Tema 3

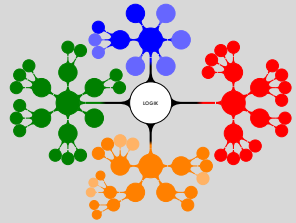
Sistemas basados en paso de mensajes

Soluciones software para desarrollar programas distribuidos

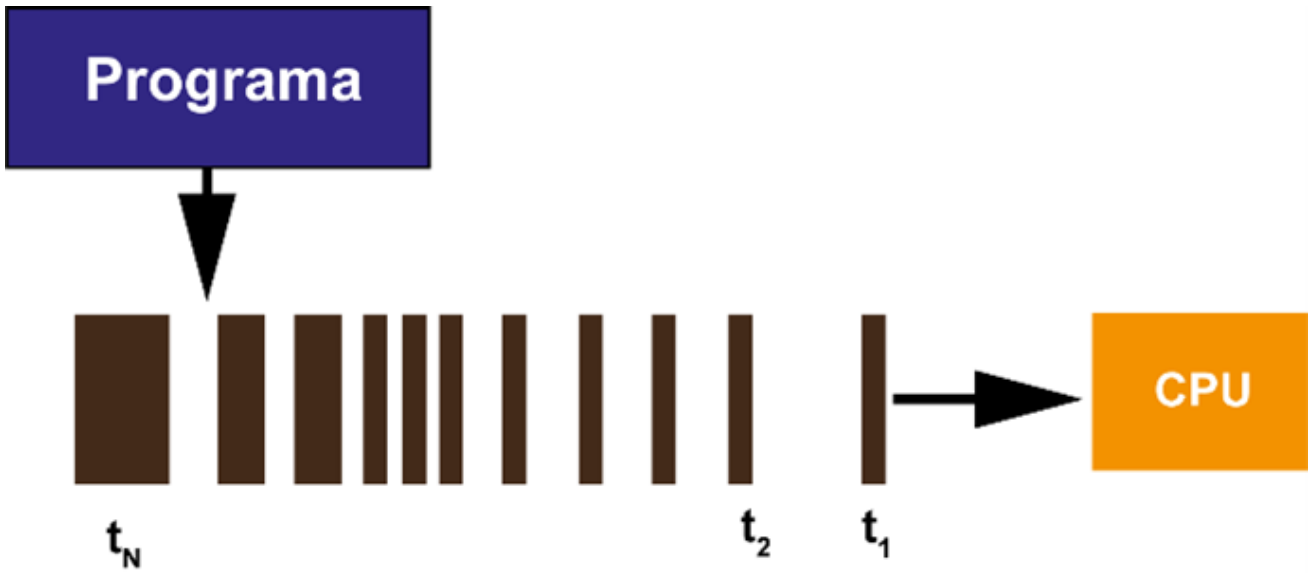
Asignatura *Sistemas Concurrentes y Distribuidos*

Fecha 25 noviembre 2022

Manuel I. Capel
manuelcapel@ugr.es
Departamento de Lenguajes y Sistemas Informáticos
Universidad de Granada



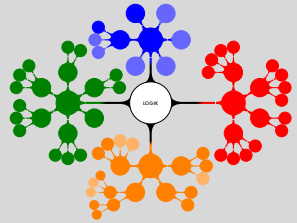
Motivación



Computador monoprocesador clásico: las instrucciones son ejecutadas 1 a 1 en la CPU

- Problemas que reducen la eficiencia del sistema
- Los computadores de programa almacenado: no pueden producirse simultáneamente una búsqueda de instrucciones y una operación de datos

Manuel I. Capel
manuelcapel@ugr.es



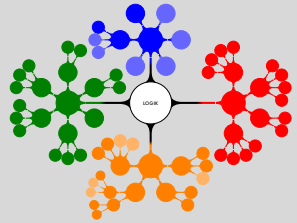
Problemas de computadores con programa único almacenado

Algunos problemas de estas arquitecturas:

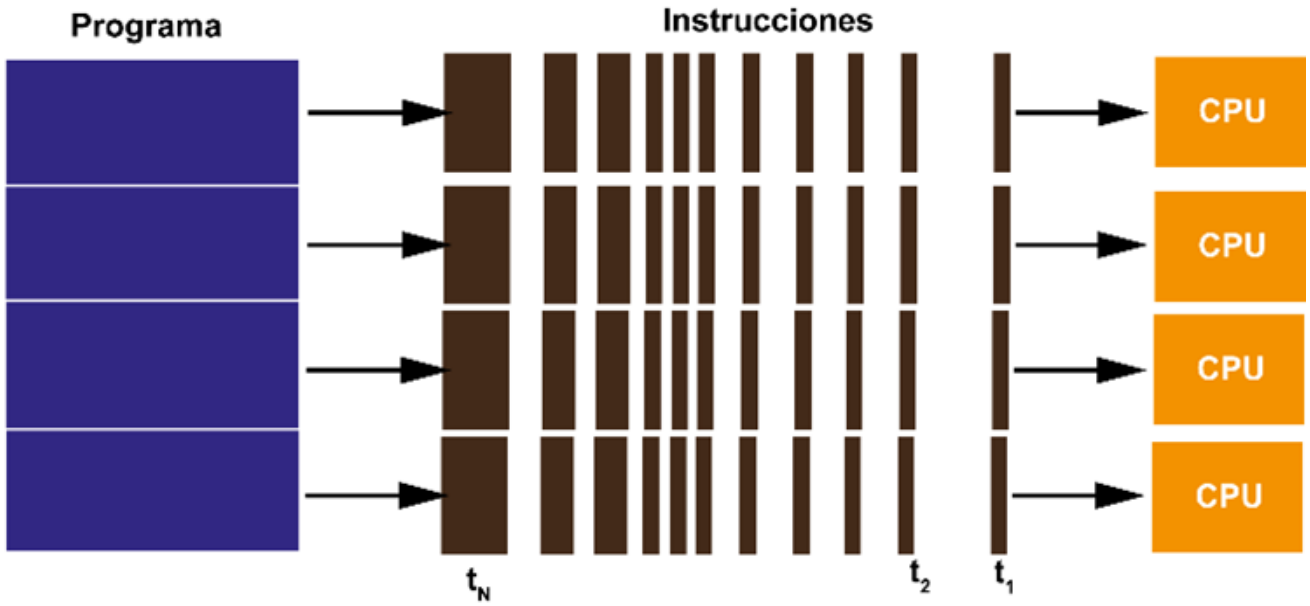
- El cuello de botella "Von Neumann"
- Limitación de velocidad del bus
- Los programas necesitan más tiempo de ejecución

- Los inconvenientes anteriores afectan al rendimiento de la computación
- Además, son más caras de producir

Manuel I. Capel
manuelcapel@ugr.es



Arquitectura que usa múltiples elementos de cómputo simultáneamente

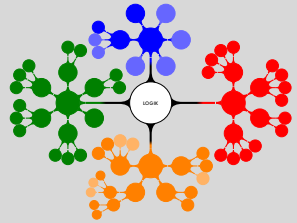


Las instrucciones son ejecutadas en varios procesadores y los datos se acceden independientemente

Solucionan los problemas de las arquitecturas clásicas de programa único almacenado:

- El sistema consume menos energía,
- Es más barato de producir y
- Proporciona mejor rendimiento porque resuelve el problema del “cuello de botella"

Manuel I. Capel
manuelcapel@ugr.es



Multiprocesamiento

Idea fundamental

Utilización de varios procesadores o *núcleos* para ejecutar los programas de una misma aplicación

Desde el punto de vista del sistema

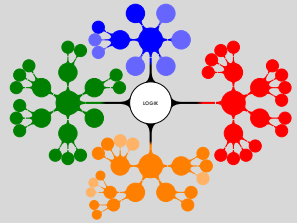
Capacidad para gestionar más de 1 procesador y re-asignar tareas entre tales procesadores durante la ejecución de los programas

- Los procesadores pueden ejecutar 1 sola secuencia o varias secuencias de instrucciones, que se ejecutarán en *contextos* múltiples y simultáneos

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

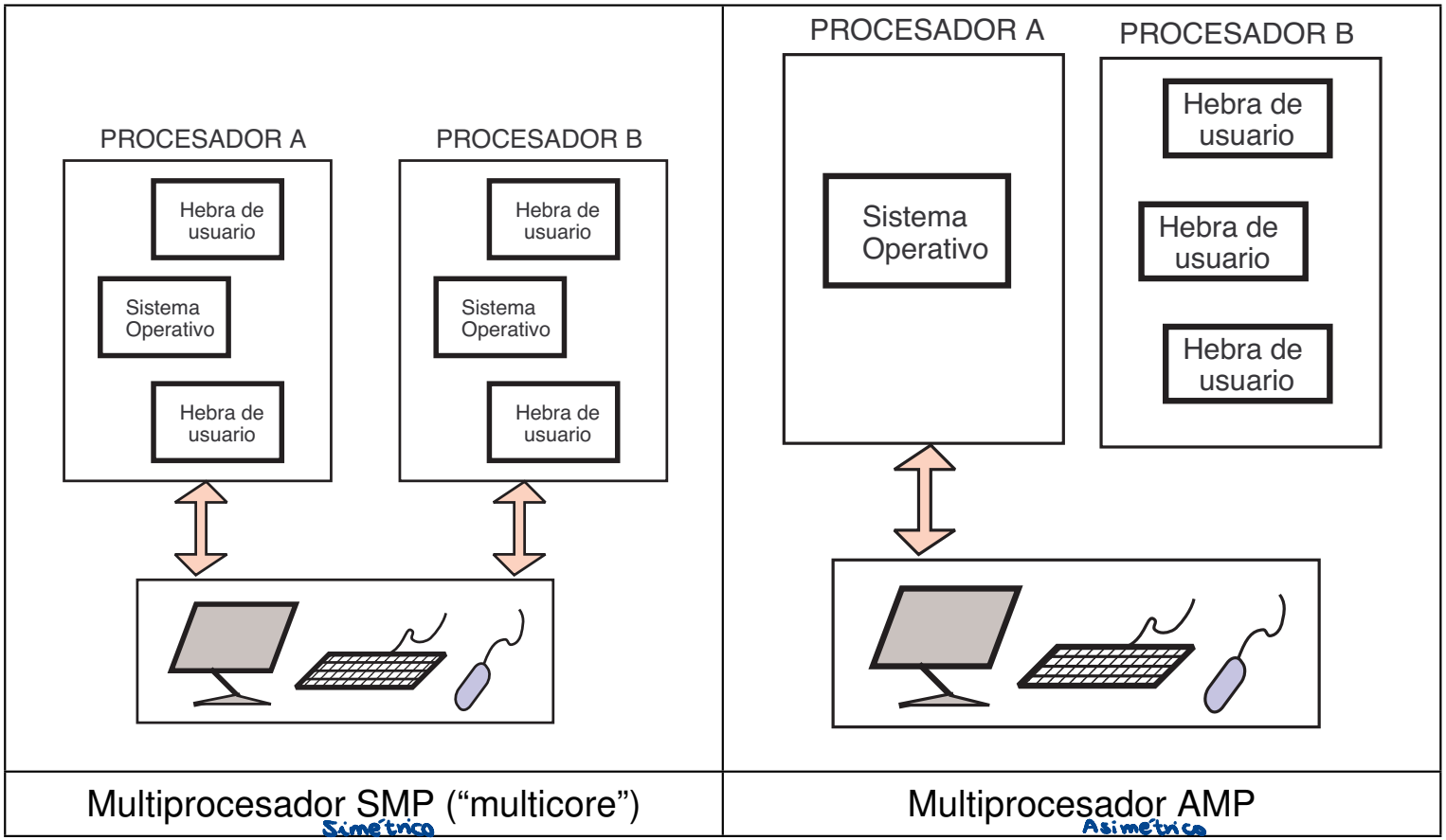
Clasificación de Flynn

Manuel I. Capel
manuelcapel@ugr.es



Mecanismos básicos en sistemas *multiprocesadores*

- El multiprocesador que resulta más útil: procesadores individuales que ejecutan sus instrucciones independientemente (modelo MIMD)
  1. Utilizar variables en memoria común a los procesadores
  2. El inconveniente principal es falta de escalabilidad.
  3. Hardware de interconexión procesadores-memorias caro



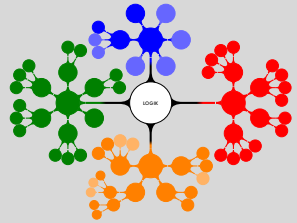
Multiprocesador SMP ("multicore") — Simétrico

-) Varios procesadores idénticos.
-) Se abstrae el control individual de cada uno de ellos mediante una instancia del SO.
-) Se interconectan mediante buses, conmutadores de barras cruzadas...

Multiprocesador AMP — Asimétrico

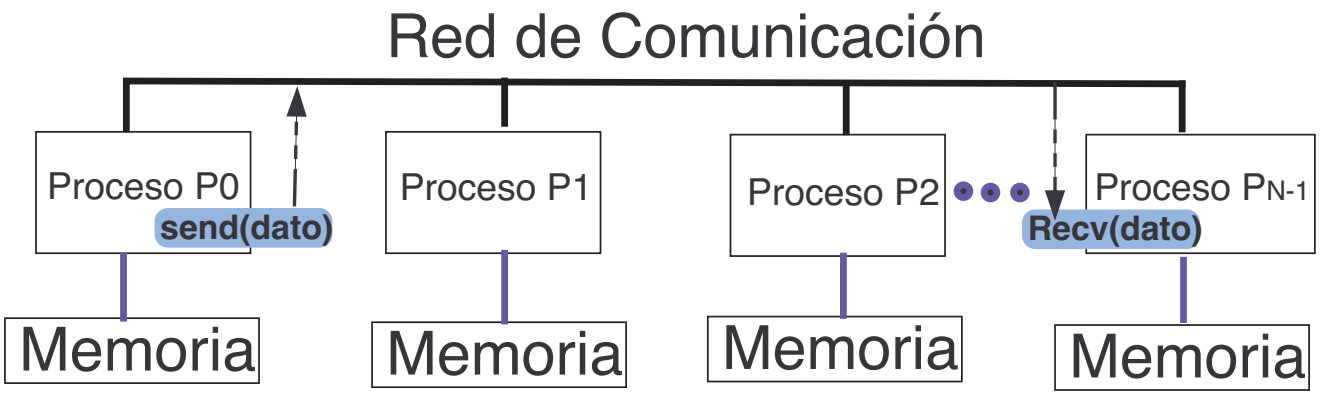
-) Se gestionan procesadores de distintos tipos.
-) Todas las CPU deben tener mismo conjunto de instrucciones de bajo nivel, pues un trabajo puede ser desplazado de una CPU a otra.
-) La CPU solo es un procesador aritmético y lógico.

Manuel I. Capel
manuelcapel@ugr.es



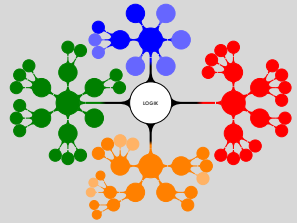
Mecanismos básicos en sistemas *multicomputadores*

- No existe memoria de acceso común a todos los computadores
  1. Son más difíciles de programar.
  2. No suelen presentar el problema de la *escalabilidad*
  3. Se necesita una notación de programación más flexible que la que usamos con multiprocesadores, por ejemplo:
     - diferentes modos de comunicación entre los procesos,
     - *no-determinismo* en la recepción de múltiples comunicaciones síncronas en servidores.



  4. Las primitivas concurrentes clásicas para multiprocesadores que suponen memoria compartida no se pueden utilizar, obviamente.

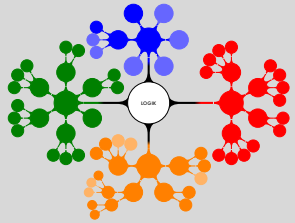
Manuel I. Capel
manuelcapel@ugr.es



Inconvenientes de la Programación Distribuida y Paralela

- Supone aprender a programar según un paradigma de programación que asume el avance de múltiples líneas de ejecución de instrucciones en los programas y sin variables globales compartidas por los procesos
- La depuración de estos programas se convierte en una tarea muy difícil
- Acceso a la red de interconexión y a la memoria, tanto a nivel de caché como a nivel de MPs

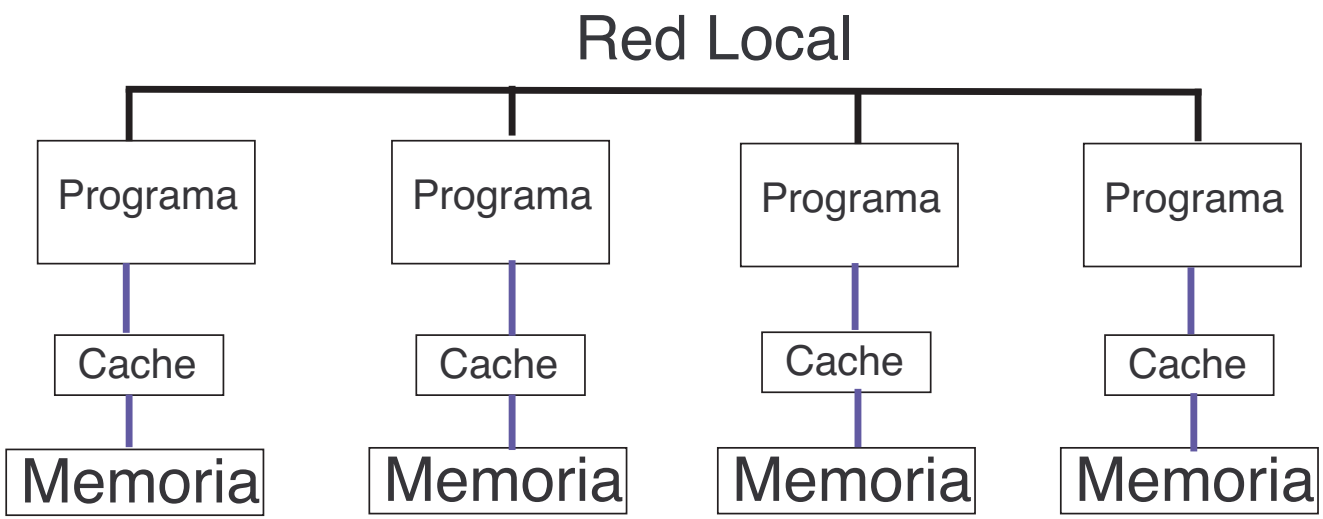
Manuel I. Capel
manuelcapel@ugr.es



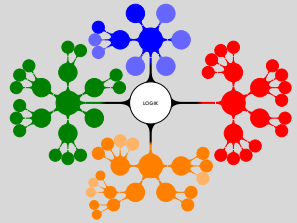
Estilo de multiprogramación SPMD

Idea fundamental

Se utiliza en programación distribuida y paralela: el código que ejecutan los procesos es idéntico pero tal código actúa sobre diferentes datos



Manuel I. Capel
manuelcapel@ugr.es

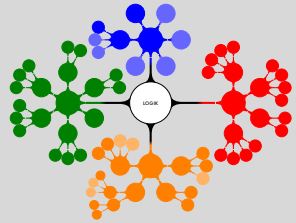


Estilo de multiprogramación SPMD–II

Características

- Variante del modelo general MIMD de Flynn, que se aproxima al SIMD, que necesita una arquitectura especial de computador
- No se necesitan computadores con arquitecturas especializadas para programar SPMD
- Los procesadores ejecutan el mismo programa pero de forma independiente
- Independiente de la compilación, a cada proceso paralelo del programa se le asignará un valor de identificador distinto
- De esta forma, cada proceso puede ejecutar una parte distinta del programa común

Manuel I. Capel
manuelcapel@ugr.es

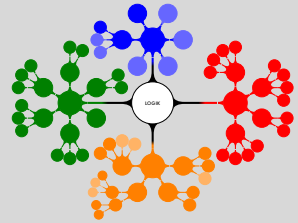


Ejemplo de código programado según un estilo SPMD

```
                Cliente Trabajador 1 Trabajador 2
                  a b e | c d f      | c d f
-----------------------------------------------------
a = 3;            3 - - | - - -      | - - -
b = 4;            3 4 - | - - -      | - - -
spmd                    |            |
c = rank();       3 4 - | 1 - -      | 2 - -
d = c + a;        3 4 - | 1 4 -      | 2 5 -
end                     |            |
e = a + d{1};     3 4 7 | 1 4 -      | 2 5 -
c{2} = 5;         3 4 7 | 1 4 -      | 5 5 -
spmd                    |            |
f = c * b;        3 4 7 | 1 4 4      | 5 5 20
end
```

- El valor de las variables definidas en el `cliente` puede ser leído por los `trabajadores`, pero no puede ser cambiado.
- Las variables definidas por los `trabajadores` pueden ser leídas o cambiadas por el `cliente`.

Manuel I. Capel
manuelcapel@ugr.es



Semántica de las operaciones de paso de mensajes

Idea fundamental

Significado (semántica) de las operaciones de comunicación ({`send(), receive()`}) entre procesos depende de 2 factores:

1. cumplimiento (o no) de la propiedad de *seguridad*
2. modo de comunicación

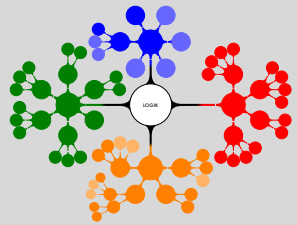
1) Propiedad de seguridad

- No se cumple: se permite alterar un dato después de que la operación de envío (`send(...)`) se ejecute y vuelva, pero antes de que termine la transmisión del valor enviado
- Se cumple: el valor del dato enviado coincide con el del dato recibido posteriormente

2) Modo de comunicación de las operaciones de paso de mensajes

- Operaciones bloqueantes
- Operaciones no-bloqueantes (con *asincronicidad*)

Manuel I. Capel
manuelcapel@ugr.es



Operaciones bloqueantes de paso de mensajes

Idea fundamental

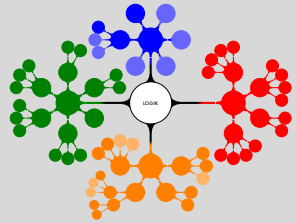
La operación de envío (`send(...)`) sólo vuelve cuando termina la transmisión del mensaje

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

Características de las operaciones bloqueantes

El que envía solo se bloquea cuando el buffer está lleno. El receptor siempre está bloqueado

Manuel I. Capel
manuelcapel@ugr.es

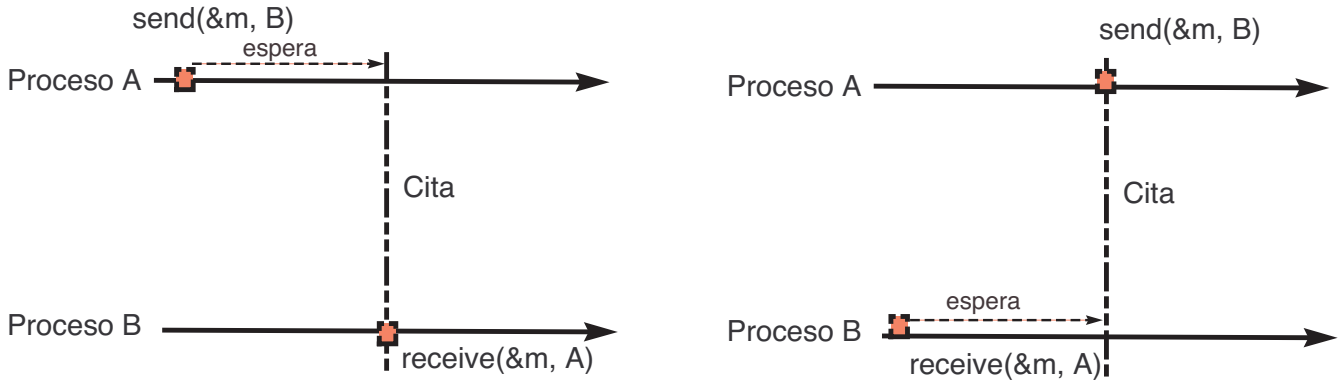


Mecanismo de *citas*

Idea fundamental

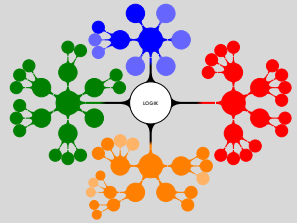
Se trata de una operación de comunicación bloqueante y sin búfer.
Cita: tiene lugar antes de que comience la transmisión de los datos

- El estado del proceso emisor se mantiene hasta que vuelve la operación de recepción en el otro proceso
- Los procesos receptor o emisor pueden sufrir espera ociosa hasta que termina la mencionada cita



Cita entre procesos con comunicación síncrona

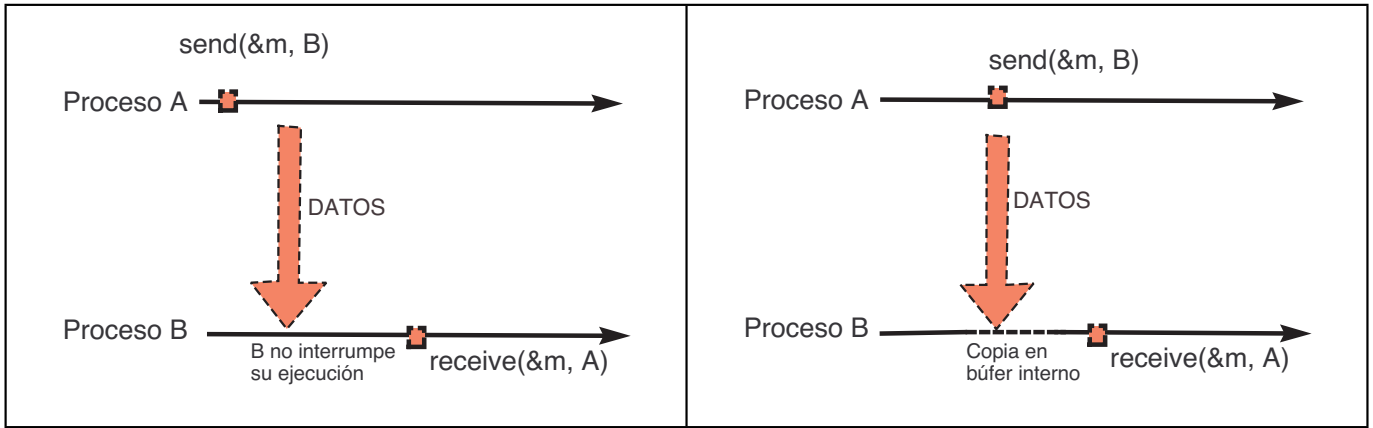
Manuel I. Capel
manuelcapel@ugr.es



Paso de mensajes bloqueante con búfer

Idea fundamental

El proceso emisor del mensaje *vuelve* inmediatamente al ejecutar la operación `send(...)` salvo que el búfer se llene. La operación `receive(...)` no *vuelve* hasta que se han recibido todos los datos en el receptor (si no hay hardware especializado)



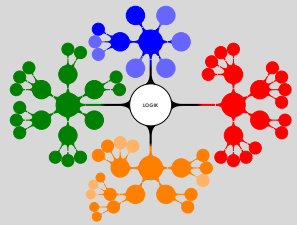
Con hardware de comunicación	Sin hardware especializado

Implementación del paso de mensajes bloqueante con búfer

Variantes

Hardware especializado	Sí	No
Bloqueo del receptor	No (transferencia inmediata)	Sí (transferencia sincronizada)

Manuel I. Capel
manuelcapel@ugr.es



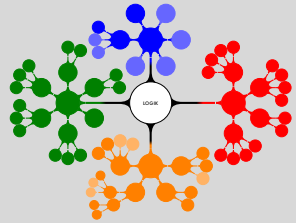
Causas de ineficiencia en la implementación del paso de mensajes bloqueante sin hardware especializado

Esquema de paso de mensajes	Motivo de la ineficiencia
Síncrono (citas)	Espera ociosa de 1 proceso
Mediante búfer	Sobrecarga por gestión del búfer

Posible solución:

- Definir operaciones send y receive que no se bloqueen
- $\Rightarrow$ Dejar en la responsabilidad del programador el asegurar que no se alteren los datos de los programas mientras están siendo transmitidos

Manuel I. Capel
manuelcapel@ugr.es



Operaciones no-bloqueantes

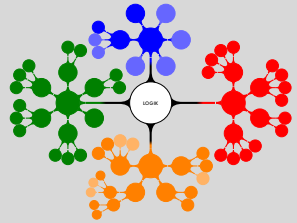
Idea fundamental

El cumplimiento de la propiedad de seguridad en los programas distribuidos con paso de mensajes se convierte en responsabilidad exclusiva del programador.

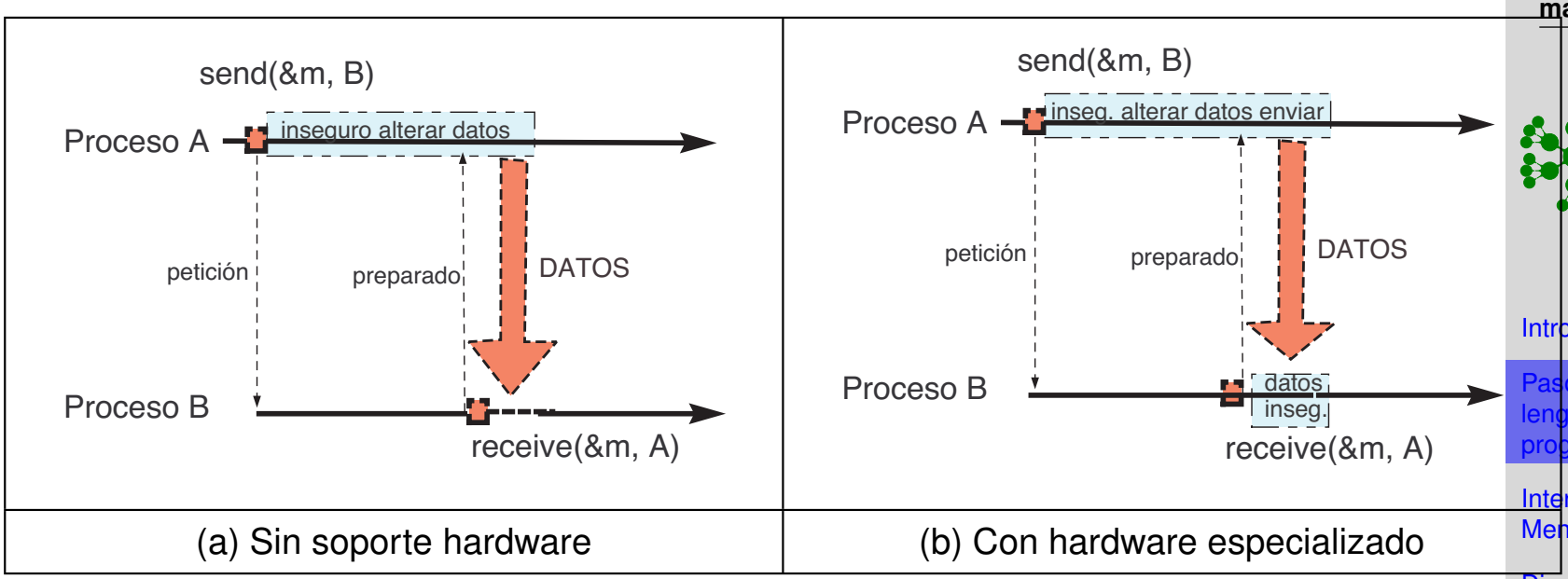
Características

- El tiempo de transmisión de los datos (emisor/`send()` – receptor/`send()`) no es *despreciable*, en general
- Las operaciones de envío se ejecutan y *vuelven* inmediatamente, antes incluso de que sea seguro modificar los datos que están en transmisión.
- En los lenguajes, se necesitan operaciones de *comprobación de estado* de los datos transmitidos
- La operación `receive(...)` vuelve o no lo hace, antes de terminarse la transmisión, dependiendo de si existe o no hardware especializado, respectivamente.

Manuel I. Capel
manuelcapel@ugr.es



Paso de mensajes no bloqueante



(a) Sin soporte hardware

(b) Con hardware especializado

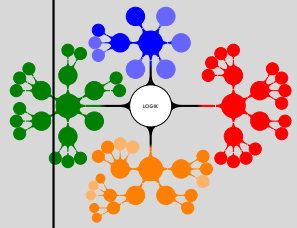
Paso de mensajes no-bloqueante sin búfer

Existe una operación de comprobación que indicaría cuándo es seguro acceder a los datos, que están siendo transmitidos, por parte del proceso receptor (caso b)

Paso de mensajes no-bloqueante con búfer

En este caso se reduce el tiempo de espera respecto del caso anterior porque la operación `receive()` provoca la transferencia inmediata de datos del búfer a la memoria propia del proceso receptor.

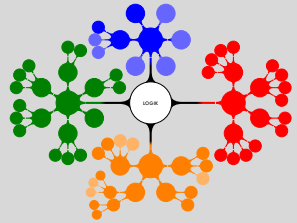
Manuel I. Capel
manuelcapel@ugr.es



Bibliotecas de paso de mensajes y patrones de interacción

- Modelo SPMD: **Message Passing Interface** (MPI)
  - Biblioteca de funciones para C, más el archivo de cabecera `mpi.h`
  - Biblioteca de funciones para FORTRAN, junto con el archivo de cabecera `mpif.h`
  - Órdenes para compilación: `mpicc, mpif77`: versiones de las órdenes habituales (`cc, f77`)
  - Órdenes específicas para ejecución de aplicaciones paralelas: `mpirun`
  - Herramientas para monitorización y depuración de programas paralelos
- No es la única biblioteca para la programación de aplicaciones paralelas y distribuidas (PVM, NX en el Intel Paragon, MPL en el IBM SP2, etc.), pero sí la más utilizada en la actualidad
- Se dispone de funciones de comunicación punto-a-punto, así como operaciones colectivas para involucrar a un grupo de procesos
- Los procesos pueden agruparse y formar *comunicadores* para permitir la definición del ámbito de las operaciones colectivas y un diseño modular de los programas distribuidos

Manuel I. Capel
manuelcapel@ugr.es



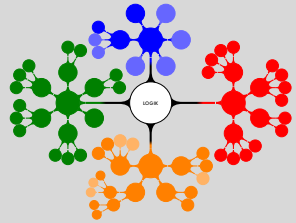
Bibliotecas de paso de mensajes y patrones de interacción-II

Ejemplo de programa que utiliza el *binding* de MPI para 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(); }
```

- Valor devuelto es `MPI_SUCCESS`: la función se ha realizado con éxito
- `MPI_COMM_WORLD` se refiere al comunicador universal, es decir, que incluye a todos los procesos de nuestro programa
- Se pueden definir otros comunicadores. Todas las funciones de MPI necesitan como argumento un comunicador

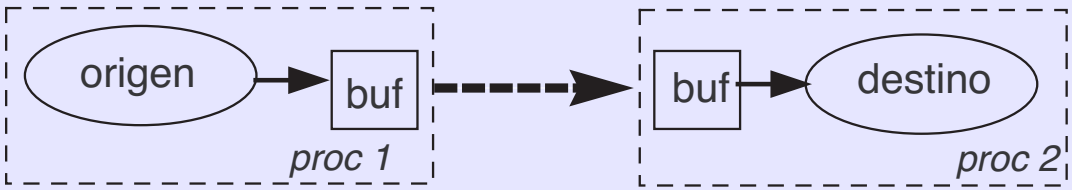
Manuel I. Capel
manuelcapel@ugr.es



Mensaje de MPI

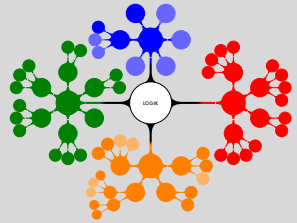
Bloque de datos trasferido entre procesadores y consiste en:

1. *Envoltorio* del mensaje:
   - destino / origen
   - etiqueta
   - comunicador



2. *Cuerpo* del mensaje:
   - búfer
   - contador
   - tipo de datos

Manuel I. Capel
manuelcapel@ugr.es



Operaciones de paso de mensajes utilizando MPI-II

```
int MPI_Send(void*bufer, int cont, MPI_Datatype t_datos,
             int destino, int etiqueta, MPI_Comm comunicador)
```

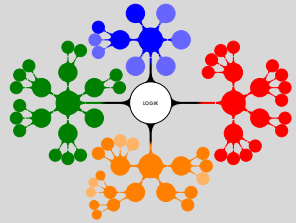
Campos de una operación MPI de envío con buffer

```
int MPI_Recv(void*bufer, int cont, MPI_Datatype t_datos,
             int origen, int etiqueta, MPI_Comm comunicador,
             MPI_Status *estado)
```

Sirve para comprobar la seguridad

Campos de una operación MPI de recepción con buffer

Manuel I. Capel
manuelcapel@ugr.es



Semántica de las operaciones de paso de mensajes bloqueantes

Concordancia entre mensajes

- La *envoltura* (origen, etiqueta, comunicador) del mensaje enviado ha de coincidir con la envoltura del mensaje recibido.

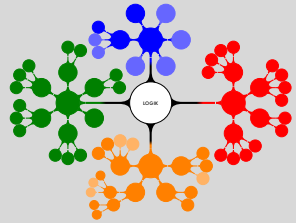
Bloqueo de las operaciones de comunicación

- Las operaciones: `MPI_Send`, `MPI_Ssend`, `MPI_Recv` se suspenden (no "vuelven") hasta que se *completan*

Completación de las operaciones

- `MPI_Send`: cuando termina de copiar el mensaje en el búfer `RMI`
- `MPI_Ssend`: cuando se produce la *cita* con el proceso que llama a `MPI_Recv` y hay concordancia entre las *envolturas* en cada parte de la comunicación
- `MPI_Recv`: existe ya un mensaje pendiente *conforme* (concordancia y compatibilidad de tipos) a la declaración de parámetros de esta operación de comunicación

Manuel I. Capel
manuelcapel@ugr.es



Semántica de las operaciones de paso de mensajes bloqueantes-II

Situaciones de error

- El tamaño del mensaje recibido es mayor que el esperado (programa aborta)
- Los tipos de datos declarados en el emisor y en el receptor son incompatibles (resultado indefinido)

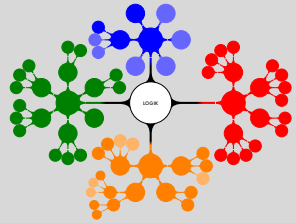
Sustitución de comodines

- Se puede sustituir `MPI_ANY_SOURCE` en el campo *origen* y `MPI_ANY_TAG` en el campo *etiqueta* sin que se produzca error en la operación de comunicación
- Sin embargo, el campo *comunicador* carece de *comodín*

Obtener información de los mensajes recibidos

- estado/tamaño: `MPI_Get_Count(`*`status,t_datos,cont`*`)`
- estado: campo `MPI_Status` (parámetro de `MPI_Recv`)

Manuel I. Capel
manuelcapel@ugr.es



Comunicación no bloqueante en `MPI`

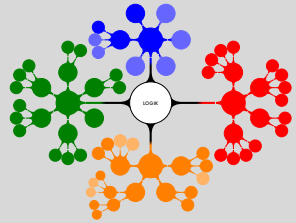
Idea fundamental de estas operaciones

- Evitar situaciones de *interbloqueo* de procesos debidas a una *incorrecta secuenciación* en el orden de ejecución de las operaciones de envío y recepción por parte de los procesos que intervienen en una comunicación
- Impedir el bloqueo de los procesos emisores de mensajes debido al desbordamiento del búfer interno al recibir

Operaciones `MPI` de envío/recepción no bloqueantes

- `MPI_Isend`: Inicia envío, pero vuelve de la llamada antes de comenzar a copiar el mensaje en el buffer
- `MPI_Irecv` : Inicia recepción pero vuelve de la llamada antes de comenzar a recibir ningún mensaje
- `MPI_Test(MPI_Request*r,int*flag,MPI_Status*s)` Chequea si la operación no bloqueante (identificada por el argumento `r`) ha finalizado: argumento `flag >0`

Manuel I. Capel
manuelcapel@ugr.es



Comunicación no bloqueante en MPI-II

Sondeo de estado de un mensaje

```
MPI_Iprobe(int origen, int etiqueta, MPI_Comm comunicador,
           int*flag, MPI_Status*estado)
```

Si hay mensaje pendiente (flag $>$ 0), entonces hay que recibirlo con una llamada a `MPI_Recv`. Comprobación existencia de mensaje no bloqueante.

```
MPI_Probe(int origen, int etiqueta, MPI_Comm comunicador,
          MPI_Status*estado)
```

Comprobación existencia de mensaje bloqueante. Esperar un mensaje sin conocer su procedencia, etiqueta o tamaño.

Esperar la completación de una operación

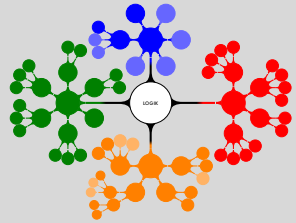
```
MPI_Wait(MPI_Request*solicitud, MPI_Status*estado)
```

Bloquea al proceso que la llama hasta que la operación identificada por `solicitud` termina de forma segura

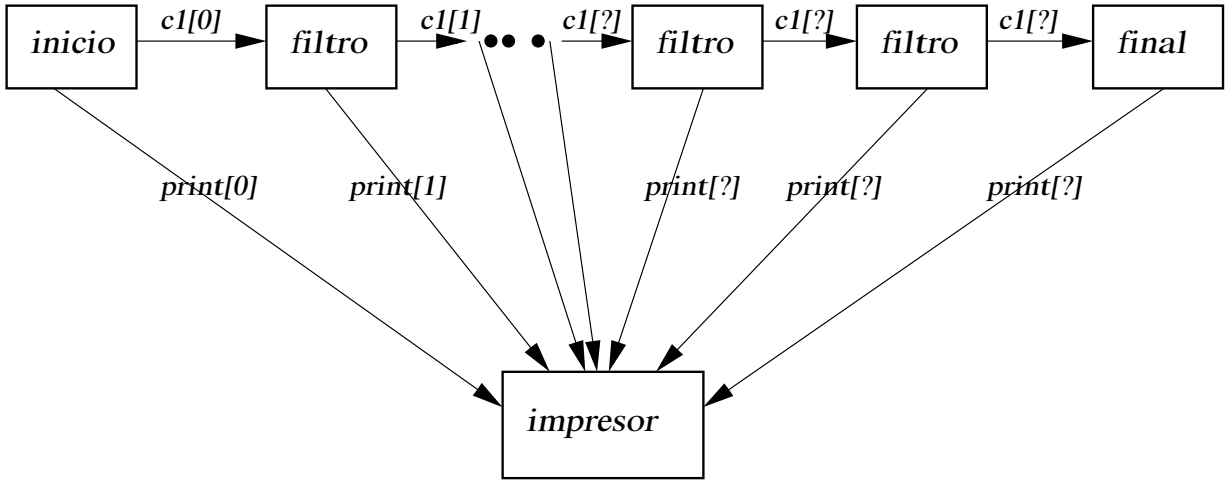
```
MPI_Request_free(MPI_Request*solicitud)
```

Libera al objeto `solicitud` de forma explícita

Manuel I. Capel
manuelcapel@ugr.es

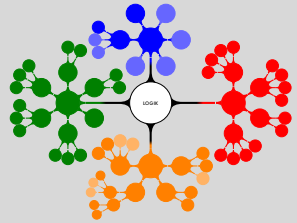


Ejemplo: *pipeline* de procesos que genera números primos



Generación de la serie de números primos con la *criba de Erastótenes*

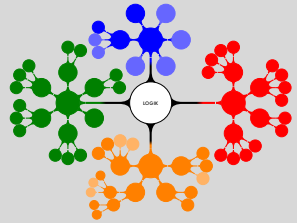
Manuel I. Capel
manuelcapel@ugr.es



Esquema de la solución con MPI

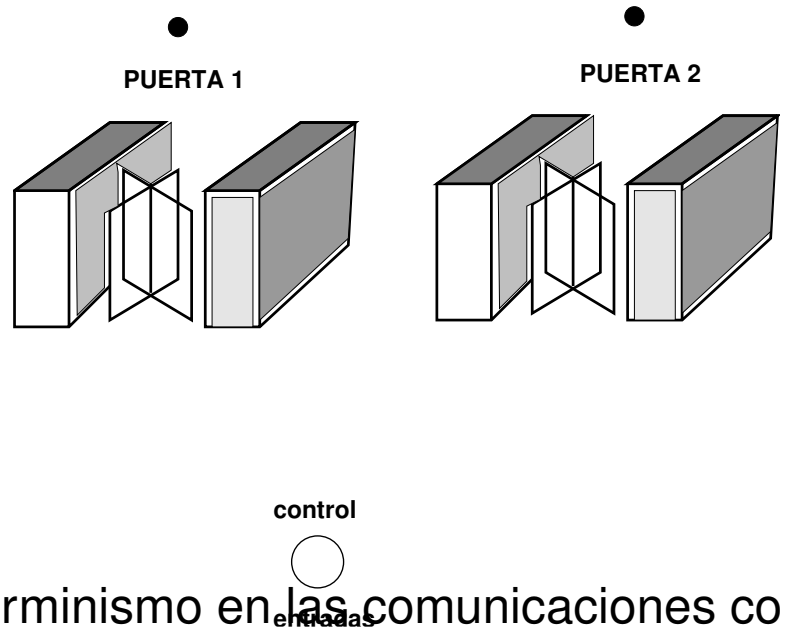
```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size( MPI_COMM_WORLD, &size );
   if (rank==0){ x=2; /* primer numero primo*/ i=1;
     MPI_Send(&x,1,MPI_INT,size-1,0,MPI_COMM_WORLD);
     while(!fin){
       i+=2; x=i;
       MPI_Send(&x,1,MPI_INT,rank+1,0,MPI_COMM_WORLD);
       MPI_Irecv(&x,1,MPI_INT,size-1,MPI_ANY_TAG,
           MPI_COMM_WORLD, &request);
       ....} }
     else if (rank == size-1){/*este es el impresor*/
     ... else{/*representa a los procesos filtros*/
      MPI_Recv(&valor,1,MPI_INT,rank-1,0,MPI_COMM_WORLD,&
          status);
      MPI_Send(&valor,1,MPI_INT,size-1,0,MPI_COMM_WORLD);
         while(!fin){
      MPI_Recv(&x,1,MPI_INT,rank-1,0,MPI_COMM_WORLD,&
          status);
           if (rank<(size-2))
           if (x%valor!=0)
           MPI_Send(&x,1,MPI_INT,rank+1,0,MPI_COMM_WORLD);
           MPI_Irecv(&x,1,MPI_INT,size-1,MPI_ANY_TAG,
               MPI_COMM_WORLD,&request);
           if (x==size-1) fin=true; } }
 MPI_Finalize();
```


Manuel I. Capel
manuelcapel@ugr.es



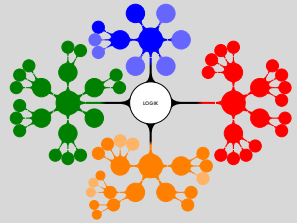
Tipos de procesos

- **Filtros**: son procesos transformadores de datos.
- **Clientes**: son procesos desencadenantes de algo.
- **Servidores**: son procesos reactivos.
- **Pares**: procesos idénticos que cooperan para resolver 1 problema.



Ejemplo: no determinismo en las comunicaciones con paso de mensajes

Manuel I. Capel
manuelcapel@ugr.es

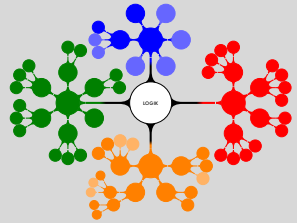


Modelos y lenguajes de programación distribuida

- Programación de procesos servidores con paso de mensajes síncrono:
  1. Inadecuación de las órdenes condicionales deterministas (`if, switch, ...`) de los lenguajes secuenciales para implementar servidores
  2. Sentencias no-deterministas en los lenguajes de programación facilitan la implementación de sistemas que reaccionan frente estímulos procedentes de su entorno.
- Solución incorrecta al problema del museo si se suponen operaciones bloqueantes de paso de mensajes:

```
Process P1::                Process P2::                Process P3::
                                                         /*Controlador*/
for(i=1;i<20i++){           for(i=1;i<20;i++){
 send(P3,1);                 send(p3,1);                for (j=1;j<20;j++){
 /*pasa 1 persona*/         /*pasa 1 persona*/          receive(P1, temp);
 }                           }                          cont=cont + temp;
                                                        receive(P2, temp);
                                                        cont=cont + temp;}
                                                       printf("%d", cont);

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


Manuel I. Capel
manuelcapel@ugr.es



Paradigma cliente/servidor de programación distribuida

Ideas fundamentales

-Comunicación muchos-a-uno,
-Cada comunicación es un *par*: (datos de entrada, resultados)
-Selección no-determinista entre varias comunicaciones posibles, en lado del servidor

Características

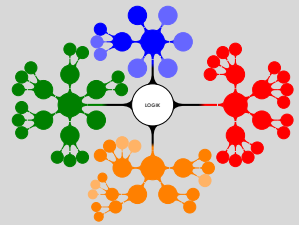
- Proceso cliente: solicita 1 servicio enviando un mensaje al servidor. Los procesos clientes tienen un carácter activo, ya que envían mensajes solicitando un servicio.
- Proceso servidor: tiene un carácter pasivo; recibe una petición de servicio de los clientes, devuelve un mensaje con los posibles resultados.

Simulación *no segura* utilizando paso síncrono de mensajes:

```
proceso cliente[i]
 *[TRUE ->

  Send(servidor, peticion())
  Receive(servidor,respuesta)]
```

```
proceso servidor
*[ (i:0..n) condicion(i);
cliente(i)::Receive(peticion());
  realizar.servicio
  Send(cliente[i],resultado)]
```


Manuel I. Capel
manuelcapel@ugr.es



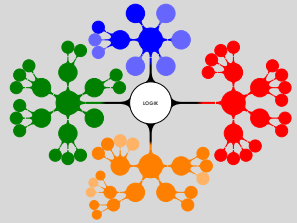
Inconvenientes de las comunicaciones de bajo nivel

- Código poco seguro: si falla el *servidor* o un cliente → bloqueo,
- Par `(Send(serv.,peticion()),Receive(serv.,resp.)` debe ser una única transacción lógica, no entrelazable con instrucciones de un tercer proceso.

Modelo de llamadas remotas:

- Soporta comunicación en los 2 sentidos
- Permite implementar con flexibilidad el modelo *Cliente/Servidor*
- Se permite a varios procesos llamar concurrentemente a un procedimiento
- El procedimiento llamado encapsula una serie de instrucciones que son ejecutadas inmediatamente
- Admite varias implementaciones:
  - *Llamada a procedimiento remoto* (RPC)
  - y la *Invocación Remota* (o modelo basado en citas)
- El significado de *remoto* es diferente para las RPC's y las invocaciones remotas

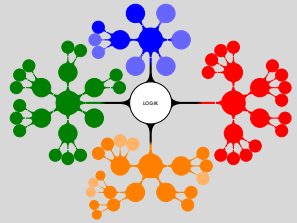
Manuel I. Capel
manuelcapel@ugr.es



RPI y RPC

- Procedimiento remoto: procedimiento *global* llamado por los procesos *clientes*.
- Concepto de *RPC*:
  - permite llamadas a procedimientos entre procesos ubicados en máquinas distintas,
  - sintaxis similar a la *llamada a procedimiento*, pero semántica completamente diferente,
  - el *servidor* ha de ejecutar un programa para atender las llamadas de los clientes.
- Semántica de las *RPCs:*
  - Se envían los argumentos de la llamada al servidor, bien directamente o través de un proceso creado a tal efecto.
  - Se bloquea el proceso cliente que ejecuta la llamada. Puede dejar su procesador libre.
  - Se re-construyen los argumentos de la llamada en el servidor, se ejecuta el procedimiento y se re-envían los resultados.
  - El servidor suele ligar un nombre simbólico al puerto del que recibe las llamadas.
  - Se pueden tener varias instancias de un mismo procedimiento ejecutándose concurrentemente.

Manuel I. Capel
manuelcapel@ugr.es



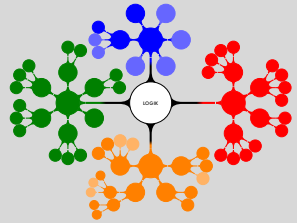
Órdenes con guarda

Sentencia “orden con guarda"

- Sirve para construir una sentencia (**Select**) de los lenguajes de programación distribuida que evita el bloqueo de servidores si sólo se dispone de paso de mensajes síncrono desde los procesos clientes
- Pasan a ser las sentencias componentes básicas de las construcciones alternativas y repetitivas de los lenguajes concurrentes del tipo anterior (paradigma cliente-servidor e invocaciones remotas)

```
<orden.alternativa>::= if <conjunto.ordenes.con.guarda> fi
<orden.repetitiva>::= do <conjunto.ordenes.con.guarda> do
<conjunto.ordenes.con.guarda>::= <orden.con.guarda>{[]<
    orden.con.guarda>}
<orden.con.guarda>::= <guarda> -> <lista.sentencias>
<guarda>::= <expresion.booleana> |
            <expresion.booleana>;receive(<argumentos>);|
            receive(<argumentos>)
}
```

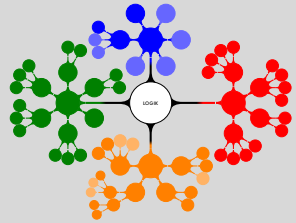

Manuel I. Capel
manuelcapel@ugr.es



Ejemplo: solución correcta al problema del museo

```
PUERTA(i:1..2)::
{  int s=0;
  do
   if (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
        }
     []
       (s<HORA.CIERRE && NOT PERSONA())->
        DELAY.UNTIL(s+1);//espera hasta el siguiente
            instante
        s:=s+1;]//cuenta otro tick
     []
        TRUE->DELAY.UNTIL (TIME() + 16*3600);
        //es la hora de cierre del museo; hay
        //que esperar 16 horas para activar el controlador
            .
        // TIME() devuelve una cuenta en segundos.
  fi
  do;
  send(CONTROL, Start();}
...
```

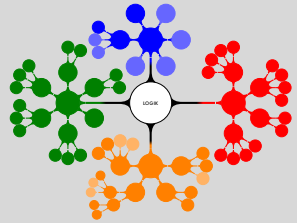

Manuel I. Capel
manuelcapel@ugr.es



Ejemplo: solución correcta al problema del museo-II

```
CONTROL::
{  int cont= 0;
  if  receive(PUERTA(1), Start());
      //desde cualquiera de los sensores
    [] //de las puertas se arranca
      receive(PUERTA(2), Start());//el controlador
  fi
  do
     []*[(j:1..2) receive(PUERTA(j), S())-> cont:= cont
         +1];
 //cuenta una persona mas, porque ha recibido la se\~nal
 //de cualquiera de las 2 puertas (no se puede saber cual)
  od;
  printf("numero de personas",%d, cont));
}

main(){
  cobegin PUERTA;CONTROL coend;}
```


Manuel I. Capel
manuelcapel@ugr.es

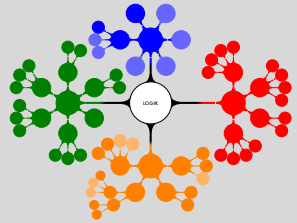


Invocaciones remotas

Implementación del modelo en los lenguajes de programación

- Se trata de sentencias de determinados lenguajes de Programación Concurrente
  - Llamar a un procedimiento que pertenece y es controlado por otro proceso posiblemente ubicado en una máquina remota.
  - El procedimiento remoto es una secuencia de una ó más instrucciones situadas en cualquier parte del código de un proceso.
- El proceso que llama y el proceso al cual pertenece el procedimiento remoto pueden ejecutarse por el mismo procesador.
- Se implementa definiendo un punto de entrada y una o varias sentencias de aceptación asociadas.

Manuel I. Capel
manuelcapel@ugr.es



Definición de los puntos de entrada

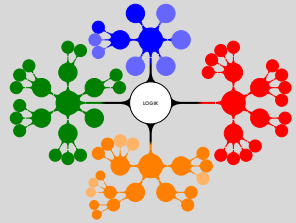
Una cola que atiende las llamadas de los procesos según el orden de llegada (cola FIFO) para contener las llamadas pendientes de los procesos clientes.

```
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
```


Manuel I. Capel
manuelcapel@ugr.es



Comunicación mediante citas

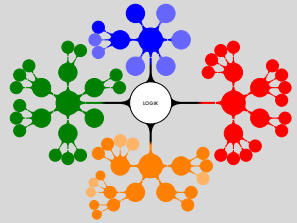
Condiciones

- El código asociado a una invocación remota (*"cuerpo de la cita"*) se ejecuta sólo cuando el proceso que define el punto de entrada *acepta* ejecutarlo
- Hay que declarar, al menos, 1 sentencia de aceptación por cada punto de entrada

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

```
PROCESO.A              PROCESO.B
  ENTRY E(...);
BEGIN                  BEGIN
  ACCEPT E(...);         P.E(...)
END                    END
```


Manuel I. Capel
manuelcapel@ugr.es

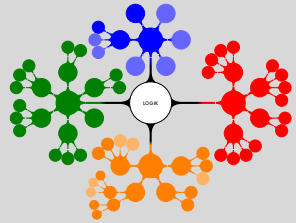


Introducción a la sentencia (Select) (o espera selectiva)

Ideas fundamentales

- Se trata de una versión de la *orden alternativa* (selección no determinista entre un conjunto de órdenes con guarda) que poseen algunos lenguajes de programación distribuida (Ada, SR, ...)
- Permite programar servidores cuyos clientes sean anónimos
- Mantiene un modo síncrono de comunicación cliente/servidor pero el servidor podría mantener varias comunicaciones preparadas, y recibir el mensaje de una sin importar el orden temporal en que se enviaron los mensajes
- Resuelve el problema de la recepción no determinista de una entre varias señales pendientes sin dependencias del orden temporal de envío (*problema del museo*)

Manuel I. Capel
manuelcapel@ugr.es



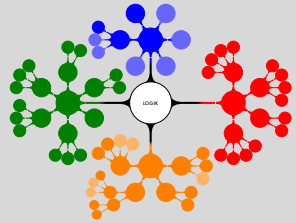
Instrucción de espera selectiva: Select

Semántica de la instrucción Select:

- Cada bloque que comienza en `when` se denomina una *alternativa* ($\equiv$ *orden guardada*)
- En cada alternativa, el texto desde `when` hasta `do` se denomina *guarda* de dicha alternativa, que puede incluir una expresión lógica ($\texttt{condicion}_i$) y una orden de recibir mensajes
- Las instrucciones `receive` nombran a otros procesos del programa concurrente ($\texttt{proceso}_i$), y cada uno referencia una variable local ($\texttt{variable}_i$), donde eventualmente se recibirá un valor del proceso emisor asociado.

```
select
  when condicion1 receive( variable1, proceso1 ) do
       sentencias1
  when condicion2 receive( variable2, proceso2 ) do
       sentencias2
 ...
  when condicionn receive( variablen, proceson ) do
       sentenciasn
end
```


Manuel I. Capel
manuelcapel@ugr.es



Sintaxis de las guardas

Guarda de una orden Select:

- La expresión lógica puede omitirse:

```
when receive( mensaje, proceso ) do
sentencias
```

o bien, equivalentemente:

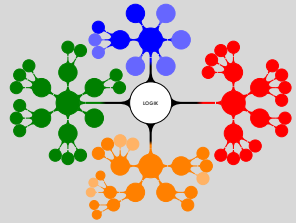
```
when (true) receive( mensaje, proceso ) do
        sentencias
```

- La sentencia receive puede no aparecer:

```
when condicion do
        sentencias
```

entonces, decimos que ésta es una *guarda* sin sentencia de entrada

Manuel I. Capel
manuelcapel@ugr.es



Evaluación y ejecución de las *guardas*

Guarda ejecutable durante la ejecución de un proceso `P`:

- La condición de la guarda se evalúa en ese momento a `true`
- Si tiene sentencia de entrada, entonces el proceso origen nombrado ya ha iniciado en ese momento una sentencia `send` (de cualquier tipo) con destino al proceso `P`, que concuerda con el `receive`

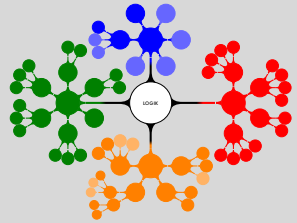
Guarda potencialmente ejecutable durante la ejecución de un proceso `P`:

- La condición de la guarda se evalúa a `true`
- Contiene una sentencia de entrada que nombra a un proceso que aún no ha iniciado la ajecución de una sentencia `send` hacia el proceso `P`

Guarda no ejecutable:

Se refiere al caso restante de evaluación de una guarda, es decir, cuando la condición de la guarda se evalúa como `false`

Manuel I. Capel
manuelcapel@ugr.es



Evaluación de las guardas de una instrucción Select:

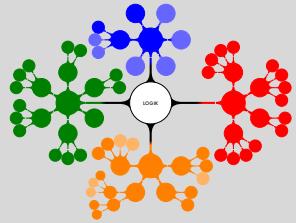
Evaluación de condiciones y estado de los procesos emisores:

Se produce cuando el flujo de control llega a una instrucción **select**, de esta forma se clasifican las guardas y se selecciona una alternativa para su ejecución

Determinación de la alternativa a ejecutar:

1. Si hay guardas ejecutables con sentencia de entrada, se selecciona aquella cuyo `send` se inició antes (esto garantiza a veces la equidad)
2. Si hay guardas ejecutables, pero ninguna tiene una sentencia de entrada, se selecciona *no determinísticamente* una cualquiera
3. Si no hay ninguna guarda ejecutable, pero sí hay guardas potencialmente ejecutables, la instrucción *espera* (suspende el proceso) hasta que alguno de los procesos nombrados en esas guardas inicie un `send`; en ese momento acabará la espera y se seleccionará la guarda correspondiente a ese proceso
4. Si no hay guardas ejecutables ni potencialmente ejecutables, no se puede seleccionar ninguna guarda y el programa suele producir un error o levantar una excepción

Manuel I. Capel
manuelcapel@ugr.es

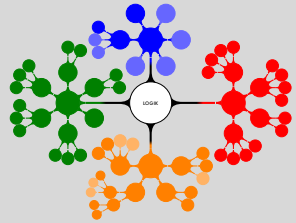


Ejecución de la instrucción select-II

Consideraciones importantes:

- Hay que tener en cuenta que la ejecución de una instrucción `select` conlleva potencialmente esperas y, por tanto, se pueden producir esperas indefinidas (interbloqueo)
- Existe un instrucción **select** con prioridad: la selección de la alternativa a ejecutar dejaría de ser *no determinística*, porque el orden en el que aparecen las alternativas establecerá la prioridad de ejecución de cada una
- Para implementar un proceso servidor, la instrucción **select** ha de programarse dentro de un bucle: en cada iteración se vuelve a evaluar las guardas y se selecciona no deterministícamente 1 de ellas para su ejecución

Manuel I. Capel
manuelcapel@ugr.es



Ejemplo: productor-consumidor con búfer FIFO

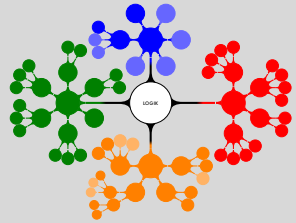
```
{ Productor (P) }
while true do
begin
  v:=Produce();
  s_send(v,B);
end
```

```
{ Intermedio (B) }
var esc, lec, cont : integer := 0 ;
buf : array[0..tam-1] of integer ;
begin
 while true do
  select
   when cont < tam receive(v,P) do
      buf[esc]:= v ;
      esc  := (esc+1) mod tam ;
      cont := cont+1 ;
   when 0 < cont receive(s,C) do
      s_send(buf[lec],C);
      lec  := (lec+1) mod tam ;
      cont := cont-1 ;
  end
end
```

```
{ Consumidor (C) }
while true do
begin
  s_send(s,B);
  receive(v,B);
  Consume(v);
end
```

No se conoce de antemano el orden de las peticiones de inserción/extracción. Las guardas garantizan la propiedad de seguridad en el acceso concurrente al búfer.

Manuel I. Capel
manuelcapel@ugr.es



Instrucción Select con guardas indexadas

Sintaxis

A veces es necesario replicar una alternativa. En estos casos se puede usar una sintaxis que evita reescribir el código muchas veces, con esta sintaxis:

```
for indice := inicial to final
        when condicion receive( mensaje, proceso ) do
                sentencias (indice);
```

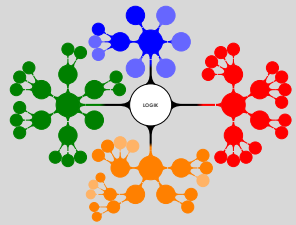
Tanto la condición, como el mensaje, el proceso o las sentencias componentes pueden contener referencias a la variable indice

Equivalencia:

La construcción con índices es equivalente a:

```
when condicion receive( mensaje, proceso ) do
        sentencias { se sustituye indice por inicial }
when condicion receive( mensaje, proceso ) do
        sentencias { se sustituye indice por inicial + 1 }
  ...
when condicion receive( mensaje, proceso ) do
```


Manuel I. Capel
manuelcapel@ugr.es



Ejemplo: Select con guardas indexadas

`suma` es un vector de n enteros, y `fuente[0]`, `fuente[1]`, etc... son n procesos:

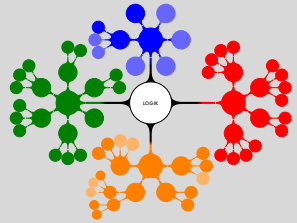
```
for i := 0 to n-1
        when suma[i] < 100 receive( numero, fuente[i] ) do
          suma[i] := suma[i] + numero ;
```

Lo cual es equivalente a:

```
when suma[0] < 100 receive( numero, fuente[0] ) do
        suma[0] := suma[0] + numero ;
when suma[1] < 100 receive( numero, fuente[1] ) do
        suma[1] := suma[1] + numero ;
...
when suma[n-1] < 100 receive( numero, fuente[n ? 1] ) do
        suma[n-1] := suma[n-1] + numero ;
```

En un **select** se pueden combinar una o varias alternativas indexadas con alternativas normales no indexadas.
Resultado de la orden: se recibirá 1 número de un proceso `fuente[i]` si `suma[i]` es menor que 100 (antes de recibir)

Manuel I. Capel
manuelcapel@ugr.es

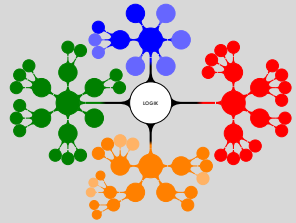


Select con sentencia else

Se trata de programar un proceso servidor que sume los números recibidos desde *n* procesos, que se ejecutan cada uno continuamente, hasta que cada suma iguale o supere el valor 100. Las guardas son siempre *ejecutables* hasta llegar a la situación de terminación.

```
var suma : array[0..n-1] of integer := (0,0,...,0) ;
continuar : boolean := true ;
numero : integer ;
  begin
    while continuar do begin
      select
        for i := 0 to n - 1
          when suma[i] < 100 receive( numero, emisor[i] )
              do
           suma[i] := suma[i]+numero ; { sumar }
           continuar := true ; {metainstrucción: no es
               necesaria pero se incluye para clarificar}
          end
          else continuar:= false;
    end
  end
```


Manuel I. Capel
manuelcapel@ugr.es



Implementación del modelo RMI en Java

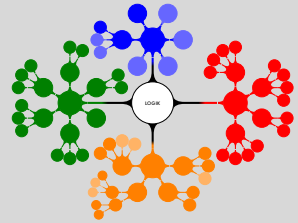
- *Remote Method Invocation* (RMI): tecnología estándar de Java para programación distribuida.
- RMI: basada en *llamadas remotas* y POO.
- En la parte del *servidor*:
  - Sub-interfaz de *Remote*:

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

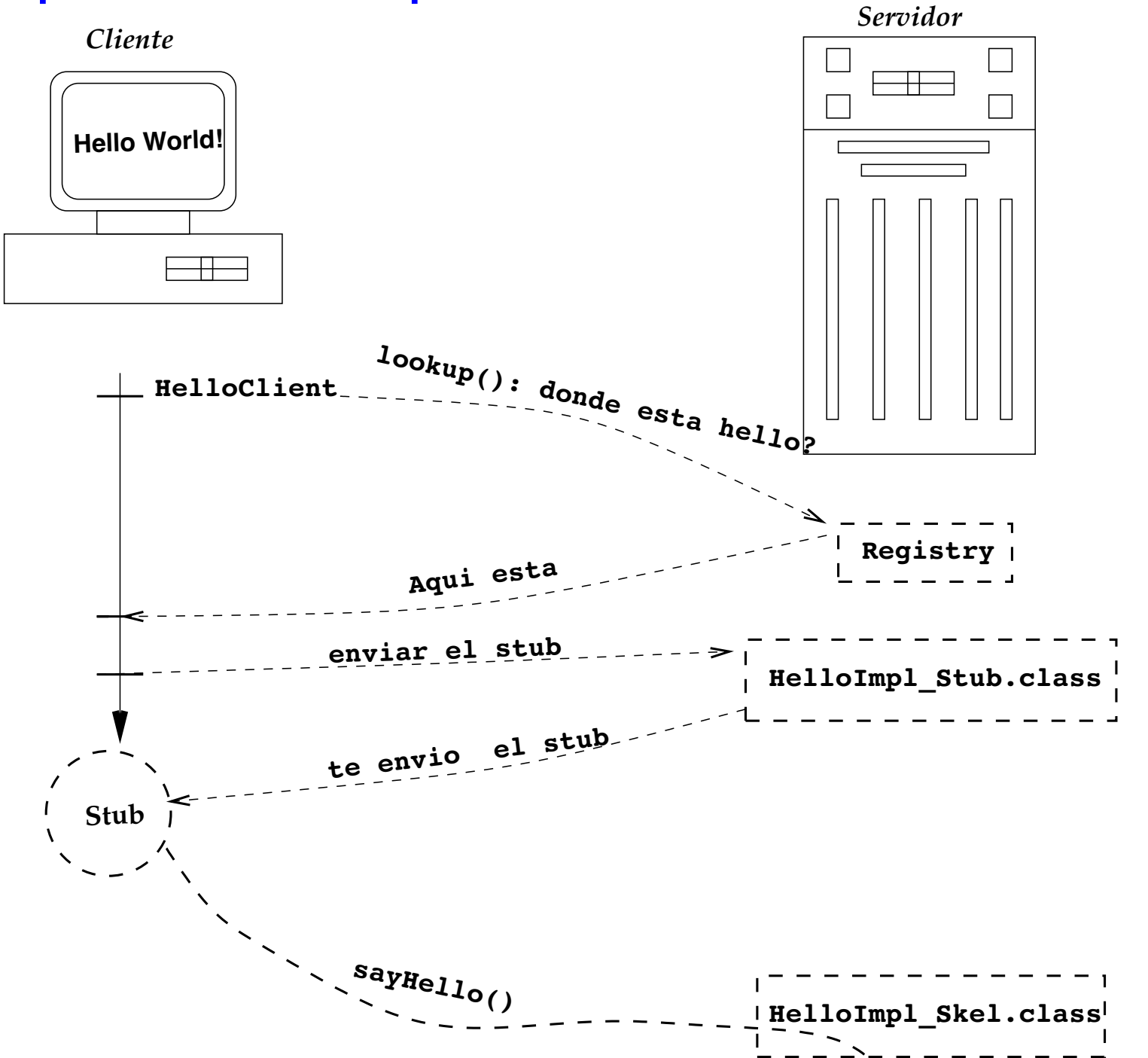
  - Clase que implementa los métodos remotos:

```
public class HelloImpl extends
    UnicastRemoteObject implements Hello {

  public HelloImpl() throws RemoteException{
    super();
  }
  public String sayHello() throws RemoteException
      {
    returns ``Hello World!!!'';
  }
```

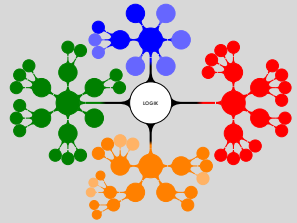

Manuel I. Capel
manuelcapel@ugr.es



Arquitectura de una Aplicación con RMI en Java



Manuel I. Capel
manuelcapel@ugr.es



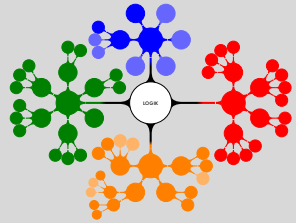
Implementación del modelo RMI en Java–II

```
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){ ...} } }
```

- En la parte de los clientes:

```
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){...}
}
```


Manuel I. Capel
manuelcapel@ugr.es

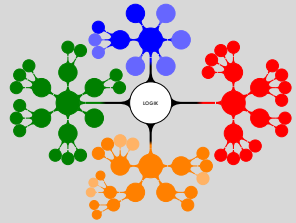


Semántica del paso de parámetros con RMI

Tipo de paso de parámetros e interfaces implementadas

- Java *pasa por referencia* los parámetros referidos a objetos locales a una máquina.
- Es bastante ineficiente pasar por referencia objetos que contienen sólo datos entre máquinas remotas.
- Reglas de paso de parámetros para llamadas a 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 implementarán las clases:
    - *remote interface:* se pasan por referencia,
    - *serializable interface:* se pasan por copia.

Manuel I. Capel
manuelcapel@ugr.es

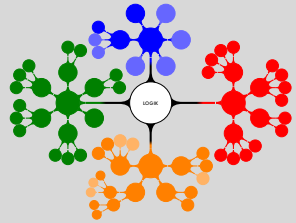


Semántica del paso de parámetros con RMI–II

Disfuncionalidades en las llamadas a métodos remotos

- Coexistencia de 2 semánticas diferentes para la misma sintaxis de llamada a los métodos de una misma variable-objeto.
- La ejecución de la llamada a un método puede producir diferentes resultados, según sea local o remoto.
- Se permite el polimorfismo cuando se programa con RMI, por tanto, no hay forma de averiguar estáticamente si un objeto es local o remoto cuando se invocan sus métodos.

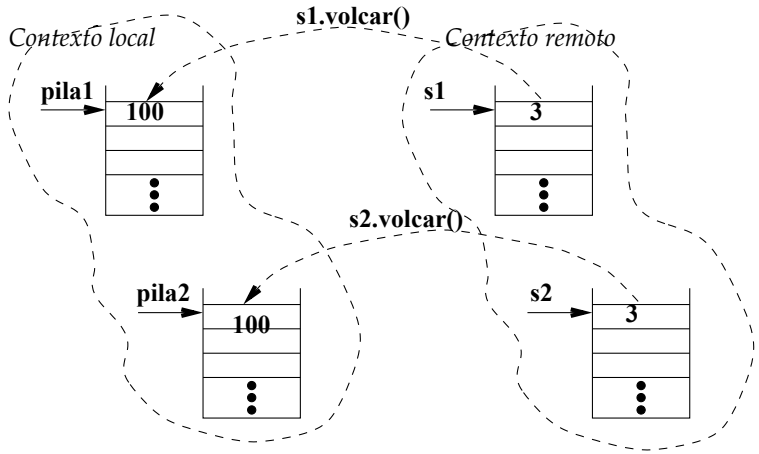
Manuel I. Capel
manuelcapel@ugr.es



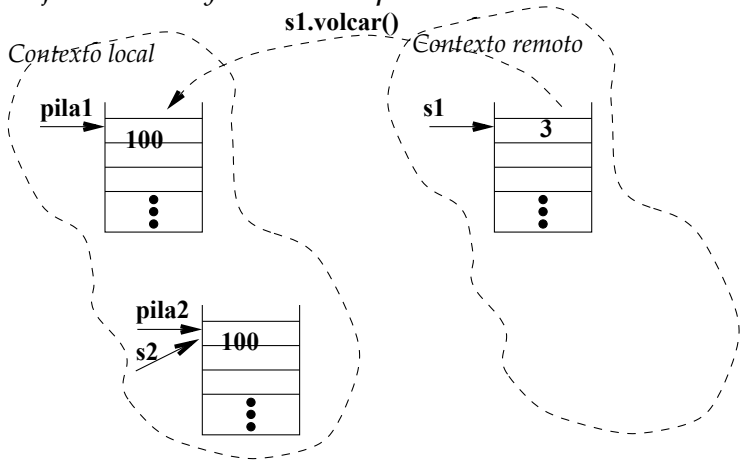
Dificultad en la utilización del paso de parámetros en RMI

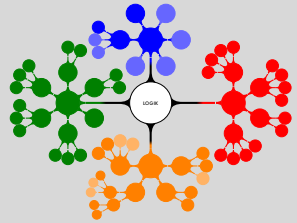
```
public interfaca PilaRemota extends
 Serializable{

   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());
   }
}
```

Objetos remotos como parametros actuales



Objetos remotos y locales como parametros actuales





Bibliografía

Para más información, ejercicios, bibliografía adicional:

Capítulo 3: Sistemas basados en paso de mensajes de "Programación Concurrente y de Tiempo Real". M.I.Capel(2022), Garceta (Madrid).

"Paso de mensajes síncrono/asíncrono":
`https://en.wikipedia.org/wiki/Message_passing#Synchronous_versus_asynchronous_message_passing`

"Citas- como mecanismo distribuido síncrono":
`http://www.dalnefre.com/wp/2010/07/message-passing-part-1-synchronous-rendezvous/`

"Canales `core.async` de Clojure":
`https://clojure.org/news/2013/06/28/clojure-clore-async-channels`

"Sentencia de elección no determinística":
`https://www.quora.com/Network-Programming/Network-Programming-How-is-select-implemented`

Manuel I. Capel
manuelcapel@ugr.es