Capítulo 3

Sistemas basados en paso de mensajes

3.1 Introducción

Los programas de un computador *Von Neumann* clásico se conciben para su ejecución determinista como un único programa secuencial. Sin embargo, el deseo de mayor velocidad de ejecución ha tenido como consecuencia la introducción del paralelismo en los programas y la aparición de los multiprocesadores y el multiprocesamiento.

Multiprocesador

Se trata de sistemas de computador que incluyen varias unidades de procesamiento (CPU), denominados genéricamente *procesadores*, que comparten la memoria principal y periféricos con el objetivo de ejecutar programas simultáneamente. Actualmente se habla de *multiprocesamiento simétrico*, también conocido como SMP según su acrónimo en inglés. Se trataría de una arquitectura hardware para un tipo de multiprocesador en la cual varios procesadores idénticos están conectados a una sola memoria principal compartida, abstrayendo el control individual de cada uno de ellos mediante una instancia del sistema operativo, que actúa de interfaz con los usuarios y programas de aplicación. La arquitectura SMP (ver figura 3.1) es hegemónica actualmente, especialmente para los denominados *procesadores multinúcleo* que la usan para tratar a los diferentes núcleos de un procesador como si fueran procesadores distintos. En la arquitectura SMP, los procesadores pueden ser interconectados mediante *buses*, *conmutadores de barras cruzadas* o redes de *tipo malla* dentro del propio chip. Ejemplos de algunos sistemas SMP, que merecen citarse por la difusión que han obtenido, son los siguientes: AMD(Athlon II, Opteron), Intel(CoreiX, Xeon, Itanium), Sun (UltraSPARC) y ARM (MPCore).

A fin de completar esta revisión, haremos referencia al término *procesamiento asimétrico* (AMP) (ver figura 3.1). Históricamente el modelo de multiprocesador AMP fue un recurso de software provisional para poder gestionar sistemas con procesadores de diferentes tipos antes de la aparición de los sistemas SMP. En los sistemas AMP la CPU no es más que el procesador aritmético y lógico que ejecuta las aplicaciones de usuario, no posee la funcionalidad de procesamiento gráfico, *multienhebramiento*, etc. de los procesadores multinúcleo actuales. Además, todas las CPUs de un multiprocesador AMP han de poseer el mismo conjunto de instrucciones de bajo nivel para las aplicaciones que las utilicen, ya que un trabajo en ejecución

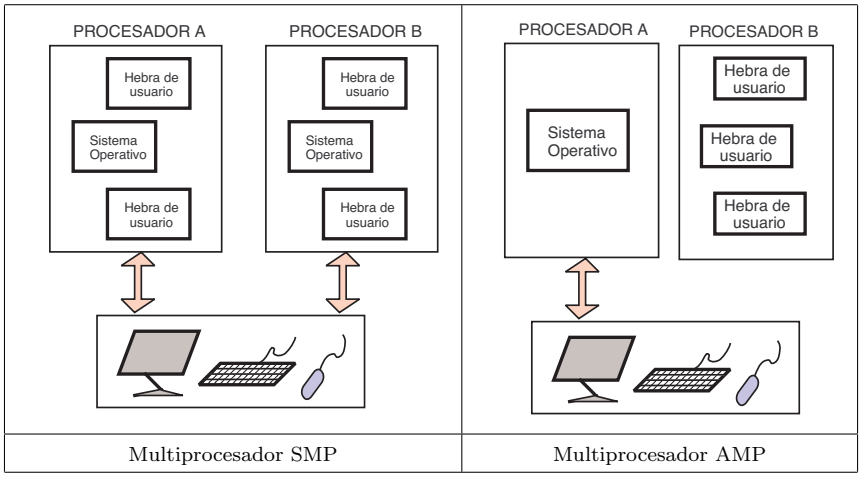

Figura 3.1: Tipos de multiprocesadores actuales

podría ser reasignado dinámicamente de una CPU a otra.

Los sistemas AMP presentaban un problema de programabilidad, ya que los sistemas operativos de la época se desarrollaban para una única CPU. No se llegó a resolver satisfactoriamente hasta la aparición de los sistemas SMP, en los que el sistema operativo y las aplicaciones bajo el control de este se ejecutan en todos los procesadores simultáneamente, con un tick de reloj de granularidad fina común. Ejemplos de sistemas AMP, que se utilizaron en la época anteriormente mencionada, son los siguientes: IBM(65MP), Burroughs B5000, CDC (6700), DEC (PDP-11 y VAX), Multics y UNIVAC (1108). En la actualidad los sistemas cuyas CPUs no son todas iguales suelen seguir otros modelos AMP, tales como multiprocesamiento de acceso a memoria no uniforme (NUMA) y multiprocesamiento en *racimos* (clustered).

Multiprocesamiento

Multiprocesamiento, o *multiproceso*, consiste en la utilización de dos o más CPUs en un mismo sistema multiprocesador para ejecutar los programas de una misma aplicación. El término *multiprocesamiento* también se refiere a la habilidad de un sistema de este tipo para gestionar más de un procesador y/o poder reasignar tareas entre dichos procesadores durante la ejecución de los programas mencionados.

El término multiprocesamiento se utiliza a veces para referirse a la ejecución de múltiples procesos software concurrentes en un sistema, en contraposición con la ejecución de un único proceso en cualquier instante de la ejecución de un programa, típico de los sistemas monoprocesador. Sin embargo, tal concepto resulta más apropiado indicarlo con cualquiera de los términos *multiprogramación* o *multitarea*, ya que estos suelen implicar una implementación en software, mientras que *multiprocesamiento* es un término más apropiado para describir la ejecución de un programa por múltiples procesadores–hardware. Un sistema puede realizar ambas cosas: multiprocesamiento y multiprogramación, o sólo una de éstas. Actualmente son muy raros los sistemas de computador que sólo tengan un procesador mononúcleo y que, por tanto, trabajen sin presentar ningún tipo de multiprocesamiento o multiprogramación.

Desde el punto de vista del modelo de ejecución de instrucciones de un programa por parte de un multiprocesador, los procesadores pueden utilizarse para una única secuencia o varias secuencias de instrucciones que se ejecutan en contextos múltiples. Flynn propuso una

clasificación (ver tabla 3.1) de los tipos de multiprocesamiento que se ha mantenido hasta la fecha.

	Instrucción única	Múltiples instrucciones
Datos únicos	`SISD`	`MISD`
Múltiples datos	`SIMD`	`MIMD`

Tabla 3.1: Taxonomía del multiprocesamiento.

El modelo `SIMD` describe a los multiprocesadores que ejecutan una única secuencia de instrucciones en diferentes contextos de ejecución. Es decir, todos los procesadores podrían sincronizarse para ejecutar la misma instrucción simultáneamente, pero que el resultado afectase a datos ubicados en distintas localizaciones de la memoria. El tipo SIMD está indicado para el procesamiento paralelo de vectores, en los cuales una gran cantidad de datos pueden ser divididos en partes que son modificadas independientemente. En este modelo una única secuencia de instrucciones dirige la operación de múltiples CPUs para que realicen las mismas manipulaciones sobre los datos, incluso sobre grandes cantidades diferentes a la vez.

Los multiprocesadores del modelo `MISD`, por el contrario, ejecutarían instrucciones distintas, que pertenecen posiblemente a diferentes secuencias de ejecución de los procesos de un programa, pero que afectan a la misma zona de memoria. El modelo `MISD` principalmente ofrece la ventaja de la redundancia en los cálculos, ya que múltiples procesos realizan las mismas tareas sobre los mismos datos, reduciendo las posibilidades de que se produzcan resultados incorrectos si una de éstas fallase. Las arquitecturas `MISD` pueden conllevar el realizar comparaciones entre diferentes procesadores para detectar fallos.

Aparte de las características de redundancia y seguridad de este tipo de multiprocesamiento, el modelo `MISD` posee muy pocas ventajas y resulta muy caro de implementar. No mejora el rendimiento de los programas. Puede ser implementado de tal manera que la existencia de los diferentes procesadores resulte transparente al software. Un ejemplo de la utilidad del modelo `MISD` es el proceso progresivo de imágenes, donde cada pixel de una imagen es *encauzado* a través de varios procesadores hardware que realizan pasos de transformación de dicha imagen.

En el modelo `MIMD` el procesamiento de las secuencias de instrucciones de los programas se divide en múltiples hebras, cada una de las cuales posee su propio estado del procesador dentro de uno o múltiples procesos definidos por software. Dado que ahora la mayoría de los procesadores son multinúcleo y trabajan con múltiples hebras que esperan ser planificadas, ya se trate de hebras del sistema o de los programas de usuario, este modelo de arquitectura está recomendado para hacer buen uso de los recursos de bajo nivel de los procesadores actuales.

El uso de multiprocesadores que siguen el modelo `MIMD` puede suscitar problemas de contención de recursos por los procesos e incluso conducir a interbloqueos, ya que las hebras pueden entrar en conflicto, de una forma impredecible, en su acceso a los recursos, que resulta difícil de prever y gestionar eficientemente.

Implementación del modelo de ejecución

La implementación del modelo `MIMD` necesita una codificación especial a nivel del sistema operativo, pero no requiere cambios sustanciales en las aplicaciones desarrolladas para un sistema monoprocesador, salvo que los programas de usuario utilicen múltiples hebras. Es decir, el modelo `MIMD` es transparente para los programas con una única hebra de control si el programa

no cede voluntariamente dicho control al sistema operativo durante su ejecución [1].

Por otra parte, con multiprocesadores que siguen el modelo `MIMD`, tanto el software del sistema como los programas de usuario pueden necesitar utilizar construcciones software, tales como semáforos o *cerrojos*, para impedir que una hebra interfiera a otra si se da el caso en que ambas entrelacen sus ejecuciones mientras referencian los mismos datos. La necesidad de acceso exclusivo a determinados datos durante la ejecución de las hebras asíncronas incrementa la complejidad del código, hace bajar el rendimiento de los programas, y convierte en imprescindible la verificación del código, aunque no hasta el punto de anular las ventajas del multiprocesamiento. Conflictos similares suelen aparecer entre los procesadores a nivel del hardware, por ejemplo, situaciones de contención y corrupción de datos en el cache. Normalmente dichos problemas deben ser resueltos a nivel de hardware, o mediante una combinación de software y hardware utilizando instrucciones para borrar el cache en los programas.

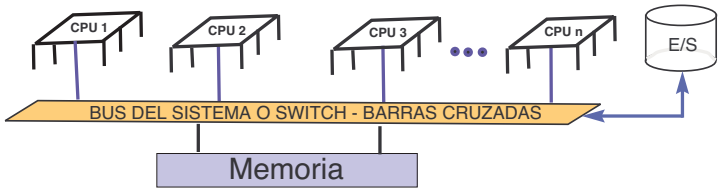

Figura 3.2: Representación de un multiprocesador con memoria compartida

Hay varias posibilidades de implementación del modelo MIMD:

1. Utilizar variables en memoria común a los procesadores (ver figura 3.2)
   - En este tipo de multiprocesador la sincronización entre los procesos está basada en la secuencialización que se produce al intentar acceder varios procesos a una misma dirección de memoria.
   - Es típica en este modelo un tipo de programación concurrente basado en monitores, semáforos, regiones críticas, etc.
   - Se necesitan dispositivos especiales para que los procesadores puedan acceder con eficiencia a posiciones de memoria compartida.
   - El inconveniente principal de estas arquitecturas es la falta de escalabilidad.

2. No existe memoria común, toda la comunicación y sincronización ha de llevarse a cabo a través de una red de comunicaciones (ver figura 3.3). Por eso se les llama *multicomputadores*.
   - Son más difíciles de programar que los multiprocesadores de memoria común, sin embargo presentan otras ventajas que los hacen ser las máquinas actualmente más utilizadas.
   - No presentan el problema de la escalabilidad de los multiprocesadores de memoria común.
   - Se necesita una notación de programación lo más flexible posible que permita expresar los diferentes modos de comunicación entre los procesos, así como expresar el no-determinismo en las comunicaciones.

[1] por ejemplo, provocando una llamada al sistema operativo

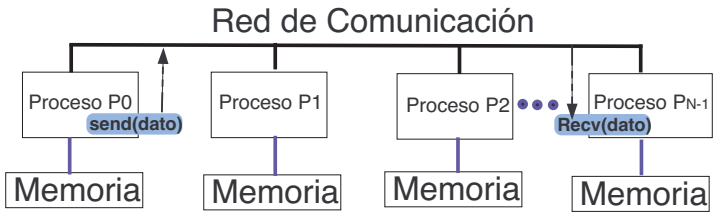

Figura 3.3: Representación de un multicomputador

- Las primitivas concurrentes clásicas: semáforos, regiones críticas, monitores, etc. no son adecuadas para programar los multicomputadores.

Estructura SPMD de un programa de paso de mensajes

Ejecutar un programa diferente en cada proceso, según el modelo general de la figura 3.3, puede ser algo difícil de manejar. Se suele utilizar un estilo de programación distribuida denominado *Single Program Multiple Data* (SPMD) en el cual el código que ejecutan los procesos es idéntico, pero sobre datos diferentes (ver figura 3.4). El estilo SPMD es una variante del modelo general `MIMD` que se aproxima al `SIMD`, en tanto en cuanto los procesadores ejecutan un mismo programa, pero no tienen que sincronizarse en la ejecución de cada una de las instrucciones individuales como ocurriría en un multiprocesador `SIMD`.

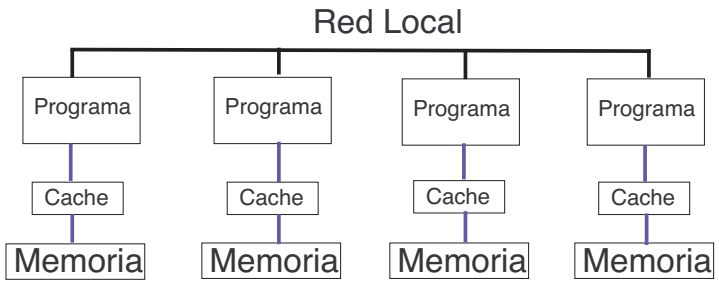

Figura 3.4: Representación del modelo de programación SPMD

Cada programa incluye la lógica interna necesaria para que cada proceso ejecute la tarea que le corresponda de acuerdo con el valor de su identificador, que será distinto para cada uno de ellos, y suele establecerse en una etapa de configuración distribuida posterior a la compilación.

3.2 Mecanismos básicos en sistemas basados en paso de mensajes

En los sistemas multicomputador, los procesos de un programa concurrente se comunican y sincronizan mediante *paso de mensajes*. Por *comunicación* entendemos que los procesos se envían y reciben mensajes en lugar de acceder en escritura o lectura a las variables compartidas del programa. La *sincronización* entre procesos se produce como consecuencia de que la recepción de un mensaje es posterior al envío del mismo y, de forma general, la ejecución de la operación de recibir supone la espera de la disponibilidad del mensaje en la parte receptora.

Las primitivas básicas de paso de mensajes se refieren al envío y recepción de una serie de datos entre el emisor del mensaje y el receptor:

- `send(<lista de variables>,<identificador_destino>)`
- `receive(<lista de variables>,<identificador_origen>)`

Para programar correctamente ha de tenerse en cuenta qué esquema concreto de *identificación de la comunicación* utiliza el lenguaje de programación o el sistema, así como el *modo de sincronización* o semántica de las primitivas de paso de mensajes anteriores.

Esquemas de identificación de los procesos comunicantes

Se trata de determinar cómo se identifican mutuamente los procesos emisores y receptores de mensajes durante la ejecución de un programa distribuido. En el caso de la llamada *denominación directa*, se utilizan los identificadores de los procesos para que el emisor pueda señalar explícitamente al receptor y viceversa. El problema principal que presenta esta opción consiste en que los identificadores de los procesos se han de asignar previamente a la ejecución del programa y dicha asignación se ha de mantener durante ésta. Como consecuencia de esto, cualquier cambio en la identificación de los procesos requiere recompilar el código. Este tipo de denominación tiene como ventaja principal que no produce ningún retardo debido a la identificación, aunque sólo permite comunicaciones uno–a–uno entre los procesos.

```
Proceso P0
int dato;
Produce(dato);
send(&dato,P1);
```

```
Proceso P1
int x;
receive(&x,P0);
Consume(x);
```

El esquema de identificación entre procesos que presenta una mayor flexibilidad, ya que permite varias configuraciones de comunicación entre grupos de procesos, se llama *denominación indirecta*. Se basa en la utilización de un objeto intermedio denominado *buzón* entre los procesos comunicantes, de esta forma los procesos designan al buzón como destino u origen de los mensajes que van a intercambiarse y, por tanto, se evita la restricción de los enlaces uno–a–uno que implica la utilización del esquema de denominación directa. Existen tres tipos de buzones, dependiendo de la relación que se establezca entre los procesos.

Relación	1-a-1	muchos-a-1	muchos-a-muchos
Tipo	Canales	Puertos	Buzones generales

Tabla 3.2: Tipos de buzones

El destino de los mensajes que se envían a través de un puerto es un único nodo, pero el origen del mensaje puede ser uno entre un conjunto y no es necesario especificarlo. Por tanto, los puertos pueden servir para multiplexar múltiples puntos de destino en un único nodo de la red. Cada proceso en los programas recibirá la información remota a través de sus puertos, lo cual permite la utilización de más de un servicio de red simultáneamente. Los puertos son parte de la capa de transporte en el modelo TCP/IP y de la capa de sesión en el modelo ISO.

En el caso de los buzones generales el destino de los mensajes envíados por un proceso puede ser cualquier nodo de la red. Así como también, cualquier nodo puede ser origen de un

mensaje recibido en el destino. Se pueden entender como puertos que multiplexan tanto los puntos de destino como los de origen en un único nodo de la red. Como consecuencia de ello tienen una implementación más complicada e ineficiente que los puertos si no existe una red de comunicaciones especializada. En general, el envío de un mensaje implica su transmisión al resto de la red y la recepción significa notificar la disponibilidad del mensaje a todos.

```
Channel of Integer Buzon;
  Proceso P0                          Proceso P1
  int dato;                           int x;
  Produce(dato);                      receive(&x,Buzon);
  send(&dato,Buzon);                  Consume(x);
```

Los canales se pueden entender como un tipo especial de puerto que sólo tiene un nodo origen. También como un servicio de comunicación orientado a establecer conexiones entre los procesos de las aplicaciones software, similar al concepto de *circuito virtual* entre nodos de una red. Con los canales se puede transmitir un flujo de datos simple, de tal forma que dichos datos son entregados en el orden en que se enviaron, sin que su división en paquetes o marcos ocasione que se desordenen durante la transmisión por la red. Los canales se pueden implementar utilizando Transmission Control Protocol (TCP), mediante un protocolo específico que proporciona circuitos virtuales encima del protocolo IP[2], que no está orientado a establecer conexiones confiables entre nodos y, por tanto, no garantiza que se mantenga el orden de envío de los datos en la recepción. El protocolo X.25 proporciona comunicación nodo-a-nodo confiable y garantiza al mismo tiempo la calidad de servicio en las comunicaciones; este protocolo proporciona identificadores de canales virtuales (VCI) en las aplicaciones.

Desde el punto de vista de la implementación, si dos procesos interactuantes se ubican en un mismo procesador, entonces el medio de transmisión de los mensajes puede ser simplemente la memoria local del procesador. Si, por el contrario, estos están en diferentes procesadores, entonces el paso de mensajes entre los dos procesos ha de ser realizado a través de un medio de comunicaciones físico que conecte a ambos. Actualmente, el paso de mensajes ha de ser entendido como una primitiva de comunicación y de sincronización entre procesos más cercana a los lenguajes de programación que a la plataforma [Andrews, 1991][3].

Semántica de las operaciones de paso de mensajes

El significado o *semántica* de estas operaciones puede ser diferente, es decir, el resultado de su ejecución podría variar dependiendo de la *seguridad* y el *modo de comunicación* que necesite un programa o aplicación. Por tanto, existen diferentes versiones de las operaciones de paso de mensajes que garantizan o no la seguridad en la transmisión de los datos y diferentes modos de comunicación.

La propiedad de *seguridad* en el paso de mensajes se cumple por parte de la operación *send* cuando la ejecución de esta operación garantiza que el valor recibido por el proceso destino sea el que tenían los datos justo antes de la llamada. En el ejemplo siguiente se considera que la operación *send* en el proceso P_0 posee semántica segura y que el proceso P_1 siempre imprimirá el valor `100` cada vez que se ejecute dicho código:

[2]el circuito virtual es establecido identificando el par de direcciones de los *sockets* de red receptor y emisor, es decir, sus direcciones IP y números de puerto

[3]plataforma de computación = sistema operativo + nivel de red (según la normalización ISO)

```
Proceso P0
int dato=100;
send(&dato,P1);
```

```
Proceso P1
int x;
receive(&x,P0);
imprime(x);
```

Una operación de envío con semántica *insegura* podría ocasionar que el valor recibido por P_1 fuese distinto de `100` si, por ejemplo, el valor de `dato` es alterado después de volver la llamada a *send* pero antes de que el sistema comience a transmitir el valor de la variable.

Por otra parte, la llamada a la operación *receive* no necesariamente detiene al proceso receptor P_1 y, por tanto, el valor de la variable `x` podría ser alterado por otras instrucciones antes de que termine la transmisión física de `dato`. Normalmente, los sistemas que ofrecen una operación *receive* no bloqueante también poseen una operación de comprobación que indica en qué momento se pueden alterar los datos que son recibidos y de esta forma se pueda programar con una operación *receive* semánticamente segura.

3.2.1 Operaciones bloqueantes

La semántica de este tipo de operaciones de paso de mensajes implica que la llamada a la operación *send* sólo vuelve cuando se garantice la propiedad de seguridad (ver sección 3.2). No siempre ocurriá que el proceso receptor haya recibido el dato cuando termine la ejecución de *send*, sino más bien que los cambios que se hayan producido en los datos durante la comunicación no violarán la citada semántica de estas operaciones.

Modo comunicación	Hardware especializado	Sincronización	Seguridad
Sin búfer	-	Sí (citas)	Sí
Con búfer	Sí	Relajada	Sí
	No	Sí	Sí

Tabla 3.3: Características de las operaciones bloqueantes

Paso de mensajes síncrono (sin búfer)

En el paso de mensajes síncrono la comunicación se lleva a cabo mediante un enlace directo entre los procesos participantes. Supongamos que un proceso A le envía datos a un proceso B. Cuando el proceso A ejecute la operación *send* se esperará hasta que el proceso B ejecute la operación de recepción *receive*. Antes que los datos se puedan transmitir físicamente los procesos han de encontrarse preparados para participar en el intercambio, lo cual exige una *cita* entre el emisor y el receptor (ver figura 3.5). Es decir, de forma similar a lo que ocurre en el emisor, la llamada a *receive* en el proceso receptor se suspende hasta que el otro proceso llame a la operación *send*.

El concepto de *cita* entre procesos comunicantes implica:

- Sincronización entre emisor y receptor para que se produzca el intercambio.
- El proceso emisor podrá realizar aserciones acerca del estado del receptor en el punto de sincronización.

- Puede considerarse un tipo de comunicación análoga a una conversación telefónica o un *chat* de dos participantes.

Las operaciones bloqueantes proporcionan a los programas de aplicación un tipo de comunicación que respeta la semántica de seguridad en el paso de mensajes en todos los casos, sin embargo suele tener una implementación ineficiente en los sistemas si los procesos participantes en una *cita* no están preparados para al mismo tiempo. En la figura 3.5 se puede ver que los procesos emisor o receptor sufren espera ociosa si el otro participante no ha llegado todavía a ejecutar su operación de recepción o envío del mensaje, respectivamente.

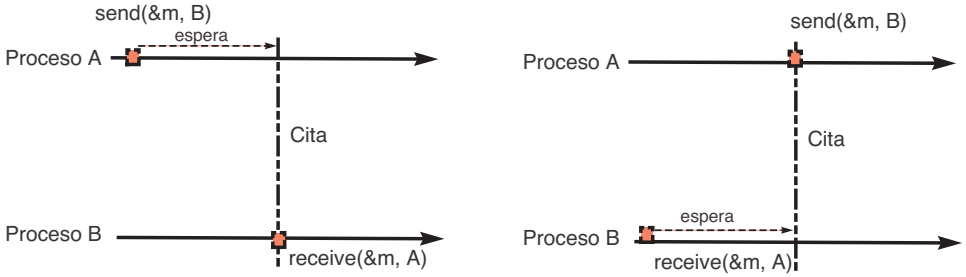

Figura 3.5: Cita entre procesos con operaciones de comunicación síncronas

Otro problema que afecta a las propiedades de seguridad sería la posibilidad de que aparezcan interbloqueos si no se intercambian las operaciones de envío y recepción que se refieran al mismo mensaje. El siguiente código producirá el bloqueo indefinido de ambos procesos si las operaciones *send* y *receive* son bloqueantes:

```
Proceso P0
send(&dato_1,P1);
receive(&x_2,P1);
```

```
Proceso P1
send(&dato_2,P0);
receive(&x_1,P0);
```

Paso de mensajes con búfer

En este tipo de paso de mensajes, la operación *receive* posee la misma semántica (significado/comportamiento) que en el paso de mensajes síncrono. Sin embargo, la primitiva de paso de mensajes `send()` posee una semántica diferente. El medio de comunicación entre los procesos no es ahora un enlace directo entre los 2 procesos que participan en la comunicación, sino más bien una cola de mensajes. La existencia de un hardware de comunicación especializado permite al proceso A continuar con su ejecución después de llamar a la operación *send*, ya que cuando el proceso A envía al proceso B, el mensaje se añade al búfer que representa la cola de mensajes pendientes de ser recibidos. Para recibir un mensaje, el proceso B ejecuta la operación *receive* que elimina el mensaje situado en la cabeza de la cola de mensajes, continuando después con su ejecución. Si no hay mensajes que recibir, es decir, el búfer se encuentra vacío, entonces la primitiva de comunicación `receive()` bloqueará al proceso receptor hasta que algún proceso emisor provoque que el búfer deje de estar vacío.

Implementaciones de bajo nivel

Existen dos variantes, interesantes de comentar, al modelo básico de paso de mensajes bloqueante. La primera se aplica a los sistemas que utilizan *canales* (ver sección 3.2) como medio

de comunicación. Consiste en que algunos sistemas implementan una primitiva denominada operación *vacío*, que testea el contenido de un determinado canal, y devuelve el valor `true` si no hay mensajes. La utilidad de esto sería la de prevenir el bloqueo de la llamada a la operación *receive* cuando, en ausencia de mensajes presentes en el canal, se puede aprovechar el tiempo realizando algún trabajo útil alternativo a la espera ociosa.

La segunda variante consiste en que la mayoría de los sistemas basados en paso de mensajes bloqueante utilizan un *búfer* interno[4] de longitud fija en el receptor (ver figura 3.6). Cuando este ejecuta la operación *receive*, el sistema comprueba si el mensaje está ya disponible en el búfer y, si lo está, copia los datos en el área de memoria donde el proceso receptor espera recibir el mensaje. Si la plataforma donde se ejecuta cuenta con hardware especializado, la transferencia de los datos se iniciará inmediatamente después de que sean copiados en el búfer interno, sin que se vea interrumpido el proceso receptor, por tanto, decimos que la sincronización entre los procesos emisor y receptor se ve relajada en este caso (ver tabla 3.3). Por el contrario, si no se cuenta con un hardware de comunicaciones específico, el proceso emisor interrumpirá al receptor, interviniendo ambos procesos en la transferencia interna de datos al búfer del receptor. Posteriormente, cuando el receptor llama a la operación *receive*, se copia el mensaje aludido desde el búfer a la zona de memoria asignada para recibirlo.

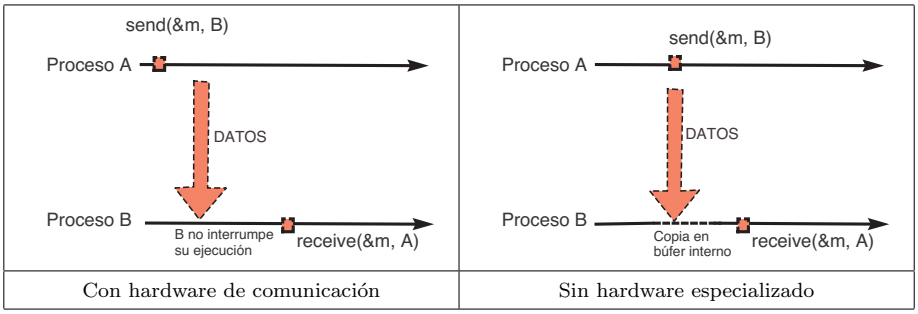

Figura 3.6: Implementación del paso de mensajes bloqueante con búfer

3.2.2 Operaciones no-bloqueantes

Las operaciones bloqueantes garantizan comunicaciones con semántica segura respecto de los datos que se transmiten, pero adolecen de ineficencia a la hora de su implementación en plataformas que no posean un hardware de comunicaciones especializado:

Paso de mensajes	Motivo de la ineficiencia
Síncrono	Espera ociosa
Con búfer	Sobrecarga por gestión del búfer

Tabla 3.4: Problemas de implementación del paso de mensajes bloqueante

Por consiguiente, en lugar de utilizar este tipo de operaciones se podría pensar en definir operaciones *send* y *receive* que no bloquean y dejar en la responsabilidad del programador el

[4]no confundir con la estructura de datos programada por el usuario en programas que siguen el modelo productor-consumidor

asegurar la semántica en el paso de mensajes que programe sus aplicaciones. Esto se puede lograr permitiendo que las operaciones aludidas devuelvan el control al programa donde se ejecutan antes incluso de que sea seguro modificar los datos. El programador se encargará de asegurar que no se alteren los datos de los programas mientras están siendo transmitidos si esto puede ocasionar errores en las aplicaciones. Para poder llevarlo a cabo, han de existir *sentencias de comprobación de estado* que indican si en un momento dado se pueden alterar los datos sin provocar que la semántica deje de ser segura. De esta forma, una vez iniciada la operación de paso de mensajes, el programa podría realizar cualquier cálculo que no dependa de la finalización de la operación y comprobará la terminación de ésta cuando sea necesario.

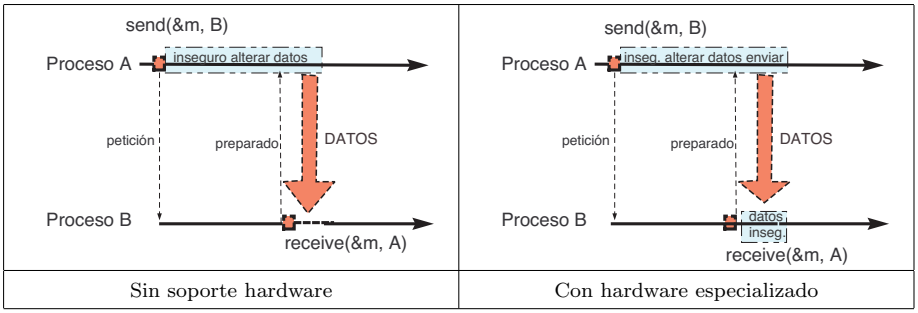

Figura 3.7: Paso de mensajes no-bloqueante sin búfer

Paso de mensajes sin búfer

La ejecución de una operación *send* con operaciones no bloqueantes informa al sistema que hay un mensaje pendiente, pero el proceso emisor continua su ejecución después de haberlo enviado. De esta manera se pueden iniciar otros cálculos, no necesariamente relacionados con la comunicación, por parte del programa mientras el mensaje se encuentra en transmisión. Cuando en el otro proceso se confirme la llamada a la operación *receive* del mensaje, se iniciará la comunicación física entre el emisor y el receptor. Si no existe soporte de hardware especializado, el proceso receptor se suspende desde que el sistema está preparado para recibir los datos hasta el final de la transmisión. Con soporte de hardware especializado, la operación *receive* vuelve inmediatemente, aunque no se hayan terminado de transmitir los datos del mensaje. Existe una operación de comprobación que indicará cuándo es seguro acceder a los datos que están siendo comunicados (ver figura 3.7).

Paso de mensajes con búfer

La diferencia fundamental con el modo *sin búfer* consiste en que cuando se llama a la operación *receive* se inicia la transferencia de los datos del mensaje desde el búfer interno al área de memoria del receptor donde espera hallarlos. Como consecuencia, se reduce el tiempo de espera en el receptor durante el cual un acceso a dichos datos es inseguro.

3.2.3 Tipos de procesos en programas de paso de mensajes

1. **Filtros**: son procesos transformadores de datos. Reciben flujos de datos de sus canales de entrada, realizan algún cálculo en los flujos de datos, y envían los resultados a los canales de salida.
2. **Clientes**: son procesos desencadenantes de algo. Los clientes hacen peticiones a los

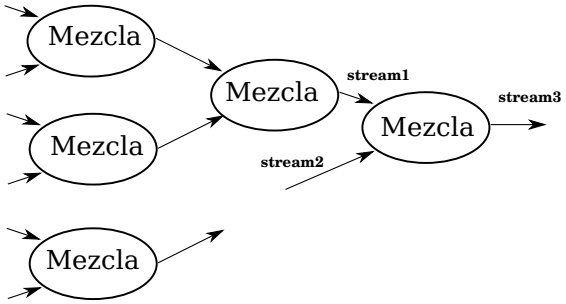

Figura 3.8: Procesos de tipo filtro.

procesos servidores y desencadenan reacciones en los servidores. Los clientes inician una actividad, cuando ellos eligen el momento, y a menudo se esperan hasta que se haya servido la petición.

3. **Servidores**: son procesos reactivos. Esperan hasta que se hagan las peticiones y, entonces, reaccionan a una petición. La acción específica que toman depende de la petición, los parámetros de la petición y el estado del servidor. El servidor puede responder inmediatamente o puede tener que guardar la petición para responderla después. Un servidor es un proceso que nunca termina y que a menudo sirve a más de un cliente.

```
const EOS = high (int); \\ fin del marcador de flujo
op stream1 (x:int), stream2 (x:int), stream3 (x:int);
process mezcla{
  int  v1, v2;

  receive stream1 (v1);
  receive stream2 (v2);

  do (v1 < EOS and v2 < EOS)
    if (v1 <= v2)
       send stream3 (v1);
       receive stream1 (v1);
    [] v2 <= v1 ->
       send stream3 (v2);
       receive stream2 (v2);
    fi
  od;

  if (v1 = EOS)
    send stream3 (v2);
  [] (v1 <> EOS) ->
    send stream3 (v1);
  fi;

  send stream3 (EOS)
}
```

Figura 3.9: Implementación de los procesos filtro "mezcla"

4. **Pares**: son procesos idénticos que interaccionan para proporcionar un servicio o resolver un problema.

3.3 Modelos y lenguajes de programación distribuida

En lo que sigue nos centraremos en los multicomputadores y en los modelos y lenguajes de programación adecuados para estas plataformas de computación.

Para poder programar procesos que sean del tipo procesos *servidores*, los lenguajes introducen una nueva sentencia de programación denominada *orden guardada*. La semántica de esta orden proviene de la idea de considerar la selección entre varias alternativas de forma *no–determinista* como una *ayuda mental* [Dijkstra, 1975], más que como un inconveniente, en el desarrollo de programas distribuidos. Hasta el artículo aludido, los informáticos pensaban en las estructuras no-determinísticas como *algo a eliminar* en los programas, ya que se consideraban como una fuente de posibles errores en la fase de mantenimiento posterior del software. La práctica tradicional era eliminarlo total o parcialmente en la fase de codificación, intentándose preveer qué alternativa en concreto tomaría un programa en el momento de su ejecución. Sin embargo, contar con sentencias no-deterministas en los lenguajes de programación puede tener sentido para facilitar la implementación de un determinado tipo de sistemas que reaccionan frente estímulos procedentes de su entorno.

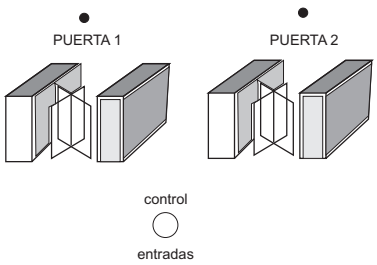

Figura 3.10: Modelo hardware de control de entrada a un museo

Considérese, por ejemplo, el caso de la implementación de un controlador para contar la entrada a un museo con dos puertas (ver figura 3.10). Los sensores de detección de presencia de cualquiera de las puertas pueden enviar al controlador una señal de que ha entrado una nueva persona; pero no se puede predecir el orden de envío de las señales, de hecho podrían recibirse al mismo tiempo.

```
P1::                        P2::                        P3::\\Proceso controlador
  for(i=1;i<20;i++){        for(i=1;i<20;i++){          for (j=1;j<20;j++){
      send(P3,1);              send(p3,1);                 receive(P1, temp);
      --pasa 1 persona         --pasa 1 persona            cont+=temp;
  }                         }                              receive(P2, temp);
                                                           cont+=temp;
                                                        }
                                                        printf("%d", cont);

main(){
   cobegin P1;P2;P3 coend;
}
```

Figura 3.11: Implementación inadecuada de un controlador para detectar entradas

Una implementación totalmente determinista de controlador, como la que se muestra en la figura 3.11, es totalmente errónea. Si suponemos paso de mensajes de tipo síncrono, el proceso

P3, que representa al controlador, podría bloquearse si no se producen entradas desde la puerta 1, aunque entren personas por la puerta 2. El proceso P3 que implementa al controlador es un proceso de tipo *servidor* y, por tanto, no conoce de antemano al proceso cliente que se va a comunicar con él en cada instante.

3.3.1 Espera selectiva con órdenes guardadas

Un *servidor* es un tipo de proceso que ha de estar preparado para recibir un mensaje de cualquiera de sus clientes, sin que el orden en que se produzcan dichas comunicaciones pueda ser determinado a priori. Además, dependiendo del estado de los datos en el servidor, en cada iteración de dicho proceso podrían estar permitidas las comunicaciones de sólo algunos clientes. Pensar por ejemplo en un proceso servidor que implementa una cola circular (*búfer*) con las operaciones clásicas de *inserción* para los productores y *eliminación* para los consumidores. Cuando se llene el búfer de datos, no se recibirá más de los productores hasta que al menos un consumidor establezca la comunicación y elimine un dato. Análogamente, cuando el búfer se quede vacío, no se aceptarán comunicaciones con los consumidores hasta después de recibir al menos un dato de un productor. Cuando el búfer se encuentre en un estado distinto de los dos anteriores, se podrán aceptar comunicaciones de los productores y consumidores en un orden *no determinado*.

Construcción	Propósito
Orden guardada	Permitir una comunicación condicional con un cliente
Espera selectiva	no determinismo en selección de alternativas *guardadas*

Tabla 3.5: Órdenes estructuradas para lenguajes con operaciones de comunicación síncronas

Órdenes guardadas

Dijkstra propuso unas nuevas construcciones, para ser incluidas en los lenguajes de programación distribuidos y con operaciones de comunicación síncronas, que llamó *ordenes guardadas*. Las mencionadas órdenes pasan a ser las sentencias componentes básicas de la construcción denominada *espera selectiva* de un lenguaje del tipo anterior:

```
   <espera.selectiva>::= SELECT <conjunto.ordenes.guardadas> END SELECT
<conjunto.ordenes.guardadas>::= <orden.guardada>OR <orden.guardada>}
          <orden.guardada>::= <guarda> -> <lista.sentencias>
                    <guarda>::= <expresion.booleana> |
                    <expresion.booleana>;receive(<argumentos>);|
                     receive(<argumentos>)
```

Se dice que una orden guardada está *preparada* para ser ejecutada si la expresión booleana (o condición) que precede a la instrucción *receive* se evalúa como cierta y la propia operación *receive* está lista para recibir el mensaje.

Es el propio proceso servidor que programa la orden de espera selectiva quien selecciona en cada llamada a la orden SELECT, de una forma totalmente incontrolable desde los procesos

que constituyen su entorno, qué orden guardada concreta va a ejecutarse entre las incluidas en el subconjunto de las *preparadas*. La orden puede incluirse dentro de un bucle, de tal forma que en cada iteración se ejecutará una nueva instancia de la orden SELECT.

```
PUERTA(i:1..2)::
{ int s=0;
  do
   SELECT (s<HORA.CIERRE && PERSONA())->
       {send(CONTROL, S()); // envia una se\~nal de entrada de persona
        DELAY.UNTIL(s+1); // espera hasta el siguiente instante
        s:=s+1;} // cuenta un nuevo tick de reloj
     OR
       (s<HORA.CIERRE && NOT PERSONA())->
       {DELAY.UNTIL(s+1); // espera hasta el siguiente instante
        s:=s+1;} // cuenta otro tick
     OR
       TRUE->DELAY.UNTIL (TIME() + 16*3600); // es la hora de
       // cierre del museo; hay que esperar 16 horas.
       // TIME() devuelve una cuenta en segundos.
  END SELECT
  while(true);
  send(CONTROL, Start();
}
CONTROL::
{ int cont= 0;
  SELECT  receive(Start(), PUERTA(1)); // desde cualquiera de los sensores
      OR                               // de las puertas se arranca
          receive(Start(), PUERTA(2)); // el controlador
  END SELECT
  do
     SELECT receive(S(), PUERTA(1))-> cont:= cont+1;
         OR
            receive(S(), PUERTA(2))-> cont:= cont+1;
     END SELECT
     // cuenta una persona mas, porque ha recibido la se\~nal
     // de cualquiera de las 2 puertas (no se puede saber cual)
  while(true);
  printf("numero de personas",%d, cont));
}
main(){
  cobegin PUERTA;CONTROL coend;
}
```

Figura 3.12: Implementación del controlador con órdenes con guarda

Resumen de la semática de la orden SELECT:

- El subconjunto de órdenes guardadas *preparadas* se determina una sola vez al comienzo de su ejecución.
- Para volver a determinar qué órdenes están preparadas hay que ejecutar nuevamente la orden.
- Mientras un proceso cliente no realice algún envío que empareje con alguna de las órdenes guardadas cuya condición se evaluó como cierta, la orden producirá un bloqueo ya que no posee ninguna orden guardada ejecutable en ese momento.
- Algunos lenguajes[5] permiten programar SELECT con alternativas prioritarias, pero en estos casos la selección dejaría de ser no determinista.

Por tanto, las órdenes guardadas permiten implementar selecciones no-deterministas en los programas, es decir, la alternativa realizada e incluso el estado final no dependen únicamente del estado inicial que tuviera el proceso antes de ejecutar la orden con guarda seleccionada.

En la figura 3.12 se puede ver una implementación correcta, con órdenes guardadas y espera selectiva, inspirado en un lenguaje distribuido [Hoare, 1985], para el ejemplo del controlador para la entrada de personas.

3.4 Bibliotecas de paso de mensajes

Actualmente MPI (Message Passing Interface) se ha convertido en un estándar, siendo una interfaz muy útil para la realización de aplicaciones paralelas basadas en paso de mensajes [Snir et al., 1999]. El modelo de programación paralela y distribuida que se puede realizar con MPI es el denominado MIMD, aunque se suele utilizar bastante con un modelo que es un caso particular, el denominado modelo SPMD (ver tabla 3.1). En el modelo SPMD todos los procesos ejecutan el mismo programa, aunque no necesariamente la misma instrucción al mismo tiempo. MPI es, como su nombre indica, un interfaz, lo que quiere decir que el estándar no exige una determinada implementación del mismo. Lo importante es dar al programador una colección de funciones para que este diseñe su aplicación, sin que tenga necesariamente que conocer el hardware concreto sobre el que se va a ejecutar, ni la forma en la que se han implementado las funciones que emplea.

El desarrollo de la interfaz MPI y sus implementaciones se ha debido a MPI Forum, un grupo formado por investigadores de universidades, laboratorios y empresas involucrados en la computación paralela, también denominada de *altas prestaciones* (HPPC, según su acrónimo en inglés). Básicamente MPI pretende definir un único entorno de programación, que garantice la total portabilidad de las aplicaciones paralelas, basado en una única interfaz y sin especificar cómo se debe llevar a cabo la implementación de ninguna de ellas. También ofrece a los usuarios y programadores implementaciones, de dominio público, de dicho entorno que aseguran un nivel de calidad con el objetivo de propiciar la difusión del estándar.

[5] Ada 95, por ejemplo, permite definir alternativas de mayor prioridad en la selección no determinista

Los elementos básicos de MPI son una definición de un interfaz de programación independiente del lenguaje, más una colección de implementaciones de ese interfaz (o *bindings*, como se dice sucintamente) para los lenguajes de programación más extendidos en la comunidad usuaria de computadores paralelos, es decir: Ada, C y FORTRAN. Cualquiera que quiera utilizar MPI para desarrollar software ha de trabajar con una implementación de MPI que constará de, al menos, los siguientes elementos:

- Una biblioteca de funciones para C, más el archivo de cabecera `mpi.h` con las definiciones de esas funciones y de una colección de constantes y macros.
- Una biblioteca de funciones para FORTRAN, junto con el archivo de cabecera `mpif.h`.
- Comandos para compilación, típicamente `mpicc, mpif77`, que son versiones de las órdenes de compilación habituales (`cc, f77`), que incorporan automáticamente las bibliotecas MPI.
- Órdenes específicas para la ejecución de aplicaciones paralelas, normalmente denominada `mpirun`.
- Herramientas para monitorización y depuración de programas paralelos.

MPI no es, evidentemente, el único entorno disponible para la elaboración de aplicaciones paralelas, ya que existen otras muchas alternativas, tales como:

- Utilizar las bibliotecas de programación propiedad del computador paralelo disponible:
  - NX en el Intel Paragon
  - MPL en el IBM SP2, etc.
- PVM (Parallel Virtual Machine), que posee unas características similares a MPI e intenta hacer que una red de estaciones de trabajo funcione como un multicomputador.
- Lenguajes de programación paralelos o que incluyan directivas de paralelismo.
- Lenguajes de programación secuenciales, junto con compiladores que paralelicen automáticamente el código producido por los programas.

Como anteriormente se ha comentado, MPI está diseñado pensando en el desarrollo de aplicaciones SPMD. Este tipo de programas lanzan en paralelo N copias de un mismo programa, que se ejecutan por procesos asíncronos, es decir, hay que programar en el código de los procesos las instrucciones necesarias de semáforos para sincronizarlos. Dado que los procesos en MPI poseen un espacio de memoria completamente separado, el intercambio de información, así como la sincronización entre ellos, se ha de hacer exclusivamente mediante paso de mensajes. MPI ofrece a los programadores tanto operaciones de paso de mensajes bloqueantes como no bloqueantes. Las primeras facilitan una programación más fácil y segura, mientras que las segundas permiten optimizar el rendimiento de los programas distribuidos al enmascarar las sobrecargas debidas a la comunicación y transmisión de datos. En MPI se dispone de operaciones punto-a-punto para 2 procesos comunicantes, así como funciones u operaciones colectivas para involucrar a un grupo de procesos. Los procesos pueden agruparse y formar

comunicadores, lo que permite una definición del ámbito de las operaciones colectivas, así como propician el desarrollo de un diseño modular de las aplicaciones.

A continuación se ve un ejemplo de programa que utiliza el *binding* de MPI para el lenguaje de programación C:

```
# include "mpi.h"
#include <iostream>
using namespace std;
main (int argc, char **argv) {
  int nproc; /*Numero de procesos */
  int yo; /* Mi direccion: 0<=yo<=(nproc-1)*/
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
  MPI_Comm_rank(MPI_COMM_WORLD, &yo);
  /* CUERPO DEL PROGRAMA */
  cout<<"Soy el proceso " <<yo<<" de "<<nproc<<endl;
  MPI_Finalize();
}
```

En el código anterior se nos presentan 4 de las funciones más utilizadas de MPI: `MPI_Init`() para iniciar la ejecución paralela, `MPI_Comm_size`() para determinar el número de procesos que participan en la aplicación, `MPI_Comm_rank`(), para que cada proceso obtenga su identificador dentro de la colección de procesos que componen la aplicación, y `MPI_Finalize`() para terminar la ejecución del programa.

Si utilizamos MPI, entonces los nombres de todas las funciones han de comenzar con `MPI_`, la primera letra que sigue siempre es mayúscula, y el resto son minúsculas. La mayoría de las funciones MPI devuelven un entero, que hay que interpretarlo como un diagnóstico. Si el valor devuelto es `MPI_SUCCESS`, la función se ha realizado con éxito. La palabra clave `MPI_COMM_WORLD` se refiere al comunicador universal, es decir, un comunicador predefinido por MPI que incluye a todos los procesos de nuestro programa. Se pueden definir otros comunicadores en MPI y todas las funciones de comunicación de MPI necesitan como argumento un comunicador.

3.5 Mecanismos de alto nivel en sistemas distribuidos

En el modelo de paso de mensajes caracterizado por comunicación síncrona y paso de mensajes unidireccional, cada canal de comunicación es utilizado para pasar información en una dirección solamente, entre un único proceso emisor y un único proceso receptor. Con estas primitivas de paso de mensajes de bajo nivel se puede implementar cualquier tipo de interacción entre procesos. Pero no se adaptan a todos los esquemas comunicación. Por ejemplo, para el esquema cliente-servidor se tiene una implementación demasiado forzada.

```
proceso cliente                    proceso servidor
 do                                      do
   TRUE ->                                 (i:0..n) condicion(i);
     send(servidor,peticion());              receive (cliente[i], peticion()) ->
     receive(servidor,respuesta);             realizar.servicio();
 od                                           send(cliente(i),resultado);
                                         od
```

La solución anterior produce un código poco seguro, sobre todo si las peticiones de servicio de los clientes han de aguardar un mensaje por parte del servidor que indique que se ha completado. De forma similar, el servidor puede verse afectado por un cliente no fiable que falle antes de recibir el segundo mensaje (`receive(servidor,respuesta)`). De darse esto se produciría el bloqueo del proceso servidor.

El par `(send(servidor,peticion()), receive(servidor,respuesta))` ha de ser una única transacción lógica y no es adecuado representarla como 2 operaciones de paso de mensajes síncronos independientes. Por lo tanto, el canal de comunicación entre el proceso cliente y el servidor debería soportar comunicación en los 2 sentidos.

3.5.1 El modelo de *llamadas remotas*

Es un modelo de comunicación de paso de mensajes síncrono, pero que tiene muchas de las características de las *llamadas a procedimiento* de los lenguajes secuenciales:

- Permite implementar de una forma flexible los procesos con una relación tipo cliente-servidor.
- Se permite a varios procesos llamar concurrentemente a un procedimiento que es poseído y controlado por otro proceso (comunicación muchos-a-1).
- Una llamada remota a procedimiento puede implicar paso de información en 2 direcciones.
- El procedimiento llamado encapsula una serie de instrucciones que son ejecutadas en nombre del proceso llamador (cliente), antes de que se devuelva ningún resultado.
- Este modelo admite varias implementaciones; de las cuales se van a estudiar: *llamada a procedimiento remoto* (RPC) y la *invocación remota* (o modelo basado en citas).
- *remoto* para las RPC's significa en un procesador distinto y *remoto* para las invocaciones remotas significa en un proceso distinto.

3.5.2 Llamada a Procedimiento Remoto *RPC*

Concepto de Procedimiento Remoto
Es un mecanismo que permite a un programa, ejecutándose en un nodo de una red, ejecutar un procedimiento en otra máquina. Las llamadas a procedimientos o métodos remotos tienen una sintaxis similar a la llamada a un procedimiento dentro de un programa secuencial, pero una

semántica completamente diferente. Ha de existir un programa ejecutántose en un procesador remoto (servidor) para ejecutar las llamadas. Podemos pensar en el procedimiento remoto como un *procedimiento global* que es llamado por los procesos clientes. El procedimiento es ejecutado por un proceso creado a tal efecto en el servidor que ejecuta su cuerpo y devuelve un mensaje con los resultados.

Algunas implementaciones de este modelo se han demostrado particularmente útiles y eficientes en sistemas distribuidos. Dicho modelo ha sido adoptado como mecanismo de comunicación a bajo nivel en versiones distribuidas del sistema operativo UNIX.

Una descripción general de la semántica de la *llamada a procedimiento remoto* es la siguiente:

- Se envían los argumentos de la llamada al servidor, bien directamente, o través de un proceso creado a tal efecto (*stub*).
- Se bloquea el proceso cliente que ejecuta la llamada. Este proceso puede dejar su procesador libre mientras se gestiona la llamada remota.
- Se reconstruyen los argumentos de la llamada en el servidor, se ejecuta el procedimiento y se reenvían los argumentos resultado al proceso cliente.
- El servidor suele ligar un nombre simbólico al puerto del que recibe las llamadas para ejecutar un procedimiento remoto determinado.
- Se pueden tener varias instancias de un mismo procedimiento ejecutándose concurrentemente, si cada llamada de un proceso cliente crea un nuevo proceso en el servidor. En este caso, las variables del procedimiento han de poder ser accedidas concurrentemente, por lo tanto hay que asegurar la exclusión mutua en dicho acceso.

3.5.3 Implementación del modelo en Java: RMI

La tecnología estándar que propone Java [Lea, 2001] para programación distribuida se denomina *Remote Method Invocation (RMI)*. Dicha tecnología está basada en el modelo de llamadas remotas y en la programación orientada a objetos, de ahí que se hable de invocación de métodos remotos.

Utilizando RMI y el lenguaje de programación Java en la parte del servidor se ha de programar una clase que implemente los métodos que van a ser llamados remotamente. La información concreta acerca de qué métodos de una clase pueden ser llamados por otros procesos no se encuentra dentro de la clase, sino que se ha de incluir en una sub-interfaz de `Remote`:

```
public interface Hello extends Remote{
  public String sayHello() throws java.rmi.RemoteException;
}
```

La sub-interfaz *Remote* permite, en el nivel de implementación, para cada servidor RMI, registrar los objetos disponibles y los nombres de los métodos que pueden ser llamados remotamente.

```
public class HelloImpl extends UnicastRemoteObject implements Hello {

  public HelloImpl() throws RemoteException{
    super();
  }
  public String sayHello() throws RemoteException{
    returns ‘‘Hello World!!!’’;
  }
  public static void main(String args[]){
    try{
      HelloImpl h = HelloImpl();
      Naming.rebind(‘‘hello’’, h);
      System.out.println(‘‘Hello server ready.’’);
    }
    catch(RemoteException re){
      ...
    }
    catch (MalformedURLException e){
      ...
    }
  }
}
```

Se ha de declarar un método constructor en la clase que contiene al método remoto, ya que la cláusula que especifica tirar la excepción es obligatoria.

En el método `main()` se liga el objeto devuelto por el constructor al nombre simbólico ‘‘`hello`’’. Esta asociación entre un nombre simbólico y un objeto es incluida en el registro de nombres del servidor y hecha accesible a futuros procesos clientes.

Los procesos clientes, para poder utilizar el servicio del método `sayHello()`, han de programar en su código:

```
public static void main (String args[]){
  System.setSecurityManager(new RMISecurityManager());

  try{
    Hello h = (Hello) Naming.lookup(‘‘rmi://ockham.ugr.es/hello’’);
    String message = h.sayHello();
    System.out.println(‘‘HelloClient:’’ + message);
  }
  catch(remoteException re){
   ...
  }
}
```

Según se puede ver representado en la figura 3.13, antes de que el servidor pueda comenzar a aceptar llamadas de los procesos clientes, se han de generar los *stubs* y los *skeletons*. Cada

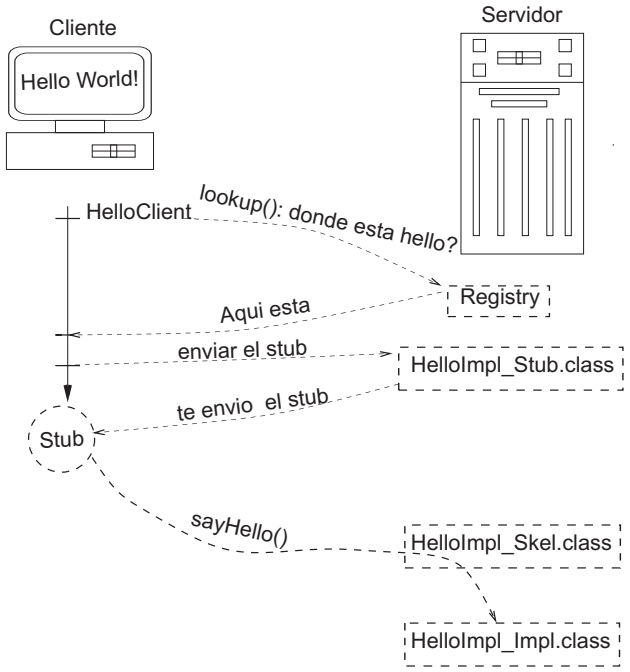

Figura 3.13: Representación de la ejecución de `HelloImpl.class`

stub contiene la signatura[6] de los métodos incluidos en la interfaz remota. Un *skeleton* tiene una función similar al *stub*, pero en la parte del servidor. Los *stubs* y *skeletons* son generados automáticamente en el servidor, a partir del código fuente de la clase (`HelloImpl` del ejemplo anterior).

Posteriormente habría que iniciar la ejecución del servidor de nombres (registry) y después lanzar el programa en el servidor (`java HelloImpl&`).

3.5.4 Semántica del paso de parámetros en RMI

Java, en las llamadas a métodos, pasa por referencia los parámetros referidos a objetos locales (no *por copia*). Si se mantuviera el mismo sistema de paso de parámetros para las llamadas a métodos remotos, el lenguaje sería poco útil para desarrollar programas distribuidos, ya que sería bastante ineficiente pasar por referencia objetos que contienen sólo datos, tales como: arrays, registros o cadenas. Si, por ejemplo, se pasa *por referencia* un array de 100,000 elementos a un método remoto, para procesar completamente dicho array se necesitaría realizar 100,000 accesos a una máquina remota.

Para evitar la disfunción que produciría disponer en Java sólo de *paso por referencia* en las llamadas a los métodos, cuando se programa con RMI, se establecen las siguientes reglas de paso de parámetros para métodos remotos:

- los argumentos de una llamada que sean de un tipo primitivo se pasarán *por copia*,
- los parámetros referidos a objetos se pasarán por referencia, o no, dependiendo de las interfaces que implementen las clases a las que pertenecen dichos métodos:

[6] nombre, lista y tipo de parámetros de una función o método

```
public interfaca PilaRemota extends
 Serializable{
   public int[ ] volcar()
      throws RemoteException;
        return celdas;
 }
 public class RMI_test{
  static PilaRemota s1 = null;
  static PilaRemota s2 = null;
   public static void main(String[ ] args){
  //objetos-pila creados en diferentes sitios
      s1.insertar(3);
      s2.insertar (3);
      int[ ] pila1 = s1.volcar();
      int[ ] pila2 = s2.volcar();
      pila1[0] = 100;
      pila2[0] = 100;
      System.out.println("pila1: "+s1.cima());
      System.out.println("pila2: "+s2.cima());
    }
}
```

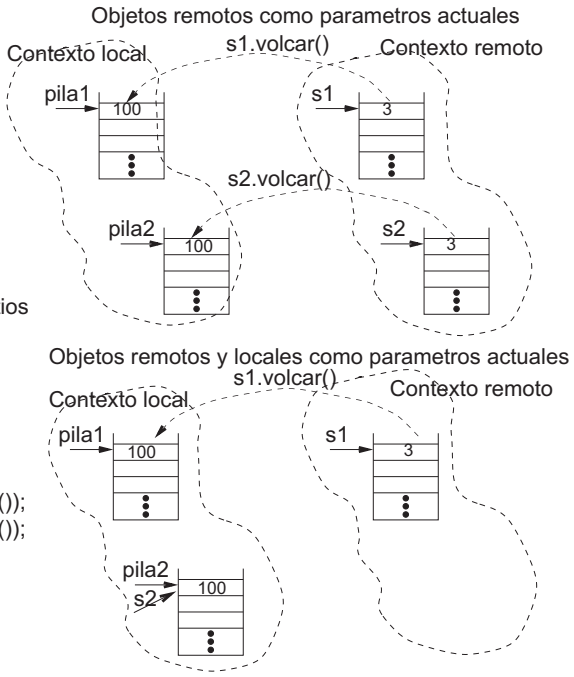

Figura 3.14: Paso de parámetros por referencia y copia en Java

- *remote interface:* se pasan por referencia,
- *serializable interface:* se pasan por copia.

Java permite la coexistencia de 2 semánticas distintas para la llamada a los métodos de una misma variable-objeto. La sintaxis para invocar a los métodos es la misma, independientemente de que el método pertenezca a la interfaz de un objeto local o remoto respecto del que lo llama, sin embargo, la ejecución de un método puede producir diferentes resultados, según sea local o remoto, según se muestra en la figura 3.14.

Puesto que Java permite el polimorfismo cuando se programa con RMI, no hay forma de averiguar de manera estática[7] si un objeto es local o remoto cuando se invocan sus métodos. Como consecuencia de ello, suele ser complicado programar aplicaciones distribuidas correctas, con la semántica del paso de parámetros en las invocaciones a métodos remotos de la definición actual del lenguaje Java.

3.5.5 Implementación del modelo con *invocación remota*

Son sentencias de un lenguaje de Programación Concurrente que permiten a un proceso llamar a un procedimiento que pertenece y es controlado por otro proceso. En este caso, el procedimiento remoto es una secuencia de una ó más instrucciones que pueden estar situadas en cualquier parte del código de un proceso.

A diferencia de las RPC's el proceso que llama y el proceso al cual pertenece el procedimiento remoto pueden ejecutarse por el mismo procesador.

[7]por ejemplo, siguiendo todas las referencias a los objetos de un programa.

Cada invocación remota se implementa definiendo un *punto de entrada* y una o varias *sentencias de aceptación* asociadas a dicha entrada. Ada [Barnes, 1994] es un lenguaje de programación que sigue este modelo y ha alcanzado mayor difusión hasta la fecha.

Definicion de los *puntos de entrada*

En un proceso se define un punto de entrada por cada uno de los procedimientos que pueden ser llamados desde otros procesos. Las entradas son definidas en el contexto del proceso que las posee: un proceso que hace una llamada tiene que indicar el punto de entrada y el proceso que la posee. A diferencia de las RPC's, las entradas no son nombres globales del sistema, por lo tanto, pueden estar repetidas en diferentes procesos.

Mientras que un canal podía tener como máximo un proceso esperando comunicación, los puntos de entrada pueden tener cualquier número de llamadas pendientes. Los datos se comunican entre los procesos a través de los parámetros declarados en los puntos de entrada; se sigue una notación similar a la declaración de parámetros en los procedimientos:

Los procesos clientes llaman a los puntos de entrada; ya no es necesario declarar canales entre cada cliente y el proceso servidor.

Por cada punto de entrada existe una cola que atiende las llamadas de los procesos según el orden de llegada (cola FIFO) para contener las llamadas pendientes de los procesos clientes.

```
PROCESO BUFFER
  ENTRY   depositar(dato:  tipo.dato);
  ENTRY   tomar(VAR x:tipo.dato);
```

No se especifica ningún orden de declaración de los procedimientos remotos; los procesos pueden hacer llamadas de sus puntos de entrada que presenten circularidad:

```
PROCESO.A          PROCESO.B
  ENTRY E;           ENTRY F;
BEGIN              BEGIN
  B.F;               A.E
END                END
```

Comunicación mediante citas

El código correspondiente a la invocación remota se ejecutará cuando el proceso que tiene definida la entrada lo permita, a diferencia del procedimiento llamado durante una RPC, que es ejecutado inmediatamente, puesto que se trata de una *llamada a procedimiento* similar a las de los lenguajes secuenciales.

Por cada punto de entrada, en el códido de un proceso, hay que declarar al menos una sentencia de aceptación que contiene el código a ejecutar cuando se produzca la llamada al

punto de entrada. La ejecución de la sentencia de aceptación bloquea hasta que se produce la llamada de otra tarea al punto de entrada (si existieran varias llamadas esperando en la cola del punto de entrada, se atendería la primera).

Por lo tanto, la sentencia de aceptación se ejecuta sólo cuando las dos procesos están preparados para la comunicación:

```
Proceso P                        Proceso P'
 ENTRY E(...)                      ....
 ...                              BEGIN
 BEGIN                              P.E(...);
   ACCEPT E(...);                   ...
   ...                             END;
 END;
```

La comunicación que se establece entre el proceso `P` y el proceso `P'` se denomina *cita* o *rendez-vous*, dicho mecanismo pertenece a un modelo síncrono de comunicación.

Antes y después de producirse la cita los dos procesos se ejecutan asíncronamente.

La sintaxis de la sentencia de aceptación varía si se permite compartir variables globales entre procesos concurrentes, en este caso, se ha de añadir un cuerpo a dicha sentencia, esto es, una zona de código que se ejecuta en exclusión mutua.

3.6 Problemas resueltos

Ejercicio 1

Simular un semáforo general con citas.

Solución:

```
wait()                          signal()                        proceso
------                          --------                        -------
ENTRY semaforo_wait(int S);     ENTRY semaforo_signal(int S)    semaforo_signal(S)
BEGIN                           BEGIN                           ...
  accept_semaforo_wait(S);        accept_semaforo_signal(S);    semaforo_wait(S)
    when s>0                      do
    do                              S++;
        S--;                      end do;
    end do:                     END
END
```

Ejercicio 2

Simular una cita mediante semáforos.

Solución:

```
inic(S1,0)
inic(S2,0)
inic(mutex,1)


P1                  P2
--                  --
wait(S1);           signal(S1)
wait(mutex);        wait(S2)
//cita
signal(mutex);
signal(S2);
```

Ejercicio 3

Un tren tiene N asientos libres. Un pasajero sube al tren y ocupa su plaza, tras lo cual quedaría 1 asiento libre menos. Si todas las plazas del tren se ocupan o si durante 30 minutos no toma asiento ningún nuevo pasajero, entonces el tren realiza 1 viaje, tras lo cual vuelven a quedar N asientos libres. Suponer que para realizar 1 viaje, el tren ha de tener ocupado al menos 1 asiento; ya que si el tren tuviera todos sus asientos libres, se quedaría esperando indefinidamente a que se subiera 1 pasajero y ocupara su asiento. Suponer las siguientes condiciones para poder programar la actuación del tren y de los pasajeros con *citas*: 1) hay un número ilimitado de pasajeros y cada uno realiza 1 solo viaje, Suponer que 1 proceso pasajero cuando consigue asiento termina; 2) el tren está realizando viajes continuamente. Se pide resolver el problema en los 2 casos siguientes:

- **Se supone 1 único tren, por lo tanto se puede programar su actuación como un solo proceso servidor.**
- **Ahora se suponen 2 trenes. En este caso si 1 pasajero intenta subir al tren y no puede, esperará 1 minuto, tras lo cual, lo intentará con el segundo tren; si tampoco pudiera subir al segundo, lo volvería a intentar con el primer tren transcurrido 1 minuto y así sucesivamente.**

Solución:

```
Tren
----
ENTRY llega();
libres=0;
do{
    SELECT
        when libres>0 ->
            accept llega(); libres--;
    or
        when libres == 0;
            delay 0;
            // realizar viaje
            sleep(random()%MAX);
            libres=N;
    or
        when libres < N ->
            delay 30*60; // espera 30 min
            // realizar viaje
            libres=N;
    END SELECT
}while(true);
```

```
Pasajero (caso 1)
------------
do{
    Tren.llega();
}while(true);

Pasajero (caso 2)
------------
do{
    SELECT
        Tren1.llega(); return;
    or
        delay 1*60; // espera
    END SELECT
    SELECT
        Tren2.llega(); return;
        delay 1*60;
    END SELECT
}while(true);
```

Ejercicio 4

Se tienen N procesos cliente que interactúan con el proceso servidor de 1 cajero automático. Los procesos cliente obtienen dinero del cajero realizando la siguiente operación:

- **Informan de su identidad al cajero y solicitan una cantidad de dinero.**
- **El cajero responde con la cantidad solicitada si el cliente tiene suficiente saldo, si no, el cajero responde denegando la petición al cliente.**
- **Para ingresar dinero en el cajero los procesos sólo tienen que identificarse e indicar la cantidad ingresada.**
- **Suponer que cada cliente tiene inicialmente 10 unidades de saldo y que el cajero posee 100 unidades de efectivo para servir las peticiones de los clientes.**

Cuando el cajero agota completamente las 100 unidades de efectivo, no podrá servir peticiones de ningún tipo hasta pasada 1 hora, transcurrido ese tiempo se vuelven a reponer las 100 unidades de efectivo. Los ingresos de los clientes no incrementan las unidades de efectivo que tiene el cajero.

Solución:

```
ENTRY sacar(num_cliente: in int; cantidad:in/out int));
ENTRY ingresa(num_cliente: in int; cantidad:in/out int));
int efectivo=100;
int saldo[num_cliente];
do{
    SELECT
        when (efectivo>0) ->
            accept sacar(num_cliente,cantidad) do
                if (ifectivo>=cantidad and saldo[num_cliente>=cantidad]){
                    efectivo-=cantidad;
                    saldo[num_cliente]-=cantidad;
                } else cantidad=-1;
            end do;
    or
        when (efectivo>0)
            accept ingresar(num_cliente,cantidad)do
                saldo[num_cliente]+=cantidad;
            end do;
    or
        when (efectivo==0)
            delay 60*60;
            efectivo=100;
    END SELECT
}while(true);
```

Ejercicio 5

Programar una versión distribuida del problema de la cena de los 5 filósofos, suponiendo que se dispone de 5 procesos *filósofo*, 5 procesos *tenedores* cuyo comportamiento es el de un semáforo y un proceso habitación que limita la entrada a un máximo de 4 filósofos para que no se llegue a la situación de interbloqueo (cada uno de los filósofos ha cogido 1 tenedor y ninguno puede adquirir nunca más el que le falta para poder comer de la fuente de los espaguetis). El comportamiento de un proceso filósofo es un ciclo indefinido que repite: *pensar, entrar en la habitación, coger su tenedor izquierdo, coger su tenedor derecho, comer, soltar tenedor izquierdo, soltar tenedor derecho, salir de la habitación.*

Solución:

```
Process Tenedor(i)
------------------
do{
    receive(Filosofo(i),coger);
    receive(Filosofo(i),dejar);
}while(true)

Process Filosofo(i)
-------------------
do{
    send(Habitacion, entrar);
    send(Tenedor(i), coger); send(Tenedor((i+1)%5), coger);
    send(Tenedor(i), dejar); send(Tenedor((i+1)%5), dejar);
    send(Habitacion, salir);
}while(true)

Process Habitacion
------------------
do{
    SELECT
        when ocupado<4;
            receive(Filosofo(i), entrar);
            ocupado++;
        or
            receive(Filosofo(i),salir);
            ocupado--;
    END SELECT;
}while(true)
```

Ejercicio 6

Suponer que en un centro de proceso de datos hay 2 impresoras `A` y `B` que son similares, pero no idénticas. Tres clases de procesos clientes utilizan las impresoras: aquellos que necesitan utilizar la impresora `A`, los que necesitan utilizar la impresora `B` y aquellos otros que pueden utilizar cualquiera de las 2 impresoras: `A` o `B`. Cada tipo de proceso cliente ejecuta una llamada de petición de impresora y otra llamada para liberarla. Programar con citas un proceso servidor para asignar las impresoras anteriores. Se supone que 1 proceso cliente no puede quedarse con una impresora para siempre.
Solución:

```
Clientes A                  Clientes tipo=A|B
----------                  -----------------
do{                         do{
    servidor.cedeA();           servidor.cede(tipo);
    // usa impresora            // usa impresora
    servidor.libera(A);         servidor_libera(tipo);
}while(true);               }while(true);

Servidor
--------
ENTRY cedeA();
ENTRY cedeB();
ENTRY cede(tipo:out tipo);
ENTRY libera(tipo: int tipo);
bool libreA=true;libreB=true;
do{
    SELECT
        when libreA ->
            accept cedeA();libreA=false;
    or
        when libreB ->
            accept cedeB();libreB=false;
    or
        when (libreA or libreB) ->
            accept cede (tipo: out tipo) do
                if(libreA){libreA=false;tipo=A;}
                else{libreB=false;tipo=B;}
    or
        accept libera(tipo: int tipo) do
            if tipo==A libreA=true;
            else libreB=true;
        end do;
    END SELECT
while(true);
```

3.7 Problemas propuestos

1. ¿Podría llegar a bloquearse temporalmente la orden de envío `send(...)` en el paso de mensajes asíncrono (*no bloqueante*) y buferizado? A qué se debería dicho bloqueo si es que crees que se puede producir.

2. ¿Qué operaciones de paso de mensajes de la biblioteca OpenMPI utilizarías para implementar el mecanismo de sincronización denominado *cita* entre 2 procesos?

3. ¿Qué problema tendría el siguiente código, que programa 2 operaciones de paso de mensajes *no bloqueante* y *no buferizado*, respecto de los valores finales de las variables `x, y`, declaradas en el programa y qué funciones de OpenMPI programarías en el espacio en blanco para resolver dicho problema?

```
int main(int argc, char *argv[]) { int rank, size, vecino, x, y;
MPI_Status status; MPI_Request request_send,request_recv;
MPI_Init(&argc, &argv); MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size( MPI_COMM_WORLD, &size ); y=rank*(rank+1); if (rank
mod 2 == 0) vecino=rank+1; else vecino=rank-1;
// Las siguientes operaciones pueden aparecer en cualquier orden
MPI_Irecv(&x,1,MPI_INT,vecino,0,MPI_COMM_WORLD,&request_recv);
MPI_Isend(&y,1,MPI_INT,vecino,0,MPI_COMM_WORLD,&request_send );
...
```

4. Explicar qué ocurre si en la ejecución de una orden de selección no-determinista todas las alternativas tienen condiciones evualadas a verdadero y una sentencia de aceptación (a continuación de cada condición), pero ningún otro proceso ha realizado todavía la llamada requerida por alguna de estas alternativas.

5. ¿Cuál sería el resultado de la ejecución de una selección no–determinista si todas las condiciones–guarda de sus alternativas se evalúan como falsas?

6. Considérense las siguientes tareas en un lenguaje distribuido con invocaciones remotas:

```
1 Process P1
2 do{
3 P2.A;
4 P3.B;
5 P4.G;
6 P3.E;
7 accept F;
8 P3.C;
9 }while(true);
10End P1;
```

```
1 Process P2
2 do{
3  select
4   accept A do
5     P3.D;
6   or
7   accept H do
8    P3.E;
9 end select;
10 while{true};
11End P2
```

```
1 Process P3
2 do{
3  accept D;
4  accept B;
5  accept E do
6  P1.F;
7 End P3;
8 accept C;
9 }while(true);
10End P3;
```

```
1 Process P4
2 do{
3  accept G;
4  T2.H;
5 }while(true);
6 End P4;
```

¿Puede producirse bloqueo global de los procesos anteriores? Si fuera así, indicar una secuencia de entrelazamiento que lleve a tal situación. Indicar el proceso y el número de línea de cada sentencia cuya ejecución provoque el bloqueo de cada proceso.

7. Programar la selección no–determinista del proceso `S1` a continuación para que se mantenga suspendida esperando la llamada de los otros procesos a su punto de entrada `E1()` durante 10.0 unidades de tiempo, transcurrido ese tiempo sin que se acepte la espera selectiva terminará.

```
Proceso S1(){                    Proceso S2(){
  ENTRY E1();                      ENTRY E2();


   SELECT                           SELECT
      ACCEPT E1();                     ACCEPT E2();



   END SELECT;                      END SELECT;
```

8. Programar la selección no–determinista del proceso `S2` para que si no hay ninguna llamada pendiente al punto de entrada `E2()`, por parte de algún cliente, la espera selectiva se cancele inmediatamente y, tras un retraso de 20.0 unidades de tiempo, la orden termine.

9. ¿Por qué en una orden de selección no determinista se pueden programar varias alternativas `delay <retraso>`, pero son incompatibles con la claúsula `ELSE`?

10. Dos tipos de personas, representados por los tipos de procesos `A` y `B` entran en 1 habitación. La habitación tiene una puerta muy estrecha por la que cabe 1 sola persona. La actuación de las personas es la siguiente:
    - Una persona de tipo `A` no puede abandonar la habitación hasta que encuentre a 10 personas de tipo `B`
    - 1 persona de tipo `B` no puede abandonar la habitación hasta que no encuentre a 1 persona d tipo `A` y otras 9 personas de tipo `B`.

    Siempre se ha de cumplir la siguiente condición: *en la habitación no hay personas de tipo* `A` *o hay menos de 10 personas de tipo* `B`. Si en algún momento no se cumple, tendrían que salir 1 persona de tipo `A` y 10 personas de tipo `B` para que se volviera a cumplir la condición. Cuando las personas salen de la habitación, no pueden entrar nuevas personas de ningún tipo hasta que salgan todos.

    Implementar una solución al problema anterior con citas. Para ello será necesario escribir un proceso servidor encargado de llevar a cabo la sincronización global de forma correcta. El código de los procesos cliente podría seguir el esqueleto mostrado más abajo. Las entradas `llamaA` y `llamaB` sirven para que los procesos cliente avisen al proceso servidor que desean sincronizarse de acuerdo con el esquema de sincronización indicado para resolver este problema. Las entradas `esperaA` y `esperaB` sirven para bloquear a los procesos clientes hasta que el número de clientes que han llamado a `llamaA` y `llamaB` sea el requerido para formar un grupo de personas de salida.

```
Process type tipoA {
   servidor.llamaA;
   servidor.esperaA;
   ...
}
```

```
Process type tipoB {
   ...
   servidor.llamaB;
   servidor.esperaB;
}
```

11. Terminar de programar la orden de selección siguiente, que implementa un semáforo binario de exclusión mutua ($0 \leq s \leq 1$), con sólo 2 puntos de entrada.

```
Proceso Mutex (int i){
 ENTRY wait();
 ENTRY post();
 int s=0;
  do{
    SELECT


    END SELECT;
 }while(true);
}
```

Con la implementación del semáforo anterior se pretende resolver el problema del acceso a la sección crítica por parte de dos procesos concurrentes. Para inicializarlo correctamente, 1 de los 2 procesos ha de ejecutar primero la llamada `Mutex(0).post()`:

```
void * p1( void * )
{
  //Solo lo inicializa 1 proceso
  Mutex(0).post();
  do {
     //Fuera de la seccion critica
     Mutex(0).wait();
     //Acceder a la
     //seccion critica
     //
     Mutex(0).post();
  }while(true);
  return NULL ;
}
```

```
void * p2( void * )
{
  do {
      //Fuera de la seccion critica
      Mutex(0).wait();
      //Acceder a la
      //seccion critica
      //
      Mutex(0).post();
  }while(true);
  return NULL ;
}
```

12. Programar el controlador de una máquina dispensadora de bebidas automática. Para cada bebida, acepta peticiones de los clientes: `ACCEPT peticion(num_bebida : IN int)`. Puede llegar a tener en total N latas de bebidas disponibles de los diferentes tipos. Cuando se agotan todas las latas entra en un periodo de mantenimiento durante 2 horas, tras el cual se reponen todas y la máquina vuelve a estar en servicio. Cada 24 horas se reponen las latas que faltan si se ha consumido alguna; esta operación tarda en realizarse 1/2 hora.
    - Completar el código de controlador de la máquina de bebidas, programando una selección no determinista, que se repite continuamente de acuerdo con las condiciones anteriores del problema:
    - Suponer ahora que hay 2 máquinas de bebidas en el edificio, con un funcionamiento totalmente equivalente. En este caso, si un cliente intenta obtener una bebida de 1 de las máquinas y no puede (la máquina está fuera de servicio), esperará 1 minuto, tras lo cual se desplazará al otro extremo del edificio para intentar sacar la bebida de la otra máquina. Si tampoco pudiera obtener una bebida, espera 1/2 hora a que la segunda máquina vuelva a estar en servicio. Si tampoco obtuviera la bebida con

la segunda máquina, lo volvería a intentar con la primera máquina, repitiendo las mismas acciones que realizó anteriormente, y así sucesivamente.

13. Conceptos generales sobre paso de mensajes. Responda Verdadero/Falso:

    (a) Los sistemas de paso de mensajes sólo pueden utilizarse en sistemas en los que no existe memoria compartida entre los procesos.

    (b) Las operaciones de paso de mensajes pueden ser utilizadas como primitivas de comunicación y sincronización, independientemente de si la plataforma es distribuida (*multicomputador*) o centralizada (*multiprocesador*).

    (c) Los procesos distribuidos y las hebras no son compatibles, por eso una aplicación que programe operaciones de paso de mensajes sólo admite que sus procesos estén ubicados en computadores diferentes.

    (d) Los canales de comunicación entre procesos distribuidos sólo pueden tener 1 proceso en cada extremo (*emisor* y *receptor*)

14. Responda Verdadero/Falso a las siguientes afirmaciones sobre los diferentes tipos de paso de mensajes:

    (a) Las ordenes de paso de mensajes sícronas nunca pueden llegar a bloquearse porque no se llenan búferes de mensajes que puedan desbordarse.

    (b) La operaciones de paso de mensaje síncrono (`Ssend()` y `Recv()`) nunca producen error si la sincronización de los procesos es correcta.

    (c) No hay forma de prevenir que las operaciones de recepción de mensajes síncrona, sin búfer, pueda bloquear a un proceso servidor si el proceso emisor ha terminado o ha abortado su ejecución.

    (d) En el caso de que un proceso intente enviar un valor a otro de forma asíncrona (no bloqueante y sin búfer) no debe utilizar los datos que va a enviar hasta que una función que ejecuta posteriormente le indique que puede hacerlo.

    (e) En el caso de paso de mensajes asíncrono el proceso receptor puede modificar los datos recibidos (`vector`) en cuanto vuelva la función `IRecv(&vector, N , ...)`.

15. Responda Verdadero/Falso a las siguientes afirmaciones sobre la sentencia `Select`:

    (a) Las condiciones–guardas de las alternativas sólo se evalúan 1 vez.

    (b) Si una condición se evalúa como falsa, se provoca el bloqueo indefinido del proceso que llama a la cita.

    (c) Si todas las condiciones de las alternativas regulares del `select` son falsas entonces se provoca un error de ejecución del programa.

    (d) Es conveniente programar siempre una claúsula `ELSE` en cada sentencia `Select` para evitar que se produzca el error si todas las alternativas están cerradas en 1 ejecución.

16. Responda Verdadero/Falso a las siguientes afirmaciones sobre la sentencia `Accept`:

    (a) El resultado del proceso $\texttt{P}_1$ a la orden de aceptación `ACCEPT f(IN x1, IN/OUT x2)` que ejecuta el proceso P_2 sólo tiene efecto en el proceso P_2, que luego envía dicho resultado al proceso $\texttt{P}_1$.

(b) Los procesos $\texttt{P}_1$ :: $\texttt{P}_2$.f(a, b) y $\texttt{P}_2$ :: ACCEPT f(IN $\texttt{x}_1$, IN/OUT $\texttt{x}_2$) no sólo se sincronizan mediante una cita, la ejecución de la función $f(IN\ x_1,\ IN/OUT\ x_2)$ se realiza localmente por el procesador que ejecuta el proceso $\texttt{P}_2$ pero cambia el estado del proceso remoto $\texttt{P}_1$.

(c) Después de ejecutar la función $f(IN\ x_1,\ IN/OUT\ x_2)$, el proceso $\texttt{P}_2$ envía los valores de los argumentos x y y de vuelta al proceso $\texttt{P}_1$.

(d) El proceso $\texttt{P}_2$ adquiere una dirección de memoria que referencia a la variable $\texttt{x}_2$ y que le envía el proceso $\texttt{P}_1$.En dicha dirección de memoria se asigna le valor final de la variable.

17. Responda Verdadero/Falso a las siguientes afirmaciones sobre las alternativas de la sentencia Select:

```
Proceso S1(){                    Proceso S2(){
  ENTRY E1();                      ENTRY E2();

   SELECT                           SELECT
                                     when C=>
    when C =>                          ACCEPT A;
    ACCEPT A;

  or DELAY 10.0;                   or DELAY 20.0;
      DELAY 10.0;
  END SELECT;                        END SELECT;
```

(a) La sentencia Select (S1) sólo terminaría si no se ha producido ninguna llamada a su punto de aceptación transcurridas 20.0 unidades de tiempo.

(b) Si la condición C de la sentencia Select (S1) se evalúa como falsa, entonces la ejecución de ambas sentencias Select es indistinguible.

(c) La sentencia Select (S1) siempre esperará 10.0 unidades de tiempo las llamadas a su punto de aceptación.

(d) Si no se producen llamadas a su punto de aceptación en ninguna de las sentencias Select anteriores en un plazo de 20.0 unidades de tiempo, el funcionamiento de ambas sentencias Select resulta ser el mismo.

18. Si existe un proceso $\texttt{P}_1$ que llama a un método remoto o.m(...) de un objeto o de la clase C, que se encuentra implementada en el ordenador donde se ejecuta el proceso $\texttt{P}_2$ (o *sirviente*). Para que la llamada del proceso $\texttt{P}_1$ al método remoto o.m(...) pueda se resuelta con el protocolo RMI, se ha de verificar una de las siguientes afirmaciones:

(a) Los procesos $\texttt{P}_1$ y $\texttt{P}_2$ han de ejecutarse en el mismo ordenador.

(b) El proceso $\texttt{P}_1$ sólo ha de conocer la interfaz remota que define los métodos públicos que implementa la clase C.

(c) El proceso $\texttt{P}_1$ ha de tener acceso siempre a un *stub* local del objeto remoto o, que entienda los mensajes que el proceso anterior envía al objeto o.

(d) Que el proceso P_1 tenga acceso a la interfaz remota aludida y también a 1 objeto *stub* que represente al objeto remoto de la clase que implementa el citado método.

19. Considérense las siguientes tareas en un lenguaje distribuido con invocaciones remotas:

```
1 Process P1
2 do{
3 P2.A;
4 P3.B;
5 P4.G;
6 P3.E;
7 accept F;
8 P3.C;
9 }while(true);
10End P1;
```

```
1 Process P2
2 do{
3  select
4   accept A do
5     P3.D;
6   or
7   accept H do
8    P3.E;
9 end select;
10 }while(true);
11End P2
```

```
1 Process P3
2 do{
3  accept D;
4  accept B;
5  accept E do
6  P1.F;
7 End P3;
8 accept C;
9 }while(true);
10End P3;
```

```
1 Process P4
2 do{
3  accept G;
4  T2.H;
5 }while(true);
6 End P4;
```

¿Puede producirse bloqueo global de los procesos anteriores? Si fuera así, indicar una secuencia de entrelazamiento que lleve a tal situación. Indicar el proceso y el número de línea de cada sentencia cuya ejecución provoque el bloqueo de cada proceso.

20. Completar el código de controlador de una máquina de bebidas, programando una selección no determinista, que se repite continuamente de acuerdo con las condiciones del problema:

(a) Para cada bebida, acepta peticiones de los clientes: `ACCEPT peticion(num_bebida : IN int)`.

(b) Puede llegar a tener en total N latas de bebidas disponibles de los diferentes tipos.

(c) Cuando se agota todo el suministro, la máquina entra en un periodo de mantenimiento durante 2 horas, tras el cual se reponen todas las latas y la máquina vuelve a estar en servicio.

(d) Cada 24 horas se reponen las latas que faltan si se ha consumido alguna; esta operación tarda en realizarse 1/2 hora.

```
Proceso Cliente (int i) {
  int pide;//5 tipos de bebidas
  pide= random()%5;
  do{
    Controlador.peticion(pide);
    //pasa a beberla antes de pedir otra
    delay random()*i%MAX;
  }while(true);
} Proceso Controlador{
   ENTRY peticion(num\_bebida : IN int);
   //Numero total de latas en la maquina
   int Latas=N;
  public void run(){
```

```
      do{
         SELECT
             /*
              completar
              */
          END SELECT;
       }while(true);
    }//fin Controlador
```