1. MECANISMOS BÁSICOS EN SISTEMAS BASADOS EN PASO DE MENSAJES

1.1. INTRODUCCIÓN

En sistemas multiprocesador con memoria compartida es más fácil la programación (se usan cerrojos, semáforos, monitores), aunque la implementación es más costosa y la escabilidad hardware limitada.
Un sistema distribuido es un conjunto de procesos que no comparten memoria, pero se transmiten datos a través de una red. Esto facilita la distribución de datos y recursos y soluciona el problema de la escalabilidad y el elevado coste. Sin embargo presenta una mayor dificultad de programación.

1.2. VISTA LÓGICA Y MODELO DE EJECUCIÓN

Los procesos se comunican mediante envío y recepción de mensajes.

- En un mismo procesador pueden residir físicamente varios procesos.
- Por motivos de eficiencia, se suele alojar un único proceso en cada procesador.
- Cada interacción requiere cooperación entre 2 procesos.

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

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

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

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

1.3. PRIMITIVAS BÁSICAS DE PASO DE MENSAJES

El paso de un mensaje entre 2 procesos constituye una transferencia de una secuencia finita de bytes:

- Se leen de una variable del proceso emisor (var_orig).
- Se transfieren a través de una red de interconexión.
- Se escriben en una variable del proceso receptor (var_dest).
- Implica sincronización.
- Ambas variables son del mismo tipo.

Se utiliza lo siguiente:

- send(variable_origen, id_destino)
- receive(variable_destino, id_proceso_origen)

```
process P1 (Emisor)                 |  process P2
   var var_origen : integer;        |     var_dest : integer;
begin                               |  begin
   var_origen := Produce();         |     receive(var_dest, P1);
   send(var_origen, P2);            |     Consume(var_dest);
end                                 |  end
```

Esquemas de identificación de la comunicación

Denominación directa estática

El emisor identifica explícitamente al receptor y viceversa.
Se usan identificadores de procesos. Esto es un valor asociado a un proceso en tiempo de compilación.
Ventajas:

- No hay retardo para establecer la identificación.

Desventajas:

- Cambios en la identificación requiere recompilar el código.
- Sólo permite comunicación en pares.

Denominación directa con identificación asimétrica

El emisor identifica al receptor, pero el receptor no indica un emisor específico.
El receptor inicia una recepción, pero indica que acepta recibir de cualquier proceso emisor.
Cuando se recibe el mensaje, el receptor puede conocer el proceso emisor (puede formar parte de los metadatos del mensaje o ser un parámetro de salida de receive).

Denominación indirecta

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

```
var buzon : chanel of integer;
```

```
process P1                          | process P2
   var var_origen : integer;        |    var_dest : integer;
begin                               | begin
   var_origen := Produce();         |    receive(var_dest, buzon);
   send(var_origen, buzon);         |    Consume(var_dest);
end                                 | end
```

El uso de buzones da mayor flexibilidad, permite comunicaciones entre múltiples receptores y emisores. Hay 3 tipos:

- Canales (1 a 1).
- Puertos (muchos a 1)
- Buzones generales (muchos a muchos)

→ Un mensaje enviado a un buzón general permanece ahí hasta que se haya leído por todos los procesos receptores potenciales.

Declaración estática vs dinámica

Los identificadores de proceso suelen ser valores enteros, cada uno asociado a un proceso del programa concurrente. Se puede gestionar:
Estáticamente: En el código fuente se asocian los números enteros a los procesos.

- Muy eficiente en el tiempo.
- Cambios en la estructura del programa requieren adaptar el código.

Dinámicamente: El id numérico se calcula en tiempo de ejecución.

- Menos eficiente en tiempo, más complejo.
- El código es válido aunque cambie el número de procesos de cada tipo.

Secuencia de operaciones del emisor
Solicitud de envío (**SE**): Después de iniciar llama a send, el sistema de paso de mensajes (**SPM**) registra los identificadores de ambos procesos, y la dirección y el tamaño de la variable origen.
Inicio de lectura (**IL**): El SPM lee el primer byte de todos los que forman la variable origen.
Fin de lectura (**FL**): El SPM lee el último byte de la variable origen.
Secuencia de operaciones del receptor
Fin del registro de la solicitud de recepción (**SR**): Después de iniciar receive, el SPM registra los identificadores de procesos y la dirección y tamaño de la variable de destino.
Fin del emparejamiento (**EM**): El SPM espera hasta que se haya registrado una solicitud de envío que case con la recepción anterior y se emparejen.
Inicio de escritura(**IE**): El SPM escribe el primer byte de la variable destino.
Fin de escritura (**FE**): El SPM escribe el último byte de la variable destino.

Sincronización en el SPM para la transmisión
La transmisión de un mensaje supone sincronización:
- El emparejamiento se completa después de la solicitud de envío → SE < EM
- Antes de que se escriba el primer byte en el receptor, se debe haber comenzado la lectura en el emisor → IL < IE
- Pasa lo mismo con el último byte → FL < FE
- Por transitividad → IL < FE
- No hay orden entre SE - SR y EM - IL

Por tanto:
- Pasará un determinado intervalo de tiempo finito entre SE y FE
- Durante ese intervalo de tiempo se dice que el mesaje es un mensaje en tránsito.

Almacén temporal de mensajes en tránsito
El SPM necesita usar memoria para almacenar datos y metadatos de todos los mensajes en tránsito → Almacém temporal de datos.
Dicha memoria puede estar en el nodo emisor, receptor o intermedios.
El SPM puede detectar que no tiene suficiente memoria.

Seguridad de las operaciones de paso de mensajes
Operación segura: Se garantiza que el valor de var_origen antes del envío condiciona el valor de var_destino tras la recepción.
Operación insegura:
- Envío inseguro: Si es posible modificar var_origen entre SE y FL.
- Recepción insegura: Si es posible acceder a var_destino entre SR y FE, si lee datos antes de recibirlo totalmente o se modifica después de haberlo recibido parcialmente

Las operaciones seguras devuelven el control cuando se garantiza la seguridad. Existen 2 mecanismos: Envío y recepción síncronos, y envío asíncrono seguro.

OPERACIONES SÍNCRONAS
s_send (var_origen, id_proceso_receptor);
- No termina hasta que ocurran FL y EM.
- El fin de s_send ocurre después del inicio de receive.

receive (var_destino, id_proceso_emisor);

- No termina antes de que ocurra FE.
- El fin de receive ocurre después del inicio de s_send.

→ La operación s_send no devuelve el control hasta que el receive correspondiente sea alcanzado en el receptor. El emisor podrá hacer aserciones acerca del estado del receptor.

DESVENTAJAS:

- Fácil de implementar pero poco flexible.
- Sobrecarga por espera ociosa: adecuado si send y receive se inician +- a la vez.
- Interbloqueo: necesario alternar llamadas en intercambios.

ENVÍO ASÍNCRONO SEGURO

send (variable, id_proceso_receptor);
Inicia el envío de los datos designados y espera bloqueado hasta que se haya copiado la variable en un lugar seguro.
Devuelve el control después de FL.
Se suele usar junto con la recepción síncrona (receive).
El fin de send no depende de la actividad del receptor.
El fin de receive ocurre después del inicio de send.
VENTAJA

- send es más eficiente en tiempo que s_send, y preferible cuando el emisor no tiene que esperar la recepción.

DESVENTAJAS

- send requiere memoria para el almacenamiento temporal.
- El SPM puede tener que retrasar IL en el lado del emisor por falta de memoria.

Además, existe la posibilidad de interbloqueo. Aunque los envíos se hagan con send, las llamadas a receive siguen siendo síncronas.

Operaciones inseguras
Las operaciones seguras son menos eficientes en tiempo y en memoria. Alternativa:

- Las operaciones devuelven el control antes de que sea seguro modificar o acceder a los datos.
- Deben existir sentencias de chequeo de estado: indican si los datos pueden alterarse o acceder a ellos sin comprometer la seguridad.

PASO DE MENSAJES ASÍNCRONO INSEGURO

i_send (var_origen, id_proceso_receptor, var_resguardo);
Indica al SPM que comience una operación de envío al receptor.

- Se registra SE y acaba.
- No espera a FL ni a ninguna acción del receptor.
- var_estado permite consultar el estado del envío.

i_receive (var_destino, id_proceso_emisor, var_resguardo);
Indica al SPM que se inicie una recepción de un mensaje del emisor

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

Cuando un proceso hace i_send o i_receive puede continuar trabajando hasta que deba esperar a que termina la operación asíncrona.

wait_send (var_resguardo);
Se invoca por el proceso emisor, que se bloquea hasta que la operacion asociada a var_resguardo ha llegado a FL.

wait_recv (var_resguardo);
Igual que wait_send pero hasta FE.

Estas operaciones permiten a los procesos emisor y receptor hacer trabajo útil concurrentemente con la operación de envío o recepción:
-Mejora: El tiempo de espera ociosa se puede emplear en computación.
-Coste: Reestructuración de programas, mayor esfuerzo del programador.

Problema del productor consumidor

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

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

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

Consulta de estado

test_send (var_resguardo);
Función lógica que se invoca por el emisor.
True si el envío de var_resguardo ha llegado a FL.
test_recv (var_resguardo);
Función lógica que se invoca por el receptor.
True si el envío de var_resguardo ha llegado a FE.

Espera bloqueada de múltiples procesos
Usando las operaciones i_receive junto con las de test, se usa espera ocupada de forma que:
- Se espera un mensaje cualquiera proveniente de varios emisores.
- Tras recibir el primer mensaje, se ejecuta una acción independientemente de cual sea el emisor del msg.
- Entre la recepción del mensaje y la acción el retraso es normalmente pequeño.

Sin embargo, con lo visto no es posible cumplir estos requisitos usando espera bloqueada puesto que es inevitable seleccionar de antemano de qué emisor queremos recibir en primer lugar.

1.4. ESPERA SELECTIVA

La espera selectiva permite la espera bloqueada de múltiples emisores. En el problema del productor-consumidor debemos usar s_send puesto que con send la memoria para el almacenamiento temporal puede crecer indefinidamente. Para que al usar s_send no se introduzcan esperas largas usamos un proceso intermedio que acepte peticiones del productor y el consumidor usando espera selectiva (sino se bloquearía por turnos para esperar). Usamos un vector de datos pendientes de leer.

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

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

Síntesis general

Es una nueva sentencia compuesta por la siguiente sintaxis:

```
select
when condicion1 receive(variable1, proceso1) do
        sentencias1
when condicionn receive(variablen, proceson) do
        sentenciasn
end
```

Cada bloque que comienza en when se llama alternativa.
El texto entre when y do se llama garda de dicha alternativa.

Evaluación de las guardas

Una guarda es ejecutable si:

- La condición es verdadera.
- Tiene sentencia de entrada, el proceso origen nombra send con destino al proceso, que casa con el receive.

Una guarda es potencialmente ejecutable si:

- La condición es verdadera.
- El proceso nombrado por la sentencia de entrada no ha iniciado send..

Una guarda será no ejecutable en el resto de los casos (la condición es falsa).

Selección de una alternativa

Para ejecutar select, al inicio se selecciona una alternativa:

- Si hay guardas ejecutables con sentencia de entrada: Aquella cuyo send se inició antes.
- Si hay guardas ejecutables sin sentencia de entrada: Se escoge aleatoriamente una.
- Si hay guardas potencialmente ejecutables: Se espera bloqueado hasta que se inicie algún send.
- Si no hay ejecutables ni potencialmente ejecutables: No se selecciona ninguna.

Ejecución de una alternativa

- Si no se ha seleccionado ninguna guarda no se hace nada y finaliza la ejecución de select.
- Si se ha podido, se ejecuta el receive y se recibe el mensaje(si hay sentencia de entrada) y se ejecuta la sentencia asociada a la alternativa.

Hay que tener en cuenta que se puede producir interbloqueo.

Select con guardas indexadas

Si es necesario replicar una alternativa podemos utilizar la siguiente 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.

```
//Suma de los primeros números de cada proceso hasta llegar a 1000
process Fuente[i : 0...n-1];
   var numer: 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);
       cont : boolean := true;
       numero : integer;
begin
   while cont do begin
      cont := false;
      select
         for i:=0 to n-1 when suma[i]<1000 receive(numero, Fuente[i]) do
            suma[i] := suma[i]+numero;
            cont := true;
      end
   end
end
```

2. PARADIGMAS DE INTERACCIÓN DE PROCESOS EN PROGRAMAS DISTRIBUIDOS

2.1. INTRODUCCIÓN

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

2.2. MAESTRO - ESCLAVO

En este patrón de interacción intervienen un proceso maestro y múltiples procesos esclavos. El proceso maestro descompone un problema en pequeñas tareas que distribuye a los esclavos. Los procesos esclavos reciben un mensaje con la tarea, procesan la tarea y envían el resultado al maestro.

Cálculo del conjunto de Mandelbrot

Conjunto de puntos c en el plano complejo que no excederán cierto límite cuando se calculan realizando la siguiente interacción (inicialmente z=0 con z = a+bi)

Repetir $Z_{k+1} := Z_k^2 + c$ hasta $||z|| > 2$ o k > límite

Se asocia a cada píxel un color en función del número de interacciones(k) necesarias para su cálculo.

Conjunto solución = {píxeles que agoten interacciones límite dentro de un círculo de radio 2 centrado en el origen}

Paralelización sencilla: Cada píxel se calcula sin ninguna información del resto

Primera aproximación: Asignar un número de píxeles fijo a cada proceso esclavo y recibir resultados.

Problema: Algunos procesos esclavos tendrían más trabajo que otros.

Segunda aproximación: El maestro tiene asociada una colección de filas de píxeles. Cuando los procesos esclavos están ociosos esperan recibir una fila de píxeles. Cuando no quedan más filas, el maestro espera a que todos los procesos completen sus tareas e informa de la finalización del cálculo.

```
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 ne-1 receive(colores, Esclavo[j]) do
            if (quedan filas en la bolsa)
               then send(fila, Esclavo[i])
               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
```

2.3. ITERACIÓN SÍNCRONA

Paradigma de iteración síncrona

-En un bucle diversos procesos comienzan juntos en el inicio de cada iteración.
-La siguiente iteración no puede comenzar hasta que todos los procesos hayan acabado la previa.
-Los procesos suelen intercambiar información en cada iteración.

Transformación iterativa de un vector

Vamos a realizar un número de iteraciones de un cálculo que transforma un vector x en n reales. El esquema es el siguiente:

- El nº 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.
- Cada proceso se comunica con sus dos vecinos las entradas primera y última de su bloque (al inicio de cada iteración).
- Al inicio de cada proceso, se leen los valores iniciales de un vector compartido que se llama valores, al final se copian los resultados en dicho vector.

```
process Tarea[i: 0...p-1];
   var bloque: array[0...n/p-1] of float;
begin
   receive(bloque, Coordinador)
   for k:=0 to m do begin
      //Comunicación de valores extremos con los vecinos
      send(bloque[0],    Tarea[i-1 mod p]);
      send(bloque[n/p-1],Tarea[i+1 mod p]);
      receive(izquierda, Tarea[i-1 mod p]);
      receive(derecha,   Tarea[i+1 mod p]);

      for j:=0 to n/p-2 do begin
         tmp := bloque[j];
         bloque[j] := (izquierda - bloque[j] + bloque[j+1]) / 2;
         izquierda := tmp;
      end
      bloque[n/p-1] := (izquierda - bloque[n/p-1] + derecha) / 2;
   end
   send(bloque, Coordinador);
end
```

```
//Encargado de repartir los bloques con los valores iniciales y recibir los finales
process Coordinador;
   var inicial : array[0...n-1] of float;
       resultado : array[0...n-1] of float;
       bloque : array[0...n/-1] of float;
begin
   for i:=0 to n-1 do
      inicial[i] := (calcula valores iniciales);
   for j:=0 to p-1 do begin
      for k:=0 to n/p-1 do bloque[k] := inicial[j*n/p+k];
      send(bloque, Tarea[j]);
   end
   for j:=0 to p-1 do begin
      receive(bloque, Tarea[j]);
      for k:=0 to n/p-1 do resultado[j*n/p+k] := bloque[k];
   end
   for i:=0 to n-1 do
      print("resultado[",i,"] == ", resultado[i]);
end
```

2.4. ENCAUZAMIENTO (PIPELINING)

En este caso 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. El patrón de comunicación es muy simple ya que se establece un flujo de datos entre las tareas adyacentes en el cauce.

Cauce paralelo para filtrar una lista de enteros

Dada una serie de m primos p0,p1,.., pm-1 y una lista de n enteros, a0,a1,...,an-1, encontrar aquellos números de la lista que son múltiplos de todos los m primos (n >> m)

El proceso Etapa[i] mantiene el primo pi y chequea multiplicidad con pi.

```
process Etapa[i: 0...m-1];
   var primos: array[0...m-1] of float := {p0,p1,...,pm-1};
       izquierda : integer := 0;
begin
   while 0 <= izquierda do begin
      if i==0 then leer(izquierda);
      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 print(izquierda);
      end
   end
end
```

3. MECANISMOS DE ALTO NIVEL EN SISTEMAS DISTRIBUIDOS

3.1. INTRODUCCIÓN

Los mecanismos vistos hasta ahora presentan un nivel bajo de abstracción. Las llamadas a procedimiento remoto (RPC) y la invocación remota de métodos (RMI) están basados en el método habitual por el cual un proceso llamador hace una llamada a procedimiento:

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

En el modelo de invocación remota o llamada a procedimiento remoto (RPC) es el proceso llamado el que ejecuta el código del procedimiento:

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

En RCP el flujo de comunicación es bidireccional (petición-respuesta) y se permite que varios procesos invoquen un procedimiento gestionado por otro proceso.

3.2. EL PARADIGMA CLIENTE-SERVIDOR

Proceso servidor: Gestiona un recurso y ofrece un servicio a otros procesos para que puedan acceder al recurso.

Proceso cliente: Necesita el servicio y envía un mensaje de petición al servidor.

```
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
```

Problemas de la solución

- Si el servidor falla el cliente se queda esperando una respuesta que no llegará.
- Si un cliente no invoca el receive el servidor realiza s_send síncrono, el servidor quedará bloqueado.

El mecanismo de llamada a procedimiento remoto proporciona una solución.

3.3. LLAMADA A PROCEDIMIENTO (RCP)

La diferencia principal respecto a una llamada a procedimiento local es que el programa invoca el procedimiento (clientes) y el procedimiento invocado pueden pertenecer a máquinas diferentes del sistema distribuido.

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

- En el nodo cliente se invoca un procedimiento remoto como si se tratara de una llamada a procedimiento local.
- El representante del cliente empaqueta los datos de la llamada usando un determinado formato. Esto se suele denominar marshalling o serialización.
- El representante del cliente envía el mensaje con la petición de servicio al nodo servidor.
- El programa del cliente se quedará bloqueado esperando la respuesta.
- En el nodo servidor, el sistema operativo desbloquea el proceso servidor para que se haga cargo de la petición y el mensaje es pasado al representante del servidor.
- El representante del servidor desempaqueta los datos del mensaje y ejecuta una llamada al procedimiento local con los parámetros del mensaje.
- Una vez finalizada la llamada, el representante del servidor empaqueta los resultados en un mensaje y lo envía al cliente.
- El SO del nodo cliente desbloquea el proceso que hizo la llamada para recibir el resultado.
- El representante del cliente desempaqueta el mensaje y pasa los resultados al invocador del procedimiento.

Representación de los datos:

Puesto que los nodos pueden tener diferente hardware o SO, los mensajes se envían usando una representación intermedia y los representantes se encargan de hacer las conversiones necesarias.

Paso de parámetros:
Por valor: Basta con enviar al representante del servidor los datos del cliente.
Por referencia: El objeto referenciado debe enviarse desde el proceso cliente hacia el servidor.

3.4. JAVA REMOTE METHOD INVOCATION (RMI)

En un entorno distribuido, un objeto podría invocar métodos de un objeto localizado en un nodo o proceso diferente del llamador. Para hacerlo el proceso llamador debe proporcionar el nombre del método y los parámetros, pero además debe identificar el objeto remoto y el proceso o nodo en el que reside.

Interfaz remota y representantes

La interfaz remota especifica los métodos del objeto remoto que están accesibles para los demás objetos.
Remote Method Invocation (RMI): Acción de invocar un método de la interfaz remota de un objeto remoto. (Sigue la misma sintaxis que un objeto local).
El cliente y el servidor deben conocer la interfaz de la clase del objeto remoto. En el cliente el proceso llamador es responsable de implementar la comunicación con el servidor (stub).
En el servidor es responsable de esperar la llamada, recibir los parámetros, invocar la implementación del método, obtener los resultados y enviarlos de vuelta (skeleton).

Referencias remotas

Los stubs usan la definición de 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:
El contenido de la referencia remota es gestionado por stub y el enlazador.
Enlazador: Servicio de un sistema distribuido que registra las asignaciones de nombres a referencias remotas.

3.5. SERVICIOS WEB

Gran parte de la comunicación en internet ocurre via servicios web.
-Se usan protocolos HTTP o HTTPS en la capa de aplicación.
-Se usa codificación de datos basada en XML o JSON.
-Es posible usar protocolos complejos pero lo más frecuente es el método REST que se caracteriza porque los clientes solicitan un recurso o documento especificando su URL y el servidor responde con el recurso o notificando un error. Además cada petición es independiente de otras.

Llamadas y sincronización

Las peticiones de recursos o documentos pueden hacerse desde una aplicación cualquiera ejecutándose en el cliente aunque lo más frecuente es usar un programa Javascript ejecutándose en un navegador en el nodo cliente.
Las peticiones pueden gestionarse de forma:
Síncrona: El proceso cliente espera bloqueado la llegada de la respuesta.
Asíncrona: El proceso cliente envía la petición y continua.