Recursos compartidos con paso de mensajes (EN CONSTRUCCIÓN: NO IMPRIMIR)

Julio Mariño

Universidad Politécnica de Madrid
Babel research group
http://babel.upm.es/teaching/concurrencia

Concurrencia, mayo 2016

- Hasta ahora hemos programado sistemas concurrentes en los que la vía principal de interacción entre procesos era la lectura y escritura de variables compartidas. De igual modo, el primer problema que tuvimos que resolver fue el de evitar *situaciones de carrera* en el acceso a dichas variables.
- Sin embargo, cada vez hay más sistemas concurrentes *distribuidos*, donde cada proceso interactúa con otros sin compartir memoria. Es decir, cada proceso maneja su propio espacio de variables locales y la comunicación con otros procesos es mediante el envío de copias de sus datos locales usando una red de comunicaciones.
- Esto elimina el problema de las situaciones de carrera, pero obliga a definir mecanismos de *paso de mensajes* que resuelvan de manera cómoda el problema de la comunicación: sistemas de direcciones, sincronización, etc.
- Este tema pretende dar unas nociones básicas sobre la programación de sistemas concurrentes distribuidos dentro del marco CSP (*Communicating Sequential Processes*), usando como vehículo la librería Java JCSP.

sistemas distribuidos y paso de mensajes

- El elemento común a cualquier sistema distribuido es la presencia de operaciones de *envío* y *recepción* de mensajes a través de algún tipo de red de comunicación, ya sea local o extensa.
- A pesar de que existen lenguajes de programación basados en paso de mensajes desde hace varias décadas, no existe en la actualidad un lenguaje o modelo de programación que se haya impuesto claramente a los demás. Algunos de los lenguajes vivos que usan extensamente paso de mensajes son Ada (muy usado en aviónica, software militar y aplicaciones críticas), Erlang, Go, etc.
- Mencionamos a continuación diferentes características que permiten clasificar los sistemas basados en paso de mensajes:
- según el direccionamiento: En el direccionamiento *basado en procesos* los mensajes se direccionan usando algún tipo de identificador del proceso que los ha de recibir. Esto es poco flexible en sistemas donde se crean y destruyen procesos dinámicamente.
  En el direccionamiento *basado en canales*, los mensajes se envían o reciben a través de entidades intermedias que deben ser visibles por ambas partes. Por ejemplo, si un proceso a quiere enviar una copia de su variable local x a un proceso b, que almacenaría ese valor en una variable y, pueden acordar usar un canal intermedio c de manera que a ejecutaría:
  `c.send(x);`
  y el proceso b ejecutaría:
  `y = c.receive();`
  La mayor parte de los lenguajes modernos basados en paso de mensajes usan canales.

POLITÉCNICA

sistemas distribuidos y paso de mensajes (ii)

- según la sincronización de los canales: Recibir un dato de un canal es una operación intrínsecamente bloqueante: el proceso receptor debe esperar a que haya un dato en el canal para proseguir ejecución. Sin embargo, el envío puede o no ser bloqueante. En un canal *asíncrono* el envío *no* es bloqueante: el proceso que ejecuta send deja el mensaje en el canal y prosigue ejecución. En un canal *síncrono*, la operación de envío bloquea hasta que hay un proceso intentando recibir por el mismo canal. Nosotros vamos a trabajar con canales *síncronos*.
- según la aridad de los canales: En un canal *1:1* solo se permite enviar a un proceso y solo permite recibir a un proceso. En un canal *n:1* se permite que varios procesos envíen por un él pero solo un proceso tiene derecho a leer del canal. Estos son los dos tipos de canales que vamos a usar. Existen canales *n:m* (a veces llamados *buzones*), pero no los usaremos.

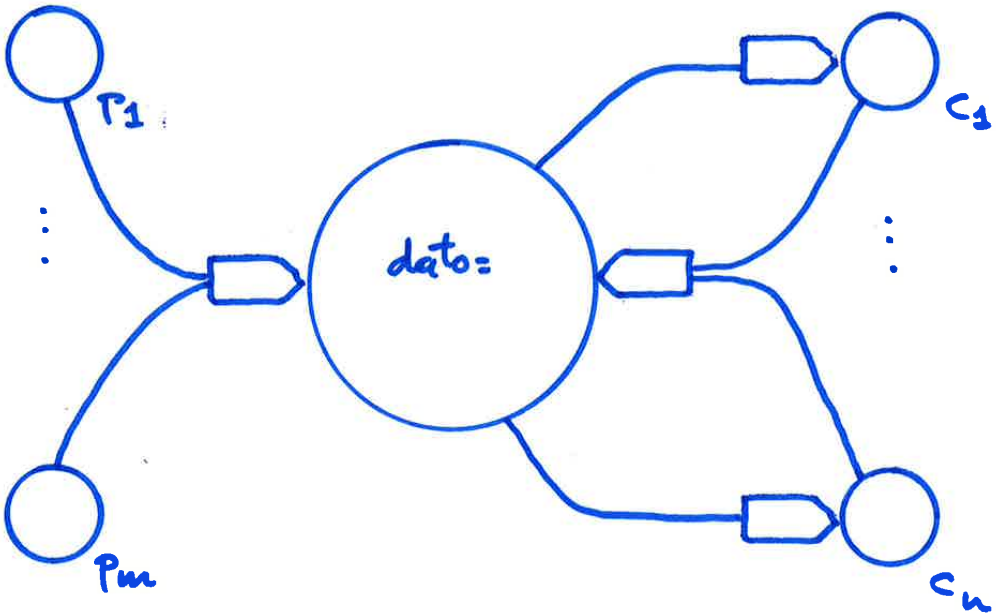
POLITÉCNICA

recursos compartidos y esquema *cliente/servidor*

- En un sistema distribuido ya no podemos hacer que los procesos interactúen mediante variables compartidas.
- La idea básica para implementar recursos será basarse en un esquema cliente/servidor:
  - El estado del recurso residirá en el espacio de memoria de un proceso que llamaremos *servidor del recurso*, y
  - las operaciones del recurso se realizarán mediante *peticiones* de los procesos implicados al servidor.
  - Cuando una operación del recurso devuelva un valor, el servidor tendrá que enviar un mensaje por un canal de respuesta a la petición recibida.
- A continuación veremos un ejemplo básico (almacén de un dato) realizado con paso de mensajes síncrono. Las peticiones de los clientes al servidor van por canales n:1 y la respuesta del servidor a los consumidores por canales 1:1.

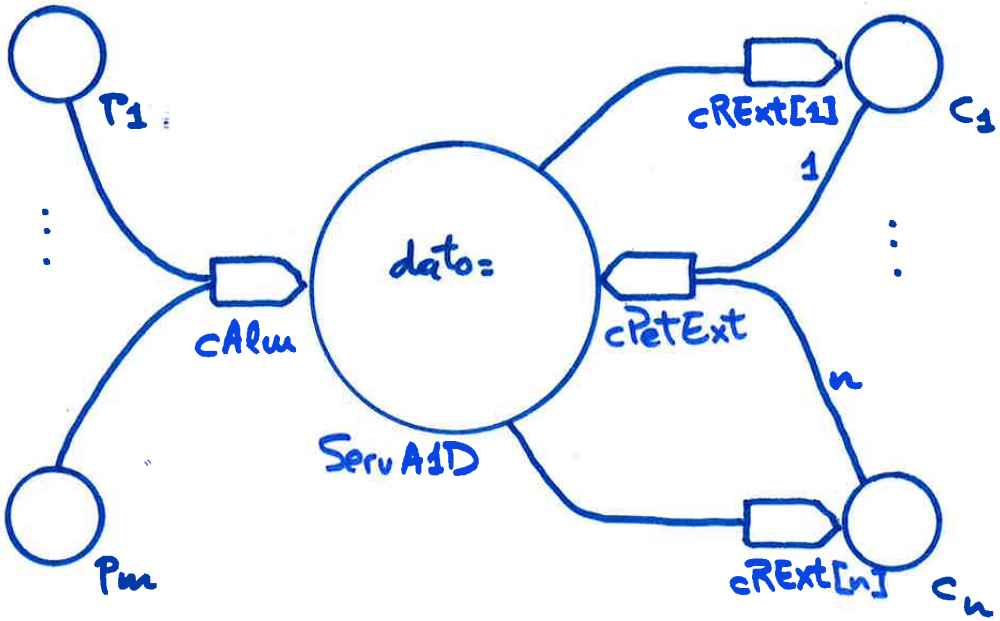
POLITÉCNICA

almacén de un dato con cliente/servidor



CA

almacén de un dato con cliente/servidor



CA

almacén de un dato con cliente/servidor

solución con un vector de canales respuesta

- Asumimos que los consumidores vienen dados de manera estática por un vector de threads, por lo que la respuesta puede enviarse igualmente usando un vector de canales 1:1. En este caso la petición de los consumidores consiste en el índice de ambos vectores.

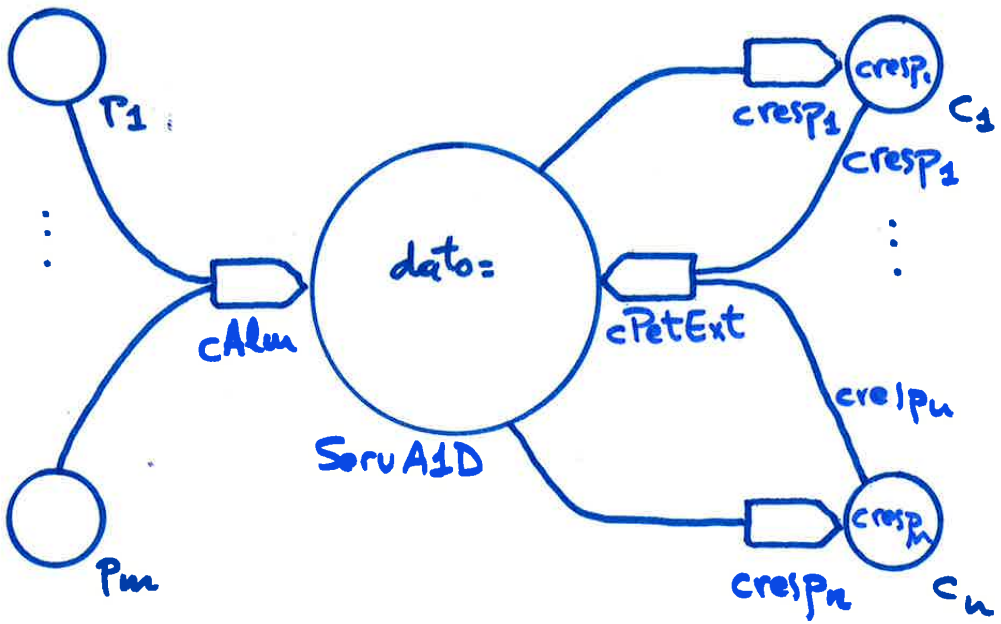
productor(i)	servidor	consumidor(j)

`while(true){`
`  d = producir();`
`  cAlm.send(d);`
`}`
	`while(true){`
	`  dato = cAlm.receive();`
	`  ncons=cPetExt.receive();`
	`  cRExt[ncons].send(dato);`
	`}`
		`while(true){`
		`  cPetExt.send(j);`
		`  d = cRExt[j].`
		`  receive();`
		`  consumir(d);`
		`}`

POLITÉCNICA

almacén de un dato con cliente/servidor

solución con canales respuesta como petición



almacén de un dato con cliente/servidor

solución con canales respuesta como petición

- Esta variación no requiere que el número de consumidores sea conocido de antemano. La petición de los consumidores consiste ahora en un canal por el cual se recibirá posteriormente la respuesta.

productor(i)	servidor	consumidor(j)

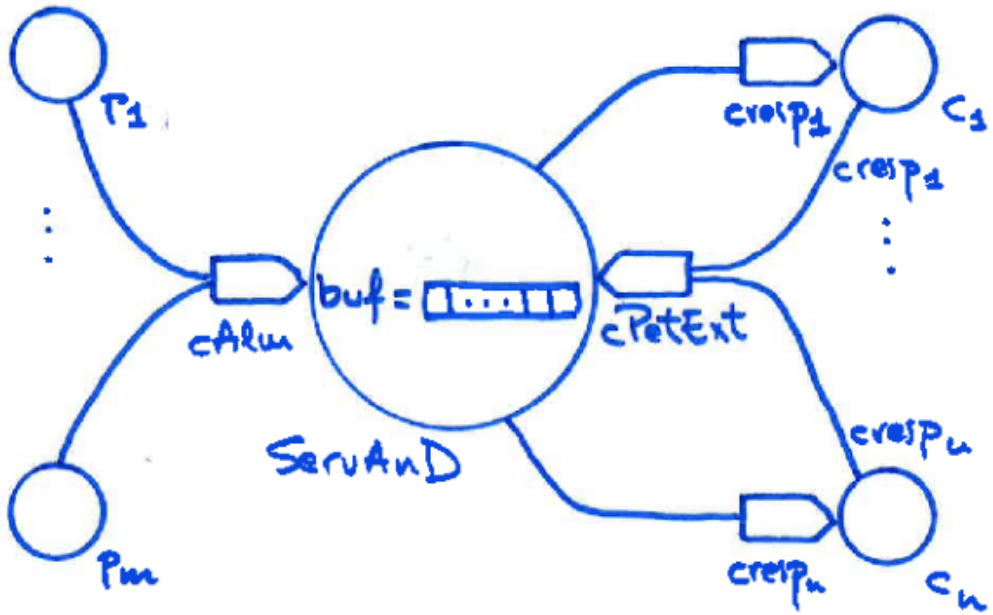
while(*true*){
  d = producir();
  cAlm.send(d);
}
	cresp = ...
	while(*true*){
	  dato = cAlm.receive();
	  cresp=cPetExt.receive();
	  cresp.send(dato);
	}
		cresp = ...
		while(*true*){
		  cPetExt.send(cresp);
		  d = cresp.receive();
		  consumir(d);
		}

POLITÉCNICA

almacén de *n* datos

algo no funciona...

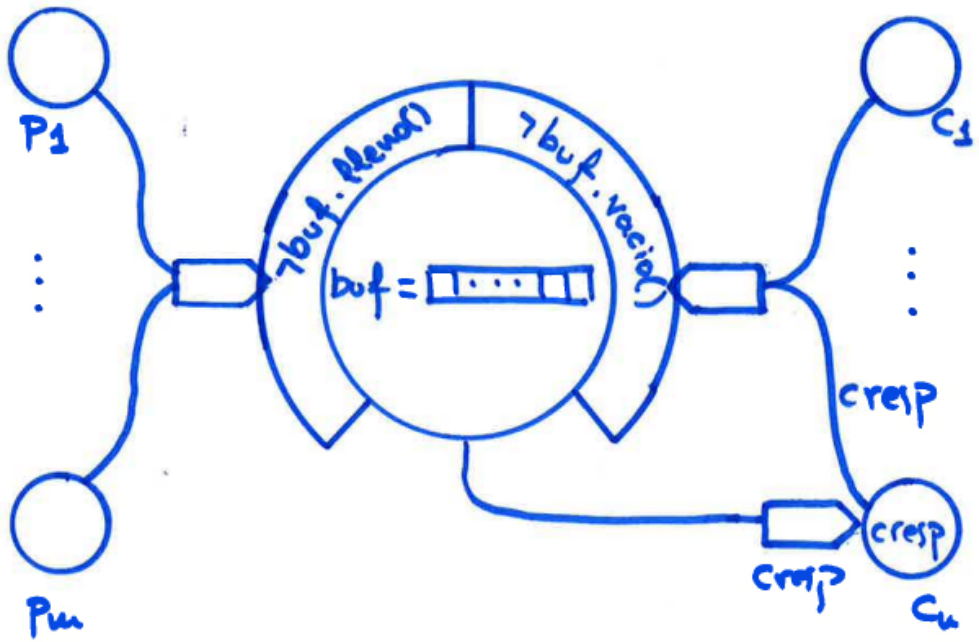
- Para adaptar la solución anterior, el servidor debería estar atendiendo *simultáneamente* a ambos canales, ya que las peticiones de almacenar y extraer no son excluyentes.



almacén de *n* datos

solución con *recepción alternativa condicional*

- Recepción alternativa condicional:
  - Permite al receptor atender varios canales simultáneamente (alternativa), y
  - desactivar previamente alguno de esos canales si no se cumple alguna condición.
- En una recepción alternativa el receptor bloquea hasta que hay mensaje en *alguno* de los canales *activos*.
- En este caso se usarían como condiciones para la activación de los canales las CPREs de almacenar y extraer.



POLITÉCNICA

almacén de *n* datos

solución con *recepción alternativa condicional*

productor(i)	servidor	consumidor(j)

`while(true){`
`  d = producir();`
`  cAlm.send(d);`
`}`
	`cresp = ...`
	`while(true){`
	`  select (*)`
	`  when(!buf.lleno()){`
	`  buffer.enqueue(cAlm.`
	`  receive());`
	`  }`
	`  when(!buf.vacio()){`
	`  cresp=cPetExt.receive();`
	`  cresp.send(buffer.front())`
	`  ;`
	`  buffer.dequeue();`
	`  }`
	`  end select;`
	`}`
		`cresp = ...`
		`while(true){`
		`  cPetExt.send(cresp)`
		`  ;`
		`  d = cresp.receive()`
		`  ;`
		`  consumir(d);`
		`}`

- (*) La construcción *select* mostrada arriba es un pseudocódigo inspirado en la construcción análoga del lenguaje Ada.
- Realizaremos este concepto en Java mediente el uso de las clases definidas en la librería JCSP.

POLITÉCNICA

Communicating Sequential Processes

- Communicating Sequential Processes (CSP) fue propuesto originalmente por Anthony Hoare como un lenguaje sencillo para especificar ejecución concurrente, comunicación y sincronización de procesos distribuidos de una manera independiente de la arquitectura.
- Sus elementos básicos son:
  - comunicación síncrona
  - recepción alternativa condicional
- Una breve introducción a la propuesta original de CSP se encuentra en el artículo de Andrews & Schneider que ya tenéis. Una descripción por el propio Hoare es el artículo [Hoare78]. Un buen texto académico (pero con un lenguaje propio del autor) es [Andrews91, Cap.8]. [Hoare85] es el primer libro dedicado enteramente a CSP.
- Con los años, la propuesta inicial, muy concisa, se ha ido enriqueciendo hasta convertirse en un cálculo de procesos distribuidos soportado por potentes herramientas de análisis, etc. Un texto bastante completo es [Roscoe2005].
- CSP ha influido en el diseño de un buen número de lenguajes de programación, como *occam*, *Ada*, *Strand* o *Erlang* (aunque en este último la comunicación no es síncrona).

POLITÉCNICA

JCSP

una librería Java que implementa CSP

- JCSP es una librería Java que traduce las ideas de CSP a una jerarquía de clases e interfaces Java con sus correspondientes métodos, *sin tocar la estructura del lenguaje*.
- La ventaja de esto es que los programas JCSP compilan con cualquier compilador de Java. La desventaja es que generalmente hay que escribir más que para hacer algo equivalente en lenguajes como Ada o Erlang que soportan las construcciones de manera nativa.
- JCSP es un proyecto académico, no comercial, desarrollado por profesores y alumnos de la Universidad de Kent y se encuentra disponible en `http://www.cs.kent.ac.uk/projects/ofa/jcsp/`.
- Canales
  - clases `Channel`, `One2OneChannel`, `Any2OneChannel`, `ChannelInput`, `ChannelOutput`.
- Recepción alternativa
  - clases `Alternative`, `AltingChannelInput`.
  - interfaz `Guard`.
- Creación de procesos
  - Clases `CSProcess`, `Parallel`, etc.

POLITÉCNICA

- Los canales tienen “dos mitades”: la mitad por la que se envía (de clase `ChannelOutput`), y la mitad por la que se recibe (de clase `ChannelInput`):

```
One2OneChannel c1, c2 = Channel.one2one();
ChannelInput ci = c1.in();
ChannelOutput co = c2.out();
co.write(7);
```

- Los canales no son genéricos: lo que se envía son `Object`, lo que obliga a veces a hacer *castings* explícitos para convertir lo que se recibe por un canal:

```
x = (Integer)ci.read();
```

POLITÉCNICA

- Para atender al mismo tiempo a varios canales, usamos una subclase de `ChannelInput`: `AltingChannelInput`.

```
AltingChannelInput input1 = c1.in();
AltingChannelInput input2 = c2.in();
```

- Construimos una estructura de recepción alternativa pasando un vector de `AltingChannelInput` al constructor de la clase `Alternative`:

```
AltingChannelInput[] inputs = {input1, input2};
Alternative servicios = new Alternative(inputs);
```

- Los métodos `select` y `fairSelect` devuelven el índice del vector por donde llega un mensaje:

```
int seleccion = servicios.fairSelect();
```

- Si se pasa un vector de booleanos, inhabilita los canales con posiciones a falso:

```
boolean sincCond[2] = new boolean[2];
...
int seleccion = servicios.fairselect(sincCond);
```

POLITÉCNICA

almacén de un dato con JCSP

código del servidor

```
import es.upm.babel.cclib.Producto;
import org.jcsp.lang.*;
public class Almacen1DatoJCSP implements CSProcess {
    // Punto de recepcion de los canales para almacenar y extraer
    private ChannelInput petAlmacenar = null;
    private ChannelInput petExtraer = null;

    public Almacen1DatoJCSP (final ChannelInput petAlmacenar,
                             final ChannelInput petExtraer) {
        this.petAlmacenar = petAlmacenar;
        this.petExtraer  = petExtraer;
    }

    public void run() {
        // declaramos aqui el estado del recurso
        Producto item;
        boolean  hay_dato = false; // innecesario
        // para contestar a las peticiones de extraer:
        ChannelOutput resp;
        // el servidor ejecuta un bucle de servicio infinito:
        while (true) {
            // inicialmente, solo podemos almacenar
            item = (Producto) petAlmacenar.read();
            // despues, solo podemos extraer...
            resp = (ChannelOutput) petExtraer.read();
            resp.write(item);
        } // fin bucle servidor
    } // fin servidor
}
```

POLITÉCNICA

almacén de un dato con JCSP

código de los clientes: Productores

```
// Clase de los procesos productores
class ProductorCSP implements CSProcess {
    // por este punto de envio enviaremos los datos
    private ChannelOutput petAlmacenar;

    // Evitando construcciones incorrectas
    private ProductorCSP() {
    }

    public ProductorCSP(ChannelOutput petAlmacenar) {
        this.petAlmacenar = petAlmacenar;
    }

    public void run() {
        while (true) {
            Producto p = Fabrica.producir();
            petAlmacenar.write(p);
        }
    }
}
```

POLITÉCNICA

almacén de un dato con JCSP

código de los clientes: Consumidores

```
// Clase de los procesos consumidores
class ConsumidorCSP implements CSProcess {
    // por este punto de envio pediremos los datos...
    private ChannelOutput petExtraer;
    // enviando el canal por el que queremos que nos respondan:
    private One2OneChannel chResp = Channel.one2one();

    // Evitando construcciones incorrectas
    private ConsumidorCSP() {
    }

    public ConsumidorCSP(ChannelOutput petExtraer) {
        this.petExtraer = petExtraer;
    }

    public void run() {
        while (true) {
            petExtraer.write(chResp.out());
            Producto p = (Producto) chResp.in().read();
            Consumo.consumir(p);
        }
    }
}
```

POLITÉCNICA

almacén de n datos con JCSP

código del servidor

```
public class AlmacenSelectCondicional implements CSProcess {
    private AltingChannelInput petAlmacenar = null;
    private AltingChannelInput petExtraer = null;
    private Queue<Producto> cola = new LinkedList<Producto>();
    final int MAX = 10;
    public AlmacenSelectCondicional (final AltingChannelInput petAlmacenar,
                                     final AltingChannelInput petExtraer) {
        this.petAlmacenar = petAlmacenar;
        this.petExtraer  = petExtraer;
    }
    public void run() {
        final int ALMACENAR = 0;
        final int EXTRAER = 1;
        final AltingChannelInput[] entradas = {petAlmacenar, petExtraer};
        // Recepcion alternativa
        final Alternative servicios =  new Alternative (entradas);
        // Sincronizacion condicional en la select
        final boolean[] sincCond = new boolean[2];
        while (true) {
            ChannelOutput resp;
            Producto item;
            // Preparacion de las precondiciones
            sincCond[ALMACENAR] = cola.size() < MAX;
            sincCond[EXTRAER] = cola.size() > 0;
            switch (servicios.fairSelect(sincCond)) {
            case ALMACENAR:
                item = (Producto) petAlmacenar.read();
                cola.add(item);
                break;
            case EXTRAER:
                resp = (ChannelOutput) petExtraer.read();
                resp.write(cola.peek());
                cola.poll();
                break;
            }
        }
    }
}
```

POLITÉCNICA

traduciendo recursos compartidos a servidores con JCSP

- Cuando las operaciones del recurso tienen CPREs a *Cierto* o bien la CPRE no depende de ningún parámetro de la operación, el esquema que se sigue es similar al visto en el ejemplo anterior (un canal por operación y la CPRE como condición de activación del canal).
- Pero si tenemos CPREs dependientes de parámetros de una operación el servidor no conoce de antemano la condición para activar o desactivar el canal. En este caso, disponemos básicamente de dos estrategias:
  - replicación de canales: Se trata de crear tantos canales como sea necesario de manera que cada uno de ellos pueda ser activado mediante una instancia de la CPRE correspondiente a un conjunto de llamadas. Esta estrategia es análoga a la *indexación de parámetros* en monitores.
  - peticiones aplazadas: Mantenemos un único canal por petición (que siempre está activo). La petición contiene los datos de la llamada a la operación necesarios para evaluar la CPRE y un canal 1:1 (al menos) donde se deja esperando al cliente hasta que la petición puede ser atendida. Esta técnica es análoga a la *indexación de procesos cliente* en monitores. El canal de bloqueo juega el papel que jugaban antes las variables *condition*.

POLITÉCNICA

ejemplo: Alarmas

C-TAD Alarma

OPERACIONES

ACCIÓN Notificar: $\mathbb{Z}[i]$

ACCIÓN Detectar: $\mathbb{Z}[i] \times \mathbb{Z}[i] \times \mathbb{Z}[o]$

SEMÁNTICA

DOMINIO:

TIPO: $Alarma = \mathbb{Z}$

INVARIANTE: $0 \leq \text{self} \wedge \text{self} < 50$

INICIAL: $self = 23$

PRE: $0 \leq t \wedge t < 50$

CPRE: Cierto

Notificar(t)

POST: $self = t$

PRE: $0 \leq min \wedge min \leq max \wedge max < 50$

CPRE: $min \leq self \wedge self \leq max$

Detectar(min,max,t)

POST: $t = self^{pre} \wedge self = self^{pre}$

- Código siguiendo ambas técnicas disponible en la web de la asignatura.

POLITÉCNICA