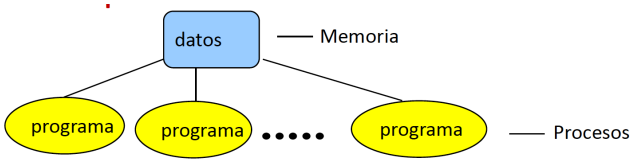
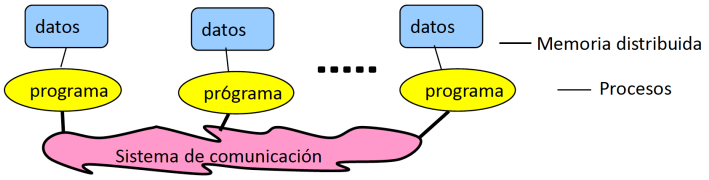
PASO DE MENSAJES

La comunicación entre procesos se puede hacer mediante memoria compartida (aparece el problema de la exclusión mutua, monitores, semáforos); o usando paso de mensajes entre los procesos (donde no comparten ninguna variable y se intercambian información, realizando así los cómputos).

Son útiles cuando abordamos aplicaciones donde no hay memoria común: entornos distribuidos como los basados en la red (bancos, sistema de ventas de entradas, servicios web, …). Pero también como decisión de diseño, ya que se evitan las condiciones de carrera e implementar la exclusión mutua.

Memoria compartida	**Paso de mensajes**
	

Cada proceso interactúa con otros sin compartir memoria: maneja su propio espacio de variables locales y la comunicación con otros procesos es mediante el envío de copias de sus datos locales usando un sistema de comunicaciones. De esta forma se eliminan las situaciones de carrera, pero obliga a definir mecanismos de paso de mensajes: sistemas de identificación, sincronización, etc.

Estructura de los diseños:
Los diseños de los programas concurrentes distribuidos con paso de mensajes suelen ser algo diferentes a los de memoria compartida. Puede (y en muchos casos es imprescindible) añadir procesos al diseño que actúan de "intermediarios" entre los procesos, para asegurar una comunicación y sincronización correctas.

Los diseños más generales se llaman "Arquitecturas" del sistema distribuido.

- Arquitecturas Cliente/Servidor: los procesos clientes solicitan servicios de acceso a datos o realizar operaciones (por ejemplo, en otro computador); mientras que los procesos servidores (normalmente uno) administran recursos.
- Arquitecturas igual a igual (peer-to-peer): no hay clientes ni servidores fijos, los procesos pueden comportarse como unos u otros y la comunicación es directa.
- Otras arquitecturas, como mallas/grids, middleware, etc.

La más usual (y a la que convertiremos nuestros CTADs) es la de cliente/servidor, donde un proceso adicional ayuda a la interacción entre los otros procesos.

Identificación de procesos:
Se puede realizar de forma directa, usando el identificador del proceso para enviar (`send(pid, mensaje); mensaje = receive();`); o de forma indirecta usando el nombre de un canal (`Channel c; c.write(mensaje); mensaje = c.read();`).

En programas reales puede haber cientos de procesos. Identificarlos por su nombre (o id) es muy complejo y da lugar a errores. Los canales permiten desvincular el proceso del medio por el que se comunican, pudiendo recibir por el mismo canal varios procesos.

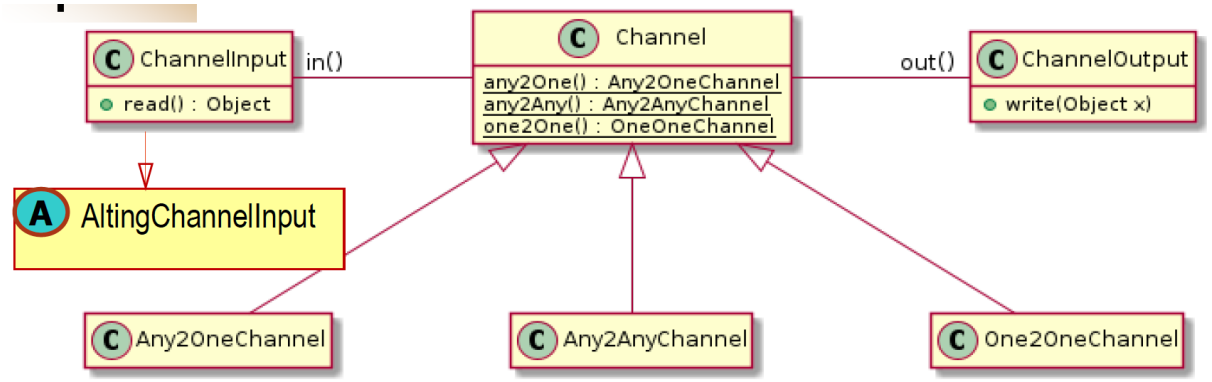
Canales:
La comunicación se puede hacer de manera

- ASÍNCRONA: los envíos se producen sin esperar a que se reciban;
  y el receptor lo recibirá cuando pueda. Ejemplos: Erlang, TCP → BUZONES
- SÍNCRONA: el emisor y el receptor intercambian datos a la vez.
  El que llegue antes espera. Ejemplo: ADA, Occam, Java con CSP (JCSP) → CANALES

En los buzones (asíncronos), los envíos se almacenan en un buffer esperando a ser recibidos. Las recepciones también pueden esperar si no se ha producido el envío. Son más propensos a errores: los buffer se pueden llenar, no hay constancia de la recepción real de los mensajes (aunque se puede programar), la programación puede ser más difícil, pueden ser más ineficientes, …). Pero también es la más cercana a los sistemas físicos y la propia red y para ciertas aplicaciones (telefonía) son muy útiles. Se pueden simular con comunicación síncrona y viceversa.

¿A cuántos procesos da servicio un canal?

- **1 a 1:** Solo un proceso comunica por un canal y solo uno recibe por ese canal → `One2OneChannel`
- **N a 1:** Varios procesos envían por el mismo canal pero solo uno puede recibir → `Any2OneChannel`
- **N a N:** Todos los procesos pueden enviar, todos pueden recibir → `Any2AnyChannel`



Los canales se descomponen en uno de entrada (`canal.in()`) y otro de salida (`canal.out()`).
• Los canales no tienen tipo (pasan un `Object`). Hay que hacer un cambio de tipo (*casting*) al recibir.
• Podemos entender la clase abstracta `AltingChannelInput` como sinónimo de `ChannelInput`.

```
//Biblioteca de canales: JCSP
import org.jcsp.lang.*;

Any2OneChannel canal = Chanel.any2one();

//Preparación para envío
ChannelOutput salida = canal.out();
//Send
salida.write(msg);

//Preparación para recepción
ChannelInputentrada = canal.in();
//Receive
Tipo msg = (Tipo) entrada.read();

//Declaración de procesos – Heredan de CSProcess
classProceso implementsCSProcess{
  ... // Estado, incluye los canales de comunicación
  public Proceso (...ars...){...} //+ un canal
  public void run () { ... }
}

public static void main(String[] args) {
  // Creación de canales y procesos
  Any2OneChannel c = Channel.any2one();
  // Creaciónde procesos
  CSProcess pi = new Proceso(..., c.out(), c.in());
  // Lanzamiento procesos: Todos juntos con un array
  new Parallel(new CSProcess[]{p1, p2,..., pn}).run();
  // Sin join
}
```

Ejemplo Incrementador/Decrementador:

```
class Inc implements CSProcess{
  private ChannelOutput co;
  public Inc(ChannelOutputco){
    this.co = co;
  }
  public void run() {
    for (int i= 0; i< N; i++){
      System.out.println("Enviando +1");
      co.write(1);
    }
  }
}
```

```
class Dec implements CSProcess{
  private ChannelOutput co;
  public Dec(ChannelOutput co){
    this.co = co;
  }
  public void run() {
    for (int i= 0; i< N; i++){
      System.out.println("Enviando-1");
      co.write(-1);
    }
  }
}
```

```
class Contador implements CSProcess{
  private ChannelInput canal;
  public Contador(ChannelInput cn){
    canal = cn;
  }
  public void run() {
    int n = 0;
    while (true) {
      int x = (int) canal.read();
      n = n + x;
      System.out.println(n);
    }
  }
}
```

```
public static void main(String[] args) {
  Any2OneChannel c = Channel.any2one();
  CSProcess inc = new Inc(c.out());
  CSProcess dec = new Dec(c.out());
  CSProcess cont= new Contador(c.in());
  new Parallel(new CSProcess[]
              {inc,dec,cont}).run();
}
```

Ejemplo Almacén1 - Buffer1

Varios procesos producen un dato que guardan en una variable. Varios procesos miran si hay un dato y lo consumen. Si no hay datos, los consumidores esperan. Si hay un dato almacenado, los productores esperan.

Los procesos podrían comunicarse directamente. En el diseño del grafo de procesos es fundamental identificar los canales de comunicación. En este caso, es imprescindible un único canal para que los productores puedan comunicarse con todos los consumidores. Además debe ser un canal N:N (ya que hay varios productores y consumidores). Según el diseño, `Productor` usa `canal.out();` y `Consumidor` usa `canal.in();`

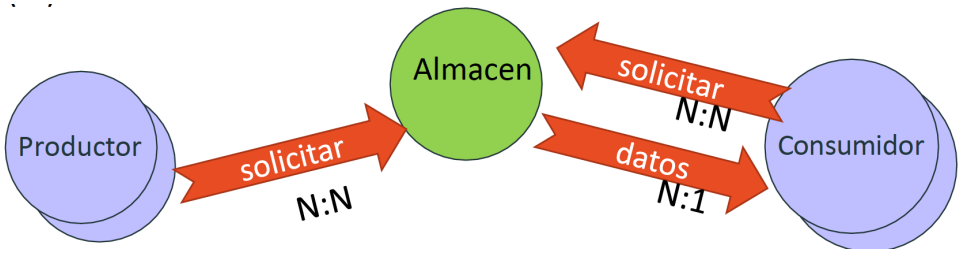
```
class Productor implements CSProcess{
  private ChannelOutput c_co;
  //Envíos
  public Productor(ChannelOutput co){
    c_co = co;
  }
  public void run(){
    for (int i = 0;i<100;i++){
      << Producir d>>;
      c_co.write(d);
    }
  }
}
```

```
class Consumidor implements CSProcess{
  private ChannelInput c_ci;
  //Recepciones
  public Consumidor(ChannelInput ci){
    c_ci = ci;
  }
  public void run() {
    for (int i = 0;i<100;i++){
      d = (Dato) c_ci.read();
      <<Consumir d>>;
    }
  }
}
```

```
public static void main(String[] args) {
  Any2AnyChannel canal = Channel.any2any(); //Declaración canal
  CSProcess prod = new Productor(canal.out()); // Parte envíos
  CSProcess cons = new Consumidor(canal.in()); // Parte recepción
  new Parallel(new CSProcess[]{prod,cons}).run();
  // Podrían ser varios. Si no, el canal es One2One
}
```

En este caso, los procesos podrían comunicarse directamente, pero supone que los productores no pueden ir a su ritmo (cuando hicieran el `p_co.write` se bloquearían si todavía no hay consumidor). Modificamos el diseño para usar cliente/servidor y añadir un proceso que guarda el valor del Almacén.

Diseño de los canales:
– Un canal `solicitar` para comunicar los `productores` con el Almacen (es N:N). Cuando el productor tiene un dato lo envía al Almacen. Si no tiene un dato lo recibe directamente. Cuando tiene uno almacenado no recibe (y el productor espera).
– Un canal `datos` para comunicar a los `consumidores` con el Almacen (es N:1). Pero el Almacen no sabe cuándo el `consumidor` precisa un dato. Si directamente se lo envía por datos sin necesitarlo el Almacen se bloqueará innecesariamente. **Solución:** El consumidor le dice al Almacen que necesita un dato (re)utilizando el canal `solicitar`



Según el diseño, se usan así los canales:

- Productor usa solicitar.out()
- Consumidor usa solicitar.out() y datos.in()
- Almacen usa solicitar.in() y datos.out()

```
class Productor implements CSProcess{
  private ChannelOutput c_co;
  //Envíos
  public Productor(ChannelOutput co){
    c_co = co;
  }
  public void run(){
    for (int i = 0;i<100;i++){
      << Producir d>>;
      c_co.write(d);
    }
  }
}
```

```
class Consumidor implements CSProcess{
  //Recepciones
  private ChannelInput c_ci;
  //Peticiones
  private ChannelOutput c_pet;
  public Consumidor(ChannelInput ci,
      ChannelOutput co){
    c_ci = ci;
    c_pet = co;
  }
  public void run() {
    for (int i = 0;i<100;i++){
      c_pet.write(true);
      d = (Dato) c_ci.read();
      <<Consumir d>>;
    }
  }
}
```

```
class Almacen implements CSProcess {
  private ChannelInput alm_sol;
  private ChannelOutput alm_co;
  private Dato valor;
  public Almacen (ChannelInput ci, ChannelOutput co){
    alm_sol = ci;
    alm_co = co;
    valor = null;
  }
  public void run() {
    Dato d; boolean consumidorEsperando = false;
    while (true) {
      Object msg = alm_sol().read();
      if (msg instanceof Dato) { // Mensaje del productor
        valor = (Dato) msg;
        if (consumidorEsperando) { // Consumir en espera. Se manda
          alm_co.write(valor);
          consumidorEsperando = false;
        } // En el else se cumple que el mensaje es del consumidor
      } else if (valor == null) // No hay dato
         consumidorEsperando = true;
      } else { // Se envía
         alm_co.write(valor);
         valor = null;
      }
    }
  }
}

public static void main(String[] args) {
  Any2AnyChannel solicitar = Channel.any2any();
  Any2OneChannel datos = Channel.any2one();
  CSProcess prod1 = new Productor(solicitar.out());
  CSProcess prod2 = new Productor(solicitar.out());
  CSProcess cons1 = new Consumidor(datos.in(), solicitar.out());
  CSProcess cons2 = new Consumidor(datos.in(), solicitar.out());
  CSProcess alm = new Almacen(solicitor.in(), datos.out());
  new Parallel(new CSProcess[] {prod1, prod2, cons1, cons2, alm}).run();
}
```

Select

Los procesos deben poder recibir mensajes de diferentes remitentes. Pero hay que recordar que las recepciones son bloqueantes. ¿Cómo podemos recibir de varios canales a la vez? El bloqueo de recepciones no permite que el proceso de recepción reciba ningún mensaje de ningún proceso que no sea el que ha especificado.

La única posibilidad es que todos envíen por el mismo canal, como hemos hecho en el Almacén. Pero resulta difícil saber de qué proceso viene el mensaje. Además, podemos necesitar recibir los mensajes dependiendo de alguna condición adicional (sincronización condicionada). Se necesita una nueva construcción del lenguaje para trabajar con esta idea, las guardas y la recepción alternativa condicional:
– Una guarda es una condición booleana condi + una posible recepción por un canal ci.
– Un proceso puede atender a varias guardas a la vez.
– Cuando una de ellas se cumple (cond cierta, un mensaje espera a ser recibido por ci), se ejecuta la recepción y el bloque de instrucciones asociado.

La sintaxis CSP (diseños):

– Guarda: `ci?xi ∧ condi→Ii`

– Select:

```
c1?x1 ∧ cond1→I1
c2?x2 ∧ cond2→I2
...
cn?xn ∧ condn→In
```

c_i	Canal
$c_i?x_i$	Recepción de x_i (variable) por el canal c_i
$cond_i$	Condición
I_i	Instrucciones si se selecciona la guarda

Semántica de la select:
– Una guarda es seleccionable si condi se evalúa a cierto y ya se ha hecho un envío (write) al canal ci.out()y se puede recibir xi(xi= ci.in().read())
– Si ninguna guarda es seleccionable, se espera a que haya al menos una.
– Si hay varias guardas seleccionables, se elige una de forma no determinista.
– No determinista ⇒ Se puede seleccionar cualquiera, una guarda seleccionable se selecciona en algún momento.

Select en JCSP:

Las guardas se almacenan en dos vectores: uno para las condiciones y otro para los canales de entrada.

```
AltingChannelInput[n-1] entradas ={c1,c2,… cn};
boolean[n-1] sincCond;
Alternative servicios =new Alternative(entradas);
```

`AltingChannelInput` → Almacena un canal de entrada ≡ `ChannelInput`
`Alternative` → Estructura que almacena los canales de las guardas

Las guardas y los canales se pueden almacenar de uno en uno. Suele ser habitual enumerar las guardas y a cada número ponerle un nombre legible (COND1, …, CONDN) que identifique la guarda (`final int COND1= 0; ...; final int CONDN= N-1;`).

La construcción select en JCSP usa el switch de Java:

```
sincCond[COND1] = cond1; ...; sincCond[CONDN] = condn;
switch (servicios.fairSelect(sincCond)) {
case COND1: x1=c1.read();I1; break;
case COND2: x2=c2.read();I2; break;
...
case CONDN: xN=cN.read();IN; break;
}
```

`servicios.fairSelect(sincCond)`

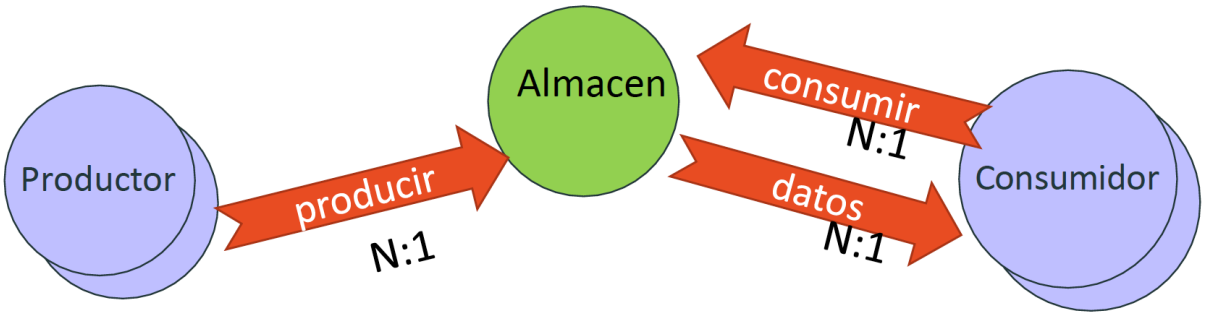
• evalúa las guardas (condi, ci) = evaluar la condición y comprobar si ya se ha hecho un write por el canal.
• condi está en sincCond y ci en servicios
• Si la condición es true y si ya se ha hecho el write, la guarda es elegible.
• La instrucción selecciona una de las guardas i elegibles de forma no determinista.
• Devuelve el índice i (int).
• Si ninguna guarda es seleccionable, espera a que lo sea.

En casos complejos (algún proceso que tiene que enviar por un canal o el propio que lo ejecuta se paran abruptamente) devuelve una excepción, que no vamos a capturar para que el código sea más legible.

```
try {
  choice = servicios.fairSelect(sincCond);
} catch (ProcessInterruptedException e){}
```

Almacen1 con select:

Añadimos un canal con el que los consumidores indican al Almacen que quieren un dato (es N:1). Los procesos Productor y Consumidor se programan igual, lo único que cambia es la declaración de canales en el main y la llamada al constructor de Consumidor. Los canales son ahora N:1, que son más ligeros, y renombramos.



Según el diseño, se usan así los canales:
– Productor usa producir.out()
– Consumidor usa datos.out() y consumir.in()
– Almacen usa producir.in(), consumir.in() y datos.out()

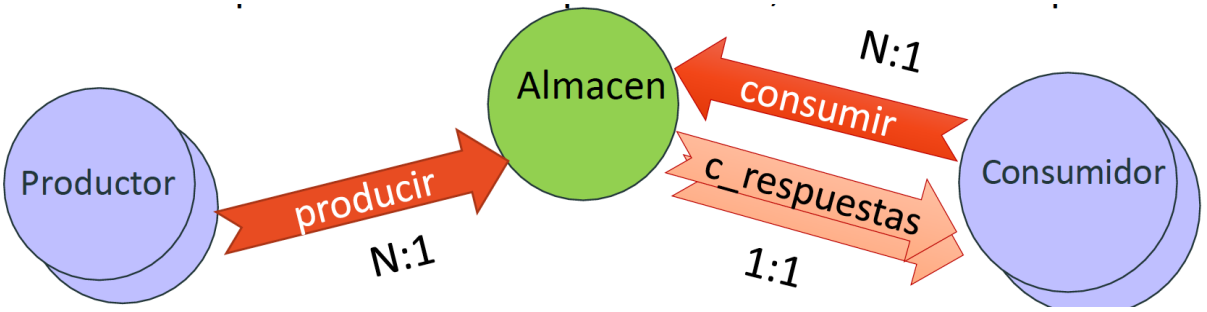
```
public static void main(String[] args) {
  Any2OneChannel producir = Channel.any2one();
  Any2OneChannel datos = Channel.any2one();
  Any2OneChannel consumir = Channel.any2one();
  CSProcess prod1 = new Productor(producir.out());
  CSProcess prod2 = new Productor(producir.out());
  CSProcess cons1 = new Consumidor(datos.in(), consumir.out());
  CSProcess cons2 = new Consumidor(datos.in(), consumir.out());
  CSProcess alm = new Almacen(producer.in(), consumer.in(), datos.out());
  new Parallel(new CSProcess[] {prod1, prod2, cons1, cons2, alm}).run();
}

class Almacen implements CSProcess {
  private AltingChannelInput c_producir, c_consumir;
  private ChannelOutput c_datos;
  private boolean haydato;
  private Dato valor;
  public Almacen (AltingChannelInput prd, AltingChannelInput cons,
                  AltingChannelOutput dt) {
    c_producir = prd; c_consumir = cons; c_datos = dt;
    valor = null; haydato = false;
  }
```

Diseñamos las guardas:
– Almacen puede recibir de c_producir(). La recepción se podrá realizar si no hay ningún dato almacenado (esto es, haydato== false). **Guarda**: `c_producir? x ∧ ¬ haydato → valor = x; haydato = true`
– Almacen puede recibir de c_consumir (), solicitando el dato a consumir que solo tiene sentido si lo hay, (es decir haydato == false). **Guarda**: `c_consumir? b ∧ haydato → c_datos.write(valor); haydato = false`

```
public void run() {
  final int GUARDAR = 0; final int EXTRAER = 1:
  Object msg;
  boolean[2] sincCond;
  AltingChannelInput[2] entradas = {c_producir, c_consumir};
  Alternative servicios = new Alternative(entradas);
  while (true) {
    sincCond[GUARDAR] = !haydato;
    sincCond[EXTRAER] = haydato;
    switch (servicios.fairSelect(sincCond)) {
    case GUARDAR: msg = c_producir.read();
      valor = (Dato) msg;
      haydato = true;
      break;
    case EXTRAER: msg = c_consumir.read();
      c_datos.write (valor);
      haydato = false;
      break;
    }
  }
}
```

Almacen1 con canal de retorno:

El usar expresamente los canales de respuesta puede provocar algunos problemas e implica conocer su nombre y características por el Almacén. Una solución más general es que el consumidor cree un canal para la respuesta y le informa al almacén de cuál es para que le conteste por ese canal. Evita confundir los canales.




```
class Consumidor implements CSProcess{
  private ChannelOutput c_consumir; //Avisos al almacén
  // Canal por el que queremos que nos respondan:
  private One2OneChannel c_respuesta;
  public Consumidor (ChannelOutput co){
    c_consumir = co;
    c_respuesta = Channel.one2one();
  }
  public void run() {
    for (int i = 0;i<100;i++){
      c_consumir.write(c_respuesta.out());
      d = (Dato) c_respuesta.in().read();
      <<Consumir d>>;
    }
  }
}

class Productor implements CSProcess{
  private ChannelOutput p_co; //Envíos
  // Canal por el que queremos que nos confirmen
  private One2OneChannel c_ack;
  public Productor (ChannelOutput co){
    p_co = co;
    c_ack = Channel.one2one();
  }
  public void run(){
    for (int i = 0;i<100;i++){
      << Producir d>>;
      Pair<Dato,ChannelOutput> envio = new Pair<Dato,ChannelOutput> (d,c_ack.out()>;
      p_co.write(envio);
      c_ack.in().read(); // No nos importa lo que transmita
    }
  }
}
```

```
class Almacen implements CSProcess {
  private AltingChannelInput c_producir, c_consumir;
  private boolean haydato;
  private Dato valor;
  public Almacen (AltingChannelInput prd, AltingChannelInput cons) {
    a_producir = prd;
    c_consumir = cons;
    haydato = false;
    valor = null;
  }
  public void run() {
    final int GUARDAR = 0; final int EXTRAER = 1:
    Object msg;
    ChannelOutputresp; boolean[2] sincCond;
    AltingChannelInput[] entradas = {c_producir,c_consumir};
    Alternative servicios = new Alternative(entradas);
    while (true) {
      sincCond[GUARDAR] = !haydato;
      sincCond[EXTRAER] = haydato;
      switch (servicios.fairSelect(sincCond)) {
      case GUARDAR: msg = c_producir.read();
        valor = (Pair<Dato, ChannelOutput>) msg;
        haydato = true;
        msg.value().write(null); //2° del par es canal
        break;
      case EXTRAER: msg = c_consumir.read();
        resp = (ChannelOutput) msg;
        resp.write(valor);
        haydato = false;
        break;
      }
    }
  }
}

public void main(String[] args) {
  Any2OneChannel producir = Channel.any2one();
  Any2OneChannel consumir = Channel.any2one();
  CSProcess prod1 = new Productor(producir.out());
  CSProcess prod2 = new Productor(producir.out());
  CSProcess cons1 = new Consumidor(consumir.out());
  CSProcess cons2 = new Consumidor(consumir.out());
  CSProcess alm = new Almacen(producer.in(), consumer.in());
  new Parallel(new CSProcess[] {prod1, prod2, cons1, cons2, alm}).run();
}
```