica métodos del objeto remoto accesibles para demás objetos + excepciones derivadas (p.ej.., respuesta tardía servidor).

- Remote Method Invocation (RMI): acción de invocar un método de la interfaz remota de un objeto remoto.
    - Sigue la misma sintaxis que sobre un objeto local.



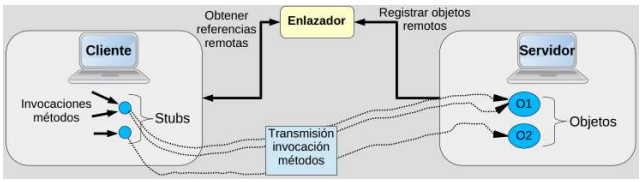
Cliente y servidor deben conocer interfaz remota (nombres + parámetros métodos accesibles)

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

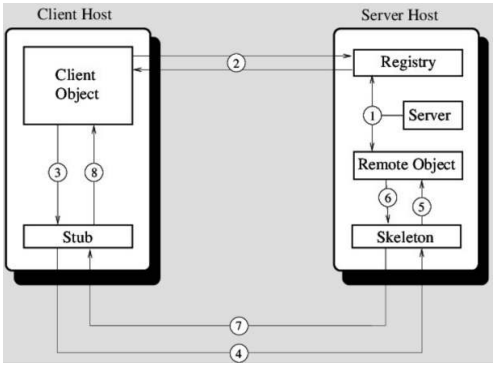
Stub y skeleton hacen transparente al programador detalles de comunicación y empaquetamiento datos (tanto en cliente como en servidor).

REFERENCIAS REMOTAS

- Los stubs usan la definición de la interfaz remota.
- Objetos remotos residen en servidor y son gestionados por el mismo.
- Procesos clientes manejan referencias remotas a objetos remotos:
- Referencia remota: permite al cliente localizar objeto remoto en sist. distribuido. Incluye: dirección IP servidor, puerto escucha y el identificador del objeto.
    - Contenido no directamente accesible, gestionado por stub y enlazador.
- Enlazador: Servicio sist. dist., Mapping {nombres} → {referencias remotas}.



Ejemplo: Interacción en Java RMI

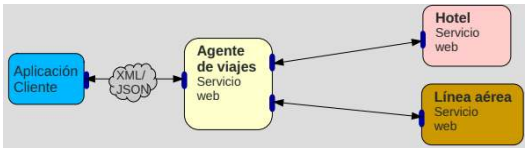


e. SERVICIOS WEB

CARACTERÍSTICAS

Actualmente, gran parte de la comunicación en Internet ocurre vía los servicios web.

- Protocolos HTTP o HTTPS en capa aplicación sobre TCP/IP en capa transporte.
- Codificación de datos: basada en XML o JSON (JavaScript Object Notation).
- Es posible usar protocolos complejos (p.ej.SOAP), pero generalmente se usa el método REST (Representational State Transfer), caracterizado por:
    - Clientes solicitan recurso o documento especificando su URL.
    - Servidor responde enviando recurso en versión actual o notificando error.
    - Cada petición es independiente de otras: enviada respuesta, servidor no guarda información de estado de sesión/cliente (REST es stateless).



LLAMADAS Y SINCRONIZACIÓN

Peticiones de recursos/documentos desde:

- una aplicación cualquiera ejecutándose en el cliente.
- Un programa Javascript ejecutándose en navegador en nodo cliente (más frecuente).

Gestión de peticiones:

- Síncrona: Proceso cliente espera bloqueado la respuesta.
    - No aceptable en aplicaciones web interactivas (paraliza interacción usuario).
- Asíncrona: Proceso cliente envía petición y continúa.
    - Al recibir respuesta, se ejecuta una función (designada por cliente al hacer petición) que tiene respuesta como argumento.