

# Concurrencia con Paso de Mensajes en Java MPJ-Express

## Tema 6 - Programación Concurrente y de Tiempo Real

Antonio J. Tomeu<sup>1</sup>

<sup>1</sup>Departamento de Ingeniería Informática  
Universidad de Cádiz

PCTR, 2019

1. Concepto de Paso de Mensajes
2. Arquitectura del Modelo
3. Identificación de Procesos
4. Semántica de las Operaciones
5. Bibliotecas de Paso de Mensajes
6. MPJ-Express
7. Funciones Básicas de MPJ-Express
8. Comunicadores en MPJ-Express. El objeto Status
9. Comunicación Punto a Punto
10. Comunicación Colectiva

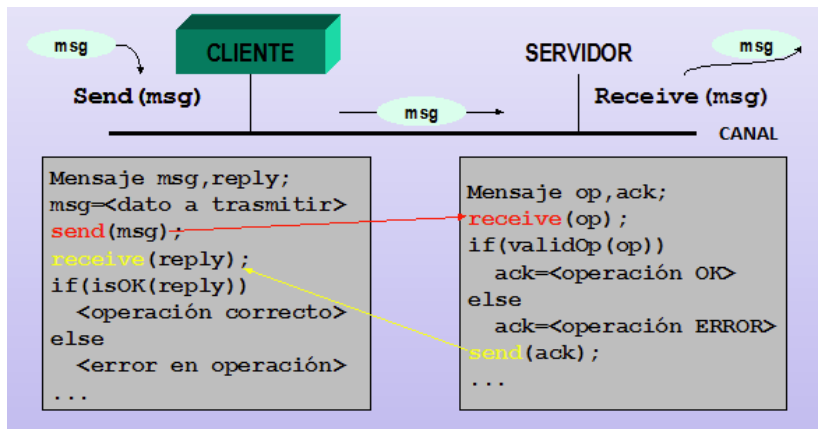
# Concepto de Paso de Mensajes

- ▶ Todos los modelos teóricos de control de la concurrencia analizados (y sus implementaciones reales en lenguajes como Java o C++) se basan en la existencia de **memoria compartida**, a través de la cuál las hebras se comunican a efectos de sincronizar su ejecución
- ▶ En sistemas donde no existe memoria compartida (sistemas distribuidos, clusters de procesadores), ese conjunto de técnicas no es aplicable
- ▶ Se necesita un nuevo modelo, que estructure la comunicación y sincronización entre hebras mediante el envío de mensajes (a través de algún medio de comunicación), en lugar de leer y escribir en una memoria común que ya no existe.
- ▶ Ahora las hebras (y en general las tareas) se comunican mediante operaciones de envío y recepción de información de naturaleza explícita.

# Modelo de Paso de Mensajes: Arquitectura Básica

- ▶ Se utiliza el concepto de canal como medio común a las tareas que se comunican
- ▶ Un canal es una abstracción software que puede modelar a diferentes realidades físicas, como una red de comunicaciones, internet, o incluso un espacio de memoria común, si este existe y así se decide
- ▶ Ahora los tareas envían información al canal (mediante una operación tipo *send(msg)*)...
- ▶ ... o la reciben desde el canal (mediante una operación tipo *receive(msg)*)

# Modelo de Paso de Mensajes: Arquitectura Básica



# Modelo de Paso de Mensajes: Ventajas e Inconvenientes

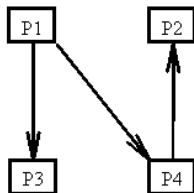
- ▶ Es aplicable en prácticamente cualquier arquitectura de computadores:
  - ▶ en sistemas de tipo distribuido
  - ▶ en sistemas paralelos con memoria común
  - ▶ en sistemas paralelos sin memoria común
  - ▶ en arquitecturas heterogéneas (cluster de procesadores)
- ▶ No existe el problema del acceso a datos compartidos y toda la problemática que el mismo conlleva.
- ▶ La programación de soluciones es ahora probablemente más compleja, ya que es necesario considerar nuevos aspectos ligados a la comunicación: direcciones, sincronización, etc.
- ▶ Es necesario un nuevo enfoque conceptual para distribuir un problema complejo entre tareas que ahora no comparten memoria, y que están distribuidas entre máquinas que probablemente son físicamente diferentes.

# Identificación de Procesos

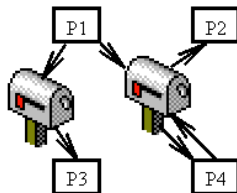
- ▶ Es necesario determinar cómo se identifican entre sí los procesos emisores y receptores de una aplicación (típicamente distribuida)
- ▶ Identificación por Denominación Directa
  - ▶ se utilizan directamente los identificadores de los procesos
  - ▶ deben asignarse previamente a la ejecución del programa, y deben mantenerse
  - ▶ si se cambian, hay que recompilar el código de la aplicación
- ▶ Identificación por Denominación Indirecta
  - ▶ se utiliza un objeto intermedio denominado **buzón**
  - ▶ los procesos designan al mismo buzón como destino u origen de los mensajes que se intercambian

# Denominación Indirecta: Tipos de Buzones

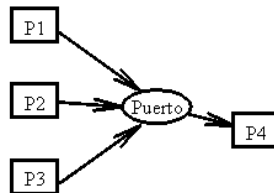
Relación Tipo	1-1 Canales	muchos-1 Puertos	muchos-muchos Buzones Generales
------------------	----------------	---------------------	------------------------------------



Direccionamiento  
directo



Direccionamiento  
indirecto (buzones)



Direccionamiento  
indirecto (puertos)



# Estructura de Procesos (según tipo de denominación)

## Código 1: codigo\_t6/dirdirecto.java

```
1 //direccionamiento directo
2 Proceso P0          Proceso P1
3 int dato;           int x;
4 produce(dato);       receive(&x, P0);
5 send(&dato, P1);     consume(x);
```

## Código 2: codigo\_t6/dirindirecto.java

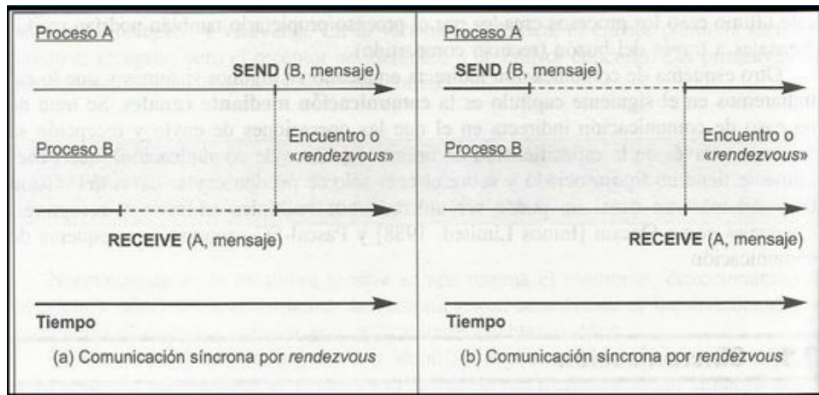
```
1 //direccionamiento indirecto
2 Channel of Integer Buzon;
3 Proceso P0          Proceso P1
4 int dato;           int x;
5 produce(dato);       receive(&x, Buzon);
6 send(&dato, Buzon);  consume(x);
```

# Semántica de las Operaciones send y receive

- ▶ La semántica de las operaciones puede variar
- ▶ Dependerá de la seguridad y del modo de comunicación
- ▶ Propiedad de seguridad: una operación send es segura cuando existen garantías de que el valor recibido por el proceso destino es el que tienen los datos justo antes de la llamada en el proceso de origen.

- ▶ Operaciones Síncronas: el intercambio de un mensajes es una operación atómica que exige la participación simultánea del emisor y el receptor (rendezvous)
- ▶ Operaciones Asíncronas: el emisor envía el mensaje sin bloquearse, y el emisor lo puede recoger más tarde; requiere de la existencia de un buzón intermedio

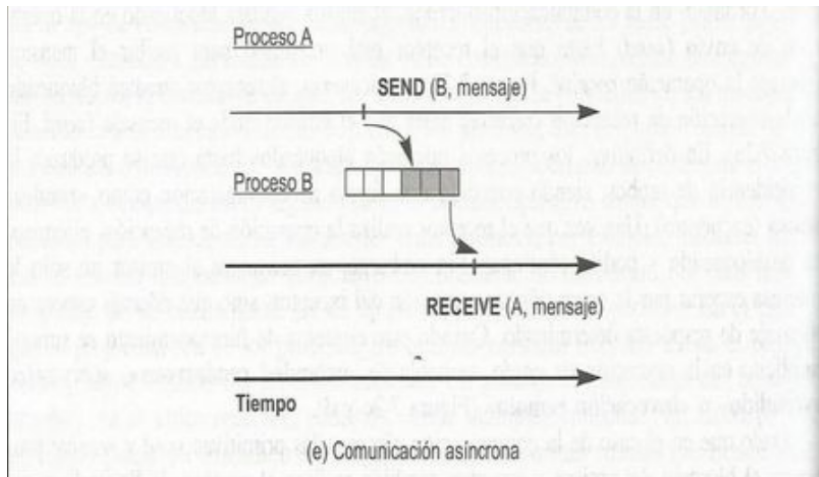
# Semántica de Operaciones Síncronas (*rendezvous* o cita)



# Semántica de Operaciones Síncronas (rendezvous o cita)

- ▶ La comunicación se lleva a cabo mediante un enlace directo entre los procesos implicados
- ▶ Las operaciones son bloqueantes. Un proceso A no envía si el proceso B de destino no está preparado para recibir. Mientras, A espera.
- ▶ Un proceso B espero bloqueado para recibir a que haya un proceso A que efectúe un envío.
- ▶ Por tanto, antes de que los datos se envíen de forma física, ambos procesos han de estar preparados para participar en el intercambio. Debe producirse una cita o rendezvous.
- ▶ El concepto de cita, por tanto, implica:
  - ▶ sincronización entre emisor y receptor
  - ▶ el emisor puede realizar aserciones acerca del estado del receptor en el punto de sincronización

# Semántica de Operaciones Asíncronas



- ▶ receive posee la misma semántica que el paso de mensajes síncrono. Extraerá los mensajes del buzón intermedio, pero bloqueará al receptor si dicho buzón está vacío.
- ▶ send posee una semántica diferente, ya que ahora no tiene carácter bloqueante.
- ▶ El proceso emisor envía (send) su mensaje al buzón, y continúa con su ejecución normal. Eventualmente, el envío podría llegar a tener carácter bloqueante si el el buzón intermedio llegara a llenarse.

- ▶ Actualmente existen múltiples bibliotecas que implementan el modelo de paso de mensajes
- ▶ MPI (C y C++, Fortran, etc.) y sus versiones para Java (MPJ-Express) son muy utilizadas para programación paralela con paso de mensajes en arquitecturas multicore de memoria común y clusters de procesadores.
- ▶ JCSP es una implementación más abstracta y más cercana al concepto de canal expuesto
- ▶ La programación en Java con sockets y con el *framework* RMI ofrece soporte a la programación distribuida basada en el paso de mensajes con un mayor nivel de abstracción.



# Message Passing Interface (MPI)

- ▶ Es uno de los *frameworks* más utilizados para programación paralela y distribuida mediante paso de mensajes entre procesos
- ▶ En teoría, MPI soporta el nivel MIMD de la taxonomía de Flynn
- ▶ Sin embargo, en la práctica, la mayoría de aplicaciones escritas con él utilizan el modelo SPMD (Single Program Multiple Data), donde todos los procesos ejecutan el mismo programa sobre datos diferentes.
- ▶ Obviamente, esos programas no necesariamente han de ejecutar la misma instrucción al mismo tiempo.
- ▶ MPJ-Express es una implementación de MPI para Java (sobre multicore y clusters)

# Funciones de MPJ-Express

- ▶ MPJ-Express proporciona funciones de comunicación punto a punto para dos procesos, y funciones colectivas que involucran a un grupo de procesos
- ▶ Los procesos pueden agruparse mediante el concepto de comunicador, lo que permite determinar el ámbito de las funciones colectivas y el diseño modular
- ▶ MPJ-Express incluye funciones para realizar operaciones:
  - ▶ Básicas
  - ▶ Comunicaciones punto a punto
  - ▶ Comunicaciones colectivas
  - ▶ Control de la topología del espacio de procesos
  - ▶ Gestión e interrogatorio del entorno

# Funciones Básicas de MPJ-Express

- ▶ `MPI.Init()` para iniciar la ejecución paralela
- ▶ `MPI.COMM_WORLD.Size()` para determinar el número de procesos en ejecución paralela
- ▶ `MPI.COMM_WORLD.Rank()` para que cada procesos obtenga su identificador dentro de la colección de procesos que forman la aplicación
- ▶ `MPI.Finalize()` para terminar la ejecución del programa

# Tipos de Datos de MPJ-Express

TIPO DE DATO DE MPJ-Express	TIPO DE DATO DE JAVA
MPI.BYTE	byte
MPI.CHAR	char
MPI.SHORT	short
MPI.BOOLEAN	boolean
MPI.INT	int
MPI.LONG	long
MPI.FLOAT	float
MPI.DOUBLE	double
MPI.OBJECT	Object

# Primer Ejemplo con MPJ-Express

## Código 3: codigo\_t6/HelloWorld.java

```
1  import mpi.*;
2  public class HelloWorld {
3
4  public static void main(String args[]) throws Exception {
5      MPI.Init(args);
6      int me = MPI.COMM_WORLD.Rank();
7      int size = MPI.COMM_WORLD.Size();
8      //cuerpo ejecutable de la tarea...
9      System.out.println("Hola desde el proceso <"+me+">");
10     //fin del cuerpo ejecutable...
11     MPI.Finalize();
12 }
13 }
```

# Ejecución del Primer Ejemplo

Ejecutar con `mpjrun.bat -np 6 HelloWorld` y el output es:

**Código 4:** `codigo_t6/outputhelloworld.java`

```
1 C:\mpj-user>mpjrun.bat -np 6 HelloWorld
2 MPJ Express (0.44) is started in the multicore configuration
3 Hola desde el proceso <1>
4 Hola desde el proceso <4>
5 Hola desde el proceso <2>
6 Hola desde el proceso <5>
7 Hola desde el proceso <3>
8 Hola desde el proceso <0>
```

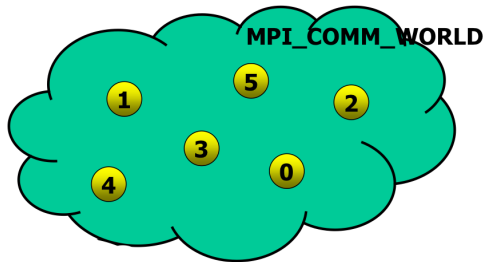
# Comunicadores de MPJ-Express I

- ▶ Un comunicador es un universo de comunicación de MPJ-Express que define a un grupo de procesos que se pueden comunicar entre sí
- ▶ Todas las funciones de comunicación de MPJ-Express necesitan involucrar a un comunicador
- ▶ Todo mensaje de MPJ-Express debe involucrar a un comunicador
- ▶ El comunicador especificado en las llamadas a operaciones de envío y recepción debe concordar para que la comunicación sea posible
- ▶ Pueden existir múltiples comunicadores y un proceso puede formar parte de varios distintos, si es necesario.

- ▶ Dentro de un comunicador los procesos se numeran (y se identifican) consecutivamente (empezando por 0). Este identificador es el rank del proceso
- ▶ Un proceso que pertenece a dos comunicadores puede tener rank diferentes en cada uno de ellos
- ▶ MPJ-Express proporciona un comunicador básico por defecto (MPI.COMM\_WORLD) que incluye a todos los procesos



# Comunicador por Defecto: MPI\_COMM\_WORLD



# Comunicaciones Punto a Punto

- ▶ Hay dos procesos: emisor y receptor
- ▶ Deben programarse explícitamente operaciones de envío (en el emisor) y de recepción (en el receptor)
- ▶ Todo mensaje de MPJ-Express debe involucrar a un comunicador
- ▶ Las operaciones más simples del API para enviar y recibir son:
  - ▶ `void Send(bufer, offset, datatype, destination, tag)`
  - ▶ `Status Recv(bufer, offset, datatype, source, tag)`
- ▶ Cada mensaje contiene un identificador o tag

# Comunicación Punto a Punto I



# Ejemplo de Comunicación Punto a Punto I

## Código 5: codigo\_t6/puntoAPunto.java

```
1  import mpi.*;
2  public class puntoAPunto {
3
4  public static void main(String args[]) throws Exception {
5      MPI.Init(args);
6      int rank = MPI.COMM_WORLD.Rank();
7      int size = MPI.COMM_WORLD.Size();
8      int emisor = 0; int receptor = 1;
9      int tag = 100; int unitSize = 1;
10
11     if(rank==emisor){ //codigo del emisor
12         int bufer[] = new int[1];
13         bufer[0] = 1200;
14         MPI.COMM_WORLD.Send(bufer, 0, unitSize, MPI.INT, receptor,
15                               tag);
16     } else{ //codigo del receptor
17         int revbufer[] = new int[1];
18         MPI.COMM_WORLD.Recv(revbufer, 0, unitSize, MPI.INT, emisor,
19                               tag);
```

# Ejemplo de Comunicación Punto a Punto II

```
18     System.out.println("Receptor ha recibido el dato:  
19         "+revbufer[0]);  
20     }  
21     MPI.Finalize();  
22  
23 }  
24 }
```

## Código 6: codigo\_t6/helloWorld2.java

```
1  import mpi.*;
2
3  class helloWorld2 {
4      static public void main(String[] args) throws MPIException {
5          MPI.Init(args);
6          int myrank = MPI.COMM_WORLD.Rank();
7          int size    = MPI.COMM_WORLD.Size();
8          if(myrank == 0) {
9              char [] message = "Hello, there".toCharArray();
10             for(int proceso=1; proceso<size; proceso++)
11                 MPI.COMM_WORLD.Send(message, 0, message.length, MPI.CHAR,
12                                     proceso, 99);
13         }
14         else {
15             char [] message = new char [20];
16             MPI.COMM_WORLD.Recv(message, 0, 20, MPI.CHAR, 0, 99);
17             System.out.println("received:" + new String(message) +
18                               ":",");
19         }
20     }
21 }
```

# Comunicación Uno a Muchos II

```
18     MPI.Finalize();  
19 }  
20 }
```

- ▶ El resultado de una operación de recepción es un objeto de clase Status
- ▶ Este objeto permite obtener información del mensaje recibido y de su tamaño
- ▶ Esto se produce cuando los procesos que envían y reciben sincronizan en el mensaje



# Ejemplo de Uso de Status I

## Código 7: codigo\_t6/puntoAPuntoStatus.java

```
1  import mpi.*;
2  public class puntoAPuntoStatus {
3
4  public static void main(String args[]) throws Exception {
5      MPI.Init(args);
6      int rank = MPI.COMM_WORLD.Rank();
7      int size = MPI.COMM_WORLD.Size();
8      int emisor = 0; int receptor = 1;
9      int tag = 100; int unitSize = 1;
10
11     if(rank==emisor){ //codigo del emisor
12         int bufer[] = new int[1];
13         bufer[0] = 1200;
14         MPI.COMM_WORLD.Send(bufer, 0, unitSize, MPI.INT, receptor,
15                               tag);
16     } else{ //codigo del receptor
17         int revbufer[] = new int[1];
18         Status status = new Status();
```

# Ejemplo de Uso de Status II

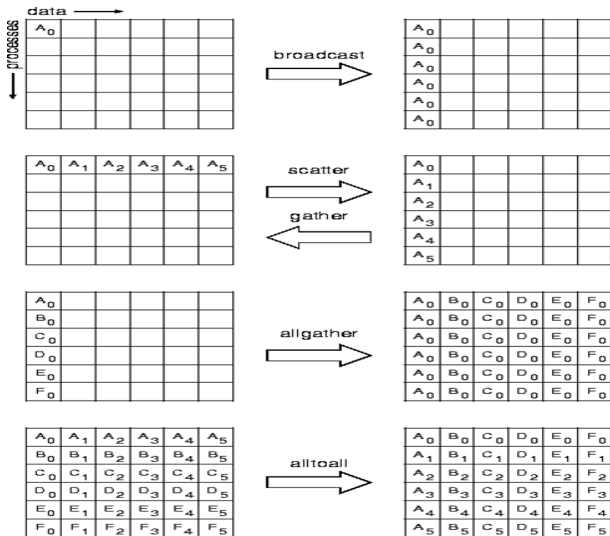
```
18     status=MPI.COMM_WORLD.Recv(revbufer, 0, unitSize, MPI.INT,  
    emisor, tag);  
19     System.out.println("Receptor ha recibido el dato:  
    "+revbufer[0]);  
20     System.out.println("Se han recibido  
    "+status.Get_count(MPI.INT)+" datos");  
21 }  
22  
23 MPI.Finalize();  
24  
25 }  
26 }
```

- ▶ Las operaciones Send y Recv descritas son bloqueantes
- ▶ El proceso que las usa permanece bloqueado hasta que la operación que invocan está *completada*
- ▶ Esto se produce cuando los procesos que envían y reciben sincronizan entre sí en el mensaje

- ▶ Afectan a todos los procesos de un comunicador concreto
- ▶ Todos los procesos del comunicador las deben llamar con parámetros iguales
- ▶ Están definidas en la clase Intracomm

OPERACIÓN	SEMÁNTICA
MPI_Barrier	Sincroniza todos los procesos del comunicador
MPI_Broadcast	Envía un dato desde un proceso a los demás
MPI_Gather	Recolecta datos de todos los procesos
MPI_Scatter	Reparte datos de un proceso a los demás
MPI_Reduce	Realiza operaciones sobre datos distribuidos
MPI_Scan	Operación de reducción de prefijo

# Operaciones de Comunicación Colectiva I



## Código 8: codigo\_t6/difusion.java





```
1  import mpi.*;
2  import java.util.*;
3  public class difusion {
4
5  public static void main(String args[]) throws Exception {
6      MPI.Init(args);
7      int rank = MPI.COMM_WORLD.Rank();
8      int size = MPI.COMM_WORLD.Size();
9      int emisor = 0;
10     int [] data = new int[10];
11
12
13     if(rank==emisor){
14         Random input = new Random();
15         System.out.println("Generando enteros...");
16         for(int i=0; i<10; i++)
17             data[i]=input.nextInt();
18     }
```

# Uso de Broadcast II

```
19 //El metodo Bcast debe llamarse desde el emisor y desde los
    receptores
20 MPI.COMM_WORLD.Bcast(data, 0, 10, MPI.INT, 0);
21
22 if(rank!=emisor){
23     System.out.println("vector recibido...");
24     for(int i=0; i<10; i++)System.out.print(data[i]+" ");
25 }
26 MPI.Finalize();
27 }
28 }
```



# Referencias

-  Carpenter, B. y Fox, G.  
*MPIJava 1.2: API Specification*  
[https://www.researchgate.net/publication/2821816\\_mpiJava\\_12\\_API\\_specification](https://www.researchgate.net/publication/2821816_mpiJava_12_API_specification), 2000
-  Capel, M. y Rodríguez, S.  
*Sistemas Concurrentes y Distribuidos, Capítulo 3*  
Copicentro (Granada), 2012
-  MPJ-Express Project  
<http://mpj-express.org/docs/javadocs/index.html>, 2019
-  Palma, J. et al.  
*Programación Concurrente, Capítulo 8*  
2004

