Metodología para la Programación de Recursos Compartidos en Ada (PARTE II: RENDEZ-VOUS)

Unidad de Programación

LSIIS

Rev: 313, 8 de noviembre de 2005

Índice

1. Introducción

La metodología de desarrollo de programas concurrentes que proponemos en esta asignatura se basa en

1. El análisis del problema como un conjunto de procesos e interacciones entre dichos procesos (recursos).
2. El refinamiento y especificación formal de dichas interacciones usando una notación similar a la de los tipos abstractos de datos (CTADs) que incluye información de sincronización por condición.
3. La traducción de dichos recursos, de una manera sistemática y correcta a un lenguaje de programación. Se entiende por correcta una traducción que asegura las propiedades de seguridad del sistema.
4. El análisis y refinamiento iterativo del código resultante para asegurar las propiedades de vivacidad del sistema.

En la primera parte de este documento (PARTE I: OBJETOS PROTEGIDOS) se detallan las diferentes etapas de la metodología (salvo el análisis) y se aplican a la generación de código Ada basado en objetos protegidos. En esta segunda parte no nos extenderemos en los aspectos comunes de la metodología y nos centraremos en las dos últimas fases, aplicando el método a la generación de código basado en procesos distribuidos y comunicación mediante *rendez-vous*

Los lenguajes para programación concurrente suelen clasificarse en lenguajes basados en memoria compartida y basados en paso de mensajes. Muchos textos usaban métodos de programación completamente diferentes para ambos tipos, cuando no se limitaban a proporcionar recetas ad-hoc a diferentes problemas. Nuestro enfoque parte de un análisis común que no distingue entre diferentes arquitecturas. Los recursos compartidos se convierten en procesos servidor, de acuerdo con una filosofía cliente-servidor, lo que permite una traducción bastante elegante de la especificación del recurso.

Durante unos cuantos cursos se usó una traducción bastante fiel de *CSP* (*Communicating Sequential Processes*) implantada en el lenguaje académico Cc-Módula para enseñar programación concurrente basada en paso de mensajes. Si bien nadie discute la claridad y simplicidad de CSP como ventajas para la enseñanza de la concurrencia, la práctica no ha jugado a su favor, ya que son pocos los lenguajes de programación que han optado por realizarla fielmente.[1]

Al optar por el lenguaje Ada como medio común para la enseñanza de la programación, nos alejamos ligeramente de los mecanismos clásicos – monitores, CSP – al adoptar los propuestos en este lenguaje de programación – *objetos protegidos*, *rendez-vous* – pero los conceptos subyacentes siguen siendo muy similares.

En estas notas se exponen unas recomendaciones metodológicas para el desarrollo de programas concurrentes empleando el mecanismo de *rendez-vous* que Ada proporciona para la comunicación y sincronización de procesos distribuidos. No obstante, el énfasis se hará en la resolución de problemas de concurrencia de forma correcta, más que en profundizar en aspectos particulares del lenguaje Ada. El objetivo es que todas las ideas expuestas a continuación sean aplicables al usar otros lenguajes de programación.

2. Concurrencia basada en paso de mensajes

Los mecanismos lingüísticos que nos facilitan la programación de procesos concurrentes en los lenguajes de programación de alto nivel tienen su origen en problemas que los sistemas operativos tuvieron que resolver hace varias décadas.

[1] El lenguaje de programación *Occam*, diseñado para la programación de la arquitectura paralela *transputer* es la excepción más destacada. Más recientemente, parte de la propuesta occam-CSP ha sido realizada en Java en forma de librerías — *JCSP*, *CCJ*.

Así, los mecanismos basados en memoria compartida tienen como primer objetivo garantizar la exclusión mutua, problema que surge cuando varios procesos tienen que compartir una memoria, periféricos, etc. — multiprogramación. Garantizada la exclusión mutua, se entiende que la comunicación entre procesos se consigue mediante las lecturas/escrituras en variables compartidas.

Con posterioridad a la aparición de los sistemas operativos multiprogramados surge una necesidad diferente: la de conectar varios ordenadores entre sí mediante redes de comunicaciones que permitan la compartición de recursos o el acceso a información distante. Ahora la situación es diferente: no existe el problema del acceso simultáneo a variables compartidas puesto que cada ordenador dispone de su propio espacio de memoria, pero hay que resolver el problema de la comunicación.

Aunque estos dos contextos han dado lugar a los dos paradigmas antes mencionados, hay que aclarar que los lenguajes hacen (bastante) abstracción de la arquitectura que los ha originado, encontrándose tanto lenguajes basados en paso de mensajes ejecutándose en arquitecturas monoprocesador (o multiprocesador de memoria común) como lenguajes basados en memoria compartida cuyos programas pueden ejecutarse igualmente en una arquitectura distribuida.

2.1. Comunicación y sincronización

El quid del paso de mensajes reside en establecer los medios para el intercambio de información (mensajes) entre procesos distribuidos. Llamaremos genéricamente *canales* a esos medios. En los lenguajes basados en paso de mensajes, como veremos a continuación, no sólo son un medio de comunicación entre procesos, sino también de sincronización.

Atendiendo a diferentes decisiones de diseño que se pueden tomar a la hora de definir las características de estos canales podemos llegar a diferentes taxonomías para los lenguajes basados en paso de mensajes.

2.1.1. Asíncrono vs. síncrono

Esta es una diferencia esencial, pues afecta al comportamiento dinámico de un sistema de procesos distribuidos. Un acto de comunicación es *síncrono* cuando requiere la simultaneidad de emisor y receptor. Cuando hablamos cara a cara, por teléfono o por *chat* con un amigo estamos participando en un acto de comunicación síncrono.

Un acto de comunicación *asíncrono* no requiere esa simultaneidad. El emisor envía un mensaje que puede ser recibido *más tarde* o incluso no ser recibido jamás. Cuando enviamos una carta, o correo electrónico, o *sms* a nuestro amigo de antes no lo recibirá en el acto y, de hecho sólo podemos hacer una estimación vaga de cuándo tardará en recibirlo, de si seguirá siendo nuestro amigo cuando lo reciba o si lo llegará a leer alguna vez.

En programación concurrente decimos que hay canales síncronos, donde la operación de enviar un mensaje bloquea al emisor hasta que un receptor decida recibir el mensaje, y canales asíncronos, donde el emisor puede efectuar el envío y seguir ejecutándose independientemente de lo que haga el(los) receptor(es) del mensaje.

Para concretar algo más, supongamos una notación basada en canales explícitos donde los procesos disponen de una operación de envío (*Send*) y de una operación de recepción (*Receive*). Si tenemos dos procesos A y B que se comunican a través del canal síncrono S:

```
principal:  S:Channel;
            ...
        A:  X:T;          B:  Y:T;
            ...               ...
            Send(S, X);       Receive(S, Y);
            ...               ...
```

el proceso A no podrá seguir ejecutando más alla del *Send* si el B no ha llegado a ejecutar el *Receive* correspondiente. Si el canal hubiera sido asíncrono, el proceso A podría haber seguido ejecutando después del *Send*, independientemente de la velocidad del proceso B. Por supuesto, la ejecución de la operación *Receive* siempre es bloqueante.

Por tanto los canales y sus operaciones son elementos no sólo de comunicación sino de sincronización entre procesos. Como tales han de ser manejados con cuidado, ya que algunos grafos de comunicación por paso de mensajes pueden dar lugar a interbloqueos.

Consideremos, por ejemplo, dos procesos A y B que se envían mensajes mutuamente. El fragmento de código que sigue claramente conduce a interbloqueo:

```
principal:  S,T:Channel; -- síncronos
            ...
       A:   X,W:D;                    B:   Y,Z:D;
            ...                             ...
            Send(S, X);                     Send(T, Y);
            Receive(T, W);                  Receive(S, Z);
            ...                             ...
```

Ambos procesos quedarán bloqueados en las operaciones de envío sin poder progresar, ya que esperan por la ejecución de las operaciones de recepción del otro, en un círculo vicioso. Sin embargo, el siguiente esquema de envío/recepción es perfectamente usable:

```
principal:  S,T:Channel; -- síncronos
            ...
       A:   X,W:D;                    B:   Y,Z:D;
            ...                             ...
            Send(S, X);                     Receive(S, Z);
            Receive(T, W);                  Send(T, Y);
            ...                             ...
```

A simple vista podría parecer que es más cómodo trabajar con paso de mensajes asíncrono, pues la probabilidad de interbloqueos es menor. Sin embargo, hay una serie de problemas asociados con el paso de mensajes asíncrono, siendo la más clara la necesidad de crear colas de mensajes con cada canal. Al poder haber una gran diferencia entre las velocidades de ejecución de emisor y receptor, el tamaño de dichas colas puede hacerse arbitrariamente grande. En el peor de los casos, el emisor puede estar enviando mensajes inútilmente a un receptor interbloqueado.

Por ello, en el ámbito de los lenguajes de programación concurrente, los canales suelen ser síncronos. No obstante, hay nichos donde es razonable usar canales asíncronos. Un caso es cuando emisor y receptor pueden estar geográficamente muy alejados y el retardo ocasionado por sincronizarlos puede ser inaceptable. El ejemplo típico son los canales del protocolo *TCP/IP — sockets*.

Otro caso es el de ciertos lenguajes para programación paralela. Ahí el objetivo fundamental es la velocidad del conjunto de procesos, por encima de la facilidad o elegancia de programación. Se asume que el programador ha sido capaz de afinar las velocidades relativas de los procesos de tal manera que los problemas de encolado no puedan darse. En la programación concurrente *normal*, por el contrario, nunca podemos asumir nada sobre las velocidades relativas de los procesos.

2.1.2. Nombrado implícito vs. nombrado explícito

Una primera decisión a tomar es si los canales van a ser elementos visibles, tangibles del lenguaje de programación (nombrado explícito) o bien van a ser ocultos al programador (nombrado implícito).

La primera opción asume que los canales *son un tipo de datos* del lenguaje y que, como tales, han de ser declarados, inicializados, destruidos, etc. A cambio, podemos disponer de ellos en estructuras de datos, como vectores y matrices, pasarlos como argumentos de funciones y procedimientos, etc.

La otra opción consiste en ocultar al máximo al programador la existencia de los canales para centrarse más en los procesos y en construcciones de programación de más alto nivel, como la llamada a procedimientos y funciones, etc.

A simple vista podría parecer que la opción del nombrado implícito es superior, ya que libera de distracciones innecesarias al programador y reduce la probabilidad de errores provocados por

un manejo equivocado de los canales. En efecto, esa es la tendencia en los lenguajes modernos de alto nivel en general, y en Ada 95 en particular.

Sin embargo, hay técnicas de programación muy útiles que no serían posibles sin el uso de canales como tipos de datos:

- Muchos algoritmos paralelos se basan en una estructura regular – vectorial o matricial – de replicación de procesos. En esos casos es útil poder declarar explícitamente vectores o matrices de canales para comunicar dichos procesos.
- Otra situación, que se da con cierta frecuencia, es cuando se tiene un proceso servidor y un número de clientes que puede variar dinámicamente en tiempo de ejecución. Una técnica habitual consiste en que cada proceso cliente se identifique, al realizar una petición al servidor, con el canal por el que desea recibir la respuesta.
- Finalmente, la técnica de desbloqueos explícitos que usaremos en esta metodología se basará en el uso de canales nombrados explícitamente.

2.1.3. Esquemas de comunicación

Un aspecto relacionado con lo anterior es el de cómo interactúan diferentes procesos gracias a los canales de comunicación. Se suelen considerar los siguientes esquemas de comunicación:

1:1 Cada canal comunica a un único emisor con un único receptor. Este suele ser el esquema habitual cuando se tienen canales implícitos.

n:1 Varios procesos emisores pueden enviar mensajes por un canal del cual lee un único receptor. Es un esquema bastante habitual, sobre todo con canales explícitos síncronos (ver más adelante): *ports*.

n:m Se permite que varios procesos envíen y varios lean de un mismo canal. No es una opción que haya tenido demasiado éxito aunque existen unas cuantas propuestas de carácter académico — con canales asíncronos (ver más adelante): *buzones*.

Podemos ver que otra de las ventajas del nombrado explícito de canales es disponer de esquemas de comunicación diferentes al 1:1.

2.2. La necesidad de la recepción alternativa

Imaginemos un proceso *Servidor* que tuviera que atender dos tipos de peticiones, por los canales $C1$ y $C2$. Si su código correspondiera al siguiente esquema

```
Servidor:  Pet1:T1;
           Pet2:T2;
           loop
           ...
           Receive(C1, Pet1);
           atender la petición 1
           Receive(C2, Pet2);
           atender la petición 2
           ...
           end loop;
```

la velocidad del servidor, y de todo el sistema, iría marcada por la más lenta de entre los tres procesos, situación claramente mejorable. Una solución (chapuza) sería mandar todas las peticiones por el mismo canal, pero eso suele ser incómodo ya que podemos querer usar tipos diferentes para las distintas peticiones.

Para evitar estos problemas, los lenguajes de alto nivel para programación concurrente proporcionan algún tipo de mecanismo de *recepción alternativa*, es decir, permiten que un proceso

esté pendiente de recibir por uno de entre varios canales. La semántica de estos mecanismos – que veremos con detalle más adelante – suele ser la siguiente:

- Si cuando el receptor llega al punto de la recepción alternativa hay mensajes listos en alguno de los canales, tomará uno de esos mensajes *al azar* olvidándose de los demás envíos. El resto de emisores que estaban bloqueados seguirán bloqueados hasta que el receptor tome su mensaje.
- Si, por el contrario, el receptor llega al punto de la recepción alternativa y ninguno de los emisores ha ejecutado aún su *Send*, el receptor quedará bloqueado a la espera del primero de los mensajes.

2.3. Mecanismos de sincronización

A lo largo de la historia de la programación concurrente se han ido proponiendo diversos mecanismos para la comunicación entre procesos distribuidos. Aquí nos limitaremos a reseñar algunos que han conducido a los conceptos que usaremos en estas notas.

2.3.1. CSP

CSP (*Communicating Sequential Processes*) fue propuesto por C.A.R. Hoare en la década de los 70 e introdujo el concepto de recepción alternativa en paso de mensajes. Parte de su popularidad se debe a su realización en el lenguaje Occam.

En principio, el nombrado de canales es implícito, y lo que hace el emisor es dar el nombre del proceso receptor y viceversa:

```
X ! Receptor; -- envía X al receptor
Y ? Emisor; -- recibe Y del emisor
```

Los canales son síncronos, 1:1 y estáticos (conocidos en tiempo de compilación).

CSP proporciona una construcción para la recepción alternativa que incluye, mediante guardas, la posibilidad de codificar sincronización por condición:[2]

```
alt
    Cond1 & X1 ? Emisor1;
        atender al emisor 1
    Cond2 & X2 ? Emisor2;
        atender al emisor 2
    ...
    CondN & XN ? EmisorN;
        atender al emisor N
end alt;
```

La semántica de esta construcción es la siguiente:

- Si cuando el receptor llega al punto de la recepción alternativa hay mensajes listos en alguno de los canales cuya guarda se haya evaluado a cierto, tomará uno de esos mensajes olvidándose de los demás envíos, y ejecutará exclusivamente las instrucciones que siguen a esa recepción. El resto de emisores que estaban bloqueados seguirán bloqueados hasta que el receptor tome su mensaje.
- Si, por el contrario, el receptor llega al punto de la recepción alternativa y ninguno de los emisores ha ejecutado aún su envío, el receptor quedará bloqueado a la espera del primero de los mensajes que llegue por uno de los canales con una guarda a cierto.

[2] Al igual que en el documento de objetos protegidos, usaremos pseudo-Ada para ilustrar propuestas anteriores.

A modo de ejemplo, este es el código para un proceso que sirve un buffer compartido a un productor y un consumidor:

```
loop
    alt
        not(Lleno(Buffer)) & Dato ? Productor;
            Insertar(Buffer,Dato);
        not(Vacio(Buffer)) & Pet ? Consumidor;
            Extraer(Buffer,Dato);
            Dato ! Consumidor;
    end alt;
end loop;
```

2.3.2. RPC

RPC – Remote Procedure Call, llamada a procedimientos remotos – es, en principio, un conjunto de librerías pensadas para simplificar la programación de TCP/IP en un sistema UNIX.

Si comparamos RPC con cualquier lenguaje de programación de alto nivel, puede parecer bastante primitivo, pero introduce novedades que significan un avance respecto a los *sockets* originales. La más destacada, y que da nombre a la propuesta, consiste en agrupar toda la secuencia de envíos y recepciones que tienen lugar entre cliente y servidor en única operación de llamada a procedimiento síncrona — recuérdese que los sockets son asíncronos, pero se puede simular el comportamiento síncrono introduciendo mensajes de confirmación adicionales.

Supongamos que tenemos un par de procesos, *Cliente* y *Servidor*. *Servidor* declararía uno o más procedimientos públicos que pueden ser invocados por los procesos clientes:

```
task type Servidor is
    op Servicio1 (parámetros);
    op Servicio2 (parámetros);
    ...
    op ServicioN (parámetros);
end Servidor;
task body Servidor is
    op Servicio1 (...) is
        ...
    end Servicio1;
    ...
    op ServicioN (...) is
        ...
    end ServicioN;
end Servidor;
```

Desde el punto de vista del cliente, la semántica de la llamada es como la de cualquier otra llamada a procedimiento. Sin embargo, lo que sucede es que el proceso cliente se bloquea tras enviar los parámetros al servidor, que es quien realmente lleva a cabo la ejecución del procedimiento. Éste, al terminar su trabajo, envía un mensaje con el resultado al cliente, al que desbloquea.

2.3.3. Rendez-Vous

El lenguaje Ada propone, como mecanismo para la comunicación y sincronización de procesos, el llamado *Rendez-Vous*. Podemos verlo como una mezcla de las ideas en CSP y RPC. La expresividad y semántica es la de una construcción *alt* de CSP – comunicación síncrona, guardas locales, recepción alternativa no determinista – pero la sintaxis es más concisa al adoptar los servicios la forma de procedimientos remotos.

Estos procedimientos se denominan *entries* (como en objetos protegidos) y cada parámetro de entrada oculta un envío del cliente al servidor y cada parámetro de salida un envío del servidor al cliente.

La notación de *rendez-vous*, que detallaremos en la próxima sección, permite expresar de forma muy elegante soluciones cliente/servidor a muchos problemas. Lamentablemente, no es suficientemente expresiva para algunos casos, lo cual motivó, entre otras cosas, la introducción de los objetos protegidos y el *requeue* en Ada 95. Nuestro enfoque será el de complementar el mecanismo de rendez-vous con un uso esporádico de canales explícitos para subsanar las limitaciones de esta notación y así proporcionar una metodología aceptablemente coherente para procesos distribuidos en Ada.

3. Rendez-Vous

Como acabamos de comentar, el rendez-vous de Ada está claramente orientado a una visión cliente/servidor. El recurso compartido será "propiedad" de un proceso servidor, que habrá de declarar un serie de servicios públicos a los procesos cliente:

```
task type Tipo_Servidor is
    entry Operación1 (parámetros);
    ...
    entry OperaciónN (parámetros);
end Tipo_Servidor;
```

Dentro de nuestra metodología, estos servicios serán las operaciones de la interfaz de un CTAD-SOL. Los procesos clientes pueden invocar esas operaciones usando una sintaxis similar a la de un objeto protegido:

Tarea_Servidora.*OperacionX* (*parámetros*);

El código del proceso servidor suele consistir en un bucle dentro del cual se ejecuta una estructura de recepción alternativa de peticiones — *select*:

```
select
    when condición =>
        accept Operacion1 (parametros) do
            ...
        end;
        <sentencias fuera del rendez-vous>
or
    when condición =>
        accept Operacion2 (parametros) do
            ...
        end;
        <sentencias fuera del rendez-vous>
or
...
or
    when condición =>
        accept OperacionN (parametros) do
            ...
        end;
        <sentencias fuera del rendez-vous>
end select
```

La semántica de la *select* es similar a la del *alt* de CSP/Occam. En primer lugar se evalúan las guardas, y aquellas que resulten ser falsas son descartadas. Las cláusulas *accept* cuya guarda ha evaluado a *Cierto* son los servicios disponibles, en este instante, para los clientes.

- Si, en el momento de llegar el servidor a la ejecución de la *select*, algún cliente ha ejecutado una llamada a alguno de los servicios disponibles, se seleccionará uno de ellos.
- Si, por el contrario, ningún cliente ha comunicado todavía su interés en ninguno de los servicios disponibles, el servidor se quedará bloqueado y atenderá a la primera petición que reciba.

Cuando se selecciona uno de los servicios, se ejecuta exclusivamente el código correspondiente a su cláusula *accept*. Como se puede ver, este código consta de dos partes bien diferenciadas: el rendez-vous propiamente dicho, que abarca desde el "`do`" hasta el "`end`" que lo cierra, y el resto de código para esa cláusula.

El rendez-vous es la zona de sincronización entre cliente y servidor:el cliente cuya petición está siendo atendida queda bloqueado hasta que el servidor alcanza el final del rendez-vous. Las instrucciones restantes ya pueden ser ejecutadas de manera concurrente por cliente y servidor, a diferencia de lo que ocurría con objetos protegidos.

Es importante tener esto en cuenta pues si se coloca alguna operación lenta, bloqueante o propensa a fallar en el rendez-vous se provocaría un problema de falta de concurrencia o interbloqueo al cliente. Como regla general, este tipo de operaciones debe colocarse siempre fuera del rendez-vous.

El rendez-vous es también el ámbito de visibilidad de los parámetros formales de la cláusula *accept*. Esto es razonable, pues nos prohibe modificar los parámetros de salida más allá del rendez-vous, cuando el cliente ya se ha llevado su copia.

Al igual que en CSP y análogamente a lo que sucedía con objetos protegidos, las guardas sólo pueden hacer referencia a variables de estado del servidor, nunca a los parámetros formales del *accept*.

3.1. Esquemas de código

La situación más sencilla que nos podemos encontrar es la de un CTAD donde ninguna de sus CPREs depende de parámetros de entrada. En este caso el servidor del recurso tendrá una *entry* por cada operación del CTAD y un bucle principal donde se pondrán a disposición de los clientes aquellas operaciones cuya CPRE sea cierta:

```
task type Tipo_Servidor is
    entry Operación1 (parámetros);
    ...
    entry OperaciónN (parámetros);
end Tipo_Servidor;
task body Tipo_Servidor is
    <declaración/inicialización del estado del servidor>
begin
    <resto de la inicialización>
    loop
        select
            when CPRE1 =>
                accept Operacion1 (parametros) do
                    ...
                end;
                <sentencias fuera del rendez-vous>
        or
        ...
```

```
        or
            when CPREN =>
                accept OperacionN (parametros) do
                    ...
                end;
                <sentencias fuera del rendez-vous>
        end select
    end loop
end Tipo_Servidor
```

Si la CPRE depende de parámetros de entrada pasamos a un esquema con bloqueo en dos fases relativamente similar al que se usaba con objetos protegidos: una rama de la *select* con la guarda a *Cierto* en la que se envía la información necesaria para evaluar la CPRE y un posterior bloqueo al cliente hasta que la CPRE se cumple y se decide servir la operación.

Para llevar a cabo este bloqueo en dos fases haremos uso del paquete genérico *Channel* que proporciona un tipo de canales sencillos con operaciones *Send* y *Receive* — nombrado de canales explícito:

```
generic
   type Message is private;
package Channels is
   type Channel is private;
   procedure Create (C : out Channel);
   procedure Send (C : in out Channel;
                   M : in     Message);
   procedure Receive (C : in out Channel;
                      M :    out Message);
   procedure Destroy (C : in out Channel);
private
  ...
end Channels;
```

Como se puede ver, el paquete es genérico en el tipo de los mensajes que se envían por él. Necesitaremos diferentes instancias del tipo para distintos usos. La implementación que os proporcionamos está basada en objetos protegidos, pero puede pensarse en otras realizaciones alternativas.

Disponiendo de este tipo es posible realizar un bloqueo en dos fases donde el bloqueo que en objetos protegidos correspondía a la *entry* privada corre ahora a cargo de un envío explícito del servidor al cliente:

```
task type Tipo_Servidor is
    entry Operación1 (parámetros);
    ...
    entry OperaciónN (parámetros);
end Tipo_Servidor;
task body Tipo_Servidor is
    <declaración/inicialización del estado del servidor>
begin
    <resto de la inicialización>
    loop
        select
            ...
        or
            when True =>
                accept OperaciónX (parámetros entrada + canales respuesta) do
                    <almacenar petición>
```

```
                end;
        or
        ...
        end select
        while hay peticiones pendientes que se puedan atender loop
            <extraer (CanalRespuesta, DatosPetición)>
            <realizar operación>
            Send(CanalRespuesta, respuesta/confirmación);
        end loop;
    end loop
end Tipo_Servidor;
```

El proceso cliente tendrá que ejecutar una llamada a la *entry* seguida de una recepción incondicional:

```
CResp : InstanciaDeChannel.Channel;
...
Tipo_Servidor.OperaciónX (..., CResp);
Receive (CResp, respuesta/confirmación);
...
```

El mensaje del servidor suele contener las variables de salida de la operación del recurso, en caso de existir, o una simple confirmación en caso contrario.

Aunque este es un esquema general de codificación del servidor, comúnmente se opta por esquemas mixtos, en los que las operaciones cuya CPRE no depende de los datos de entrada se sincronizan por la guarda y el resto usa canales explícitos para sincronizar al cliente.

3.2. Desbloqueo explícito

El esquema que acabamos de ver proporciona también un mecanismo para la programación de desbloqueos explícitos mucho más directo y elegante que el que usábamos con objetos protegidos. Dependiendo de

a) la(s) estructura(s) de datos que usemos para almacenar las peticiones pendientes y

b) el criterio de selección para acceder a ellas

obtendremos diferentes versiones de una solución (correcta desde el punto de vista de las propiedades de seguridad) atendiendo a diferentes consideraciones de vivacidad.

El desbloqueo basado en la ejecución de un *Send* desde el servidor se parece mucho más al que se puede realizar mediante un *Signal* usando semáforos o un *Continue* usando monitores. Recordemos que con objetos protegidos sólo podíamos aspirar a *simular* el desbloqueo explícito forzando a que todas las guardas, salvo una, se hicieran falsas.

Las ventajas que esto conlleva, desde el punto de vista de la programación, son las siguientes:

Código más elegante Un problema que teníamos con objetos protegidos era que las propiedades de vivacidad/prioridad obligaban a fortalecer las guardas de las entradas, con lo cual por una parte se perdía la correspondencia directa con las CPREs y, por otra parte, se corría el riesgo de provocar falta de concurrencia e incluso interbloqueo.

Menor probabilidad de inanición Otro problema de los desbloqueos explícitos en objetos protegidos era que el desbloqueo de una serie de procesos en espera había de ser programado como una *cascada de desbloqueos*, donde cada proceso que termina su ejecución en el objeto protegido debe establecer las condiciones necesarias para que se desbloquee el siguiente y así sucesivamente. La lógica que esto implica es muy propensa a errores, siendo muy fácil provocar inanición de algunos procesos.

Con el esquema del servidor, el bucle que sigue a la *select* debe estar programado de tal manera que se asegure que el servidor no volverá a pasar por la *select* hasta que no queden peticiones por procesar. Esto evita que se "cuelen" peticiones nuevas, reduciéndose grandemente el riesgo de inanición.

En la sección 4.2 veremos un ejemplo.

4. Ejemplos

4.1. Problema del productor/buffer/consumidor (sincronización automática, independencia de los argumentos de entrada)

Se trata de un problema clásico en que dos procesos cooperan de modo que uno de ellos se encarga de generar uno a uno una serie de datos, y el otro proceso se encarga de procesar dichos datos, también uno a uno, y en el mismo orden.

La idea es que ambos procesos operen concurrentemente. Si el tiempo de generar o procesar un dato es variable, para evitar esperas innecesarias se utiliza un buffer intermedio en que se van almacenando los datos generados por el primer proceso, en espera de ser procesados por el segundo. Se supone que este buffer tiene una capacidad limitada y constante.

El buffer constituye un elemento o recurso compartido por los dos procesos. La actualización correcta del buffer al almacenar o extraer elementos exige, en general, que estas operaciones se hagan con acceso exclusivo a los datos internos del buffer, por lo que resulta apropiado programarlo como una abstracción de datos encapsulada como un objeto protegido.

Propiedades de la abstracción de datos

- Los elementos se almacenan uno a uno, y se extraen posteriormente en el mismo orden.

Propiedades de seguridad

- No se puede almacenar y extraer a la vez.
- El productor no puede almacenar en el buffer si éste está lleno.
- El consumidor no puede extraer del buffer si éste está vacío.

En este problema no hay propiedades particulares de vivacidad/prioridad.

Especificación abstracta

Las propiedades anteriores se recogen en la siguiente especificación:

C-TADSOL TipoBufferAcotado
USA TipoCola, TipoElemento
OPERACIONES
ACCIÓN Poner: *TipoBufferAcotado[es]* $\times$ *TipoElemento[e]*
ACCIÓN Tomar: *TipoBufferAcotado[es]* $\times$ *TipoElemento[s]*

SEMÁNTICA
DOMINIO:
TIPO: *TipoBufferAcotado = (cuantos:*$\mathbb{N}$ $\times$ *cola:TipoCola(TipoElemento))*
INVARIANTE: $\forall$ buf $\in$ TipoBufferAcotado $\bullet$ buf.cuantos = NumElementos (buf.cola)
INICIAL(buf**):** buf.cuantos^{sal} = 0 $\wedge$ buf.cola^{sal} = CrearVacia

PRE: cierto
***CPRE*:** *El buffer no está totalmente lleno.*
CPRE: buf.cuantos $<$ MaxElementos
Poner (buf, elemento)
***POST*:** *El buffer tiene un elemento mas, situado al final de la cola*
POST: buf.cuantos^{sal} = buf.cuantos^{ent} + 1$\wedge$
buf.cola^{sal} = Insertar(buf.cola^{ent}, elemento)

PRE: cierto
***CPRE*:** *El buffer no está vacío.*
CPRE: buf.cuantos $>$ 0
Tomar (buf, elemento)
***POST*:** *El buffer tiene un elemento menos, retirado del principio de la cola*
POST: buf.cuantos^{sal} = buf.cuantos^{ent} - 1$\wedge$ elemento^{sal} = Primero(buf.cola^{ent}) $\wedge$
buf.cola^{sal} = Borrar(buf.cola^{ent})

Al tratarse de una especificación donde ninguna de las CPREs depende de parámetros de entrada, la traducción a procesos distribuidos con rendez-vous será inmediata.

Estructura del programa

El programa constará de dos tipos de procesos cliente, *Productor* y *Consumidor*, y un proceso *TheBufServer* que actuará como servidor del recurso.

Los procesos adoptarán la forma:

```
task type BufServer is
   ...
end BufServer;
...
TheBufServer : BufServer;
...
-- definición de los productores
task type Productor;
task body Productor is
   Dato : TDato;
begin
   loop
      < producir dato >
      TheBufServer.Poner (Dato);
   end loop;
end Productor;

-- definición de los consumidores
task type Consumidor;
task body Consumidor is
   Dato : TipoDato;
begin
   loop
      TheBufServer.Tomar (Dato);
      < consumir dato >
   end loop;
end Consumidor;

Productor1 : Productor;
Consumidor1 : Consumidor;
```

Definición del recurso

El recurso es encapsulado como variable local a un proceso servidor. Este proceso declara dos entradas públicas que son invocadas por los clientes:

```
task type BufServer is
   entry Poner (Dato : in TDato);
   entry Tomar (CRP : in Channel);
end BufServer;
task body BufServer is
   Buf : Buffer;
begin
   loop
      ...
   end loop;
end BufServer;
```

Diseño de la sincronización (de seguridad)

El enunciado distinguía tres restricciones de seguridad. La primera de ellas, la exclusión entre operaciones, es implícita en el objeto protegido. Las otras dos se recogen en la tabla de sincronización correspondiente a las situaciones de bloqueo:

Operación	C-PRE informal	dep.	C-PRE codificada
Poner (Item : TDato[ent])	Buffer no lleno	no	$n < MAX$
Tomar (Item : TDato[sal])	Buffer no vacío	no	$n > 0$

Implementación de la sincronización (de seguridad)

La integración de la sincronización de seguridad en el esquema inicial de código es inmediata, al no haber ninguna dependencia de datos de entrada.

```
task type BufServer is
   entry Poner (Item : in TDato);
   entry Pet (Item : out TDato);
end BufServer;
task body BufServer is
   Buf : Buffer;
begin
   loop
      select
         when not (IsFull (Buf)) =>
            accept Poner (Item : in TDato) do
               -- es petición de meter
               Push (Item, Buf);
            end Poner;
      or
         when not (IsEmpty (Buf)) =>
            accept Tomar (Item : out TDato) do
               -- es petición de sacar
               Pop (Buf, Item);
            end Tomar;
      end select;
   end loop;
end BufServer;
```

Operación ejecutada / **POST ampliada**	**Operación bloqueada C-PRE**	
	Poner $n < MAX$	Tomar $n > 0$
Poner $n^{sal} = n^{ent} + 1(\text{post}) \wedge n^{ent} \geq 0(\text{inv.})$	$n < MAX$	$cierto$
Tomar $n^{sal} = n^{ent} - 1(\text{post}) \wedge n^{ent} \leq MAX(\text{inv.})$	$cierto$	$n > 0$

Figura 1: Tabla de desbloqueo para el buffer acotado.

El análisis de la tabla de desbloqueos (fig. 1) y la ausencia de requisitos de vivacidad adicionales hace que no sea necesario retocar este código.

Obsérvese, en efecto, que si por algún motivo hubiera más probabilidades de atender las llamadas de *Poner* que las de *Tomar*, la precondición de aquellas se acabaría haciendo falsa, y viceversa.

4.2. Cuenta bancaria compartida (*sin* y *con* desbloqueos explícitos)

Vamos a usar este ejemplo para:

a) Mostrar cómo tratar aquellos casos en que una CPRE depende de un parámetro de entrada.

b) Mostrar las técnicas de desbloqueo explícito.

Se trata de sincronizar accesos concurrentes a una cuenta bancaria, con las condiciones de seguridad de la exclusión mutua en la actualización del saldo y que éste nunca debe hacerse negativo.

Especificación formal de la interacción

C-TADSOL Cuenta

OPERACIONES
ACCIÓN Ingreso: *Tipo_Cuenta[es]* $\times$ *Tipo_Saldo[e]*
ACCIÓN Reintegro: *Tipo_Cuenta[es]* $\times$ *Tipo_Saldo[e]*

SEMÁNTICA
DOMINIO:
TIPO: *Tipo_Cuenta = (Saldo : Tipo_Saldo)*
INICIAL(b**):** b.Saldo = ...

CPRE*: *Cierto
CPRE: Cierto
Ingreso (b, s)
POST*: *Incrementamos el saldo
POST: b.Saldo^{sal}= b.Saldo^{ent}+ s

CPRE*: *Hay saldo suficiente
CPRE: b.Saldo $\geq$ s
Reintegro (b, s)
POST*: *Se decrementa el saldo
POST: b.Saldo^{sal}= b.Saldo^{ent}- s

Como se puede observar, la CPRE de la operación *Reintegro* depende de uno de los parámetros de entrada, y no es factible desglosar esta operación en cada una de las posibles cantidades que queremos sacar.

La solución que proponemos para estos casos consiste, al igual que en el caso de los objetos protegidos, en realizar un bloqueo en dos fases. Una primera rama del rendez-vous, con la guarda a *Cierto*, se usa para pasar al servidor la información necesaria para evaluar la CPRE. El segundo bloqueo – el que en objetos protegidos se realizaba mediante una entrada privada y quizás un *requeue* – se realizará ahora mediante un canal del servidor al cliente.

Este canal tiene tres funciones:

- Identificar al cliente.
- Enviar al cliente los parámetros de salida de la operación, si los hubiera.
- Mantener al cliente bloqueado hasta que se compruebe que la CPRE se cumple.

Estructura del programa

Tendremos dos tipos de procesos, que llamaremos *padres* (que ingresan dinero) e *hijos* (que lo gastan). Las operación *Reintegro* del recurso compartido tendrán además un parámetro de entrada extra donde se pasa el canal de confirmación:

```
MiCuenta : ServidorCuenta;

-- Los que ingresan y sacan dinero son tareas
-- Los padres ingresan dinero..
task type Papa_Type;
task body Papa_Type is
begin
   loop
      ...
      MiCuenta.Ingreso(500.0);
   end loop;
end Papa_Type;

-- ..y los hijos lo gastan
task type Nene_Type;
task body Nene_Type is
   CResp : Channel_Int.Channel;
   Cuanto : Saldo_Type;
   Conf : Integer;
begin
   Create(Cresp);
   loop
      ...
      <calcular Cuanto>
      MiCuenta.Reintegro(Cuanto, CResp);
      -- ahora hay que esperar por la confirmación
      Receive(CResp, Conf);
   end loop;
end Nene_Type;

-- Lanzamos los procesos al declararlos
-- como componentes de un vector.
Papas : array(1..Num_Papas) of Papa_Type;
```

```
Nenes : array(1..Num_Nenes) of Nene_Type;
```

Desarrollo de la sincronización de seguridad

Mostramos aquí la traducción del recurso al proceso servidor, o al menos la parte que es común a las diferentes políticas de servicio:

```
task type Cuenta is
   entry Reintegro (Cant : in Saldo_Type; CResp : Channel_Int.Channel);
   entry Ingreso (Cant : in Saldo_Type);
end Cuenta;

task body Cuenta is
   Saldo : Saldo_Type := Saldo_Inicial;
   -- realmente habría que leer de la BD
   Cantidad_Pendiente : Pendientes;
begin
   loop
      select
         when True => -- siempre permitimos ingresar
            accept Ingreso (Cant : in Saldo_Type) do
               Saldo := Saldo + Cant;
            end Ingreso;
      or
         when True =>
            accept Reintegro (Cant : in Saldo_Type; CResp : Channel) do
               <añadir Cant y CResp al conjunto de peticiones pendientes>
            end Reintegro;
      end select;
      -- ahora se atiende a los reintegros
      while <hay reintegros pendientes y saldo para atender> loop
         <escoger (Cant1, Cresp1) y eliminarlos del conjunto>
         Saldo := Saldo - Cant;
         Send(CResp, 1); -- el cliente estará esperando
      end while;
   end loop;
end Cuenta;
```

Vamos a ver que el mismo esqueleto de código nos va a servir para desarrollar varias versiones, con diferentes propiedades de vivacidad, dependiendo de qué estructura de datos usemos para almacenar y seleccionar la siguiente petición a atender.

Versión con atención por orden de llegada

Como ya sabemos, si nos limitamos a seleccionar una petición al azar perjudicaremos, a la larga, a las más elevadas. De hecho, existe un riesgo evidente de inanición.

La solución más drástica contra este riesgo de inanición es atender las peticiones de reintegro estrictamente por orden de llegada — si bien reducimos notablemente la concurrencia del sistema. Para conseguir este comportamiento basta con usar una cola para almacenar las peticiones pendientes:

```
task type Cuenta is
   entry Reintegro (Cant : in Saldo_Type; CResp : Channel_Int.Channel);
   entry Ingreso (Cant : in Saldo_Type);
end Cuenta;
```

```
task body Cuenta is
   Saldo : Saldo_Type := Saldo_Inicial;
   -- realmente habría que leer de la BD
   ColaCantidades : Cola_Cantidad.TCola := Cola_Cantidad.CrearVacia();
   ColaCanales : Cola_Canal.TCola := Cola_Canal.CrearVacia();
   Cant1 : Tipo_Saldo;
   CResp1 : Canal_Conf.Channel;
begin
   loop
      select
         when True => -- siempre permitimos ingresar
            accept Ingreso (Cant : in Saldo_Type) do
               Saldo := Saldo + Cant;
            end Ingreso;
      or
         when True =>
            accept Reintegro (Cant : in Saldo_Type; CResp : Channel) do
               Cola_Canal.Push(ColaCanales,CResp);
               Cola_Cantidad.Push(ColaCantidades,Cant);
            end Reintegro;
      end select;
      -- ahora se atiende a los reintegros
      while not(Cola_Canal.IsEmpty(ColaCanales)) and
              (Saldo >= Cola_Cantidad.First(ColaCantidades)) loop
         Cola_Canal.Pop(ColaCanales,CResp1);
         Cola_Cantidad.Pop(ColaCantidades,Cant1);
         Saldo := Saldo - Cant1;
         Send(CResp1, 1); -- el cliente estará esperando
      end while;
   end loop;
end Cuenta;
```

Como la petición que determina si se atiende o no a las demás es siempre la más antigua, es habitual usar un esquema donde sólo se guarda una petición en el servidor, y la guarda de *Reintegro* se cierra mientras existe petición en espera. Mediante este esquema nos ahorramos las colas en el servidor, ya que el encolado pasa a suceder en la llamada a la entrada:[3]

```
task type Cuenta is
   entry Reintegro (Cant : in Saldo_Type; CResp : Channel_Int.Channel);
   entry Ingreso (Cant : in Saldo_Type);
end Cuenta;

task body Cuenta is
   Saldo : Saldo_Type := Saldo_Inicial;
   -- realmente habría que leer de la BD
   HayPendiente : Boolean := False;
   CantidadPendiente : Saldo_Type;
   CanalRespuesta : Canal_Conf.Channel;
begin
   loop
      select
```

[3]Esto presupone que una atención por algún orden estricto (que puede ser de llegada, dependiendo del compilador) es adecuada.

```
         when True => -- siempre permitimos ingresar
            accept Ingreso (Cant : in Saldo_Type) do
               Saldo := Saldo + Cant;
            end Ingreso;
      or
         when not HayPendiente => -- OJO!!
            accept Reintegro (Cant : in Saldo_Type; CResp : Channel) do
               CanalRespuesta := CResp;
               CantidadPendiente := Cant;
            end Reintegro;
      end select;
      -- ahora se atiende al reintegro
      if HayPendiente and CantidadPendiente <= Saldo then
         Saldo := Saldo - CantidadPendiente;
         HayPendiente := False;
         Send(CanalRespuesta, 1); -- el cliente estará esperando
   end loop;
end Cuenta;
```

Versión atendiendo prioritariamente a las peticiones mayores

Otra forma de evitar que se perjudique a las peticiones más elevadas es *atender prioritariamente a las peticiones de mayor dinero*, independientemente de los problemas de inanición que puedan ocasionarse.

Para llevarla a cabo necesitamos una estructura de datos que permita la extracción del elemento *con la mayor clave menor o igual que una dada*. Obsérvese que no es exactamente una *cola con prioridad*, pues en éstas se extrae simplemente el elemento con la mayor clave. La estructura que estamos buscando es relativamente habitual en programas como manejadores de memoria que siguen la política *best fit*, es decir, la de buscar el menor hueco de memoria con tamaño suficiente para acomodar el bloque solicitado.

No consideraremos aquí una representación determinada. Si buscamos un prototipo rápido de realizar bastará con usar un par de colas ordenadas. Si buscamos una representación más eficiente, entonces podemos optar por un árbol de inserción binario. La operación de extraer el elemento con la mayor clave menor o igual que una dada se puede realizar – al igual que la inserción – con complejidad logarítmica.

Supondremos la existencia del tipo *ColaPrioAcotada* y las operaciones

```
Insertar (Cola : in out ColaPrioAcotada;
          Cant : in     Saldo_Type;
          Canal: in     Channel_Int.Channel)
-- Inserta Canal en la Cola de acuerdo con el valor de Cant

Extraer (Cola : in out ColaPrioAcotada;
         Saldo: in     Saldo_Type;
         Hay  :    out Boolean;
         Cant :    out Saldo_Type;
         Canal:    out Channel_Int.Channel);
-- Si existe alguna petición menor o igual que Saldo, Hay toma el
-- valor True, Cant el valor de la mayor petición menor o igual que
-- Saldo, Canal el del canal de confirmación de la petición y la
-- petición es extraída de la cola.
-- En caso contrario, Hay toma el valor False.
```

El código del servidor quedaría entonces como sigue:

```
task type Cuenta is
   entry Reintegro (Cant : in Saldo_Type; CResp : Channel_Int.Channel);
   entry Ingreso (Cant : in Saldo_Type);
end Cuenta;
task body Cuenta is
   Saldo : Saldo_Type := Saldo_Inicial;
   -- realmente habría que leer de la BD
   ColaPendientes : ColaPrioAcotada.ColaPrioAcotada := CrearVacia();
   Hay : Boolean := False;
   CantidadPendiente : Saldo_Type;
   CanalRespuesta : Canal_Conf.Channel;
begin
   loop
      select
         when True => -- siempre permitimos ingresar
            accept Ingreso (Cant : in Saldo_Type) do
               Saldo := Saldo + Cant;
            end Ingreso;
      or
         when True => -- aplazamos el bloqueo
            accept Reintegro (Cant : in Saldo_Type; CResp : Channel) do
               Insertar(ColaPendientes, Cant, CResp);
            end Reintegro;
      end select;
      -- ahora se atiende al reintegro
      loop
         Extraer (ColaPendientes, Saldo, Hay,
                  CantidadPendiente, CanalRespuesta);
         exit when not Hay;
         -- sólo si Hay
         Saldo := Saldo - CantidadPendiente;
         Send(CanalRespuesta, 1); -- el cliente estará esperando
      end loop;
   end loop;
end Cuenta;
```