El paso de mensajes permite resolver problemas de programación concurrente en sistemas donde no es posible la comunicación de procesos entre variables compartidas, como por ejemplo los sistemas distribuidos. La única posibilidad que existe de comunicación en sistemas de estas características es el paso de mensajes, en el mismo los procesos intercambian mensajes entre ellos mediante operaciones explicitas de envío (*send*) y recepción (*receive*) que constituyen las primitivas básicas en cualquier sistema de comunicación de este tipo.

Identificación en el proceso de comunicación
Entendemos por identificación en el proceso de comunicación la forma en que el emisor indica a quién va dirigido el mensaje, en base a esto podemos hablar de comunicación ***directa*** e ***indirecta.***
La comunicación **directa** se puede dividir en *simétrica* y *asimétrica*, en la primera, el emisor nombra al receptor y viceversa; en la segunda el emisor continua identificando al receptor, pero el receptor no identifica a un emisor concreto.
En la comunicación **indirecta** no se identifica explícitamente a los procesos emisor y receptor. La comunicación se realiza depositando mensajes en un almacén intermedio que es conocido por los procesos interesados en la comunicación. A ese almacén intermedio se lo conoce con el nombre de buzón. En esa modalidad las primitivas de comunicación tendrían la siguiente forma:

SEND (buzonA, mensaje) *"enviar mensaje al buzón A"*
RECEIVE (buzonA, mensaje) "recibir un mensaje del buzón A"

Un buzón puede ser utilizado por más de dos procesos, e incluso entre dos procesos podemos emplear diferentes buzones (es decir, podemos tener más de un enlace de comunicación entre dos procesos, lo cuál no era posible en la comunicación directa). Por lo tanto este esquema de comunicación es más flexible que el anterior, ya que no solo nos permite llevar a cabo comunicaciones del tipo uno a uno, sino también comunicaciones uno a muchos, muchos a uno y muchos a muchos.

Sincronización
En todo proceso de comunicación, las partes interesadas (emisoras y receptoras) pueden no coincidir en el tiempo a la hora de realizar la operación de envío y recepción, podemos hablar entonces de dos tipos de comunicación: **síncrona** o **asíncrona**. Para ilustrar el funcionamiento en ambos casos podemos poner como ejemplo la diferencia existente entre el sistema postal y el sistema telefónico. En el primero de los casos donde la comunicación es claramente asíncrona, el emisor puede realizar la operación de envío sin que para ello sea necesario la coincidencia en el tiempo por parte del receptor. Esto generará la necesidad de almacenar los mensajes en *buffers* hasta que el receptor vaya retirando los mismos. Sin embargo, en el caso del sistema telefónico, la comunicación es síncrona, dado que debe darse la coincidencia en el tiempo de las operaciones envío y recepción; por lo tanto en la comunicación síncrona, el emisor quedará bloqueado en la operación de envío (*send*) hasta que el receptor esté preparado para recibir el mensaje (ejecutar la operación *receive*), o viceversa; es decir, los procesos quedarán bloqueados hasta que se produzca la coincidencia entre ambos. Dado que en el caso de síncrona, las primitivas send y receive provocan el bloqueo del emisor y receptor, también reciben el nombre de llamada y recepción bloqueante, y a su vez a la comunicación asíncrona también se la conoce con el nombre de llamada y recepción no bloqueante. Vale la pena aclarar que en el caso de la comunicación asíncrona el envío podría llegar a ser bloqueante si el *buffer* es de tamaño finito y el envío continuo de mensajes llena el *buffer*.
En teoría se puede dar cualquier combinación, las más habituales son: llamada (*send*) y recepción (*receive*) bloqueante; llamada y recepción no bloqueante; pero hay una alternativa muy interesante y utilizada que es: llamada no bloqueante y recepción bloqueante; teniendo en mente este modelo propuesto supongamos que tenemos dos procesos ejecutandose concurrentemente y quieren acceder a un recurso compartido (supongamos escribir en una base

de datos), mediante la utilización de mensajes con el modelo asíncrono descrito (envío no bloqueante y recepción bloqueante) podemos proponer la siguiente solución:

```
process P1;
var
   testigo: integer;
begin
   repeat
      ...
      RECEIVE(buzon, testigo);
      (* SECCION CRITICA *)
      SEND(buzon, testigo);
      ...
   forever
end;

process P2;
var
   testigo: integer;
begin
   repeat
      ...
      RECEIVE(buzon, testigo);
      (* SECCION CRITICA *)
      SEND(buzon, testigo);
      ...
   forever
end;
```

Como se puede observar, para garantizar la exclusión mutua se emplea un buzón y antes de lanzarse ambos procesos se envía un mensaje *(testigo)* al buzón; en este caso el contenido del mensaje es irrelevante ya que su uso solo es para sincronización. Un proceso antes de entrar en su sección crítica efectúa una operación *receive* sobre el buzón, y el proceso que reciba el *testigo* será el que tenga acceso a la sección crítica; al terminar su sección crítica, el proceso envía nuevamente el testigo al buzón, permitiendo la entrada de otro proceso a su sección crítica.

JMS (Java Service Message)
Estándar de mensajería que permite crear, enviar, recibir y leer mensajes, haciendo confiable la comunicación. El término “JMS client” se refiere a una aplicación o componete Java que usa la API y proveedor JMS para enviar y recibir mensajes.
JMS soporta dos estilos de mensajería: point-to-point y publish-and-subscribe. Antes que un cliente pueda usar un proveedor JMS para enviar y recibir mensajes, el cliente debe decidir que estilo de mensajería quiere usar.
Un cliente puede consumir un mensaje de manera sincrónica o asincronica.
JMS es una especificación y no un producto. Un proveedor JMS como ActiveMQ, IBM, Progress Software, etc. ofrecen un producto de mensajería que implementa la especificación. En nuestros ejemplos y ejercicios usaremos ActiveMQ (Apache) como proveedor JMS.

Veamos un ejemplo de comunicación asíncrona en JMS:

proyecto: **jms-async-ejemplo**