

Universidad Simón Bolívar

Departamento de Computación y T.I

Sistemas de operación III

CI-4822

Comunicación entre Procesos
por pase de mensajes

Prof. Yudith Cardinale
Ene - Mar 2011

Introducción

- Comunicación entre Procesos:
 - Memoria Compartida: sistemas centralizados, sistemas paralelos
 - Cliente-servidor: sistemas distribuidos (RPC, RMI)
 - Pase de mensajes: sistemas centralizados, distribuidos y paralelos
 - Comunicación basada en *streams*: sistemas distribuidos. El tiempo y orden de los mensajes es vital (transferencia de audio y video)
 - Comunicación en grupo: sistemas centralizados, distribuidos y paralelos

Introducción

- Comunicación por Memoria Compartida:
 - El sistema de operación se encarga de proveer el servicio de comunicación
 - Resuelven problemas de sincronización y exclusión mutua
 - Existen mecanismos de bajo nivel (semáforos, contadores de eventos y secuenciadores) y de alto nivel (monitores)

Introducción

- Comunicación por Pase de Mensajes:
 - Para aplicaciones en Sistemas Distribuidos y Paralelos
 - Memoria Distribuida
 - Se puede usar combinado con memoria compartida
 - Soportado por llamadas al sistema de operación o a través de librerías especiales
 - Las primitivas principales son:
`send(destination, this_msg, msg_length)`
`receive(source, a_msg, &how_long)`

Introducción

- Sistemas donde la comunicación por memoria compartida es apropiada:
 - Sistemas con poca protección (PCs)
 - Sistemas en tiempo real
 - Sistemas con soporte de Multi-hilos
 - Sistemas uniprocador o multiprocesadores con memoria compartida

Introducción

- Sistemas donde la comunicación por memoria compartida no es apropiada:
 - Sistemas protegidos (sistemas multiusuarios)
 - Sistemas con computadores independientes conectados por redes
 - Sistemas con soporte de migración

Aspectos relacionados a la comunicación por pase de mensajes

- Pérdida de mensajes: se puede usar Acks e intervalos de tiempos para retransmitir
- Se pierden los Acks: se puede usar número de secuencias de mensajes y protocolos de reconocimiento
- Confiabilidad: autenticación y encriptamiento

Aspectos relacionados a la comunicación por pase de mensajes

- Tamaño del mensaje:
 - Fijo: es fácil para el sistema de operación pero difícil para el programador. Ocurre fragmentación interna
 - Variable: es fácil para el programador pero difícil para el sistema de operación. Ocurre fragmentación externa.
- Enlaces o canales de comunicación

Enlaces de comunicación

- Enlaces físicos: memoria, buses, cable coaxial, microondas, fibra óptica, etc.
- Enlaces lógicos: provistos a nivel de software y sirven para administrar la comunicación
- Aspectos importantes:
 - ¿Cómo se establecen?
 - ¿A cuántos procesos está asociado?
 - ¿Cuántos enlaces son posibles entre cada par de procesos?
 - Capacidad del enlace y tamaño del mensaje
 - Es bidireccional o unidireccional

Enlaces de comunicación

- De acuerdo a cómo son definidos los enlaces, se tienen diferentes implementaciones de los *send* y *receive*:
 - Comunicación directa o indirecta
 - Comunicación simétrica o asimétrica
 - Comunicación síncrona o asíncrona
 - Comunicación transitoria o persistente
 - Con buffering explícito o automático
 - Enviar los mensajes por copia o por referencia

Comunicación Directa Simétrica

- Nombran explícitamente al proceso
- No hay entidades intermediarias
- El receptor debe conocer la identidad de todos los posibles emisores (es una mala solución para servidores)
- El enlace se establece automáticamente
- Un enlace se asocia sólo con dos procesos
- Entre cada par de procesos existe sólo un enlace
- Los enlaces pueden ser unidireccionales o bidireccionales

```
send(tothisprocess, this_msg, msg_length)  
receive(fromthisprocess, a_msg, &how_long)
```

Comunicación Directa Simétrica: ejemplos

```
...  
void producer() {  
    int item;  
    message m;  
    while (true) {  
        produce_item(&item);  
        receive(consumer,&m);  
        built_message(&m,item);  
        send(consumer,&m);  
    }  
}
```

```
...  
void consumer() {  
    int item;  
    message m;  
    for (i=0;i <N; i++)  
        send(producer,&m);  
    while (true) {  
        receive(producer,&m);  
        extract_item(&m,&item);  
        send(producer,&m);  
        consumer_item(item);  
    }  
}
```

```
}
```

Comunicación Directa Simétrica: ejemplos

```
...  
void producer() {  
    int item;  
    message m;  
    repeat  
        ...  
        produce_item(m);  
        ...  
        send(consumer,m);  
    until false;  
}
```

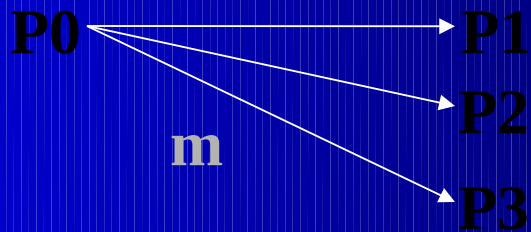
```
...  
void consumer() {  
    int item;  
    message m;  
    repeat  
        receive(producer,m);  
        ...  
        consume_item(m);  
        ...  
    until false;  
}
```

Comunicación Directa Simétrica

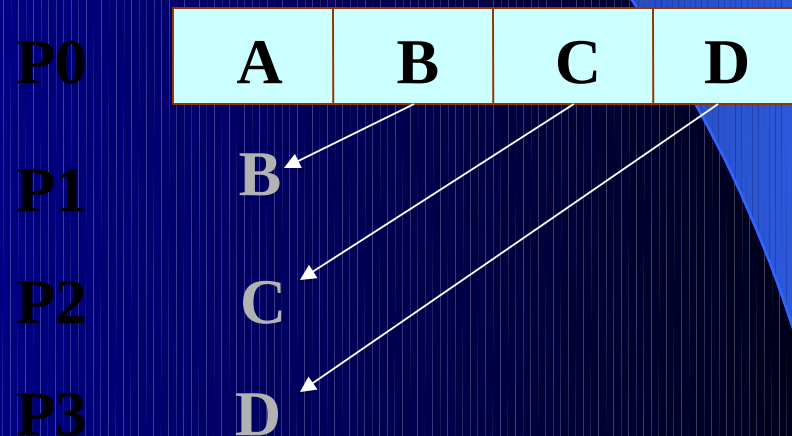
- La comunicación directa también es **simétrica** por ser comunicación uno-a-uno.
- La comunicación directa **asimétrica** permite comunicar más de 2 procesos al estilo uno-a-muchos.
- La comunicación directa permite comunicación transitoria en la cual los mensajes son mantenidos por el subsistema de comunicación sólo mientras se ejecutan en el emisor y receptor

Comunicación Directa Asimétrica

- *broadcast*($P0, m$) o *multicast*($P0, m$):



- *scatter*($P0, m[i]$):



Comunicación Directa Asimétrica

➤ *gather(P0,m)*: P0



P1

B

P2

C

P3

D

➤ *allgather()*, *alltoall()*

➤ *send(P,m)* *id=receive(m)*: el proceso P está listo para recibir un mensaje de cualquier proceso.

Comunicación Indirecta

- *Send (M, mensaje) Receive(M, mensaje)*
- Un enlace se establece entre un par de procesos sólo si comparten un buzón (*mailbox*), un puerto o un pipe.
- Un enlace se puede asociar con más de un par de procesos
- Entre cada par de procesos puede haber más de un enlace
- Los enlaces pueden ser unidireccionales o bidireccionales
- Permite comunicación persistente: el mensaje se puede mantener en el subsistema de comunicaciones

Comunicación Indirecta

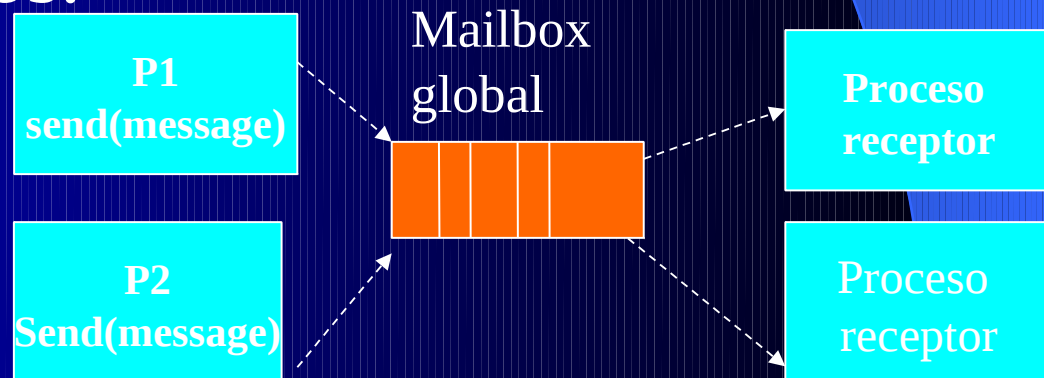
- ¿Qué sucede si varios procesos leen a la vez?: Puede haber problemas de inconsistencia. Se puede resolver:
 - Permitiendo sólo un enlace asociado a cada par de procesos
 - Permitiendo sólo un receive ejecutarse a la vez
 - Permitiendo que el sistema de operación seleccione arbitrariamente el proceso que hará la recepción

Comunicación Indirecta: Buzón o Mailbox

- Propiedad del subsistema de comunicación, el cual provee llamadas para:
 - Crear un buzón
 - send, receive a través del buzón
 - Destruir un buzón
 - Protección por grupos
- Puede ser usado por múltiples enviados y receptores que no necesariamente se conocen
- Son bidireccionales

Comunicación Indirecta: Buzón o Mailbox

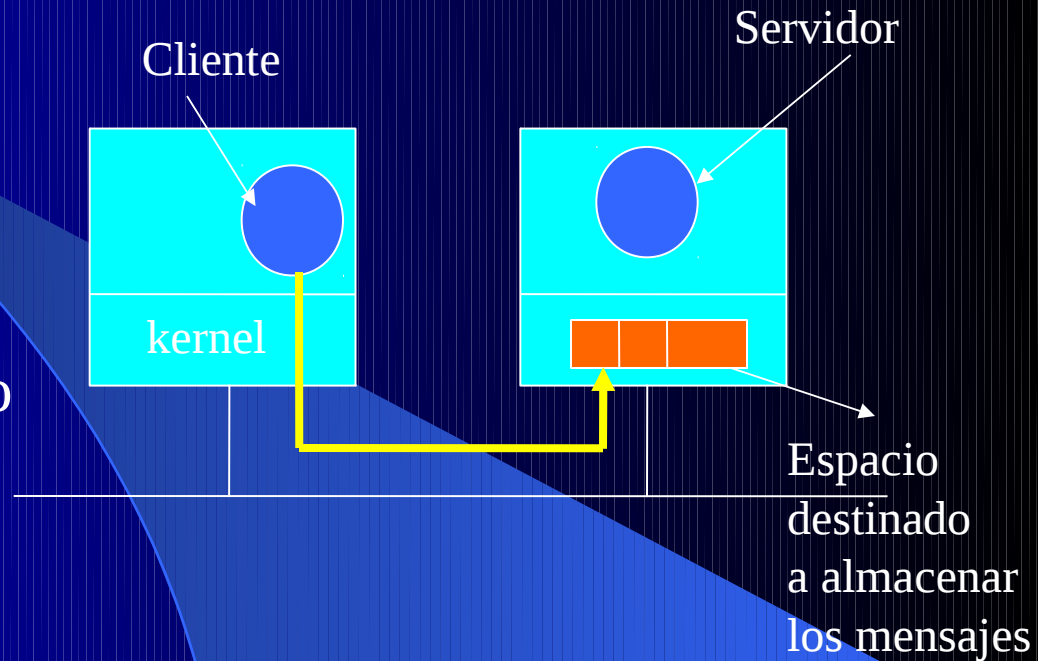
- Se puede añadir fácilmente un nuevo servidor en sustitución de uno que ha fallado
- Este mailbox persiste aunque finalicen los procesos que solicitaron su creación.
- El sistema tiene que soportar la existencia de un objeto cuyo nombre pueden conocer el resto de los procesos.



Comunicación Indirecta: Buzón o Mailbox

¿Qué pasa si el mailbox está lleno?

- Borrar el mensaje
- El kernel espera un tiempo hasta contar con espacio en el buffer.
- No dejar a ningún proceso enviar mensajes si no hay espacio



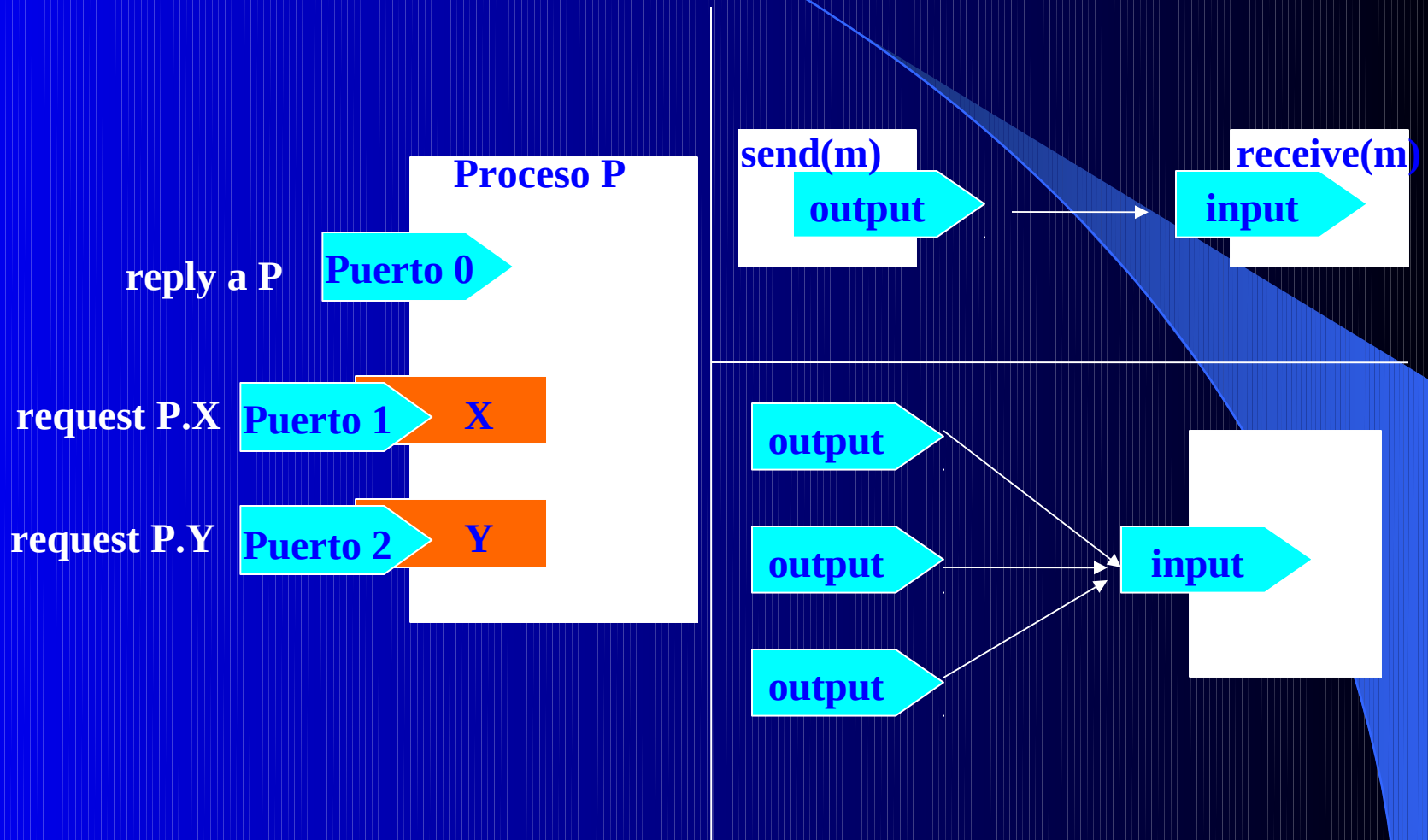
Comunicación Indirecta: Puertos

- Propiedad de los procesos. El proceso que creó el puerto (y su descendencia) son los únicos procesos que pueden utilizarlo.
- Cuando el proceso muere, muere el puerto
- Son unidireccionales
- Un proceso puede definir múltiples puertos
- El **puerto** tiene exactamente un receptor pero puede tener múltiples emisores.

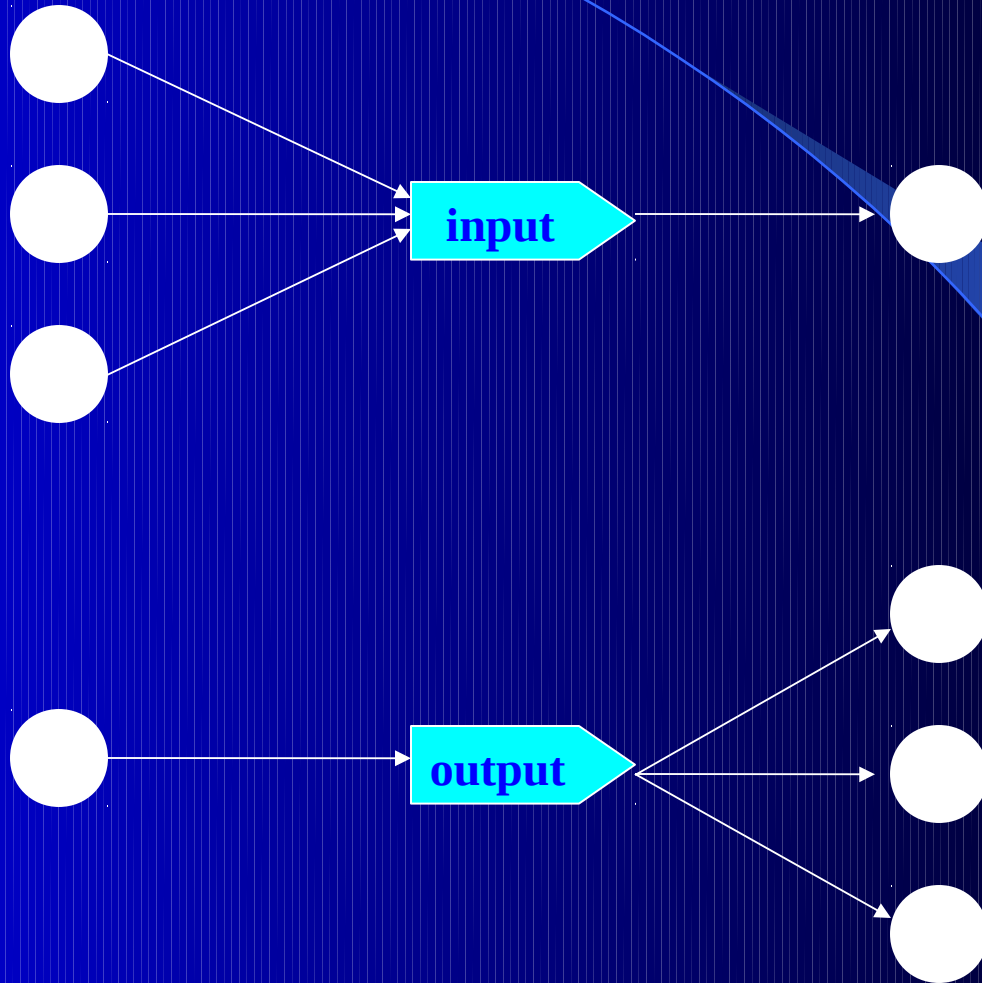
Comunicación Indirecta: Puertos

- Los procesos pueden utilizar múltiples puertos para recibir mensajes.
- Cualquier proceso que conozca el número de puerto de otro proceso, puede enviarle mensajes.
- Generalmente los servidores hacen público su número de puerto para que sea utilizado por los clientes.

Comunicación Indirecta: Puertos



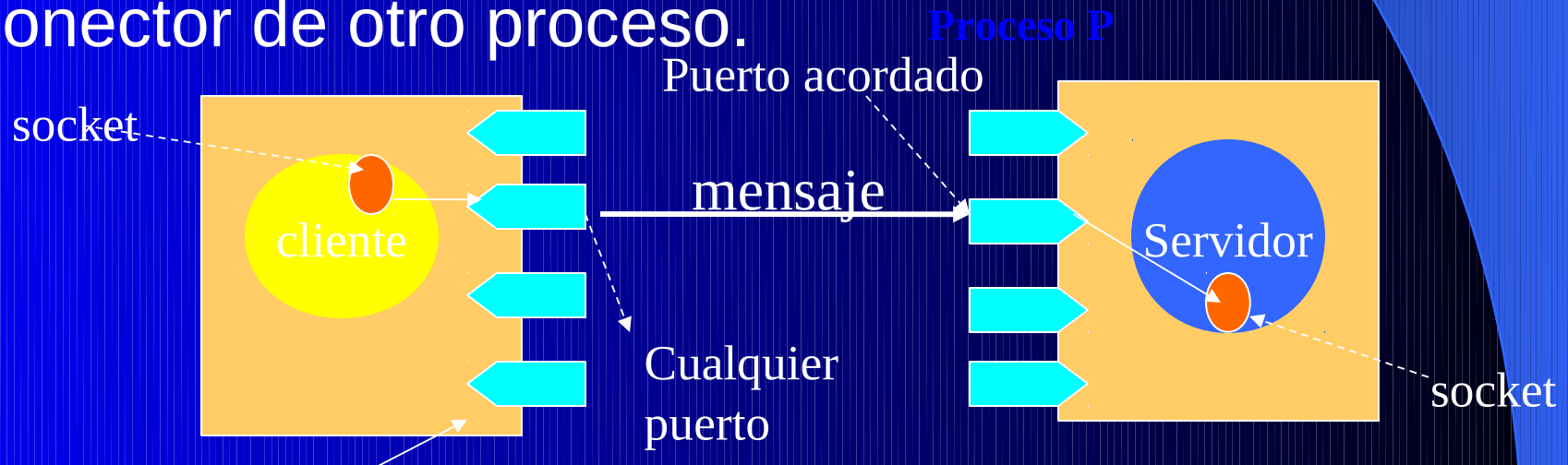
Comunicación Indirecta: Puertos



Comunicación Indirecta: Puertos

Sockets

- La abstracción de sockets se utiliza para la comunicación UDP y TCP
- La comunicación consiste entre la transmisión de un mensaje entre un conector de un proceso y un conector de otro proceso.



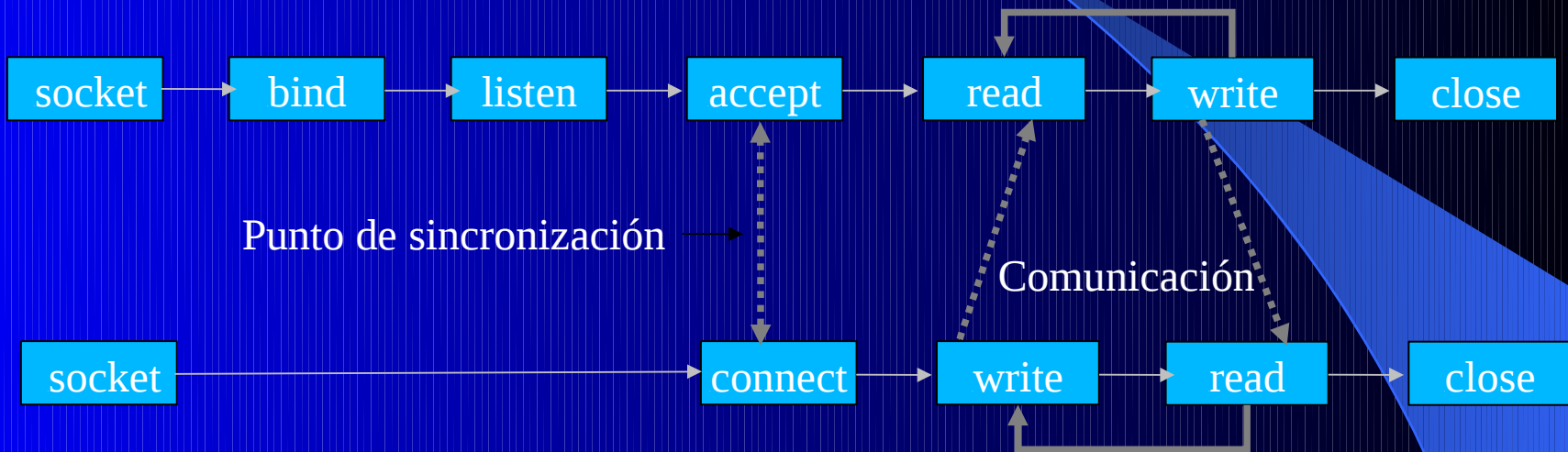
Comunicación Indirecta: Puertos

Sockets

- Los conectores deben estar asociados a un puerto local y a una dirección Internet .
- Los procesos pueden usar el mismo conector para leer y escribir mensajes.
- Cada computador permite 2^{16} puertos
- Cada proceso puede utilizar varios puertos para recibir mensajes, pero un proceso no puede compartir puertos con otros procesos del mismo computador.
- Cada conector se asocia con un protocolo concreto que puede ser UDP o TCP.

Comunicación Indirecta: Puertos

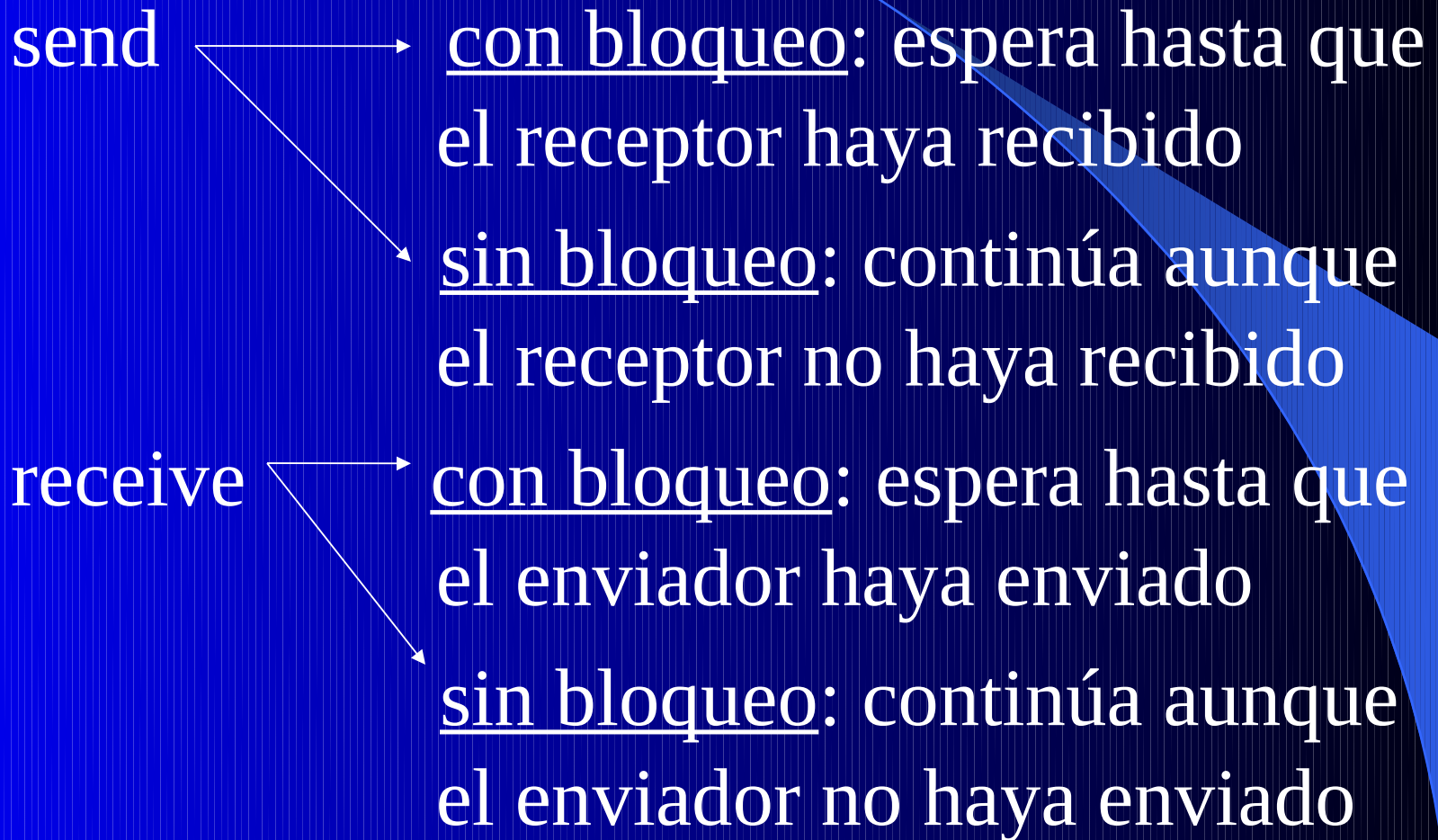
➤ Sockets



Comunicación Indirecta: Pipes

- No tienen noción de mensajes (*stream* de bytes)
- Generalmente no se comparten
- Permanecen en memoria principal si son no nominales y se guardan en memoria secundaria si son nominales
- Cuando el pipe está lleno el proceso que escribe se bloquea
- Cuando el pipe está vacío el proceso que lee se bloquea
- Son unidireccionales

Comunicación Síncrona y Asíncrona



Comunicación Síncrona y Asíncrona

Bloqueantes(síncronas):

- El emisor (*send*) se suspende hasta tanto el mensaje no es recibido.
- Una llamada a *receive* no retorna hasta que el mensaje no ha sido colocado en el buffer que especifica el receptor.
- En algunos sistemas el receptor puede especificar de qué emisor desea recibir, en cuyo caso permanecerá bloqueado hasta que lleguen los mensajes que le interesan.

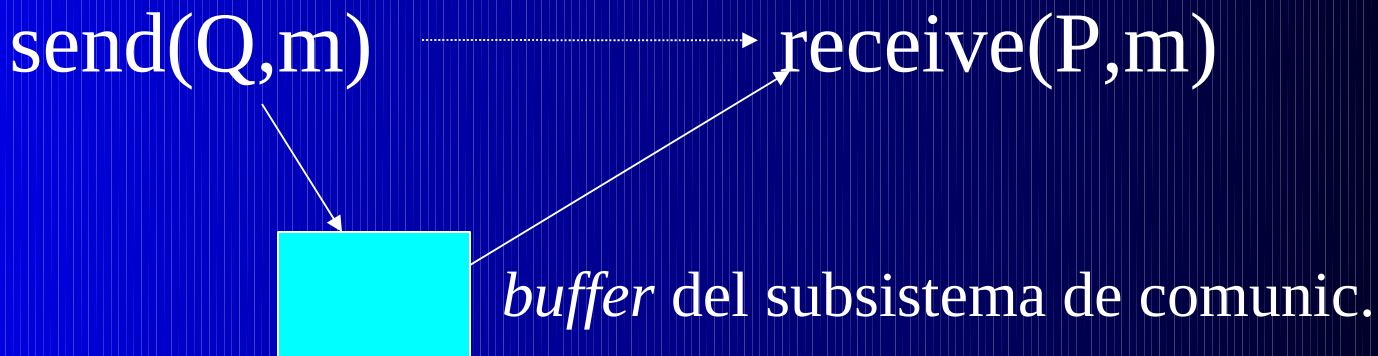
Comunicación Síncrona y Asíncrona

No bloqueantes(asíncronas):

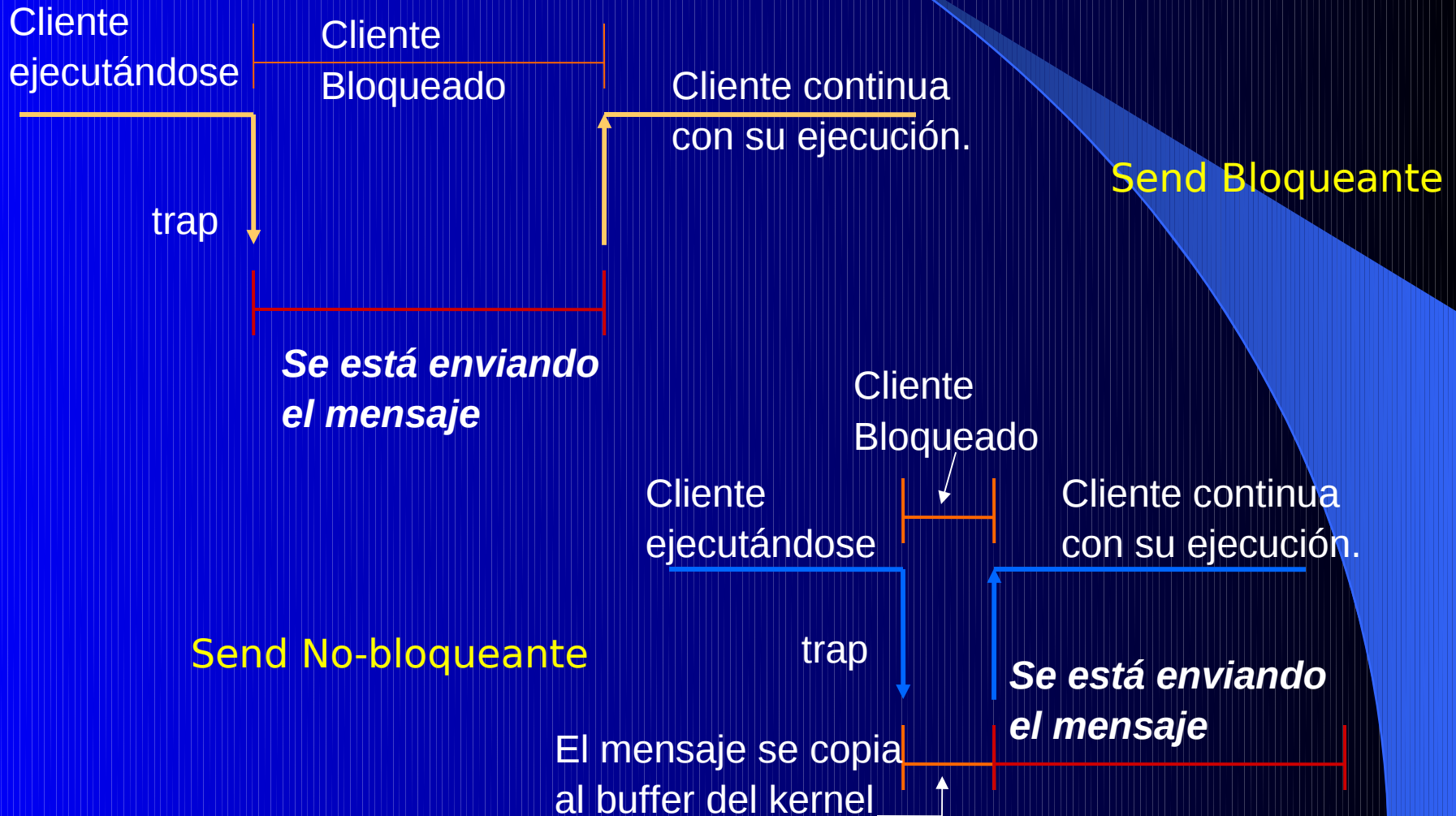
- Un *send* no bloqueante retorna el control al llamador inmediatamente antes que el mensaje sea enviado.
- Un *receive* no bloqueante le indica al kernel el buffer donde se dejará el mensaje y la llamada retorna inmediatamente.

Comunicación Síncrona y Asíncrona

- Normalmente la comunicación sin bloqueo está acompañada de *buffering*. Cuando se ejecuta el *send*, éste se bloquea hasta que el mensaje se copie en el *buffer* del subsistema de comunicaciones.



Comunicación Síncrona y Asíncrona



Comunicación Síncrona y Asíncrona

- *Receive* asíncrono: ¿cómo se entera el llamador que el mensaje ha llegado?
 - *wait* explícito: el receptor se bloquea cuando él lo desea.
 - Adicionalmente se puede ofrecer una primitiva *test* no bloqueante o un *conditional_receive* no bloqueante.
 - Usar interrupciones para avisar al receptor.

Comunicación Transitoria o Persistente

➤ *Transitoria:*

- Los mensajes son almacenados por el sistema de comunicaciones sólo mientras los procesos enviador y receptor se están ejecutando
- Si alguno de los 2 procesos falla, el mensaje es eliminado
- Típicamente son servicios de comunicación a nivel de transporte
- Enfoque *store-and-forward*

➤ *Persistente:*

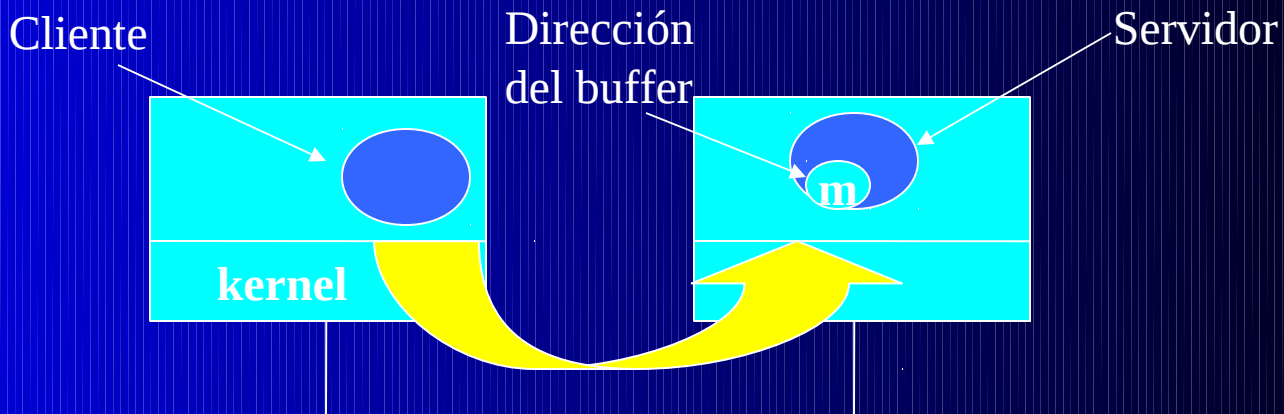
- Los mensajes son almacenados por el sistema de comunicaciones hasta que el receptor los tome.
- No requiere la ejecución simultánea de enviador y receptor

Buffering

- Capacidad de los enlaces de mantener mensajes no recibidos
- El *buffering* en los enlaces puede ser de:
 - Capacidad 0 (Sin buffer)
 - Capacidad Limitada
 - Capacidad Ilimitada (teórica)

Buffering

- Capacidad 0 (Sin buffer):
 - Define una comunicación síncrona con bloqueo.
 - Comunicación Rendezvous
 - Fácil de implementar y segura
 - Poco flexible
 - `receive(p, &m)`: el proceso está preparado para recibir un único mensaje y se dispone de un único buffer en el espacio de direcciones del proceso usuario.



Buffering

➤ Las primitivas no-bloqueantes requieren de buffering, i.e. un lugar donde el SOP pueda almacenar temporalmente los mensajes que se han enviado pero no han sido recibidos:

➤ Capacidad Limitada:

- Buzones, puertos, pipes ...
- Define una comunicación asíncrona
- La comunicación se bloquea sólo si está lleno o vacío el buffer
- Permite concurrencia
- El enviador no está seguro si el receptor leyó el mensaje

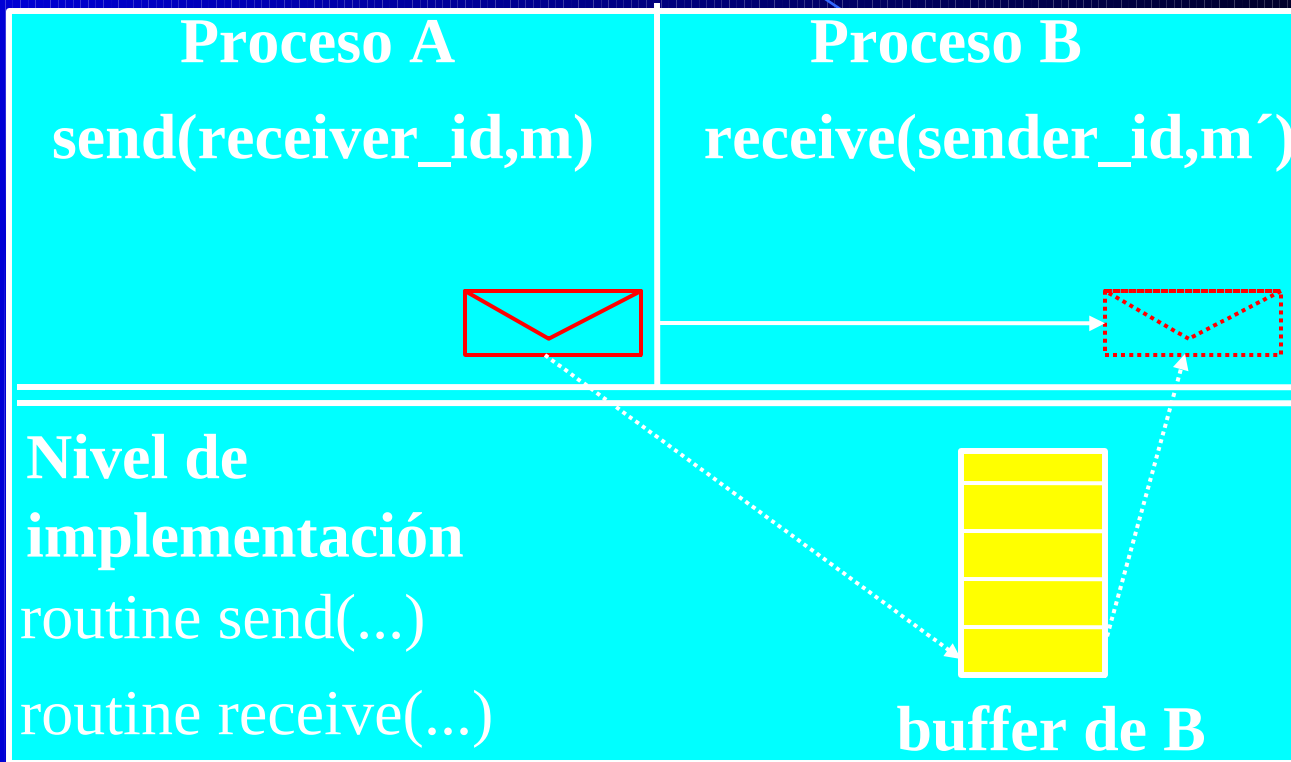
Buffering

- El *buffering* en los enlaces puede ser de:
 - Capacidad Ilimitada:
 - Define una comunicación asíncrona
 - Cuando el buffer está lleno, el send no se bloquea sino que retorna un error
 - Permite concurrencia
 - El enviador no está seguro si el receptor leyó el mensaje

Resumen de los estilos de comunicación

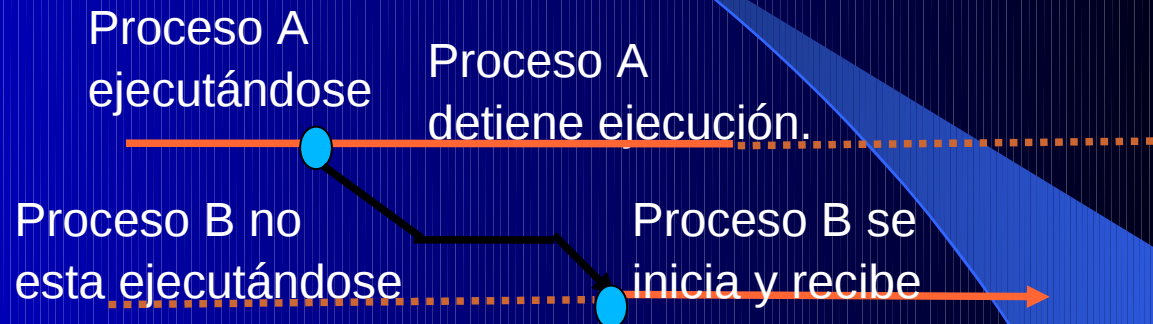
- Directa e indirecta
- Simétrica y asimétrica
- Síncrona y asíncrona
- Persistente y transitoria
- Con Buffer y sin buffer
- Ejemplos combinados:

Ejemplo 1: Comunicación Directa Asíncrona

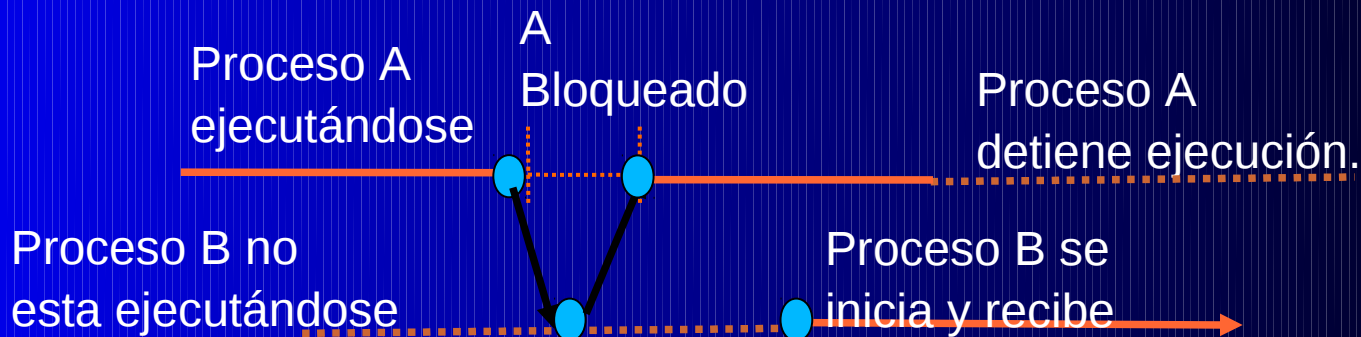


Ejemplos 2

- Comunicación asíncrona persistente: sistema de mensajes electrónicos

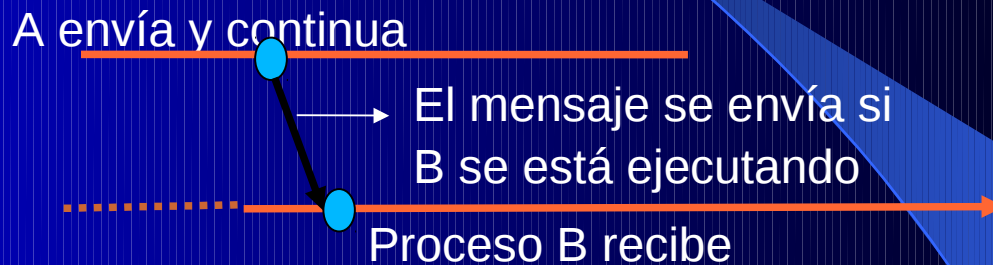


- Comunicación síncrona persistente: sistema de mensajes electrónicos



Ejemplos 3

- Comunicación asíncrona transitoria: servicios de datagramas a nivel de transporte (UDP)

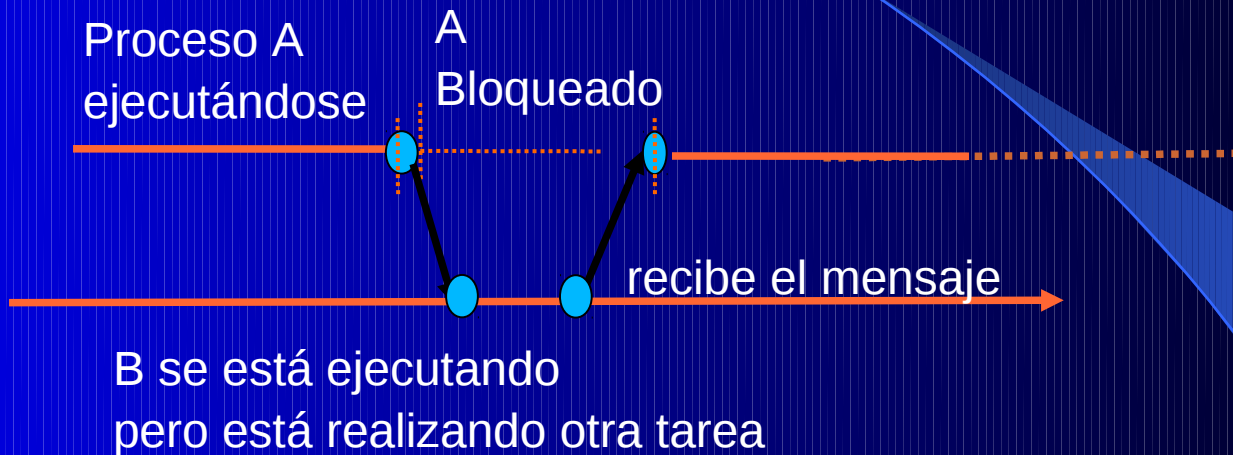


- Comunicación síncrona transitoria *receipt-based*:

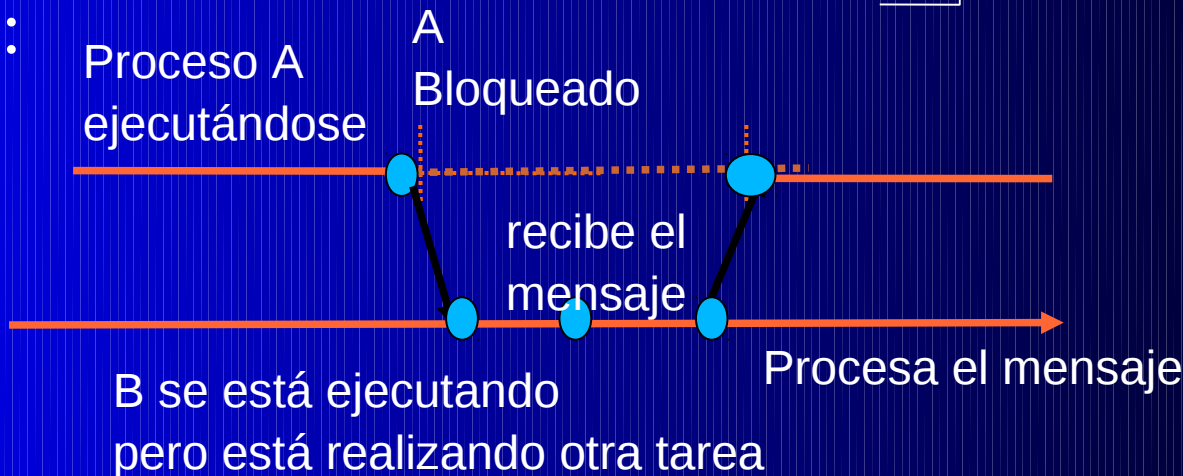


Ejemplos 4

- Comunicación síncrona transitoria *delivery-based*:



- Comunicación síncrona transitoria *response-based (request-reply)*:



MPI (Message Passing Interface)

- Es una librería de comunicación con facilidades para desarrollar programas paralelos (Fortran 77, HPF, F90, C y Java)
- Permite modelar programas paralelos con el enfoque Maestro-Esclavos
- Permite modelar topologías lógicas
- Soporta el modelo SPMD (Single Process Multiple Data)

MPI (Message Passing Interface)

- Facilidades de comunicación:
 - Punto-a-punto
 - Colectiva
 - Definición de grupo de procesos
 - Definición de comunicadores
 - Definición de topologías de procesos
 - Interfaz de *profiling*
 - Operaciones de memoria compartida
 - Soporte de *threads*

Ejemplo de programa paralelo con MPI

```
#include "mpi.h"
#define N 10
int main(int argc, char **argv) {
    int myId, numprocess,i;
    //Inicializaciones
    MPI_Init(&argc,&argv);
    MPI_COMM_SIZE(MPI_COMM_WORLD,&numprocess);
    MPI_COMM_RANK(MPI_COMM_WORLD,&myId);
    if (myId==0) {
        printf("Soy el proceso %d",myId);
        for (i=0;i<N;i++)
            MPI_Send(&i,1,MPI_INT,1,0, MPI_COMM_WORLD);
    } else {
        printf("Soy el proceso %d",myId);
        for (i=0;i<N;i++)
            MPI_Receive(&i,1,MPI_INT,0,0, MPI_COMM_WORLD, &status);
    } MPI_Finalize()
}
```

Ejemplo de programa paralelo con MPI

```
#include "mpi.h"
#include <math.h>
int main(int argc, char **argv) {
    int myId, numprocess,n,i,rc;
    double mypi, pi, h, sum, x, a;
    //Inicializaciones
    MPI_Init(&argc,&argv);
    MPI_COMM_SIZE(MPI_COMM_WORLD,&numprocess);
    MPI_COMM_RANK(MPI_COMM_WORLD,&myId);
    while (1) {
        if (myId==0) {
            printf("Introduzca el nro. de intervalos" );
            scanf("%d",&n);
        }
        MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
        if (n==0)
            break;
        else {
            for (i=myId+1;i<=n;i+=numprocess)
```

Ejemplo de programa paralelo con MPI

```
{
    realizar calculos de h y sum
}
mypi=h*sum;
MPI_Reduce(&mypi,&pi,1,MPI_DOUBLE,MPI_SUM,0,
           MPI_COMM_WORLD);
if (myid==0)
    printf("pi es aprox. %.16f ",pi);
} //else
} //while
MPI_Finalize();
}
```