Memoria distribuida

Arquitecturas de memoria distribuida

La arquitectura consiste en procesadores + memoria local + red de comunicaciones + mecanismos de sincronización/sincronización

Programa concurrente y programa paralelo

El procesamiento concurrente se refiere a la ejecución de múltiples tareas o procesos de manera aparentemente simultánea, pero no necesariamente en paralelo.

Puede llevarse a cabo en sistemas de un solo procesador mediante la alternancia rápida entre tareas o en sistemas con múltiples procesadores. No está restringido a una arquitectura particular de hardware ni a un número determinado de procesadores. Especificar la concurrencia implica especificar los procesos **concurrentes**, su **comunicación** y su **sincronización**.

La concurrencia sin paralelismo es la multiprogramación en un procesador en donde el tiempo de CPU es compartido entre varios procesos.

El procesamiento paralelo es la ejecución concurrente en múltiples procesadores con el objetivo principal de reducir el tiempo de ejecución.

Programa distribuido

Es un programa concurrente comunicado por mensajes. Supone la ejecución sobre una arquitectura de memoria distribuida, aunque puedan ejecutarse sobre una de memoria compartida (o híbrida).

Los procesos SOLO comparten canales (físicos o lógicos), nada más.

Primitivas de pasaje de mensajes: interfaz con el sistema de comunicaciones $\Rightarrow$ semáforos + datos + sincronización.

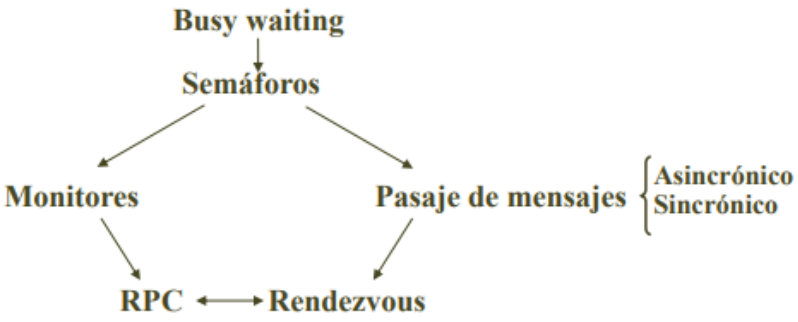
La exclusión mutua no requiere mecanismo especial y los procesos interactúan comunicándose. Solo se puede acceder a los canales mediante primitivas de envío y recepción

Patrones de comunicación

La sincronización de la comunicación interproceso depende del patrón de interacción:

- Productores y consumidores: hay filtros que reciben información de otro proceso y pueden usarla, modificarla, actualizarla y devolver los resultados.
- Clientes y servidores: muchos procesos clientes y hay algo que no pueden resolver ellos si no que necesitan que lo resuelva uno o más servidores. Este servidor lo único que hace es recibir pedidos y resolverlos. Si no hay ningún pedido ese proceso servidor no estaría procesando.
- Pares que interactúan: procesos (suelen ser idénticos) que hacen el mismo trabajo sobre conjuntos distintos de datos. Cada uno procesa y hace lo que tiene que hacer y en alguno momento tienen que interactuar entre sí.

- ***Semáforos*** ⇒ mejora respecto de ***busy waiting***.
- ***Monitores*** ⇒ combinan Exclusión Mutua implícita y señalización explícita.
- ***PM*** ⇒ extiende semáforos con datos.
- ***RPC*** y ***rendezvous*** ⇒ combina la interface procedural de monitores con PM implícito.



PMA

Los canales son cola de mensajes enviados y aun no recibidos.

***Declaración de canales* → chan *ch* (id_1 : $tipo_1$, ... , id_n : $tipo_n$)**

- **chan** entrada (char);
- **chan** acceso_disco (INT cilindro, INT bloque, INT cant, CHAR* buffer);
- **chan** resultado[n] (INT);

***Operación Send* →** un proceso agrega un mensaje al final de la cola ("ilimitada") de un canal ejecutando un *send*, que no bloquea al emisor:

send ch(expr1, ... , exprn);

***Operación Receive* →** un proceso recibe un mensaje desde un canal con *receive*, que demora ("bloquea") al receptor hasta que en el canal haya al menos un mensaje; luego toma el primero y lo almacena en variables locales:

receive ch(var_1, ... , var_n);

Las variables del receive deben tener los mismos tipos que la declaración del canal.

Receive es bloqueante.

El acceso al canal es atómico y se respeta el orden.

- empty(ch) → determina si la cola esta vacia.
    - La evaluación de empty podría ser true, y sin embargo existir un mensaje al momento de que el proceso reanuda la ejecución.

Los procesos solo comparten los canales.

- Cualquier proceso puede enviar o recibir por alguno de los canales declarados (mailbox)
- En algunos casos un canal tiene un solo receptor y muchos emisores (input port).
- Si el canal tiene un único emisor y receptor se lo denomina link: provee un "camino" entre el emisor y sus receptores.

En si PMA es mailbox.

Filtro: proceso que recibe mensajes de uno o más canales de entrada y envía mensajes a uno o más canales de salida. La salida de un filtro es función de su estado inicial y de los valores recibidos. Puede especificarse por un predicado que relacione los valores de los mensajes de salida con los de entrada

Ejemplo:

Problema: ordenar una lista de *N* números de modo ascendente. Podemos pensar en un filtro *Sort* con un canal de entrada (*N* números desordenados) y un canal de salida (*N* números ordenados).

```
Process Sort
    { receive todos los números del canal entrada;
      ordenar los números;
      send de los números ordenados por el canal OUTPUT;
    }
```

→ Solución más eficiente que la “secuencial”: red de pequeños procesos que ejecutan en paralelo e interactúan para “armar” la salida ordenada (merge network).

Dualidad entre monitores y pasaje de mensajes

Cada uno de ellos puede simular al otro.

Un monitor es un manejador de recurso. Encapsula variables permanentes que registran el estado, y provee un conjunto de procedures. Los simulamos, usando procesos servidores y pasaje de mensajes, como procesos activos en lugar de como conjuntos pasivos de procedures.

➢ Para simular Mname, usamos un proceso server Servidor.

➢ Las variables permanentes serán variables locales de Servidor.

➢ Llamado: un proceso cliente envía un mensaje a un canal de requerimiento.

➢ Luego recibe el resultado por un canal de respuesta propio

PMA es el más adecuado para el patron cliente/servidor. Permite enviar una solicitud a un solo canal y permite una respuesta a cada cliente mediante un canal de respuesta privado. Además PMA permite naturalmente que se respete un orden.

1 operación

```
chan requerimiento (int idCliente, tipos de los valores de entrada );
chan respuesta[n] (tipos de los resultados );

Process Servidor
      { int idCliente;
        declaración de variables permanentes;
        código de inicialización;
        while (true)
           { receive requerimiento (IdCliente, valores de entrada);
             cuerpo de la operación op;
             send respuesta[IdCliente] (resultados);
           }
      }

Process Cliente [i = 1 to n]
      { send requerimiento (i, argumentos);
        receive respuesta[i] (resultados);
      }
```

Múltiples operaciones

```
Process Servidor
{  int IdCliente; clase_op oper; tipo_arg args;
   tipo_result resultados;
   código de inicialización;
   while ( true)
      {  receive request(IdCliente, oper, args);
         if ( oper == op1 ) { cuerpo de op1;}
            .....
         elsif ( oper == opn ) { cuerpo de opn;}
         send respuesta[IdCliente](resultados);
      }
}
Process Cliente [i = 1 to n]
{  tipo_arg mis_args;
   tipo_result  mis_resultados;
   send request(i, opk, mis_args);
   receive respuesta[i] (mis_resultados);
}
```

Con sincronización por condición

```
type clase_op = enum(adquirir, liberar);
chan request(int idCliente, claseOp oper, int idUnidad );
chan respuesta[n] (int id_unidad);

Process Administrador_Recurso
{  int disponible = MAXUNIDADES;
   set unidades = valor inicial disponible;
   queue pendientes;
   while (true)
    {  receive request (IdCliente, oper, id_unidad);
       if (oper == adquirir)
          {  if (disponible > 0)
                {  disponible = disponible - 1;
                   remove (unidades, id_unidad);
                   send respuesta[IdCliente] (id_unidad);
                }
             else  push (pendientes, IdCliente);
          }
       else
          { if empty (pendientes)
                {  disponible= disponible + 1;
                   insert(unidades, id_unidad);
                }
             else
                {  pop (pendientes, IdCliente);
                   send respuesta[IdCliente](id_unidad);
                }
          }
    } //while
}  //process  Administrador_Recurso

Process Cliente[i = 1 to n]
{  int id_unidad;
   send request(i, adquirir, 0);
   receive respuesta[i](id_unidad);
    //Usa la unidad
   send request(i, liberar, id_unidad);
}
```

- El monitor y el Servidor muestran la dualidad entre monitores y PM: hay una correspondencia directa entre los mecanismos de ambos.
- La eficiencia de monitores o PM depende de la arquitectura física de soporte:
    - MC → Monitor
    - MD → PM

Programas con Monitores	Programas basados en PM
Variables permanentes	Variables locales del servidor
Identificadores de procedures	Canal request y tipos de operación
Llamado a procedure	send request( ); receive respuesta
Entry del monitor	receive request( )
Retorno del procedure	send respuesta( )
Sentencia wait	Salvar pedido pendiente
Sentencia signal	Recuperar/ procesar pedido pendiente
Cuerpos de los procedure	Sentencias del “case” de acuerdo a la clase de operación.

Resolución del mismo problema con sentencias de alternativa multiple. Esto se deja para casos que son muy complicados de resolver. Tiene sentido usarlo cuando un

proceso puede recibir distintos tipos de pedidos y en algunos momentos puede atender cualquiera de esos pedidos y en otros momentos puede atender solo algunos (dado por alguna condición interna de ese proceso).

```
chan pedido (int idCliente);
chan liberar (int idUnidad);
chan respuesta[n] (int idUnidad);

Process Administrador_Recurso
{   int disponible = MAXUNIDADES;
    set unidades = valor inicial disponible;
    int id_unidad, idCliente;
    while (true)
      {  if  ( (not empty(pedido) and (disponible >0) ) →
                receive pedido (idCliente);
                disponible = disponible - 1;
                remove (unidades, id_unidad);
                send respuesta[idCliente] (id_unidad);
         □ (not empty(liberar)) →
                receive liberar (id_unidad);
                 disponible= disponible + 1;
                 insert(unidades, id_unidad);
       } //if
      } //while
   }   //process  Administrador_Recurso
```

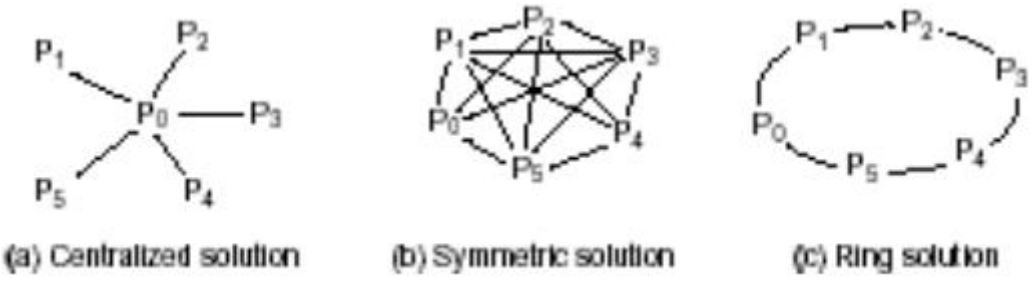
```
Process Cliente[i = 1 to n]
 {  int id_unidad;

    send pedido (i);
    receive respuesta[i](id_unidad);
     //Usa la unidad
    send liberar (id_unidad);
 }
```

Continuidad conversacional

Un cliente solicita una atención (a cualquier servidor), una vez que se establece contacto con un servidor se va a mantener una conversación interna entre los dos hasta que se termina de atender el pedido.

Pares que interactúan

Tres modelos de arquitectura: centralizado, simétrico y en anillo circular.



1. Centralizado:
- Un nodo central controla y coordina a otros nodos.
- Eficaz para la toma de decisiones centralizadas pero puede ser un punto único de fallo.
- Todos envían al procesador central y este coordina y controla los nodos.
2. Simétrico:

- Todos los nodos tienen roles y responsabilidades similares.
- Descentralizado y tolerante a fallos.
- En la arquitectura simétrica o “full conected” hay un canal entre cada par de procesos. Todos los procesos ejecutan el mismo algoritmo.

3. En anillo circular:

- Los nodos se conectan en un anillo circular.
- Cada nodo se comunica con sus vecinos más cercanos.
- Eficiente para comunicación circular o en bucle.
- Anillo donde P[i] recibe mensajes de P[i-1] y envía mensajes a P[i+1]. P[n-1] tiene como sucesor a P[0]. P[0] deberá ser algo diferente para “arrancar” el procesamiento

- ***Simétrica*** es la más corta y sencilla de programar, pero usa el mayor número de mensajes (si no hay broadcast). Pueden transmitirse en paralelo si la red soporta transmisiones concurrentes, pero el overhead de comunicación acota el speedup.
- ***Centralizada y anillo*** usan una cantidad lineal de mensajes, pero tienen distintos patrones de comunicación que llevan a distinta performance:
  - En ***centralizada***, los mensajes al coordinador se envían casi al mismo tiempo ⇒ sólo el primer *receive* del coordinador demora mucho.
  - En ***anillo***, todos los procesos son *productores y consumidores*. El último tiene que esperar a que todos los otros (uno por vez) reciban un mensaje, hacer poco cómputo, y enviar su resultado.
    Los mensajes circulan 2 veces completas por el anillo ⇒ Solución inherentemente lineal y lenta para este problema.

PMS

Los canales son de tipo link o punto a punto (1 emisor y 1 receptor).

La diferencia entre PMA y PMS es la primitiva de trasmisión Send. En PMS es bloqueante.

- El trasmisor queda esperando que el mensaje sea recibido por el receptor.
- La cola de mensajes asociada con un send sobre un canal se reduce a 1 mensaje → menos memoria.
- Naturalmente el grado de concurrencia se reduce respecto de la sincronización por PMA (los emisores se bloquean)
- Las posibilidades de deadlock son mayores en comunicación sincrónica.

- Si no quiero demora voy a tener proceso buffer.

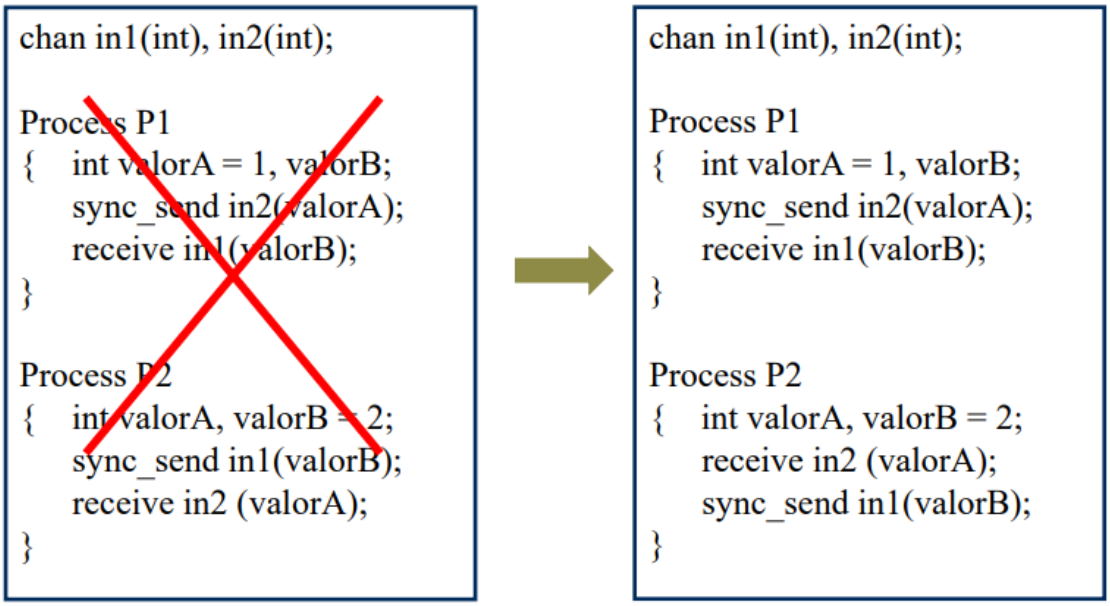
Ejemplo Productor/Consumidor:

Va al ritmo del más lento a diferencia de PMA. Hay mayor concurrencia en PMA, si quiero le mismo efecto en PMS tengo que usar buffers.

Ejemplo Cliente/Servidor.

Cuando un cliente está liberando un recurso, no habría motivos para demorarlo hasta que el servidor reciba el mensaje, pero con PMS se tiene que demorar.

Otra desventaja del PMS es la mayor probabilidad de deadlock. Hay que asegurar que todas las sentencias send y receive hagan matching.



Con PMA los procesos se suelen ejecutar a su propia velocidad porque hay buffering implícito. En PMS es necesario programar un proceso adicional para implementar buffering si es necesario.

Comunicación guardada

Hacer un AWAIT hasta que una condición sea verdadera.

Las sentencias de comunicación guardada soportan comunicación no determinística:

B; C → S;

- B puede omitirse y se asume true.
- B y C forman la guarda.
- La guarda tiene éxito si B es true y ejecutar C no causa demora.
- La guarda falla si B es falsa.
- La guarda se bloquea si B es true pero C no puede ejecutarse inmediatamente.

Las sentencias de comunicación guardadas aparecen en ***if*** y ***do***.

if B_1***; comunicación***$_1 \rightarrow S_1$***;***
• B_2***; comunicación***$_2 \rightarrow S_2$***;***
fi

Ejecución:

1. Se evalúan las guardas.
    - Si todas las guardas fallan, el if termina sin efecto.
    - Si al menos una guarda tiene éxito, se elige una de ellas (no determinísticamente).
    - Si algunas guardas se bloquean, se espera hasta que alguna de ellas tenga éxito.
2. Luego de elegir una guarda exitosa, se ejecuta la sentencia de comunicación de la guarda elegida.
3. Se ejecuta la sentencia S_i .

La ejecución del do es similar (se repite hasta que todas las guardas fallen OMG FALLEN EVANESCENCE).

En la practica la guarda es solo una recepción. En la teoría puede ser un envio o una recepción.

Paradigmas para la interacción entre procesos

3 esquemas básicos de interacción entre procesos: productor/consumidor, cliente/servidor e interacción entre pares. Estos esquemas básicos se pueden combinar de muchas maneras, dando lugar a otros paradigmas o modelos de interacción entre procesos.

Manager/Worker

Implementación distribuida del modelo Bag of Task.

"Bag of tasks" implica que varios trabajadores comparten una bolsa de tareas. Extraen tareas, las ejecutan y, en ocasiones, agregan nuevas tareas. Esto es escalable y se implementa con un proceso de gestión en un modelo Cliente/Servidor, como una forma eficiente de distribuir y equilibrar la carga de trabajo en sistemas distribuidos. Ejemplo: multiplicación de matrices ralas.

Heartbeat

Los procesos periódicamente deben intercambiar información con mecanismos tipo send/receive.

El paradigma "heartbeat" es útil para paralelizar soluciones iterativas. En un esquema "divide & conquer", se distribuye la carga entre trabajadores, y cada uno actualiza una parte de los datos. Los nuevos valores dependen de los valores mantenidos por los trabajadores o sus vecinos inmediatos. Cada paso en el proceso avanza hacia la solución. Ejemplos incluyen cálculos en cuadrículas (como imágenes) y simulaciones de fenómenos, como incendios o crecimiento biológico, utilizando autómatas celulares.

Formato general de los worker:

```
process worker [i =1 to numWorkers]
 {  declaraciones e inicializaciones locales;
    while (no terminado)
      {  send valores a los workers vecinos;
         receive valores de los workers vecinos;
         Actualizar valores locales;
      }
 }
```

Pipeline

La información recorre una serie de procesos utilizando alguna forma de receive/send.

Un pipeline es una secuencia lineal de procesos (filtros o workers) que procesan datos desde una entrada y entregan resultados a una salida. Pueden operar en paralelo en un esquema de lazo abierto (W1 en el INPUT, Wn en el OUTPUT), circular (Wn conectado a W1) o cerrado con un coordinador que maneja la retroalimentación entre Wn y W1. Esto se utiliza en procesos iterativos o cuando la aplicación requiere

múltiples pasadas por el pipeline. Un ejemplo es la multiplicación de matrices en bloques

Probe-Echo

probes (send) y echoes(receive).

La interacción entre los procesos permite recorrer grafos o árboles (o estructuras dinámicas) diseminando y juntando información

Los árboles y grafos se utilizan en aplicaciones distribuidas como búsquedas en la web, bases de datos, sistemas expertos y juegos. En arquitecturas distribuidas, los nodos se asemejan a los nodos de grafos y árboles, conectados por canales de comunicación.

El recorrido de nodos en un árbol o grafo se puede realizar secuencialmente con el algoritmo Depth-First Search (DFS), y su versión concurrente es el DFS paralelo.

El algoritmo de prueba-eco implica enviar mensajes de "prueba" desde un nodo a sus sucesores y esperar un mensaje de respuesta "eco". Se envían pruebas en paralelo a todos los sucesores. Estos algoritmos son útiles en redes donde no se conoce un número fijo de nodos activos, como en redes móviles.

Broadcast

Permiten alcanzar una información global en una arquitectura distribuida. Sirven para toma de decisiones descentralizadas.

En la mayoría de las LAN cada procesador se conecta directamente con los otros. Estas redes normalmente soportan la primitiva broadcast (broadcast ch(m)). Los mensajes broadcast se encolan en el orden de envío, pero no son atómicos, por lo que pueden ser recibidos en diferentes órdenes por otros procesos.

Broadcast se utiliza para diseminar información o resolver problemas de sincronización distribuida, como semáforos distribuidos. En este contexto, se establece un orden total de eventos de comunicación mediante el uso de relojes lógicos para garantizar la consistencia y coordinación en sistemas distribuidos.

Token Pasing

En muchos casos la arquitectura distribuida recibe una información global a través del viaje de tokens de control o datos. También permite la toma de decisiones distribuidas.

En un paradigma de interacción, se utiliza un mensaje especial llamado "token" que puede otorgar permisos o recopilar información global en arquitecturas distribuidas. Por ejemplo, se utiliza para controlar la exclusión mutua distribuida o para detectar la terminación en cómputo distribuido.

El problema de la sección crítica (SC) se presenta en programas tanto de memoria compartida como distribuida cuando se debe permitir que un único proceso acceda a un recurso compartido a la vez. Por lo general, es parte de un problema más grande, como garantizar la consistencia en sistemas de bases de datos.

Las soluciones posibles incluyen el uso de monitores activos que otorgan permisos de acceso (como locks en archivos), semáforos distribuidos (con comunicación mediante broadcast, con alto intercambio de mensajes), o sistemas de anillo de tokens descentralizados y equitativos.

Servidores Replicados

Los servidores manejan (mediante múltiples instancias) recursos compartidos tales como dispositivos o archivos.

La replicación de servidores implica tener múltiples instancias de un recurso, donde cada servidor maneja una instancia. También se utiliza para proporcionar a los clientes la ilusión de un recurso único, a pesar de que existan varias instancias en realidad.

Ejemplo: problema de los filósofos

- Modelo centralizado: los Filósofo se comunican con UN proceso Mozo que decide el acceso o no a los recursos.
- Modelo distribuido: supone 5 procesos Mozo, cada uno manejando un tenedor. Un Filósofo puede comunicarse con 2 Mozos (izquierdo y derecho), solicitando y devolviendo el recurso. Los Mozos NO se comunican entre ellos.
- Modelo descentralizada: cada Filósofo ve un único Mozo. Los Mozos se comunican entre ellos (cada uno con sus 2 vecinos) para decidir el manejo del recurso asociado a “su” Filósofo.

RPC y Rendezvous

El Pasaje de Mensajes se ajusta bien a problemas de filtros y pares que interactúan, ya que se plantea la comunicación unidireccional.

Para resolver C/S la comunicación bidireccional obliga a especificar 2 tipos de canales (requerimientos y respuestas). Además, cada cliente necesita un canal de reply distinto.

RPC (Remote Procedure Call) y Rendezvous son técnicas de comunicación y sincronización entre procesos que suponen un canal bidireccional y son ideales para programar aplicaciones C/S (RPC y Rendezvous solo sirven para C/S por lo que me anote, pero en algún momento se muestra ejemplo de pares que interactúan así que i guess se puede en ese también)

RPC y Rendezvous combinan una interfaz “tipo monitor” con operaciones exportadas a través de llamadas externas (CALL) con mensajes sincrónicos (demoran al llamador hasta que la operación llamada se termine de ejecutar y se devuelvan los resultados).

Diferencias entre RPC y Rendezvous

Difieren en la manera de servir la invocación de operaciones.

- RCP: declara un procedure para cada operación y crear un nuevo proceso (al menos conceptualmente) para manejar cada llamado (RPC porque el llamador y el cuerpo del procedure pueden estar en distintas máquinas). Para el cliente, durante la ejecución del servicio, es como si tuviera en su sitio el proceso remoto que lo sirve.
    - Soportado por Java mediante la invocación de métodos remotos (RMI).
    - Una aplicación que usa RMI tiene 3 componentes:
        - Una interfase que declara los headers para métodos remotos.
        - Una clase server que implementa la interfase.
        - Uno o más clientes que llaman a los métodos remotos.
    - El server y los clientes pueden residir en máquinas diferentes
- Rendezvous: rendezvous con un proceso existente. Un rendezvous es servido por una sentencia de Entrada (o accept) que espera una invocación, la procesa y devuelve los resultados (ADA)

RCP

- Programas se descomponen en módulos
- Es como un monitor distribuido, más todavía si tiene exclusión mutua.
- Los procesos de un módulo pueden compartir variables y llamar a procedures de ese módulo.
- Un proceso en un módulo puede comunicarse con procesos de otro módulo sólo invocando procedimientos exportados por éste.
- Los módulos tienen especificación e implementación de procedures

```
module Mname
  headers de procedures exportados (visibles)
body
  declaraciones de variables
  código de inicialización
  cuerpos de procedures exportados
  procedures y procesos locales
end
```

- Por cada llamado hay nuevo proceso que ejecuta el procedure.
- Los procesos locales son llamados background para distinguirlos de las operaciones exportadas.

Header de un procedure visible:

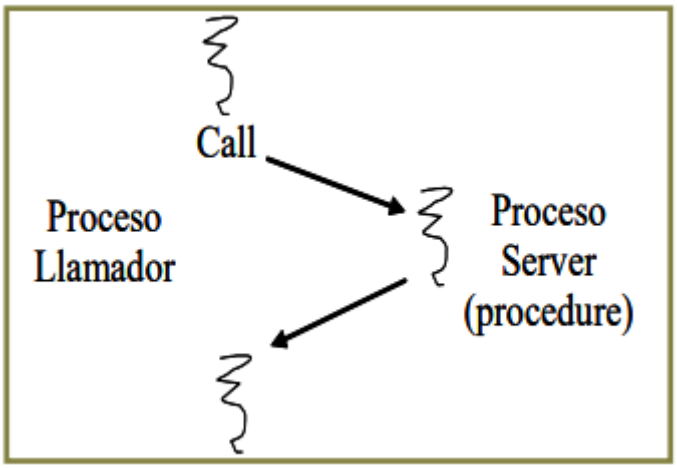
op ***opname*** (formales) [**returns** result]

El cuerpo de un procedure visible es contenido en una declaración proc:

```
proc opname(identif. formales) returns identificador resultado
  declaración de variables locales
  sentencias
end
```

Un proceso (o procedure) en un módulo llama a un procedure en otro ejecutando:

call ***Mname.opname*** **(argumentos)**



- Para un llamado local, el nombre del módulo se puede omitir.
- En un llamado intermodulo, los dos módulos pueden estar en distintos espacios: un nuevo proceso sirve el llamado, y los argumentos son pasados como mensajes entre el llamador y el proceso server.
    - Esto básicamente quiere decir que en un llamado local, los módulos se comunican directamente en la misma memoria.
- El llamador se demora mientras el proceso servidor ejecuta el cuerpo del procedure que implementa opname.
- Cuando el server vuelve de opname envía los resultados al llamador y termina. Después de recibir los resultados, el llamador sigue.
- Si el proceso llamador y el procedure están en el mismo espacio de direcciones, es posible evitar crear un nuevo proceso.
- Generalmente, un llamado será remoto por lo que se debe crear un proceso server o alocarlo de un pool preexistente
- El único rol del server es actuar en nombre del llamador
    - La sincronización entre ambos es implícita
- Se necesita que los procesos en un modulo se sincronicen.
- Dos enfoques para proveer sincronización
    - Exclusión Mutua: las variables compartidas son protegidas automáticamente contra acceso concurrente, pero es necesario programar sincronización por condición.
    - Ejecución concurrente: se necesitan mecanismos para programar exclusión mutua y sincronización por condición (cada módulo es un programa concurrente). Se puede usar cualquier método ya descripto (semáforos, monitores, o incluso rendezvous).

- Es más general asumir que los procesos se pueden ejecutar concurrentemente

Ejemplo Cliente/Servidor

- Módulo que brinda servicios de *timing* a procesos cliente en otros módulos.
- Dos operaciones visibles: ***get_time*** y ***delay(interval)***
- Un proceso interno que continuamente inicia un ***timer*** por hardware, luego incrementa el tiempo al ocurrir la interrupción de ***timer***.

```
module TimeServer
    op get_time( ) returns INT;
    op delay(INT interval, INT myid);
  body
    INT tod = 0;
    SEM m= 1;
    SEM d[n] = ([n] 0);
    QUEUE of (INT waketime, INT id's) napQ;

  proc get_time ( ) returns time
    {  time := tod; }

  proc delay(interval, myid)
    {  INT waketime = tod + interval;
       P(m);
       insert ((waketime, myid) napQ);
       V(m);
       P(d[myid]);
    }

Process Clock
  { Inicia timer por hardware;
    WHILE (true)
      { Esperar interrupción,
        luego rearrancar timer;
        tod := tod + 1;
        P(m);
        WHILE tod ≥ min(waketime, napQ)
            { remove ((waketime, id), napQ);
              V(d[id]);
            }
        V(m);
      }
  }

end TimeServer;
```

- Múltiples clientes pueden llamar a *get_time* y a *delay* a la vez ⇒ múltiples procesos "servidores" estarían atendiendo los llamados concurrentemente.
- Los pedidos de *get_time* se pueden atender concurrentemente porque sólo significan leer la variable ***tod***.
- Pero, *delay* y *clock* necesitan ejecutarse con Exclusión Mutua porque manipulan ***napQ***, la cola de procesos cliente "durmiendo".
- El valor de ***myid*** en *delay* se supone un entero único entre 0 y n-1. Se usa para indicar el semáforo privado sobre el cual está esperando un cliente.

Ejemplo: Pares que interactúan

➢ Si dos procesos de diferentes módulos deben intercambiar valores, cada módulo debe exportar un procedimiento que el otro módulo llamará.

```
module Intercambio [i = 1 to 2]
    op depositar(int);
body
    int otrovalor;
    sem listo = 0;
 proc depositar(otro)
    { otrovalor = otro;
      V(listo);
    }
 process Worker
    {  int mivalor;
       call Intercambio[3-i].depositar(mivalor);
       P(listo); .....
    }
end Intercambio
```

Rendezvous

RPC solo brinda mecanismo de comunicación intermódulo. Dentro de un módulo es necesario programar la sincronización.

Rendezvous combina comunicación y sincronización:

- Un proceso cliente invoca una operación por medio de un call, pero esta operación es servida por un proceso existente en lugar de por uno nuevo.
- Un proceso servidor usa una sentencia de entrada para esperar por un call y actuar.
- Las operaciones se atienden una por vez más que concurrentemente.

La especificación de un módulo contiene declaraciones de los ***headers*** de las operaciones exportadas, pero el cuerpo consta de un único proceso que sirve operaciones.

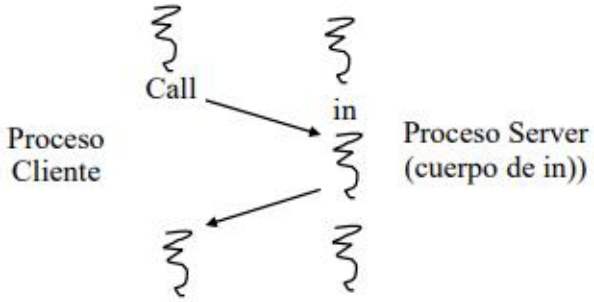
Si un módulo exporta ***opname***, el proceso server en el módulo realiza ***rendezvous*** con un llamador de ***opname*** ejecutando una ***sentencia de entrada***:

in *opname* (parámetros formales) → **S; ni**

Las partes entre ***in*** y ***ni*** se llaman ***operación guardada***.

Una sentencia de entrada demora al proceso server hasta que haya al menos un llamado pendiente de ***opname***; luego elige el llamado pendiente más viejo, copia los argumentos en los parámetros formales, ejecuta **S** y finalmente retorna los parámetros de resultado al llamador. Luego, ambos procesos pueden continuar.

A diferencia de RPC el server es un proceso activo.



Combinando comunicación guardada con rendezvous:

in op_1 ($formales_1$) **and B_1 by e_1 → S_1;**
□ ...
□ op_n ($formales_n$) **and B_n by e_n → S_n;**
ni

- Los B_i son ***expresiones de sincronización*** opcionales.
- Los e_i son ***expresiones de scheduling*** opcionales.

} Pueden referenciar a los parámetros formales.

Ahora van unos lindos ejemplos que puede que lea o no cuando estudie:

Buffer Ilimitado

```
module BufferLimitado
   op depositar (typeT), retirar (OUT typeT);
body
   process Buffer
     {  queue buf;
        int cantidad = 0;

        while (true)
          {  in depositar (item) and cantidad < n →  push (buf, item);
                                                     cantidad = cantidad + 1;
           □ retirar (OUT item) and cantidad > 0 →  pop (buf, item);
                                                     cantidad = cantidad - 1;
             ni
          }
     }
end BufferLimitado
```

Filósofos Centralizado

```
module Mesa
   op tomar(int), dejar(int);
body
   process Mozo
      { bool comiendo[5] =([5] false);
        while (true)
             in tomar(i) and not (comiendo[izq(i)] or comiendo[der(i)])  → comiendo[i] = true;
             □ dejar(i) → comiendo[i] = false;
             ni
      }
end Mesa

module Persona [i = 0 to 4]
Body
  process Filosofo
     { while (true)
         {  call Mesa.tomar(i);
            come;
            call Mesa.dejar(i);
            piensa;
         }
  }
```

Time server

A diferencia del ejemplo visto para RPC, *waketime* hace referencia a la hora que debe despertarse.

```
module TimeServer
   op get_time (OUT int);
   op delay (int);
   op tick ();

body TimeServer
   process Timer
      { int tod = 0;
        while (true)
             in get_time (OUT time) → time = tod;
             □ delay (waketime) and waketime <= tod  by waketime → skip;
             □ tick () → tod = tod + 1; reiniciar timer;
             ni
      }
end TimeServer
```

Alocador SJN

```
module Alocador_SJN
   op pedir(int), liberar();

body
   process SJN
          {  bool libre = true;
             while (true)
                in pedir (tiempo) and libre by tiempo → libre = false;
                □ liberar ( ) → libre = true;
                ni
           }
end SJN_Allocator
```

ADA

- Desarrollado por el Departamento de Defensa de USA
- tasks (tareas) que pueden ejecutar independientemente y contienen primitivas de sincronización.
- puntos de invocación (entrada) → entrys especificados en la parte visible (header de la tarea).
- Una tarea puede aceptar la comunicación con otro proceso mediante accept

- Se puede declarar un type task, y luego crear instancias de procesos (tareas) identificado con dicho tipo
- Rendezvous es el principal mecanismo de sincronización en Ada y también es el mecanismo de comunicación primario

➢ La forma más común de especificación de task es:

```
TASK nombre IS
    declaraciones de ENTRYs
end;
```

➢ La forma más común de cuerpo de task es:

```
TASK BODY nombre IS
    declaraciones locales
BEGIN
    sentencias
END nombre;
```

➢ Una especificación de TASK define una única tarea.

➢ Una instancia del correspondiente ***task body*** se crea en el bloque en el cual se declara el TASK.

Entry:

- Declaración de ***entry simples y familia de entry*** (parámetros IN, OUT y IN OUT).
- ***Entry call.*** La ejecución demora al llamador hasta que la operación E terminó (o abortó o alcanzó una excepción). → **Tarea.entry (parámetros)**
- ***Entry call condicional***:

```
select entry call;
       sentencias adicionales;
else
       sentencias;
end select;
```

- ***Entry call temporal:***

```
select entry call;
       sentencias adicionales;
or delay tiempo
       sentencias;
end select;
```

La tarea que declara un entry sirve llamados al entry con *accept*:

accept *nombre* **(parámetros formales) do** sentencias **end** ***nombre*;**

Demora la tarea hasta que haya una invocación, copia los parámetros reales en los parámetros formales, y ejecuta las sentencias. Cuando termina, los parámetros formales de salida son copiados a los parámetros reales. Luego ambos procesos continúan.

La ***sentencia wait selectiva*** soporta comunicación guardada.

select *when* $B_1 \Rightarrow$ ***accept*** E_1; sentencias$_1$
or ...
or ***when*** $B_n \Rightarrow$ ***accept*** E_n; sentencias$_n$
end select;

- Cada línea se llama ***alternativa***. Las cláusulas ***when*** son opcionales.
- Puede contener una alternativa ***else, or delay, or terminate.***
- Uso de atributos del entry: ***count, calleable.***

Ejemplo de arriba en ADA (los otros no los pongo porque ya es mucho para algo que voy a pasar de largo).

Alocador SJN

```
PROCEDURE SchedulerSJN IS
Task Alocador_SJN is
        entry pedir (tiempo, id: IN integer);
        entry liberar;
End Alocador_SJN ;

Task Type Cliente Is
    entry Identificar (identificacion: IN integer);
    entry usar;
End Cliente;
ArrClientes: array (1..C) of Cliente;
Task Body Cliente Is
    id: integer;   tiempo: integer;
BEGIN
    ACCEPT Identificar (identificacion : IN integer) do id := identificación;  End Identificar;
    loop
        //trabaja y determna el valor de tiempo
        Alocador_SJN.pedir(id, tiempo);
        Accept usar;
        //Usa el recurso
        Alocador_SJN.liberar;
    end loop;
End Cliente;
```

```
Task Body Alocador_SJN is
      libre: boolean := true;
      espera: colaOrdenada;
      tiempo, aux: integer;
Begin
   loop
      aux := -1;
      select
          accept Pedir (tiempo, id: IN integer) do
                if (libre) then  libre:= false;  aux := id;
                else  agregar(espera, (id, tiempo));  end if;
          end Pedir;
      or accept  liberar;
          if (empty (espera)) then libre := true;
          else sacar(espera, (aux, tiempo);  end if;
      end select;
      if (aux <> -1) then ArrClientes(aux).usar;  end if;
   end loop;
End Alocador_SJN ;

BEGIN
     for i in 1..C loop
          ArrClientes(i).identificacion(i);
     end loop;
END SchedulerSJN;
```