Tema 3. Paso de mensajes

Rafael Jesús Segura Sánchez
Grado en Ingeniería en Informática, 2º Curso

Objetivos Generales

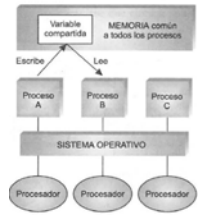
- Resolver problemas de programación concurrente en sistemas sin memoria compartida.
- Analizar las características básicas que definen el comportamiento de un modelo de comunicación.
- Definir el modelo de comunicación asíncrono mediante el uso de buzones como mecanismo de sincronización entre procesos.
- Explicar mecanismos de bajo y alto nivel para el paso de mensajes síncrono

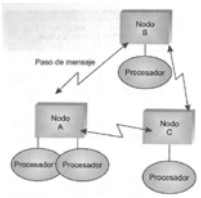
Contenidos

- Mecanismos básicos en sistemas basados en paso de mensajes.
- Mecanismos de bajo nivel de paso de mensajes:
  - Mecanismos asíncronos.
  - Mecanismos síncronos
- Mecanismos de alto nivel en sistemas distribuidos.
  - Invocación Remota (IR)
  - Llamada a procedimiento remoto (RPC)
  - Sistemas basados en objetos distribuidos

Introducción

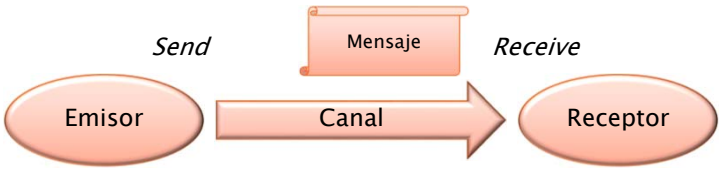
- Sistemas débilmente acoplados:
  - no comparten memoria, reloj, etc.,
  - no es posible la comunicación entre procesos mediante variables compartidas
  - No hay problemas de exclusión mutua a datos compartidos
  - Las comunicaciones se realizan empleando redes de comunicaciones menos fiables:
    - pérdida de mensajes, o desorden en la llegada
    - Heterogeneidad de los nodos:
      - plataformas hardware y software diferentes,
      - diferencias de rendimiento
      - ...





Introducción

- La comunicación y sincronización entre procesos se hace mediante el paso de mensajes:
  - operaciones explícitas de envío (send) y recepción (receive)



Introducción

- Aspectos de diseño a considerar:
  - Identificación en el proceso de comunicación
    - Denominación
    - Direccionamiento.
  - Sincronización.
  - Características del canal
    - capacidad,
    - flujo de datos,
    - ...

Identificación en el proceso de comunicación

- Forma en que el emisor indica a quién va dirigido el mensaje, y viceversa,
- Puede ser:
  - Directa
  - Indirecta

Identificación Directa

- El emisor identifica al receptor del mensaje

```
Send (A, msg)      // Envía el mensaje msg a A
Receive (B, msg)   // Recibe un mensaje msg de B
```

- Ventajas:
  - Seguridad
  - Ausencia de retardos en la identificación.
- Desventajas:
  - cualquier cambio que se produzca en las identificaciones de los procesos obligará a modificar el código asociado
  - sólo puede existir un enlace de comunicación entre emisor y receptor,
    - Imposibilidad de realizar transmisión de mensajes por diferentes canales en función de la naturaleza de la información transmitida.

Identificación Directa

- Direccionamiento directo en aplicaciones cliente/servidor
  - Dificultades:
    - Un servidor y numerosos clientes (no conocidos a priori)
    - Dificultad de asignación de los nombres a los procesos, sin que exista duplicidad de nombres
  - Direccionamiento Asimétrico
    - el emisor continúa identificando al receptor, pero el receptor no identifica a un emisor concreto

```
Send (A, msg)        // Envía el mensaje msg a A
Receive (Id, msg)    // Recibe un mensaje msg (Id identifica al emisor)
```

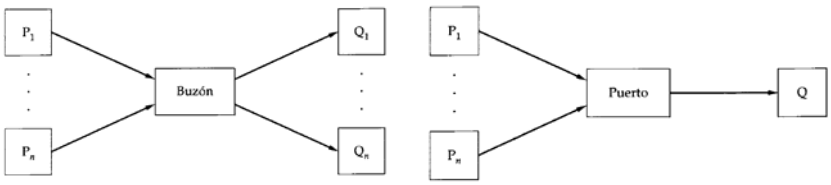
Identificación Indirecta

- No se identifica explícitamente a los procesos emisor y receptor.
  - La comunicación se realiza depositando los mensajes en un almacén intermedio (buzón) que se supone conocido por los procesos interesados en la comunicación.

```
Send (buzónA, msg)        // Envía el mensaje msg al buzón A
Receive (buzónA, msg)     // Recibe el mensaje msg del buzón A
```

Identificación Indirecta

- Un buzón puede ser utilizado por más de dos procesos e incluso entre dos procesos podemos emplear diferentes buzones.
- más flexible que el anterior,
  - permite llevar a cabo comunicaciones uno a uno, uno a muchos, muchos a uno (aplicaciones cliente/servidor) y muchos a muchos.



Muchos a muchos

Muchos a uno

Identificación Indirecta

- Asociar buzones a procesos:
  - Modo estático:
    - los procesos declaran de antemano el buzón que van a compartir
  - Modo dinámico:
    - el sistema operativo ofrece llamadas al sistema para conectarse o desconectarse de un buzón

Identificación Indirecta

- Propiedad del buzón
  - Puede ser:
    - Del proceso: los buzones existen mientras exista el proceso
    - Del Sistema Operativo
  - El propietario del buzón, es el único receptor
    - Buzón extensible mediante:
      - llamadas al sistema operativo que proporcionen dichos servicios,
      - creación dinámica de procesos por parte del proceso propietario del buzón.
        - los procesos creados por el proceso propietario también podrían recibir mensajes a través del buzón (recurso compartido)

Identificación Indirecta

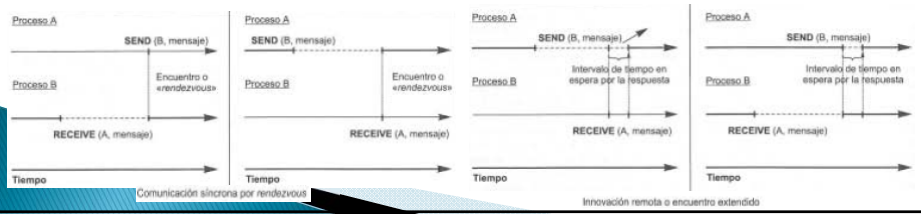
- Comunicación mediante canales:
  - las operaciones de envío y recepción se realizan a través de la especificación de un canal (enlace de comunicación),
  - El canal tiene un tipo asociado y sobre el cual sólo se pueden enviar datos del mismo tipo.
  - Además, un canal no puede ser utilizado por múltiples emisores y receptores

Sincronización

- Coincidir en el tiempo a la hora de realizar la operación de envío y recepción del mensaje
- Tipos de comunicación:
  - Asíncrona
  - Síncrona
  - Mixta:
    - El emisor pueda continuar su trabajo cuando realiza una operación de envío.
    - El receptor se bloquea hasta recibir el mensaje
    - Útil en implementación de servicios de impresión o similares.

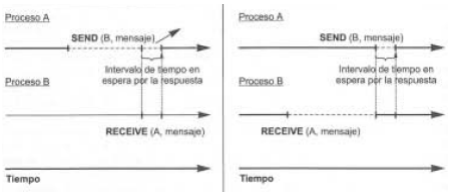
Sincronización

- Comunicación síncrona
  - coincidir en el tiempo las operaciones send y receive
  - Provocan el bloqueo de emisor y receptor
  - Tipos:
    - Rendezvous (encuentro):
      - Una vez que el receptor realiza la operación de recepción, el emisor es desbloqueado y podría continua
    - Extended Rendezvous (Invocación remota):
      - El emisor espera un mensaje de respuesta determinado



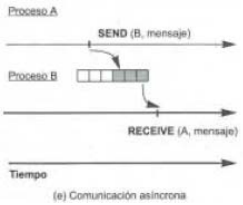
Sincronización

- Comunicación síncrona
  - Extended Rendezvous o Invocación remota:
    - El emisor no sólo le interesa esperar por la operación de recepción del receptor, sino que además espera un mensaje de respuesta determinado



Sincronización

- Comunicación Asíncrona
  - el emisor puede realizar la operación send sin que para ello sea necesario la coincidencia en el tiempo de la operación receive del receptor.
  - No bloqueante.
  - necesidad de almacenar los mensajes en buffers
  - Bloqueante si el buffer tiene tamaño finito
  - Puede simularse un esquema síncrono



(e) Comunicación asíncrona

```
PROCESO A
...
send(B, mensaje)
receive(B, reconocimiento)
...
```

```
PROCESO B
...
receive(A, mensaje)
send(A, reconocimiento)
...
```

Sincronización

- Sincronización no bloqueante
  - La comprobación de la confirmación recae en el programador
- Sincronización bloqueante:
  - más fácil de implementar, pero menos flexible.
  - el emisor tiene la confirmación de que el receptor ha recibido el mensaje,
  - Si el emisor del que esperamos el mensaje no lo envía ⇒ Bloqueo infinito del receptor
    - Solución: Añadir tiempo de espera

```
Receive (buzónA, msg, t)  // Recibe el mensaje msg del buzón A durante un tiempo t
```

    - Bloqueo condicionado solo si hay mensajes pendientes

Características del canal y de los mensajes

Tipos de enlace atendiendo al flujo de los datos

- Unidireccional:
  - la información fluye siempre en un sentido entre los dos interlocutores.
  - Única posibilidad en comunicaciones asíncronas
- Bidireccional:
  - la información podría ir en los dos sentidos.
- Cuando la comunicación es síncrona las dos opciones son posibles

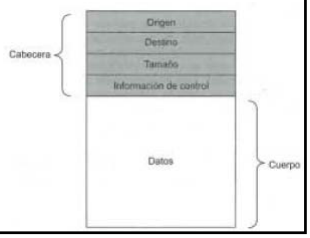
Características del canal y de los mensajes

Capacidad del canal

- posibilidad que tiene el enlace de comunicación de almacenar los mensajes enviados por el emisor cuando éstos no son recogidos de forma inmediata por el receptor
- tres tipos de canales:
  - canal de capacidad cero:
    - nos existe un almacén (buffer) donde se van almacenando los mensajes
    - Comunicación síncrona
  - canal de capacidad finita:
    - Puede producir bloqueo en emisor (buffer lleno) o receptor (buffer vacío)
  - canales de capacidad infinita:
    - Peligro de colapso si algún emisor envía mensajes constantemente

Características del canal y de los mensajes

- Tamaño de los mensajes permitidos por el canal:
  - Tamaño fijo (usualmente pocos bytes)
    - Facilidad de implementación
    - Necesidad de trocear datos ⇒Desorden de llegada
  - Tamaño variable:
    - Más flexible
    - Usar de memoria dinámica (más costosa)
    - Mensaje = cabecera + cuerpo



Características del canal y de los mensajes

Mensaje con tipo o sin tipo

- Definir el tipo de dato que va a fluir por el canal
- Puede ser
  - un tipo fijo:
    - Más fácil de implementar
    - Rigidez
    - Comprobación de errores
  - Sin tipo:
    - Requiere conocer reglas de interpretación de datos
    - Dificultad de comprobar errores.

Características del canal y de los mensajes

- Paso por copia o por referencia
  - Paso por copia:
    - efectuar una copia exacta de los datos (mensaje) que el emisor quiere enviar desde el espacio de direcciones del proceso emisor al espacio de direcciones del proceso receptor
    - Más seguro
  - Paso por referencia:
    - enviarle al receptor la dirección en el espacio de direcciones del emisor donde se encuentra el mensaje (paso por referencia).
    - exige que los procesos interlocutores compartan una memoria
    - Más eficiente.

Condiciones de error en paso de mensajes

- Posibles errores:
  - pérdida de mensajes,
  - ruidos en la transmisión.
  - Bloqueos:
    - de emisor (no llega mensaje enviado por fin proceso receptor)
    - de receptor (no recibe mensaje esperado por fin proceso emisor)
- ¿Quién detecta los errores?
  - El sistema operativo
    - Detectar error e informar al emisor
  - Los propios procesos.
  - Los protocolos de red.
- Implementación de sistemas tolerantes a fallos

Espera selectiva

- En las aplicaciones cliente/servidor, los procesos servidores ejecutan algún servicio en función de las peticiones que van recibiendo de procesos clientes.
- Los servidores no saben en qué orden se van a realizar las peticiones por parte de los clientes,
  - cuando no están atendiendo alguna solicitud, deberían estar dispuestos para poder atender cualquier petición.
- Sentencia select:
  - permite la espera selectiva en varias alternativas

Espera selectiva

- Sentencia select:
  - se evalúan todas las alternativas y se escoge una de forma aleatoria
  - En caso de que ningún proceso haya realizado una operación de envío sobre los buzones, el proceso quedará bloqueado hasta que se produzca al menos uno de estos evento
  - Algunos lenguajes incorporan prioridad en la alternativa a elegir

```
select
        receive (buzon1, msg);
        sentencias1;
or
        receive (buzon2, msg);
        sentencias2;
or
…
or
        receive (buzonN, msg);
        sentenciasN;
end select:
```

```
select
      for i=0 to N replicate
            Receive (buzon[i] msg[i]);
            Sentencias;
      or
            Receive (otro,otromsg);
            sentenciasOtro;
endselect:
```

Espera selectiva

- Sentencia select con guardas:
  - se evalúan las guardas de todas las ramas y se consideran abiertas todas aquellas alternativas cuyo guarda sea TRUE.
  - A partir de ese momento el comportamiento es exactamente igual al select.
  - No se reevalúan las guardas en caso de bloqueo

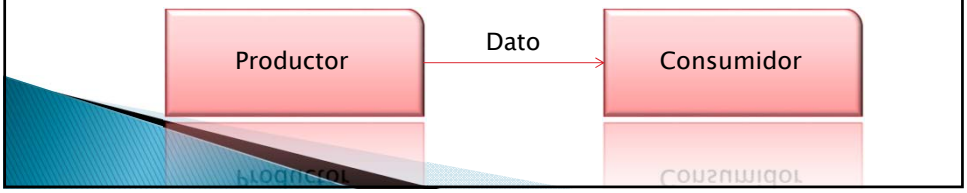
```
Select
        when condicion1 ⇒
        receive (buzon1, msg);
        sentencias1;
Or
        when condicion2 ⇒
        receive (buzon2, msg);
        Sentencias2;
or
…
or
        when condicionN ⇒
        receive (buzonN, msg);
        sentenciasN;
end select:
```

Espera selectiva

- Productor/Consumidor (1ª Solución)

```
Process Productor
Repeat
      .....
      Producir Dato
      Send (Consumidor, Dato)
      ....
Forever
```

```
Process Consumidor
Repeat
      .....
      Receive (Productor, Dato)
      Consumir dato
      ....
Forever
```



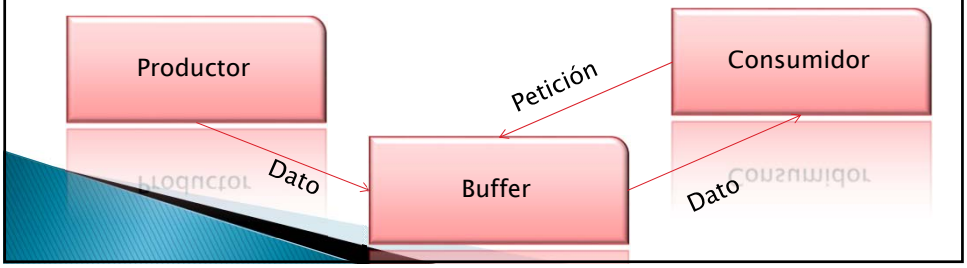
Espera selectiva

- Productor/Consumidor (2ª solución)

```
Process Productor
Repeat
      .....
      Producir Dato
      Send (Buffer, Dato)
      ....
Forever
```

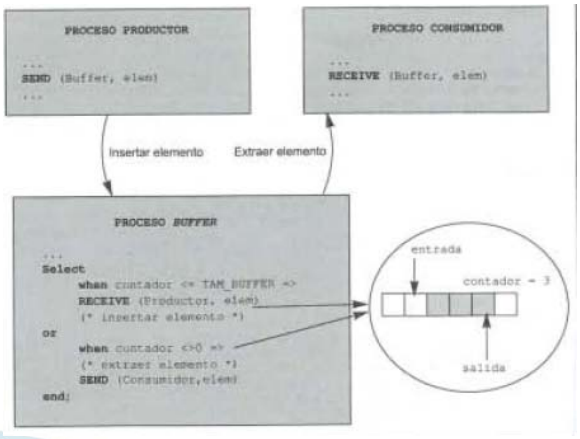
```
Process Buffer
Repeat
      .....
      Receive (Productor, dato)
      Receive (Consumidor, Petición)
      Send (Consumidor, dato)
      ....
Forever
```

```
Process Consumidor
Repeat
      .....
      Send (Buffer, petición)
      Receive (Buffer, Dato)
      Consumir dato
      ....
Forever
```



Espera selectiva

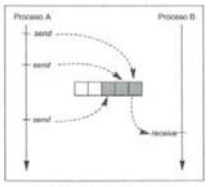
- Productor/Consumidor (3ª solución)



Paso de mensajes asíncrono

Introducción

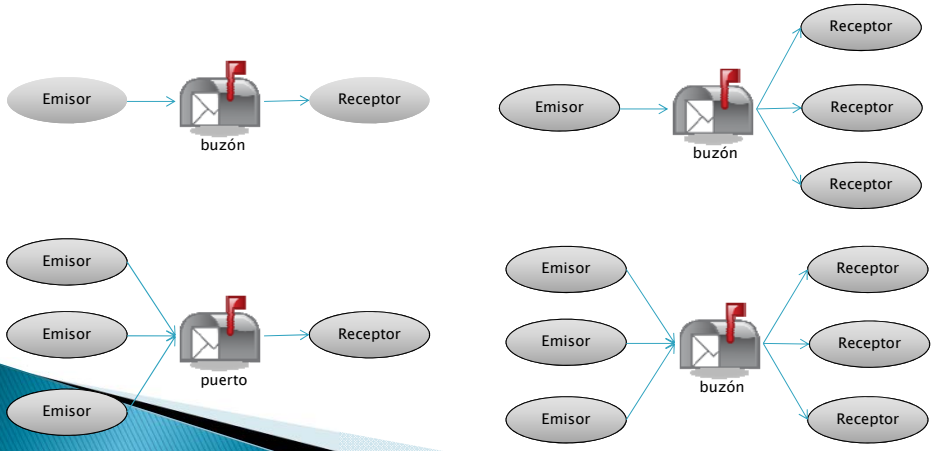
- En la comunicación asíncrona las operaciones de envío (*send*) y recepción (*receive*) no han de coincidir en el tiempo.



- Las operaciones suelen ser no bloqueantes.
  - Se precisa de mensajes de comprobación de recepción

Introducción

- Se requiere de *buffers:*
  - almacen temporal de los mensajes.



Introducción

Ventajas:

- Se producen menos cambios de contexto
- Facilidad de uso en problemas tipo productor/consumidor.
- Desacoplo entre procesos emisores y receptores

Inconvenientes:

- Dificultad si se requiere de confirmación de recepción.
- Implementación,
- Desacoplo entre procesos emisores y receptores,
- Imposibilidad de determinar el tamaño del buzón
  - uso de memoria dinámica.
  - Buzón de tamaño limitado con bloque de emisor.
  - Bloqueo sistemático de procesos receptores mediante envío masivo de mensajes a un puerto

Elementos de paso de mensajes asíncrono

Declaración del Buzón

- Nombre_buzón: mailbox [1..N] of <tipo>;
- Tamaño limitado ⇒ operaciones bloqueantes
- Gestión FIFO de mensajes
- Puede ser compartido por diferentes procesos ⇒ Permitir comunicación 1 → 1,1 →M, N →1, N→M

Envío

- Send (b, msg)
- Envía el mensaje msg al buzón b
- Se inserta al final de la cola
- Si full(b) ⇒ bloqueo del proceso

Recepción

- Receive(b, msg)
- Recibe del buzón b el mensaje msg
- Se extrae el primero que entró en el buzón
- ¿Debe ser bloqueante? NO siempre, para poder atender otras peticiones.

Empty (b):

- Comprobar si hay mensajes en el buzón
- Útil para recepción bloqueante (si el buzón está vacío, no ejecuto receive)

Elementos de paso de mensajes asíncrono

- Semántica de operaciones:
  - Las operaciones send y receive se consideran indivisibles
  - Se supone que el orden de llegada coincide con el orden de envío
  - No existen errores en los envíos:
    - Los mecanismos de seguridad recaen en la capa de comunicaciones
  - Supondremos recepción bloqueante:
    - Si fuera no bloqueante,
      - while empty (buzon) do;

Exclusión mútua mediante paso de mensajes asíncronos

- Simular semáforo binario
- Buzón con un único elemento (testigo), insertado por el proceso padre
  - Antes de entrar en sección crítica, recibir el testigo
  - Al salir de la sección crítica, dejar el testigo

```
Process Main
        send (B,testigo)
        Cobegin
                for i=1..N Pi
        Coend;
Forever
```

```
Process Pi
Repeat
        receive (B, token);
        Sección Crítica;
        send (B,token)
Forever
```

Si empty(B), se bloquea

Productor/consumidor

- Versión bloqueante en recepción

```
Process Productor;
var
      elemento: item;
begin
      repeat
            Producir (elemento);
            send (buzon, elemento);
      forever
end;
```

```
Process Consumidor;
var
      elemento: item;
begin
      repeat
            receive (buzon, elemento);
            Consumir (elemento);
      forever
end;
```

```
Process main;
var
      buzón: mailbox [1..N] of item
begin
      cobegin
            Productor();
            Consumidor();
      Coend;
end;
```

Productor/consumidor

- Versión no bloqueante en recepción

```
Process Productor;
var
      elemento: item;
begin
      repeat
            Producir (elemento);
            send (buzon, elemento);
      forever
end;
```

```
Process Consumidor;
var
      elemento: item;
begin
      Repeat
            while empty (buzón) do;
            receive (buzon, elemento);
            Consumir (elemento);
      forever
end;
```

```
Process main;
var
      buzón: mailbox[1..N]of item
begin
      cobegin
            Productor();
            Consumidor();
      Coend;
end;
```

Lectores/escritores: Prioridad en la lectura

```
Process main;
Var
        nl: integer; // Nº de lectores leyendo o esperando para leer
        mutex: mailbox of item; // buzón para exclusión mutua
        wrt: mailbox of item; // buzón para sincronización
        token: item;
Begin
        nl=0; token=cualquiervalor;
        send (mutex,token); send (wrt, token);
        cobegin
                for i=1..N lector_i;      // N es el número de lectores
                for i=1..M escritor_j;    // M es el número de escritores
        Coend;
end;
```

Lectores/escritores: Prioridad en la lectura

Incrementar lectores (en exc. mutua)
Si es el primer lector, esperar a que alguien escriba
Leer del recurso
Decrementar lectores (en exc. mutua)
Si es el último lector, desbloquear a un escritor

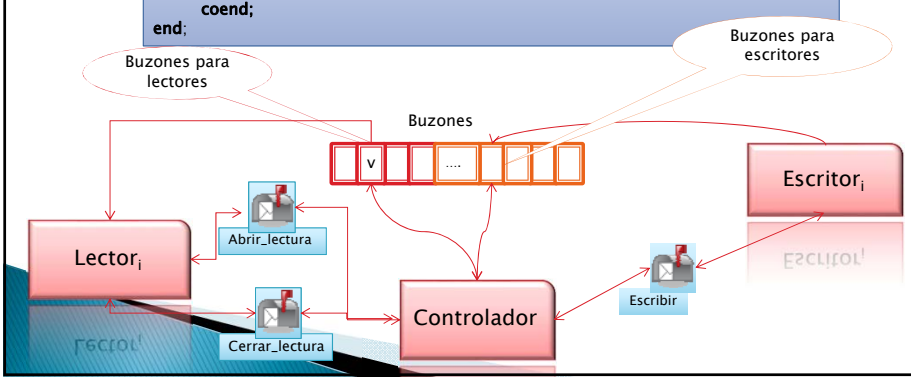
```
Process type lector;
var
        token: item;
begin
        repeat
                receive (mutex, token); // Para exclusión mutua en acceso a nl
                nl=nl+1;
                // Se impide que entre un escritor a escribir
                // El último lector dejó paso a un escritor (véase if (nl=0) )
                if (nl=1) receive (wrt, token); // Es equivalente a comprobar empty en no bloqueante
                send (mutex, token);
                Leer del recurso;
                receive (mutex, token);  // Para exclusión mutua en acceso a nl
                nl=n1-1;
                // El último lector intenta desbloquear a algún escritor
                if (nl=0) then send (wrt, token); // El if nl=1 anterior tiene sentido junto con esta sentencia
                send (mutex, token);
        forever
end;
```

Lectores/escritores: Prioridad en la lectura

```
Process type escritor;
var
        testigo: item;
begin
        repeat
                receive (wrt, testigo); // Para exclusión mutua en acceso al recurso
                Escribir en el recurso;
                send (wrt, testigo);
        forever
end;
```

Lectores/escritores: Prioridad en la escritura

```
Process main;
Var
        i,j:integer;
Begin
        cobegin
                Controlador ();
                for i=1..N lector(i);          // N es el número de lectores
                for i=1..M escritor(j+N); // M es el número de escritores
        coend;
end;
```



Lectores/escritores: Prioridad en la escritura

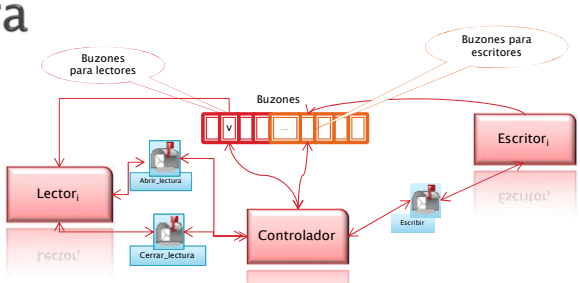
```
Process type lector (id);
var
        msg: item; // item es un tipo struct con dos campos: un int para el Pid, y el valor
begin
        Repeat
                msg:=id; // Cualquier valor
                Send (abrir_lectura, msg); //Indica que el lector id quiere leer
                Receive (buzon[id], msg); // Recibe el mensaje depositado en su buzón
                Leer del recurso
                Send  (cerrar _lectura, msg); // Indica que ha terminado de leer
        forever
end;
```

```
Process type escritor (id);
var
        msg: item;
begin
        Repeat
                msg.id=id; // Identifico a quien escribe
                msg.valor= ...// Cualquier valor
                send (escribir, msg); // Es el controlador quien escribe en el recurso
                receive (buzon[id], msg); // Para exclusión mutua en acceso al recurso
        forever
end;
```

Lectores/escritores: Prioridad en la escritura

```
Process type controlador;
Var
        aviso:item;
        nl:integer;
Begin
        nl:=0;
        repeat
                select
                        when empty (escribir) ⇒  // Para que pasen los lectores
                                Receive (abrir_lectura, aviso);
                                nl:=nl+1;
                                Send (buzon[aviso.id], aviso);
                        or
                                Receive (cerrar_lectura, aviso) ;
                                nl:=nl-1;
                        or
                                when (nl=0) ⇒           // No quedan lectores leyendo
                                        Receive (escribir, aviso);
                                        Escribir en el recurso el valor de aviso
                                        Send (buzon[aviso.id], aviso)
                endselect
        forever
end;
```

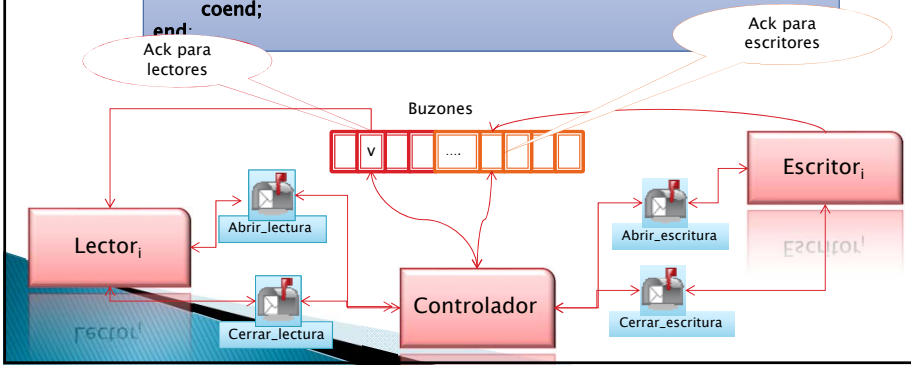
Lectores/escritores: Prioridad en la escritura



- El controlador realiza la escritura:
  - ↑ Pros:
    - ↑ Se puede escribir en exclusión mutua
    - ↑ Código más general
  - ↓ Contras:
    - ↓ El controlador debe poder acceder al recurso para escribir
    - ↓ Hay que enviar la información a escribir al controlador
- Solución: Que escriba el escritor

Lectores/escritores: Prioridad en la escritura

```
Process main;
Var
      i,j:integer;
Begin
      cobegin
            Controlador ();
            for i=1..N lector(i);          // N es el número de lectores
            for i=1..M escritor(j+N); // M es el número de escritores
      coend;
end;
```



Lectores/escritores: Prioridad en la escritura

```
Process type lector (id); // No hay dos procesos lectores con el mismo id
var
        msg: item;
begin
        Repeat
                msg:=id;
                Send (abrir_lectura, msg); //Indica que el lector i quiere leer
                Receive (buzon[id], msg); // Recibe el mensaje depositado en su buzón
                Leer del recurso
                ....
                Send  (cerrar _lectura, msg); // Indica que ha terminado de leer
        forever
end;
```

```
Process type escritor (id); // No hay dos procesos escritores con el mismo id

var
        msg: item;
begin
        Repeat
                msg.id=id;
                send (abrir_escritura, msg);  // Indicar que se desea leer
                receive (buzon[id], msg); // Para exc. mutua en acceso al recurso
                Escribir en el recurso
                Send (cerrar_escritura,msg); // Indicar que se ha terminado de escribir
        forever
end;
```

Lectores/escritores: Prioridad en la escritura

```
Process type controlador;
var
        msg: item;
        nl:integer;
        escribiendo: boolean;
Begin
        nl:=0;
        Escrtibiendo:=FALSE; // Inicialmente no hay nadie escribiendo
        repeat
                Select
                        when empty (abrir_escritura) ⇒  // Para que pasen los lectores
                                Receive (abrir_lectura, msg);
                                nl:=nl+1;
                                Send (buzon[msg.id], msg);
                        or
                                Receive (cerrar_lectura, msg) ; // Ha acabado un lector
                                nl:=nl-1;
                        or
                                when (nl=0) and not escribiendo ⇒ // No quedan lectores leyendo ni hay nadie escribiendo
                                        Receive (abrir_escritura, msg);
                                        Escribiendo:=TRUE;
                                        Send (buzon[msg.id], msg);
                        or
                                Receive (cerrar_escritura, mmsg);
                                escribiendo:=FALSE;
                endselect
        forever
End;
```

Filósofos

```
Program filosofos;
Var
      Pido_palillos: array[0 .. N-1] of buzon_filosofo;
      palillos_concedidos: array[0 .. N-1] of buzon_filosofo;
      suelto_palillos: array[0.. N-1] of buzon_flilosofo;
      palillos: array[0 .. N-1] of integer;
      i: integer;
Begin
      for i:=0 to N-1 do palillos[i]:=1;
      cobegin
            Controlador;
            for i:=0 to N-1 do Filosofo (i)
      Coend
end;
```

Filósofos

```
Process type filosofo (id);
Var
      msg: item;
Begin
      repeat
            Pensar;
            Send (pido_palillos[id], msg);
            Receive (palillos_concedidos[id], msg);
            Comer;
            Send (suelto_palillos[id], msg);
      forever
end;
```

Filósofos

```
Process Controlador;
Var
      msg: item;
      i:integer;
Begin
      repeat
            select
                  or
                        for i=0 to N replicate
                              when (palillos[i]=1) and (palillos[(i+1) mod N]=1 ⇒
                                    Receive (pido_palillos[i], msg);
                                    palillos[i]:=0; palillos[(i+1) mod N]:=0;
                                    Send (palillos_concedidos[i], msg);
                  or
                        for i=0 to N replicate
                              Receive (suelto_palillos[i], msg);
                              palillos[i]=1; palillos[(i+l) mod N]:=1;
            endselect
      forever
end;
```

Ejemplo: Aeropuerto

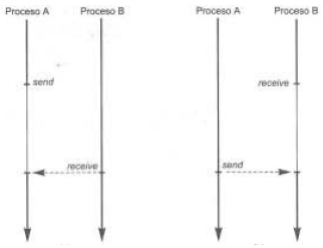
```
// El proceso Avión: Estará en un bucle infinito de despegar y aterrizar.
// Cada avión tiene su id que lo identifica, y un estado que dice si quiere despegar (true) o aterrizar (false).
Process avion (id, despegar){
        integer myID = id;          //Id = despegar;          //Estado en el de cada avión.
        boolean estado que está el avión en cada momento.

//El mensaje es una estructura de datos donde se guarda el id del avión, además de otra información adicional que se quiera mandar.
message msg;

begin
        repeat
                msg.id = myID;
                if(estado){
                        send(pedir_despegue, msg);
                        receive(puede_despegar[myID], msg);
                        estado = false;
                        send(he_despegado, msg);
                }else{
                        send(pedir_aterrizaje, msg);
                        receive(puede_aterrizar[myID]);
                        estado = true;
                        send(he_aterrizado, msg);
                }
        forever
End
```

```
/* El proceso TorreControl: Da permiso a los aviones para que despeguen o aterricen,  * gestionando las dos pistas de aterrizaje disponibles. Para cada pista, el proceso mantiene un
booleano que indica si est· libre (true) * o est· ocupada (false).                                                                                                    */
        Process TorreControl{
mailbox puede_despegar[nAviones], puede_aterrizar[nAviones]; //Buzones con los que el proceso TorreControl se comunica con los ditintos procesos  Avion:
mailbox pedir_despegue, pedir_aterrizaje, he_aterrizado, he_despegado; //Buzones con los que los distintos procesos Avion se comunican con el proceso TorreControl

boolean p_despegue = true;
boolean p_aterrizaje = true;

Begin
Repeat
                select
                                //Alternativa 1: que la pista de aterrizaje estÉ libre, y haya un aviÛn para aterrizar.
                                for 1 to nAviones replicate                     //Para todos los aviones.
                                when p_aterrizaje                               //Si la pista est· libre.
                                receive(pedir_despegue, msg);                   //Recivimos una peticiÛn.
                                p_aterrizaje = false;  //Actualizamos el valor de la pista.
                                send(puede_aterrizar[msg.id], msg);             //Avisamos al aviÛn de que puede aterrizar.
                or
                                //Alternativa 2: que un aviÛn que pidiÛ pista, ya haya aterrizado.
                                receive(he_aterrizado, msg);
                                p_aterrizaje = true;
                or
                                //Alternativa 3: que la pista de despegue estÉ libre, y haya un aviÛn para despegar.
                                for 1 to nAviones replicate                     //Para todos los aviones.
                                when p_despegue                                 //Si la pista est· libre.
                                receive(pedir_despegue, msg);                   //Recivimos una peticiÛn.
                                p_despegue = false;                             //Actualizamos el valor de la pista.
                                send(puede_despegar[msg.id], msg);              //Avisamos al aviÛn de que puede despegar.
                or
                                //Alternativa 2: que un aviÛn que pidiÛ pista, ya haya despegado.
                                receive(he_despegado, msg);
                                p_despegue = true;
                forever
end
```

Paso de mensajes síncrono

Introducción

- Primitivas de envío y recepción son bloqueantes.
- Se tiene certeza de recepción
- No se precisa buffer de mensajes
- Más fácil de implementar
- Comunicación
  - 1 a 1
  - unidireccional



Uso de canales

- Declaración
  - ch: **channel of** <tipo>;
- Operaciones
  - ch ! s // Envía s al canal
  - ch ? r // Recibe r del canal
- Canales de sincronización
  - Se usan exclusivamente para sincronización (sin tipo)
    - ch: **channel of synchronous**;
    - ch ! any // Envía mensaje de sincronización s al canal ch
    - ch ? any // Recibe mensaje de sincronización s al canal ch

Espera selectiva

```
Select
        ch ? msg1
        sentencias1;
or
        ch ? msg2;
        sentencias2;
or
…
or
        ch ? msgN
        sentenciasN;
end select:
```

```
Select
        for i=0 to N replicate
                ch[i] ? msg[i]
                Sentencias;
        or
                another ? Msg[otro];
                sentenciasOtro;
endselect:
```

Espera selectiva con guardas

- La ejecución del select comienza evaluando las guardas (no indivisible)
- A continuación, se chequean las entradas con guardas abiertas, y se elige una (indivisible)

```
Select
        when condicion1 ⇒
                ch ? msg1
                sentencias1;
or
        when condicion2 ⇒
        ch ? msg2;
        sentencias2;
or
…
or
        ch ? msgN
        sentenciasN;
end select:
```

Espera selectiva con terminate

- Funcionamiento:
  - El proceso termina si no existen llamadas pendientes y el resto de procesos que pueden realizar llamadas han terminado o se encuentran esperando a su vez en una sentencia select

```
Select
        ch ? msg1
        sentencias1;
…
or
        terminate;
end select:
```

Espera selectiva con else

- Funcionamiento:
  - Si ninguna de las alternativas se puede atender de inmediato, entonces se ejecutará la alternativa else

```
Select
        ch ? msg1
        sentencias1;
or
        ch ? msg2;
        sentencias2;
or
…
else
        sentencias;
end select:
```

Espera selectiva con timeout

- Funcionamiento:
  - Si pasados n segundos desde que se ejecutó la sentencia select no ha sido posible ejecutar alguna de las alternativas, entonces se ejecutará la alternativa con timeout

```
Select
        ch  ? msg1
        sentencias1;
or
        ch  ?  msg2;
        sentencias2;
or
…
timeout n
        sentencias;
end select:
```

Espera selectiva con prioridad

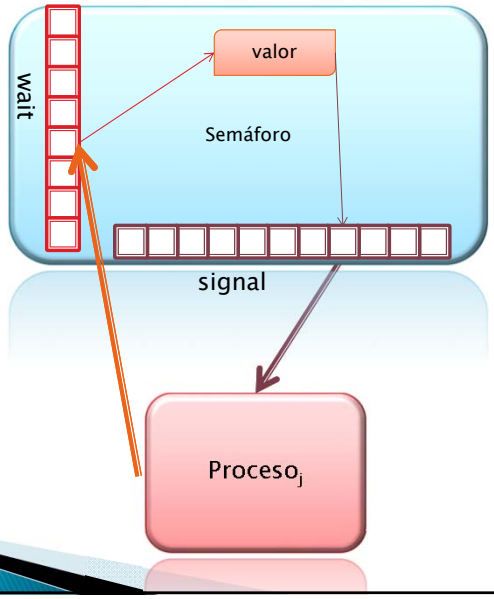
- Se eligen alternativas según una prioridad.
- Puede ser:
  - Estática: según el orden de declaración.
  - Dinámica: se establece en tiempo de ejecución.

```
Pri Select
        ch  ? msg1
        sentencias1;
or
        ch  ?  msg2;
        sentencias2;
or
…
end select:
```

Estados de procesos en select

- Si no hay alternativas válidas, ⇒ *blocked* si no hay parte else.
- Vuelve a estado *executable* si se produce una llamada en una alternativa abierta.
- El estado es "*terminated*" si en tiempo de ejecución se detecta que todos los procesos se encuentran en "termstate" o que ya están "terminated".
- Un proceso que se bloquea en un select con una alternativa de tiempo de espera es considerado "*delayed*".
  - Será *executable* cuando
    - transcurra el tiempo especificado, o
    - se produce una alternativa abierta, o
    - se produce una interrupción apropiado,
    - cualquiera de estos eventos que ocurra primero.

Exclusión mutua mediante canales



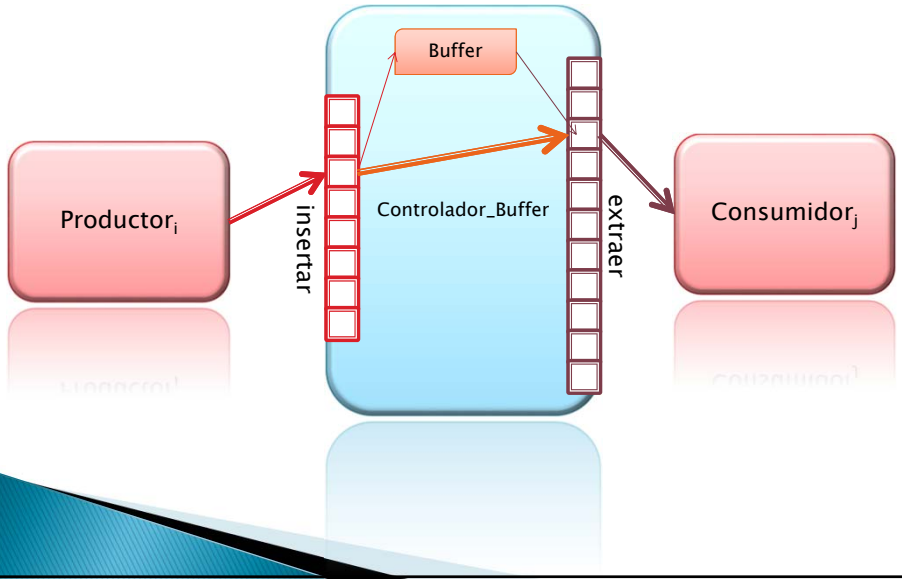
Exclusión mutua mediante canales

```
Process Semáforo (s:integer);
Var
      i,valor:integer;
begin
      valor:=s;
      repeat
            select
                  or
                        for i=0 to N replicate
                              when valor>0 ⇒
                                    wait[i] ? any;
                                    valor:=valor-1
                  or
                        for i=0 to N replicate
                              signal[i] ? any;
                              valor:=valor+1;
                  or
                        terminate;
            end
      forever
end;
```

```
Process type proceso (id:integer);
Var
Begin
      ...
      wait[id] ! any;
      SECCION CRITICA
      signal [id] ! any;
      ...
end;
```

```
Program exc_mutua;
Var
      wait: array[1..N] of channel of synchronous;
      signal: array[1..N] of channel of synchronous;
Begin
      cobegin
            semaforo (1); // El valor inicial del semáforo
            for i:=1 to N do proceso (i);
      Coend
end;
```

Productor/consumidor



Productor/consumidor

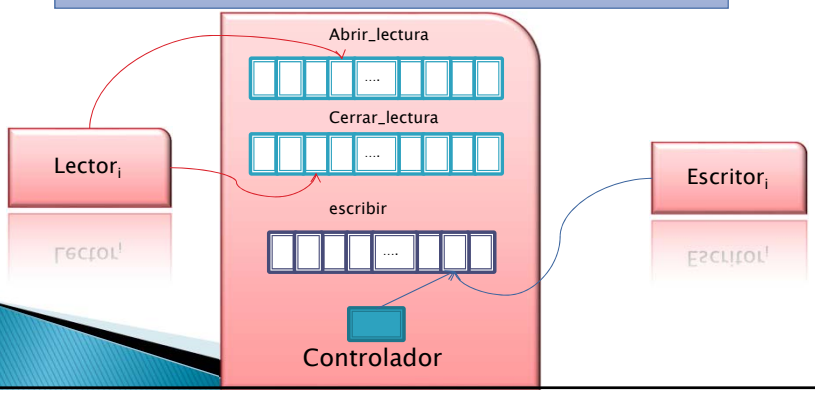
```
Process Productor (id:integer);
var
      i: <tipo>;
begin
      Repeat
            Producir (i);
            insertar[id] ! i;
            Actualizar (terminar)
      Until terminar
end;
```

```
Process Consumidor (id:integer);
var
      i:<tipo>
begin
      Repeat
            extraer[id] ? i;
            Consumir (i);
            Actualizar (terminar)
      Until terminar;
end;
```

```
Process Controlador_buffer;
Var
      buffer: array [0..SIZE] of <tipo>;
      i, j, cola, cabeza, nelem, canti, : integer;
begin
      cola:=0; cabeza:=0; nElem:=0;
      repeat
            select
                  for i=1 to NPROD replicate
                  when nElem < TAMBUFER ⇒
                        insertar [i] ? buffer[cabeza];
                        cabeza:=(cabeza+1) mod SIZE;
                        nElem:=nElem+1;
            or
                  for j:=1 to NCONS replicate
                  when nElem > 0 ⇒
                        extraer[j] ! buffer[cola];
                        cola:= (cola+1) mod SIZE;
                        nElem:= nElem-1;
            or
                  terminate;
            endselect
      forever
end;
```

Lectores/escritores: Prioridad en la lectura

```
Process main;
Begin
      cobegin
            for i=1..N lector(i);          // N es el número de lectores
            for i=1..M escrito)j+N);  // M es el número de escritores
            controlador
      Coend;
end;
```



Lectores/escritores: Prioridad en la lectura

```
Process lector (id:integer);
Var
      v: <tipo>;
begin
      Repeat
            abrir_lectura [id] ! any;
            Leer del recurso
            cerrar_lectura[id] ! any;
            Actualizar (terminar)
      Until terminar;
end;
```

```
Process escritor (id:integer);
var
      v:<tipo>
begin
      Repeat
            Producir (v)
            escritura (id) ! v;
            Actualizar (terminar)
      Until terminar;
end;
```

Lectores/escritores: Prioridad en la lectura

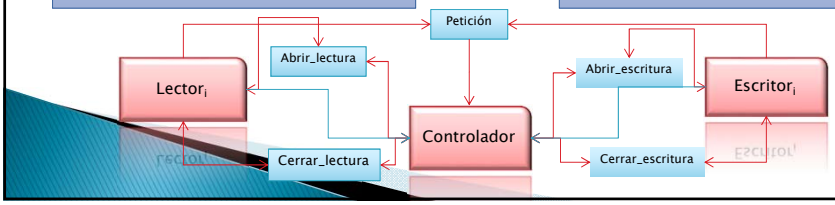
```
Process type controlador;
var
      msg: item;
      nl:integer;
Begin
      nl:=0;
      repeat
            select
                  for cont1:=1 to NUMLEC replicate
                        abrir_lectura[cont1] ? any;
                        nl:=nl+1;
                  or
                        for cont2:=1 to NUMLEC replicate
                        cerrar_lectura[cont2J ? any;
                        nl:=nl-1;
                  or
                        for cont3:=1 to NUMESC replicate
                        when nl=0 ⇒
                        escritura[cont3] ? valor;
                        varcomp:=valor;
                  or
                        terminate;
            endselect
      forever
end;
```

Lectores/escritores: Prioridad en la lectura

- Modificación de la solución para que escriba el escritor

```
Process lector (id:integer);
Var
      v: <tipo>;
begin
      Repeat
            abrir_lectura [id] ! any;
            Leer del recurso
            cerrar_lectura[id] ! any;
            Actualizar (terminar)
      Until terminar;
end;
```

```
Process escritor (id:integer);
var
      v:<tipo>
begin
      Repeat
            abrir_escritura[id] ! any;
            Escribir en el recurso
            cerrar_escritura (id) ! any;
            Actualizar (terminar)
      Until terminar;
end;
```



Lectores/escritores: Prioridad en la lectura

```
Process type controlador;
Var
      nl:integer;
Begin
      nl:=0; escribiendo:=FALSE;
      repeat
            select
                  for cont1:=1 to NUMLEC replicate
                        abrir_lectura[cont1] ? any;
                        nl:=nl+1;
                  or
                        for cont2:=1 to NUMLEC replicate
                              cerrar_lectura[cont2] ? any;
                              nl:=nl-1;
                  or
                        for cont3:=1 to NUMESC replicate
                        when (nl=0) and not escribiendo ⇒
                              abrir_escritura[cont3] ! any;
                              escribiendo:=TRUE;
                  or
                        for cont4:=1 to NUMESC replicate
                              cerrar_escritura[cont4] ! any;
                              escribiendo:=FALSE;
                  or
                        terminate;
            endselect
      forever
End;
```

Lectores/escritores: Prioridad en la escritura

- Variaciones:
  - Usar prioridad en el select
  - Mantener el número y tipo de peticiones realizadas
    - Si numPetEscr>0 y nl=0 entrarán escritores
      - Manteniendo exc. Mutua mediante variable escribiendo
- Necesitamos:
  - Un vector de canales para almacenar el tipo de peticiones
  - Un contador para conocer cuantas peticiones de escritura tenemos pendientes.

Lectores/escritores: Prioridad en la escritura

```
Process lector (id:integer);
Var
      v: <tipo>;
begin
      Repeat
            peticion[id] ? 'R' //Quiero leer
            abrir_lectura [id] ! any;
            Leer del recurso
            cerrar_lectura[id] ! any;
            Actualizar (terminar)
      Until terminar;
end;
```

```
Process escritor (id:integer);
var
      v:<tipo>
begin
      Repeat
            peticion[id] ? 'W' //Quiero escribir
            abrir_escritura[id] ! any;
            Escribir en el recurso
            cerrar_escritura (id) ! any;
            Actualizar (terminar)
      Until terminar;
end;
```

Lectores/escritores: Prioridad en la escritura

```
Process type controlador;
Var
        nl, nuymPetW,cont1,cont2,cont3,cont4,cont0:integer;
Begin
        nl:=0; escribiendo:=FALSE; numPetW:=0;
        repeat
                select
                        for cont0: =1 to NUMLEC+NUMESC replicate
                                petición[cont0] ? tipopetición;
                                if tipopetición='W'
                                then numPetW++;
                        or
                                for cont3:=1 to NUMESC replicate
                                when (nl=0) and not escribiendo ⇒
                                        abrir_escritura[cont3] ! any;
                                        escribiendo:=TRUE;
                        or
                                for cont4:=1 to NUMESC replicate
                                        cerrar_escritura[cont4] ! any;
                                        escribiendo:=FALSE;
                                        numPetW--;
                        or
                                for cont1:=1 to NUMLEC replicate
                                when numpeticionesW=0 ⇒
                                        abrir_lectura [cont1] ? any;
                                        nl: =nl+1;
                        or
                                for cont2:=1 to NUMLEC replicate
                                        cerrar_lectura[cont4] ! any;
                                        nl:=nl-1;
                        or
                                terminate;
                endselect
        forever
End;
```

Filósofos

```
Program filosofos;
Var
        pido_palillos: array[0 .. N-1] of channel of synchronous;
        suelto_palillos: array[0.. N-1] of channel of synchronous;
        i: integer;
Begin
        for i:=0 to N-1 do palillos[i]:=1;
        cobegin
                Controlador;
                for i:=0 to N-1 do Filosofos[i] (i)
        Coend
end;
```

Filósofos

```
Process type filosofo (id:integer);
Begin
      repeat
            Pensar;
            pido_palillos [id] ! any:
            Comer;
            suelto_palillos[id] ! any;
      Until terminar;
end;
```

Filósofos

```
Process Controlador;
Var
      msg: item;
      i:integer;
Begin
      for i:=0 to N-1 do palillos[i]:=1;
      repeat
            select
                  or
                        for i:=0 to N-1 replicate
                        when palillos[i]=1 and palillos[(i+1) mod N] =1 ⇒
                              pido_palillos[i] ? any;
                              palillos[i]=0;
                              palillos[(i+1) mod N]=0;
                  or
                        for i:=0 to N- 1 replicate
                              suelto_palillos[i] ? any;
                              palillos[i]= 1;
                              palillos[(i+1) mod N]:=1;
                  or
                        terminate
            endselect
      forever
end;
```

Mecanismos de comunicación de alto nivel

Rafael Jesús Segura Sánchez

Introducción

- Utilizar como abstracción la llamada a procedimiento
  - El proceso continua por la siguiente instrucción de la llamada
- Invocación Remota (RI):
  - Se invoca a un procedimiento de otro proceso
  - El proceso invocador queda bloqueado en espera de los resultados.
  - Es un esquema de comunicación síncrono y en el que el flujo de información es bidireccional.
  - Se trata de un esquema de comunicación ideal para desarrollar aplicaciones cliente/servidor.
- Llamada a Procedimiento Remoto (RPC):
  - El procedimiento invocado está en otra máquina.
  - Igual que RI pero puede ser asíncrono para llevar a cabo procesamiento paralelo.

Invocación Remota

- Esquema de comunicación síncrono también conocido como *encuentro extendido* (extended rendezvous).
  - El proceso emisor queda bloqueado esperando por una respuesta del proceso receptor.
  - Cuando esa respuesta tiene lugar, el proceso emisor continúa normalmente
- Diferencias con canales:
  - El flujo de datos puede ser bidireccional.
  - Esquema de comunicación asimétrico:
    - el emisor necesita conocer la identidad del destino pero no ocurre lo mismo con el receptor,

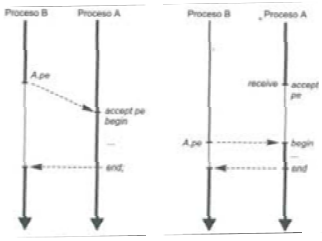
Invocación Remota

- La comunicación entre procesos tiene lugar a través de los «puntos de entrada».
  - El encuentro o cita entre dos procesos se produce como consecuencia de la llamada de un proceso a un punto de entrada declarado en otro proceso.
  - La comu… …ocesos se establece a través del paso de pa… …etros
  - Establec… …prioridades en las llamadas recibidas

Declaración del Punto de entrada

Invocación remota

```
process A;
entry pe (a: integer; var b: integer) ;
begin
end;
process B;
var
      cont: integer;
begin
      // llamada al punto de entrada pe del proceso A
      A.pe(3,cont)
end;
```



Sentencia accept

- se encuentran en el proceso donde se declaran los puntos de entrada.
- Al menos un *accept* por cada punto de entrada
- Sintaxis:
  - **accept** <id_punto_entrada>(<argumentos>) **do** <bloque>
- Funcionamiento:
  - Si se llega a *accept* y no hay llamadas pendientes se bloquea A.
  - Una vez que tiene lugar la cita, el proceso llamado ejecuta todas las sentencias incluidas en la sentencia *accept* y al terminar devuelve los datos de salida y desbloquea al proceso llamador.
  - Cada proceso continua.

Espera selectiva

- Similar a la vista en temas anteriores:
  - No se puede usar replicate
  - Usar guardas, timeout, terminate, else, prioridad.

```
Select
        accept pe1 (<args1>) do
                sentencias1;
or
        accept pe2 (<args2>) do
                sentencias2;
or
…
or
        accept peN (<argsN>) do
                sentenciasN;
end select:
```

Exclusión mutua mediante RI

```
Process type proceso (id:integer);
Var
Begin
        …
        Semaforo.wait();
        SECCION CRITICA
        Semaforo.signal()
        …
end;
```

```
Process Semáforo (s:integer);
entry wait;
entry signal;
var
        valor, cont: integer;
begin
        valor:=s;
        repeat
                select
                        when valor>0 ⇒
                        accept  wait do
                        valor:=valor-1;
                or
                        accept signal do
                        valor:=valor+1;
                or
                        terminate;
                end
        forever
end;
```

Productor/consumidor

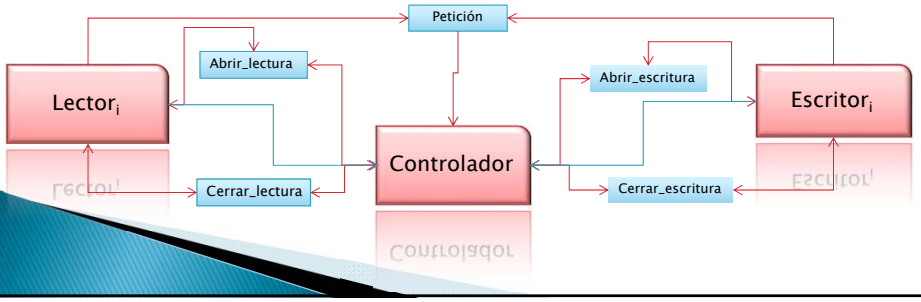
```
Process Productor (id:integer);
var
      i: <tipo>;
begin
      Repeat
            Producir (i);
            Controlador.insertar (i);
            Actualizar (terminar)
      Until terminar
end;
```

```
Process Consumidor (id:integer);
var
      i:<tipo>
begin
      Repeat
            Controlador.extraer (i);
            Consumir (i);
            Actualizar (terminar)
      Until terminar;
end;
```

```
Process Controlador_buffer;
entry extraer (var elem:<tipo>);
entry insertar (elem:<tipo>);
Var
      buffer: array [0..SIZE] of <tipo>;
      cola, frente, nelem : integer;
begin
      cola:=0; frente:=0; nElem:=0;
      repeat
            select
                  when nElem <> 0 ⇒
                        accept extraer (var elem:<tipo>) do
                              elem:=buffer[cola];
                        cola:= (cola+1) mod SIZE;
                        nElem:= nElem-1;
            or
                  when nElem <= TAMBUFER ⇒
                        accept insertar (elem:<tipo>) do
                              buffer[frente]:=elem;
                        frente:=(frente+1) mod SIZE;
                        nElem:=nElem+1;
            or
                  terminate;
            endselect
      forever
end;
```

Lectores/escritores: Prioridad en la lectura

```
Process main;
Begin
      cobegin
            for i=1..N lector_i;      // N es el número de lectores
            for i=1..M escritor_j;    // M es el número de escritores
            Controlador;
      Coend;
end;
```



Lectores/escritores: Prioridad en la lectura

```
Process lector (id:integer);
Var
      v: <tipo>;
begin
      Repeat
            Controlador.peticion(L);
            Controlador.abrir_lectura();
            Leer del recurso
            Controlador.cerrar_lectura();
            Actualizar (terminar)
      Until terminar;
end;
```

```
Process escritor (id:integer);
var
      v:<tipo>
begin
      Repeat
            Producir (v)
            Controlador.peticion(W);
            Controlador.abrir_escritura();
            Escribir en el recurso
            Controlador.cerrar_escritura();
            Actualizar (terminar)
      Until terminar;
end;
```

Lectores/escritores: Prioridad en la lectura

Se aceptan peticiones de lectura

Los lectores entran si no se está escribiendo

Los escritores entran si no hay lectores y no se está escribiendo

Al terminar de escribir, poner escribiendo a false

```
Process type controlador;
entry peticion (tipo:<enum>);
entry abrir_lectura(); entry cerrar_lectura();
entry abrir_escritura(); entry cerrar_escritura();
var
      tipo:<enum>;
      nl:integer;
begin
      nl:=0;
      repeat
            select
                  accept peticion (tipo) do
                        if tipo=L then nl:=nl+1
                  or
                        when (not escribiendo) do
                              accept abrir_lectura do null
                  or
                        accept cerrar_lectura do nl:=nl-1;
                  or
                        when nl=0 and (not escribiendo) ⇒
                        accept abrir_escritura do
                              escribiendo:=TRUE;
                  or
                        accept cerrar_escritura do
                              escribiendo:=FALSE;
                  or
                        terminate;
            endselect
      forever
end;
```

Lectores/escritores: Prioridad en la escritura

```
Process lector (id:integer);
Var
      v: <tipo>;
begin
      Repeat
            Controlador.peticion(L);
            Controlador.abrir_lectura();
            Leer del recurso
            Controlador.cerrar_lectura();
            Actualizar (terminar)
      Until terminar;
end;
```

```
Process escritor (id:integer);
var
      v:<tipo>
begin
      Repeat
            Producir (v)
            Controlador.peticion(W);
            Controlador.abrir_escritura();
            Escribir en el recurso
            Controlador.cerrar_escritura();
            Actualizar (terminar)
      Until terminar;
end;
```

Lectores/escritores: Prioridad en la escritura

Se aceptan peticiones de escritura

Los lectores entran si no hay escritores

Al terminar de escribir, decrementar el número de escritores

```
Process type controlador;
entry peticion (tipo:<enum>);
entry abrir_lectura(); entry cerrar_lectura();
entry abrir_escritura(); entry cerrar_escritura();
var
      tipo:<enum>; nl:integer;
begin
      nE:=0;
      repeat
            select
                  accept peticion (tipo:<enum>) do
                        if tipo=W then nE:=nE+1
                  or
                        when (nE=0) do ⇒
                        accept abrir_lectura do nl:=nl+1;
                  or
                        accept cerrar_lectura do nl:=nl-1;
                  or
                        when (nl=0) and (not escribiendo) ⇒
                        accept abrir_escritura do
                              escribiendo:=TRUE;
                  or
                        accept cerrar_escritura do begin
                              escribiendo:=FALSE;
                              nE:=nE-1
                        end;
                  or
                        terminate;
            endselect
      forever
end;
```

Filósofos

```
Program filosofos;
Var
      i: integer;
Begin
      for i:=0 to N-1 do palillos[i]:=1;
      cobegin
            Controlador;
            for i:=0 to N-1 do Filosofos[i] (i)
      Coend
end;
```

```
Process type filosofo (id:integer);
Begin
      repeat
            Pensar;
            Controlador.pido_palillos (id):
            Comer;
            Controlador.suelto_palillos(id);
      Until terminar;
end;
```

Filósofos

```
Process Controlador;
entry pido_palillos (int id);
entry suelto_palillos (int id);
Begin
      for i:=0 to N-1 do palillos[i]:=1;
      repeat
            select
                  accept pido_palillos (i) do
                  if (palillos[i]=1) and (palillos[(i+1) mod N]=1) then begin
                        palillos[i]=0;
                        palillos[(i+1) mod N]=0;
                  end;
            or
                  accept suelto_palillos(i) do begin
                        palillos[i]= 1;
                        palillos[(i+1) mod N]:=1;
                  end;
            or
                  terminate
            endselect
      forever
end;
```

RPC

Rafael Jesús Segura Sánchez

Introducción

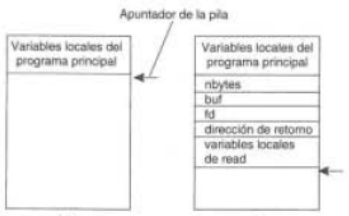
- Los métodos de paso de mensajes no ocultan la comunicación.
- Birrel y Nelson (1984) sugirieron permitir que los programas llamaran a procedimientos ubicados en otras máquinas (RPC)
- Cuando un proceso de la máquina A llama a un procedimiento de la máquina B, el proceso que llama desde A se suspende, y la ejecución del procedimiento llamado ocurre en B.
  - La información puede transportarse en los parámetros desde quien llama hasta el que es llamado, y puede regresar en el procedimiento resultante.
  - Ningún mensaje de paso es visible para el programador.

Introducción

- Problemas básicos a resolver:
  - Resolver la ejecución en espacios de dirección diferentes,
  - Resolver el paso de parámetros, lo cual puede ser complicado, en especial si las máquinas no son idénticas.
  - Por último, una o ambas máquinas pueden fallar, y cada una de las posibles fallas ocasiona diferentes problemas.

Llamada a procedimiento

- ¿Cómo funciona una llamada a procedimiento convencional?
  - Almacenar en la pila los parámetros, la dirección de retorno y las variables locales.
  - Al terminar,:
    - colocar el valor de retorno en un registro,
    - borrar la información de contexto de la pila y
    - continuar la ejecución por la dirección de retorno.
- Correspondencia entre parámetros actuales y formales:
  - Orden, número y tipo
- Paso de parámetros:
  - Por valor
  - Por referencia
  - Otros:
    - Por copia-restauración:
    - Por nombre



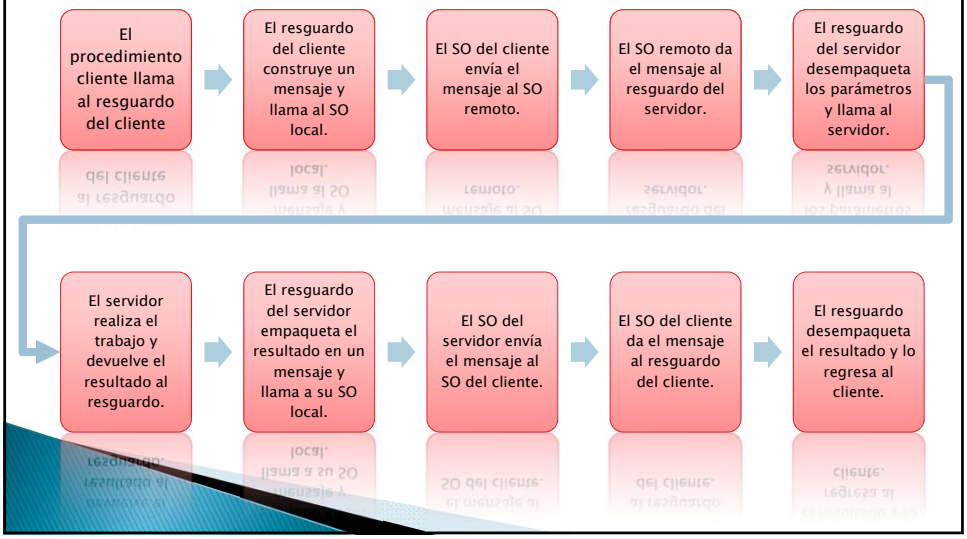
Resguardos del cliente y servidor

- En un sistema tradicional, la rutina se extrae de la biblioteca mediante el enlazador y se inserta en el programa objeto.
- La RPC logra su transparencia de manera análoga.
  - Cuando el procedimiento usado es un procedimiento remoto, se coloca en la biblioteca una versión diferente del procedimiento llamada **resguardo** (**stubs** o sustituto del cliente).
  - En cliente:
    - Invocar como llamada local
    - Empaquetar los parámetros en un mensaje y enviarlo al servidor,
    - Bloquearse hasta retorno.

Resguardo del cliente y servidor

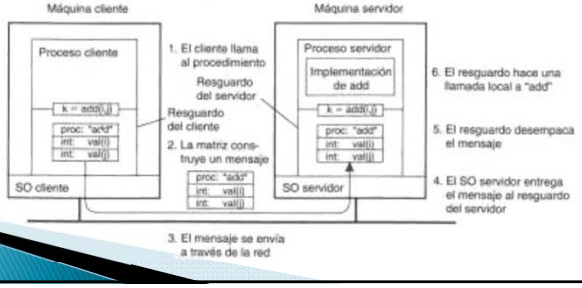
- Cuando el mensaje llega al servidor, el sistema operativo lo pasa al resguardo del servidor.
  - transforma las peticiones entrantes en llamadas a procedimientos locales.
- En servidor:
  - El proceso servidor estará escuchando en un **receive** y se habrá bloqueado esperando mensajes de entrada.
  - El resguardo del servidor desempaqueta los parámetros del mensaje y después llama al procedimiento servidor de la manera usual
  - El servidor realiza su trabajo y después, devuelve el resultado.
  - Cuando el resguardo del servidor recupera el control después de que la llamada se ha completado, empaqueta el resultado en un mensaje y llama a **send** para devolverlo al cliente.
  - El servidor hace nuevamente una llamada a **receive**, para esperar la siguiente petición entrante.
- En cliente (de nuevo)
  - el sistema operativo del cliente ve que está dirigido hacia el proceso cliente (o en realidad a la parte del resguardo del cliente, pero el sistema operativo no puede advertir la diferencia).
  - El mensaje se copia al buffer en espera y el proceso cliente se desbloquea.
  - El resguardo del cliente inspecciona el mensaje, desempaqueta el resultado, lo copia para quien la llamó, y lo devuelve en la forma usual.

Resumen



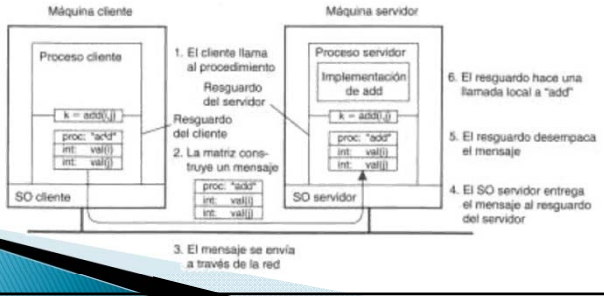
Paso de parámetros

- Paso por valor:
  - Empaquetar parámetros en un mensaje se conoce como ordenamiento de parámetros.
  - Supongamos add(i, j),
  - El resguardo del cliente toma sus dos parámetros y los coloca en un mensaje como se indica.
  - Coloca el nombre o el número del procedimiento por llamar en el mensaje porque el servidor puede soportar muchas llamadas diferentes, y se le debe informar cuál llamada es requerida.



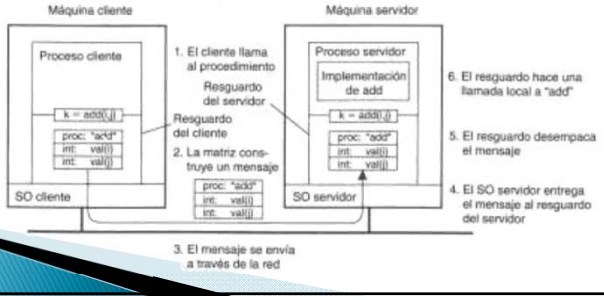
Paso de parámetros

- Paso por valor:
  - Cuando el mensaje llega al servidor, el resguardo lo examina para ver qué procedimiento se necesita y realiza entonces la llamada adecuada.
  - Si el servidor también da soporte a otros procedimientos remotos, el resguardo del servidor podría contar con una instrucción de cambio para seleccionar el procedimiento por llamar, de acuerdo con el primer campo del mensaje.
  - La llamada real del resguardo al servidor parece la llamada cliente original, excepto por que los parámetros son variables inicializadas desde el mensaje entrante.
  - Cuando el servidor ha terminado, el resguardo del servidor obtiene nuevamente el control; toma el resultado enviado por el servidor y lo empaqueta en un mensaje.
  - Este mensaje es enviado de vuelta al resguardo del cliente, que lo desempaca para extraer el resultado y regresa el valor al procedimiento cliente en espera.



Paso de parámetros

- Paso por valor:
  - Problemas:
    - La representación de los tipos de datos puede diferir:
      - Caracteres: ASCII, EBCDIC
      - Enteros: complemento a uno versus complemento a dos
      - números en coma flotante
    - Intel numera sus bytes de derecha a izquierda mientras que otras, como las Sun SPARC, los numeran de forma inversa.



Paso de parámetros

- Paso por referencia
  - Problema:
    - un puntero tiene significado sólo dentro del espacio de direcciones del proceso en el que se utiliza.
  - Solución:
    - Prohibir los punteros !!!! (Absurdo)
  - Otra solución:
    - Copiar el parámetro en el mensaje y enviarlo al servidor.
    - El resguardo del servidor usa una referencia al parámetro recibido en el mensaje (modificándola)
    - Cuando el servidor termina, el mensaje original puede enviarse de vuelta al resguardo del cliente, quien lo copia y devuelve al cliente.
    - La llamada por referencia se reemplazó con una llamada por copia-restauración.
  - Optimización posible:
    - Identificar si el parámetro es de E/S, E o S.
      - Si es de solo entrada para el servidor no hay que devolverlo
      - Si es de solo salida no es necesario mandarlo.

Paso de parámetros

- Especificación de parámetros y generación de resguardos
  - para ocultar una llamada a un procedimiento remoto es necesario que quien llama y quien es llamado coincidan en el formato de los mensajes que intercambian, y que sigan los mismos pasos cuando pasan estructuras de datos complejas.
    - Ambos lados de una RPC deben seguir el mismo protocolo, o la RPC no funcionará correctamente.
  - Deben conocerse las representaciones de los tipos (o coincidir)
  - Además, el llamador y el llamado coincidan en el intercambio real de mensajes.

Paso de parámetros

- Uso de Lenguaje de Definición de Interfaces (IDL).
  - Permite definir interfaces.
  - La interfaz especificada en un IDL, se compila en un resguardo del cliente y en un resguardo del servidor, junto con las interfaces adecuadas de tiempo de compilación o de tiempo de ejecución.
  - Simplifica el desarrollo de aplicaciones cliente-servidor basadas en RPC.
  - los sistemas middleware basados en RPC ofrecen un IDL para dar soporte al desarrollo de aplicaciones.

RPC asíncrona

- La RPC es bloqueante en el cliente.
  - Innecesario cuando no hay un resultado por devolver.
  - Ej: Transferir dinero, iniciar servicio remoto, …
- RPC asíncrona:
  - No bloquea al cliente

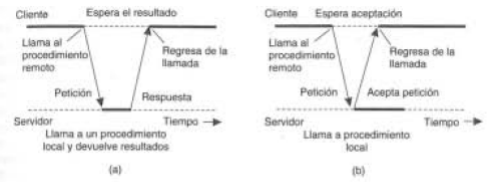


Figura 4-10. (a) Interacción entre cliente y servidor en una RPC tradicional. (b) Interacción utilizando RPC asíncronas.

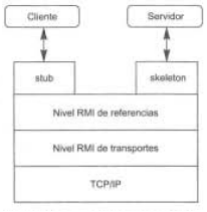
RPC asíncrona

- Útil cuando el cliente no está preparado para recibir la respuesta.
  - Se ha pedido algo al servidor, se continúa la ejecución hasta que el servidor lo haga, y éste notifica que el resultado está listo (Uso de 2 RPCs asíncronas)
  - También se conoce como RPC síncrona diferida

Figura 4-11. Cliente y servidor interactúan mediante dos RPC asíncronas.

RMI en Java

- Disponible desde JDK 1.1
- RMI (Remote Method Invocation)
  - Un objeto que proporciona servicios, (el servidor), y otros que demandan servicios, (los clientes).
  - RMI está estructurado en diversos niveles. El programador sólo se tiene que preocupar de especificar el código del cliente y el servidor.



RMI en Java

- El servidor debe especificar los servicios que ofrece. Esto se hace en una interfaz que ha de derivar de la interfaz Remote.
  - Partiendo de esta descripción se generan dos clases encargadas de sostener la comunicación entre cliente y servidor.
    - Clase stub: se encuentra en la parte cliente y ofrece la misma interfaz que el objeto servidor.
    - Clase skeleton: permanece en la máquina donde reside el servidor.
      - recibir las peticiones del stub,
      - enviárselas al objeto servidor,
      - esperar por una respuesta
      - enviársela de nuevo al stub.

Introducción

- Common Object Request Broker Architecture
- Es una arquitectura para la gestión de objetos distribuidos
  - Multiplataforma
  - Independiente del lenguaje
- Ventajas:
  - Permite que varias aplicaciones cooperen incluso si
    - Están en distintas máquinas
    - Con diferentes SO
    - Con diferentes tipos de CPU
    - Implementadas con diferentes lenguajes
  - Es un estándar

Introducción

- Diferencias con RPC
  - En RPC se lama a una función específica (con sus argumentos)
  - En ORB la llamada se hace a un método específico de un objeto:
    - Polimorfismo

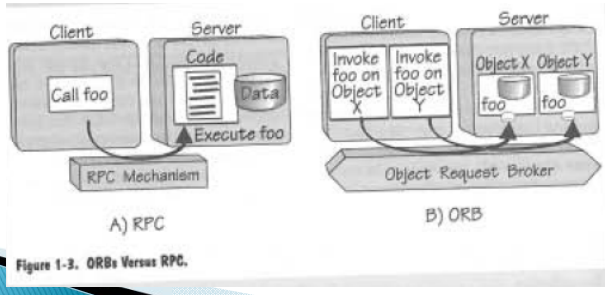
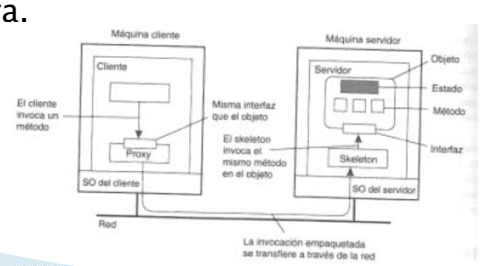


Figure 1-3. ORBs Versus RPC.

Objetos distribuidos

- Objetos:
  - Un objeto encapsula datos (estado) y operaciones sobre esos datos (métodos).
  - Se accede a los datos mediante la interfaz.
- Objeto distribuido:
  - La interfaz puede estar en una máquina y el método en otra.



Objetos distribuidos

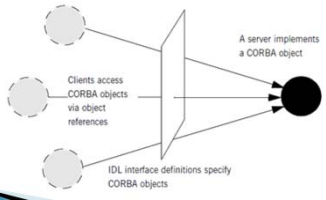
- Objetos en tiempo de compilación:
  - Se compila la definición de clase obteniendo un código que permite crear una instancia real de un objeto.
- Objetos en tiempo de ejecución:
  - Se usa un adaptador de objeto (patrón de diseño), que actúa como envolvente alrededor de la implementación con el único propósito de darle la apariencia de un objeto.
  - Los objetos se definen únicamente en función de las interfaces que implementan.
  - La implementación de una interfaz puede ser registrada en un adaptador, el cual posteriormente hace que la interfaz esté disponible para invocaciones (remotas).
  - El adaptador se encargará de que las solicitudes de invocación sean atendidas y, por tanto, de proporcionar una imagen de objetos remotos a sus clientes.

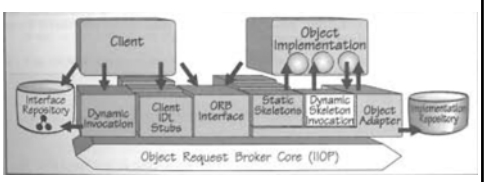
Objetos distribuidos

- Persistencia de objetos:
  - Objetos persistentes:
    - continúa existiendo aun cuando ya no esté contenido en el espacio de dirección de cualquier proceso de servidor.
    - un objeto persistente no depende de su servidor.
    - El servidor que actualmente está manejando el objeto persistente puede guardar su estado en un almacenamiento secundario y luego salir.
      - Posteriormente, un servidor recién iniciado puede leer el estado del objeto en su almacenamiento y colocarlo en su propio espacio de dirección y ocuparse de solicitudes de invocación.
  - Objeto transitorio:
    - existe sólo en tanto el servidor que lo está alojando exista.
    - en cuanto el servidor deja de funcionar, el objeto deja de existir.

Arquitectura CORBA

- Los servicios que provee un objeto se "ofrecen" a través de su interfaz
  - Definidas mediante el IDL (Interface Definition Language)
  - Los objetos se identifican por las referencias de los objetos que se indican en los IDL



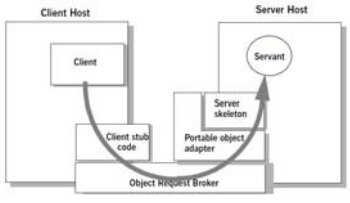


Arquitectura CORBA

- Dynamic Invocation Interface (DII)
  - puede ser utilizado cuándo no se tiene acceso a las interfaces del servidor en tiempo de compilación
  - En tiempo de ejecución los detalles de la descripción del interfaz se obtienen del Repositorio de Interfaces
- Dynamic Skeleton Interface (DSI)
  - Permite a los servidores implementarse sin skeletons compilados estáticamente
  - Definir objetos con comportamiento dinámico

Arquitectura CORBA

- Un adaptador de objeto portátil, o POA, relaciona objetos abstractos CORBA a sus implementaciones reales
- Ventajas:
  - cambiar la implementación de un objeto es transparente para el resto de la aplicación.
  - un POA permite a un servidor ser portables entre distintas aplicaciones.
- El objeto servidor (servant) puede ser estático o dinámico.
- Las políticas del POA determinan si las referencias a objetos son persistentes o transitorias,
- Un servidor puede tener uno o más POA anidados



Cliente/servidor en CORBA

- En CORBA la terminología cliente y servidor no es muy estricta
  - Servidor es la aplicación que contiene objetos
  - cliente es quién realiza las peticiones sobre dichos objetos
- Una aplicación CORBA puede jugar ambos roles, incluso al mismo tiempo.
- Los servicios se definen como objetos CORBA con sus interfaces IDL
- Servicios básicos:
  - Ciclo de Vida del Objeto:
  - Control de concurrencia,
  - Nombres
  - Transacciones
  - Control de eventos
  - …