

Sistemas Concurrentes y Distribuidos.

Tema 3. Sistemas basados en paso de mensajes..

Dpt. Lenguajes y Sistemas Informáticos
ETSI Informática y de Telecomunicación
Universidad de Granada

Curso 13-14

Índice

Sistemas Concurrentes y Distribuidos.

Tema 3. Sistemas basados en paso de mensajes..

- 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

Indice de la sección

Sección 1

Mecanismos básicos en sistemas basados en paso de mensajes

1.1. Introducción

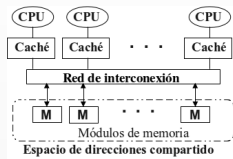
1.2. Vista logica arquitectura y modelo de ejecución

1.3. Primitivas básicas de paso de mensajes

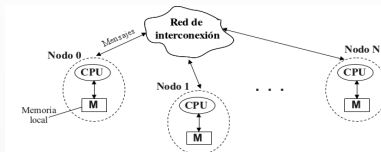
1.4. Espera selectiva

Introducción. Multiprocesador con mem. compartida vs. Sistema Distribuido

Multiprocesador de mem. compartida



Sistema Distribuido



► Multiprocesador:

- 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.

► Sistema Distribuido:

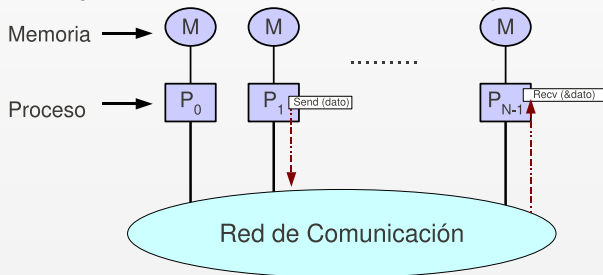
- Distribución de datos y recursos.
- Soluciona el problema de la escalabilidad y el elevado coste.
- Mayor dificultad de programación: No hay direcciones de memoria comunes y mecanismos como los monitores son inviables.

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.

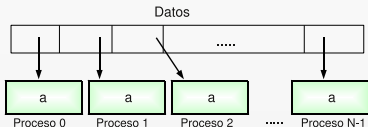
Vista lógica de la arquitectura

- ▶ N procesos, cada uno con su espacio de direcciones propio (memoria).
- ▶ En un mismo procesador podrían residir físicamente varios procesos, aunque es muy común la ejecución 1 proceso-procesador.
- ▶ 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

- ▶ Ejecutar un programa diferente sobre cada proceso puede ser algo difícil de manejar.
- ▶ **Enfoque común: Estilo SPMD (Single Program Multiple Data).**
 - ▶ El código que ejecutan los diferentes procesos es idéntico, pero sobre datos diferentes.
 - ▶ El código incluye la lógica interna para que cada tarea ejecute lo que proceda dependiendo de la identidad del proceso.

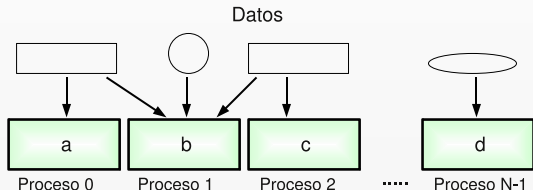


```
{ ejemplo: Proceso[0] envia y Proceso[1] recibe }
process Proceso[ nump : 0..1 ];
begin
  if nump = 0 then begin {si soy 0}
    dato := Produce();
    send( dato, Proceso[1]);
  end else begin {si soy 1}
    receive( dato, Proceso[0] );
    Consume( dato );
  end
end
```

Estructura de un programa de paso de mensajes. MPMD

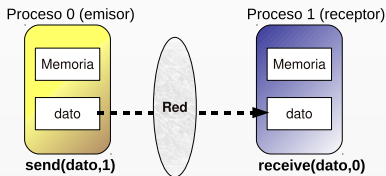
► Estilo MPMD (Multiple Program Multiple Data):

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



Primitivas básicas de paso de mensajes

- **Envío:** **send** (*expresion, identificador_proceso_destino*)
- **Recepción:** **receive** (*variable, identificador_proceso_origen*)



Las primitivas **send** y **receive** permiten:

- **Comunicación:** Los procesos envían/reciben mensajes en lugar de escribir/leer en variables compartidas.
- **Sincronización:** la recepción de un mensaje es posterior al envío. Generalmente, la recepción supone la espera del mensaje por parte del receptor.

Esquemas de identificación de la comunicación

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

Existen dos posibilidades:

Denominación directa

- Emisor identifica explícitamente al receptor y viceversa.
- Se utilizan los identificadores de los procesos.

Denominación indirecta

Los mensajes se depositan en almacenes intermedios que son globales a los procesos (buzones).

Denominación directa

- **Ventaja principal:** No hay retardo para establecer la identificación.
- **Inconvenientes:**
 - Cambios en la identificación requieren recompilar el código.
 - Sólo permite enlaces 1-1.

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

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



- Existen **esquemas asimétricos**: el emisor identifica al receptor, pero el receptor no indica emisor:
 - **receive** (*variable*, *id.origen*): en *id.origen* se devuelve el emisor del mensaje.
 - **receive** (*variable*, *ANY*): *ANY* indica que se puede recibir de cualquier emisor.

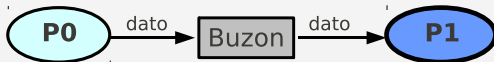
Denominación indirecta

- Presenta una mayor **flexibilidad** al permitir la comunicación simultánea entre varios procesos.
- **Declaración estática/dinámica:**
 - Estática: fija fuente/destino en tiempo de compilación. No apropiada en entornos cambiantes.
 - Dinámica: más potente pero menos eficiente

```
var buzon : channel of integer ;
```

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

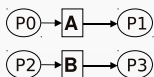
```
process P1 ;  
var midato : integer ;  
begin  
    receive( midato, buzon );  
    Consume( midato );  
end
```



Denominación indirecta (2)

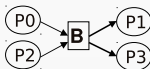
- Existen tres tipos de buzones: canales (uno a uno), puertos (muchos a uno) y buzones generales (muchos a muchos).

Canales (1 a 1)



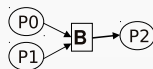
- Tipo fijo

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.

Comportamiento de las operaciones de paso de mensajes

- ▶ El comportamiento de las operaciones puede variar dependiendo de las necesidades. Ejemplo:

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

```
process Receptor ;  
var b : integer ;  
begin  
    receive( b, Emisor ) ;  
    imprime( b ) ;  
end
```

- ▶ El comportamiento esperado del **send** impone que el valor recibido en **b** sea el que tenía **a** (100) justo antes de la llamada a **send**.
- ▶ Si se garantiza esto, se habla de comportamiento *seguro* (se habla de que el programa de paso de mensajes es seguro).
- ▶ Generalmente, un programa no seguro (en el ejemplo, si pudiera recibirse 0 en lugar de 100), se considera incorrecto, aunque existen situaciones en las que puede interesar usar operaciones que no garantizan dicha seguridad.

Instantes críticos en las operaciones de paso de mensajes

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

```
process Receptor ;  
var b : integer ;  
begin  
    receive( b, Emisor ) ;  
    imprime( b ) ;  
end
```

Instantes relevantes en el emisor:

- E_1 : Emisor acaba de indicar al Sistema de Paso de Mensajes (SPM) la dirección de a y el identificador del Receptor.
- E_2 : SPM comienza la lectura de datos en a .
- E_3 : SPM termina de leer todos los datos en a .

Instantes relevantes en el receptor:

- R_1 : Receptor acaba de indicar al SPM la dirección de b y el identificador del Emisor.
- R_2 : SPM comienza a escribir en b .
- R_3 : SPM termina de escribir en b .

Ordenación de los instantes relevantes

Los eventos relevantes citados deben ocurrir en cierto orden:

- ▶ El emisor puede comenzar el envío antes que el receptor, o al revés. Es decir, no hay un orden temporal entre E_1 y R_1 (se cumple $E_1 \leq R_1$ o $R_1 \leq E_1$).
- ▶ Lógicamente, los tres eventos del emisor se producen en orden, y también los tres del receptor. Se cumple que: $E_1 < E_2 < E_3$ y también que $R_1 < R_2 < R_3$.
- ▶ Antes de que se escriba el primer byte en el receptor, se debe haber comenzado ya la lectura en el emisor, por tanto $E_2 < R_2$.
- ▶ Antes de que se acaben de escribir los datos en el receptor, se deben haber acabado de leer en el emisor, es decir $E_3 < R_3$.
- ▶ Por transitividad se puede demostrar que R_3 **es siempre el último evento**.

Mensajes en tránsito

Por la hipótesis de progreso finito, el intervalo de tiempo entre E_1 y R_3 tiene una duración no predecible. Entre E_1 y R_3 , 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 de envío y/o recepción podrían **no ser seguras**, es decir, el valor que el emisor pretendía enviar podría no ser el mismo que el receptor finalmente recibe, ya que se usan zonas de memoria, accesibles desde los procesos involucrados, que podrían escribirse cuando el mensaje está en tránsito.

- **Operación de envío-recepción segura:**

Ocurre cuando se puede garantizar a priori que el valor de a en E_1 coincidirá con el valor de b justo tras R_3 .

- **Operación de envío-recepción insegura:**

Una operación puede ser no segura en dos circunstancias:

- se puede modificar el valor de a entre E_1 y E_3 (**envío inseguro**)
- se puede modificar el valor de b entre R_2 y R_3 (**recepción insegura**).

Tipos de operaciones de paso de mensajes

Operaciones seguras

- Devuelven el control cuando se garantiza la seguridad.
- No siempre significa que el receptor haya recibido el dato cuando finalice el **send**, sino que cambiar los datos no viola la seguridad.
- Existen 2 mecanismos de paso de mensajes seguro:
 - Envío y recepción síncronos.
 - Envío asíncrono seguro

Operaciones inseguras

- Devuelven el control de forma inmediata sin garantizar la seguridad.
- Es responsabilidad del usuario asegurar que no se alteran los datos mientras el mensaje está en tránsito.
- Deben existir sentencias adicionales para comprobar el estado de la operación.

El estándar **MPI** incluye funciones con todas estos comportamientos.

Operaciones síncronas. Comportamiento

```
s_send( variable, id_proceso_receptor ) ;
```

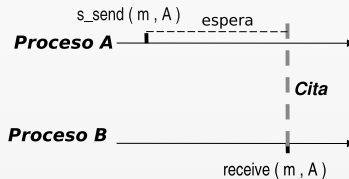
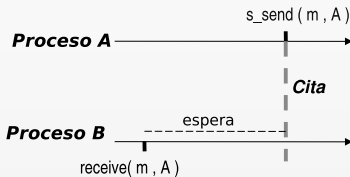
- ▶ Realiza el envío de los datos y espera bloqueado hasta que los datos hayan terminado de escribirse en la memoria local designada por el proceso receptor en su llamada a la operación de recepción.
- ▶ Es decir: devuelve el control después de R_3 .

```
receive( variable, id_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 *variable*.
- ▶ Es decir: devuelve el control después de R_3 .

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

```
{ P0 }  
s_send( a, P1 );  
receive( b, P0 );
```


```
{ P1 }  
s_send( a, P0 );  
receive( b, P1 );
```



Corrección

```
{ P0 }  
s_send( a, P1 );  
receive( b, P0 );
```

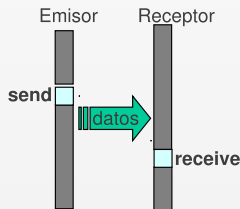
```
{ P1 }  
receive( b, P1 );  
s_send( a, P0 );
```



Envío asíncrono seguro (1)

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

- ▶ Inicia el envío de los datos designados y espera bloqueado hasta que hayan terminado de leerse todos los datos de `variable` en una memoria local del Emisor (buffer). Tras la copia de datos a memoria local, devuelve el control sin que tengan que haberse recibido los datos en el receptor.
- ▶ Por tanto, devuelve el control después de E_3 .
- ▶ Se suele usar junto con la recepción síncrona (**receive**).



Envío asíncrono seguro. Ventajas e Inconvenientes

- Alivia sobrecargas de espera ociosa a costa de gestión de almacenamiento temporal (buffer).
- Sólo menos ventajosas en programas altamente síncronos o cuando la capacidad del buffer sea un asunto crítico.
- **Impacto del buffer finito:** Se deben escribir programas con requisitos de almacenamiento acotados.
 - Si $P1$ es más lento que $P0$, $P0$ podría continuar siempre que hubiese buffer. Si el buffer se agota, $P0$ se bloquearía.

```
{ P0 } for i:=1 to 10000 do begin
    a := Produce() ;
    send( a, P1 ) ;
end
```

```
{ P1 } for i:=1 to 10000 do begin
    receive( a, P0 ) ;
    Consume( a ) ;
end
```

- **Interbloqueo con buffer:** No olvidar que las llamadas a `receive` siguen siendo síncronas.

```
{ P0 } receive( b, P1 );
      send( a, P1 );
```

```
receive( b, P0 ); { P1 }
send( a, P0 );
```


Operaciones inseguras

- **Operaciones seguras:** Garantizan la seguridad, pero presentan ineficiencias:
 - sobrecarga de espera ociosa (sin buffer).
 - sobrecarga de gestión de buffer (con buffer).
- **Alternativa:** Requerir que el programador asegure la corrección y usar operaciones `send/receive` con baja sobrecarga.
 - 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).
 - Es responsabilidad del usuario asegurar que el comportamiento es correcto.
 - 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 el estado de la operación.

Paso de mensajes asíncrono inseguro. Operaciones

i_send(*variable*, *id_proceso_receptor*, *var_resguardo*) ;

- Indica al SPM que comience una operación de envío al proceso receptor designado. No espera a que terminen de leerse los bytes del emisor, ni a que terminen de escribirse en el receptor.
- Es decir: devuelve el control después de E_1 .
- *var_resguardo* permite consultar el estado del envío.

i_receive (*variable*, *id_proceso_receptor*, *var_resguardo*) ;

- Indica al SPM que el proceso receptor desea recibir en la zona de memoria local designada en *variable* un mensaje del proceso emisor indicado. Se devuelve el control sin esperar a que se complete el envío ni a que llegen los datos.
- Es decir: devuelve el control después de R_1 .
- *var_resguardo* permite consultar después el estado de la 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. Para esto 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 E_3 .

```
wait_rcv( 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 R_3 .

Paso de mensajes asíncrono inseguro (no bloqueante)

- ▶ El proceso que ejecuta **i_send** informa al SPM de un mensaje pendiente y continúa.
 - ▶ El programa puede hacer mientras otro trabajo (`trabajo_util`)
 - ▶ El SPM iniciará la comunicación cuando corresponda.
 - ▶ Operaciones de chequeo indican si es seguro tocar/leer los datos.
- ▶ **Mejora:** El tiempo de espera ociosa se puede emplear en computación
- ▶ **Coste:** Reestructuración programa, mayor esfuerzo del programador.

```
process Emisor ;  
  var a : integer := 100 ;  
begin  
  i_send( a, Receptor, resg );  
  { trabajo util : 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 util : no accede a 'b' }  
  trabajo_util_receptor();  
  wait_rcv (resg);  
  imprime( b );  
end
```

Ejemplo: Productor-Consumidor con paso asíncrono

Se logra simultaneizar transmisión, producción y consumición:

```
process Productor ;  
  var x, x_env : integer ;  
begin  
  x := Producir()  
  while true do begin  
    x_env := x ;  
    i_send(x_env, Consumidor, resg);  
    x := Producir() ;  
    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);  
    Consumir(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 a E_3 , devuelve **true**, sino devuelve **false**.

```
test_rcv ( var_resguardo ) ;
```

- Función lógica que se invoca por el receptor. Si el envío asociado a *var_resguardo* ha llegado a R_3 , 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  
    { trabajo util: no escribe en 'a' }  
    trabajo_util_emisor();  
    a := 0 ;  
  end
```

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

Si el trabajo util 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 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 ;
            imprime(" recibido de 1 : ", b1 );
        end
        if not r2 and test_recv( resg2 ) then begin
            r2 := true ;
            imprime(" recibido de 2 : ", b2 );
        end
    end
end
end
```


Espera Selectiva. Introducción al problema

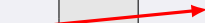
- ▶ Los modelos basados en paso de mensajes imponen ciertas restricciones.
- ▶ No basta con **send** y **receive** para modelar el comportamiento deseado.

Productor-Consumidor con buffer de tamaño fijo

- ▶ Se asumen operaciones síncronas.
- ▶ **Problema:** Acoplamiento forzado entre los procesos.

```
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
```



Introducción al problema. Mejora

- ▶ **Mejora:** Gestión intercambio mediante proceso Buffer.
 - ▶ Productor puede continuar después envío.
 - ▶ **Problema:** El buffer sólo puede esperar mensajes de un único emisor en cada instante.
- ▶ Aspecto común en aplicaciones cliente-servidor.
 - ▶ No se conoce a priori el cliente que hace la petición en cada instante.
 - ▶ Servidor debe estar preparado para recibir sin importar orden.

```
{ Productor (P) }  
process P ;  
begin  
  while true do begin  
    v:=Produce();  
    s_send(v,B);  
  end  
end
```

```
{ Buffer (B) }  
process B ;  
begin  
  while true do begin  
    receive(v,P);  
    receive(s,C);  
    s_send(v,C);  
  end  
end
```

```
{ Consumidor (C) }  
process C ;  
begin  
  while true do begin  
    s_send(s,B);  
    receive(v,B);  
    Consume(v);  
  end  
end
```

Espera selectiva con alternativas guardadas (1)

Solución: usar **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**.
- ▶ En cada alternativa, el texto desde **when** hasta **do** se llama la **guarda** de dicha alternativa, la guarda puede incluir una expresión lógica (*condicion*_{*i*})
- ▶ Los **receive** nombran a otros procesos del programa concurrente (*proceso*_{*i*}), y cada uno referencia una variable local (*variable*_{*i*}), donde eventualmente se recibirá un valor del proceso asociado.

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**.

Evaluación de las guardas.

Cuando el flujo de control llega a un **select**, se evalúan las condiciones de todas las guardas, y se verifica el estado de todos los procesos nombrados en los **receive**. De esta forma se clasifican las guardas, y 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 un select

Para ejecutar un select primero se intenta seleccionar una guarda de acuerdo con el esquema descrito arriba.

- ▶ Si no se puede seleccionar ninguna guarda, no se hace nada (no hay guardas ni ejecutables, ni potencialmente ejecutables).
- ▶ Si se ha podido seleccionar una guarda, entonces 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).

Existe **select** con prioridad: la selección no sería arbitraria (el orden en el que aparecen las alternativas establece la prioridad).

Productor-Consumidor con buffer limitado

- ▶ Buffer no conoce a priori orden de peticiones (Inserción/Extracción).
- ▶ Las guardas controlan las condiciones de sincronización (seguridad).

```
{ Productor (P) }
while true do
begin
  v:=Produce();
  s_send(v,B);
end
```

```
{ Buffer (B) }
var esc, lec, cont : integer := 0 ;
buf : array[0..tam-1] of integer ;
begin
  while true do
    select
      when cont<=tam receive(v,P) do
        buf[esc]:= v ;
        esc := (esc+1) mod tam ;
        cont := cont+1 ;
      when cont ≠ 0 receive(s,C) do
        s_send(buf[lec],C) ;
        esc := (esc+tam-1) mod tam ;
        cont := cont-1 ;
    end
  end
end
```

```
{ Consumidor (C) }
while true do
begin
  s_send(s,B) ;
  receive(v,B) ;
  Consume(v) ;
end
```


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

Este ejemplo suma los primeros numeros recibidos de cada uno de los n procesos fuente hasta que cada suma iguala o supera al valor 1000:

```
var suma      : array[0.. $n$ -1] of integer := (0,0,...,0) ;
    continuar : boolean := false ;
    numero    : integer ;
begin
    while continuar do begin
        continuar := false ; { terminar cuando  $\forall i$  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
```

aquí hemos supuesto que los procesos fuente están definidos como:

```
process fuente[ i : 0.. $n$ -1 ] ;
begin
    .....
end
```

Indice de la sección

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)

Introducción

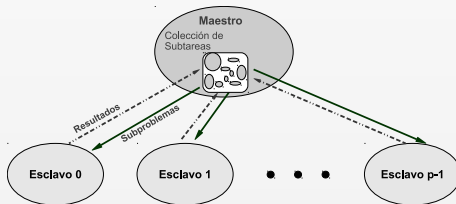
Paradigma de interacción

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-Esclavo.
 - 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).

Maestro-Eslavo

- ▶ 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.

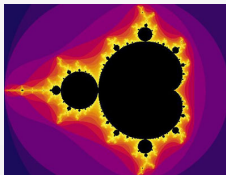


Ejemplo: Cálculo del Conjunto de Mandelbrot

- **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
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
end
```

Iteración síncrona

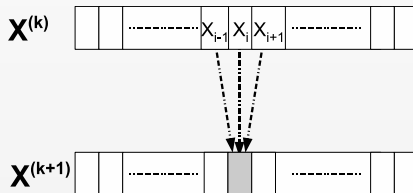
- **Iteración:** En múltiples problemas numéricos, un cálculo se repite y cada vez se obtiene un resultado que se utiliza en el siguiente cálculo. El proceso se repite hasta obtener los resultados deseados.
- A menudo se pueden realizar los cálculos de cada iteración de forma concurrente.
- **Paradigma 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:

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

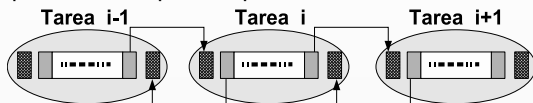
$$x_{-1}^{(k)} = x_{n-1}^{(k)}, \quad x_n^{(k)} = x_0^{(k)}.$$



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 local (no compartido) }
begin
  for j := 0 to  $n/p-1$  do bloque[j] := valores[i* $n/p+j$ ] ; { lee valores }
  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 siguiente }
    { 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
  for j := 0 to  $n/p-1$  do valores[i* $n/p+j$ ] := bloque[j] ; { escribe resultados }
end

```

Encauzamiento (pipelining)

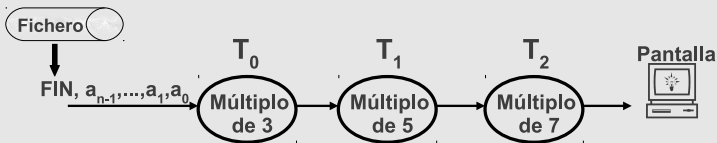
- El problema se divide en una serie de tareas que se han de completar una después de otra
- Cada tarea se ejecuta por un proceso separado.
- Los procesos se organizan en un cauce (pipeline) donde cada proceso se corresponde con una *etapa* del cauce y es responsable de una tarea particular.
- Cada etapa 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 **Etapa** $[i]$ (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)

```

{ vector (compartido) con la lista de primos }
var primos : array[0.. $m-1$ ] of float := {  $p_0, p_1, p_2, \dots, p_{m-1}$  } ;

{ procesos que ejecutan cada etapa: }

process Etapa[ i : 0.. $m-1$  ] ;
    var izquierda : integer := 0 ;
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
end

```


Indice de la sección

Sección 3

Mecanismos de alto nivel en sistemas distribuidos

3.1. Introducción

3.2. El paradigma Cliente-Servidor

3.3. Llamada a Procedimiento (RPC)

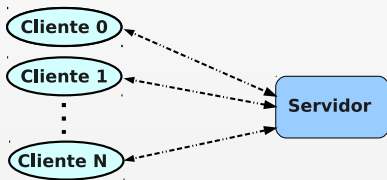
3.4. Java Remote Method Invocation (RMI)

Introducción

- ▶ Los mecanismos vistos hasta ahora (varios tipos de envío/recepción espera selectiva, ...) presentan un bajo nivel de abstracción.
- ▶ Se verán dos mecanismos de mayor nivel de abstracción: **Llamada a procedimiento remoto (RPC)** e **Invocación remota de métodos (RMI)**.
- ▶ Están basados en la forma de comunicación habitual en un programa: *llamada a procedimiento*.
 - ▶ El llamador proporciona nombre del proc + parámetros, y espera.
 - ▶ Cuando proc. termina, el llamador obtiene los resultados y continúa.
- ▶ En el **modelo de invocación remota**:
 - ▶ El llamador invoca desde un proceso o máquina diferente de donde se encuentra el procedimiento invocado.
 - ▶ El llamador se queda bloqueado hasta que recibe los resultados (esquema síncrono).
 - ▶ El flujo de comunicación es bidireccional (petición-respuesta).
 - ▶ Se permite que varios procesos invoquen un procedimiento gestionado por otro proceso (esquema muchos a uno).

El paradigma Cliente-Servidor

- Paradigma más frecuente en programación distribuida.
- 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( 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
  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.

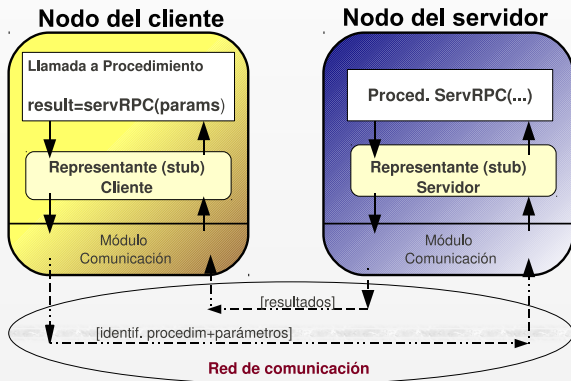
Introducción a RPC

- ▶ **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.



Esquema general de una RPC. Inicio en el nodo cliente

- 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.

Esquema general de una RPC. Pasos en el nodo servidor y recepción de resultados en cliente

- 1 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.
- 2 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.
- 3 Una vez finalizada la llamada, el representante del servidor empaqueta los resultados en un mensaje y lo envía al cliente.
- 4 El sistema operativo del nodo cliente desbloquea al proceso que hizo la llamada para recibir el resultado que es pasado al representante del cliente.
- 5 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 En RPC, los parámetros de la llamada se pueden pasar:

- *por valor*: en este caso basta con enviar al representante del servidor los datos aportados por el cliente.
- *por referencia*: En este caso se pasa un puntero, por lo que se han de transferir también los datos referenciados al servidor. Además, en este caso, cuando el procedimiento remoto finaliza, el representante del servidor debe enviar al cliente, junto con los resultados, los parámetros pasados por referencia que han sido modificados.

El modelo de objetos distribuidos

▸ Invocación de métodos en progr. orientada a objetos

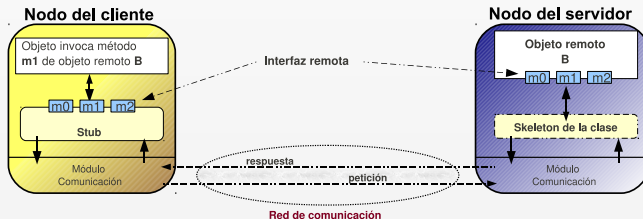
- En programación orientada a objetos, los objetos se comunican entre sí mediante invocación a métodos.
- Para invocar el método de un objeto hay que dar una referencia del objeto, el método concreto y los argumentos de la llamada.
- La interfaz de un objeto define sus métodos, argumentos, tipos de valores devueltos y excepciones.

▸ Invocación de métodos remotos

- En un entorno distribuido, un objeto podría invocar métodos de un objeto localizado en un nodo o proceso diferente del sistema (**objeto remoto**) siguiendo el paradigma cliente-servidor como ocurre en RPC.
- Un objeto remoto podría recibir invocaciones locales o remotas. Para invocar métodos de un objeto remoto, el objeto que invoca debe disponer de la *referencia* del objeto remoto del nodo receptor, que es única en el sistema distribuido.

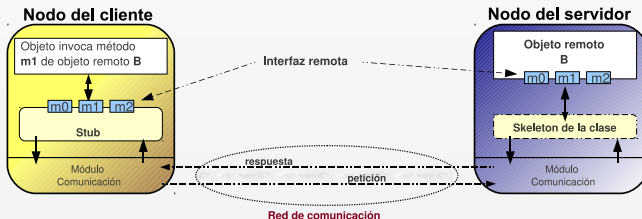
El modelo de objetos distribuidos. Interfaz remota y representantes

- **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.



El modelo de objetos distribuidos. Interfaz remota y representantes

- El esquema de RMI es similar a la RPC:
 - **En el cliente:** un representante local de la clase del objeto receptor (*stub*) implementa el marshalling y la comunicación con el servidor, compartiendo la misma interfaz que el objeto receptor.
 - **En el servidor:** la implementación de la clase del objeto receptor (*skeleton*) recibe y traduce las peticiones del stub, las envía al objeto que implementa la interfaz remota y espera resultados para formatearlos y enviárselos al stub del cliente.



El modelo de objetos distribuidos. Referencias remotas

- ▶ Los stubs se generan a partir de 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.
 - ▶ Obtienen una referencia unívoca con la localización del objeto remoto dentro del sistema distribuido (dirección IP, puerto de escucha e identificador).
- ▶ 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. Mantiene una tabla con pares (*nombre, referencia remota*).
 - ▶ En el cliente, se usa para obtener la referencia de un objeto remoto a partir de su nombre (`lookup()`).
 - ▶ En el servidor, se usa para registrar o ligar (`bind()`) sus objetos remotos por nombre de forma que los clientes pueden buscarlos.

Java RMI

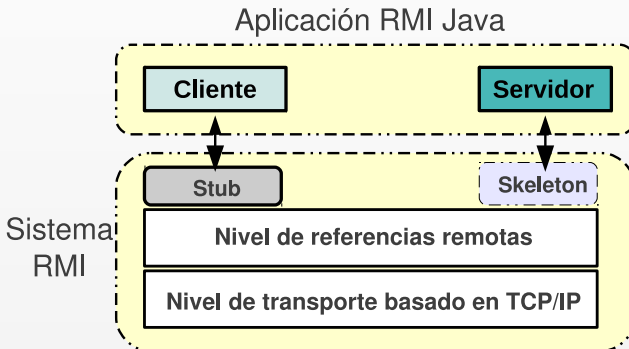
- ▶ Mecanismo que ofrece Java (desde JDK 1.1) para invocar métodos de objetos remotos en diferentes JVMs.
- ▶ Permite que un programa Java exporte un objeto para que esté disponible en la red, esperando conexiones desde objetos ejecutándose en otras máquinas virtuales Java (JVM).
- ▶ La idea es permitir la programación de aplicaciones distribuidas de forma integrada en Java como si se tratara de aplicaciones locales, preservando la mayor parte de la semántica de objetos en Java.
- ▶ Permite mantener seguro el entorno de la plataforma Java mediante gestores de seguridad y cargadores de clases.

Particularidades de Java RMI

- Las **interfaces remotas** se definen como cualquier interfaz Java pero deben extender una interfaz denominada `Remote` y lanzar excepciones remotas para actuar como tales.
- En una llamada remota, los parámetros de un método son todos de entrada y la salida es el resultado de la llamada.
 - Los parámetros que son objetos remotos se pasan por referencia.
 - Los parámetros que son objetos locales (argumentos no remotos) y los resultados se pasan por valor (las referencias solo tienen sentido dentro de la misma JVM).
 - Cuando el servidor no tiene la implementación de un objeto local, la JVM se encarga de descargar la clase asociada a dicho objeto.
- En Java RMI, el enlazador se denomina `rmiregistry`.

Arquitectura de Java RMI

- ▶ Generalmente las aplicaciones RMI constan de **servidor** y **clientes**.
- ▶ El sistema RMI proporciona los mecanismos para que el servidor y los clientes se comuniquen e intercambien información.



Arquitectura de Java RMI. Nivel de aplicación

Nivel de Aplicación: Los clientes y el servidor implementan la funcionalidad de la aplicación RMI:

- **Servidor:**

- Crea objetos remotos.
- Hace accesible las referencias a dichos objetos remotos. Para ello, se registran los objetos remotos en el `rmiregistry`.
- Se mantiene activo esperando la invocación de métodos sobre dichos objetos remotos por parte de los clientes.

- **Clientes:**

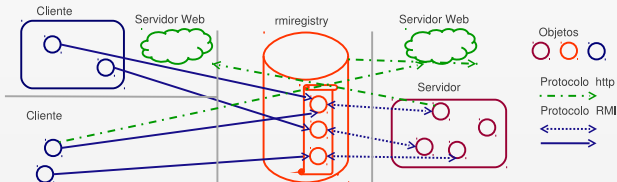
- Deben declarar objetos de la interfaz remota.
- Deben obtener referencias a objetos remotos (stubs) en el servidor.
- Invocan los métodos remotos (usando el stub).

Este tipo de aplicaciones se suelen denominar **Aplicaciones de objetos distribuidos**

Arquitectura de Java RMI. Nivel de aplicación(2)

Las aplicaciones de objetos distribuidos requieren:

- ▶ **Localizar objetos remotos:**
 - ▶ Para ello, las aplicaciones pueden registrar sus objetos remotos utilizando el servicio `rmiregistry`, o puede enviar y devolver referencias a objetos remotos como argumentos y resultados.
- ▶ **Comunicarse con objetos remotos:**
- ▶ **Cargar bytecodes de objetos** que se pasan como parámetros o valores de retorno.



Arquitectura de Java RMI. Sistema RMI

▸ Nivel de stubs

- El papel del *Skeleton* lo hace la plataforma RMI (a partir de JDK 1.2).
- El stub se genera automáticamente bajo demanda en tiempo de ejecución (a partir de JDK 1.5, antes se usaba un compilador).
- El stub tiene la misma interfaz que el objeto remoto y conoce su localización. Realiza todo el *marshalling* necesario para la llamada remota.
- Debe haber un stub por cada instancia de una interfaz remota.

▸ Nivel de referencias: Se encarga de:

- Interpretar las referencias remotas que manejan los stubs para permitirles acceder a los métodos correspondientes de los objetos remotos
- Enviar y recibir los paquetes (resultantes del *marshalling* de las peticiones/respuestas) usando los servicios del nivel de transporte.

▸ Nivel de transporte: Se encarga de conectar las diferentes JVMs en el sistema distribuido. Se basa en el protocolo de red TCP/IP.

Pasos para implementar una aplicación Java RMI

Suponiendo, por simplicidad, que el cliente usa un único objeto remoto, los pasos serían:

- 1 **Crear la interfaz remota**
- 2 **Implementar el objeto remoto**
- 3 **Implementar los programas clientes**

1. Crear la Interfaz remota

- La interfaz remota declara los métodos que pueden invocar remotamente los clientes.
- Deben extender la interfaz `Remote` del paquete `java.rmi`.
- Los métodos definidos deben poder lanzar una excepción remota.

Ejemplo: Interfaz de un contador remoto `Contador_I.java`

```
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface Contador_I extends Remote {
    public void incrementa(int valor) throws RemoteException;
    public int getvalor() throws RemoteException;
}
```

2. Implementar el objeto remoto

- Inicialmente se deben de definir los métodos de la interfaz remota:

Ejemplo: Clase para contador remoto `Contador.java`

```
import java.rmi.*;
import java.rmi.server.*;

public class Contador implements Contador_I{
    private int valor;

    public Contador() throws RemoteException {
        valor=0;
    }

    public void incrementa(int valor) throws RemoteException {
        this.valor+=valor;
    }

    public int getvalor() throws RemoteException {
        return valor;
    }
    ...}
```

2. Implementar el objeto remoto(2)

Una vez definidos los métodos de la interfaz, se debe crear el objeto remoto que implementa la interfaz remota y exportarlo al entorno RMI para habilitar la recepción de invocaciones remotas. Eso implica:

- 1 Crear e instalar un **gestor de seguridad** (*security manager*).
- 2 Crear y exportar el objeto remoto.
- 3 Se ha de registrar la referencia a dicho objeto remoto (el stub) en el servidor de nombre (`rmiregistry`) asociándole un nombre. Esto se hace con `rebind(nombre, referencia)`.

Estos pasos se podrían incluir en el programa ppal de la clase que implementa la interfaz remota (como en el siguiente ejemplo) o en cualquier otro método de otra clase.

2. Implementar el objeto remoto(3)

Ejemplo: método main para contador remoto Contador.java

```
...

public static void main(String[] args) {
    // Instalacion del gestor de seguridad
    if (System.getSecurityManager() == null) {
        System.setSecurityManager(new SecurityManager());
    }
    try {
        String nombre = "contador";
        Creacion de una instancia de la clase
        Contador_I cont = new Contador();
        La exportamos y le damos nombre en el RMI registry
        Contador_I stub =(Contador_I)
            UnicastRemoteObject.exportObject(cont, 0);
        Registry registry = LocateRegistry.getRegistry();
        registry.rebind(nombre,stub);
        System.out.println("Contador ligado");
    } catch (Exception e) {
        System.err.println("Contador exception:");
        e.printStackTrace();
    }
}
```

3. Implementar los programas clientes

Implementar los programas clientes

- ▶ El cliente también debe instalar un gestor de seguridad para que el stub local pueda descargar la definición de una clase desde el servidor.
- ▶ El programa cliente deberá obtener la referencia del objeto remoto, consultando el servicio de enlazador, `rmiregistry`, en la máquina del servidor, usando `lookup(...)`.
- ▶ Una vez obtenida la referencia remota, el programa interactúa con el objeto remoto invocando métodos del stub usando los argumentos adecuados (como si fuera local).

3. Implementar los programas clientes (2)

Ejemplo: Programa cliente del contador remoto `Cliente.java`

```
import java.rmi.*;

public class Cliente {

    public static void main(String args[]) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
        }
        try {
            // Obteniendo una referencia a RMI registry en nodo servidor
            // El primer argumento del programa debe ser el nodo servidor
            Registry registry = LocateRegistry.getRegistry(args[0]);
            //Se invoca lookup en el registry para buscar objeto remoto
            // mediante el nombre usado en la clase Contador
            Contador_I cont = (Contador_I)(registry.lookup("contador"));
            cont.incrementa(2);
            System.out.println("VALOR="+cont.getvalor());
        } catch (Exception e) {
            System.err.println("Contador_I exception:");
            e.printStackTrace();
        }
    }
}
```

Bibliografía del tema 3.

Para más información, ejercicios, 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 las transparencias del tema 3.