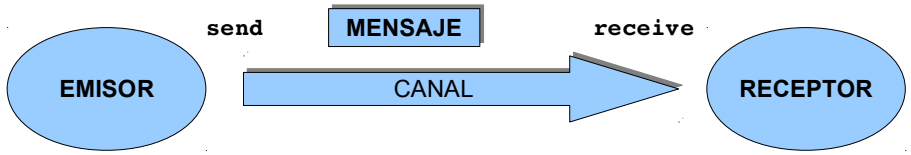
TEMA 5

Soluciones Basadas en el Paso de Mensajes

- Cuando no existe la posibilidad de usar mecanismos de memoria compartida se necesitan otras herramientas de control de la concurrencia.
- En esta situación, típica de los **sistemas distribuidos**, el **paso de mensajes** es la herramienta que se utiliza.
- La comunicación se realiza mediante **redes** de comunicaciones que conlleva una serie de problemas asociados como la perdida de mensajes, la llegada desordenada de los mismos o la heterogeneidad del hardware y software de cada nodo.

- Las primitivas básicas para el intercambio de información son **`send`** y **`receive`**.
- Podemos entenderlas como una extensión de los semáforos, donde además hay **intercambio de información**.



- Aunque este mecanismo es empleado en sistemas distribuidos también es posible hacerlo en sistemas con memoria compartida.

2. Aspectos de Diseño

- Para definir el paso de mensajes necesitamos definir los siguientes **aspectos:**
  1. Identificación de procesos (nombrado, direccionamiento)
  2. Sincronización
  3. Características del canal
- Según la definición de estos aspectos, será posible especificar una variedad de esquemas de comunicación mediante el paso de mensajes.

2. Aspectos de Diseño

1. Identificación de procesos

- Es la forma en la que el emisor indica a quién va dirigido el mensaje y viceversa (de quien lo espera el receptor). Existen dos variantes:
- **Denominación Indirecta**: Los mensajes son enviados de forma anónima, generalmente a un depósito intermedio; esto es los procesos no se conocen. El deposito suele ser conocido como **buzón**.

```
P1: send(m1,BuzonA)

P2: receive(m2,BuzonA)
```

  - Con este mecanismo se permite la comunicación 1:1, 1:N, N:1 y N:N.

2. Aspectos de Diseño

1. Identificación de procesos

- **Denominación Directa**: Los mensajes se envían y/o reciben a procesos concretos. En esta caso, existen dos tipo diferentes:
  - Denominación Directa Simétrica: Ambos procesos se conocen y se nombran

```
P1: send(m1, P2)

P2: receive(m2, P1)
```

  - Denominación Directa Asimétrica: El emisor identifica al receptor, pero no al revés.

```
P1: send(m1, P2)

P2: receive(Id, m2)
```


2. Aspectos de Diseño

2. Sincronización

- Como norma general el proceso receptor cuando requiere un mensaje esperará hasta que éste llegue. Sin embargo, el emisor puede comportarse de forma diferente. Así, existen dos alternativas:
- **Asíncrona**: el emisor envía el mensaje y continúa, sin preocuparse si el mensaje será recibido.
- **Síncrona**: el emisor espera a que el receptor esté dispuesto para recibir el mensaje. Este tipo de paso de mensaje se conoce como **rendez-vous** o cita.
  Una variante del rendez-vous simple consiste en emitir un mensaje de respuesta, esta alternativa se conoce como **rendez-vous extendido** o invocación remota.

3. Canal de comunicación y mensajes

- El canal de comunicación tienes sus propias características:
- **Flujo de Datos**: Según la circulación de la información puede ser:
  - Unidireccional: desde el emisor al receptor
  - Bidireccional: desde el emisor al receptor pero con respuesta desde el receptor al emisor.
- **Capacidad del canal.** Es la cantidad de mensajes pendientes de entregar que puede haber en el canal:
  - Capacidad cero. La comunicación debe ser síncrona
  - Capacidad finita, según el tamaño del *buffer* de almacenamiento

3. Canal de comunicación y mensajes

- **Tamaño de los mensajes**. Podrán ser de tamaño fijo (todos del mismo tamaño, normalmente pequeño) o variable.
- **Canales con tipo o sin tipo**: Algunos canales sólo permiten enviar mensajes del tipo para el que se han definido.
- **Envío de copia o referencia:** Tendrían las mismas características que el paso de parámetros a una función.
- **Errores en las comunicaciones**. Hay que detectarlas y operar en consecuencia. Suele ser tarea de los protocolos de red. Nosotros consideraremos que los mensajes llegan siempre sin error.

Espera selectiva

- Cuando se emplea un mecanismo de comunicación **cliente/servidor**, donde un proceso recibe peticiones de varios procesos, el proceso servidor debería ser capaz de atender peticiones a través de varios canales a la vez.
- No obstante, dado que la operación **`receive`** espera hasta recibir el mensaje, puede ocurrir que éste llegue a un canal sobre el que no se está haciendo la espera.
- Para permitir hacer la espera en varios canales se dispone de la sentencia **`select.`**

Espera selectiva

- La sintaxis de la sentencia **`select`** es:
- Al llegar a la sentencia **`select`**, el proceso selecciona de forma aleatoria alguna de las ramas sobre las que se haya realizado una operación de envío.
  De no haber ninguna, el proceso se bloque en espera de que llegue alguna de ellas.

```
select
  receive(....);
  sentencias;
or
  receive(....);
  sentencias;
or
 . . . .
or
  receive(....);
  sentencias;
end select;
```

2. Aspectos de Diseño

Espera selectiva

- Una variante más potente la constituye la sentencia **`select`** con guardas.
- No todas las clausulas deben llevar guardas.
- Al llegar a **`select`** solo se consideran abiertas aquellas clausulas con la condición cierta.
- La condición sólo se evalúa al comenzar la sentencia **`select`**, no tras realizar el posible bloqueo.

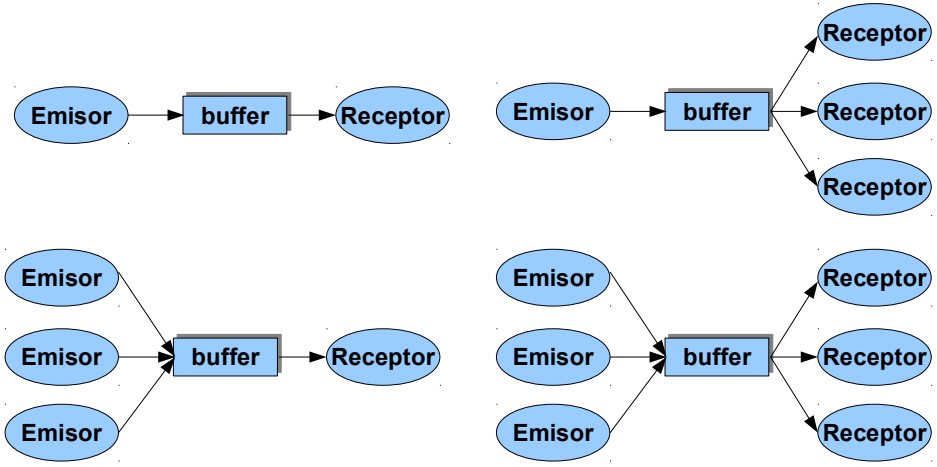
```
select
  when condicion1 =>
  receive(....);
  sentencias;
or
  when condicion1 =>
  receive(....);
  sentencias;
or
 . . . .
or
  when condicion1 =>
  receive(....);
  sentencias;
end select;
```


3. Paso de mensaje asíncrono

- En este caso, el emisor envía el mensaje y continúa, sin esperar a que sea leído
- Según el comportamiento de **`receive`** tenemos dos alternativas:
  - Cuando no hay mensajes se espera la llegada de uno.
  - Cuando no hay mensajes se continúa (devolviendo algún valor que indique la ausencia del mensaje)
- En cualquier caso, es necesaria la existencia de un *buffer*, donde se almacenen los mensajes. Normalmente, a dicho *buffer* se le conoce como **buzón**.

3. Paso de mensaje asíncrono

- Con este esquema de comunicación es posible establecer relaciones entre emisores y receptores del tipo uno a uno, uno a muchos, muchos a uno o muchos a muchos.



3. Paso de mensajes asíncrono

- En nuestra propuesta, vamos a establecer comunicaciones **muchos a muchos**.
- Para la comunicación necesitaremos poder definir buzones. La sintaxis será la siguiente:

```
var
  nombre_buzon: mailbox of tipo;
  nombre_buzon: mailbox [1..N] of tipo;
```

- Donde la primera declaración asume un buzón sin límite, y la segunda limita el tamaño del buzón.
- Para el caso del buzón limitado, la operación de envío provocará el bloqueo del proceso emisor.
- La gestión de las colas asociadas al buzón serán FIFO.

3. Paso de mensajes asíncrono

- Las operaciones permitidas sobre un buzón son:
  - **`send(buzon, mensaje)`**. El emisor deja el mensaje en el buzón y continua. El tipo del mensaje debe ser el mismo que el del buzón. Si el buzón es limitado y está lleno, el proceso espera a poder poner el mensaje cuando haya sitio.
  - **`receive(buzon, mensaje)`**. El receptor saca el primer mensaje de la cola del buzón y continua. Si el buzón está vacío espera a que se deje un mensaje en él para recogerlo y continuar.
  - **`empty(buzón)`**. Devuelve verdadero o falso según esté el buzón vacío o no. La condición sólo es valida durante la comprobación, dado que luego el estado del buzón puede cambiar por la ejecución concurrente de otros procesos.
- Todas estas operaciones se consideran atómicas.

3. Paso de mensajes asíncrono

- Para ilustrar el uso de mensajes vamos a resolver el problema de la exclusión mutua para dos procesos. Dicha solución es fácilmente extensible a N procesos.

```
program ejemplo
var
   buzon: mailbox of integer;
   testigo: integer;

//DEFINICION DE LOS PROCESOS P1 Y P2

begin
  cobegin
    send(buzon, testigo);
    P1;
    P2;
  coend
end
```


3. Paso de mensajes asíncrono

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

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

NOTA: El contenido del mensaje no es significativo es esta solución

3. Paso de mensajes asíncrono

- **El problema del productor-consumidor**
- **El problema de los lectores y los escritores (Prioridad en lectura)**
- **El problema de los lectores y los escritores (Prioridad en escritura)**
- **El problema de los filósofos**

3. Paso de mensajes asíncrono

- Conclusiones:
  - Este tipo de modelo de comunicación permite establecer relaciones **muchos a muchos**.
  - Se supone además, que la **capacidad del buzón** es ilimitada, salvo que se indique explícitamente.
  - El principal inconveniente de este mecanismo es que resulta muy **ineficiente** cuando el emisor necesita asegurarse de que el receptor ha recibido el mensaje. En estos casos hay que realizar la comprobación explícitamente.

4. Paso de mensajes síncrono

- En el caso anterior, la primitiva **`send`** era no bloqueante.
- En el paso de mensajes síncrono tanto la primitiva **`send`** como **`receive`** son **bloqueantes**.
- Con este mecanismo, el emisor se asegura de que el receptor ha recibido el mensaje antes de seguir adelante.
- Dada su naturaleza, al no almacenar mensajes, no es necesario disponer de un almacén (*buffer*) de mensajes, lo que hace este mecanismo más sencillo de implementar.
- El mecanismo de mensajes síncrono que vamos a estudiar son los **canales**.

4. Paso de mensajes síncrono

- Un **canal** permite enlazar dos procesos, de forma que las operaciones de envío y recepción sean bloqueantes.
- Se pueden considerar como una instrucción de asignación distribuida.
- La **relación** que se establece es de **1 a 1** entre emisor y receptor, estableciendo un flujo de datos unidireccional.
- Además los canales suelen tener un **tipo**, que es el de los datos que se envían
- En Pascal-FC se usa la sintaxis:

```
var
  ch: channel of tipo;

ch ! s; //Envía el valor de s al canal ch
ch ? r; //Recibe del canal ch un valor que asigna a r
```


4. Paso de mensajes síncrono

- Si se desea usar un canal sin tipo, donde el valor de la información no sea relevante, y el canal, por tanto, tan sólo se use con fines de sincronización, se declarará de tipo **`synchronous`**.

```
var
  ch: channel of synchronous;

ch ! any; //Envía mensaje de sincronización al canal ch
ch ? any; //Recibe mensaje de sincronización al canal ch
```

- La variable **`any`** es automáticamente definida por el compilador al usar un canal **`synchonous`**, por tanto no hay que definirla.

4. Paso de mensajes síncrono

- En Pascal-FC existen diferentes variantes para realizar la espera selectiva sobre canales con la sentencia **select**.
- Espera selectiva **básica**:

```
select
  ch1 ? mensaje1;
  sentencias;
or
  ch2 ? mensaje2;
  sentencias;
or
 . . . .
or
  chN ? mensajeN;
  sentencias;
end;
```


Programación Concurrente y Distribuida. Curso 2014-15

4. Paso de mensajes síncrono

- En Pascal-FC existen diferentes variantes para realizar la espera selectiva sobre canales con la sentencia **select**.
- Espera selectiva **básica**:

```
select
  ch[1] ? mensaje[1];
  sentencias;
or
  ch[2] ? mensaje[2];
  sentencias;
or
 . . . .
or
  ch[N] ? mensaje[N];
  sentencias;
end;
```

```
select
  for cont=1 to N replicate
    ch[cont] ? mensaje[cont];
    sentencias;
end;
```


Programación Concurrente y Distribuida. Curso 2014-15

4. Paso de mensajes síncrono

- Espera selectiva con **guardas.**
- La condición sólo se evalúa al comienzo de la sentencia **select** y no vuelve a hacerse hasta que se vuelva a ejecutar dicha sentencia.

```
select
  when condicion1 =>
  ch1 ? mensaje1;
  sentencias;
or
  ch2 ! mensaje2;
  sentencias;
 . . . .
or
  when condicionN =>
  chN ? mensajeN;
  sentencias;
end;
```


Programación Concurrente y Distribuida. Curso 2014-15

4. Paso de mensajes síncrono

- Espera selectiva con **terminate.**
- La elección de **terminate** provoca la salida de la sentencia **select.**
- Esta alternativa se elige cuando ninguna de las otras tiene llamadas pendientes, y los procesos que pueden provocar dichas llamadas han finalizado, o están bloqueados en una sentencia **select** con una alternativa **terminate.**

```
select
  ch1 ? mensaje1;
  sentencias;
or

or
  terminate;
end;
```

4. Paso de mensajes síncrono

- Espera selectiva con **`else.`**
- La elección de **`else`** se realiza cuando ninguna de las otras puede realizarse.
- Solo puede haber una alternativa con **`else`** y no puede tener guardas.

```
select
  ch1 ? mensaje1;
  sentencias;
or
  ch2 ? mensaje2;
  sentencias;

else
  sentencias;
end;
```


4. Paso de mensajes síncrono

- Espera selectiva con **`timeout.`**
- Si pasados *n* segundos desde que se ejecutó **`select`** no se ha ejecutado ninguna de las alternativas, entonces se ejecuta la de **`timeout`** (puede tener guardas).

```
select
  ch1 ? mensaje1;
  sentencias;
or
  ch2 ? mensaje2;
  sentencias;

or
  timeout n;
  sentencias;
end;
```

NOTA:
Las sentencias **`terminate`**, **`else`** y **`timeout`** son excluyentes entre sí, por tanto, no pueden mezclarse en un mismo **`select`**.

4. Paso de mensajes síncrono

- Espera selectiva con **`prioridad.`**
- Hasta ahora, si se podían seleccionar varias alternativas, se elegía una de ellas de forma aleatoria.
- Con la sentencia con prioridad, si varias alternativas son posibles se elegirá la primera por orden de aparición.

```
pri select
  ch1 ? mensaje1;
  sentencias;
or
  ch2 ? mensaje2;
  sentencias;


end;
```


4. Paso de mensajes síncrono

- **Simulación de Semáforos con Canales**
- **El problema del productor-consumidor**
- **El problema de los lectores y los escritores (Prioridad en lectura)**
- **El problema de los lectores y los escritores (Prioridad en escritura)**
- **El problema de los filósofos**