

6- Sincronización y comunicación basada en mensajes

1. Introducción

El **paso de mensajes** es una alternativa al uso de variables compartidas, ya que éstas presentan algunos inconvenientes como la necesidad de una zona de memoria compartida para todos los hilos y procesos que tengan que acceder a la misma información. Además, no son una solución genérica, ya que no son adecuadas en sistemas distribuidos o donde los procesos residan en distintas máquinas.

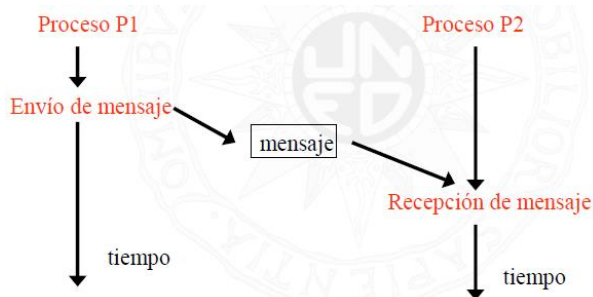
En cambio, con el paso de mensajes se usa una única construcción que varía según el modelo de lenguaje. La semántica de la comunicación basada en mensajes se define sobre tres cuestiones:

- Modelo de sincronización: hay tres modelos distintos y, a diferencia del uso de variables compartidas, sí hay una comunicación entre procesos en lugar de un acceso a una zona de memoria compartida sin saber si hay otros hilos o procesos que hagan uso de las mismas.
- Método de nombrado de procesos.
- Estructura del mensaje.

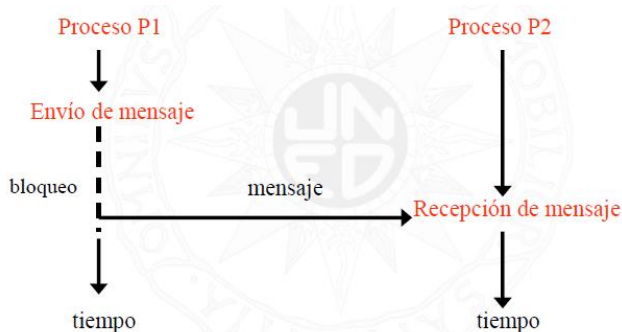
2. Sincronización de procesos

Las variaciones en el modelo de sincronización dependen de la semántica asociada al envío. Pueden darse 3 tipos de comunicaciones:

- **Asíncrona (sin espera):** el emisor envía un mensaje al receptor y continua con sus acciones inmediatamente después de enviar el mensaje. Por igual, el receptor recibe el mensaje y continua con su tarea. Este tipo de comunicación se usa para informar de sucesos que ya han tenido lugar

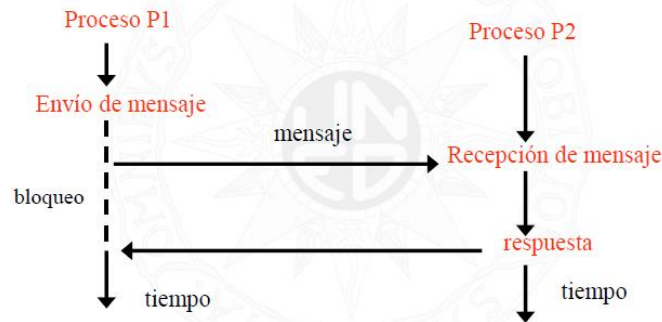


- **Síncrona:** el emisor envía el mensaje, pero no continúa su actividad hasta que no llega la señal de recepción por parte del segundo proceso, existiendo por tanto un periodo de tiempo en el que el emisor queda bloqueado hasta que recibe confirmación de recepción. Este modelo requiere de un búfer para almacenar los mensajes que va enviando el emisor.



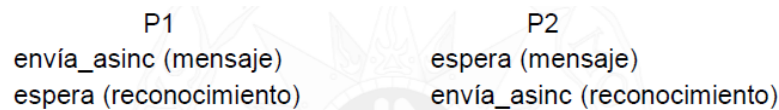
- **Invocación remota:** es un subtipo del modelo de comunicación síncrona, ya que el emisor queda bloqueado durante un periodo mayor de tiempo, pues espera a que el segundo proceso elabore una respuesta en función

del envío que se hizo. Así al principio se trata de una comunicación síncrona, quedando a la espera no de la confirmación de recepción del mensaje sino en espera de una respuesta del receptor.

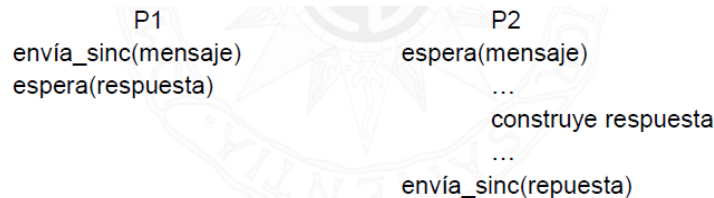


Con esto, se pueden dar algunas relaciones entre las distintas formas de envío:

- Relaciones síncronas mediante envíos asíncronos:



- Invocación remota mediante envíos síncronos:



Aunque con estas relaciones a partir de envíos asíncronos se pueden obtener resultados síncronos, los envíos asíncronos presentan una serie de inconvenientes:

- Se necesitan potencialmente infinitos buffers para almacenar la información de los mensajes que el proceso receptor no hubiera leído todavía.
- La comunicación asíncrona está anticuada y los envíos generalmente se programan para recibir un ACK.
- El modelo asíncrono necesita más comunicaciones y un esfuerzo de implementación mayor, dando lugar a programas más complejos y donde es más difícil de probar la corrección del sistema.
- No es una buena práctica simular una comunicación asíncrona mediante envíos síncronos debido a su complejidad y dificultad, pues necesita de la creación de buffers. Todo ello implica que es perjudicial sobre las prestaciones del sistema.

3. Nombrado de procesos y estructura de mensajes

El nombrado de procesos se puede clasificar en torno a dos categorías:

- Dirección frente a indirección: en ambos casos es necesario acudir a un nombre que identifique a los procesos o socios de la comunicación.
 - **Nombrado directo**: el emisor de un mensaje nombra explícitamente al receptor (envía <mensaje> a <nombre-proceso>). Tiene como ventaja su simplicidad.
 - **Nombrado indirecto**: el emisor nombra a alguna entidad intermedia, la cual se suele conocer como canal, buzón, enlace o tubería y que es la que se comunica con el receptor (envía <mensaje> a <buzón>). La comunicación puede seguir siendo síncrona y su ventaja es que facilita la descomposición del software.
- Simetría
 - **Nombrado simétrico**: tanto el emisor como el receptor se puede nombrar entre sí (directa o indirectamente):
 - Directo: envía <mensaje> a <nombre-proceso> o espera <mensaje> de <nombre-proceso>.

- Indirecto: envía <mensaje> a <buzón> o espera <mensaje> de <buzón>.
- **Nombrado asimétrico:** se da si el receptor no nombra una fuente específica, pero acepta mensajes de cualquier fuente (espera <mensaje>). A semeja al paradigma cliente-servidor, donde los procesos *servidor* proporcionan servicios como respuesta a los mensajes de cualquier proceso *cliente*. Una implementación debe ser capaz de mantener una cola de procesos esperando por el servidor.

En este último caso, si además de asimétrico es indirecto, hay que considerar otros asuntos, ya que el intermediario debería tener una estructura:

- **Muchos a uno:** cualquier número de procesos podría escribir en él, pero sólo un proceso podría leer de él (todavía concuerda con el paradigma cliente-servidor).
- **Muchos a muchos:** muchos clientes y muchos servidores.
- **Uno a uno:** un cliente y un servidor. Con esta estructura no es necesario que el sistema de soporte de ejecución mantenga colas.
- **Uno a muchos:** esta situación es útil cuando un proceso desea enviar una solicitud a un grupo de procesos trabajadores, sin importarle qué proceso sirve la solicitud.

Los tipos de nombrado que se utilizan en los envíos pueden ser mediante:

- **Hilos o procesos:** usa un ID que está unívocamente asignado a cada proceso. De esta forma, cada orden de enviar o recibir, incorpora dicho parámetro para identificar el socio de comunicación adecuado. Es un método directo, y puede ser tanto simétrico como asimétrico. Este tipo de nombrado es común en paquetes de programación concurrentes implementados en algunos lenguajes como Java.
- **Entrada o puertos:** los emisores deben incluir el nombre del puerto de entrada del receptor, que habitualmente es un módulo, o un proceso con uno o varios hilos dentro. Es empleado típicamente en Ada, donde una tarea puede recibir un mensaje que ha sido enviado a una de sus entradas mediante la ejecución de una declaración de aceptación. Cada entrada pertenece exactamente a una tarea, y todos los mensajes enviados a la misma entrada deben ser recibidos por una única tarea. Es un método directo y asimétrico.
- **Canales:** tanto emisor como receptor declaran un canal con un nombre dado y utilizan el nombre del canal para realizar el intercambio de mensajes. Es un método indirecto y asimétrico. Se usa en lenguajes como Ocaml.

Los métodos de entrada, puertos y canales requieren, al ser asimétricos, que el servidor tenga una cola de procesos. En el caso de lenguaje como Java, se pueden implementar cualquiera de los tres usando la librería `java.nio` para el uso de canales o `java.net`.

En cuanto a la **estructura de los mensajes** que puedan ser enviados, la situación ideal sería poder hacer uso de cualquier tipo de objeto y no tener restricción ninguna. Sin embargo, el hecho de que emisor y receptor tengan distintos tipos de objetos, dificultan la comunicación y la estructura de un mensaje puede variar según el receptor/emisor. Por ello, algunos lenguajes se restringen dichas estructuras (muchos lenguajes modernos no presentan restricciones de tipo).

4. Espera selectiva

En el paso de mensajes anteriores, el receptor debe esperar a que finalice todo el proceso para que se libere la comunicación establecida. Sin embargo, un receptor puede desear esperar a que le llame cualquiera de varios procesos o que los procesos servidores ignoren el orden en el que han recibido llamadas de los clientes.

En esta situación, algunos procesos receptores que puedan ser llamados por varios posibles emisores, se puede utilizar la espera activa para permitir a los procesos receptores esperar de forma selectiva cierto número de mensajes a la vez. Esta comunicación está basada en el uso de guardas.

Una operación con **guarda** se ejecuta si ésta se evalúa a verdadero (su condición). A diferencia de una operación selectiva, las guardas son no deterministas, es decir, tiene que haber una o varias guardas que se evalúen como verdadero. En caso de que más de una guarda sea verdadera, se produce una elección arbitraria y si ninguna es evaluada positivamente, la sentencia es abortada.


Cuando se aplica el uso de guardas en espera selectiva, los elementos guardados son operadores de mensaje (normalmente de recibir) y el planificador elige de manera no determinista cómo se ejecutan los procesos (varias guardas evaluadas a cierto).

Por ejemplo, si se tienen un hilo búffer con dos entradas:

- Depósito: para que un segundo hilo mande un mensaje al primero, el hilo búffer debe aceptarlo y notificar la recepción (aceptarYContestar()).
- Retirada: para que el segundo hilo solicite información al primero, el hilo búffer recibe la petición, la procesa y contesta (aceptar() y contestar()).


Implementación 1

```
while(true){
    if(buffer no lleno){
        item=deposito.aceptarYContestar();
        //procesar ítem
    }
    if(buffer no vacío){
        retirada.aceptar();
        //generar ítem contestación
        retirada.contestar(item);
    }
}
```

 Bloqueo esperando una llamada a depósito
Puede que llegue una llamada a retirada

Implementación 2

```
while(true){
    if(buffer no vacío){
        retirada.aceptar();
        //generar ítem contestación
        retirada.contestar(item);
    }
    if(buffer no lleno){
        item=deposito.aceptarYContestar();
        //procesar ítem
    }
}
```

 Bloqueo esperando una llamada a retirada
Puede que llegue una llamada a depósito

Para evitar esta situación, se debe implementar una espera selectiva que permita al hilo búffer esperar para aceptar ambas entradas a la vez:

```
Seleccionar:
    if(buffer no lleno) then aceptar una llamada a la entrada
    deposito y depositar el ítem;
    □ (buffer no vacío) then aceptar una llamada a la entrada
    retirada y devolver el ítem;
Fin seleccionar
```

En Ada, la espera selectiva se puede implementar con la sentencia `select`, que permite realizar la espera de más de una petición, definir un tiempo de espera si no aparece una petición en un tiempo determinado, rechazar una oferta de comunicación o finalizar si ningún cliente invoca sus servicios. En el caso de Java no está implementada de manera directa, pero puede simularse, aunque ofrece una estructura de **bloque guardados** mediante el uso de `synchronized`, `wait()`, `notify()` y `notifyAll()`.

Con el uso de bloques guardados en Java se permite gestionar la espera en una situación de paso de mensajes haciendo uso de hilos sincronizados suspendiendo y reactivando su ejecución según sea necesario. De esta forma se evitan comprobaciones o esperas inútiles. Sin embargo, sólo es aplicable en aquellos casos donde emisor y receptor tengan acceso a una memoria compartida.

```
public synchronized void bloqueGuardado(){
    //solamente se realiza el bucle una vez por evento
    while(!condicion){
        try{
            wait(); //suspendido hasta que otro hilo llame a notify
        } catch(InterruptedException e){}
    }
    //instrucciones cuando condicion sea verdadero
}
```

5. Modelos en sistemas distribuidos

Los **sistemas distribuidos** son sistemas con más de un elemento de procesamiento que operan de forma autónoma, cooperando para realizar tareas comunes, por lo que es importante establecer comunicaciones adecuadas entre procesadores que pueden ser muy heterogéneos. Así, es necesario establecer mecanismos que permitan:

- La traducción de datos y partición en paquetes transparente a los procesos.
- Los mensajes recibidos se encuentren en buenas condiciones.
- Los mensajes sean del tipo esperado.
- No haya restricciones para la comunicación en relación a los tipos de datos.

En Java, el paquete `java.net` permite implementar sistemas distribuidos de dos formas distintas:

- **TCP** (Transmission Control Protocol): establece una conexión previa y trabaja mediante canales. Una vez establecida la conexión, emisor y receptor se comunican a través del canal haciendo la entrega de mensajes de manera confiable y en orden.
- **UDP** (User Datagram Protocol): es más simple que el protocolo TCP, pero es menos fiable. Los mensajes se envían de forma independiente (sin haber establecido un canal) sin que se garantice la entrega ni el orden de entrega. En este caso, el direccionamiento se hace mediante nombres basados en puertos.

Otra opción permite usar la librería `java.rmi`, aunque requiere que todos los sistemas estén implementados en Java.

Para enviar y recibir mensajes UDP, Java cuenta con los objetos `DatagramSocket` y `DatagramPacket`.

```
DatagramSocket mi_socket = new DatagramSocket(puerto_1);
DatagramPacket mi_mensaje = new DatagramPacket(buffer, tamaño,
ip, puerto_2);
//Inicializar mensaje
mi_socket.send(mi_mensaje)
```

Para las comunicaciones TCP, Java implementa los canales mediante `ServerSocket` y `Socket`. La comunicación se realiza mediante un stream de datos de entrada y salida.

```
ServerSocket mi_servidor_socket = new ServerSocket(puerto);
Socket conexion_cliente = mi_servidor_socket.accept();

DataInputStream entrada = new
    DataInputStream(conexion_cliente.getInputStream());
PrintStream salida = new
    PrintStream(conexion_cliente.getOutputStream());
// Procesar el mensaje entrante en el servidor
String s = entrada.readLine();
. . .
//Enviar un mensaje al cliente
out.println("Hola cliente, soy el servidor");

Socket conexion_servidor = new
    Socket(nombre_servidor, puerto);
DataInputStream entrada = new
    DataInputStream(conexion_servidor.getInputStream());
PrintStream salida = new
    PrintStream(conexion_servidor.getOutputStream());
//Procesar el mensaje entrante del cliente
String s = entrada.readLine();
. . .
//Enviar un mensaje al servidor
out.println("Hola servidor, soy el cliente");
```