UNIVERSIDAD DE
LAS PALMAS DE GRAN CANARIA

E.T.S.I. DE TELECOMUNICACIÓN

Drago: un lenguaje para programar aplicaciones distribuidas tolerantes a fallos y cooperativas

Francisco Javier Miranda González

Angel Alvarez Rodríguez

Sergio Arévalo Viñuales

Francisco José Guerra Santana

1995

A mis padres,
a Pino.

RESUMEN

Drago es un lenguaje experimental desarrollado para soportar la implementación de aplicaciones distribuidas estáticas cooperativas y tolerantes a fallos. El modelo de programación de Drago se basa en el paradigma de grupos, y soporta dos abstracciones de grupo: abstracción de *grupo replicado*, que permite implementar aplicaciones tolerantes a fallos mediante redundancia modular, y abstracción de *grupo cooperativo*, que proporciona soporte para implementar un conjunto de miembros distribuidos que cooperan para conseguir un objetivo común. Drago proporciona estas abstracciones mediante la *especificación de grupo*. Una especificación de grupo contiene declaraciones de constantes, tipos, excepciones y puntos de entrada remotos. Todo miembro de un grupo tiene visible todas las declaraciones contenidas en la especificación de dicho grupo y debe proporcionar todos los servicios remotos especificados en ella (mediante puntos de entrada remotos).

La *cita remota con un grupo* es el único mecanismo de comunicación remota y sincronización remota que proporciona Drago. Es una extensión al mecanismo de cita de Ada 83, para la cual la cita debe ser aceptada por todos los miembros vivos del grupo invocado. Drago permite que los miembros de un grupo cooperativo se comuniquen de forma transparente a los clientes remotos (mediante la *comunicación intragrupo*) y reciban notificaciones de fallo de los miembros del grupo que fallen.

El *agente* es la unidad de distribución de Drago. Un agente es un tipo de objeto abstracto: tiene estado interno (no accesible directamente desde el exterior) y operaciones remotas especiales (especificadas mediante los puntos de entrada remotos), que pueden ser llamadas remotamente desde otros agentes mediante *citas remotas con grupos*. El agente puede tener operaciones locales (procedimientos, funciones y puntos de entrada locales) y código de inicialización. Cada agente reside en un nodo de la red, aunque varios agentes pueden residir en el mismo nodo. Un programa distribuido Drago consta de varios agentes que residen en varios nodos de la red. Drago proporciona dos tipos de agentes: *agentes replicados* y *agentes cooperativos*. La principal diferencia entre ambos es que los agentes replicados no pueden tener tareas internas, mientras que los agentes cooperativos sí. Los agentes replicados solamente pueden ser miembros de grupos replicados. Los agentes cooperativos solamente pueden ser miembros de grupos cooperativos.

ABSTRACT

Drago is an experimental language developed to support the implementation of cooperative and fault-tolerant static distributed applications. Drago's programming model is based on the groups paradigm, and supports two group abstractions: *replicated group* abstraction, which gives support for the implementation of fault-tolerant applications by means of modular redundancy, and *cooperative group* abstraction, which supports the implementation of a set of distributed members that cooperate in order to achieve a common goal. Drago provides these abstractions by means of *group specifications*. A group specification contains declarations of constants, types, exceptions and remote entry points. Every member of a group has available all the declarations of the corresponding group specification, and must provide all the remote services specified in it (by means of remote entries).

Group Remote Rendezvous is the only way remote communication and synchronization is provided by Drago. It is an extension to Ada 83 rendezvous which must be accepted by all the living members of the specified group. Drago allows the members of cooperative groups to communicate internally (by means of *intra-group communication*) and provides them a failure notification facility.

The *agent* is the unit of distribution of Drago. An agent is a special kind of abstract object: it has internal state (not directly accesible from outside the agent) and special remote operations (specified by means of remote entries) that can be remotely called. The agent may have local operations (procedures, functions and local entries) and initialization code. Each agent resides at a single node of the network, although several agents may reside at the same node. A Drago distributed program is composed of a number of agents residing at a number of nodes. Drago provides two kind of agents: *replicated agents* and *cooperative agents*. The main difference between them is that replicated agents can not have internal tasks, while cooperative agents can. Replicated agents can only be members of replicated groups, and cooperative agents can only be members of cooperative groups.

Indice

Indice de Figuras

Prólogo

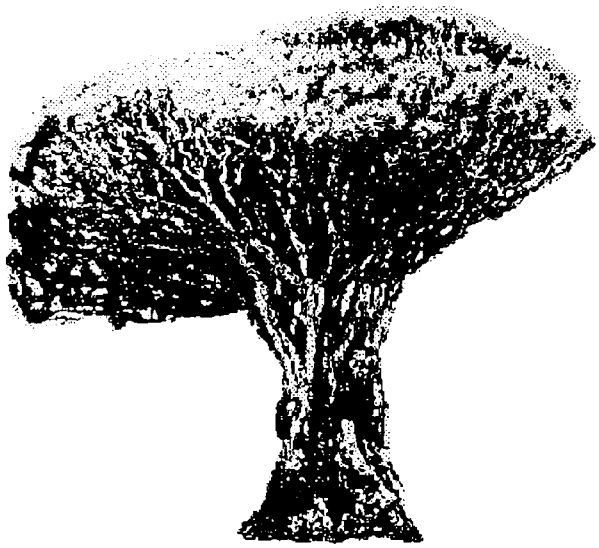
La tesis doctoral presentada en este documento se desarrolló en dos etapas: una etapa de tres años en la Escuela Técnica Superior de Telecomunicación de Madrid (en el departamento de Ingeniería Telemática) y otra de un año en la Escuela Técnica Superior de Telecomunicación de Las Palmas.

Durante los dos primeros años se realizaron los cursos de doctorado simultaneados con una formación básica en sistemas distribuidos. Durante el tercer año se desarrolló una primera versión de Drago[1] que fue perfilada mediante ejemplos y debatida en el seminario de sistemas distribuidos. Al final de esta etapa se intentó desarrollar un traductor que generase código Ada 83 y escribir el manual de referencia correspondiente. Esta primera versión fue finalmente descartada porque:

- Su implementación mediante un preprocesador no era tan sencilla como habíamos pensado.
- Algunos detalles semánticos del lenguaje eran demasiado complicados para reflejarlos claramente en el manual de referencia.
- Se consideró conveniente desarrollar el nuevo lenguaje como una ampliación del lenguaje de programación Ada 83 (permitiendo así reutilizar código escrito en Ada 83), y esta primera versión se alejaba de Ada 83.

Durante el cuarto año se desarrolló e implementó la versión final de Drago. Drago ha sido implementado mediante un preprocesador que genera código Ada 83 y se ejecuta actualmente sobre una red de estaciones de trabajo Sun[2].

[1] Drago es el nombre de un árbol endémico de Canarias y del resto de la flora macaronésica.
[2] Sun Microsystems.



Objetivo

El principal objetivo de esta tesis doctoral era desarrollar un lenguaje que facilitase la programación en entornos de procesamiento distribuidos. Los programas escritos con el nuevo lenguaje podrían distribuirse para su ejecución entre los nodos de una red, logrando un mejor aprovechamiento de los recursos de que se disponiera. Nos planteamos este objetivo como resultado de la experiencia con sistemas de programación tolerantes a fallos de más bajo nivel conceptual, en especial Isis. Pensamos que, para mejorar la práctica de la programación de este tipo de aplicaciones, era fundamental tener un soporte linguístico adecuado, y no sólo conjuntos de subrutinas, sin la posibilidad de disponer de las comprobaciones automáticas que realizan los compiladores de lenguajes modernos. No tendría mucho sentido embarcarse en la construcción de aplicaciones tolerantes a fallos partiendo de un soporte linguístico básico poco fiable.

El lenguaje debía facilitar la utilización del paralelismo inherente de los sistemas distribuidos así como facilitar la programación de aplicaciones tolerantes a fallos del hardware.

Estructura del documento

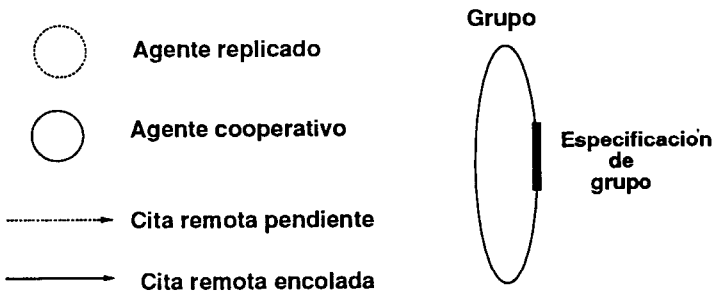
Este documento se ha estructurado en los siguientes capítulos:

- **Capítulo 1:** Antecedentes y estado del arte en el diseño de lenguajes para programación distribuida.
- **Capítulo 2:** Presentación y justificación de los elementos de Drago.
- **Capítulo 3:** Ejemplos de aplicaciones distribuidas tolerantes a fallos escritos con Drago.
- **Capítulo 4:** Ejemplos de aplicaciones distribuidas cooperativas escritos con Drago.
- **Capítulo 5:** Modelo de traducción de Drago al lenguaje de programación Ada 83.
- **Apéndice:** Manual de referencia técnica.

Símbolos utilizados

Los símbolos utilizados en las figuras de este capítulo tienen el siguiente significado:

- Un circulo punteado representa un agente replicado.
- Un circulo en línea continua representa un agente cooperativo.
- Una flecha punteada representa una cita remota pendiente (una cita remota que ha sido iniciada por un agente y que ha sido recibida por todos los miembros de grupo, pero que aún no ha sido encolada por los miembros del grupo en la cola asociada al punto de entrada correspondiente —ver apéndice, apartado A.4.3—).
- Una flecha en línea continua representa una cita remota encolada.
- Una elipse representa un grupo.
- Un rectángulo negro representa la especificación de un grupo.



Agradecimientos

Quiero expresar mi sincero agradecimiento a Francisco Guerra Santana por todos los debates que hemos tenido acerca de Drago y por haber diseñado e implementado todos los protocolos de comunicación con grupos que utiliza el entorno de tiempo de ejecución de Drago. A Angel Álvarez y Sergio Arévalo por su guía y firme apoyo en el desarrollo de Drago. Y a los componentes del seminario de sistemas distribuidos por su colaboración en la revisión de Drago.

También quiero agradecer a Antonio Núñez su firme apoyo, gracias al cual pude compaginar mis estudios de doctorado en Madrid con mi trabajo como profesor en la Escuela Técnica Superior de Telecomunicación de Las Palmas, y al departamento de Ingeniería Telemática de Madrid el haberme permitido cursar los estudios de doctorado en su seno.

Finalmente quiero agradecer a Juan Antonio de la Puente haberme permitido utilizar las estaciones de trabajo del Grupo de Ingeniería de Control durante la etapa de desarrollo de Drago en Madrid, y a Carlos Mena y Pedro Perez Carballo el sacarme de algún que otro apuro con la red de comunicación de la Escuela Técnica Superior de Ingenieros de Telecomunicación de Las Palmas.

Capítulo 1

Introducción

Los *Sistemas de Computación Distribuidos* son sistemas compuestos de múltiples procesadores autónomos que no comparten memoria primaria[1] pero que cooperan mediante el envío de mensajes a través de una red de comunicaciones. Las aplicaciones distribuidas pueden construirse, básicamente, de tres formas: directamente sobre el hardware, sobre un sistema operativo, o utilizando un lenguaje especial para programación distribuida.

El primer enfoque proporciona total control sobre todas las primitivas que proporciona el hardware, tales como vectores de interrupción e interfaces con dispositivos de comunicación. Aunque este mecanismo puede ser flexible y permite una utilización eficiente de los recursos hardware disponibles, tiene el gran inconveniente de ser muy dependiente del hardware y hace la programación muy laboriosa. No analizaremos esta aproximación.

En el segundo enfoque (sobre un sistema operativo), se dice que tenemos un *Sistema Operativo Distribuido*. Con respecto a la metodología de programación, se caracteriza porque generalmente se programa mediante un lenguaje secuencial ampliado con rutinas de biblioteca que invocan a primitivas del sistema operativo. Estas aplicaciones pueden hacerse independientes del hardware, pero siempre dependerán del sistema operativo. Sin embargo, la principal desventaja de este enfoque es que, generalmente, las estructuras de datos y de control de los lenguajes secuenciales no son adecuadas para programación distribuida. Por ejemplo, si se desea pasar una estructura de datos compleja a un proceso remoto, el sistema operativo es incapaz de empaquetar la estructura en mensajes para transmitirlos por la red (ya que no sabe cuál es la representación interna de las estructuras de datos del lenguaje). Para solucionarlo, el programador debe insertar, en el emisor, código que aplane la estructura en una secuencia de bytes, y en el receptor código que realice el proceso inverso. Otro inconveniente es que generalmente las primitivas del S.O. aceptan y entregan los mensajes siguiendo una política FIFO, y en algunas aplicaciones puede ser interesante seleccionar un determinado mensaje de acuerdo con el estado del receptor y/o el contenido del mensaje (para lo cual generalmente habría que hacer varias llamadas al núcleo).

La forma elegante de resolver estos problemas es hacer que el lenguaje proporcione el soporte para la distribución (encargándose de realizar las conversiones de forma totalmente

[1] Cada procesador ejecuta su propia secuencia de instrucciones y utiliza sus propios datos locales (ambos almacenados en su memoria local).

transparente al programador). Esto proporciona además las siguientes ventajas: legibilidad, portabilidad y comprobación de tipos.

1.1 Características de los lenguajes para programación distribuida

Los lenguajes diseñados para programar sistemas distribuidos deben tener en cuenta tres características fundamentales que distinguen a la programación distribuida de la secuencial [BST89] [Bal90]:

- **Paralelismo**: permitir ejecutar partes de un programa (unidades distribuibles) en procesadores diferentes.
- **Comunicación y sincronización**: proporcionar mecanismos de cooperación entre las unidades distribuibles.
- **Tolerancia a fallos**: proporcionar mecanismos que soporten fallos parciales del sistema.

1.1.1 Paralelismo

Es la primera característica que debe tenerse en cuenta en un lenguaje para programación distribuida. Denominamos paralelismo a la posibilidad de tener más de un trozo de un mismo programa ejecutándose a la vez[2]. Esto es posible en un sistema distribuido porque, por definición, todo sistema distribuido tiene más de un procesador. En general, la capacidad de expresar paralelismo es una característica muy importante, ya que permite reflejar lo que realmente está ocurriendo en el sistema. El *pseudo-paralelismo*, técnica que permite que un procesador ejecute cuasi-simultáneamente varios trozos de un programa [Wan89], es tan útil en la programación distribuida como en la programación centralizada, pero deben tenerse en cuenta sus diferencias con respecto al paralelismo real. En algunos lenguajes no se esconde al programador la diferencia entre ambas, y se permite que el programador ubique explícitamente (o *mapee*) trozos de programas en unidades de procesamiento. Esto pone una mayor flexibilidad en manos de los programadores, pero también supone mayor complejidad (por ejemplo, puede utilizarse memoria compartida entre procesos que se ejecutan en pseudo-paralelismo).

Los lenguajes pueden proporcionar paralelismo de forma implícita (la división del código en segmentos paralelos la hace el compilador) o de forma explícita (el lenguaje proporciona estructuras para que el programador especifique las unidades paralelas). La elección entre ambos tipos de paralelismo depende principalmente de si puede implementarse el compilador correspondiente. El paralelismo implícito ha sido aplicado con éxito en máquinas *data flow* y en máquinas vectoriales [BST89]. El paralelismo implícito en máquinas asíncronas MIMD fue estudiado en el proyecto ParaScope [CCH+88] y después de años de experiencia, sus diseñadores llegaron a la conclusión de que *las técnicas automáticas no son suficientes para conseguir un alto rendimiento en sistemas paralelos asíncronos; el programador debe involucrarse en algún punto en la especificación del paralelismo*. En conclusión, dado el estado actual de

[2] [Bal91] analiza de forma práctica diferentes paradigmas de programación paralela.

desarrollo en la tecnología de compiladores, para un lenguaje de programación distribuida el paralelismo explícito es más efectivo que el paralelismo implícito.

La granularidad del paralelismo varía de un lenguaje a otro. En un lenguaje secuencial, la unidad de paralelismo es el programa completo. La unidad de paralelismo de un lenguaje distribuido puede ser:

- **Proceso** (DP [Han78], Ada [ANS83], NIL [SY83], CSP [Hoa85], C-concurrente [GR89], Lynx [Sco91]).
- **Objeto** (SmallTalk concurrente [HW89], Emerald [RTL+91]).
- **Sentencia** (Occam [May83])
- **Expresión** (VAL [Per87]).
- **Cláusula**[3] (Prolog concurrente, PARLOG [CG86]).

En general, cuanto mayor sea el coste de comunicación, tanto mejor resulta utilizar unidades mayores de paralelismo. Por esta razón, para programación distribuida en sistemas débilmente acoplados, solamente veremos a continuación un poco más en detalle los tres primeros modelos (proceso, objeto y sentencia).

1. **Procesos** : En la mayoría de los lenguajes procedimentales diseñados para programación distribuida, el paralelismo está basado en la noción de *proceso*. Diferentes lenguajes tienen diferentes nociones de proceso, pero, en general, un proceso representa un procesador lógico que ejecuta código secuencialmente y que tiene su propio estado y datos.

2. **Objetos** : Un objeto es una unidad autocontenida que encapsula datos y comportamiento (normalmente en forma de procedimientos), y que interactúa con el mundo externo exclusivamente a través de algún mecanismo de paso de mensajes. Los datos contenidos en el objeto son visibles solamente desde dentro del objeto. El comportamiento de un objeto lo define su *clase*, la cual tiene una lista de operaciones que pueden invocarse enviando un mensaje al objeto. El mecanismo de *herencia* permite definir una nueva clase como una extensión de otra previamente definida. Los lenguajes que soportan objetos, pero que carecen de herencia, suelen denominarse *basados en objetos* [CS91].

   Los lenguajes secuenciales orientados a objetos se basan en un modelo de objetos pasivos: un objeto se activa cuando recibe un mensaje de otro objeto. Mientras el receptor está activo, el emisor está bloqueado esperando el resultado. Después de retornar el resultado, el receptor se vuelve pasivo, y el emisor continúa. Para introducir paralelismo puede extenderse el modelo secuencial de las siguientes formas: permitiendo a un objeto estar activo sin haber recibido un mensaje; permitiendo al objeto receptor continuar su ejecución después de retornar su resultado; permitiendo el envío de un mensaje a varios objetos simultáneamente y permitiendo que el emisor continúe en paralelo con el receptor [BST89]. Los dos primeros métodos asignan un proceso paralelo a cada objeto, teniendo como resultado un modelo basado en objetos activos. El tercer método puede

[3] Sólo en lenguajes lógicos.

implementarse utilizando *multicast*. El último método puede implementarse mediante transmisión asíncrona de mensajes, o permitiendo que un objeto tenga internamente múltiples flujos de control (*threads*).

3. **Sentencias paralelas**: Utilizadas por pocos lenguajes, ya que proporcionan poco soporte para la construcción de grandes programas paralelos. Más aún, como indican Andrews y Schneider [AS83], son menos expresivas que los procesos paralelos y generalmente crean un número fijo de unidades paralelas.

1.1.2 Comunicación y sincronización

La segunda característica que debe tenerse en cuenta en el diseño de un lenguaje para programación distribuida es cómo van a cooperar los trozos de programa que se ejecutan en procesadores diferentes. Esta cooperación implica dos tipos de interacción: sincronización y comunicación. Básicamente, la comunicación entre procesos puede ser de dos tipos: comunicación mediante paso de mensajes y comunicación mediante datos compartidos[4]

a) Comunicación mediante paso de mensajes

La primitiva más básica de interacción mediante paso de mensajes es el envío de un mensaje punto a punto entre un proceso (el emisor) y otro proceso (el receptor), aunque no todas las interacciones basadas en el paso de mensajes implican solamente un emisor y un receptor. En la comunicación mediante mensajes deben tenerse en cuenta las siguientes características:

1. **Fiabilidad**: normalmente los lenguajes solamente proporcionan paso de mensajes *fiable* (el entorno de tiempo de ejecución del lenguaje, o el sistema operativo, generan automáticamente mensajes de confirmación de forma totalmente transparente al programador).
2. **Envío**: el emisor comienza la interacción de forma *explícita* (mediante el envío de un mensaje o una llamada remota a un procedimiento). Puede realizarse de dos formas: síncrona o asíncrona.
   (a) **Síncrona**: en la transmisión síncrona de mensajes el emisor se bloquea hasta que el receptor ha aceptado el mensaje (explícita o implícitamente). De esta forma, el emisor y el receptor no solamente se intercambian datos, sino que también se sincronizan.
   (b) **Asíncrona**: en la transmisión asíncrona, el emisor no espera a que el receptor esté preparado para aceptar el mensaje. Conceptualmente, el emisor continúa inmediatamente después de enviar el mensaje (la implementación del lenguaje puede bloquear momentáneamente al emisor hasta que el mensaje se haya copiado para la transmisión, pero este retraso no se refleja en la semántica). Este modelo crea algunos problemas semánticos: dado que el emisor no espera, y puede seguir

[4] El modelo que proporciona el lenguaje para expresar la comunicación, y la implementación de dicho modelo, pueden ser completamente diferentes.

enviando mensajes, el receptor debe tener un buffer de almacenamiento de mensajes no atendidos. Esto introduce un problema de desbordamiento. Hay dos opciones posibles para tratar con este problema:

- Considerar que la transferencia fallará si no hay buffer disponible. Tiene el inconveniente de que la transmisión pasa a ser menos fiable.
- Utilizar control de flujo (el emisor se bloquea hasta que el receptor acepta algunos mensajes). El problema de esta solución es que se introduce sincronización entre el emisor y el receptor (lo cual es contrario a la semántica de la llamada), y puede dar lugar a interbloqueo (*deadlock*).

Los mecanismos síncronos son más sencillos de programar [TVR85], pero fuerzan al emisor a esperar hasta que el receptor ha recibido el mensaje. Para aplicaciones cuyo objetivo es alcanzar grandes velocidades, puede ser una limitación severa. Este problema puede resolverse introduciendo un proceso agente entre el emisor y el receptor, o permitiendo múltiples *threads* en el emisor.

3. **Recepción**: el receptor puede manejar los mensajes recibidos de dos formas:

   (a) *Explícita*: el receptor ejecuta algún tipo de sentencia de aceptación del mensaje especificando qué mensajes acepta, y las acciones asociadas a dichos mensajes.

   (b) *Implícita*: la recepción de un mensaje supone automáticamente la ejecución de cierto código. Normalmente se crea un nuevo flujo de control dentro del proceso receptor[5].

4. **Aceptación condicional**: es posible tener un control más fino si se permite la aceptación condicional de mensajes, dependiendo de los argumentos del mensaje (tal y como ocurre en SR [And81], BSP [Geh84] y C-concurrente [GR89]). Por ejemplo, un servidor de ficheros solamente puede aceptar una nueva petición para abrir un fichero si el fichero en sí no está bloqueado.

5. **Orden de aceptación**: algunos lenguajes permiten al programador controlar el orden de aceptación de los mensajes. Normalmente se aceptan en orden FIFO, pero ocasionalmente es útil cambiar el orden de acuerdo con el tipo de mensaje, el emisor, o el contenido del mensaje.

6. **Direccionamiento**: el direccionamiento puede hacerse de dos formas:

   (a) *Directa*: utilizando el nombre estático del proceso. En este caso el esquema de comunicación puede ser de dos tipos:

      i. *Simétrico* si el emisor y el receptor se nombran mutuamente.

      ii. *Asimétrico* si solamente el emisor nombra al receptor. En este caso el receptor puede interactuar con cualquier emisor[6].

[5] La recepción implícita puede obtenerse fácilmente combinando la recepción explícita con los procesos (Ej. creando un proceso que se encargue de atender dichos mensajes).

[6] Las interacciones con recepción implícita de mensajes son siempre asimétricas.

(b) *Indirecta*: se utiliza una expresión evaluada en tiempo de ejecución. Para ello es necesario utilizar un objeto intermedio, normalmente denominado buzón (*mailbox*), al cual dirige el emisor el mensaje, y del cual escucha el receptor. Puede haber varios receptores escuchando al mismo buzón.

Veamos a continuación varios modelos de comunicación basados en el paso de mensajes que han sido utilizados en diversos lenguajes.

1. **Modelo 1: Transmisión punto a punto.**

   Algunos lenguajes proporcionan primitivas de envío/recepción de mensajes punto a punto. La transmisión punto a punto expresa la comunicación en un sentido (del emisor al receptor), pero en la práctica muchas interacciones entre procesos son, por naturaleza, de dos sentidos. Por ejemplo, en el modelo cliente-servidor, el cliente solicita un servicio del servidor, y espera la contestación del servidor. Este comportamiento puede simularse utilizando dos mensajes punto a punto, pero un mecanismo de un nivel superior facilita la programación y puede ser más eficiente de implementar. Existen básicamente dos tipos:

   - Cita (*rendezvous*).
   - Llamada remota a procedimiento (*RPC*).

2. **Modelo 2: Cita** (*rendezvous*)

   Es un modelo de comunicación síncrono bidireccional que se basa en tres conceptos:

   (a) Declaración de punto de entrada, que es como la declaración de un procedimiento y está compuesta por un nombre y, opcionalmente, algunos parámetros.

   (b) Sentencia de aceptación (*accept*), que especifica las sentencias que se ejecutan cuando se acepta la llamada al punto de entrada.

   (c) Llamada a punto de entrada, que toma la forma de una llamada a procedimiento.

   Las dos primeras partes forman parte del código del servidor, mientras que la tercera pertenece al cliente. La interacción ocurre cuando el emisor llama al punto de entrada y el receptor ejecuta una sentencia de aceptación. Cuando ambos se han sincronizado, el receptor ejecuta el bloque de sentencias asociado al *accept*, pudiendo acceder a los parámetros recibidos en la llamada, y asignar valores a los parámetros de salida. Cuando el receptor finaliza la ejecución del bloque de sentencias asociado al *accept*, ambos procesos continúan su ejecución en paralelo.

3. **Modelo 3: Llamada remota a procedimiento.**

   Modelo síncrono bidireccional en el que la comunicación se realiza aparentemente mediante una llamada a un procedimiento, pero con la diferencia de que la llamada y el cuerpo del procedimiento pueden estar en procesadores diferentes. Cuando se realiza la llamada, el receptor ejecuta el código del procedimiento y retorna cualquier posible información en los parámetros de salida. Durante la ejecución, el llamador se queda bloqueado hasta que se termina la ejecución del procedimiento remoto (mientras que en

Llamada remota a procedimiento versus llamada remota a punto de entrada

Sintácticamente los puntos de entrada y los procedimientos son muy parecidos. Según [AMN88], las principales diferencias son:

- El protocolo subyacente mediante el cual se seleccionan y procesan las llamadas concurrentes (los puntos de entrada suponen sincronización implícita).
- El contexto en el que se ejecuta el código del llamador.

– Ventajas del uso de llamada remota a punto de entrada (cita remota): La llamada remota a punto de entrada tiene las siguientes ventajas [AMN88]:

- La ejecución de una sentencia *accept* se realiza en el flujo de control de la tarea llamada, mientras que la semántica de los procedimientos requiere que se ejecuten en el flujo de control de la tarea llamadora.
- La cita proporciona una visión más clara de la carga de trabajo de cada nodo, permitiendo una distribución más equitativa.
- El programador no tiene que preocuparse por el control de acceso concurrente a las variables globales (problema que existe siempre que se utilizan procedimientos con efectos laterales).

– Ventajas del uso de llamada remota a procedimiento: Su principal ventaja [AMN88] (pág. 19) es que esconde en la interfaz del módulo los detalles de su estructura interna (las tareas que contiene). Esto no es posible con el mecanismo de cita, ya que la tarea del cliente debe nombrar explícitamente la tarea del servidor, así como la *entry*. Como resultado, el programa fuente de los clientes contiene información de las tareas de que constan los servidores que utiliza. Esto significa que un cambio en la estructura interna de un servidor nos obliga a modificar todos sus clientes.

Haciendo que la interfaz de los servicios remotos sea puramente procedimental se evita este problema. Los resultados de esta decisión no siempre son positivos ya que:

- A menudo es importante para un cliente conocer la política de planificación de un servidor con respecto a las operaciones que proporciona.
- Los clientes pierden la posibilidad de hacer llamadas condicionales y temporizadas. Es posible obtener un efecto similar introduciendo parámetros adicionales que indiquen un *time-out* o la condición de éxito de una llamada, pero esto pasa la responsabilidad de medir el *time-out* al servidor.

En resumen, no existe un argumento decisivo a favor del uso de llamada a punto de entrada o a procedimiento para soportar la comunicación entre los nodos.

la cita, el emisor es desbloqueado desde que se termina la ejecución de la sentencia de aceptación). La aceptación de llamadas remotas suele ser implícita (aunque no siempre), y como resultado se crea un nuevo flujo de control dentro del receptor.

4. **Modelo 4: Radiado.**

   Muchas redes utilizadas por los sistemas de computación distribuidos tienen mecanismos que permiten la transmisión de mensajes de uno a muchos —radiado (*broadcast*) o radiado selectivo (*multicast*)—. Con el radiado se envía un mensaje a todos los procesadores de la red; con el radiado selectivo se envía a un determinado subconjunto de dichos procesadores. Desgraciadamente, la red no garantiza que el mensaje llegue a todos los destinos.

   La comunicación uno a muchos tiene varias ventajas sobre la comunicación uno a uno, tales como:

   - En muchas de las redes locales más populares se necesita aproximadamente el mismo tiempo para realizar un radiado (sea selectivo o no), que para enviar un mensaje a un determinado procesador.
   - Una primitiva de radiado puede garantizar un cierto orden de mensajes que no puede obtenerse fácilmente mediante mensajes uno a uno [BJ87]. Por ejemplo, una primitiva de radiado que entregue los mensajes a todos los destinos en el mismo orden es muy útil para una actualización consistente de datos replicados [JK86] [BT88].
   - El radiado puede favorecer nuevos estilos de programación.

   El radiado y el radiado selectivo son útiles para implementar algoritmos distribuidos [Dol93], por lo que es interesante que los lenguajes para sistemas distribuidos proporcionen algún tipo de soporte para este mecanismo. StarMod [LC85] y BSP [Geh84] son lenguajes diseñados para ejecutar aplicaciones distribuidas sobre redes que proporcionan soporte de radiado.

b) Comunicación mediante datos compartidos

A primera vista puede parecer poco natural utilizar datos compartidos para la comunicación entre los procesadores de un sistema distribuido, ya que dichos sistemas no tienen memoria física compartida. Sin embargo, en la literatura existen numerosos ejemplos de aplicaciones distribuidas y algoritmos que se benefician mucho del soporte de datos compartidos, incluso si no existe memoria física compartida [SZ90, NL91].

El paradigma de datos compartidos tiene las siguientes ventajas sobre el paradigma de paso de mensajes [BT88]:

- Los datos compartidos están accesibles a cualquier proceso, mientras que un mensaje siempre transmite información entre dos procesos específicos,
- La asignación sobre datos compartidos tiene conceptualmente un efecto inmediato; en contraste, existe un retraso medible entre el envío y la recepción de un mensaje.

Sin embargo, la memoria compartida tiene como principal desventaja que requiere tomar precauciones para prevenir que múltiples procesos intenten simultáneamente modificar la misma variable.

Las variables compartidas simples no son apropiadas para los sistemas distribuidos. En la práctica ninguno de los lenguajes conocidos las utilizan (posiblemente debido a consideraciones de rendimiento). Sin embargo, existen otros modelos de comunicación basados en datos compartidos que son más apropiados para sistemas distribuidos. Estos modelos imponen ciertas restricciones a los datos compartidos, haciendo que la implementación sea más sencilla y eficiente. Entre estos modelos podemos resaltar el de las estructuras de datos compartidas (*Espacio de tuplas*), utilizado en el lenguaje *Linda* [ACD86] y el modelo de objetos de datos compartidos, utilizado en el lenguaje *Orca* [BKT92a, LKBT92].

1. **Modelo de estructuras de datos compartidas** Las estructuras de datos compartidas (*Espacio de tuplas*) son conceptualmente una memoria compartida (aunque su implementación no requiere memoria física compartida [D.85]). Sus elementos se denominan *tuplas*.

   Al contrario que las variables compartidas normales, las tuplas no tienen direcciones, sino que se direccionan por su contenido. La forma de especificar una tupla es mediante el valor o el tipo de cada uno de sus campos; una tupla es encontrada con éxito si tiene el valor o el tipo pasado como parámetro en la operación. En caso de que haya varias tuplas candidatas, se elige una de forma arbitraria. Si no hay tupla candidata, la operación se bloquea hasta que otro proceso añada una tupla candidata.

   Para modificar una tupla, debe extraerse, modificarse, y finalmente insertarse. Para ello se dispone de tres operaciones: **Read** para leer una tupla, **In** para leer y extraer una tupla y **Out** para añadir una nueva tupla. Como estas operaciones son atómicas, el resultado de realizar varias operaciones simultáneamente sobre la misma tupla es equivalente a ejecutarlas en algún orden secuencial (no definido). En particular, si dos procesos intentan extraer la misma tupla, solamente uno de ellos lo conseguirá, y el otro se bloqueará.

2. **Modelo de objetos de datos compartidos**

   El modelo de comunicación de Orca [BKT92a] se basa en objetos de datos compartidos (*shared data object model)*, con las siguientes características:

   - Las estructuras de datos compartidas se encapsulan en objetos de datos pasivos, que se manipulan mediante operaciones definidas por el usuario.
   - Las estructuras de datos compartidas constan internamente de dos partes: una parte de especificación y una parte de implementación.
   - Aunque los objetos son lógicamente compartidos por los procesos, su implementación no necesita memoria física compartida. En el peor caso, una operación sobre un objeto ubicado en un procesador remoto puede implementarse mediante una llamada remota. La idea general es que la implementación gestione la distribución física de los objetos entre los procesadores, y reduzca el coste de acceso tanto como sea posible.

Los objetos de datos compartidos de Orca son básicamente tipos abstractos de datos que definen operaciones sobre datos, pero esconden la implementación real de la estructura de datos y las operaciones.

1.1.3 Tolerancia a fallos

Decimos que un sistema es tolerante a fallos cuando, a pesar de que fallen algunas partes del sistema, el comportamiento del sistema es consistente con su especificación [Jal94].

El problema principal al que se enfrenta un sistema distribuido cuando falla un nodo de la red es que pierde parte de su estado de ejecución, porque lo que está almacenado en la memoria principal de dicho nodo desaparece. El mecanismo básico para evitar este problema es la *memoria estable*, con la cual, en caso de fallo, puede reejecutarse el programa desde un estado consistente. Sin embargo, para que la recuperación frente a fallos sea posible es necesario que el programa no tenga efectos laterales (ya que al reejecutarse podría alcanzar un estado inconsistente). Para evitar los efectos laterales surgieron las *transacciones atómicas*, mecanismo que permite especificar que una operación se ejecute hasta su finalización, o no tenga ningún efecto. Las transacciones atómicas han sido utilizadas en los lenguajes Emerald [RTL+91] e InterBase [CEB92].

Memoria estable

La memoria estable [Lam81] se implementa mediante memoria no volátil, p. ej. discos. El método consiste en replicar la información, esto es, cada bloque lógico se almacena en dos bloques físicos. Esta replicación puede hacerse sobre discos diferentes (*discos espejo*) o en zonas diferentes del disco que tengan probabilidad de fallo común mínima (p. ej. en cilindros, pistas y superficies diferentes).

La forma de recuperación de fallos es la siguiente: Se examina cada uno de los pares de bloques físicos. Si ambos son iguales y no se detectan errores (examinando el *checksum*), no se toma ninguna medida. Si un bloque contiene un error detectable (por degradación o por escritura a medias), se sustituye su contenido con el valor del otro bloque. Si ninguno de los dos bloques contiene errores detectables, pero difieren en el contenido (esto significa que se pudo realizar la escritura en uno de ellos pero no en el otro), entonces se reemplaza el contenido de uno de ellos (cualquiera) con el valor del otro. Este método de recuperación asegura que una escritura en memoria estable, bien tiene éxito completamente, o bien no produce ningún cambio. El intento de escribir en memoria estable tiene éxito sólo si se escriben las dos copias.

Transacciones atómicas

Todo programa distribuido puede verse como un conjunto de procesos paralelos que están realizando operaciones sobre objetos de datos. Las operaciones que afectan a objetos almacenados en memoria secundaria se convierten en permanentes una vez que se han realizado. A veces este comportamiento no es deseable (si el proceso que está accediendo a varios datos interrelacionados muere, deja la base de datos en un estado inconsistente). La solución a éste problema consiste en agrupar las operaciones en *transacciones atómicas* (también denominadas *acciones atómicas* o simplemente *transacciones*). Un grupo de operaciones es atómico si tiene las siguientes propiedades:

- **Indivisibilidad**. Una transacción es indivisible si, vista desde el exterior, no tiene estados intermedios (desde el exterior se ve que se ejecutan todas sus operaciones o no se ejecuta ninguna). Su implementación puede hacerse:
  - Serializando la ejecución de todas las transacciones atómicas. Esto tiene el inconveniente de restringir severamente el paralelismo, degradando el rendimiento.
  - Sincronizando los procesos mediante cerrojos (*locks*).
- **Recuperabilidad**. Una transacción es recuperable si todos los objetos involucrados pueden restaurarse a su estado inicial en caso de que la transacción falle. Esto puede obtenerse, por ejemplo, mediante actualizaciones en varias copias (*versiones*) [Lam81].

Las transacciones atómicas surgieron en el entorno de las bases de datos, pero los lenguajes de programación puede proporcionar abstracciones muy convenientes para objetos de datos e invocaciones de transacciones atómicas, ya que el sistema de tiempo de ejecución puede encargarse de gestionar automáticamente los cerrojos y las versiones de las copias [LS83, Lis88].

En algunos sistemas no es suficiente sólo recuperar el estado; es necesario además que el sistema continúe proporcionando el servicio sin ninguna interrupción. Para ello suelen utilizarse técnicas de replicación de procesos (redundancia activa) sobre hardware con independencia de fallo [Sch90, BKT92b, TKB92].

Las dos arquitecturas básicas para programar aplicaciones tolerantes a fallos hardware mediante replicación son primario con respaldo(s) (*primary-backup* [BMST92]) y redundancia modular (*modular redundancy*). En el esquema de primario con respaldo solamente un componente (el primario) funciona normalmente, y en caso de que falle, uno de los respaldos se activa y sustituye al primario. En el esquema de redundancia modular, también denominado en la literatura enfoque de máquina de estados [Sch90], todos los componentes (dos o más) realizan la misma función, y existe algún mecanismo de votación en sus salidas para filtrar los fallos (ver figura 1.1).

Desde el punto de vista del lenguaje, existen básicamente tres formas de proporcionar tolerancia a fallos [CGR] [BST89]:

1. Proporcionar algún mecanismo que notifique los fallos de los procesadores. Esta solución es la más sencilla en cuanto al diseño del lenguaje y pasa todo el problema al programador. Una forma de notificar el fallo es retornar un error a todos los procesos que intenten comunicarse con un procesador fallido. Otra forma es permitir que el programador defina una rutina de manejo del error, y que esta rutina sea llamada automáticamente cuando se detecta el fallo de algún proceso o procesador. El principal inconveniente

Máquinas de estados

El enfoque de máquina de estados es un método general para implementar servicios tolerantes a fallos en sistemas distribuidos[a]. Una máquina de estados consta de **variables de estado**, que codifican su estado, y **comandos**, que transforman su estado. Cada comando se implementa mediante un programa determinista; la ejecución del comando es indivisible con respecto a otros comandos, y modifica las variables de estado y/o produce alguna salida.

Los clientes de la máquina de estados le hacen peticiones para que ejecute comandos. La petición contiene: el nombre de la máquina de estado, el comando a ejecutar y cualquier información adicional necesaria para la ejecución del comando. La salida de la ejecución de la máquina de estados puede enviarse a: un actuador (ej. en un sistema de control de procesos), algún periférico (ej. un disco o un terminal) o a clientes que esperan respuestas de peticiones anteriores. La máquina de estados sólo procesa una petición al tiempo, en un orden que es consistente con la causalidad potencial. Además, los clientes de la máquina de estados pueden hacer las siguientes suposiciones acerca del orden en que se procesan las peticiones:

- Las peticiones emitidas por un cliente a la máquina de estados *sm* se procesan en *sm* en el orden en que fueron emitidas.
- Si la petición r hecha por el cliente c a la máquina de estados *sm* pudo causar que el cliente c' hiciese una petición r', entonces *sm* procesa r antes que r'.

Las salidas de la máquina de estados están completamente determinadas por la secuencia de peticiones que procesan, independientemente del tiempo y de otra actividad en el sistema.

Máquinas de estado tolerantes a fallos

La versión tolerante a t fallos de la máquina de estados puede implementarse replicando dicha máquina de estados y ejecutando una réplica en cada uno de los procesadores de un sistema distribuido. Dado que cada réplica comienza en el mismo estado inicial y ejecuta las mismas peticiones en el mismo orden, todas hacen lo mismo y producen las mismas salidas. Así, si suponemos que cada fallo puede afectar como máximo a un procesador, y por tanto a una réplica de la máquina de estados, combinando la salida de las réplicas podemos obtener la salida de la máquina de estados tolerante a t fallos.

Cuando los procesadores pueden tener fallos bizantinos, el conjunto que implementa una máquina de estados tolerante a t fallos debe tener al menos $2t+1$ réplicas, y la salida del conjunto es la salida producida por la mayoría de las réplicas. Esto es así porque con $2t+1$ réplicas, la mayoría de las salidas continúa siendo correcta incluso después de t fallos. Si los procesadores solamente pueden sufrir fallos tipo "fallo-parada", entonces es suficiente que el conjunto tenga $t+1$ réplicas, y la salida del conjunto puede ser la salida producida por cualquiera de sus miembros. Esto es así porque los procesadores de fallo-parada solamente producen salidas correctas, y tras t fallos seguirá quedando aún una réplica viva.

La clave para implementar una máquina de estados tolerante a t fallos consiste en asegurar **coordinación de réplicas**. Esto significa que todas las réplicas reciben y procesan la misma secuencia de peticiones. Esto puede descomponerse en dos requisitos relacionados con la propagación de peticiones a las réplicas de un conjunto:

- **Acuerdo:** Toda máquina de estados correcta (*non-faulty*) recibe todas las peticiones.
- **Orden:** Toda máquina de estados correcta procesa las peticiones que recibe en el mismo orden relativo.

El requisito de acuerdo regula el comportamiento de un cliente interactuando con las réplicas de la máquina de estado, y el requisito de orden regula el comportamiento de la máquina de estados con respecto a peticiones de varios clientes. Aunque la coordinación de réplicas podría descomponerse de otras formas, la descomposición acuerdo-orden es una elección natural, ya que se corresponde con la separación existente del cliente con respecto a las réplicas de la máquina de estados.

[a]Este resumen ha sido extraído del artículo [Sch90].

Figura 1.1: Máquinas de estado

de este enfoque es que, en general, los programadores tienen que escribir cantidades significativas de código extra para hacer sus aplicaciones tolerantes a fallos.

2. Proporcionar mecanismos de alto nivel para expresar qué procesos y datos son importantes y cómo deben recuperarse cuando ocurra un fallo. Ejemplos de lenguajes que utilizan este enfoque son Emerald [RTL+91], que proporciona soporte para transacciones atómicas, y C concurrente tolerante a fallos [CGR, AG], que proporciona soporte para implementar procesos replicados.

3. Enfoque transparente, en el cual el sistema operativo o el entorno de programación se encarga de proporcionar tolerancia a fallos de forma totalmente transparente a la aplicación. Aunque este enfoque suele ser ineficiente[7], ha sido utilizado en el lenguaje NIL [SY83] mediante una técnica denominada *recuperación optimista* [SY85], y en el lenguaje Orca [KMBT92]).

1.2 Configuración del sistema

Una de las formas más efectivas de apoyar la flexibilidad de configuración de los sistemas distribuidos es descomponer la construcción del programa distribuido en dos fases diferentes y separadas:

1. **Fase de programación de componentes**: en esta primera fase el programador define los componentes distribuibles con los que se configurará el sistema final. En este punto el diseñador solamente debe preocuparse por la función y estructura de los componentes distribuibles, y la forma en que interactúan.

2. **Fase de configuración del sistema**: en esta segunda fase, mediante algún tipo de lenguaje de especificación de la configuración, el programador especifica qué nodos software se necesitan, y cómo deben distribuirse entre los nodos físicos de la red.

La separación de estas fases permite desarrollar las aplicaciones sin preocuparnos de ninguna configuración lógica o física. Maximizando esta independencia se consigue que el lenguaje soporte un mayor número de aplicaciones. Por ello es muy importante que los componentes distribuibles conozcan la menor cantidad de información posible sobre la configuración, y que todas las decisiones de configuración puedan aplazarse hasta la fase de configuración. C-concurente [GR89], SR [And81], Argus [Lis88] y Orca [BKT92a] permiten asignar los procesos a procesadores específicos, mientras que NIL [SY83], Linda [ACD86], Ada 95 [mt93] ignoran este problema o se lo dejan a la implementación.

[7] Porque, sin un conocimiento específico de la aplicación, el compilador no puede distinguir entre aquellos procesos (o datos) que son vitales a la aplicación, y los que son menos importantes.

1.3 Programación con grupos

Los sistemas distribuidos suelen programarse mediante el paradigma cliente/servidor [Tan92]. De acuerdo con este paradigma, un ente (el cliente) solicita servicios a otro ente (el servidor), que puede estar ubicado remotamente. Con el crecimiento actual de las redes de ordenadores y la correspondiente descentralización de actividades, se ha desarrollado una extensión de este paradigma, que permite tratar con la multiplicidad de nodos del sistema distribuido: el paradigma de comunicación con grupos [VRV92, VR92].

Un grupo es una agrupación de objetos que comparten una semántica de aplicación común [LCN90]. La abstracción de grupo permite al programador ver un conjunto de objetos como si fuesen una entidad lógica individual. El grupo no permite a sus usuarios ver su estructura interna ni las interacciones entre sus miembros. Los grupos se utilizan generalmente para [LCN90]:

1. Abstraer las características comunes de sus miembros y los servicios que proporcionan. El grupo proporciona una interfaz externa común a todos ellos.

2. Encapsular el estado interno y esconder las interacciones entre los miembros del grupo.

3. Construir objetos mayores utilizando el grupo como un bloque básico de composición.

La comunicación entre los clientes externos y un grupo servidor se denomina comunicación intergrupo; la comunicación interna entre los miembros del grupo se denomina comunicación intragrupo. Decimos que un grupo es cerrado cuando solamente posee comunicación intragrupo (solamente sus miembros pueden comunicarse); en caso contrario decimos que es abierto. El modelo abierto es más general y se corresponde con el modelo cliente-servidor utilizado en los sistemas operativos distribuidos. El modelo cliente-servidor puede extenderse fácilmente a comunicación con grupos haciendo que el cliente y el servidor sean grupos. La comunicación con grupos abiertos puede utilizarse para implementar servicios tolerantes a fallos [KTV93b, RG92].

Decimos que un grupo es estático cuando tras su creación no permite que se incorporen nuevos miembros; en caso contrario decimos que el grupo es dinámico [LM94].

Desde hace varios años se han desarrollado algunos proyectos de investigación que proporcionan soporte de grupos, tales como V-kernel [CZ85], Circus [Coo85], Isis [BCJ+90, Bir93a], Amoeba [KTV93a, KT94] y Horus [RBC+93, RTB93] [8]. Recientemente se han realizado estudios para introducir seguridad en grupos [RBG92] y para gestionar comunicación con grupos a gran escala [BS94].

Con el objetivo de profundizar en el desarrollo de dicha tecnología, hemos centrado esta tesis doctoral en el diseño de un lenguaje que facilite la programación de aplicaciones distribuidas cooperativas y tolerantes a fallos mediante el paradigma de programación con grupos.

[8] Horus incropora toda la experiencia de Isis. Es un rediseño de Isis con el objetivo de hacer el sistema más simple y rápido [GBRR92].

Isis

Isis [BCJ+90] es un conjunto de herramientas que permiten programar aplicaciones distribuidas dinámicas[a] tolerantes a fallos. La unidad básica de distribución de Isis es el *proceso*. Los procesos puede agruparse para constituir una unidad de abstracción mayor que es el *grupo*. Los grupos Isis son dinámicos (pueden entrar o salir del grupo nuevos procesos durante el tiempo de existencia del grupo). Esto permite que el grupo que implementa un servicio pueda crecer, reducirse, moverse a diferentes máquinas, o añadir nuevas capacidades sin interrumpir nunca el servicio y sin que los usuarios noten dichos cambios. Los procesos Isis pueden ser miembros de varios grupos simultáneamente. Veamos brevemente cómo es el paralelismo, la comunicación, la sincronización y la tolerancia a fallos en Isis.

- **Paralelismo**. Los procesos Isis constan internamente de varias tareas. Una tarea Isis tiene la misma apariencia que una función C (compartiendo el mismo espacio de direcciones y variables globales que el resto de las tareas y funciones del proceso). La diferencia es que las tareas Isis pueden ser invocadas por el sistema y comenzar su ejecución en respuesta a ciertos eventos. El más común es la recepción de un mensaje. Las tareas que pueden activarse en respuesta a la entrega de un mensaje se llaman puntos de entrada (*entries*).

  La planificación de Isis es estrictamente *fifo* y *no expulsiva*[b]. Sin embargo, Isis permite que una tarea abandone el entorno Isis y comience a ejecutarse concurrentemente junto con otras tareas activas[c]. En caso de que dicha tarea intente entrar en el entorno Isis, se verá forzada a bloquearse hasta que el planificador de Isis le permita continuar.

- **Comunicación**. El mecanismo básico de comunicación entre procesos de Isis es el radiado de un mensaje a un grupo. Todos los miembros del grupo reciben una copia del mensaje cumpliendo las propiedades del *entorno virtualmente síncrono* de Isis. Estas propiedades son las siguientes:

  - **Ordenación global de eventos**: Todos los procesos observan los mismos eventos[d] y en el mismo orden.
  - **Causalidad**: Cuando un evento precede causalmente a otro, es observado en todo el sistema antes que aquel al cual precede.
  - **Atomicidad**: La notificación de eventos se realiza a todos los miembros de un grupo o a ninguno.

  Un proceso que envía un mensaje a un grupo puede indicar que desea esperar por un número determinado de respuestas. Un proceso que recibe un mensaje puede enviar una respuesta o reenviarlo a otro proceso (que se encargará entonces de enviar la respuesta al que envió originalmente el mensaje).

  Un proceso puede solicitar que Isis le notifique determinados eventos, tales como cambios en los miembros de grupos de procesos, terminación de procesos y fallos de máquinas. Esto puede utilizarse para repartir la carga del sistema entre los miembros vivos del grupo (tanto en caso de incorporación de nuevos miembros, como en el caso de abandono de alguno o fallo de algún nodo).

- **Sincronización**. Isis proporciona un mecanismo para que las tareas esperen por determinados eventos, así como mecanismos para que una tarea avise a otras tareas que ha ocurrido un determinado evento. Dichos mecanismos utilizan las variables condición (*condition*), que son básicamente una cola en la cual se colocan las tareas que están esperando por un determinado evento.
- **Tolerancia a fallos**. Isis proporciona soporte para implementar aplicaciones tolerantes a fallos mediante el modelo de redundancia modular y el modelo de primario con respaldo(s).

[a] Permite reconfigurar la aplicación en tiempo de ejecución.

[b] Excepto cuando se utiliza el mecanismo de notificación urgente de Isis.

[c] Esto permite explotar el paralelismo real del sistema.

[d] Los eventos Isis ocurren cuando: una máquina cae o se recupera; se crea un grupo de procesos o cambia el número de miembros de un grupo; se entrega un mensaje.

1.4 Resumen

Los lenguajes diseñados para programar sistemas distribuidos deben tener en cuenta tres aspectos fundamentales que diferencian a la programación distribuida de la secuencial: paralelismo (permitir ejecutar partes de un programa distribuido en procesadores diferentes), comunicación y sincronización (proporcionar mecanismos de cooperación entre las unidades distribuibles), y tolerancia a fallos (proporcionar mecanismos para soportar fallos parciales del sistema).

Se han considerado los *procesos* y los *objetos* como unidades de abstracción convenientes para sistemas distribuidos débilmente acoplados. Se han descrito los modelos existentes para comunicación y sincronización entre unidades distribuidas (modelos basados en paso de mensajes y modelos basados en memoria compartida). También se han descrito mecanismos de tolerancia a fallos (memoria estable, transacciones atómicas y replicación).

Se ha visto que una de las formas más efectivas de apoyar la flexibilidad de configuración, consiste en descomponer la construcción del programa distribuido en dos fases diferentes: una fase de programación de componentes y una fase de configuración del sistema.

Finalmente se ha presentado el paradigma de comunicación con grupos, paradigma utilizado desde hace varios años para programar sistemas distribuidos. Para profundizar en el desarrollo de dicha tecnología, hemos centrado esta tesis en el diseño de un lenguaje que facilite la programación de aplicaciones distribuidas mediante el paradigma de programación con grupos.

Capítulo 2

Drago

Drago es un lenguaje experimental, diseñado para facilitar la implementación de aplicaciones distribuidas tolerantes a fallos y cooperativas, mediante grupos abiertos estáticos. Es el resultado de varios años de diseño, en los que hemos desarrollado dos versiones, que describimos brevemente a continuación. La segunda versión se presenta de manera completa en el apéndice A.

Durante el desarrollo de la primera versión analizamos las interacciones entre grupos y realizamos una clasificación de grupos atendiendo a su comportamiento externo. El resultado fueron dos tipos de grupos: grupos replicados y grupos cooperativos. Denominamos *agente* a la unidad de distribución de Drago y consideramos conveniente proporcionar dos tipos de agentes: agente replicado, para implementar grupos replicados, y agente cooperativo, para implementar grupos cooperativos. La diferencia principal entre ellos es que, para asegurar la consistencia de las réplicas, un agente replicado no puede contener tareas.

Decidimos utilizar la llamada remota a procedimiento [BN84] como paradigma básico de comunicación remota. Esto fue debido a que es el mecanismo básico que proporciona Ada 95 [mt93, xmt93] para comunicación remota (aunque Ada 95 también soporta acceso remoto a objetos), y no queríamos que Drago se alejase demasiado de la línea de evolución actual de Ada. Para tratar con grupos desarrollamos una extensión de la llamada remota a procedimiento (una llamada remota a todos los miembros de un grupo). Sin embargo, llegamos a la conclusión de que si la llamada remota a procedimiento es reentrante, no es posible asegurar que todas las réplicas pasen por los mismos estados. Para resolver este problema decidimos entonces que los agentes replicados tuviesen una semántica de ejecución similar a la de los objetos protegidos de Ada 95 [mt93], e implementamos las operaciones remotas mediante puntos de entrada remotos.

Para aumentar el paralelismo de los agentes replicados, decidimos que las llamadas remotas se ejecutasen mediante un planificador por petición, y que los puntos de comunicación remota fuesen puntos de planificación. Esto permite que un servidor replicado pueda continuar proporcionando servicio mientras espera los parámetros de salida de una llamada remota suya. Esta decisión alejó a Drago de Ada, ya que Ada no presupone si el planificador de tareas es expulsivo o no.

Para mantener la homogeneidad del lenguaje, conservamos en los agentes cooperativos la semántica de ejecución de los agentes replicados, pero permitimos que tuviesen tareas Ada. Esto permite que los programas Drago reutilicen código escrito mediante tareas Ada. Como Ada no impone el tipo de planificador de las tareas, y necesitábamos que las llamadas remotas se ejecutasen mediante un planificador por petición, decidimos que los agentes cooperativos tuviesen internamente dos planificadores: un planificador expulsivo que ejecuta los flujos de control que atienden las llamadas remotas, y el planificador de Ada que ejecuta las tareas Ada.

Utilizamos esta primera versión de Drago para programar ejemplos de aplicaciones tolerantes a fallos y aplicaciones cooperativas. Estos ejemplos sirvieron para concretar detalles semánticos de Drago. Sin embargo, la complejidad del lenguaje, debido principalmente al modelo de dos planificadores en los agentes cooperativos, dificultó la redacción del manual de referencia de esta versión del lenguaje y su implementación mediante un preprocesador que generase código Ada 83 [ANS83].

En la segunda versión, acercamos la semántica de ejecución de los agentes a la semántica de ejecución de las tareas Ada. Para ello consideramos conveniente que el mecanismo de comunicación remota fuese una extensión de la cita entre tareas; así surgió la cita remota con un grupo. Para que los agentes pudiesen aceptar citas remotas diseñamos extensiones de las sentencias Ada de aceptación de cita. Reescribimos con esta versión los ejemplos desarrollados para la versión anterior de Drago, y llegamos a la conclusión de que el lenguaje era más sencillo, claro y, generalmente, más expresivo que el lenguaje de la versión anterior. Sin embargo, perdía una de las características más notables de la versión anterior: la posibilidad de que un servidor replicado pudiese simultáneamente atender nuevas peticiones mientras esperaba los parámetros de salida de una llamada remota suya. Para resolver este problema diseñamos un mecanismo de cita remota asíncrona, la cita remota con reencolamiento. El resultado fue la semántica de ejecución actual del agente replicado. Al igual que en la primera versión, los agentes cooperativos pueden contener tareas Ada, lo que les permite realizar trabajo útil mientras atienden peticiones remotas. Los ejemplos de los capítulos 3 y 4, el modelo de traducción descrito en el capítulo 5 y el manual de referencia del apéndice, se corresponden con esta segunda versión de Drago.

Este capítulo está estructurado en cinco partes. En la primera se describen algunas generalidades de ambas versiones del lenguaje: el modelo de sistema distribuido sobre el que se ejecutan las aplicaciones y los objetivos de diseño del lenguaje. En la segunda parte se presenta el modelo de programación con grupos de Drago. En la tercera se describe la primera versión del lenguaje y en la cuarta se muestra la versión actual de Drago. Finalmente, en la quinta parte se presenta una descripción general de las características que proporciona Drago para programar aplicaciones distribuidas replicadas, así como aplicaciones distribuidas cooperativas.

2.1 Generalidades

En este primer apartado presentamos algunas generalidades comunes a las dos versiones de Drago: el modelo de sistema distribuido sobre el que se ejecutan las aplicaciones Drago y los objetivos de diseño de Drago.

2.1.1 Modelo de sistema distribuido

Las aplicaciones Drago se ejecutan sobre un sistema distribuido con las siguientes características:

- **Procesadores de fallo silencioso.** Cuando falla un procesador no realiza ninguna acción incorrecta: simplemente deja de funcionar sin notificar el fallo. En la literatura se consideran básicamente dos caracterizaciones de este tipo de procesadores: procesadores de *fallo-parada* [SS83], que cuando fallan notifican al resto de los procesadores el fallo antes de detenerse, y procesadores de *fallo-silencioso* [PBS+88], que cuando fallan simplemente dejan de funcionar. Elegimos procesadores de fallo silencioso porque consideramos que es una semántica más general que la de fallo-parada.
- **No hay memoria compartida.** Drago presupone un sistema distribuido en el que no existe memoria compartida entre los nodos. De esta forma, el único mecanismo de comunicación y sincronización entre los nodos es el intercambio de mensajes a través de la red de comunicación.

No hacemos ninguna suposición sobre la topología de la red. La red puede ser una red de área local o puede estar formada por un conjunto de redes de area local conectadas mediante una red troncal.

2.1.2 Objetivos de diseño

Los objetivos de diseño de Drago fueron los siguientes[1]:

- **Expresividad:** Debe facilitar la especificación de paralelismo, comunicación, sincronización y tolerancia a fallos (características fundamentales que diferencian la programación distribuida de la secuencial).
- **Semántica simple:** Debe tener una semántica simple y clara. En particular, debe integrar suavemente las construcciones secuenciales y distribuidas.
- **Fuerte tipado:** Debe ser un lenguaje fuertemente tipado. La programación distribuida, al igual que la programación concurrente, requiere más seguridad que la programación secuencial puesto que:

[1] Estos objetivos son un subconjunto de los objetivos del lenguaje Orca [Bal90, BKT92a]. No hemos considerado el objetivo de *eficiencia* porque Bal considera que el lenguaje debe tener una *eficiencia razonable*, concepto que no consideramos bien definido.

  - Los errores de los programas distribuidos son más difíciles de detectar, incluso con mensajes de error claros.
  - Un error no detectado en parte de un programa puede afectar de forma oscura a la operación de otras partes correctas del mismo programa.

- **Legibilidad:** Debe facilitar la legibilidad de los programas, ya que ésta afecta al mantenimiento de las aplicaciones distribuidas.
- **Portabilidad:** Debe ser implementable en múltiples configuraciones distribuidas, procesadores y sistemas operativos.

Decidimos desarrollar Drago como una extensión de un lenguaje existente que estuviese bien definido y fuese popular, modular y fuertemente tipado. De esta forma centramos nuestro esfuerzo en el desarrollo de las nuevas características del lenguaje (distribución, comunicación y sincronización remota, etc.). Consideramos varios candidatos (Mesa [Xer85], Modula-2 [Wir85], Ada 83 [ANS83]), y se eligió Ada 83 porque:

- Su semántica de parámetros formales (parámetros de modo **in**, **out** o **in out**) se adapta muy bien a la comunicación remota.
- Proporciona soporte de excepciones, mecanismo conveniente para notificar fallos en la comunicación remota [BN84].
- Disponemos de compiladores Ada para varios sistemas operativos y procesadores.

Una vez considerado Ada como lenguaje base, se consideró como objetivo importante **integrar suavemente Drago y Ada 83** para facilitar la reutilización de código Ada 83 en las aplicaciones distribuidas escritas con Drago.

2.2 Programación con grupos

Los sistemas distribuidos suelen programarse mediante el paradigma cliente/servidor [Tan92]. De acuerdo con este paradigma, un ente (el cliente) solicita servicios a otro ente (el servidor), que puede estar ubicado remotamente. La llamada remota a procedimiento [BN84] proporciona un nivel de abstracción suficiente para programar este tipo de aplicaciones distribuidas. Con el crecimiento actual de las redes de ordenadores, se ha desarrollado una extensión de este paradigma, que permite tratar con la multiplicidad de nodos del sistema distribuido: el paradigma de comunicación con grupos.

Desde hace varios años se han desarrollado algunos proyectos de investigación que proporcionan soporte de comunicación con grupos, tales como V-kernel [CZ85], Circus [Coo85], Isis [BCJ+90] y Horus [RBC+93]. Sin embargo, exceptuando C concurrente tolerante a fallos [CGR], no hemos encontrado en la literatura ningún otro lenguaje que proporcione soporte para programar aplicaciones distribuidas en base a grupos. Por esta razón, y con el objetivo de profundizar en el desarrollo de dicha tecnología, se ha diseñado una extensión de Ada 83 que facilita la programación de aplicaciones distribuidas mediante el paradigma de programación con grupos.

2.2.1 Caracterización de un grupo

Un grupo Drago es un agrupación de unidades de distribución de Drago, denominadas agentes, que comparten una semántica de aplicación común. La abstracción de grupo permite al programador ver un conjunto de agentes como si fuesen una entidad lógica individual. El grupo no permite a sus usuarios ver su estructura interna ni las interacciones entre sus miembros.

La comunicación entre los clientes externos y un grupo servidor se denomina comunicación **intergrupo**; la comunicación interna entre los miembros del grupo se denomina comunicación **intragrupo** (ver figura 2.1). Se dice que un grupo es **cerrado** cuando solamente posee comunicación intragrupo (solamente sus miembros pueden comunicarse); en caso contrario decimos que es **abierto**. El modelo abierto es más general y se corresponde con el modelo cliente-servidor utilizado en los sistemas operativos distribuidos. El modelo cliente-servidor puede extenderse fácilmente a comunicación con grupos haciendo que el cliente y el servidor sean grupos.

Decimos que un grupo es **estático** cuando tras su creación no permite que se incorporen nuevos miembros en el grupo; en caso contrario decimos que es **dinámico**. Drago está diseñado para programar aplicaciones distribuidas utilizando grupos abiertos estáticos.

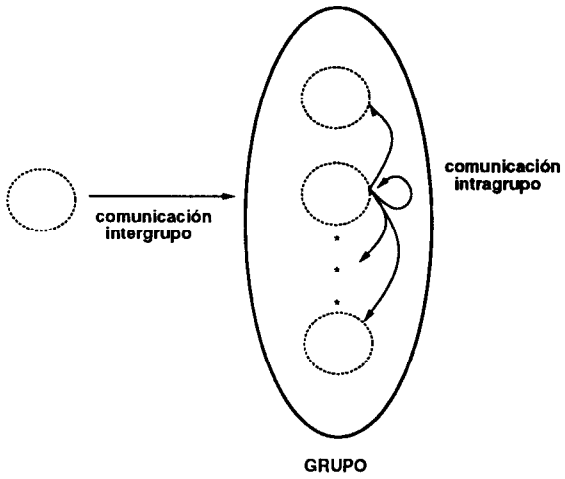


Figura 2.1: Tipos de comunicación de un grupo

2.2.2 Tipos de grupos

Hemos analizado los grupos atendiendo a su comportamiento externo y los hemos clasificado en dos tipos: grupos **replicados** y grupos **cooperativos**. [LCN90] realiza esta clasificación en términos de grupos deterministas y grupos no-deterministas, pero no hemos considerado conveniente su terminología porque los miembros de un grupo determinista pueden no ser réplicas.

Los grupos replicados están compuestos por réplicas idénticas y se utilizan para tolerar fallos mediante el enfoque de *redundancia modular* [Sch90]. La idea base tras este enfoque de tolerancia a fallos consiste en escribir una aplicación mediante un autómata finito determinista,

colocar varias réplicas de este autómata en nodos diferentes del sistema distribuido que tengan probabilidades de fallo independientes y alimentar todas las réplicas con los mismos mensajes y en el mismo orden. Como el código que implementa las operaciones remotas del autómata es determinista, todas las réplicas pasan por los mismos estados y producen la misma secuencia de salida (el grupo se comporta como si estuviese compuesto por un único miembro). De esta forma, el programa continúa disponible con completa funcionalidad mientras quede al menos una réplica viva (proporcionando una alta disponibilidad de servicio).

Los grupos cooperativos se utilizan principalmente para distribuir datos y carga de trabajo entre los miembros del grupo. Los miembros cooperan de alguna forma definida por la semántica de la aplicación, para conseguir un objetivo común.

Drago podría proporcionar una única abstracción de grupo (como hace Isis [BCJ+90]) y dejar que el programador la utilice para implementar el tipo de grupo que desee. Sin embargo, hemos analizado las interacciones de estos tipos de grupos y hemos encontrado que la diferencia semántica de ambos tipos hace que requieran un soporte de comunicación diferente y una semántica de ejecución de sus miembros diferente. Veámoslo en detalle. Para los clientes, la interacción con ambos tipos de grupo es diferente puesto que:

- Cuando invocan una operación de un grupo replicado, solamente esperan una respuesta (ver figura 2.2). Como todas las respuestas que reciben son idénticas (porque se supone que todas las réplicas son idénticas y se ejecutan sobre procesadores de *fallo silencioso*), los clientes deben descartar todas las respuestas recibidas menos una.

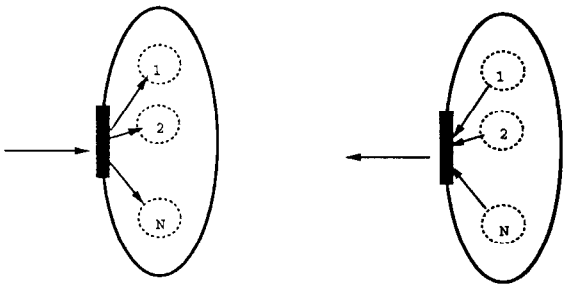


Figura 2.2: Interacción con un grupo replicado

- Cuando invocan una operación de un grupo cooperativo, el número de respuestas que deben tratar depende de la semántica de la aplicación (cada miembro del grupo puede proporcionar una respuesta diferente —ver figura 2.3—).

También, para los miembros de un grupo, su interacción con otros grupos es diferente dependiendo de si pertenecen a un grupo replicado o no:

- Si son miembros de un grupo replicado, deben realizar algún tipo de acuerdo para que del grupo solamente salga una petición al grupo destino (ya que se supone que deben comportarse como un único miembro). Ver figura 2.4.

- Si son miembros de un grupo cooperativo, cada miembro puede tener unos requisitos de comunicación diferentes de los del resto de los miembros. Por esta razón, cada miembro

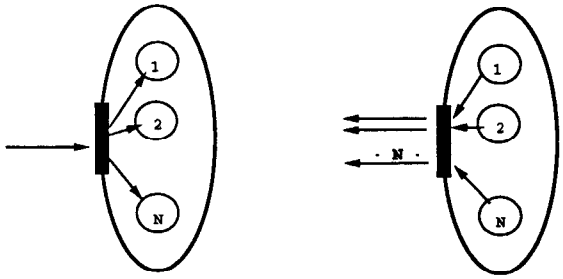


Figura 2.3: Interacción con un grupo cooperativo

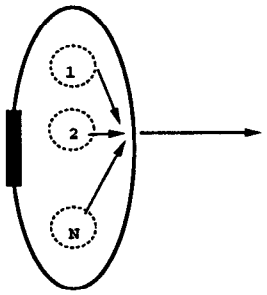


Figura 2.4: Interacción de un grupo replicado con el exterior

realiza su interacción con otros grupos de forma totalmente independiente (ver figura 2.5).

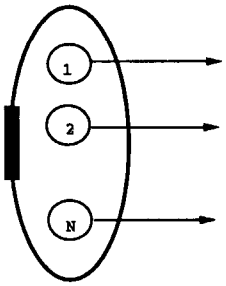


Figura 2.5: Interacción de un grupo cooperativo con el exterior

La interacción entre los miembros del grupo también es diferente en ambos tipos de grupos:

- Los miembros de grupos replicados no necesitan comunicarse entre sí para pasarse información de estado, ya que son réplicas idénticas y por tanto tienen exactamente el mismo estado.
- Los miembros de grupos cooperativos pueden necesitar comunicarse, mediante comunicación intragrupo, para pasar o consultar información de estado al resto de los miembros.

En conclusión, hemos considerado conveniente que Drago proporcione soporte a dos abstracciones de grupo: abstracción de grupo replicado y abstracción de grupo cooperativo.

La abstracción de grupo replicado considera que todos los miembros del grupo son réplicas idénticas que se comportan como un único miembro. La abstracción de grupo cooperativo considera que todos los miembros del grupo son independientes (los miembros solamente comparten la interfaz del grupo).

2.2.3 Características de la comunicación con grupos

Para asegurar que todos los miembros del grupo reciben los mismos mensajes de comunicación remota en el mismo orden, requisito necesario para programar grupos replicados, hemos considerado conveniente que el subsistema de comunicación de Drago entregue los mensajes en todos los miembros de un grupo cumpliendo las siguientes propiedades (una extensión de las propiedades del entorno virtualmente síncrono de Isis:

- **Ordenación global:** Todos los miembros de un grupo observan los mismos mensajes en el mismo orden.
- **Causalidad:** Un mensaje m_2 que fue enviado como resultado de procesar un mensaje anterior m_1 es observado por todos los miembros después de m_1. La causalidad facilita la programación de las aplicaciones puesto que es la forma en que ocurren los hechos cotidianos (la causa precede al efecto [Bir93b]).
- **Atomicidad:** Los mensajes enviados a un grupo son recibidos por todos los miembros del grupo o por ninguno. En particular, si ocurren fallos mientras se está realizando la comunicación remota con un grupo, el mensaje será recibido por todos o ninguno de los miembros *supervivientes* del grupo destino (incluso si el emisor falla).
- **Uniformidad:** Si cualquier miembro del grupo, fallido o no, recibe un mensaje m_1, entonces todos los miembros no fallidos reciben también m_1. Sin uniformidad puede alcanzarse un estado inconsistente si un miembro del grupo modifica el entorno después de recibir un mensaje y después falla (la característica de atomicidad no asegura que el resto de los miembros reciba el mensaje). En general, un protocolo uniforme evita la necesidad de manejar las peticiones remotas iniciadas por los miembros antes de fallar [GAAM93]). Conviene resaltar que el subsistema de comunicación de Isis no tiene esta propiedad.

Estas propiedades permiten a los programadores diseñar sus aplicaciones distribuidas como si fuesen a ejecutarse en un entorno completamente síncrono. Sin embargo, la implementación real se ejecuta de forma asíncrona, explotando así el paralelismo inherente de la arquitectura distribuida [BCJ+90].

2.3 Primera versión

Este apartado describe los elementos de la primera versión de Drago: la interfaz del grupo, las cláusulas de contexto de grupos, la unidad de distribución, los puntos de entrada remotos, la llamada remota a un grupo, y los dos planificadores de los agentes cooperativos.

2.3.1 Descriptor de grupo

La programación de aplicaciones distribuidas mediante el paradigma cliente-servidor requiere interfaces claras que especifiquen los servicios que ofrece cada servidor distribuido. Para fomentar el diseño modular, inicialmente consideramos conveniente que el lenguaje tuviese una unidad de compilación especial en la que se especificasen las interfaces de los servicios remotos, y una unidad de compilación en la que se implementasen dichos servicios (ver figura 2.6).

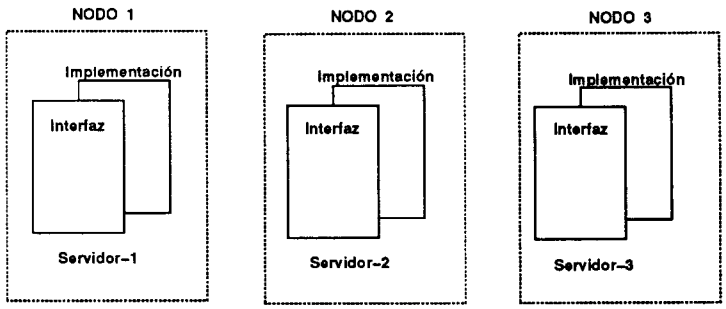


Figura 2.6: Varios servidores remotos con su interfaz correspondiente

Al incorporar en Drago el paradigma de comunicación con grupos, surgió la necesidad de proporcionar soporte linguístico que permitiese especificar los servicios que ofrece el grupo. Para ello añadimos una nueva unidad de compilación: el descriptor de grupo (ver figura 2.7).

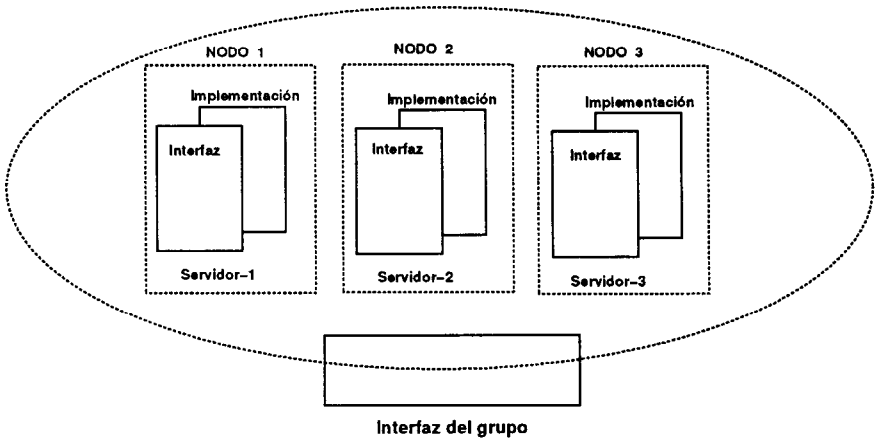


Figura 2.7: Descriptor de grupo (I)

Inicialmente el descriptor de grupo solamente contenía una referencia a los miembros del

grupo. Todos los miembros debían proporcionar la misma interfaz hacia el exterior y el compilador se encargaba de comprobar que las implementaciones contenían todas las operaciones del interfaz. Como esto introducía demasiada redundancia, se consideró conveniente que el descriptor del grupo proporcionase también una única interfaz con los servicios que ofrece el grupo (ver figura 2.8).

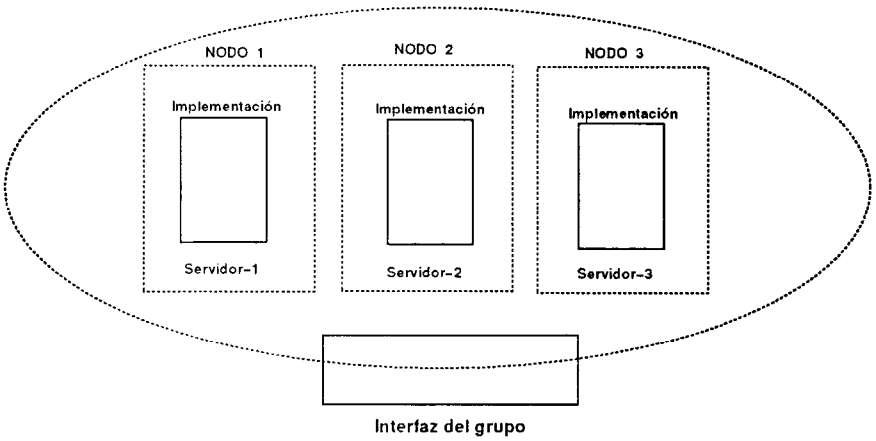


Figura 2.8: Descriptor de grupo (II)

Para proporcionar soporte linguístico a los dos tipos de grupos descritos en el apartado 2.2.2, introdujimos dos tipos de descriptor de grupo: descriptor de grupo replicado y descriptor de grupo cooperativo. Los grupos cooperativos necesitan soporte para que sus miembros se comuniquen sin que los clientes remotos vean la interfaz de esta comunicación. Para proporcionar el control de visibilidad necesario, se consideró conveniente que la interfaz del descriptor de grupo cooperativo constase de dos secciones: una sección que contuviese la interfaz de servicios remotos que ofrece el grupo a los clientes remotos (sección intergrupo), y otra que tuviera la interfaz de comunicación entre los miembros del grupo (sección intragrupo —ver figura 2.9—). El descriptor de grupo replicado no tiene la sección intragrupo porque, al ser réplicas idénticas, sus miembros no necesitan comunicarse.

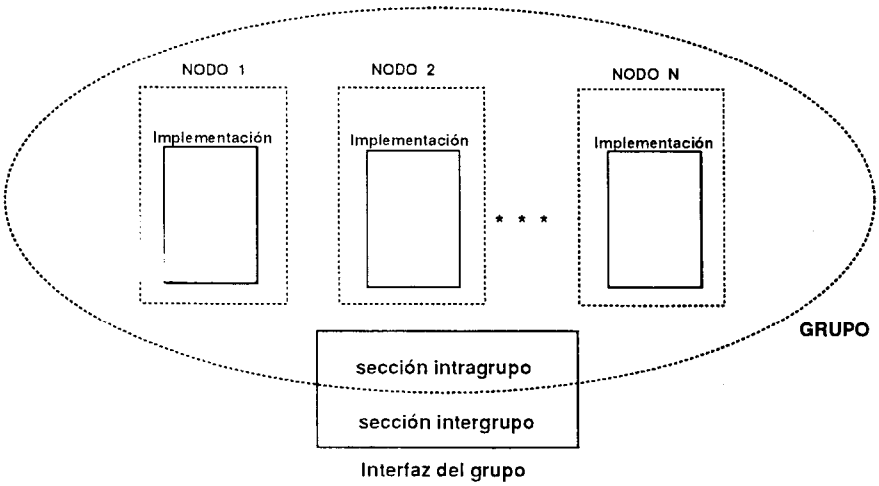


Figura 2.9: Descriptor de grupo (III)

Finalmente se consideró conveniente que el descriptor del grupo no tuviese ninguna información referente al número de miembros del grupo, ni a su ubicación en los nodos de la red. Toda esta información puede especificarse mediante un lenguaje de configuración adicional (cuya definición se ha dejado libre a la implementación). De esta forma, aún siendo un lenguaje para programar con grupos estáticos, conseguimos que tenga un cierto nivel de reconfiguración ya que:

- Sin necesidad de recompilación puede ejecutarse una misma aplicación con una configuración diferente (un número de miembros diferentes, o los miembros ubicados en diferentes nodos de la red).
- En caso de que fallen algunos miembros de un grupo cooperativo, si el grupo es capaz de enmascarar el fallo y la interfaz no especifica el número de miembros, el grupo puede continuar proporcionando su funcionalidad a los clientes remotos.

Interfaz de un grupo cooperativo

El grupo replicado se comporta *como si* estuviese formado por un único miembro. Esto significa que el cliente remoto recibe un único valor en los parámetros de salida de los servicios del grupo. Por ello, la interfaz de los servicios del grupo no refleja la multiplicidad de las réplicas; es, básicamente, la interfaz de los servicios de un único servidor distribuido. Sin embargo, en un grupo cooperativo cada miembro del grupo puede proporcionar una respuesta diferente al cliente remoto. Esto introduce un problema de especificación en la interfaz, ya que cada miembro del grupo necesita que la interfaz del grupo solamente refleje los parámetros de salida que él proporciona, y los clientes remotos necesitan que la interfaz refleje claramente las respuestas de todos los miembros del grupo.

Aunque el lenguaje StarMod [Coo80, LC85] resuelve este problema especificando el servicio igual que si estuviese implementado por un único servidor (ver figura 2.10), consideramos que la correcta especificación de los servicios a los clientes remotos es más importante que la correcta especificación de los servicios a los miembros del grupo. La razón de ello es que el usuario de la aplicación distribuida solamente ve la interfaz de los servicios del grupo (no sabe cómo está implementada).

Como las formaciones (*arrays*) reflejan bien este concepto de varias respuestas de un mismo tipo, decidimos no introducir ningún mecanismo adicional en el lenguaje. Al poder fallar los miembros del grupo cooperativo, la especificación de este tipo de parámetros de salida debe ser variable. Para ello se consideró conveniente su especificación mediante formaciones irrestringidas (formaciones que no especifican el número de elementos que contienen). El siguiente fragmento de programa muestra la especificación de un grupo de dispositivos distribuidos en el que, al invocar la operación remota *Obtener_Estado*, cada miembro del grupo proporciona su propio estado.

StarMod

StarMod es un lenguaje de alto nivel para programación distribuida, derivado de Modula [Wir77] e inspirado en Distributed Processes [Han78] . Al igual que en Modula, el módulo es la unidad básica de StarMod. Un módulo prefijado con la palabra **network** encapsula un programa distribuido que se ejecuta en una red de procesadores sin memoria compartida. Un módulo prefijado con la palabra **processor** describe un procesador lógico y encapsula todos los procesos que se ejecutan en un único procesador. El mecanismo básico de comunicación entre procesos de StarMod son los puertos. Los puertos locales, puertos declarados dentro de un módulo de procesador, permiten la comunicación con un procesador lógico. Los puertos de red [LC85], puertos declarados dentro de un módulo de red, permiten enviar un mensaje a todos los procesadores lógicos de un programa distribuido. Un puerto de red que especifica un valor de retorno espera obtener un valor de cada módulo de procesador del programa distribuido. Por tanto, un puerto de red con un valor de retorno de tipo T, retorna realmente una formación de valores de tipo T. El índice de la formación es un subrango que contiene los nombres de todos los módulos de procesador del programa (que pueden verse como constantes predeclaradas).

El siguiente ejemplo es la implementación de una aplicación distribuida con estructura de estrella:

```
network module RedEstrella;
    const NumeroSatelites = 5;
    port Radiado(dato:integer):integer;

    processor module Centro;
        import Radiado;
        var dato:integer;
            respuestas:array[Processors] of integer;
    begin
        dato:=...;
        respuestas := Radiado(dato);
        for k:=1 to NumeroSatelites do
            /* Procesar "respuestas[ Satelite[k] ]" */
        end for;
    end Centro;

    processor module Satelite[NumeroSatelites];
        import Radiado;
        var MiRespuesta:integer;
    begin
        region
            Radiado:begin Radiado:=MiRespuesta;end;
        end region;
    end Punto;

end RedEstrella;
```

Como puede apreciarse, la declaración del puerto *Radiado* no refleja que el módulo *Centro* recibe de hecho una formación de respuestas.

Figura 2.10: StarMod

```
group descriptor Dispositivos is
    type Estado is (on,off);
    type Respuesta is
        array(positive range <>) of Estado;
    entry Obtener_Estado(R:out Respuesta);
end Dispositivos;
```

Para que los clientes remotos puedan reservar espacio para almacenar las respuestas de todos los miembros vivos del grupo cooperativo, se introdujo una extensión del atributo **range**, que al aplicarlo a un identificador de grupo proporciona el rango de miembros vivos del grupo.

```
declare
    A: Respuesta(Dispositivo'range);
    ...
begin
    ...
    Obtener_Estado(A);
    ...
end;
```

2.3.2 Cláusulas de contexto

Para programar con grupos, las unidades de distribución del lenguaje deben poder especificar si tienen una relación de *pertenencia* con un grupo (es miembro de un grupo) o si tienen una relación de *uso* de un grupo (es cliente de un grupo —ver figura 2.11—).

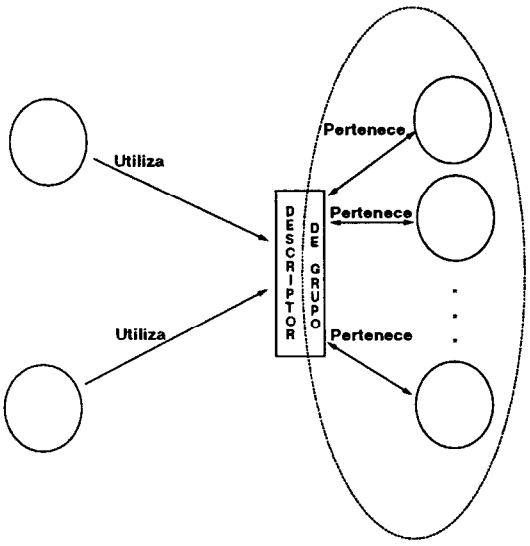


Figura 2.11: Relaciones entre las unidades de distribución y los grupos

Cuando un programa Ada utiliza un paquete de biblioteca, especifica esta relación mediante la cláusula de contexto **with**. Para mantener el lenguaje lo más cercano posible a Ada 83,

decidimos especificar la relación de uso de un grupo mediante una extensión de esta cláusula de contexto: la cláusula **with group**.

with_group_clause ::=
with group *group*_simple_name {, *group*_simple_name } ;

Como Ada 83 no posee ningún mecanismo que permita reflejar la relación de pertenencia, decidimos proporcionar soporte linguístico a esta relación mediante otra cláusula de contexto: la cláusula **for group**

for_group_clause ::=
for group *group*_simple_name {, *group*_simple_name } ;

2.3.3 Agente

Para proporcionar un soporte linguístico a la unidad de distribución de Drago, intentamos utilizar como unidad de distribución de Drago alguna de las unidades de compilación de Ada 83. Sin embargo, no las consideramos convenientes porque proporcionan poco nivel de abstracción y modularidad para un lenguaje distribuido.

No consideramos convenientes las unidades genéricas de Ada 83 porque son básicamente plantillas, con o sin parámetros, que se utilizan para obtener el subprograma o paquete correspondiente ([ANS83], apartado 12.1). Tampoco consideramos convenientes las tareas Ada 83 porque:

- No son una unidad de compilación de Ada 83, aunque varios proyectos la han propuesto como unidad de distribución de un lenguaje distribuido (XMS [GC85], DIADEM [AMN88], C-concurrente tolerante a fallos [CGR]).
- La semántica de la cita entre tareas Ada es difícil de definir en un entorno distribuido, especialmente los aspectos de temporización ([AMN88]).
- Las tareas Ada 83 no permiten declarar constantes o tipos en su especificación.

Por esta razón se introdujo una nueva unidad de distribución, a la que denominamos agente. El agente es básicamente un tipo especial de objeto abstracto. Tiene estado interno que no es accesible directamente desde el exterior y proporciona los servicios de los grupos a los que pertenece (ver figura 2.12). Consideramos dos tipos de agentes: agentes replicados, cuya semántica de ejecución facilita la implementación de aplicaciones replicadas, y agentes cooperativos para implementar aplicaciones cooperativas.

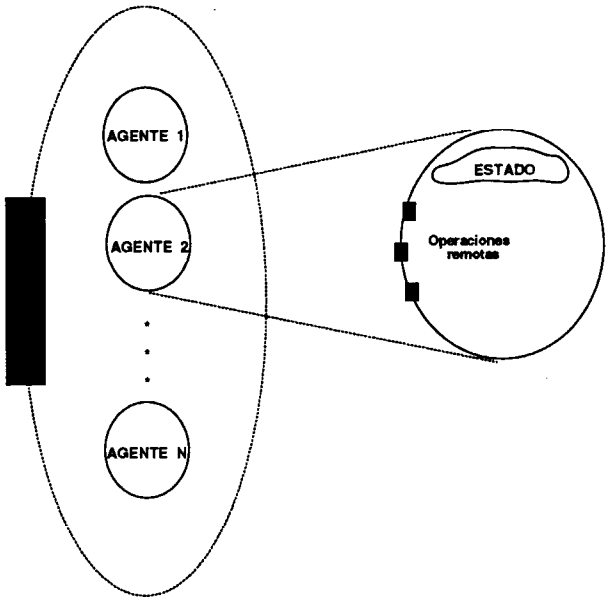


Figura 2.12: La unidad de distribución es el agente.

2.3.4 Llamada remota a procedimiento con radiado uniforme

El mecanismo básico de comunicación remota que proporciona Ada 95 [mt93] es la llamada remota a procedimiento. Por esta razón inicialmente intentamos que las operaciones remotas de los agentes se implementasen mediante procedimientos remotos. Sin embargo, como la llamada remota a procedimiento establece una relación 1 a 1 entre el cliente y el servidor, y necesitabamos un mecanismo de comunicación 1 a N (el cliente llama a todos los miembros del grupo) diseñamos una extensión de la llamada remota a procedimiento en la que el destino de la llamada fuese un grupo. Denominamos a este mecanismo *llamada remota a procedimiento con difusión*[2].

La semántica propuesta para esta llamada remota bloquea al llamador hasta que todos los miembros del grupo han ejecutado el procedimiento remoto[3]; si falla algún miembro del grupo se eleva la excepción **Group_Error** en el llamador. En el caso de llamada remota a un grupo replicado, el entorno de tiempo de ejecución puede dar por finalizada la llamada remota en cuanto alguno de los miembros ha completado la cita remota (ya que todos los miembros son réplicas idénticas y proporcionan los mismos valores en los parámetros de salida).

La sintaxis propuesta para esta llamada remota utiliza la notación punto de Ada 83, pero en vez de nombrar un paquete nombra un grupo (junto con una operación declarada en una especificación de grupo). Las operaciones remotas pueden tener parámetros de entrada (modo

[2]Este mecanismo no es nuevo. Sun [Cor91] posee una llamada remota a procedimiento con difusión. Sin embargo, su semántica de ejecución de *al menos una vez* no proporciona el nivel de abstracción de la llamada remota con difusión uniforme de Drago.

[3]Es posible que una aplicación no requiera que todos los miembros del grupo ejecuten el procedimiento remoto; hay aplicaciones que solamente necesitan que lo ejecute la mayoría de los miembros del grupo o un número determinado de miembros. Esto podría especificarse mediante algún *pragma*, pero se ha dejado como línea futura de estudio.

in) y salida (modo **out**). La tupla [operación remota, parámetros de entrada] constituye el mensaje que se envía a un grupo cuando se inicia una llamada remota; al completarse la llamada remota cada miembro del grupo envía al cliente remoto la tupla [parámetros de salida]. La implementación de Drago gestiona automáticamente la construcción y envío de los mensajes de llamada/respuesta por la red, asegurando que todos los miembros reciben las mismas peticiones de cita remota y en el mismo orden.

2.3.5 Puntos de entrada remotos

Para que el agente implementase las operaciones remotas mediante procedimientos remotos decidimos que la estructura interna del agente fuese similar a la de un paquete de biblioteca Ada. Así pues, los agentes tenían inicialmente una estructura interna similar a la siguiente:

```
agent Version_1_0 is
    procedure Servicio_1(...) is
        ...
    end;
    ...
    procedure Servicio_N(...) is
        ...
    end;
begin
    – Sección de inicialización
    ...
end Version_1_0;
```

La semántica de ejecución de la llamada remota a procedimiento de Ada 95 es reentrante y permite atender varias llamadas de forma concurrente ([mt93], apartado I-5, párrafo 13). Para alejarnos lo menos posible de Ada, consideramos que nuestros procedimientos remotos debían mantener dicha semántica. De esta forma, el agente puede atender de forma concurrente las llamadas de varios clientes remotos. Sin embargo, esto impide implementar servidores replicados (ver figura 2.13).

Para evitar este problema decidimos que los procedimientos remotos no fuesen reentrantes. De esta forma sólo puede haber un cliente remoto ejecutando cada una de las operaciones remotas. Sin embargo, esta solución no evita el problema cuando el agente replicado atiende llamadas simultáneas sobre procedimientos remotos diferentes. Además introduce un problema de ortogonalidad en el lenguaje, ya que para reutilizar código Ada los procedimientos internos al agente deben ser reentrantes, mientras que los procedimientos remotos no serían reentrantes.

Analizamos la semántica de ejecución de las unidades de distribución de Isis para ver cómo resolvían este problema y llegamos a la conclusión de que el modelo básico de ejecución de los procesos Isis es una implementación de monitores remotos [And91] con la semántica de los monitores del lenguaje Mesa [Xer85] (ver figura 2.15).

Estudiamos la posibilidad de que el agente tuviese la semántica de ejecución de un monitor distribuido. Sin embargo, consideramos conveniente que las operaciones remotas sólo pudiesen bloquearse al principio (a modo de regiones críticas condicionales). Segun Bal [Bal90], ésto hace que los programas sean mucho más legibles.

Supongamos que tenemos un agente replicado que implementa varias operaciones remotas que deben ejecutarse en exclusión mutua, y el único mecanismo de comunicación remota que tenemos es la llamada remota a procedimiento reentrante. Aparentemente podemos resolver este problema utilizando los mecanismos clásicos para conseguir exclusión mutua: semáforos, monitores, regiones críticas, tareas, etc. Supongamos que las réplicas reciben dos llamadas remotas consecutivas. Como la llamada remota es reentrante, en todas las réplicas se crean dos flujos de control f_1 y f_2 para que ejecuten estas operaciones remotas (ver A en figura 2.14).

Supongamos que f_1 comienza a ejecutar la operación remota, pero antes de sincronizar con la tarea que proporciona la exclusión mutua el planificador de Ada expulsa f_1 y comienza a ejecutar f_2. En este caso, en esta réplica f_2 se ejecutaría en exclusión mutua antes que f_1, mientras que en otra réplica podría ejecutarse f_1 antes que f_2 (alcanzando así un estado inconsistente de las réplicas —ver B en figura 2.14—).

En conclusión, aunque el código de las réplicas sea determinista, no podemos asegurar que todas las réplicas pasen por los mismos estados.

Figura 2.13: La semántica de la llamada remota a procedimiento de Ada 95 impide implementar servicios replicados

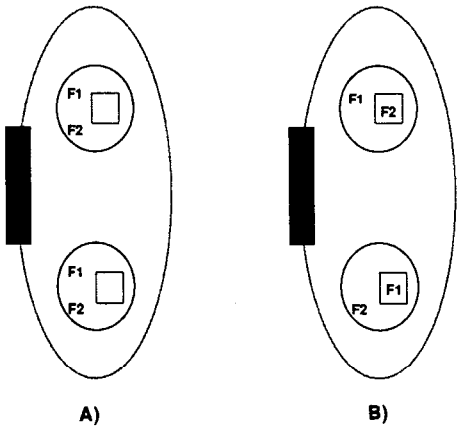


Figura 2.14: Las réplicas no pasan por los mismos estados

Análisis del modelo de monitor de Isis

Al comparar la semántica de funcionamiento de los procesos Isis [BCJ+90] con la semántica del monitor de Hoare [Hoa74] encontramos la siguiente diferencia: si al realizar la operación *señalar* hay procesos bloqueados:

- En el modelo de Hoare el llamador se suspende temporalmente, y se reactiva uno de los procesos bloqueados en la variable condición.
- En el modelo de Isis el llamador continúa su ejecución (simplemente avisa que dicho flujo de control puede continuar su ejecución en algún instante futuro).

Esta diferencia hace que el monitor de Isis siga un modelo "*signals as hints*" (de acuerdo con la clasificación de Andrews [AS83]), que es el modelo que utiliza el lenguaje Mesa [LR80]. Sin embargo, aunque el modelo de Isis es bastante cercano al modelo de monitor de Mesa, tiene las siguientes diferencias:

- En Mesa, no se asegura que otro proceso no entre en el monitor antes que el proceso que está bloqueado en una variable condición, y que ha sido señalado, continúe su ejecución ([AS83], pág 111), mientras que Isis no permite que otra petición remota entre en el monitor mientras éste pueda continuar ejecutando alguna tarea que ya está dentro del monitor ([BCJ+90], pág 160).
- En Isis, las primitivas *sleep()* y *bcast()* ceden el control del monitor.

En resumen, el modelo de proceso de Isis casi coincide con el modelo de monitor del lenguaje Mesa, al que se le ha incluido la característica de ceder el monitor al utilizar las primitivas de comunicación remota (*bcast*) y espera termporizada (*sleep*).

Figura 2.15: Modelo de monitor de Isis

Los objetos protegidos de Ada 95 ([mt93], apartado 9.4) proporcionan una semántica de ejecución similar a la deseada. El objeto protegido puede contener procedimientos, funciones y puntos de entrada. Los procedimientos protegidos proporcionan acceso de lectura y escritura en exclusión mutua ([mt93], apartado 9.5.1., párrafo 1). Las funciones protegidas sólo proporcionan acceso de lectura en exclusión mutua ([mt93], apartado 9.5.1, párrafo 1). Los puntos de entrada protegidos son básicamente procedimientos que tienen asociada una expresión que, de forma similar a las guardas de los puntos de entrada de las tareas Ada, permiten encolar la ejecución de la operación protegida ([mt93], apartado 9.5.2). Consideramos conveniente simplificar el modelo de los objetos protegidos y nos quedamos exclusivamente con lo que necesitábamos para programar aplicaciones distribuidas: los puntos de entrada protegidos. De esta forma surgieron los puntos de entrada remotos, que sustituyeron a los procedimientos remotos, y los agentes pasaron a tener una estructura interna similar a la siguente:

```
agent Version_1_1 is

    entry Servicio_1(...) when <Expresión> is
        ...
    end;
    ...
    entry Servicio_N(...) when <Expresión> is
        ...
    end;

begin
    – Sección de inicialización
    ...
end Version_1_1;
```

2.3.6 Sentencia *requeue*

Utilizamos el modelo anterior para programar varios ejemplos de aplicaciones distribuidas tolerantes a fallos (semáforo binario tolerante a fallos, semáforo general tolerante a fallos, variable distribuida tolerante a fallos, buzón tolerante a fallos, etc.) y encontramos un problema existente en Ada 83 que ahora se hacía mucho más patente; como la expresión asociada a los puntos de entrada no puede referenciar los parámetros de la llamada, cuando la guarda de una operación remota necesita evaluarlos, el cliente debe realizar dos llamadas remotas consecutivas: la primera pasa el valor de los parámetros de la llamada para que el servidor pueda evaluar la guarda de la segunda llamada. El coste inherente de la comunicación remota hace que este problema sea importante. Además, el retraso de la comunicación remota aumenta la posibilidad de que entre las dos llamadas se cuele la llamada de otro cliente, lo que complica la implementación de los servidores remotos.

Ada 95 ha resuelto este problema proporcionando al programador una sentencia de reencolamiento ([mt93], apartado 9.5.4). Decidimos incorporar en Drago la sentencia de reencolamiento porque proporciona las siguientes ventajas: evita comunicación remota innecesaria, asegura atomicidad en la evaluación de los parámetros y el posterior reencolamiento, y proporciona transparencia a los clientes remotos (los clientes remotos no necesitan realizar dos llamadas). El siguiente fragmento de código muestra cómo utilizar el reencolamiento.

```
entry Primera_Parte(a,b,...) is
    if f(a,b,...) then
        requeue Segunda_Parte;
    end if;
    ...
end Primera_Parte;

entry Segunda_Parte(a,b,...) when <Expresión> is
    ...
end Segunda_Parte;
```

Expresión puede involucrar variables cuyos valores hayan sido leidos de los parámetros de la *Primera_Parte*.

2.3.7 Planificación por petición de las llamadas remotas

Como la llamada remota a un grupo mantiene bloqueado al llamador hasta que todos los miembros del grupo invocado han ejecutado el procedimiento remoto, cuando un agente replicado realiza una llamada remota a un grupo, permanece bloqueado hasta que todos los miembros del grupo han ejecutado el procedimiento remoto (ver figura 2.16). Este comportamiento impide que un servidor robusto, implementado mediante un grupo replicado, atienda otras llamadas remotas mientras espera la respuesta de una llamada que él ha podido hacer a su vez, para delegar parte de su servicio.

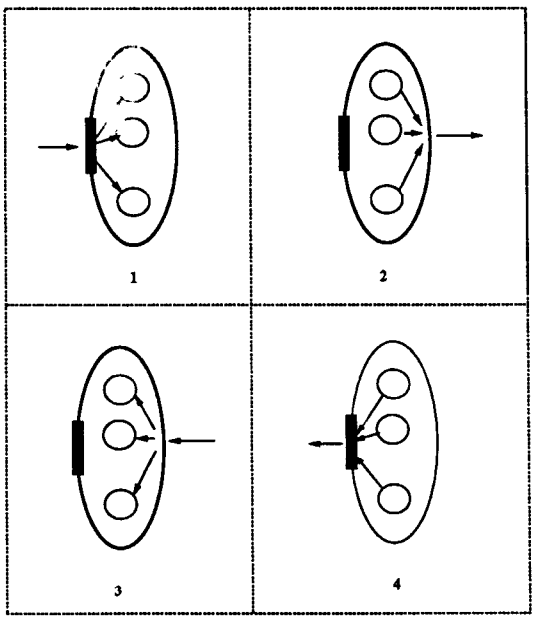


Figura 2.16: Secuencia de un servidor-cliente replicado

Tanenbaum propone como solución a este problema implementar el servidor mediante una gran máquina de estados finita ([Tan92], página 510). En vez de bloquearse al hacer una llamada, el servidor almacena el estado de la operación remota en una tabla (que como

consecuencia se queda suspendida), y acepta el siguiente mensaje del subsistema de comunicación. Este mensaje puede ser una nueva petición de operación remota o una respuesta de una operación pendiente (almacenada en la tabla). Si es una nueva petición, simplemente ejecuta la operación. Si es una respuesta, extrae de la tabla la información relevante y procesa la respuesta. En términos generales es un modelo de tareas ejecutadas mediante un planificador expulsivo, en el que los cambios de contexto ocurren cada vez que el servidor realiza una comunicación remota (ya que el estado de la computación se almacena y restaura en la tabla cada vez que se realiza una comunicación remota). Este modelo no se integra bien con el modelo de tareas de Ada 83, que no presupone si el planificador de tareas es expulsivo o no. Sin embargo, al no encontrar una solución mejor, introdujimos en el agente un modelo de planificación por petición para ejecutar las operaciones remotas.

Para asegurar que todas las réplicas tienen exactamente los mismos conjuntos de tareas planificables en todos los puntos de cambio de contexto, y siguiendo el modelo de planificación de Isis, decidimos que el planificador no introdujese una nueva llamada remota en el conjunto de tareas planificables mientras tuviera alguna tarea que pudiera continuar su ejecución. De esta forma el agente solamente crea un nuevo flujo para ejecutar una nueva llamada remota cuando no puede continuar la ejecución de ninguna de las llamadas remotas que le quedan por completar.

2.3.8 Dos planificadores en los agentes cooperativos

Denominamos trabajo cooperativo al trabajo realizado por varios miembros de un grupo para conseguir un objetivo común (definido por la semántica de la aplicación). Podemos considerar dos formas de cooperación:

1. Cada miembro del grupo hace silenciosamente su parte del trabajo. En este caso, cuando un cliente solicita que el grupo realice una operación remota, cada miembro conoce perfectamente su parte del trabajo y lo hace sin comunicarse con el resto de los miembros.
2. Los miembros del grupo se comunican para cooperar en la realización de cada trabajo.

Para mantener la homogeneidad del lenguaje, intentamos reutilizar la semántica de ejecución del agente replicado para programar las aplicaciones cooperativas. La semántica de ejecución de los agentes replicados es suficiente para programar el primer tipo de aplicaciones cooperativas, pero no es suficiente para programar aplicaciones cooperativas que requieren comunicación intragrupo. La razón de ello es que, cuando un miembro recibe una llamada L, no puede atender nuevas llamadas remotas hasta que finaliza la ejecución de L, o realiza él mismo una llamada remota. Esto impide que el resto de los miembros se comuniquen con él para cooperar en la realización de L. El problema subyacente reside en la falta de concurrencia dentro del agente replicado. Como solución, se propuso que el agente cooperativo tuviese internamente flujos de control adicionales, que le permitiesen realizar trabajo mientras atendía de forma concurrente las llamadas remotas (intergrupo o intragrupo).

Consideramos la posibilidad de proporcionar estos flujos de control adicionales mediante agentes locales. Sin embargo, los agentes locales duplican la funcionalidad que proporcionan

las tareas. Por esta razón decidimos sustituirlos por tareas Ada, y obtuvimos además las siguientes ventajas:

- Reutilizamos en Drago el modelo de cita de Ada 83 incorporando en Drago toda la expresividad de las tareas Ada.
- Conseguimos un modelo más cercano a Ada, objetivo importante si se tiene cuenta que Drago debía ser implementable mediante un preprocesador que generara código Ada.

Tal y como comentamos anteriormente, el modelo de ejecución del agente de esta primera versión de Drago se basa en un planificador por petición (descrito en el apartado 2.3.7). Sin embargo, Ada no presupone si el planificador de tareas es expulsivo o no. Para incorporar las tareas Ada decidimos que el agente cooperativo tuviese internamente dos planificadores: un planificador por petición que gestionase la ejecución de los flujos de control que ejecutan las llamadas remotas, y el planificador de Ada que gestionase la ejecución de las tareas Ada.

2.3.9 Punto de planificación en la cita con tareas locales

Siguiendo el modelo descrito en el apartado anterior (utilizar tareas locales para estructurar el agente cooperativo cuando realiza comunicación intragrupo), encontramos un problema cuando una operación remota (un punto de entrada remoto) tiene parámetros de salida: Si la operación realiza una única cita con una tarea local para que realice el trabajo, el agente no puede atender llamadas remotas posteriores (incluyendo las intragrupo) hasta que complete la cita con la tarea local.

Para evitar este problema se propuso que este tipo de aplicaciones utilizase el reencolamiento. De acuerdo con ello, el agente atiende la llamada remota que tiene parámetros de salida, realiza una primera sincronización con la tarea que realiza el trabajo correspondiente, cierra la guarda de un punto de entrada local y reencola en él la llamada. Cuando, posteriormente, la tarea local finalice su trabajo, abrirá la guarda de este punto de entrada local y sincronizará con el agente pasándole los parámetros de salida que debe retornar el agente al cliente remoto.

Este modelo tiene un fallo. De acuerdo con la semántica de ejecución, cuando el agente no tiene ninguna tarea planificable simplemente espera la siguiente llamada remota, es decir, aunque la tarea local modifique el valor de una guarda para que sea cierta, el agente no continuará la ejecución de la operación correspondiente hasta que reciba alguna llamada remota.

Como los flujos de control que ejecutan las operaciones remotas se ejecutan mediante un planificador por petición (apartado 2.3.7), decidimos considerar la cita de estos flujos de control con las tareas locales como puntos de planificación implícitos. De esta forma, la implementación de un agente cooperativo que realiza trabajo mientras atiende peticiones remotas adicionales era la siguiente:

```
1:  agent Ejemplo_Modelo_1 is
2:
3:      task T_Local is
4:          entry Calcula(a:in integer;b:out integer);
5:      end T_Local;
6:      task body T_Local is
7:      begin
8:          loop
9:              accept Calcula(a:in integer;b:out integer) do
10:             ...
11:             end;
12:         end loop;
13:     end T_Local;
14:
15:     entry Hacer(a:in integer;b:out integer) is
16:         T_Local.Calcula(a,b);
17:     end Hacer;
18:
19:     -- Otros puntos de entrada
20:     ...
21:
22: begin
23:     -- Código de inicialización del agente
24:     ...
25: end Ejemplo_Modelo_1;
```

La semántica de ejecución de este ejemplo es la siguiente: Cuando el agente recibe una llamada remota en el punto de entrada *Hacer*, si no está ejecutando ninguna llamada remota anterior, la ejecuta (puesto que no existe guarda —línea 15—) y sincroniza con la tarea local *T_Local* para pasarle los parámetros del trabajo que debe realizar (línea 16). Esta sincronización es un punto implícito de cambio de contexto, lo que permite al agente atender nuevas llamadas remotas. El planificador de Ada se encarga de continuar la ejecución de *T_Local*, mientras que el planificador de Drago se encarga de controlar la ejecución de los flujos de control (*threads*) que ejecutan las llamadas remotas. Cuando *T_Local* complete la cita, el planificador del agente marcará como lista la llamada remota que inició esta cita, permitiéndole así continuar su ejecución.

2.3.10 Problemas encontrados en esta versión

La complejidad del lenguaje, debido principalmente al modelo de agente cooperativo con dos planificadores, dificultó la redacción del manual de referencia de esta versión de Drago. Además, su implementación mediante un preprocesador que genere código Ada 83 no es trivial. En particular, es difícil conseguir que, en el caso general, la cita de un flujo de control creado para ejecutar una llamada remota con una tarea local sea un punto de planificación implícito[4]. Como conclusión, decidimos realizar una revisión del lenguaje. El resultado fue la segunda versión de Drago.

[4]El flujo que ejecuta la llamada remota puede llamar a un procedimiento que sea quien realice la cita con la tarea, y esta cita solamente debe significar cambio de contexto en el planificador por petición de Drago cuando la realiza un flujo de control que ejecuta una llamada remota, no cuando la realiza una tarea Ada que llama al procedimiento para sincronizar con otra tarea Ada.

2.4 Segunda versión

Este apartado presenta los elementos de la segunda versión de Drago: la especificación de grupo, las cláusulas de contexto de grupos, la semántica de ejecución del agente, la sentencia de aceptación de cita remota, la cita remota en sí, la cita remota con reencolamiento, los puntos de entrada de fallo y las excepciones remotas.

2.4.1 Especificación de grupo

Con el objetivo de mantener Drago lo más homogéneo posible con Ada 83, consideramos conveniente que la sintaxis de la interfaz de un grupo fuese similar a la sintaxis de la especificación de un paquete Ada 83. Para permitir que la especificación de un grupo aprovechase la potencia expresiva de los tipos privados de Ada 83 (tipos que restringen su manipulación por parte de los clientes externos) añadimos una sección privada. De esta forma la sintaxis de la especificación de un grupo quedó de la siguiente forma[5]:

group_specification ::=
    [**replicated**] **group specification** *group*_simple_name **is**
        *intergroup*_basic_declarative_item
    [**intragroup**
        *intragroup*_basic_declarative_item]
    [**private**
        *privated*_basic_declarative_item]
    end [*group*_simple_name];

Consideramos conveniente también permitir declaraciones de excepciones en la especificación de grupo, ya que de esta forma el programador puede especificar los posibles fallos del grupo. Sin embargo, mantuvimos la restricción de que la especificación de grupo no puede contener declaraciones de variables. Puede contener declaraciones de constantes, tipos (incluyendo declaraciones de subtipos, tipos privados y tipos privados limitados), excepciones remotas y puntos de entrada remotos. La figura 2.17 muestra un ejemplo de especificación de grupo que utiliza las tres secciones.

2.4.2 Cláusulas de contexto

Las cláusulas de contexto **with group** y **for group** se mantuvieron sin modificación en esta segunda versión de Drago, ya que reflejaban bien las relaciones de pertenencia y uso de grupos en los ejemplos diseñados con la primera versión.

[5] Para una descripción completa de la especificación de grupo ver el manual de referencia de Drago en el apéndice.

Ejemplo

El siguiente fragmento de código muestra la especificación de un servidor de ajedrez distribuido:

```
group specification Ajedrez_Distribuido is
    type id_oponente is private;
    type Movimiento is ...;
    entry Anotar_Oponente(Id : out id_oponente);
    entry Jugada_Oponente (Id : in id_oponente;
                           Su_jugada : in movimiento;
                           Nuestra_jugada : out movimiento);
    Movimiento_Erroneo:Exception;
intragroup
    entry Encontre_buena_jugada(Id: in id_oponente;
                           m: in movimiento);
private
    type id_oponente is ...
end Ajedrez_Distribuido;
```

Se ven las tres secciones de una especificación de grupo cooperativo. La sección intergrupo abarca desde la cabecera de la especificación del grupo hasta la palabra reservada **intragroup**. La sección intragrupo va desde la palabra reservada **intragroup** hasta la palabra reservada **private**. Finalmente la sección privada llega hasta el final de la especificación de grupo. De forma similar a Ada 83, la sección privada contiene la declaración completa de los tipos privados declarados en la sección intergrupo.

La sección intergrupo contiene la declaración del tipo privado *id_oponente*, que identifica de forma unívoca a los jugadores; el tipo *Movimiento*, que contiene los posibles movimientos en el tablero; el servicio intergrupo *Anotar_Oponente*, que permite anotar un nuevo jugador y proporciona como parámetro de salida el identificador asociado a este jugador; y el servicio intergrupo *Jugada_Oponente*, que permite a los clientes remotos informar sobre su siguiente movimiento. Si el usuario intenta realizar una jugada incorrecta el grupo propaga la excepción remota *Movimiento_Erroneo* al cliente remoto.

La sección intragrupo contiene la especificación de un punto de entrada que utilizan los miembros del grupo para comunicarse la mejor jugada que pueden hacer. Cada miembro del grupo realiza la evaluación del tablero utilizando un algoritmo diferente y cuando alguno encuentra una buena jugada la notifica al resto de los miembros mediante el punto de entrada intragrupo *Encontre_buena_jugada*, para que dejen de buscarla.

Figura 2.17: Ejemplo de especificación de los servicios de un grupo.

2.4.3 Semántica de ejecución del agente

La programación de aplicaciones distribuidas utilizando la primera versión de Drago no era demasiado elegante. En particular, como el agente ofrece permanentemente todos los servicios remotos y a veces la ejecución de una operación necesita inhibir temporalmente otros servicios remotos, el programador necesita utilizar expresiones complejas en las barreras asociadas a los puntos de entrada remotos, oscureciendo el código.

Analizamos cómo quedaría el lenguaje si hacíamos que la semántica de ejecución del agente fuese más cercana a la semántica de ejecución de las tareas Ada. Para ello proporcionamos al agente una extensión de la sentencia **accept** que le permitiese aceptar llamadas remotas, y una sentencia **select** que le permitiese especificar los servicios remotos que proporcionaba en cada instante.

Ejemplo con ambos modelos de agente

Veamos como se implementa un semáforo tolerante a fallos con ambas versiones del lenguaje:

```
replicated agent Semaforo_Bin is
   Ocupado:Boolean:=false;

   entry P when not Ocupado is
   begin
      Ocupado:=true;
   end P;

   entry V is when Ocupado is
   begin
      Ocupado:=false;
   end V;

end Semaforo_Bin;
```

```
replicated agent Semaforo_Bin is
begin
   loop
      accept P;
      accept V;
   end loop;
end Semaforo_Bin;
```

El agente de la izquierda está escrito con la primera versión del lenguaje y el de la derecha con la segunda versión. Como puede apreciarse a simple vista, la implementación mediante la segunda versión es más sencilla y clara.

Figura 2.18: Ejemplo implementado con ambas versiones de Drago.

Reescribimos con el nuevo modelo algunos ejemplos de aplicaciones replicadas que habíamos diseñado con el modelo anterior (la figura 2.18 contiene un ejemplo sencillo) y encontramos las siguientes ventajas:

1. Es más fácil de implementar: el nuevo modelo es conceptualmente más simple y mucho más cercano a Ada 83, lo que facilita su implementación.
2. Es más claro: el flujo de control del agente es más fácil de seguir por el programador.
3. Es más potente, ya que:

(a) El agente puede especificar qué peticiones remotas acepta en cada instante (lo que evita complejas condiciones en las barreras).

(b) El agente puede realizar operaciones de mantenimiento sin mantener bloqueado al llamador, algo que no podía hacerse con la primera versión.

Por todo ello, en el modelo actual de Drago el agente tiene una semántica de ejecución cercana a la de las tareas Ada; el agente tiene un flujo de control que ejecuta el bloque de sentencias del agente. El agente puede contener procedimientos, tareas y paquetes. Al igual que en la versión anterior, el estado interno del agente no es accesible directamente desde el exterior. Por otra parte, el agente contiene internamente una copia de todos los paquetes de biblioteca que utiliza ya que así evitamos que se utilicen los paquetes de biblioteca como un mecanismo indirecto para compartir estado entre agentes.

Para asegurar un comportamiento determinista, los agentes replicados no pueden contener tareas, ni tampoco pueden contenerlas los paquetes de biblioteca que utilicen. Sin embargo, los agentes cooperativos pueden contener tareas internas.

2.4.4 Sentencia *select*

Como la semántica de la sentencia **select** de Ada es no-determinista, no puede utilizarse para implementar código determinista en los agentes replicados. Para resolver este problema decidimos que la semántica de la sentencia **select** del agente fuese determinista, y modificamos la semántica de la sentencia **select** de Ada de la siguiente forma:

- Se elige siempre petición remota más antigua. De esta forma, como el subsistema de comunicación de Drago entrega los mensajes en todas las réplicas en el mismo orden (apartado 2.2.3), todas las réplicas de un grupo replicado eligen exactamente la misma petición remota.

- No puede contener la alternativa *terminate*, ya que nos resulta difícil de implementar en un entorno distribuido.

- No puede realizar espera temporizada. Para que la espera temporizada sea útil es necesario que las llamadas remotas se encolen en los puntos de entrada desde el instante en que se reciben. Como el subsistema de comunicación de Drago no asegura que todas las réplicas reciban las mismas peticiones al mismo tiempo, sólo en el mismo orden, una réplica puede recibir una petición antes de que se cumpla el plazo de la temporización, y por tanto acepte la llamada, mientras que otra réplica puede recibir la petición después de haber transcurrido el plazo de temporización y rechace la llamada, alzanzando así un estado inconsistente en las réplicas.

- No puede contener la alternative *else*, ya que conceptualmente es equivalente a una alternativa *delay 0*.

2.4.5 Cita remota

Como la semántica de ejecución del agente se acercó a la semántica de ejecución de las tareas Ada, se consideró conveniente que la llamada remota a un grupo se convirtiese en una cita remota con un grupo. De esta forma, el único mecanismo de comunicación y sincronización remota de Drago es la cita remota con un grupo.

2.4.6 Cita remota con reencolamiento

Para resolver el problema descrito en el apartado 2.3.7 (aumentar el paralelismo de los servidores replicados que realizan a su vez peticiones remotas a otros grupos) decidimos introducir en Drago algún mecanismo que permitiese realizar una llamada remota asíncrona. Ada 95 proporciona soporte para realizar llamadas remotas asíncronas, pero no permite que la llamada tenga parámetros de salida ([mt93], apartado I.4.1). Con el objetivo de resolver este problema surgió la **cita remota con reencolamiento**. Su sintaxis es la siguiente:

requeued_remote_call ::=
group_remote_call_statement **requeue in** entry_call_statement

Para recoger los parámetros de salida de la llamada asíncrona introdujimos los parámetros de entrada adicionales en la sentencia de reencolamiento: el agente puede realizar la cita remota y especificar en la sentencia de reencolamiento los parámetros de salida que pasa como parámetros de entrada al punto del reencolamiento. Por ejemplo:

Grupo.operacion(... , respuesta) requeue in E(respuesta);

Para permitir que el llamador pueda tratar las excepciones remotas que se propaguen en la llamada asíncrona, decidimos que estas excepciones se eleven en el punto de reencolamiento. Para que el programador pueda asociar al **accept** un manejador de excepciones, fue necesario modificar la sintaxis del agente de la siguiente forma[6]:

agent_accept_statement ::=
accept [*group*_simple_name.] *entry*_simple_name [formal_part] [**do**
sequence_of_statements
[**exception**
exception_handler
{,exception_handler}]
end [*entry*_simple_name]];

La ejecución de la cita remota con reencolamiento consta de dos partes. La primera parte la ejecuta el agente; la segunda parte la ejecuta el entorno de tiempo de ejecución de Drago:

[6]Modificación adoptada también en Ada 95.

1. En la primera parte el agente realiza la cita remota asíncrona pasando una copia de los parámetros de entrada (modo **in**), finaliza la ejecución de la sentencia **accept** más interna y continua su ejecución después de asegurarse que la cita llegó a su destino (para evitar problemas de causalidad).
2. En la segunda parte, cuando posteriormente se complete la cita remota asíncrona, el entorno de tiempo de ejecución de Drago realiza de forma atómica la copia de los parámetros de salida (modo **out**) en los parámetros reales y el reencolamiento de la cita remota bloqueada en el punto de entrada especificado en la sentencia de cita remota con reencolamiento.

Esta semántica permite al programador realizar la llamada asíncrona de dos formas:

Método 1:

```
accept E1(...) do
   .
   .
   Grupo.op(..., respuesta) requeue in E2(respuesta);
end E1;
– Trabajo adicional realizado mientras se completa la
– cita remota.
.
.
accept E2(..., respuesta) do
   ...
end E2;
```

Método 2:

```
loop
   select
      accept E1(...) do
         ...
         Grupo.op(..., respuesta) requeue in E2(respuesta);
      end E1;
   or
      accept E2(..., respuesta) do
      ...
      end E2;
   or
      – Otros servicios que proporciona el agente mientras
      – espera la respuesta de la cita remota.
      ...
   end select;
end loop;
```

El primer método se corresponde con el modelo tradicional de reencuentro con el flujo de control asíncrono. El segundo método permite al agente proporcionar servicio mientras espera a que se complete la cita remota (el apartado 3.5 del siguiente capítulo contiene un ejemplo que utiliza este método).

Los parámetros reales utilizados en una sentencia de cita remota con reencolamiento (la variable *respuesta* de los fragmentos de código anteriores) deben ser globales al agente. Esto es necesario para que el entorno de tiempo de Drago pueda realizar de forma atómica la actualización de los parámetros de salida de la cita remota y el posterior reencolamiento. La actualización y reencolamiento de forma atómica permite que los puntos de entrada que contienen citas remotas con reencolamiento sean reentrantes. Veamoslo en detalle.

Supongamos que un agente realiza una cita remota con reencolamiento de acuerdo con el segundo modelo descrito anteriormente. Supongamos que el agente recibe dos peticiones de cita remota sucesivas en $E1$ y por ello realiza dos citas remotas con reencolamiento consecutivas utilizando la misma variable global *respuesta*. Su comportamiento es el siguiente:

- Cuando se completa la primera cita remota y recibe la primera respuesta, el entorno de tiempo de ejecución de Drago almacena el valor del parámetro de salida en la variable global *respuesta* e inmediatamente, y de manera atómica, realiza el reencolamiento (utilizando el valor recién actualizado en *respuesta*).

- Cuando se completa la segunda cita remota y se recibe la segunda respuesta, se almacena el valor del parámetro de salida en *respuesta* y se realiza el reencolamiento de la segunda cita remota con reencolamiento (utilizando el nuevo valor de *respuesta*).

2.4.7 Sentencia *requeue* con parámetros de entrada adicionales

El reencolamiento de Ada 95 obliga a que el punto de entrada donde se realiza el reencolamiento tenga una cabecera que sea "conforme" con la cabecera de la operación que se reencola ([mt93], apartado 9.5.4). Sin embargo, la incorporación de parámetros de entrada adicionales en la cita remota con reencolamiento nos llevó a considerar que el reencolamiento tuviese parámetros de entrada adicionales. De esta forma conseguimos un lenguaje más homogéneo y permitimos programar servidores remotos más complejos. Por ejemplo, un servidor remoto puede recibir una petición remota, realizar parte del trabajo y, al no poder continuar la computación, reencolar la cita en otro punto de entrada pasando como parámetro de entrada adicional el estado actual de la computación. Posteriormente, cuando pueda continuar el trabajo, aceptará la petición reencolada recibiendo como parámetro de entrada adicional el estado que tenía la computación cuando se reencoló.

```
accept Primera_Parte(a,b) do
   ...
   requeue Segunda_Parte(Estado);
end Primera_Parte;
.
.
.
accept Segunda_Parte(a,b,Estado) do
   ...
end Segunda_Parte;
```

2.4.8 Puntos de entrada locales

Al utilizar el reencolamiento (de acuerdo con el modelo descrito en la sección anterior) consideramos necesario que el agente tuviese puntos de entrada locales (puntos de entrada que no son visibles al resto de los agentes) para reencolar las operaciones remotas.

Al igual que el resto de declaraciones locales del agente, los puntos de entrada locales deben declararse en la sección de declaraciones locales del agente, y proporcionan colas locales que sólo pueden utilizarse para el reencolamiento.

2.4.9 Comunicación intragrupo

Para realizar la comunicación intragrupo el llamador debe realizar una cita remota intragrupo (una cita remota con un punto de entrada declarado en una sección intragrupo). Como la cita remota de Drago bloquea al llamador hasta que todos los miembros del grupo han completado la cita remota y el llamador es miembro del grupo, si el llamador realiza una cita intragrupo se presenta un problema de interbloqueo. Hay dos formas de evitar este problema: realizar la cita intragrupo mediante una cita remota con reencolamiento o realizar la cita intragrupo mediante una tarea local. En el primer caso, el código de la llamada tendría la siguiente estructura:

```
agent Ejemplo_Cita_Intragrupo_1 is
   entry Fin_Cita_Intragrupo(...); - - Declaración punto entrada local
begin
   Cita_Intragrupo(....) requeue in Fin_Cita_Intragrupo(...);
   accept Cita_Intragrupo(...) do
      – Acepta y procesa su propia llamada intragrupo
      ...
   end;
   accept Fin_Cita_Intragrupo(...) do
      – Al llegar a este punto todos los miembros han
      – completado la cita remota intragrupo.
      ...
   end;
end Ejemplo_Cita_Intragrupo_1;
```

En el segundo caso el agente tiene internamente una tarea local que se encarga de realizar la cita remota intragrupo y se bloquea hasta que todos los miembros del grupo, incluyendo el llamador, han completado la cita remota. Por ejemplo:

```
agent Ejemplo_Cita_Intragrupo_2 is

   task T_Local is
      ...
   end T_Local;
   task body T_Local is
   begin
      ...
      Cita_Intragrupo(...);
   end T_Local;
```

```
begin
   ...
end Ejemplo_Cita_Intragrupo_2;
```

2.4.10 Cita local con reencolamiento

Como se comentó en el apartado 2.3.8, para que el agente cooperativo pueda atender nuevas citas remotas, intergrupo o intragrupo, mientras atiende una petición remota, debe sincronizar con una tarea local que sea quien realice dicho trabajo. Este modelo tiene el problema de que es necesario algún mecanismo que permita resincronizar al agente con la tarea para que éste recoja los parámetros de salida de la tarea y pueda contestar al cliente remoto. Para resolver este problema, en la primera versión de Drago consideramos que la cita del agente con las tareas locales fuese un punto implícito de cambio de contexto (apartado 2.3.9), solución difícil de implementar (apartado 2.3.10). Sin embargo, la incorporación de la cita remota con reencolamiento introdujo otra solución a este problema: la cita local con reencolamiento. Veamos como ejemplo un agente cooperativo escrito con el nuevo modelo:

```
agent Ejemplo_Agente_Cooperativo is
   tmp:integer;
   entry Hecho(a:in integer;b:out integer;tmp:in integer);
   task T_Local is
      entry Calcula(a:in integer;b:out integer);
   end T_Local;
   task body T_Local is
   begin
      loop
         accept Calcula(a:in integer;b:out integer) do
            ...
         end;
      end loop;
   end T_Local;

begin
   – Código de inicialización del agente
   ...
   loop
      select
         accept Hacer(a:in integer;b:out integer) do
            T_Local.Calcula(a,tmp) requeue in Hecho(tmp);
         end Hacer;
      or
         accept Hecho(a:in integer;b:out integer;tmp:in integer);
            b:=tmp;
         end Hecho;
      or
         – Otros puntos de entrada
         ...
      end select;
   end loop;
end Ejemplo_Agente_Cooperativo;
```

La semántica de ejecución de este modelo es la siguiente. El agente ofrece permanentemente todos los servicios remotos. Cuando recibe una cita remota en *Hacer*, la acepta y realiza

una cita con reencolamiento con la tarea local. Esto le permite atender peticiones remotas posteriores mientras la tarea local realiza el trabajo. Cuando la tarea local finaliza la cita, el entorno de tiempo de ejecución del agente actualiza *tmp* e inmediatamente después, y de manera atómica, realiza el reencolamiento de la cita en *Hecho* pasando como parámetro de entrada adicional el nuevo valor de *tmp*, permitiendo así al agente continuar el trabajo pendiente.

2.4.11 Excepciones remotas

Para completar la especificación de los servicios remotos que ofrece un grupo, consideramos conveniente incorporar declaraciones de excepciones (excepciones remotas) en la especificación de un grupo (ver apartado 2.4.1.

La incorporación de excepciones remotas en los grupos replicados fue sencilla, ya que todas las réplicas elevan exactamente las mismas excepciones remotas. Para notificar el fallo del grupo (cuando fallan todas las réplicas), decidimos que Drago elevase la excepción predefinida **Group_Error** en el punto de la llamada.

Sin embargo, en los grupos cooperativos, puede que un miembro propague una excepción remota diferente que el resto de los miembros. Por esta razón decidimos que la excepción no se eleve hasta que todos los miembros hayan completado la cita remota; si solamente un miembro propaga una excepción, se eleva esta excepción en el punto de la llamada; si varios miembros propagan excepciones diferentes, se eleva la excepción predefinida **Several_Exceptions**.

2.4.12 Puntos de entrada de fallo

Para implementar grupos cooperativos tolerantes a fallos es necesario que el lenguaje proporcione algún mecanismo de notificación en los miembros supervivientes[7]. Esta notificación permite programar aplicaciones cooperativas en las que se liberan los recursos del miembro que falló y a continuación los supervivientes se redistribuyen el trabajo de éste. Para ello se introdujo un punto de entrada especial que solamente puede aparecer en la sección intragrupo de la especificación de un grupo cooperativo: el punto de entrada de fallo.

El punto de entrada de fallo puede especificar un parámetro de entrada opcional en el que se identifica el miembro que falló. Cuando falla un miembro del grupo, Drago realiza automáticamente una cita remota intragrupo[8] con el punto de entrada de fallo de todos los agentes del grupo que aún están vivos.

2.5 Programación con Drago

En este último apartado presentamos una descripción general de las características de Drago para programar aplicaciones replicadas y aplicaciones cooperativas. Conviene resaltar que la

[7] Birman [BG93] recomienda una notificación consistente en todos los miembros del grupo (una notificación que cumpla las propiedades del entorno virtualmente síncrono).

[8] Con la misma semántica de ordenación global y atomicidad que el resto de las citas remotas de Drago.

semántica de ejecución de los agentes Drago y la comunicación con grupos mediante la cita remota permiten reutilizar código escrito mediante tareas Ada para implementar la versión distribuida correspondiente (e incluso replicarla, si su comportamiento es determinista).

2.5.1 Programación de aplicaciones replicadas

En resumen, para facilitar la programación de aplicaciones replicadas Drago posee las siguientes características:

- Características que facilitan la programación de los clientes del grupo:

  1. **Transparencia de comunicación.** Drago radía a todos los miembros del grupo una copia de la petición del cliente, asegurando uniformidad. El programador no necesita especificar la ubicación física de las réplicas; simplemente realiza la petición nombrando el grupo.
  2. **Transparencia de respuestas.** Drago filtra las respuestas duplicadas y proporciona a la aplicación sólo una de las respuestas del grupo. De esta forma proporciona la abstracción de que el grupo consta de un único miembro.
  3. **Transparencia de fallos.** Si falla algún miembro del grupo antes de completar la operación y queda alguna réplica viva que pueda proporcionar el servicio, Drago enmascara el fallo y proporciona a la aplicación la respuesta de otra réplica. Esto es posible porque las réplicas se ejecutan sobre procesadores con semántica de *fallo silencioso*, lo que asegura que ningún procesador erróneo produce una respuesta errónea. Como el enmascaramiento de fallos no evita la necesidad de tratar el caso en que fallan todos los miembros del grupo ([Chr91], pág. 69), cuando fallan todos los miembros de un grupo replicado Drago eleva la excepción *Group_Error* en el punto de la llamada. Esto permite a los clientes remotos enmascarar el fallo redirigiendo la petición hacia otro grupo, proporcionar mensajes de error, etc.

- Características que facilitan la programación de las réplicas:

  1. **Entrega uniforme de peticiones.** Característica necesaria para asegurar que todas las réplicas reciben exactamente las mismas peticiones y en el mismo orden [GAAM93, Sch90].
  2. **Semántica de ejecución de las réplicas.** La semántica de ejecución determinista de las sentencias Drago facilita la programación de miembros deterministas. Además, la cita remota con reencolamiento permite que las réplicas tengan un cierto grado de paralelismo.
  3. **Transparencia de llamadas a otros grupos.** Cuando todas las réplicas invocan otro grupo, Drago asegura que el grupo destino solamente recibe una petición. La programación del protocolo necesario para conseguir este objetivo en un sistema distribuido sin memoria compartida no es trivial [GAAM93] y oscurece innecesariamente las aplicaciones, lo que hace conveniente que se proporcione de forma automática.

En resumen, el entorno de tiempo de ejecución de Drago gestiona automáticamente todos los detalles de la interacción con las réplicas. Para los clientes remotos interactuar con un grupo replicado es igual que interactuar con un único agente. La programación de una réplica no necesita tener en cuenta las interacciones del grupo de réplicas con el exterior; simplemente debe asegurar que el código de las réplicas es determinista para así garantizar la consistencia de las réplicas[9].

2.5.2 Programación de aplicaciones cooperativas

Para facilitar la programación de aplicaciones cooperativas Drago posee las siguientes características:

- Características que facilitan la especificación de los servicios de un grupo cooperativo.
  - **Especificación variable del número de respuestas**, mediante formaciones irrestringidas. De esta forma pueden programarse aplicaciones cooperativas tolerantes a fallos (Drago proporciona soporte en tiempo de ejecución para que los miembros vivos se reasignen el trabajo del miembro que falló) y evitamos la necesidad de recompilar la especificación de un grupo cooperativo (y consecuentemente el cuerpo de todos sus miembros) si se ejecuta la aplicación con un número diferente de miembros.
- Características que facilitan la programación de los clientes del grupo:
  - **Atributo** *range*. En tiempo de ejecución Drago proporciona una extensión del atributo *range* que permite a los clientes remotos reservar espacio para almacenar todas las respuestas de un grupo cooperativo.
  - **Notificación de fallo**. En caso de que falle algún miembro del grupo antes de completar la operación solicitada, cuando el resto de los miembros complete la operación solicitada, Drago eleva la excepción *Group_Error* en el llamador.
- Características que facilitan la programación de los miembros del grupo:
  - **Notificación de fallos consistente**. Cuando muere un miembro de un grupo cooperativo, el resto de los miembros pueden recibir, si lo desean, notificación de tal muerte. La notificación se hace a través del punto de entrada especial de fallo, y todos los que la reciben lo hacen en el mismo orden en relación con el resto de los otros mensajes. Esto permite que un suplente sustituya al miembro que falló, que los miembros del grupo se reasignen de forma consistente el trabajo del miembro que falló o que liberen los recursos que tenía ocupados dicho miembro.

[9]La consistencia de réplicas es un requisito fuerte, pero no puede evitarse sin conocimiento de la semántica de los objetos a replicar [Coo85].

2.6 Resumen del lenguaje Drago

El modelo de programación de Drago se basa en el paradigma de comunicación con grupos. Un grupo Drago es un conjunto de agentes que comparten una semántica de aplicación común. Cada grupo es visto desde el exterior como una entidad individual que no permite a los clientes remotos ver su estructura interna ni las interacciones entre sus miembros. Drago proporciona dos abstracciones de grupo: abstracción de **grupo replicado** (un grupo de réplicas deterministas) y abstracción de **grupo cooperativo** (un conjunto de miembros que cooperan para conseguir un objetivo común). Drago proporciona estas abstracciones mediante la **especificación de grupo**. Una especificación de grupo contiene declaraciones de constantes, tipos, excepciones y puntos de entrada remotos. De esta forma, una especificación de grupo proporciona una interfaz que es compartida por todos los miembros del grupo (cuando un agente comparte una especificación de grupo decimos que es un *miembro* del grupo). Todo miembro de un grupo tiene visible todas las declaraciones contenidas en la especificación de dicho grupo y debe proporcionar todos los servicios remotos especificados en ella (mediante **puntos de entrada remotos**).

La *cita remota con un grupo* es el único mecanismo de comunicación remota y sincronización remota que proporciona Drago. Es una extensión al mecanismo de cita de Ada 83, para la cual la cita debe ser aceptada por todos los miembros vivos del grupo invocado. Drago permite que los miembros de un grupo cooperativo se comuniquen de forma transparente a los clientes remotos (mediante la *comunicación intragrupo*) y reciban notificaciones de fallo de los miembros del grupo que fallen.

El *agente* es la unidad de distribución de Drago. Un agente es un tipo de objeto abstracto: tiene estado interno (no accesible directamente desde el exterior) y operaciones remotas especiales (especificadas mediante los puntos de entrada remotos), que pueden ser llamadas remotamente desde otros agentes mediante *citas remotas con grupos*. El agente puede tener operaciones locales (procedimientos, funciones y puntos de entrada locales) y código de inicialización. Cada agente reside en un nodo de la red, aunque varios agentes pueden residir en el mismo nodo. Un programa distribuido Drago consta de varios agentes que residen en varios nodos de la red. Drago proporciona dos tipos de agentes: *agentes replicados* y *agentes cooperativos*. La principal diferencia entre ambos es que los agentes replicados no pueden tener tareas internas, mientras que los agentes cooperativos sí. Los agentes replicados solamente pueden ser miembros de grupos replicados. Los agentes cooperativos solamente pueden ser miembros de grupos cooperativos. Los agentes pueden tener puntos de entrada locales y remotos; pueden realizar citas remotas con grupos, así como aceptar y/o reencolar citas remotas.

Capítulo 3

Programación de aplicaciones tolerantes a fallos con Drago

Este capítulo presenta varios ejemplos escritos en Drago que tratan los siguientes aspectos de aplicaciones distribuidas tolerantes a fallos:

- Semáforos distribuidos tolerantes a fallos.
- Buzones distribuidos tolerantes a fallos.
- Memoria compartida distribuida tolerante a fallos
- Servidores de ficheros robustos.
- Servidores-clientes robustos.
- Problema de los filósofos (en versión distribuida tolerante a fallos).

3.1 Semáforos distribuidos tolerantes a fallos

Un semáforo S es una variable entera que tiene definidas las siguientes operaciones [BA90]:

- *Esperar(S)*: Si $S > 0$ entonces $S := S - 1$; si no, suspende la ejecución del proceso llamador (se dice que el proceso está suspendido en el semáforo S).
- *Señalar(S)*: Si hay procesos suspendidos en este semáforo despierta uno de ellos; si no, $S := S + 1$.

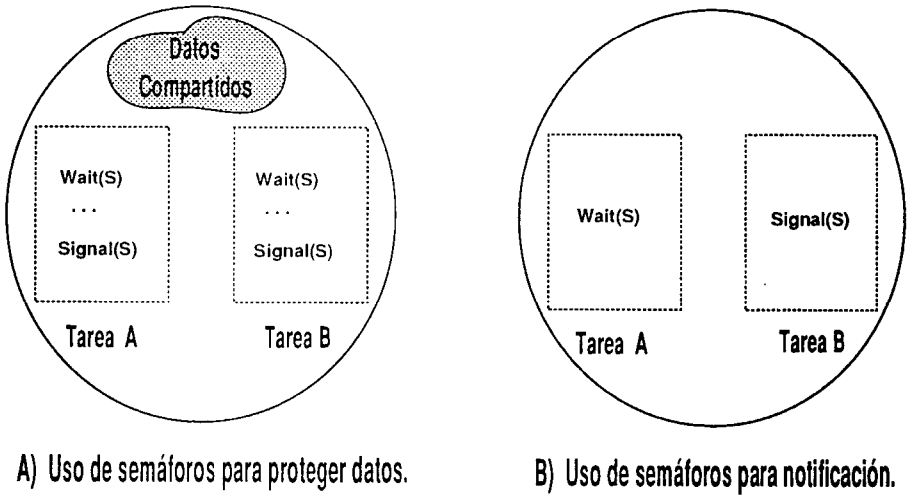


Figura 3.1: Dos aplicaciones de semáforos.

El semáforo tiene las siguientes propiedades:

1. *Esperar(S)* y *Señalar(S)* son instrucciones atómicas[1]. En particular, ninguna instrucción puede entrelazarse entre la comprobación de que $S > 0$ y el decremento de S o la suspensión del proceso llamador.
2. Todo semáforo debe tener un valor inicial no negativo.
3. La operación *Señalar(S)* debe despertar uno de los procesos suspendidos. La definición no especifica qué proceso.

Todo semáforo cumple la siguiente invariante: el número de operaciones *Esperar* completadas es, como máximo, el número de operaciones *Señalar* completadas más el valor inicial del semáforo. Un semáforo que solamente puede tener los valores 0 y 1 se denomina *semáforo binario*. Un semáforo que puede tener cualquier valor positivo se denomina *semáforo general*. Denominaremos *semáforo de difusión* a un semáforo en el que la operación *Señalar* despierta a todos los procesos suspendidos.

Los semáforos pueden utilizarse para proteger datos encapsulando el código que accede a los datos entre una pareja de llamadas (ver *A* en figura 3.1). Los semáforos pueden utilizarse también para notificar eventos. Una tarea llama a la operación *Esperar* para esperar a que ocurra el evento, y la otra avisa que ha ocurrido el evento llamando a la operación *Señalar* (ver *B* en figura 3.1).

Los semáforos fueron desarrollados por Dijkstra en 1968 [Dij68]. Normalmente se implementan mediante variables compartidas y se utilizan para sincronizar el acceso a otras variables compartidas. Sin embargo, los semáforos pueden utilizarse también para sincronizar procesos en un programa distribuido (por ejemplo, para resolver problemas de exclusión mutua tales como el bloqueo de ficheros o registros de una base de datos [Sch80])

[1]La notación original es *P(S)* para *Esperar(S)*, y *V(S)* para *Señalar(S)*. Las palabras *Esperar* y *Señalar* toman el significado de las palabras correspondientes en Holandés: *Passeren* y *Vrijmaken*.

Andrews [And91] presenta una implementación de semáforos distribuidos que utiliza relojes lógicos, colas y reconocimientos de mensajes, complicando bastante el código. Por contra, la implementación de semáforos distribuidos tolerantes a fallos en Drago es simple. Veamos en detalle la implementación de un semáforo binario tolerante a fallos, un semáforo general tolerante a fallos y un semáforo de difusión tolerante a fallos.

3.1.1 Semáforo binario tolerante a fallos

- *Este ejemplo presenta la especificación de un grupo replicado y el agente replicado correspondiente. También presenta los dos niveles de colas asociados a los agentes.*

El semáforo binario tolerante a fallos hardware se implementa en Drago mediante un grupo replicado. Todo grupo Drago (replicado o no) consta de una especificación de grupo (que proporciona la interfaz del servicio remoto) y los agentes que implementan el servicio remoto. La especificación de grupo puede contener declaraciones de constantes, tipos, excepciones y puntos de entrada remotos[2]. La cabecera de una especificación de grupo replicado debe comenzar con la palabra reservada **replicated**.

La especificación del semáforo binario tolerante a fallos solamente contiene la declaración de dos puntos de entrada remotos: *Esperar* y *Senalar*.

```
1:  replicated group specification Semaforo_Binario is
2:    entry Esperar;
3:    entry Sennalar;
4:  end Semaforo_Binario;
```

Es importante resaltar que la especificación de grupo no contiene el número de miembros del grupo (el número de réplicas que implementan el servicio) ni la ubicación de estos miembros en los nodos de la red. La forma de especificar esta información depende de la implementación.

El servicio de un grupo replicado debe implementarlo un agente replicado. La semántica de ejecución de Drago asegura que todas las réplicas reciben las mismas peticiones de cita remota en el mismo orden. Por ello, si el código que implementa el semáforo replicado es determinista, todas las réplicas se comportan exactamente igual. El código de nuestro semáforo binario es el siguiente[3]:

```
1:  for group Semaforo_Binario;
2:  replicated agent Replica_Semaforo_Binario is
3:  begin
4:    loop
5:       accept Esperar;
6:       accept Sennalar;
7:    end loop;
8:  end Replica_Semaforo_Binario;
```

[2]Las variables estan explícitamente prohibidas en una especificación de grupo.

[3]Se ha utilizado el identificador *Sennalar* en vez de *Señalar* porque los identificadores no pueden contener el carácter *ñ*.

Todo agente que es miembro de un grupo replicado debe especificar el nombre del grupo en la cabecera del agente mediante una cláusula **for group** (línea 1).

Inicialmente este agente replicado solamente acepta peticiones de cita remota en el punto de entrada remoto *Esperar* (línea 5). Si recibe varias, elige la más antigua y a continuación realiza la cita remota (ejecuta el bloque de sentencias de *Esperar*, que al ser nulo no hace nada) y finalmente completa la cita, enviando el valor de los parámetros de salida al cliente remoto (ninguno en este caso) y desbloqueando al cliente remoto. El cliente remoto desbloqueado es el propietario actual del semáforo. A continuación el agente replicado solamente acepta citas remotas en el punto de entrada remoto *Sennalar* (línea 6). De esta forma, cuando el propietario del semáforo binario quiera liberarlo deberá realizar una cita remota con el punto de entrada *Sennalar*. El agente replicado aceptará esta cita remota y a continuación volverá a aceptar la siguiente cita remota encolada en *Esperar* (que será del siguiente propietario del semáforo).

Debe resaltarse que las colas asociadas a los puntos de entrada remotos implementan de forma automática las colas asociadas a las operaciones del semáforo.

Esta implementación es tolerante a fallos porque cuando falla una réplica hay otra que proporciona el servicio a los clientes remotos (todas las réplicas tienen las mismas peticiones de cita remota en las mismas colas y en el mismo orden).

Los clientes del servicio replicado no necesitan tener en cuenta la replicación. Simplemente realizan las citas remotas como si el servicio estuviese implementado por un único miembro. Por ejemplo:

```
1:  with group Semaforo_Binario;
2:  agent Client is
3:  begin
4:     Semaforo_Binario.Esperar;
5:     -- Codigo ejecutado en exclusion mutua distribuida
6:     ...
7:     Semaforo_Binario.Sennalar;
8:  end Semaforo_Binario;
```

Todo cliente de un grupo replicado debe especificar el nombre de dicho grupo en la cabecera del agente dentro de una clausula **with group** (línea 1). Para realizar una cita remota con un grupo simplemente llama a un punto de entrada de la especificación del grupo (líneas 4 y 7).

Ejemplo de ejecución

Veamos un ejemplo de uso de este semáforo binario tolerante a fallos para conseguir exclusión mutua distribuida. Supongamos un grupo replicado compuesto por tres réplicas que implementan el semáforo binario descrito anteriormente. Supongamos dos clientes de este grupo replicado. Por último, supongamos que ambos clientes inician una cita remota con el punto de entrada remoto *Esperar* en el mismo instante (ver *A* en figura 3.2). Drago serializa las citas remotas en el mismo orden en todos los miembros del grupo y las almacena como citas remotas pendientes hasta que el agente las solicite.

Puesto que el agente no tiene aún ninguna cita remota encolada en el punto de entrada

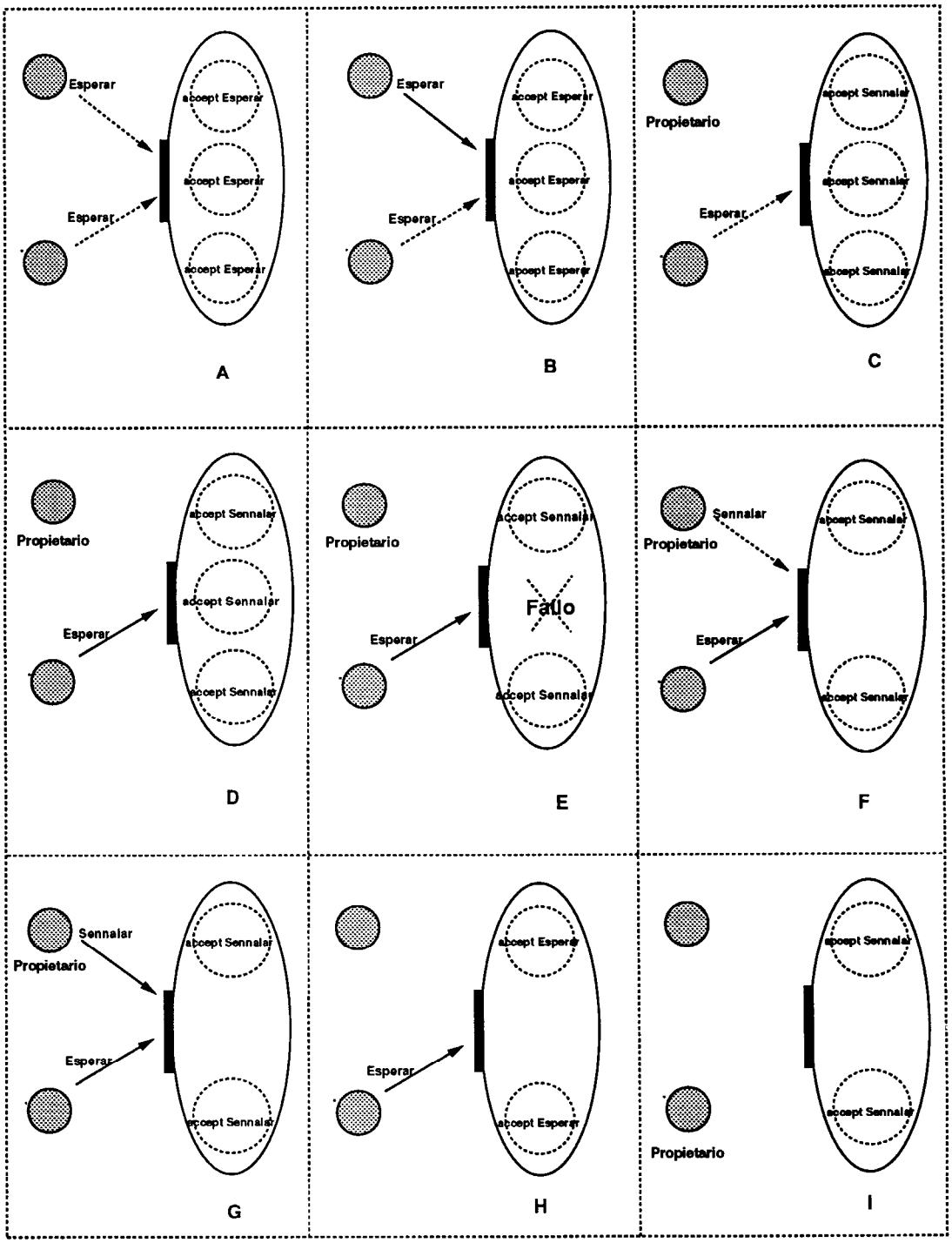


Figura 3.2: Ejemplo de ejecución del semáforo binario tolerante a fallos (I).

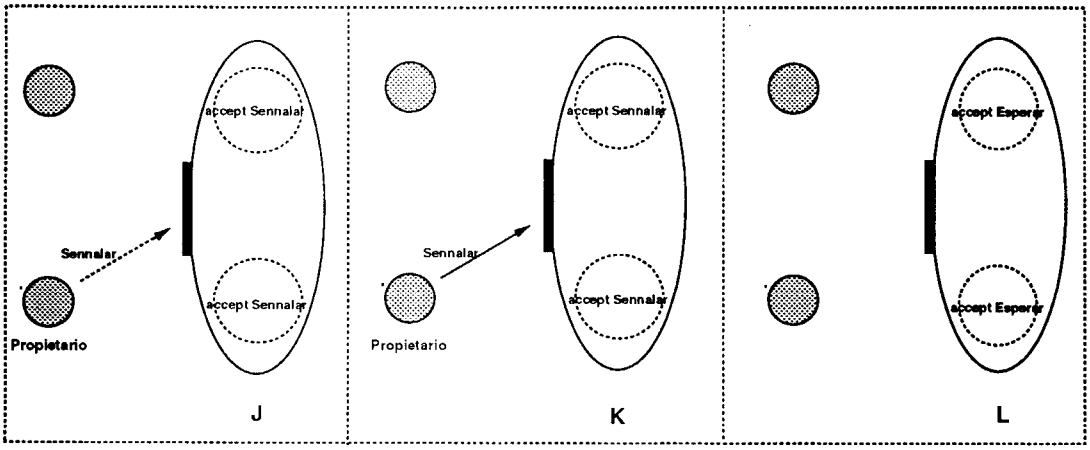


Figura 3.3: Ejemplo de ejecución del semáforo binario tolerante a fallos (II).

Esperar, cuando ejecuta la sentencia **accept Esperar**, pasa la cita remota pendiente más antigua a la cola correspondiente (ver *B* en figura 3.2) y a continuación acepta la cita con el cliente remoto (ejecuta el bloque de sentencias asociado a *Esperar* —que al ser nulo no hace nada— y completa la cita remota desbloqueando al cliente remoto). El llamador remoto desbloqueado es el propietario del semáforo binario tolerante a fallos (ver *C* en figura 3.2). Ahora todas las réplicas solamente aceptan citas remotas en el punto de entrada *Sennalar* y, por ello, el agente replicado encola la cita remota pendiente en la cola asociada a *Esperar* (ver *D* en figura 3.2).

Supongamos que una de las réplicas falla (ver *E* en figura 3.2). El fallo puede tolerarse porque hay dos réplicas que mantienen el estado del semáforo (las réplicas vivas mantienen encolada la cita remota *Esperar* y esperan una cita remota en el punto de entrada *Sennalar*). Los clientes no ven el fallo de la réplica.

Cuando el propietario del semáforo binario decide liberarlo, realiza una cita remota con el grupo en el punto de entrada remoto *Sennalar* (ver *F* en figura 3.2). Las réplicas aceptan esta cita remota (ver *G* en figura 3.2) porque es la siguiente cita remota con el único punto de entrada remoto del cual acepta citas remotas. Por ello, ejecutan el bloque de sentencias de *Sennalar* (que al ser nulo no hace nada) y completan la cita remota (desbloqueando al antiguo propietario del semáforo —ver *H* en figura 3.2—).

El agente replicado acepta de nuevo una cita remota en el punto de entrada *Esperar*. Como tiene una cita remota encolada en *Esperar*, la acepta (ver *I* en figura 3.2). Cuando completa la cita remota, el cliente remoto se desbloquea (es el nuevo propietario de este semáforo binario tolerante a fallos).

Al igual que hizo el primer propietario, cuando el nuevo propietario desee liberar el semáforo deberá realizar una cita remota en *Sennalar* (ver *J* en figura 3.3). El agente replicado aceptará la cita remota (ver *K* en figura 3.3) y repetirá el bucle para aceptar la siguiente cita remota en *Esperar* (ver *L* en figura 3.3) (el sistema replicado estará de nuevo en su estado inicial). El servicio se mantiene disponible mientras quede al menos una réplica viva (ver figura 3.4).

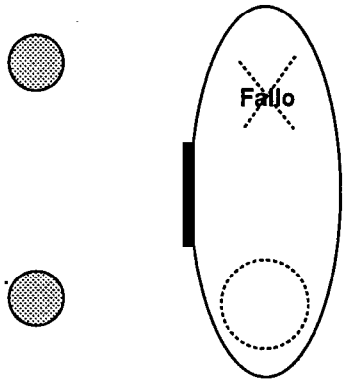


Figura 3.4: El servicio está disponible mientras quede al menos una réplica viva.

Esta implementación tolera fallos hardware en los nodos donde se ejecutan las réplicas del semáforo binario. Sin embargo, no tolera fallos en los nodos donde se ejecutan los clientes remotos que utilizan este semáforo replicado (si el propietario del semáforo falla, el semáforo se queda bloqueado). Si el usuario desea tolerar fallos en los clientes remotos, debe implementarlos mediante grupos replicados.

3.1.2 Semáforo general tolerante a fallos

- *Este ejemplo presenta la sentencia* **select**, *que permite al agente la espera selectiva de una o más alternativas. La elección puede depender de condiciones asociadas a cada alternativa.*

La especificación de grupo asociada a un semáforo general tolerante a fallos es idéntica a la especificación del semáforo binario tolerante a fallos:

```
replicated group specification Semaforo_General is
   entry Esperar;
   entry Sennalar;
end Semaforo_General;
```

El semáforo general tolerante a fallos puede implementarse de la siguiente forma:

```
 1:  for group Semaphore;
 2:  replicated agent Replica_Semaforo_General is
 3:     N:Natural := 2;
 4:  begin
 5:     loop
 6:        select
 7:           when N > 0 =>
 8:           accept Esperar do
 9:              N := N - 1;
10:           end Esperar;
11:        or
```

```
12:              accept Senalar do
13:                  N := N + 1;
14:              end Senalar;
15:          end select;
16:      end loop;
17:  end Replica_Semaforo_General;
```

Inicialmente el semáforo se inicializa al número de clientes que pueden ejecutarse en exclusión mutua distribuida (línea 3). Cuando el semáforo tiene un valor positivo (línea 7) y acepta una cita remota en *Esperar*, el semáforo se decrementa de forma atómica y se completa la cita remota (línea 10), desbloqueando así al cliente remoto.

Cuando el semáforo está a cero y un cliente remoto realiza una cita remota en *Esperar*, la cita remota se encola en la cola asociada a dicho punto de entrada hasta que el agente pueda aceptar una cita remota en *Senalar* (líneas 12..14) que incremente de forma atómica el semáforo (línea 13).

Es importante resaltar que la implementación Drago de los semáforos distribuidos tolerantes a fallos tiene la misma sintaxis que la solución mediante tareas de Ada 83 [Bar87]. Esta característica de Drago facilita al programador reutilizar código Ada 83 para implementar aplicaciones distribuidas.

Ejemplo de ejecución

Veamos un ejemplo del uso de este semáforo general para exclusión mutua distribuida. Supongamos un grupo replicado compuesto por dos réplicas. Consideremos tres clientes de este grupo replicado. De acuerdo con la implementación descrita anteriormente, inicialmente todas las réplicas inicializan el semáforo con el valor entero 2, permitiendo así que dos clientes sean simultáneamente propietarios del semáforo.

Ahora el agente replicado ejecuta la sentencia **select**. La semántica de ejecución de la sentencia **select** hace que el agente evalúe todas las condiciones donde hay peticiones de cita remota encoladas. Si hay varias alternativas disponibles, el agente elige la petición de cita remota encolada más antigua cuya condición esté *abierta*[4]. Si el agente no tiene ninguna alternativa disponible, Drago pasa la cita remota pendiente más antigua a la cola correspondiente[5], incrementa el valor del atributo **count** correspondiente y el agente reevalúa las condiciones[6]. De acuerdo con esta semántica, nuestro agente replicado evalúa las condiciones asociadas a las alternativas de la sentencia **select**. Ambas alternativas están *abiertas* y puesto que el agente no ha recibido aún ninguna cita remota, simplemente espera la siguiente petición de cita remota (ver *A* en figura 3.5).

Supongamos que todos los clientes inician simultáneamente una cita remota en *Esperar* (ver *B* en figura 3.5). De forma similar al ejemplo anterior, Drago serializa las peticiones de cita

[4]Una alternativa del *select* está *abierta* cuando no especifica ninguna condición o cuando el resultado de evaluarla es el valor lógico *True*.

[5]Cuando el agente recibe una petición de cita remota, Drago la almacena en el nivel de citas remotas pendientes hasta que el agente solicite la siguiente cita remota pendiente.

[6]Este comportamiento asegura que todas las réplicas actualizan de forma consistente el valor del atributo **count**.

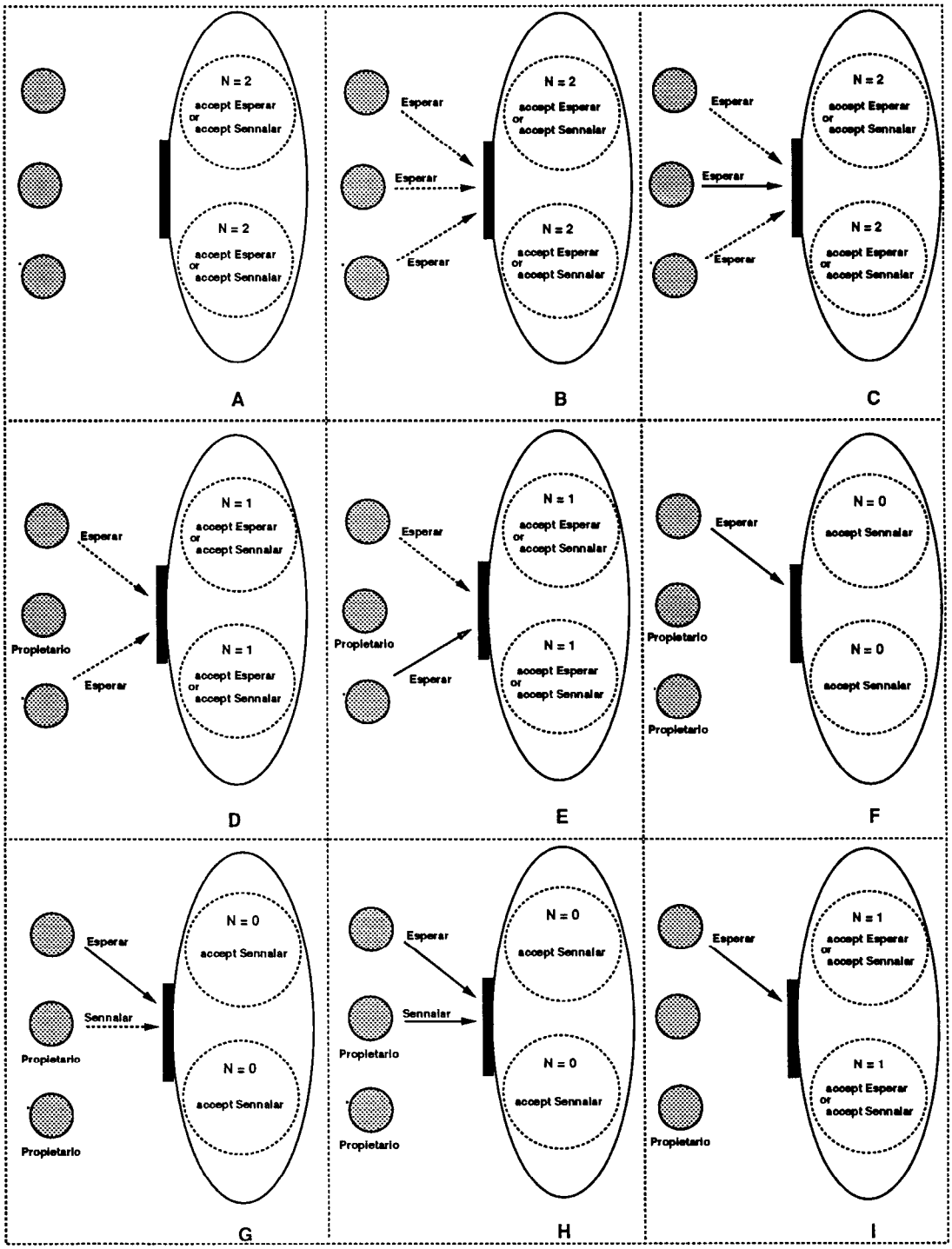


Figura 3.5: Ejemplo de ejecución del semáforo general tolerante a fallos (I).

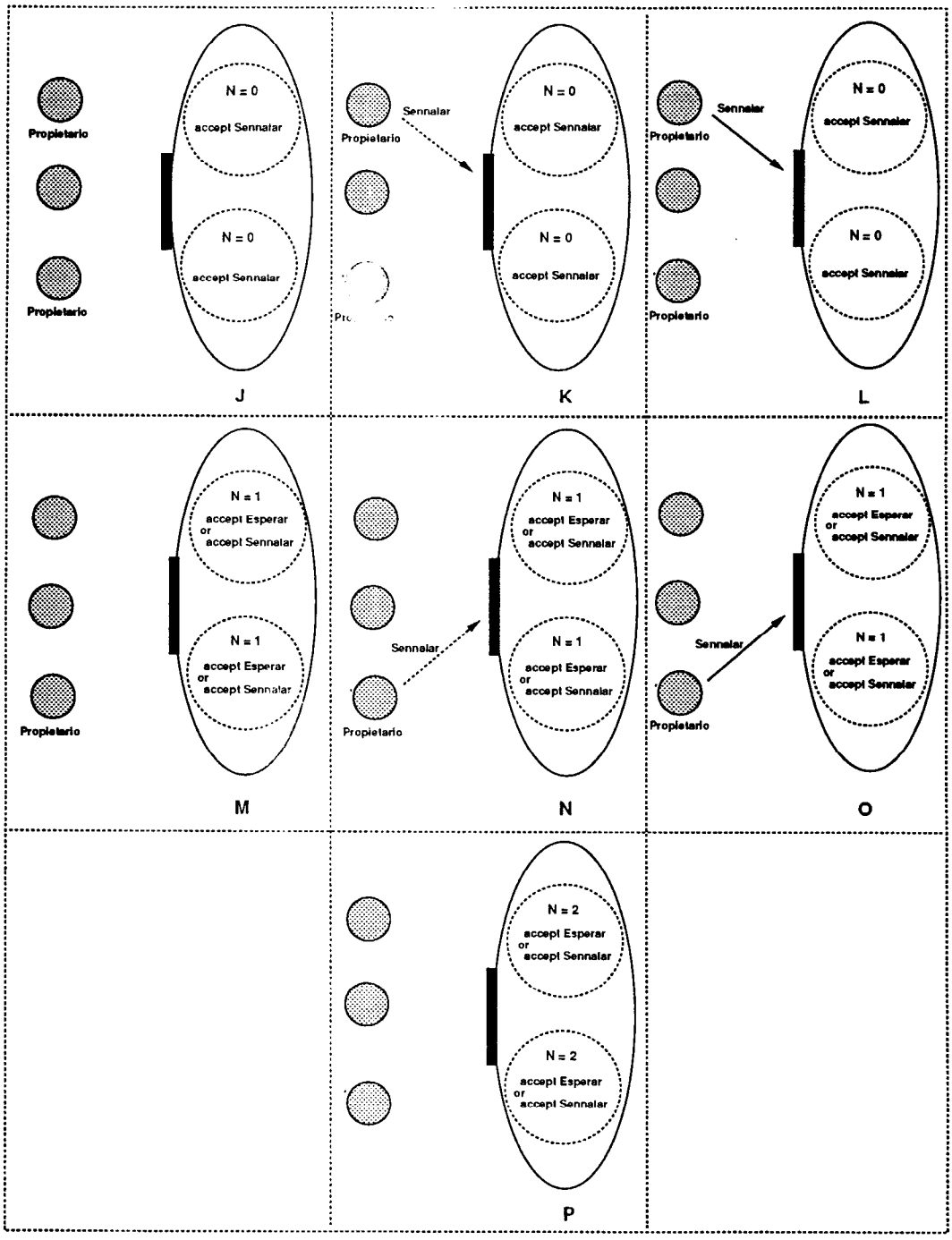


Figura 3.6: Ejemplo de ejecución del semáforo general tolerante a fallos (II).

remota en el mismo orden en todas las réplicas y las almacena como peticiones de cita remota pendientes. Puesto que el agente está esperando una cita remota, la petición de cita remota pendiente más antigua se encola en la cola correspondiente (ver *C* en figura 3.5). El agente acepta la cita remota y ejecuta el bloque de sentencias asociado a *Esperar* (decrementando el semáforo) y, finalmente, completa la cita remota (desbloqueando al cliente remoto). Este es el primer propietario del semáforo (ver *D* en figura 3.5).

El agente replicado ejecuta de nuevo la sentencia **select**. Puesto que no tiene ninguna petición de cita remota encolada, pasa la siguiente petición de cita remota pendiente a la cola correspondiente (*Esperar*), reevalúa las condiciones asociadas a las alternativas que tienen alguna petición de cita remota encolada (solamente hay una cita remota encolada en *Esperar*) y puesto que la condición de *Esperar* está abierta, el agente acepta la cita remota (ver *E* en figura 3.5), ejecuta la secuencia de sentencias asociada a *Esperar* (decrementando atómicamente el semáforo) y completa la cita remota (desbloqueando al llamador remoto). Este es el segundo propietario del semáforo (ver *F* en figura 3.5).

El agente replicado ejecuta de nuevo la sentencia **select** y de nuevo no tiene ninguna petición de cita remota encolada. Por lo tanto pasa la siguiente petición de cita remota pendiente a la cola correspondiente (*Esperar* —ver *F* en figura 3.5—) y reevalúa las condiciones asociadas a las alternativas que tienen alguna petición de cita remota encolada. Esta vez la condición está cerrada[7] y el agente vuelve a solicitar la siguiente petición de cita remota pendiente. Al no haber ninguna, simplemente espera hasta que reciba alguna.

Cuando alguno de los propietarios realiza una cita remota en *Sennalar* (ver *G* en figura 3.5), la petición de cita remota se encola en la cola correspondiente(ver *H* en figura 3.5). El agente reevalúa las condiciones asociadas a las alternativas que tienen alguna cita remota encolada, acepta la cita remota en *Sennalar* (incrementando así atómicamente el semáforo) y completa la cita remota (desbloqueando al llamador remoto). En este instante el semáforo tiene solamente un propietario (ver *I* en figura 3.5).

El agente replicado ejecuta de nuevo la sentencia **select**, reevalúa de nuevo todas las condiciones asociadas a las alternativas que tienen alguna cita remota encolada y acepta la cita remota a *Esperar* (decrementando atómicamente el semáforo). De esta forma completa la cita y desbloquea al llamador remoto. Este es el nuevo propietario del semáforo (ver *J* en figura 3.6).

De forma similar a lo expuesto anteriormente, cuando los propietarios deciden liberar el semáforo, simplemente realizan una cita remota con el grupo en el punto de entrada *Sennalar* para que el semáforo incremente su valor (ver *K*..*P* en figura 3.6).

Este servicio replicado tolera fallos hardware porque todas las réplicas tienen encoladas las mismas citas remotas en el mismo orden en todas las colas.

[7] El resultado de evaluarla es el valor lógico *False*.

3.1.3 Semáforo de difusión tolerante a fallos

- *Este ejemplo presenta la sentencia* **requeue** *de Drago que se utiliza para redirigir una cita remota aceptada a otro punto de entrada. También muestra la semántica del atributo* **count** *asociado a los puntos de entrada remotos.*

Este ejemplo presenta un semáforo de difusión[8]. Los clientes remotos esperan a que ocurra algún evento y cuando ocurre se despiertan todos y se reinicializa el evento. La dificultad de este ejemplo estriba en no despertar a los clientes remotos que llamen al punto de entrada *Esperar* después que ha ocurrido el evento, pero antes de reiniciarlo. La especificación del grupo es la siguiente:

```
replicated group specification Semaforo_Difusion is
   entry Esperar;
   entry Sennalar;
end Semaforo_Difusion;
```

El semáforo replicado tolerante a fallos puede implementarse de la siguiente forma:

```
 1:  for group Semaforo_Difusion;
 2:  replicated agent B_Semaphore_Replica is
 3:    Evento:Boolean:= False;
 4:    entry Reinicializar;
 5:  begin
 6:    loop
 7:      select
 8:         when Evento =>
 9:         accept Esperar;
10:      or
11:         accept Sennalar do
12:            if Esperar'count>0 then
13:               Evento:=True;
14:               requeue Reinicializar;
15:            end if;
16:         end Sennalar;
17:       or
18:         when Esperar'count=0 =>
19:         accept Reinicializar do
20:            Evento:=false;
21:         end Reinicializar;
22:       end select;
23:    end loop;
24:  end B_Semaphore_Replica;
```

La variable lógica *Evento* (línea 3) está normalmente a *False* y solamente es cierta mientras se están atendiendo las peticiones de cita remota encoladas en *Esperar*. El punto de entrada remoto *Esperar* (línea 9) mantiene en su cola a todos los clientes remotos que esperan a que *Evento* sea *True*.

[8] Semáforo en el que la operación *Sennalar* despierta a todos los procesos suspendidos.

La condición asociada al punto de entrada remoto *Sennalar* está siempre *abierta* (línea 11) y por ello siempre se atienden las citas remotas en *Sennalar*. Si no hay peticiones de cita remota encoladas en *Esperar* (o sea, no hay clientes remotos esperando por el evento) entonces no tiene nada que hacer y simplemente completa la cita remota. Si hay peticiones de cita remota encoladas en *Esperar* debe atenderlas sin atender a ninguna petición de cita remota posterior. Para ello abre la condición asociada a *Esperar* inicializando la variable *Evento* a *True* (línea 13), indicando así que ha ocurrido el evento, y reencola la cita remota en el punto de entrada *Reinicializar* (línea 14)[9]. La semántica de la sentencia **requeue** hace que el agente reencole la cita remota en el punto de entrada especificado y complete el bloque de sentencias de *Sennalar*.

Al completar el reencolamiento el agente repite el bucle principal (líneas 6..23) y ejecuta de nuevo la sentencia **select**. Por lo tanto evalúa de nuevo las condiciones asociadas a las alternativas que tienen alguna petición de cita remota encolada. En este caso tiene peticiones de cita remota encoladas en *Esperar* y una petición de cita remota encolada en *Reinicializar* (la que acaba de reencolarse). La condición de *Esperar* está *abierta*, pero la condición de *Reinicializar* está *cerrada* (puesto que hay citas remota encoladas en *Esperar*). Por lo tanto el agente acepta la cita remota más antigua de *Esperar*, que al ser nula simplemente completa la cita y desbloquea al llamador remoto. Este proceso de evaluación y aceptación se repite hasta que se han aceptado todas las peticiones de cita remota encoladas en *Esperar*. Ahora la condición asociada a *Reinicializar* está *abierta* y se acepta la petición de cita remota del llamador original que fue reencolado. El bloque de sentencias asociado a *Reinicializar* inicializa la variable *Evento* a *False* y completa la cita remota. En este punto el sistema está de nuevo en su estado inicial.

Nótese que si el agente replicado recibe alguna petición de cita remota mientras completa las que tenía encoladas antes de ocurrir el evento, la almacena como pendiente. Este ejemplo muestra la razón subyacente por la que los agentes replicados no tienen problemas de condiciones de carrera[10].

Ejemplo de ejecución

Veamos un ejemplo de uso de este semáforo con difusión. Supongamos un grupo replicado compuesto por tres réplicas. Consideremos cuatro clientes de este grupo replicado. De acuerdo con la implementación descrita anteriormente, inicialmente todas las réplicas esperan hasta recibir alguna petición de cita remota (ya que las réplicas no tienen aún ninguna petición de cita remota encolada).

Supongamos que un cliente remoto inicia una cita remota con el grupo en el punto de entrada remoto *Esperar* (ver *B* en figura 3.7). El agente replicado encola la petición de cita remota en la cola correspondiente (ver *C* en figura 3.7) y evalúa las condiciones asociadas a las alternativas que tienen alguna petición de cita remota encolada (en este caso solamente *Esperar*). Puesto que la condición de *Esperar* está *cerrada*, el agente espera otra petición de cita remota.

[9] El punto de entrada *Reinicializar* es un punto de entrada local que está declarado dentro del agente (línea 4).

[10] Este ejemplo, así como los comentarios, están basados en el ejemplo *broadcast Sennalar* de Barnes [Bar93] para Ada9X.

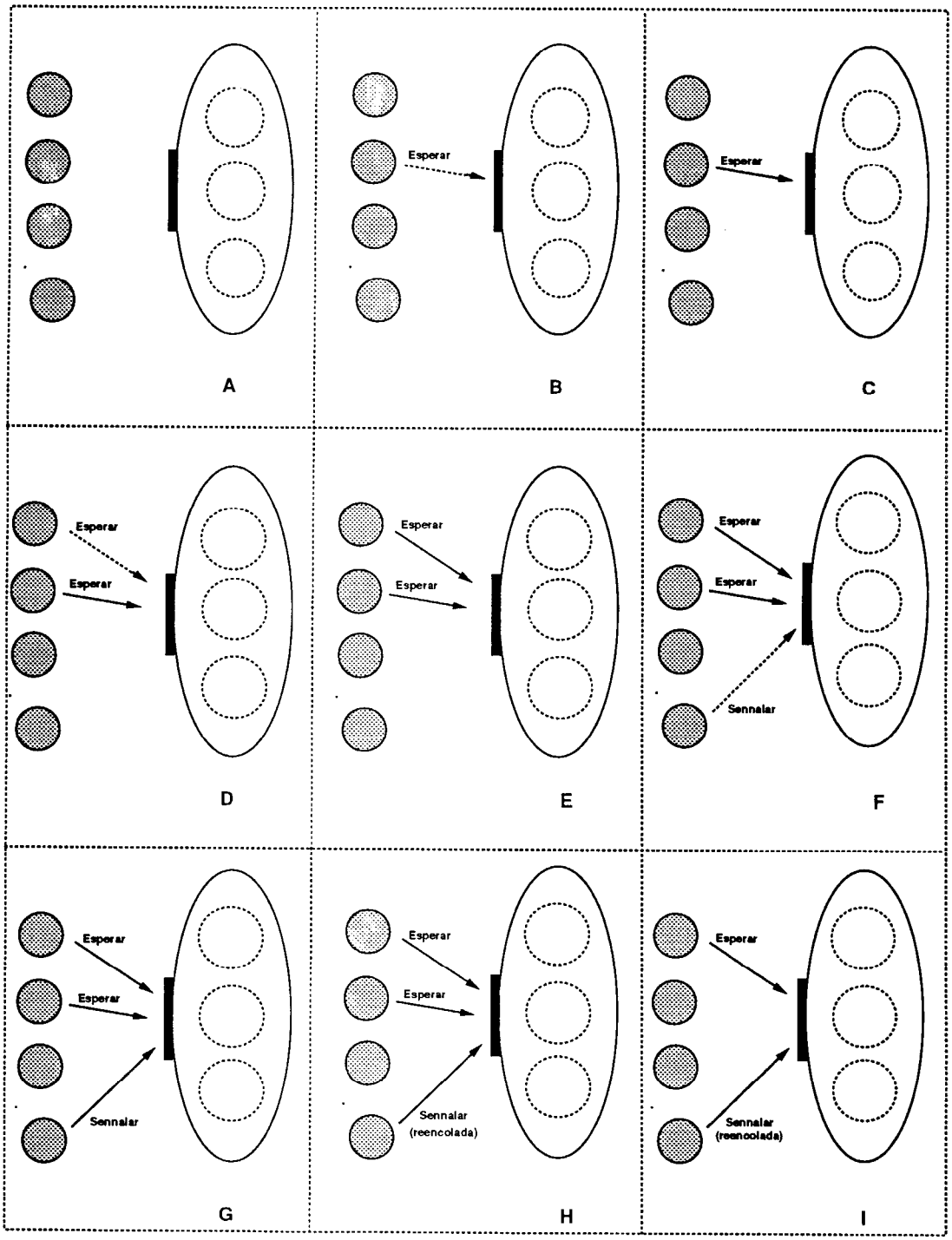


Figura 3.7: Ejemplo de ejecución del semáforo de difusión tolerante a fallos (I).

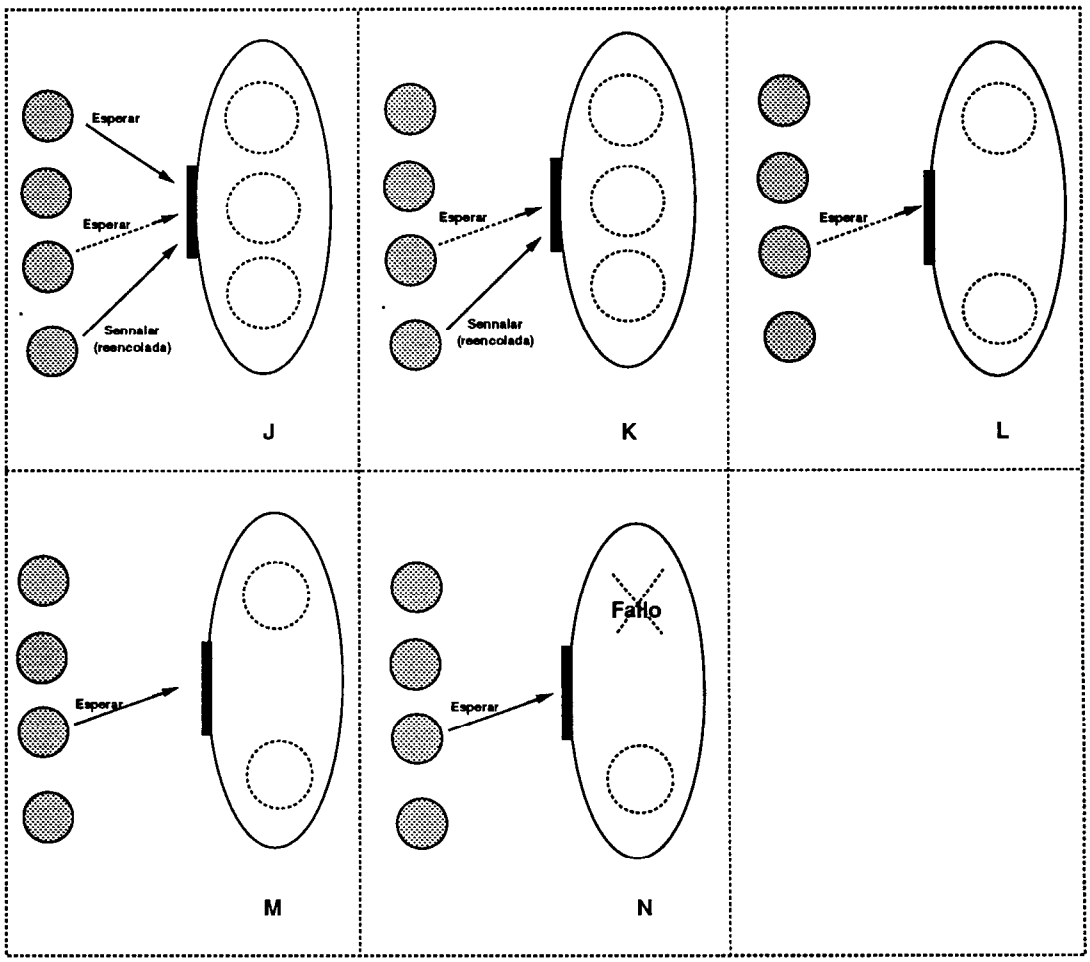


Figura 3.8: Ejemplo de ejecución del semáforo de difusión tolerante a fallos (II).

Supongamos que otro cliente remoto inicia una cita remota en *Esperar* (ver *D* en figura 3.7). De forma similar, la petición de cita remota se encola en *Esperar* (ver *E* en figura 3.7), se evalúa de nuevo la condición de *Esperar* y al estar *cerrada* el agente espera de nuevo otra petición de cita remota.

Supongamos que un cliente remoto inicia una cita remota en *Sennalar* (ver *F* en figura 3.7). La petición de cita remota se encola en *Sennalar* (ver *G* en figura 3.7) y se evalúan las condiciones asociadas a los puntos de entrada que tienen alguna petición de cita remota encolada (en este caso *Sennalar* y *Esperar*). La condición asociada a *Esperar* sigue estando *cerrada*, pero la condición asociada a *Sennalar* está *abierta*. Por lo tanto el agente replicado acepta la cita remota en *Sennalar* y comienza a ejecutar la secuencia de sentencias asociada a *Sennalar*. Como hay peticiones de cita remota encoladas en *Esperar*, inicia la variable *Evento* a *True* y reencola la cita remota en el punto de entrada local *Reinicializar* (ver *H* en figura 3.7).

El agente ejecuta de nuevo la sentencia **select** y reevalúa las condiciones asociadas a las alternativas que tienen alguna cita remota encolada. Ahora la condición asociada a *Esperar* está *abierta*, mientras que la de *Reinicializar* está *cerrada*. Por lo tanto, el agente acepta la cita remota más antigua encolada en *Esperar*, ejecuta la secuencia de sentencias de *Esperar* (que al ser nula no hace nada) y completa la cita remota, desbloqueando al llamador remoto (ver *I* en figura 3.8).

Supongamos que otro cliente remoto inicia ahora una cita remota en *Esperar*. Al igual que en los casos anteriores, Drago almacena petición de cita remota en el nivel de peticiones de cita remota pendientes (ver *J* en figura 3.8).

El agente ejecuta de nuevo la sentencia **select**, reevalúa las condiciones del **select** y acepta la siguiente petición de cita remota en *Esperar* (ver *K* en figura 3.8).

El agente ejecuta de nuevo la sentencia **select**, reevalúa las condiciones del **select** que tienen alguna petición de cita remota encolada (solamente *Reinicializar* tiene una petición de cita remota encolada). La condición de *Reinicializar* está ahora *abierta*, y por lo tanto acepta la cita remota (la cita remota del llamador remoto que originalmente inició la cita remota en *Sennalar*). El agente ejecuta el bloque de sentencias asociado a *Reinicializar*, reinicializando la variable *Evento* al valor lógico *False* y completando la cita (desbloqueando así al llamador remoto —ver *L* en figura 3.8—).

Finalmente, el agente ejecuta de nuevo la sentencia **select** y puesto que no tiene ninguna petición de cita remota encolada que pueda aceptar, encola la petición de cita remota pendiente en la cola de *Esperar* (ver *M* en figura 3.8).

El servicio está disponible mientras quede al menos una réplica viva (ver *N* en figura 3.8).

3.2 Buzón distribuido tolerante a fallos

Un buzón es un lugar donde se almacena un cierto número de mensajes (cantidad que normalmente se especifica cuando se crea el buzón). Cuando un proceso intenta enviar un mensaje a un buzón que está lleno, se suspende hasta que se extraiga un mensaje del buzón. El buzón puede utilizarse para resolver el problema del productor-consumidor.

Los buzones pueden utilizarse también en los sistemas distribuidos para comunicación entre procesos, aunque en este caso debe analizarse la tolerancia a fallos del sistema. Por ejemplo, supongamos que tenemos muchos productores y consumidores y un único buzón. Si el nodo donde se ejecuta el buzón falla, todos los productores y consumidores se quedan incomunicados. Por tanto, el buzón es un punto crítico que necesita ser replicado. El fallo de un productor o un consumidor puede no ser crítico. Si se desea tolerar fallos en los productores y consumidores, deben replicarse.

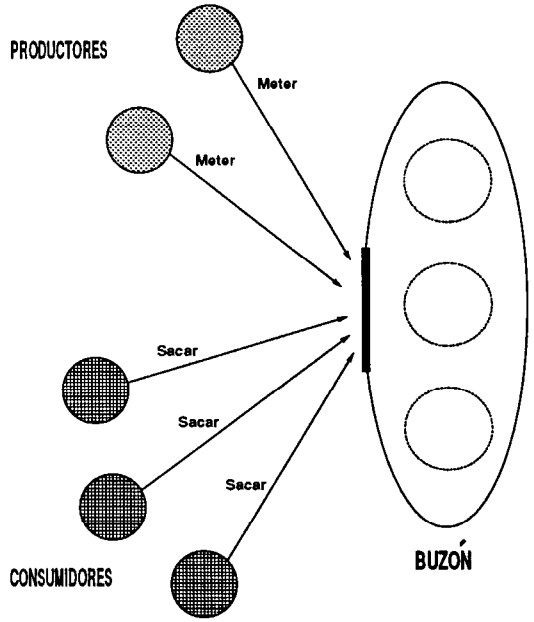


Figura 3.9: Buzón distribuido tolerante a fallos.

Veamos en detalle dos implementaciones de buzones distribuidos. La primera implementa un buzón distribuido tolerante a fallos con un buffer de un único elemento. La segunda es la implementación de un buzón distribuido tolerante a fallos con un buffer de N mensajes.

3.2.1 Buzón tolerante a fallos

- *Este ejemplo muestra cómo se pasan y reciben parámetros remotos.*

La especificación de grupo del buzón tolerante a fallos podría ser la siguiente:

```
replicated group specification Buzon is
   type Item is ...
   entry Meter(X:in Item);
   entry Sacar(X:out Item);
end Buzon;
```

La especificación del grupo contiene la declaración del tipo *Item* asociado al mensaje que se almacena en el buzón. El buzón proporciona dos puntos de entrada: uno para poner un mensaje en el buzón (*Put*) y uno para recoger un mensaje del buzón (*Get*). El agente replicado que implementa este buzón es el siguiente:

```
for group Buzon;
replicated agent Replica_Buzon is
   Buffer:Item;
begin
   loop
     accept Meter(X:in Item) do
        Buffer:=X;
     end Meter;
     accept Sacar(X:out Item) do
        X:=Buffer;
     end Sacar;
   end loop;
end Replica_Buzon;
```

Inicialmente el agente replicado solamente acepta una cita remota en el punto de entrada *Meter*. Al realizarse la cita almacena en *buffer* el contenido del parámetro de entrada *X* (el mensaje). A continuación solamente acepta una cita remota en el punto de entrada *Sacar* y al completar la cita devuelve en el parámetro de salida *X* el contenido actual de *Buffer*. Los clientes de este grupo replicado realizan las citas remotas como si el servicio lo estuviese implementando una única réplica. Por ejemplo:

```
with group Buzon;
agent Productor is
  A:Buzon.Item;
begin
  loop
    -- Producir A
    ...
    Buzon.Meter(A);
  end loop;
end Productor;
```

```
with group Buzon;
agent Consumidor is
  A:Buzon.Item;
begin
  loop
    Buzon.Sacar(A);
    -- Consumir A
    ...
  end loop;
end Consumidor;
```

3.2.2 Buzón general tolerante a fallos

La especificación de grupo asociada al buzón general tolerante a fallos es idéntica a la presentada en el ejemplo anterior:

```
replicated group specification Buzon_General is
   type Item is ...;
   entry Meter(X:in Item);
   entry Sacar(X:out Item);
end Buzon_General;
```

El agente replicado que implementa el buzón general (buzón con un buffer de *N* mensajes) tolerante a fallos es el siguiente:

```
for group Buzon_General;
replicated agent Replica_Buzon_General is
   N       :constant:=10;
   Buffer  :array(1..N) of Item;
   I,J     :positive range 1..N:=1;
   Cuenta  :natural  range 0..N:=0;
begin
   loop
     select
        when Cuenta<N =>
        accept Meter(X:in Item) do
           Buffer(I):=X;
        end Meter;
        I:=I mod N+1;
        Cuenta:=Cuenta+1;
     or
        when Cuenta>0 =>
        accept Sacar(X:out Item) do
           X:=Buffer(J);
        end Sacar;
        J:=J mod N+1;
        Cuenta:=Cuenta-1;
     end select;
   end loop;
end Replica_Buzon_General;
```

La variable *buffer* implementa el espacio de almacenamiento del buzón. Las variables *I* y *J* limitan la parte de *buffer* que está siendo utilizada y *Cuenta* indica cuantos mensajes hay en el *buffer*. Nótese lo evidentes que resultan las condiciones asociadas a los puntos de entrada. Una cita remota en *Sacar* solamente puede aceptarse cuando hay mensajes en el buzón, y una cita remota en *Meter* solamente puede ser aceptarse cuando el buzón no está lleno.

Al igual que ocurre en Ada 83, la actualización de las variables *I*, *J* y *Cuenta* no se realiza dentro de la cita remota. Esto permite al llamador remoto continuar tan pronto como sea posible[11].

Lo más importante a resaltar de este ejemplo es que esta solución es sintácticamente idéntica a la solución de Ada 83 mediante tareas, pero al implementarla con Drago el buzón está distribuido y tolera fallos.

3.3 Memoria compartida distribuida tolerante a fallos

Queremos implementar una variable compartida distribuida que tolere fallos hardware. Suponemos que el acceso a la variable replicada se proporciona mediante dos operaciones atómicas[12]: *Leer* y *Escribir*. Deseamos que los clientes remotos de esta variable compartida no necesiten tratar con detalles de la replicación. También deseamos que la variable compartida tenga una

[11] En este ejemplo esta característica no es demasiado representativa ya que la actualización de una variable consume muy poco tiempo en relación con la cita remota, pero en otros ejemplos esta actualización podría suponer un número mucho mayor de instrucciones.

[12] No se puede realizar una escritura mientras se está realizando una lectura y viceversa.

semántica fuerte: una lectura de la variable debe proporcionar el último valor escrito en ella. Esta aplicación puede implementarse en Drago mediante un grupo replicado. Por ejemplo:

```
replicated group specification Variable_Compartida is
    type Var_Type is ...;
    entry Escribir(V:in Var_Type);
    entry Leer    (V:out Var_Type);
end Variable_Compartida;
```

El agente que implementa esta variable distribuida tolerante a fallos es el siguiente:

```
for group Variable_Compartida;
replicated agent Replica_Variable_Compartida is
   Copia_Local:Var_Type;
begin
   accept Escribir(V:in Var_Type) do
      Copia_Local:=V;
   end Escribir;
   loop
      select
        accept Escribir(V:in Var_Type) do
           Copia_Local:=V;
        end Escribir;
      or
        accept Leer(V:out Var_Type) do
           V:=Copia_Local;
        end Leer;
      end select;
   end loop;
end Replica_Variable_Compartida;
```

Inicialmente el agente replicado solamente acepta una cita remota en el punto de entrada *Escribir* (porque la variable replicada aún no ha sido inicializada). Cuando recibe una petición de cita remota en *Escribir*, la acepta, almacena el valor de la variable en *Copia_Local* y completa la cita remota. A continuación acepta citas remotas en *Leer* y *Escribir* en orden de antiguedad.

La semántica de la cita remota de Drago asegura que todas las réplicas reciben las mismas peticiones de cita remota en el mismo orden, asegurando así la consistencia de las réplicas. Además, puesto que el código del agente es determinista, todas las réplicas hacen exactamente lo mismo.

3.4 Servidor de ficheros robusto

- *Este ejemplo presenta cómo la especificación de grupos replicados permite especificar interfaces completas que incluyen tipos privados y excepciones remotas.*

Los sistemas de ficheros distribuidos suelen ofrecer un servicio que permite mantener múltiples copias de ficheros en servidores de ficheros que se ejecutan sobre diferentes nodos. Las razones de ello varían de un sistema de ficheros a otro, pero principalmente son las siguientes[Tan92]:

- Incrementar la fiabilidad del sistema teniendo copias de seguridad independientes de cada fichero. Si un servidor falla, de forma temporal o permanente, el usuario no pierde ninguna información. En muchas aplicaciones esta propiedad es muy deseable.
- Incrementar la disponibilidad de los datos, permitiendo que los accesos a los ficheros replicados continúen en presencia de fallos.

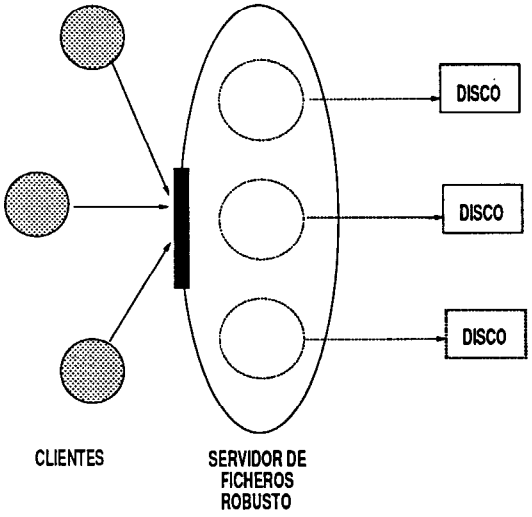


Figura 3.10: Servidor de ficheros robusto.

Drago permite implementar fácilmente un servidor de ficheros robusto. La especificación del grupo asociado a dicho servidor robusto podría ser la siguiente:

```
replicated group specification Servidor_Ficheros_Robusto is
   type Elemento_Fichero  is ...;
   type Tipo_Fichero is limited private;
   type Modo_fichero is (IN_FILE,OUT_FILE);

   entry Crear (Fichero: in out Tipo_Fichero;
                Modo   : in     File_mode := OUT_FILE;
                Nombre : in     String    := "";
                Forma  : in     String    :="");
   entry Abrir (...);
   entry Cerrar(...);
   ...

   Error_de_Estado: exception;
   Error_de_Modo  : exception;
private
   type Tipo_Fichero is ...;
end Servidor_Ficheros_Robusto;
```

Tal y como puede apreciarse, la especificación de un grupo permite declarar tipos privados y excepciones remotas. Además, los puntos de entrada remotos pueden especificar valores por defecto para los parámetros de modo **in**. El servidor de fichero replicado puede implementarse de la siguiente forma:

```
 1:  with SEQUENTIAL_IO;
 2:  for group Servidor_Ficheros_Robusto;
 3:  replicated agent Miembro_Servidor_Ficheros_Robusto is
 4:     package MI_SEQ_IO is new
 5:         SEQUENTIAL_IO(Servidor_Ficheros_Robusto.Elemento_Fichero);
 6:  begin
 7:    loop
 8:      begin
 9:        select
10:         accept Crear(Fichero: in out Tipo_Fichero;
11:                      Modo   : in     Modo_Fichero:=OUT_FILE;
12:                      Nombre : in     String   :="";
13:                      Forma  : in     String   :="") do
14:           MI_SEQ_IO.CREATE(Fichero,Modo,Nombre,Forma);
15:         exception
16:           when MI_SEQ_IO.Status_Error =>
17:              raise Servidor_Ficheros_Robusto.Error_de_Estado;
18:           when MI_SEQ_IO.Mode_Error =>
19:              raise Servidor_Ficheros_Robusto.Error_de_Modo;
20:           ...
21:         end Create;
22:       or
23:         accept Abrir(...) do
24:             ...
25:         end Abrir;
26:       or
27:             .
28:             .
29:             .
30:             .
31:       end select;
32:     exceptions
33:       when Servidor_Ficheros_Robusto.Error_de_Estado,
34:            Servidor_Ficheros_Robusto.Error_de_Modo,
35:            ...          => null;
36:     end;
37:   end loop;
38: end Miembro_Servidor_Ficheros_Robusto;
```

Este servidor de ficheros replicado utiliza el paquete genérico de Ada-83 SEQUENTIAL_IO (línea 4) para implementar la gestión local de los ficheros. Para ello instancia este paquete genérico mediante el tipo *Elemento_Fichero* declarado en la especificación del grupo y denomina a la instanciación *MI_SEQ_IO* .

La implementación consta de un bucle principal (líneas 7..37) con una sentencia **select** (líneas 9..31) donde acepta permanentemente citas remotas en todos los servicios del sistema de ficheros replicado. Supongamos que el agente replicado recibe una cita remota en el punto de entrada *Crear* (línea 10). Puesto que esta alternativa no especifica ninguna condición, el agente acepta la cita remota. La secuencia de sentencias asociada a *Crear* realiza una llamada al paquete local *MI_SEQ_IO* (al procedimiento *create* —línea 14—). Si la operación se ejecuta con normalidad, la cita remota se completa y el cliente remoto recibe los parámetros de salida (línea 21).

Si el paquete local *MI_SEQ_IO* eleva alguna excepción, el bloque de sentencias de *Crear* captura la excepción y eleva la excepción remota correspondiente (líneas 15..20). Cuando

el agente ejecuta la sentencia **end** asociada al bloque de sentencias de *Crear* (línea 21) la excepción se propaga simultáneamente al cliente remoto.

Cuando el agente sale de la sentencia **select** (línea 31) debe manejar la excepción (líneas 32..36). En caso de no manejarla, el agente finalizaría su ejecución (porque la excepción no habría sido manejada dentro del agente).

3.5 Servidor-cliente tolerante a fallos

Supongamos que se desea implementar un servicio replicado que tiene la siguiente especificación de grupo:

```
replicated group specification Grupo_1 is
   entry Leer    (Dato:out Integer);
   entry Escribir(Dato:in Integer);
end Grupo_1;
```

Para proporcionar el servicio *Leer*, los miembros del grupo necesitan realizar una cita remota con *Grupo_2*, que tiene la siguiente especificación de grupo:

```
replicated group specification Grupo_2 is
   entry Obtener(Respuesta:out Integer);
end Grupo_2;
```

Para evitar que las réplicas que proporcionan el servicio de *Grupo_1* se bloqueen mientras esperan los parámetros de salida de la cita remota con el grupo *Grupo_2*, realizan la cita remota mediante una *cita remota con reencolamiento*. La forma de implementar este servidor replicado sería la siguiente:

```
for group Grupo_1;
with group Grupo_2;
replicated agent Servidor_Cliente is
   Tmp:Integer;
   entry Esperar(Dato:out Integer;Respuesta:in Integer);
begin
  loop
    select
       accept Leer(Dato:out Integer) do
         Grupo_2.Obtener(Tmp) requeue in Esperar(Tmp);
       end Leer;
    or
       accept Esperar(Dato:out Integer;Respuesta:in Integer) do
         Dato:=Respuesta;
       exception
          when ...
               ...
       end Esperar;
    or
       accept Escribir(Dato:in Integer) do
         ...
       end Escribir;
    end select;
  end loop;
end Servidor_Cliente;
```

La sección de declaraciones del agente contiene la declaración de la variable temporal *Tmp* y la declaración del punto de entrada local *Esperar* que se va a utilizar en la cita con reencolamiento.

La secuencia de sentencias del agente consta de un bucle sin fin en el cual el agente replicado ofrece permanentemente los servicios remotos *Leer* y *Escribir*.

Cuando el agente acepta una cita en el punto de entrada remoto *Leer*, realiza una cita con reencolamiento con *Grupo_2* en el punto de entrada *Obtener* utilizando la variable global *Tmp* para almacenar el parámetro de salida *Respuesta*. Desde el entorno de tiempo de ejecución de Drago está seguro de que *Grupo_2* ha recibido la petición de cita remota (todos los miembros de *Grupo_2* han recibido la petición de cita remota en el punto de entrada *Obtener*), el agente replicado almacena la cita remota que estaba atendiendo como *cita remota pendiente de respuesta* y finaliza la sentencia accept más interna (*Leer*), pudiendo atender otras citas remotas.

Cuando finalmente el agente recibe el parámetro de salida de la cita remota en *Obtener*, realiza de forma atómica la actualización de la variable temporal *Tmp* y el reencolamiento de la *cita remota pendiente de respuesta* en el punto de entrada especificado (*Esperar*), pasando el valor actual de *Tmp* como parámetro de entrada adicional. En caso de que la cita remota con reencolamiento haya propagado alguna excepción, ésta se eleva en el punto de entrada de reencolamiento y se maneja de la forma usual.

Ejemplo de ejecución

Supongamos que un cliente inicia una cita remota con *Grupo_1* en el punto de entrada *Leer* (ver *A* en figura 3.11). Las réplicas que pertenecen a *Grupo_1* aceptan la cita remota e inician la cita remota con reencolamiento con *Grupo_2*. Desde el instante en que los miembros de *Grupo_2* tienen almacenada la petición de cita remota, las réplicas que pertenecen a *Grupo_1* pueden finalizar la ejecución de la sentencia *accept* más interna y almacenar la cita remota con reencolamiento como *cita remota pendiente de respuesta* (ver *B* en figura 3.11).

Supongamos que otro cliente inicia una cita remota con *Grupo_1* en el punto de entrada *Escribir* (ver *C* en figura 3.11). Las réplicas que pertenecen a *Grupo_1* aceptan la cita remota y ejecutan la secuencia de sentencias correspondiente hasta que completan la cita remota (ver *D* en figura 3.11). Mientras mantienen como *cita remota pendiente de respuesta* la cita remota con reencolamiento, pueden continuar aceptando nuevas citas remotas.

Cuando *Grupo_2* completa la cita remota con reencolamiento, las réplicas que pertenecen a *Grupo_1* actualizan el valor del parámetro de salida en la variable local *Tmp* y reencolan la cita remota pendiente de respuesta en el punto de entrada especificado en la sentencia de cita remota con reencolamiento (ver *E* en figura 3.11), pasando el parámetro de salida recibido como un parámetro de entrada adicional.

Posteriormente, cuando la cita remota en *Esperar* sea aceptada, las réplicas completarán la cita remota reencolada y enviarán al cliente remoto el valor recibido en el parámetro de salida de la cita remota (ver *F* en figura 3.11). De esta forma, *Grupo_1* está listo para atender nuevas citas remotas (ver *G* en figura 3.11).

3.6 Problema de los filósofos (en versión distribuida tolerante a fallos - I)

Cinco filósofos pasan su vida comiendo espaguetis y pensando. Los filósofos están sentados en una mesa circular. Cada filósofo tiene su plato correspondiente. La mesa tiene un número de tenedores igual al número de filósofos (cada filósofo tiene un tenedor a su izquierda y un tenedor a su derecha). Cuando un filósofo quiere comer debe coger los tenedores a su izquierda y derecha. Cuando termina de comer (después de un tiempo finito), debe dejar los tenedores de nuevo sobre la mesa.

Este problema fue propuesto por E.W.Dijkstra [Dij71] y ha sido muy estudiado en la literatura [BA82] [Han73]. Se ha utilizado para comprobar la bondad de las facilidades de programación concurrente de los lenguajes de programación y como técnica de comprobación de programas concurrentes. A pesar de su aparente sencillez, ilustra muchos problemas encontrados en la programación concurrente, en particular en el acceso a recursos compartidos e interbloqueos. Los tenedores son los recursos compartidos por los filósofos (que representan los procesos concurrentes).

En este apartado presentamos una solución distribuida tolerante a fallos que utiliza un gestor de tenedores. El gestor de tenedores proporciona los tenedores mediante dos operaciones remotas: *Take_forks_Pair*, utilizada por los filósofos para pedir sus tenedores y *Leave_forks_Pair*,

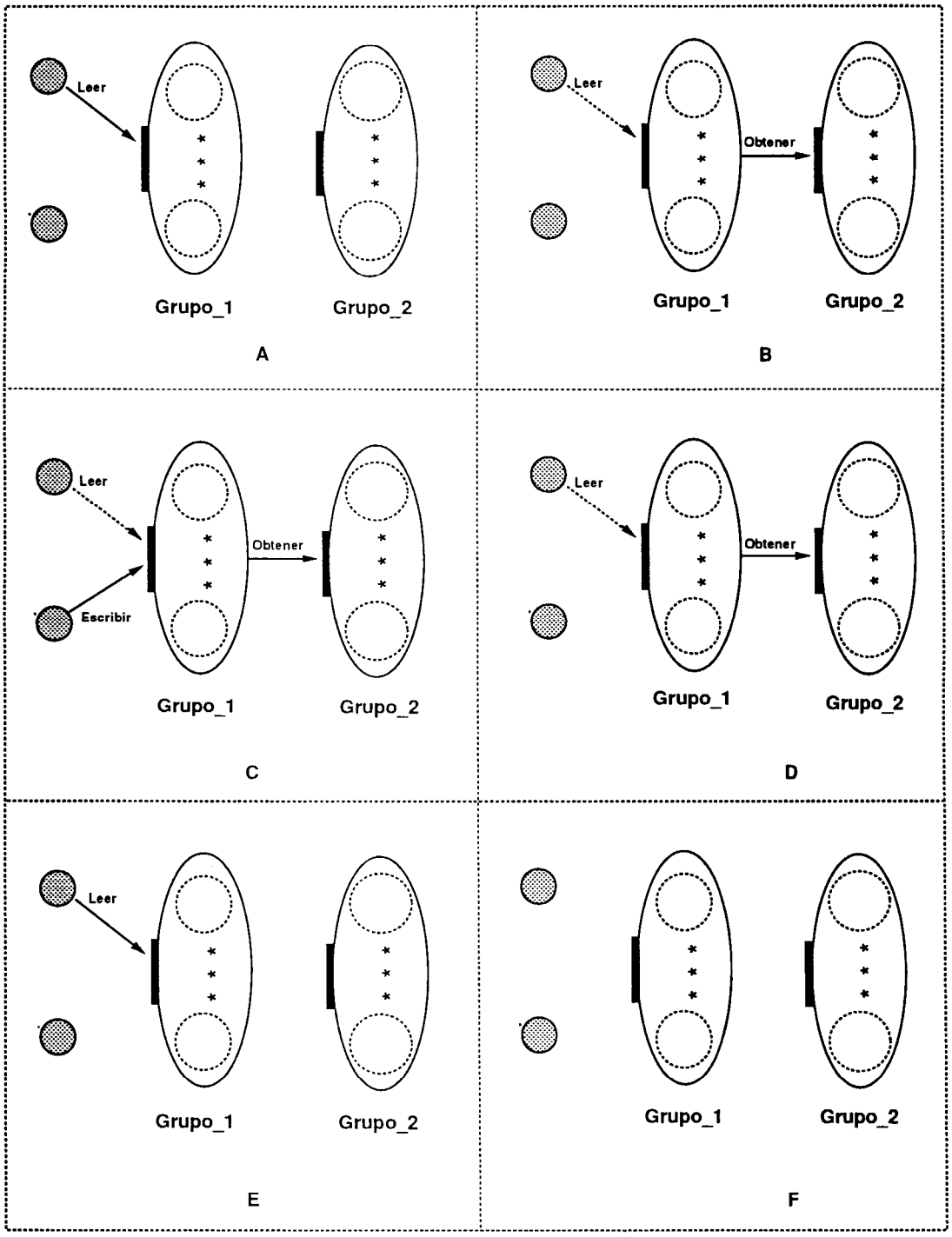


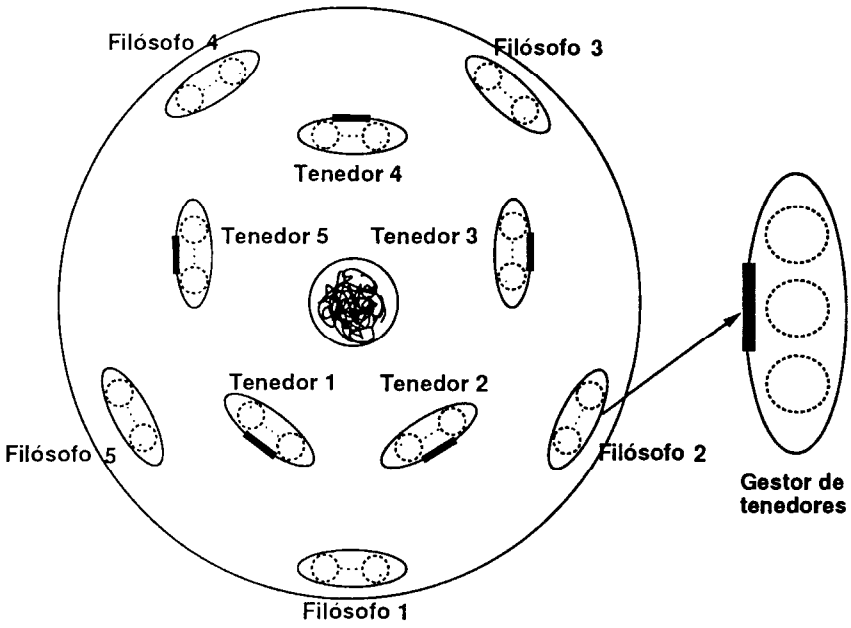
Figura 3.11: Ejemplo de ejecución del servidor-cliente.

utilizada por los filósofos para devolver sus tenedores. Puesto que el gestor de tenedores es crítico (si el nodo donde se ejecuta falla, los filósofos no pueden coger sus tenedores) debemos replicarlo. De igual forma replicamos los tenedores y los filósofos.

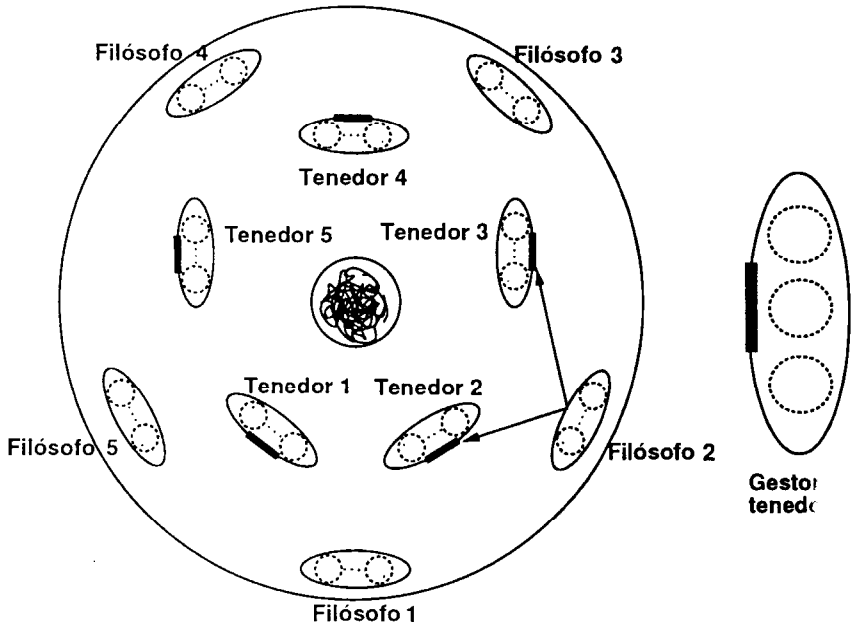
Cuando un filósofo quiere comer, debe realizar una cita remota con el gestor de tenedores para coger sus tenedores: el filósofo 1 pide los tenedores 1 y 2; el filósofo 2 pide los tenedores 2 y 3, etc. (ver *A* en figura 3.12). El filósofo se mantiene bloqueado hasta que ambos tenedores están libres. En este instante, el gestor de tenedores completa la cita remota con el filósofo y éste ya puede utilizar sus tenedores (puede llamar a los grupos replicados que implementan los tenedores que le corresponden —ver *B* en figura 3.12—).

La especificación de grupo asociada al gestor de tenedores es la siguiente:

```
replicated group specification Forks_Manager is
   entry Take_forks_Pair (i:in Positive);
   entry Leave_forks_Pair(i:in Positive);
end Forks_Manager;
```



A) Cuando un filósofo quiere comer, realiza una cita remota con el gestor de tenedores solicitando sus tenedores.



B) Cuando el gestor de tenedores completa la cita remota el filósofo puede tomar sus tenedores y comer.

Figura 3.12: Problema de los filósofos comiendo.

El gestor de tenedores puede implementarse de la siguiente forma:

```
for group Forks_Manager;
replicated agent Forks_Manager_Replica is
  Busy:array(1..5) of Boolean := False;
    -- Status of every fork. True when busy.

  entry Take_Pair_1;
  entry Take_Pair_2;
  entry Take_Pair_3;
  entry Take_Pair_4;
  entry Take_Pair_5;
    -- Declaration of local entries.
begin
  loop
     select
       accept Take_Forks_Pair(i: in Positive) do
          case i is
             when 1 => requeue Take_Pair_1;
             when 2 => requeue Take_Pair_2;
             when 3 => requeue Take_Pair_3;
             when 4 => requeue Take_Pair_4;
             when 5 => requeue Take_Pair_5;
          end case;
       end Take_Forks_Pair;
     or
       when not( Busy(1) or Busy(2) ) =>
       accept Take_Pair_1 do
          Busy(1):= true;
          Busy(2):= true;
       end Take_Pair_1;
     or
       ...
       -- TAKE_PAIR_2..TAKE_PAIR_4 are similarly
       -- written but TAKE_PAIR_5 must use BUSY(5)
       -- and BUSY(1).
     or
       accept Leave_Forks_Pair(i:in Positive) do
         Busy(i):= false;
         Busy( (i mod 5) +1):= false;
       end Leave_Forks_Pair;
     end select;
  end loop;
end Forks_Manager_Replica;
```

Veamos su funcionamiento. Cuando el gestor replicado recibe una petición de cita remota solicitando una pareja de tenedores (*Take_Forks_Pair*), reencola la cita remota en el punto de entrada local asociado a dicha pareja de tenedores. Si ambos tenedores están libres, se marcan como ocupados y se completa la cita remota (permitiendo continuar su ejecución al cliente remoto). Si alguno de los tenedores está ocupado, la cita remota se mantiene encolada en la cola asociada al punto de entrada local.

Cuando el filósofo termina de comer, realiza una cita remota en *Leave_Forks_Pair*. En este instante, se ejecuta de nuevo la sentencia **select** y si hay algún filósofo encolado que tiene ahora

disponibles sus dos tenedores, el agente acepta su cita remota y la completa (permitiéndole así coger sus tenedores y comer).

Esta implementación tiene las siguientes propiedades:

1. **No tiene problemas de interbloqueo**: El problema de interbloqueo ocurre cuando todos los filósofos toman un tenedor (por ejemplo su tenedor de la derecha) y se quedan bloqueados esperando el otro tenedor. En esta implementación los filósofos solamente realizan una cita remota al gestor de tenedores para tomar ambos tenedores[13], evitando así el problema del interbloqueo.
2. **Distribución**: Los filósofos están distribuidos por la red, al igual que los tenedores (y el gestor de tenedores).
3. **Tolerancia a fallos**: Los filósofos, los tenedores y el gestor de tenedores están replicados para tolerar fallos del hardware.
4. **Transparencia**: Los filósofos no conocen el nodo en el que se ejecutan los tenedores y el gestor de tenedores (simplemente realizan las citas remotas cualificando la cita con el identificador del grupo correspondiente)

3.7 Resumen

Se han presentado varias aplicaciones distribuidas tolerantes a fallos. El objetivo de estos ejemplos ha sido introducir la programación de aplicaciones tolerantes a fallos con Drago así como describir con ejemplos las características relacionadas con la tolerancia a fallos de Drago.

Los grupos replicados de Drago proporcionan una abstracción que facilita la programación de aplicaciones distribuidas tolerantes a fallos. Los clientes del servicio replicado no necesitan tratar con el número de réplicas ni con su ubicación física. Drago proporciona la abstracción de utilizar los servicios que proporciona el grupo como si el servicio lo proporcionase una única réplica. Las especificaciones de grupos replicados permiten al programador declarar constantes, tipos (incluyendo los tipos privados de Ada 83), excepciones y puntos de entrada remotos (que pueden tener parámetros de entrada por defecto).

La única condición impuesta por Drago para implementar servicios distribuidos tolerantes a fallos es que la aplicación debe programarse mediante código determinista (Drago no necesita conocer la semántica de la aplicación). Los agentes replicados pueden realizar una aceptación selectiva de puntos de entrada, reencolar citas remotas y realizar citas remotas con reencolamiento.

[13]Esto modifica el problema clásico propuesto por Dijkstra. En el siguiente capítulo se presenta otra solución a este problema que no requiere el gestor de tenedores.

Capítulo 4

Programación de aplicaciones cooperativas con Drago

Este capítulo presenta varios ejemplos de aplicaciones distribuidas que cooperan para alcanzar un objetivo común. Estos ejemplos han sido clasificados en varios apartados. El primer apartado presenta los siguientes ejemplos de aplicaciones paralelas:

- Un ejemplo de gestión de recursos distribuidos: el control de los paneles de información de un aeropuerto.
- Un ejemplo de cooperación paralela mediante especialización.
- Un ejemplo de un grupo trabajadores en el que cada miembro atiende un trabajo.

El segundo apartado presenta varios ejemplos utilizan el mecanismo de comunicación intragrupo de Drago. Son los siguientes:

- Una variable distribuida que es compartida solamente por los miembros de un grupo.
- Una implementación del algoritmo de Ricart y Agravala [RA81] para conseguir exclusión mútua en sistemas distribuidos.

El tercer apartado presenta un ejemplo de una implementación de un servicio de reloj distribuido que utiliza el solapamiento de grupos cooperativos.

El cuarto apartado presenta dos aplicaciones cooperativas que necesitan tratar fallos. Para ello utilizan el mecanismo de notificación de fallos en grupos cooperativos de Drago. Son los siguientes:

- Un ejemplo de una aplicación cooperativa simple tolerante a fallos.
- Otra solución distribuida tolerante a fallos al problema de los filósofos. Esta versión no requiere el gestor de tenedores utilizado en la versión presentada en el capítulo anterior.

Finalmente el último apartado contiene una breve discusión de como implementar aplicaciones tolerantes a fallos software mediante Drago.

4.1 Ejemplos de aplicaciones paralelas

El próximo ejemplo describe como se puede aprovechar el paralelismo inherente de un sistema distribuido mediante Drago.

4.1.1 Control de los paneles de información de un aeropuerto

- *Este ejemplo presenta el uso de la abstracción de grupo cooperativo para facilitar la gestión de recursos distribuidos.*

Un aeropuerto tiene muchos paneles de información que visualizan información acerca de las próximas llegadas y salidas de vuelos. Esta información debe actualizarse periódicamente en todos los paneles, independientemente del tipo de dispositivo.

Este problema se puede resolver con Drago mediante un grupo cooperativo. Cada dispositivo tiene asociado un controlador de dispositivo. El grupo proporciona la abstracción de manejar todos los diferentes controladores de dispositivo como un fuesen uno solo (facilitando así la gestión de los recursos distribuidos). Ver figura 4.1.

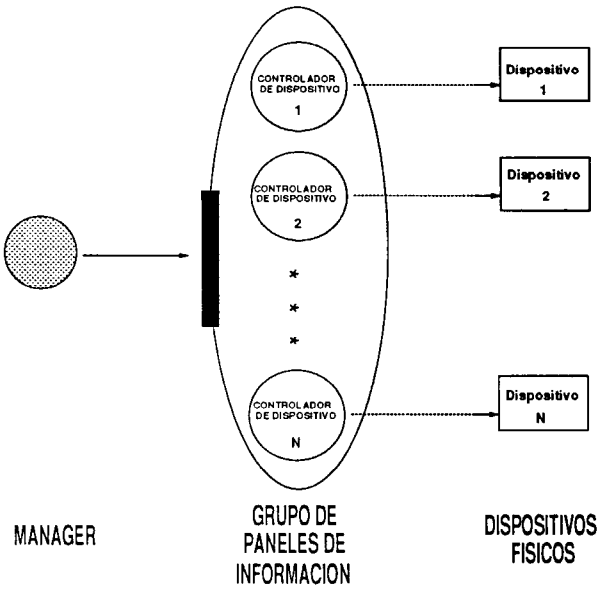


Figura 4.1: Control de recursos distribuidos.

Todos los controladores de dispositivo pertenecen a un grupo con la siguiente especificación:

```
group specification Paneles_de_Informacion is
   entry Escribir    (Item: in String);
   entry Nueva_Linea(Espaciado: in Positive:= 1);
   ...
end Paneles_de_Informacion;
```

La cabecera de una especificación de un grupo cooperativo no contiene ninguna palabra reservada (es la especificación de grupo por defecto). De forma similar a la especificación de un grupo replicado, la especificación de un grupo cooperativo no especifica los miembros del grupo[1].

La forma de implementar un controlador de un terminal de caracteres sería la siguiente:

```
 1:  for group Paneles_de_Informacion;
 2:  with Text_io;
 3:  agent Controlador_1 is
 4:  begin
 5:    loop
 6:      select
 7:         accept Escribir(Item: in String) do
 8:             Text_io.Put(Item);
 9:         end Escribir;
10:      or
11:         ...
12:      end select;
13:    end loop;
14:  end Controlador_1;
```

La clausula **for group** (línea 1) especifica todos los grupos cooperativos a los que pertenece este agente. El agente cooperativo debe atender todas las peticiones de cita remota de todos los puntos de entrada remotos declarados en los grupos a los que pertenece, y tiene visibles todas las declaraciones contenidas en las correspondientes especificaciones de grupo. En este ejemplo el agente *Controlador_1* pertenece al grupo *Paneles_de_Información* y debe manejar las peticiones de cita remota asociadas a los puntos de entrada remotos *Escribir*, *Nueva_Línea*, etc.

Este controlador de dispositivo utiliza paquetes Ada-83 *Text_io* para implementar el control del terminal de caracteres. Por ello contiene una clausula **with** en la que nombra el paquete *Text_io* (línea 2). La cabecera del agente cooperativo no tiene palabras reservadas (es el agente por defecto —línea 3—).

La implementación del servicio (líneas 4..14) consta de un bucle principal con una sentencia **select** donde se ofrecen permanentemente todos los servicios remotos.

Supongamos que el controlador recibe una petición de cita remota en el punto de entrada *Escribir*. La semántica de ejecución del agente cooperativo es idéntica a la semántica de

[1]La especificación del número de miembros del grupo y su mapeado correspondiente depende de la implementación.

Cuando finalmente se completa la cita remota, el cliente remoto es desbloqueado y procesa todas las respuestas accediendo a C (líneas 6..9). Si algún miembro falló antes de completar la cita remota, cuando todos los miembros vivos hayan completado la cita remota, se eleva la excepción *Group_Error* en el punto de la llamada. La excepción se maneja en la forma usual de Ada (líneas 10..12).

Debemos resaltar que el cliente no utiliza *Trabajo_en_Paralelo'range* para procesar las respuestas (línea 6). En vez de ésto utiliza el atributo **range** aplicado a la variable local C (línea 6). La razón de ello es que puede ocurrir que falle un miembro del grupo después de completar la cita remota pero antes de que hayan completado la cita remota todos los miembros del grupo. En este caso, al procesar el atributo **range** asociado al grupo, su valor proporciona el número de miembros actualmente vivos en el grupo. Por esta razón, si el cliente lo utiliza para procesar las respuestas, no procesa todas las respuestas recibidas (puesto que las respuestas recibidas incluyen la respuesta del miembro que falló).

Drago no proporciona ningún mecanismo para permitir que un cliente conozca cual es la respuesta enviada por cada miembro del grupo. Si una aplicación necesita esta información, puede proporcionarse mediante un parámetro de salida adicional.

4.1.3 Grupo de trabajadores paralelos

- *Este ejemplo presenta una aplicación distribuida en la que un grupo de trabajadores distribuidos se reparten el trabajo de forma equitativa (cada trabajador procesa un $1/N$ trabajos, siendo N el número de trabajadores).*

Supongamos que tenemos N trabajadores paralelos. Nuestro objetivo es distribuir la carga de trabajo entre todos los trabajadores de la siguiente forma: el trabajo 1 lo procesa el trabajador 1; el trabajo 2 lo procesa el trabajador 2; el trabajo $N+1$ lo procesa de nuevo el trabajador 1 y así sucesivamente. De esta forma, el trabajador 1 procesa los trabajos 1, $N+1$, $2N+1$, etc; el trabajador N procesa los trabajos N, $2N$, $3N$, etc.

Esta aplicación puede implementarse en Drago mediante un grupo cooperativo. La especificación del servicio remoto sería:

```
group specification Trabajadores is
   entry Hacer(Parametro: in Integer;
               Resultado: out Integer);
end Trabajadores;
```

Debemos resaltar que el parámetro de salida *Resultado* no está declarado como una formación unidimensional irrestringida. De esta forma los clientes remotos de este grupo solamente reciben una respuesta no nula. Drago descarta automáticamente el resto de las respuestas. Los trabajadores pueden estructurarse de la siguiente forma:

```
 1:   for group Trabajadores;
 2:   agent Trabajador_1 is
 3:      Yo:constant Positive:=1;
 4:      Numero_Trabajadores:Positive:=5;
 5:      Contador_Trabajos:Natural:=0;
 6:  begin
 7:    loop
 8:       accept Hacer(Parametro:in Integer;
 9:                    Resultado:out Integer) do
10:          Contador_Trabajos:=Contador_Trabajos+1;
11:          if Yo=(Contador_Trabajos mod Numero_trabajadores)+1 then
12:             -- Proceso el trabajo.
13:             ...
14:          else
15:             return null;
16:          end if;
17:       end Hacer;
18:    end loop;
19:  end Trabajador_1;
```

La constante *Yo* (línea 3) identifica a este trabajador (el trabajador *N* simplemente tendrá el valor *N* en esta constante). La constante *Numero_Trabajadores* recuerda el número de trabajadores de que consta el grupo. La variable local *Contador_Trabajos* (línea 5) se utiliza para contar los trabajos que van recibiendo los miembros del grupo.

El punto de entrada remoto *Hacer* está siempre *abierto*. Cuando el agente acepta una cita remota en *Hacer* (línea 8), incrementa *Contador_Trabajos* (línea 9) y calcula si debe procesar este trabajo. En caso afirmativo, procesa el trabajo (líneas 12,13). Si el trabajo no es para él, simplemente completa la cita remota con una sentencia **return null** (línea 15).

Esta solución tiene el siguiente inconveniente: mientras el agente está trabajando no acepta citas remotas posteriores (reduciendo así el grado de paralelismo del grupo). Este problema puede evitarse mediante una tarea local[4] (el cuerpo de *Hacer* puede crear una tarea local que sea la que realmente haga el trabajo, liberando así el flujo del agente para que pueda aceptar otras citas remotas).

Los clientes de este grupo de trabajadores realizan sus citas remotas con el grupo de la siguiente forma:

```
with group Trabajadores;
agent Cliente is
   Parametro:Integer:=...;
   Resultado:Integer;
begin
   Trabajadores.Hacer(Parametro,Resultado);
end Trabajadores;
```

La semántica de ejecución es la siguiente. El agente realiza una cita remota con el grupo *Trabajadores* en el punto de entrada remoto *Hacer* y cuando todos los miembros vivos han completado la cita remota, recibe en el parámetro de salida *Resultado* el resultado del trabajo solicitado (la respuesta del miembro que no completó la cita con la sentencia **return null**).

[4]Los agentes cooperativos pueden tener tareas locales Ada 83, mientras que los agentes replicados no.

4.2 Ejemplos de comunicación intragrupo

Este apartado presenta varios ejemplos que utilizan el mecanismo de comunicación intragrupo de Drago.

4.2.1 Variable distribuida compartida por los miembros de un grupo

- *Este ejemplo presenta el mecanismo de comunicación intragrupo que utilizan los miembros de los grupos cooperativos para comunicarse sin que lo vean los clientes remotos. También presenta los puntos de entrada locales.*

Supongamos que queremos implementar una variable distribuida que es compartida por todos los miembros de un grupo cooperativo. Supongamos que todos los miembros tienen una copia local del valor de la variable. Supongamos que la lectura de la variable se realiza consultando el valor de la copia local, mientras que la escritura actualiza todas las copias. Esta aplicación puede implementarse en Drago mediante el mecanismo de comunicación *intragrupo*. La especificación de grupo necesaria para esta aplicación sería similar a la siguiente:

```
group specification Variable_Compartida_Intragrupo is
  -- Intergroup operations
  -- ...
intragroup
  type V_Type is ...
  entry Escribir(V:in V_Type);
end Variable_Compartida_Intragrupo;
```

Esta especificación de grupo cooperativo consta de dos partes: la parte de comunicación *intergrupo* (que contiene la declaración de todos los servicios que proporciona el grupo a los clientes remotos) y la parte de comunicación *intragrupo* (que contiene todos los servicios que solamente son visibles a los miembros del grupo). Las citas remotas en puntos de entrada *intragrupo* se realizan y aceptan igual que las citas remotas en puntos de entrada *intergrupo*[5]. Esto simplifica la implementación de nuestra variable compartida distribuida porque Drago asegura que todos los miembros del grupo cooperativo reciben la misma secuencia de citas remotas independientemente del ámbito de visibilidad del punto de entrada remoto. Los miembros de este grupo pueden implementarse de la siguiente forma:

[5]La sección intragrupo solamente proporciona un control de visibilidad de servicios remotos.

```
1:  for group Variable_Compartida_Intragrupo;
2:  agent Miembro is
3:    Copia_Local:V_Type;
4:
5:    entry Leer(V:out V_Type);
6:
7:    task ...
8:      ...
9:      Variable_Compartida_Intragrupo.Write(...);
10:     ...
11:     Leer(....);
12:     ...
13:   end ...
14:
15: begin
16:   loop
17:      select
18:         accept Escribir(V:in V_TYPE) do
19:            Copia_Local:=V;
20:         end Escribir;
21:      or
22:         accept Leer(V:out V_Type) do
23:            V:=Copia_Local;
24:         end Leer;
25:      or
26:         -- Servicios intergrupo
27:         ...
28:      end select;
29:    end loop;
30: end Miembro;
```

El código que implementa el trabajo del agente se ha colocado dentro de la tarea local. De esta forma el agente puede aceptar citas remotas mientras hace su trabajo. Veamos el funcionamiento de este agente.

Cuando el agente cooperativo comienza su ejecución, crea la tarea local y ejecuta la secuencia de sentencias del agente (que acepta citas remotas en los puntos de entrada *intergrupo*, puntos de entrada *intragrupo* y en el punto de entrada local *Leer*). Los puntos de entrada locales deben declararse dentro de la sección de declaraciones del agente (línea 5).

Cuando un miembro de un grupo cooperativo quiere actualizar el valor de la variable distribuida compartida, realiza una cita remota *intragrupo* en *Escribir* (línea 9). Todos los miembros del grupo (incluyendo el llamador) reciben la petición de cita remota intragrupo. Cuando todos han aceptado y completado la cita remota (lo que se asegura que todas las copias de la variable se han actualizado) el llamador es desbloqueado y puede continuar su ejecución.

Cuando un miembro del grupo cooperativo quiere leer el valor de la variable compartida, inicia una cita con el punto de entrada local *Leer* (línea 11). El agente encola la petición de cita local en la cola asociada al punto de entrada local *Leer* y trata la petición como si fuese una petición de cita remota. Cuando el agente acepte la cita local, devolverá el valor actual de *Copia_Local* mediante el parámetro de salida (modo **out**) de *Leer* y el llamador continuará su ejecución.

Debe resaltarse que la lectura y escritura de la copia local de la variable compartida solamente la hace la secuencia de sentencias del agente. Esto hace que esta implementación tenga las siguientes características:

- **Consistencia**: Todas las copias de la variable *intragrupo* compartida se actualizan con la misma secuencia de valores en el mismo orden.

- **Exclusión mutua**: El agente proporciona exclusión mutua de ejecución de la secuencia de sentencias asociada a los puntos de entrada locales y remotos. Por ello, si el agente tuviese internamente varias tareas locales que arbitrariamente leyesen el valor de la copia local, el agente asegura que las tareas locales no realizan ninguna lectura mientras se está realizando una escritura.

Ejemplo de ejecución

Veamos un ejemplo de ejecución de esta variable intragrupo compartida. Supongamos un grupo cooperativo compuesto por tres agentes cooperativos que implementan la variable intragrupo compartida descrita anteriormente. Supongamos que un cliente de este grupo inicia una cita remota con el grupo (ver A en figura 4.2). Drago almacena esta petición de cita remota hasta que los miembros del grupo la acepten.

Supongamos que un miembro del grupo quiere escribir en la variable intragrupo compartida. Para ello realiza una cita remota intragrupo en el punto de entrada *Escribir* (ver B en figura 4.2). Al igual que ocurrió con la petición de cita remota del cliente, Drago almacena esta petición de cita hasta que el los miembros del grupo la acepten.

Puesto que el agente no tiene peticiones de cita remota encoladas, cuando los miembros del grupo aceptan una cita remota, la petición de cita remota más antigua (la petición de cita remota intergrupo) se encola en el punto de entrada correspondiente y el agente la acepta (ver C en figura 4.2). Cuando todos los miembros han completado la cita remota, el llamador recibe el valor de los parámetros de salida y finalmente continúa su ejecución (ver D en figura 4.2).

Cuando los miembros del grupo cooperativo aceptan otra cita remota, de nuevo la petición de cita remota más antigua (la petición de cita remota intragrupo) se encola en la cola correspondiente y el agente la acepta (ver E en figura 4.2). Cuando todos los miembros del grupo han completado la cita intragrupo (todos han actualizado el valor de su copia local), el llamador continúa su ejecución (ver F en figura 4.2).

Cuando un miembro de un grupo cooperativo quiere leer el valor de la variable compartida, realiza una cita con el punto de entrada local *Leer* (ver G en figura 4.2). La cita se encola en la cola asociada al punto de entrada local. Cuando el agente acepta la cita (ver H en figura 4.2), devuelve en el parámetro de salida el valor actual de la variable compartida y el llamador continúa su ejecución (se ha completado la cita en el punto de entrada local). En este estado el grupo está listo para aceptar nuevas peticiones (ver I en figura 4.2).

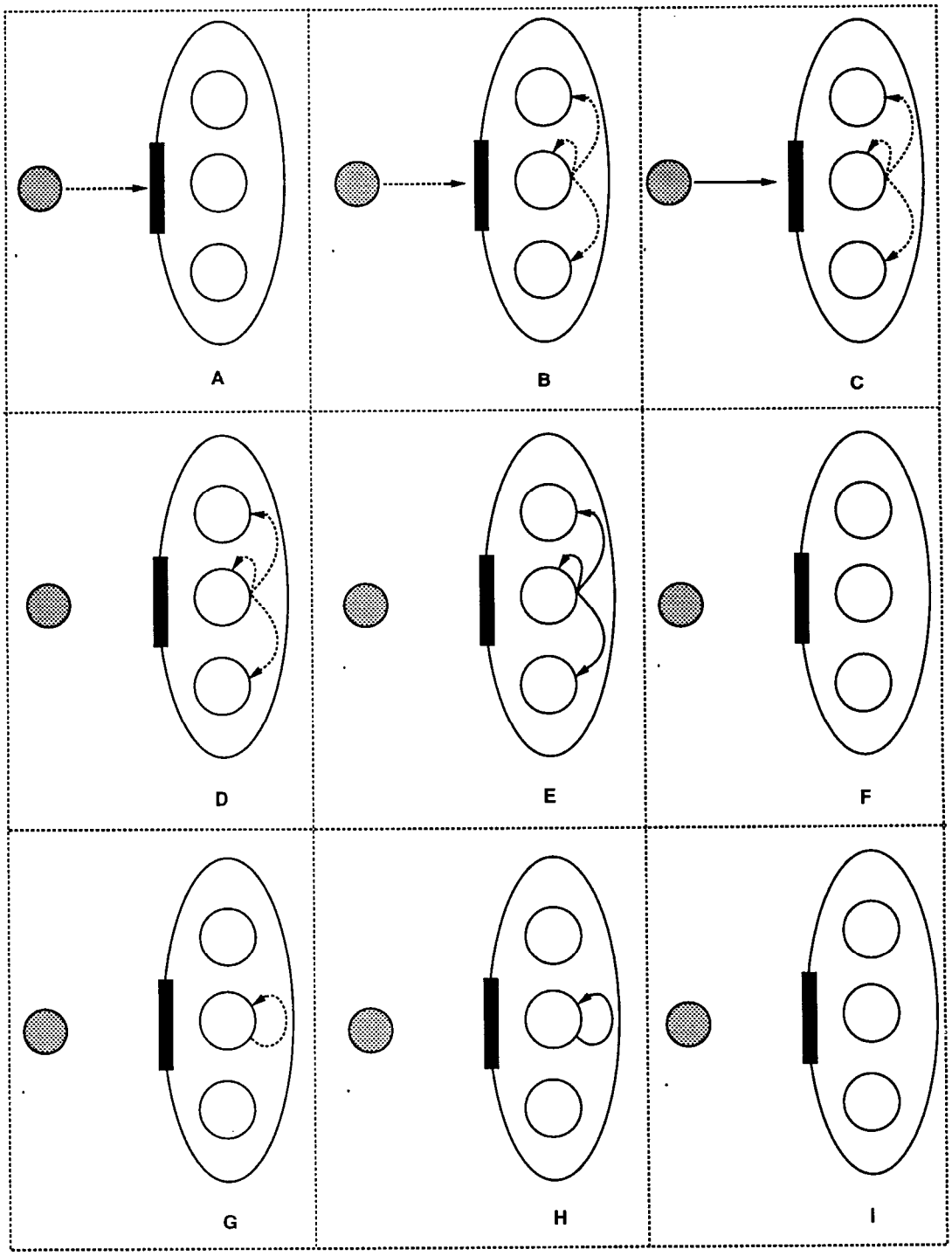


Figura 4.2: Ejemplo de ejecución de la variable distribuida intragrupo.

4.2.2 Exclusión mutua distribuida

- *Este ejemplo presenta otra aplicación de la comunicación intragrupo así como un ejemplo de uso del identificador único que asocia Drago a los miembros de un grupo.*

Los sistemas compuestos de múltiples procesos se suelen programar más fácilmente utilizando regiones críticas. Cuando un proceso necesita leer o actualizar alguna estructura de datos compartida, lo primero que hace es entrar en una región crítica para adquirir el permiso de ejecución en exclusión mutua (y asegurarse así de que ningún otro proceso accederá a dichas estructuras de datos mientras él accede). Una forma de implementar esta exclusión mutua en los sistemas distribuidos es mediante los semáforos (descritos en el capítulo anterior). En este apartado veremos una implementación de otro algoritmo que también consigue este objetivo.

En 1981 Ricart y Agravala propusieron un algoritmo para la exclusión mutua distribuida [RA81] que requiere una ordenación total de eventos en el sistema. El algoritmo trabaja de la siguiente forma. Cuando un proceso quiere entrar en una región crítica, construye un mensaje que contiene el nombre de la región crítica en la que quiere entrar, su número de proceso y una marca con información del tiempo actual (un *timestamp*). A continuación envía el mensaje al resto de procesos, conceptualmente incluyéndose él mismo. Se supone que el envío de mensajes es fiable (todos los mensajes son reconocidos). En caso de que se disponga de comunicación fiable con grupos, puede utilizarse en vez de enviar mensajes individuales. Cuando un proceso recibe un mensaje de petición de otro proceso, la acción que realiza depende de su estado respecto a la región crítica nombrada en el mensaje. Podemos diferenciar tres casos posibles:

1. Si el receptor no está en la región crítica y no quiere entrar en ella, devuelve un mensaje *OK* al emisor.

2. Si el receptor ya está en la región crítica, no responde y encola la petición.

3. Si el receptor quiere entrar en la región crítica, pero aún no lo ha hecho, compara la marca de tiempo del mensaje recibido con la marca de tiempo del mensaje que ha enviado a los demás. Si el mensaje recibido tiene una marca de tiempo más antigua, envía un mensaje de *OK* al emisor. Si su propio mensaje tiene una marca de tiempo más antigua, el receptor encola la petición recibida y no responde al emisor.

Después de enviar las peticiones pidiendo permiso para entrar en una región crítica, un proceso espera hasta que todos le han dado el permiso. Cuando recibe todos los permisos, puede entrar en la región crítica. Cuando sale de la región crítica, envía mensajes *OK* a todos los procesos que tiene encolados y los elimina de la cola[6]. La implementación con Drago del algoritmo descrito anteriormente no requiere el manejo de marcas de tiempo. Esto hace que la solución sea más simple y clara. Veámoslo en detalle.

Todos los procesos distribuidos que se ejecutan en exclusión mutua se implementan mediante agentes cooperativos que pertenecen a un grupo con la siguiente especificación de grupo:

[6]Este breve resumen ha sido extraído de [Tan92].

```
group specification Exclusion_Mutua is
intragroup
   entry Entrar(Llamador: in Member_Identifier);
end Exclusion_Mutua;
```

Los procesos distribuidos se implementan de la siguiente forma:

```
 1:  for group Exclusion_Mutua;
 2:  agent Proceso_Distribuido is
 3:     Mi_Id:Member_Identifier:=Exclusion_Mutua'Member_Identifier;
 4:     Estoy_dentro:Boolean:=false;
 5:
 6:     entry Abandonar_Region_Critica;
 7:
 8:     task ....
 9:     begin
10:         ...
11:         Exclusion_Mutua.Entrar(Yo);
12:         -- Region critica
13:         ...
14:         Abandonar_Region_Critica;
15:         ...
16:     end;
17:
18:  begin
19:     loop
20:        select
21:           when not(Estoy_Dentro) =>
22:           accept Entrar(Llamador:in Member_Identifier) do
23:              if Llamador=Mi_Id then
24:                 Estoy_dentro:=True;
25:              end if;
26:           end Entrar;
27:        or
28:           accept Abandonar_Region_Critica do
29:              Estoy_dentro:=False;
30:           end Abandonar_Region_Critica;
31:        end select;
32:     end loop;
33:  end Proceso_Distribuido;
```

Cada miembro guarda en la variable local *Mi_Id* (línea 3) su identificador único para el grupo *Exclusion_Mutua* (a través del atributo *Member_Identifier* aplicado a un nombre de grupo) e inicializa la variable local *Estoy_dentro* a *False* (línea 4), representando que el agente aún no está en la región crítica.

Cuando el agente quiere entrar en la región crítica, realiza una cita intragrupo en *Entrar* (línea 11) y espera hasta que todos los miembros del grupo lo dejen entrar. Todos los miembros del grupo *Exclusion_Mutua* reciben la petición de cita remota intragrupo. Si algún miembro está en la región crítica simplemente deja la petición de cita remota encolada hasta que salga de la región crítica (este miembro tendrá la condición asociada a *Entrar* cerrada —línea 21—). Si ningún miembro está aún en la región crítica, todos completan la cita remota intragrupo (línea 26) y la tarea interna comienza a ejecutarse en exclusión mutua (el llamador recibió su

propia cita intragrupo e inicializó su variable *Estoy_dentro* a *True* (líneas 23..25), cerrando así la condición de *Entrar* y encolando todas las citas remotas posteriores)[7].

Cuando el agente que tiene el permiso de ejecución en exclusión mutua sale de la región crítica realiza una cita en el punto de entrada local *Abandonar_Region_Critica* (línea 14). Puesto que es el único punto de entrada del agente cuya condición está *abierta* (la condición de *Entrar* está *cerrada*) el agente acepta esta cita. La secuencia de sentencias de *Abandonar_Region_Critica* simplemente inicializa la variable local *Estoy_dentro* a *False* (línea 29), abriendo así la condición asociada a *Entrar*. Ahora el agente reevalúa las condiciones asociadas a las alternativas del **select** y acepta todas las peticiones de cita remota encoladas en *Entrar*, desbloqueando a los miembros que han intentado entrar en la región crítica mientras él estaba dentro.

Esta implementación no considera los fallos de los nodos. Cuando se produce un fallo y todos los miembros vivos han completado la cita remota, se eleva la excepción *Group_Error* en el punto de la cita remota (ver A.4.10). Para tratar esta excepción debe modificarse la tarea interna del ejemplo anterior de la siguiente forma:

```
task ....
begin
    ...
    begin
       Exclusion_Mutua.Entrar(Yo);
    exception
       when GROUP_ERROR =>
            null;
    end;
    -- Region critica
    ...
    Abandonar_Region_Critica;
    ...
end;
```

Cuando la tarea quiere entrar en la región crítica, realiza una cita remota intragrupo en *Entrar*. Si se completa la cita remota sin que se eleve ninguna excepción remota, de forma similar a la implementación descrita anteriormente, el llamador entra en la región crítica. Si al completarse la cita remota se eleva la excepción *Group_Error*, el llamador sabe que algunos miembros del grupo fallaron después de recibir la petición de cita remota en *Entrar*, pero antes de completar la cita remota (el bloque de sentencias del **accept**). Debido a que la excepción se eleva solamente cuando todos los miembros vivos han completado la cita remota, el miembro que falló no evita que el llamador entre en la región crítica. Por tanto, puesto que el resto de los miembros (todos los miembros actualmente vivos) le dejaron entrar en la región crítica, el llamador simplemente descarta la excepción y entra en la región crítica.

[7] La variable *Estoy_dentro* no puede ponerse a *True* dentro de la tarea interna (después de completar el intragrupo GRC) porque el agente podría aceptar peticiones de cita remota posteriores en *Entrar* después de haber procesado su propia petición de cita remota pero antes de que el resto de los miembros le hayan concedido permiso para entrar (y por tanto entren también erróneamente en la región crítica).

4.3 Ejemplos con grupos solapados

Este apartado presenta una implementación de un servicio de reloj distribuido mediante dos grupos cooperativos solapados (sus miembros pertenecen a dos grupos).

4.3.1 Reloj distribuido

Supongamos un sistema distribuido compuesto de varios nodos con su reloj local correspondiente. Supongamos que el administrador del sistema debe actualizar todos los relojes periódicamente (por ejemplo, cada mañana) para que puedan ejecutarse correctamente algunas aplicaciones distribuidas. Aunque Drago no fue desarrollado para tratar con el concepto de tiempo en sistemas distribuidos, esta aplicación puede implementarse de la siguiente forma: todos los nodos del sistema distribuido tienen un servidor horario local (que proporciona un servicio de reloj local) que periódicamente recibe la información horaria del servidor que implementa el reloj central (que podría ser uno de ellos). Cuando las aplicaciones residentes en un nodo del sistema necesitan acceder a información horaria, solicitan esta información a su servidor horario local.

Esta aplicación puede estructurarse mediante dos grupos (figura 4.3):

1. El servidor de reloj global utiliza un grupo para transmitir periódicamente el tiempo a todos los nodos del sistema. Todos los servidores de reloj locales del sistema deben ser miembros de este grupo.

2. Cada servidor de reloj local utiliza un grupo para proporcionar el servicio a las aplicaciones locales (aplicaciones residentes en dicho nodo).

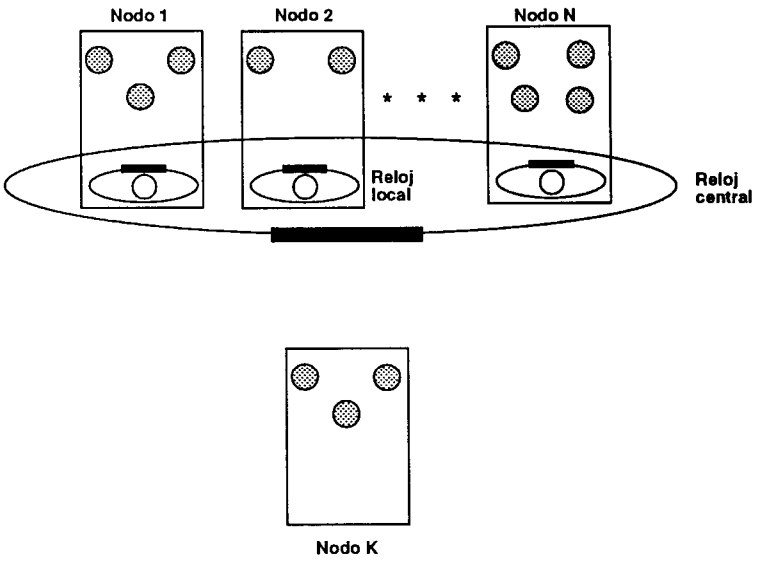


Figura 4.3: Estructura del servicio de reloj distribuido.

Cuando una aplicación necesita conocer la hora actual, realiza una cita remota con el grupo asociado al servidor de reloj local (ver *A* en figura 4.4). Periódicamente, el servidor de reloj global actualiza la información horaria de los servidores locales, realizando una cita remota con todos los servidores de reloj locales (ver *B* en figura 4.4).

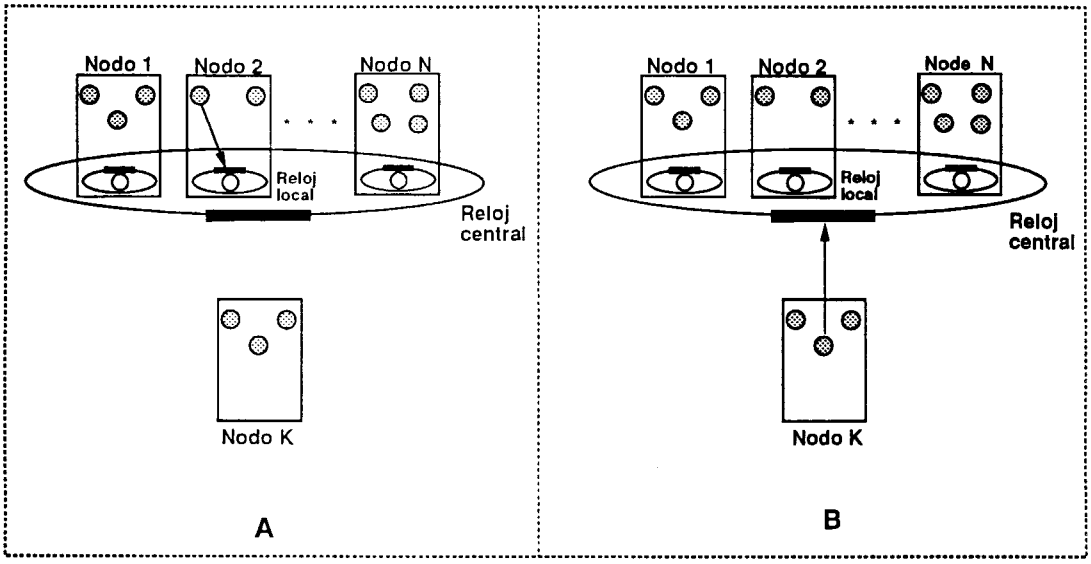


Figura 4.4: Ejemplo de ejecución del reloj distribuido.

La especificación de grupo asociada al reloj central es:

```
with Calendar;
group specification Reloj_Central is
   entry Tiempo_Global(T:in Calendar.Time);
end Reloj_Central;
```

La especificación de grupo asociada a un reloj local es:

```
with Calendar;
group specification Reloj_Local is
   entry Tiempo_Local(T:out Calendar.Time);
end Local_Clock;
```

La estructura interna de los servidores horarios locales sería la siguiente:

```
for group Reloj_Central, Reloj_Local;
with Calendar;
agent Servidor_Reloj_Local is
begin
  accept Tiempo_Global(T:in Calendar.Time) do
     -- Ajustar el valor del reloj local
     ...
  end Tiempo_Global;
  loop
    select
      accept Tiempo_Global(T:in Calendar.Time) do
         -- Ajustar el valor del reloj local
         ...
      end Tiempo_Global;
    or
      accept Tiempo_Local(T:out Calendar.Time) do
         T:=Calendar.clock;
      end Tiempo_Local;
    end select;
  end loop;
end Servidor_Reloj_Local;
```

El agente que proporciona el servicio de reloj global sería el siguiente:

```
with group Reloj_Central;
with Calendar;
agent Servidor_Reloj_Central is
begin
  loop
    Reloj_Central.Tiempo_Global(Calendar.Clock);
    delay ... ;
  end loop;
end Servidor_Reloj_Central;
```

4.4 Ejemplos de aplicaciones cooperativas tolerantes a fallos

Este apartado presenta varios ejemplos de aplicaciones cooperativas tolerantes a fallos. La forma de implementar aplicaciones cooperativas tolerantes a fallos con Drago se realiza mediante la técnica de respaldos. El primer ejemplo muestra la estructura básica de una aplicacion cooperativa que tolera fallos mediante miembros respaldos. El segundo ejemplo es otra solución al problema de los filósofos tolerante a fallos descrito en el capítulo anterior.

4.4.1 Una aplicación tolerante a fallos sencilla

- *Este ejemplo presenta la notificación de fallo que proporciona Drago en los miembros de los grupos cooperativos.*

Supongamos que tenemos un grupo de cinco agentes que implementan una aplicación distribuida cooperativa y que, para tolerar fallos, tenemos también varios agentes respaldo que pertenecen al mismo grupo. Los agentes respaldo son miembros del grupo que, mientras no están activos, simplemente completan las peticiones de cita remota con la sentencia **return null**. De este modo, los clientes remotos del grupo sólo reciben las respuestas de los cinco miembros activos. Cuando un miembro activo falla, uno de los miembros respaldo lo sustituye.

Este tipo de aplicaciones tolerantes a fallos requiere que los miembros del grupo reciban una notificación del fallo. Para ello Drago proporciona un punto de entrada especial que puede especificarse en la sección intragrupo de la especificación de un grupo cooperativo: un *punto de entrada de fallo* (para una descripción completa ver la sección A.3.4).

Por ejemplo, supongamos que el grupo cooperativo tolerante a fallos *Ejemplo* proporciona la operacion intergrupo *Hacer*, la operacion de intragrupo *Registrar_como_activo* y el punto de entrada de fallo *Alguien_ha_fallado*. La especificación correspondiente al grupo es la siguiente:

```
group specification Ejemplo is
   type Tipo_Resultado is array(1..5) of Integer;
   entry Hacer(A:in integer;B:out Tipo_Resultado);
intragroup
   entry Registrar_como_activo(Quien: in Member_Identifier);
   failure entry Alguien_ha_fallado(Quien: in Member_Identifier);
end Ejemplo;
```

De este modo, cuando un miembro del grupo falla, Drago realiza una cita remota intragrupo en el punto de entrada de fallo *Alguien_ha_fallado* pasando como parámetro de entrada el identificador del miembro que falló. Los miembros de este grupo pueden implementarse de la siguiente forma:

```
 1:  for group Ejemplo;
 2:  agent Miembro is
 3:    Mi_Id:Member_Identifier:=Grupo_trabajadores'Member_identifier;
 4:
 5:    procedure Registrar(Quien: in Member_Identifier) is
 6:        ...
 7:    procedure Quitar_del_registro(Quien: in Member_Identifier) is
 8:        ..
 9:    function Esta_activo(Quien: in Member_Identifier) return Boolean is
10:        ...
11:    function Numero_de_miembros_activos return Natural is
12:        ...
13:    task Me_registro;
14:    task body Me_registro is
15:    begin
16:       Grupo_trabajadores.Registrar_como_activo(Mi_Id);
17:    exception
18:        when Group_Error => Null;
19:    end Me_registro;
20:
21:  begin
22:    loop
23:       select
24:          when Numero_de_miembros_activos<5 =>
25:          accept Registrar_como_activo(Quien:in Member_Identifier) do
26:              Registrar(Quien);
27:          end Registrar_como_activo;
28:       or
29:          when Numero_de_miembros_activos=5 =>
30:          accept Hacer(A:in Integer;B: out Integer) do
31:             if Esta_Activo(Mi_Id) then
32:                ...
33:                B:=...
34:              else
35:                return null;
36:             end if;
37:          end Hacer;
38:       or
39:          accept Alguien_ha_fallado(Quien:in Member_Identifier) do
40:             if Esta_Activo(Quien) then
41:                Quitar_del_registro(Quien);
42:             end if;
43:          end Alguien_ha_fallado;
44:       end select;
45:    end loop;
46:  end Miembro;
```

La variable local *Mi_Id* se utiliza para recordar el identificador del agente como miembro de *Grupo_trabajadores* (línea 3).

Inicialmente todos los agentes deben registrarse en el grupo para poder ser trabajadores activos (realizando una cita intragrupo en el punto de entrada *Registrar_como_activo* —línea 16—). Esta cita no puede realizarla el bloque de sentencias del agente (de lo contrario ocurriría un interbloqueo). Por esta razón todos los miembros del grupo poseen internamente la tarea *Me_registro* (líneas 13..19) cuya única función es intentar registrarlos como miembros activos.

La cita remota de Drago asegura que todos los miembros vivos reciben las mismas peticiones de cita remota en *Registrar_como_activo* en el mismo orden. Por esta razón todos los miembros del grupo registran los mismos cinco miembros como activos (líneas 25..27). El resto de los miembros se mantienen encolados (son los suplentes).

Cuando falla un miembro del grupo, Drago realiza automáticamente una petición de cita remota intragrupo en el punto de entrada de fallo *Alguien_ha_fallado* pasando como parámetro de entrada el identificador de el miembro que falló.

Cuando el agente acepta la notificación de fallo (línea 39), comprueba si el miembro que falló estaba activo o si era un suplente. Si el miembro que falló no estaba activo simplemente completa la cita remota. Si el miembro que falló estaba activo, todos los miembros del grupo lo eliminan del registro de miembros activos (línea 41), (abriendo así la condición asociada a *Registrar_como_activo*) y completan la cita. Los miembros supervivientes reevaluan las condiciones asociadas a las alternativas del **select** y aceptan la siguiente petición de cita remota encolada en *Registrar_como_activo* (desbloqueando así a uno de los suplentes).

Como el fallo ocurre siempre mientras la petición de cita remota en *Registrar_como_activo* está encolada en todos los miembros del grupo (incluyendo el miembro que falló), cuando el nuevo suplente completa la cita remota Drago eleva la excepción *Group_Error* en el punto de la llamada (línea 16). Sin embargo, como la excepción solamente se eleva cuando todos los miembros activos han completado la operación remota (y le han aceptado como nuevo miembro activo), el suplente sabe que no ocurre nada anómalo y por ello simplemente descarta la excepción (línea 18).

4.4.2 Problema de los filósofos (versión distribuida tolerante a fallos - II)

Veamos una implementación del problema de los filósofos, en versión distribuida tolerante a fallos, que utiliza el modelo de suplentes presentado en el ejemplo anterior. Esta implementación no necesita el administrador de tenedores utilizado en el capítulo anterior. La especificación del grupo es la siguiente:

```
group Tenedores is
   entry Cojer_pareja_tenedores (i:in positive);
   entry Dejar_pareja_tenedores (i:in positive);
intragroup
   Estado_Tenedor is (Libre,Ocupado);
   entry Registrar_tenedor_activo (Quien: in members_identifier;
                                   Numero_Tenedor: out Positive);
   entry Actualizar_estado        (Numero_Tenedor: in Positive;
                                   Estado: in Estado_tenedor);
   failure entry Fallo_de_tenedor (Quien: in member_identifier);
end Tenedores;
```

Análogamente al ejemplo anterior, cuando los agentes que implementan los tenedores comienzan su ejecución, realizan una cita intragrupo en *Registrar_tenedor_activo*. El grupo solamente permite que se registren cinco tenedores como activos; el resto se mantienen encolados (son los tenedores suplentes).

Al igual que la versión presentada en el apartado 3.6, el filósofo 1 tiene asociado los tenedores 1 y 2; el filósofo 2 tiene asociado los tenedores 2 y 3; etc. Finalmente el filósofo 5 tiene asociados los tenedores 5 y 1. Cuando un filósofo quiere comer debe pedir sus tenedores realizando una cita remota con el grupo *Tenedores* en el punto de entrada *Cojer_pareja_tenedores*. El filósofo se mantiene bloqueado hasta que tiene disponible su pareja de tenedores. En cuanto se desbloquea puede empezar a comer. Cuando el filósofo deja de comer debe devolver los tenedores realizando una cita remota en el punto de entrada *Dejar_pareja_tenedores*.

Al igual que en el ejemplo anterior, cuando falla un miembro del grupo, Drago realiza automáticamente una cita remota intragrupo en el punto de entrada de fallo especificado (*Fallo_de_tenedor*). Si el tenedor que falló no estaba activo, el grupo no hace nada. Si el miembro que falló estaba activo, uno de los tenedores suplentes lo sustituye. Para reemplazarlo correctamente, todos los tenedores suplentes necesitan conocer el estado de los tenedores activos. Para ello hay varias soluciones posibles:

1. Tener un grupo replicado que proporcione una memoria tolerante a fallos (descrito en 3.3).
2. Implementar esta memoria tolerante a fallos mediante los miembros de este grupo cooperativo (de forma similar al ejemplo de variable compartida intergrupo presentado en el apartado 4.2.1).

La implementación que se muestra a continuación utiliza esta segunda solución, y por ello la especificación del grupo contiene la declaración del punto de entrada *Actualizar_estado*.

```
for group Tenedores;
agent Tenedor is
   Yo:Positive;
   Mi_Id:Member_Identifier:=Fork'Members_Identifier;
   Estado_Tenedores:array(Positive range 1..5)
                                    of Tenedores.Estado_Tenedor;

   entry Cola_tenedor;

   procedure Registrar(Quien: in Member_Identifier;
                       Numero_Tenedor:in Positive) is
      ...
   procedure Derregistrar(Quien: in Member_Identifier) is
      ...
   function Esta_Activo(Quien: in Member_Identifier) return Boolean is
      ...
   function Numero_de_tenedores_activos return Natural is
      ...
   function Tenedor_libre return Positive is ...
      -- Devuelve el numero de un tenedor libre y
      -- lo marca como ocupado
      ...

   task Registrar_tenedor;
   task Registrar_tenedor is
   begin
     Tenedores.Registrar_tenedor_activo(Mi_Id);
   exception
     when Group_Error => null;
   end Registrar_tenedor;

begin
   loop
      select
        when Numero_de_tenedores_activos<5 =>
        accept Registrar_tenedor_activo(Quien:in Members_Identifier) do
           Registrar(Quien,Tenedor_libre);
        end Registrar_tenedor_activo;
      or
        when Numero_de_tenedores_activos=5 =>
        accept Actualizar_estado(Numero_tenedor: in Positive;
                                 Estado        : in Estado_tenedor) do
           Estado_tenedor(Numero_tenedor):=Estado;
        end Actualizar_estado;
      or
        when Numero_de_tenedores_activos=5 =>
        accept Cojer_pareja_tenedores(i:in positive) do
           if Esta_activo(Mi_Id) then
              if Yo=i or Yo=(i mod 5)+1 then
                 requeue Cola_tenedor;
              end if;
           end if;
        end Cojer_pareja_tenedores;
      or
        when Numero_de_tenedores_activos=5 and then
             Esta_activo(Mi_Id) and then
             Estado(Yo)=Libre =>
        accept Cola_tenedor do
```

```
            Tenedores.Actualizar_estado(Yo,Ocupado) requeue in Esperar;
         end Cola_tenedor;
      or
         accept Esperar;
      or
         when Numero_de_tenedores_activos=5 =>
         accept Dejar_pareja_tenedores(i:in positive) do
            if Esta_activo(Mi_id) then
               if Yo=i or Yo =(i mod 5)+1 then
                  Tenedores.Actualizar_estado(Yo,Libre) requeue in Esperar;
               end if;
            end if;
         end Dejar_pareja_tenedores;
      or
         accept Fallo_de_tenedor(Quien_fallo:in Members_Identifiers) do
            if Esta_activo(Quien_fallo) then
               Derregistrar(Quien_fallo);
            end if;
         end Fallo_de_tenedor;
      end select;
   end loop;
end Tenedor;
```

La variable local *Mi_Id* se utiliza para recordar el identificador del agente como miembro del grupo *Tenedores*. La variable local *Yo* contiene el número de tenedor asociado al agente (si es un tenedor activo). La variable local *Estado* se utiliza para implementar la variable compartida por los miembros del grupo que almacena el estado de un miembro que ha fallado.

El cuerpo principal del agente consta de un bucle con una sentencia **select** que acepta las peticiones de cita remota (intergrupo e intragrupo) del agente así como la notificación de fallo en *Fallo_de_tenedor*. Inicialmente el agente solamente acepta peticiones de cita remota en el punto de entrada *Registrar_tenedores* (y el punto de entrada de notificación de fallo) y no proporciona ningún servicio hasta que el grupo tiene 5 tenedores activos.

Al igual que en el ejemplo anterior, todo agente tiene una tarea interna (*Registrar_tenedor*) que lo intenta registrar como un tenedor activo (realizando una cita remota en el punto de entrada intragrupo *Registrar_tenedor_activo*).

Cuando el grupo comienza su ejecución todos los miembros intentan registrarse como activos. Drago serializa las peticiones de cita remota correspondientes en el mismo orden en todos los miembros y los miembros asociados a las cinco primeras peticiones de cita remota se asignan a los cinco tenedores en todos los miembros del grupo[8]. A partir de este instante el grupo comienza a proporcionar servicio a los filósofos.

Cuando un filósofo desea comer, solicita su pareja de tenedores realizando una cita remota con el grupo en el punto de entrada *Cojer_pareja_tenedores* pasando como parámetro de entrada la pareja de tenedores solicitada. Todos los miembros del grupo reciben la petición de cita remota. Los miembros que no son los tenedores solicitados simplemente completan la cita remota. Los miembros que son los tenedores solicitados reencolan la petición de cita remota en el punto de entrada local *Cola_tenedor*.

[8]Se supone que la funcion *Tenedor_libre* es determinísta, y de este modo todos los miembros asignan el mismo tenedor a los mismos agentes.

El punto de entrada local *Cola_tenedor* tiene una condición que está abierta cuando el agente es un tenedor activo que está *libre*. Si el tenedor está *ocupado*, la condición se mantiene cerrada (bloqueando así las peticiones de los filósofos) hasta que el filósofo que tiene el tenedor realiza una cita en el punto de entrada remoto *Dejar_pareja_tenedores* (porque *Dejar_pareja_tenedores* actualiza el estado de una pareja de tenedores a *Libre* y de este modo abre la condición asociada a *Cola_tenedores*).

Si el agente es un tenedor activo que está libre, el agente realiza una cita remota con reencolamiento para actualizar el estado del tenedor a *Ocupado* en todos los miembros del grupo, incluyendose él mismo[9]. Cuando todos los miembros han actualizado la variable compartida *Estado*, la cita remota se reencola en el punto de entrada local *Esperar* a la espera de que el agente la acepte y complete la cita remota (desbloqueando así al filósofo que solicitó la pareja de tenedores).

Cuando falla un tenedor activo ocurre lo siguiente:

- Drago realiza una cita remota intragrupo en el punto de entrada *Fallo_miembro* pasando como parámetro de entrada el identificador del miembro que ha fallado. Todos los miembros del grupo reciben la cita. Si el miembro que falló no era un tenedor activo, no hacen nada. Si el miembro que falló era un tenedor activo, lo eliminan del registro de tenedores activos. Al eliminarlo ahora la condición de *Registrar_tenedor_activo* está abierta, mientras que las condiciones asociadas a los servicios intergrupo están cerradas. Por ello todos los miembros aceptan la siguiente petición de cita remota en *Registrar_tenedor_activo* y el siguiente suplente pasa a estar activo en todos los miembros y sustituye al tenedor que falló. De forma similar al ejemplo del apartado anterior, al completarse la cita remota en el nuevo tenedor activo se eleva la excepción *Group_Error* (que simplemente descarta porque todos los miembros vivos lo han registrado como el nuevo tenedor activo).

- Drago eleva la excepción remota **Group_Error** en todos los clientes remotos que estaban encolados en el miembro que falló (los filósofos que estaban esperando dicho tenedor).

4.5 Implementación de aplicaciones tolerantes a fallos software

La metodología conocida como *programación N-versiones* [Avi85] utiliza múltiples implementaciones de la misma especificación de módulo para enmascarar fallos del software. Cuando solicita un servicio, lo solicita a todas las implementaciones y realiza una votación entre las respuestas.

Aunque Drago no fue diseñado para tolerar fallos software, esta metodología puede implementarse con Drago mediante un grupo cooperativo. Para ello simplemente se colocan las diferentes implementaciones del mismo módulo como miembros de un grupo cooperativo. Drago asegura que todos los miembros del grupo reciben la misma secuencia de peticiones remotas en el mismo orden. El cliente debe realizar las citas con el grupo solicitando el valor

[9]Si se realizase una cita remota normal ocurre un interbloqueo porque el agente que llama no acepta su propia llamada porque está esperando las respuestas de todos los miembros, incluyendo la de él mismo.

de todas las respuestas. Cuando se completa la cita remota, el cliente puede implementar el algoritmo de votación para comprobar si el software está ejecutándose correctamente.

4.6 Resumen

Se han descrito varias aplicaciones cooperativas escritas con Drago. El propósito de estos ejemplos ha sido introducir la programación de aplicaciones cooperativas con Drago así como mostrar la semántica de ejecución de Drago.

Los grupos de Drago proporcionan un alto nivel de abstracción para programar aplicaciones cooperativas tolerantes a fallos. Todos los miembros del grupo reciben la misma secuencia de peticiones de cita remota y pueden implementar el mismo o diferente código, retornar una respuesta o retornar una respuesta nula, así como realizar citas remotas intragrupo.

Los agentes cooperativos mantienen todas las características de agentes replicados (cola de peticiones de cita remota pendientes, colas de peticiones de cita remota encoladas, sentencia de aceptación selectiva de citas remotas, reencolamiento de citas remotas aceptadas y citas remotas con reencolamiento) pero pueden tambien tener tareas internas y realizar citas remotas intragrupo.

La especificación de grupo cooperativo permite al programador especificar si el llamador remoto recibe una respuesta o si recibe un conjunto de respuestas (una formación irrestringida de parámetros de salida —uno por cada miembro vivo del grupo—). El programador puede especificar si los miembros de un grupo cooperativo reciben una notificación de fallo cuando muere un miembro del grupo. Esta notificación de fallo cumple las propiedades de la cita remota de Drago (descritas en detalle en el apartado A.3.1 del apéndice).

Capítulo 5

Traducción de Drago a Ada 83

Un objetivo importante durante el diseño de Drago era que fuese implementable mediante un preprocesador que generase código Ada. De esta forma pueden reutilizarse los compiladores de Ada existentes para conseguir una implementación rápida de Drago en un amplio rango de procesadores.

Este capítulo presenta un posible esquema de traducción de Drago a Ada 83. Esta estructurado en cuatro partes. La primera parte presenta una visión general de la estructura del código Ada 83 que se genera al traducir los programas Drago. La segunda parte describe la estructura interna de los suplentes necesarios para implementar la comunicación con grupos. La tercera parte muestra cómo se realiza la traducción de las sentencias Drago a sentencias Ada 83. La cuarta parte contiene una descripción general del preprocesador de Drago.

5.1 Estructura general del código generado

En esta primera parte se describe la estructura general del código Ada 83 que se genera al traducir un programa Drago a código Ada 83.

5.1.1 Traducción de una especificación de grupo

La especificación de un grupo contiene solamente declaraciones de constantes, tipos de datos, puntos de entrada remotos y excepciones (no contiene código). Por lo tanto no es necesario realizar ninguna traducción. Sin embargo, toda esta información será necesaria cuando se traduzca un agente que sea cliente y/o miembro de este grupo[1]. Por esta razón se almacena en disco.

[1]Los agentes que son clientes del grupo tienen visibles todas las declaraciones contenidas en las secciones intergrupo de la especificación del grupo. Los agentes que son miembros de un grupo tienen visibles todas las declaraciones contenidas en la especificación del grupo (ver el manual de referencia técnica de Drago en el apéndice de este documento).

5.1.2 Traducción de un agente

La traducción de un agente genera un programa Ada-83. Este programa Ada-83 utiliza los servicios del paquete *GROUP_IO*[2] y está estructurado mediante un único procedimiento que consta internamente de:

- Varios paquetes que contienen los suplentes de los grupos de los cuales es cliente y/o miembro el agente.
- El paquete *Entorno_Ejecucion* necesario para proporcionar el entorno de tiempo de ejecución de Drago.
- La tarea *Agente* que ejecuta el código Ada 83 resultado de la traducción del programa Drago a Ada 83.

```
procedure <Nombre_del_Agente> is

    package <Nombre_Suplente_1> is
     ...

    package <Nombre_Suplente_N> is
     ...

    package Entorno_Ejecucion is
      ...

    Task Agente is
     ...

begin
  null;
end;
```

Figura 5.1: Estructura interna del código generado.

El paquete *GROUP_IO* proporciona el soporte de tiempo de ejecución de comunicación con grupos (asegurando las propiedades de la cita remota con grupos de Drago descritas en A.3.1) y mantiene una única cola con todos los mensajes que deben entregarse al agente. Estos mensajes pueden ser peticiones de citas remotas o respuestas de citas remotas realizadas anteriormente por el agente.

El paquete *Entorno_Ejecucion* proporciona el soporte de gestión de colas de peticiones remotas del agente. Mantiene una cola de mensajes de petición de cita remota por cada punto de entrada remoto del agente y es el único que se comunica con el nivel de comunicación con grupos (paquete *GROUP_IO*).

La figura 5.2 muestra una visión general del código Ada-83 que se genera al traducir un agente.

[2]Este paquete está escrito integramente mediante código Ada 83 y forma parte de la tesis doctoral de F. Guerra [GAAM93].

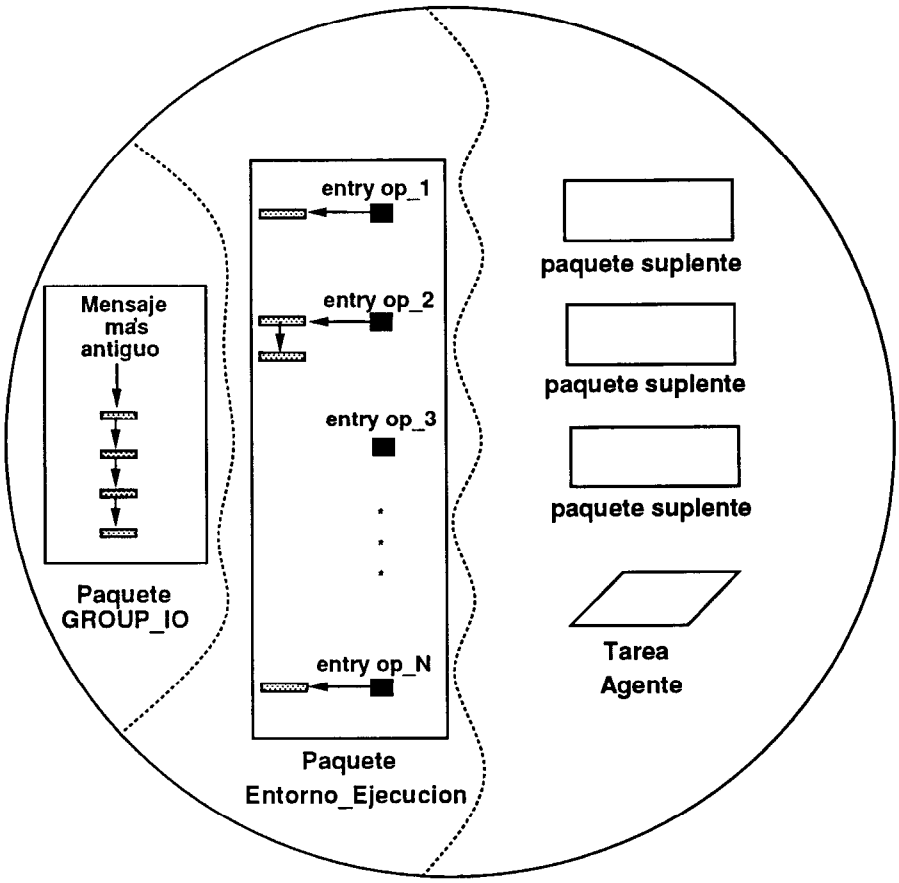


Figura 5.2: Visión general del código generado al traducir un agente.

5.2 Generación de suplentes

La implementación de la comunicación remota con grupos se realiza mediante dos tipos de suplentes por cada punto de entrada remoto declarado en una especificación de un grupo:

- **Suplente del grupo**, que se encarga de aplanar los parámetros de entrada de la cita remota, crear un mensaje de petición con el valor de los parámetros de entrada de la cita remota, enviarlo al grupo y esperar por los mensajes con los parámetros de salida. Estos suplentes son utilizados por los clientes remotos de un grupo (ver figura 5.3).
- **Suplente del cliente remoto**, que se encarga de desaplanar los parámetros de entrada del mensaje con la petición de cita remota, realizar la cita de forma local, crear un mensaje con el valor de los parámetros de salida y enviarlo al cliente remoto. Todos los miembros del grupo tiene un suplente de este tipo (ver figura 5.3).

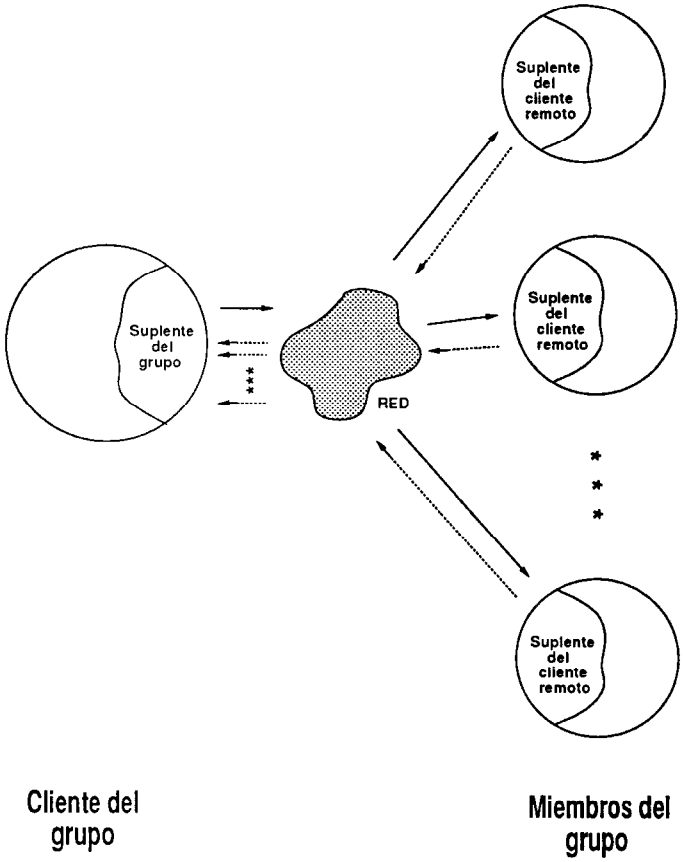


Figura 5.3: Suplentes utilizados en la comunicación remota.

A primera vista, al finalizar el análisis de una especificación de grupo podrían generarse los suplentes (puesto que la especificación del grupo proporciona toda la información de tipos necesaria para realizar la comunicación remota). Sin embargo, no es posible porque los suplentes de los clientes remotos deben llamar al código del usuario y Ada 83 no posee punteros a procedimientos. Por esta razón es necesario retrasar la generación de los suplentes hasta que se realice la traducción de los agentes.

5.2.1 Aplanado y desaplanado de mensajes

Para transmitir un mensaje por la red es necesario convertir el valor de los parámetros de entrada en una secuencia de bytes. Esto se denomina *aplanado* de los parámetros. Los receptores del mensaje[3] deben realizar el proceso inverso (*desaplanado* de los parámetros) para obtener el valor original de los parámetros. La declaración Ada 83 del tipo de dato asociado a esta secuencia de bytes es la siguiente:

```
type Byte is new positive range 0..255;
    for Byte'size use 8;
type Bytes is array(positive range <>) of Byte;
```

Cada vez que se necesita realizar el aplanado de parámetros se genera un procedimiento que realiza el aplanado de los parámetros de entrada de la siguiente forma:

```
1:  procedure Aplanar(M: out Entorno_Ejecucion.Mensaje; Param_1: in ...) is
2:      type Registro is
3:          record
4:              Param_1:...
5:              ...
6:          end record
7:      Parametros: Registro;
8:      Longitud_Registro: constant Natural:= Registro'size / Byte'size;
9:      subtype Bytes_Registro is Bytes(1..Longitud_Registro);
10:     function Convertir is
11:             new unchecked_conversion(Registro,Bytes_Registro);
12: begin
13:     Parametros.Param_1:=Param_1;
14:     ...
15:     M(1..Longitud_Registro):=Convertir(Parametros);
16: end;
```

El procedimiento de aplanado recibe como parámetros de entrada el valor de todos los parámetros que debe aplanar, y retorna un mensaje que contiene dichos parámetros (línea 1).

Las líneas 2..6 contienen la declaración de un registro con todos los parámetros a aplanar en el mensaje. Este registro se utiliza para realizar el aplanado con una única instrucción[4]. La línea 7 contiene la declaración de una variable de este tipo registro.

Las líneas 8..11 contienen las declaraciones necesarias para instanciar la función que realiza el aplanado de los parámetros de entrada: *Longitud_Registro* proporciona la longitud en bytes

[3]Recordemos que el paradigma de comunicación de Drago se basa en comunicación con grupos.

[4]De esta forma el código generado es mucho más legible que si se realiza el aplanado de forma individual (parámetro por parámetro).

del registro; *Bytes_Registro* es la declaración de una formación con una longitud en bytes igual a *Longitud_Registro*; finalmente, la función *Convertir* implementa el aplanado de los parámetros de entrada instanciando la función genérica de conversión sin comprobación de tipos de Ada 83 (*unchecked_conversion*), para que convierta una variable del tipo *Registro*, al tipo *Bytes_Registro*.

El cuerpo del procedimiento copia el contenido de los parámetros de entrada en la variable *Parametros* (líneas 13..14), y llama a la función *Convertir* para realizar el aplanado de los parámetros de entrada sobre el parámetro de salida *M* (línea 15).

De forma similar, cada vez que se necesita realizar el desaplanado de parámetros se genera un procedimiento con la siguiente estructura:

```
1:  procedure Desaplanar(M: in Entorno_Ejecucion.Mensaje; Param_1:out ...) is
2:      type Registro is
3:          record
4:              Out_Param_1:...
5:              ...
6:          end record
7:      Parametros: Registro;
8:      Longitud_Registro: constant Natural:= Registro'size / Byte'size;
9:      subtype Bytes_Registro is Bytes(1..Longitud_Registro);
10:     function Deconvertir is
11:             new unchecked_conversion(Bytes_Registro,Registro);
12: begin
13:     Parametros:=Deconvertir(M(1..Longitud_Registro));
14:     Param_1:=Parametros.Param_1;
15:     ...
16: end;
```

El procedimiento de aplanado recibe como parámetro de entrada el mensaje a desaplanar y retorna como parámetros de salida el valor de todos los parámetros contenidos en el mensaje (línea 1).

Las líneas 2..7 contienen la declaración de un registro con todos los parámetros de salida contenidos en el mensaje.

Las líneas 8..11 contienen declaraciones similares a las utilizadas en el procedimiento de aplanado para instanciar la función de conversión sin comprobación de tipos, pero ahora la conversión es del tipo *Bytes_Registro* al tipo *Registro*.

El cuerpo del procedimiento desaplana el mensaje *M* mediante la función de conversión sin comprobación de tipos (línea 13) e inicializa todos los parámetros de salida con el valor de los parámetros contenidos en el mensaje(líneas 14..15).

5.2.2 Suplente de un grupo

Cuando un agente es cliente de un grupo G se genera un paquete con el identificador G[5]. La especificación de este paquete contiene :

- Todas las declaraciones de constantes, tipos y excepciones contenidas en las secciones intergrupo y privadas de la especificación del grupo[6].
- La declaración de un procedimiento por cada punto de entrada declarado en la sección intergrupo de la especificación del grupo. El identificador de estos procedimientos es el identificador de los puntos de entrada remotos. El perfil de sus parámetros coincide exactamente con el perfil de los parámetros de los puntos de entrada remotos correspondientes[7].

Ejemplo

Supongamos la siguiente especificación de un grupo replicado que implementa un servicio de ficheros distribuido:

```
replicated group specification Servidor_Ficheros is
    entry Leer(In_Param_1:in ...;Out_Param_1:out ...);
    entry Escribir(....);
    Error_de_estado:Exception:
    Error_de_modo:Exception;
    ...
end Servidor_Ficheros;
```

Supongamos el siguiente cliente remoto del grupo:

```
with group Servidor_Ficheros;
replicated agent Cliente is
    – Código del cliente
    ...
end Cliente;
```

[5]De esta forma, el mecanismo de notación punto de los paquetes Ada proporciona automáticamente el mecanismo de notación punto para acceso a las declaraciones contenidas en la especificación del grupo.

[6]Esta información debe estar en el disco como resultado de haber traducido anteriormente la especificación del grupo.

[7]De esta forma, sin modificar la sintaxis de la instrucción de cita remota de Drago, se consigue redirigir la llamada hacia el procedimiento suplente del grupo (ver apartado 5.3.1).

La estructura del código Ada-83 generado al analizar el cliente sería la siguiente:

```
1:  procedure Cliente is
2:
3:      package Entorno_Ejecucion is
4:          ...
5:
6:      package Servidor_Ficheros is
7:          procedure Leer(In_Param_1: in ...; Out_Param_1:out ...);
8:          procedure Escribir(....);
9:          Error_de_estado: Exception;
10:         Error_de_modo : Exception;
11:         ...
12:     end Servidor_Ficheros;
13:
14:     task Agente;
15:
16:     package body Entorno_Ejecucion is
17:         ...
18:
19:     package body Servidor_Ficheros is
20:         ...
21:
22:     task body Agente is
23:         – Código del cliente
24:         ...
25:     end Agente;
26:
27: begin
28:     null;
29: end Cliente;
```

El identificador del procedimiento que contiene el código Ada 83 con la traducción del agente coincide con el identificador del agente (línea 1). Este procedimiento contiene internamente el paquete *Entorno_Ejecucion* (líneas 3..4, 16..17), el paquete con los suplentes del grupo (líneas 6..12, 19..20) y la tarea *Agente* (líneas 14, 22..25).

El identificador del paquete con los suplentes del grupo es el identificador del grupo del cual es cliente el agente: *Servidor_Ficheros* (línea 6). En su especificación contiene todas las declaraciones contenidas en la sección intergrupo de la especificación del grupo (líneas 9..11) así como la declaración de procedimientos con el mismo identificador y perfil de parámetros que los puntos de entrada remotos declarados en la sección intergrupo de la especificación del grupo (líneas 7..8) y la parte privada de la especificación del grupo.

El cuerpo de la tarea *Agente* (líneas 22..25) contiene la traducción a Ada 83 del código del agente (que describiremos en detalle más adelante).

Código Ada 83 de un procedimiento suplente de un grupo replicado

El cuerpo de los procedimientos suplente realiza las siguientes funciones:

1. Aplana el mensaje con los parámetros de entrada de la cita remota.
2. Lo envía al grupo.
3. Espera todos los mensajes con los parámetros de salida.
4. Extrae los parámetros de salida de los mensajes recibidos.

Siguiendo con el ejemplo anterior, veamos en detalle el cuerpo del procedimiento suplente asociado a la petición de cita remota en el punto de entrada *Leer*.

```
1:  procedure Leer(In_Param_1: in ...; Out_Param_1: out ... ) is
2:      Id_Cita :Entorno_Ejecucion.Id_Cita_Remota;
3:      Men_Peticion :Entorno_Ejecucion.Mensaje;
4:      Men_Respuesta:Entorno_Ejecucion.Mensaje;
5:      type Excepciones_Remotas is (Error_de_Estado,...,Otra_Excepcion,Ninguna);
6:      Excepcion_Remota:Excepciones_Remotas;
7:      Otra_excepcion:Exception;
8:
9:      procedure Aplanar_parametros_in(Men: out Entorno_Ejecucion.Mensaje;
10:                                      In_Param_1:in ...) is ...
11:
12:     procedure Desaplanar_parametros_out(Men:in Entorno_Ejecucion.Mensaje;Out_Param_1:out ...;
13:                                         Excepcion_Remota:out Excepciones_Remotas) is ...
14:
15: begin
16:     Aplanar_parametros_in(Men_Peticion,In_Param_1,...);
17:     Id_Cita:=Entorno_Ejecucion.Enviar_Peticion_Cita(Servidor_Ficheros,Leer,Men_Peticion);
18:     Entorno_Ejecucion.Recibir_Respuesta(Id_Cita,Men_Respuesta);
19:     Desaplanar_parametros_out(Men_Respuesta,Out_Param_1,...,Excepcion_Remota);
20:     case Excepcion_Remota is
21:         when Error_de_Estado => raise Error_de_Estado;
22:         ...
23:         when Otra_Excepcion => raise Otra_Excepcion;
24:         when Ninguna => null;
25:     end case;
26: end Leer;
```

Analicemos en detalle el código anterior. Comencemos por la sección de declaraciones (líneas 2..13):

- Las líneas 2..4 contienen la declaración de las variables asociadas con la petición de cita remota: la variable *Id_Cita* (línea 2) almacenará el identificador asociado a la cita remota, la variable *Men_Peticion* (línea 3) se utilizará para crear el mensaje de petición con los parámetros de entrada de la cita remota y la variable *Men_Respuesta* (línea 4) se utilizará para almacenar el mensaje con los parámetros de salida[8].

- Las línea 5 contienen la declaración del tipo enumerado *Excepciones_Remotas* que contiene un literal por cada una de las posibles excepciones remotas (así como el literal *Otra_Excepcion* para el caso de que se propague una excepción remota no visible y el literal *Ninguna* para el caso de que no se eleve ninguna excepcion remota). La línea 6 contiene la declaración de una variable de este tipo y la línea 7 contiene la declaración de la excepción local *Otra_Excepcion* que eleva el procedimiento suplente cuando cuando se propaga una excepción remota no visible.

- Las líneas 9..13 contienen los procedimientos de aplanado y desaplanado de los mensajes que se transmiten por la red (tal y como se describen en el apartado 5.2.1). El procedimiento de desaplanado proporciona los parámetros de salida y las excepciones remotas.

Veamos ahora en detalle el bloque de sentencias del procedimiento suplente:

- La línea 16 llama al procedimiento que aplana los parámetros de entrada en el mensaje *Men_Peticion*.

- La línea 17 envía el mensaje de petición de cita remota al grupo *Servidor_Ficheros* en el punto de entrada remoto *Leer*[9]. Recibe en el parámetro de salida *Id_Cita* el identificador de la cita remota.

- La línea 18 espera el mensaje de respuesta con los parámetros de salida[10]. Si muere el último miembro vivo del grupo antes de enviar su mensaje de respuesta, *Entorno_Ejecucion* eleva la excepción *Group_Error*.

- La línea 19 llama al procedimiento que desaplana los parámetros de salida (que incluyen las excepciones remotas).

- Finalmente, si se propagó alguna excepción remota, en las líneas 20..25 se eleva la excepción correspondiente.

5.2.3 Suplente de un cliente remoto

Cuando un agente es miembro de un grupo, debe proporcionar todos los servicios declarados en la especificación del grupo y, simultáneamente, puede solicitar los servicios declarados en

[8] En el caso de un suplente de un grupo cooperativo esta línea contiene la declaración de una formación de mensajes (para almacenar todos los mensajes de respuesta del grupo).

[9] *Servidor_Ficheros* y *Leer* son en este caso literales de un tipo enumerado que se utilizan para especificar el punto de entrada remoto.

[10] En el caso de un suplente de un grupo cooperativo esta línea se sustituye por un bucle que espera todas las respuestas.

la sección intragrupo de la especificación del grupo. Por lo tanto, la especificación del paquete con el identificador del grupo contiene[11]:

- Todas las declaraciones de constantes, tipos y excepciones contenidas en la especificación del grupo en las secciones intergrupo, intragrupo y privada.
- Declaración de un procedimiento suplente del grupo por cada uno de los puntos de entrada declarados en la sección intragrupo de la especificación del grupo. El identificador de estos procedimientos es el identificador de los puntos de entrada remotos. El perfil de los parámetros de este procedimiento coincide exactamente con el perfil de los parámetros del punto de entrada correspondiente.
- La declaración de un tipo tarea por cada punto de entrada declarado en la especificación del grupo (tareas suplentes de los clientes remotos)[12].

Los procedimientos suplentes realizan las mismas funciones que los procedimientos suplentes descritos en el apartado 5.2.2 y permiten a los miembros de un grupo llamar a los puntos de entrada especificados en la sección intragrupo.

El cuerpo de cada una de las tareas suplentes de los clientes remotos realiza las siguientes funciones:

1. Desaplana los parámetros de entrada de un mensaje de cita remota.
2. Realiza la cita con la tarea *Agente* en el punto de entrada especificado en el mensaje (con los parámetros de entrada contenidos en el mensaje).
3. Si la cita no ha sido reencolada, crea un mensaje con los parámetros de salida y se lo envía al cliente remoto.
4. Realiza una segunda cita con *Agente* pasándole como parámetros el valor de los parámetros de entrada y salida de la cita remota.
5. Finaliza su ejecución.

Esta segunda cita es necesaria porque:

- Si la cita remota no ha sido reencolada, es necesario avisar a la tarea *Agente* que se ha realizado el envío del mensaje de respuesta al cliente remoto[13].
- Si la cita remota ha sido reencolada, la tarea suplente del cliente remoto debe pasar el valor de los parámetros de salida a la tarea *Agente*, para que pueda implementar el reencolamiento (ver apartado 5.3.3).

[11] Si un agente es cliente y miembro de un grupo simultáneamente, se crea un único paquete que contiene todas las declaraciones y suplentes necesarios para realizar la comunicación remota como cliente y miembro de dicho grupo.

[12] Cada vez que se necesita realizar una cita remota se crea una tarea que proporciona el flujo del cliente remoto. Cuando esta tarea finaliza su trabajo simplemente finaliza su ejecución (ver apartado 5.3.1).

[13] Ya que podría darse el caso de que una cita posterior completase su trabajo y enviase su mensaje de respuesta antes que esta, con lo que se rompería la causalidad.

Ejemplo

Siguiendo con el ejemplo anterior, supongamos el siguiente agente que es miembro del grupo *Servidor_Ficheros*.

```
for group Servidor_Ficheros;
replicated agent Miembro_de_Servidor_Ficheros is
    – Código del servidor
    ...
end Miembro_de_Servidor_Ficheros;
```

La estructura del código Ada-83 que se genera al traducir este agente es la siguiente:

```
1:  procedure Miembro_Servidor_Ficheros is
2:
3:      package Entorno_Ejecucion is
4:          ...
5:      package Servidor_Ficheros is
6:          task type Suplente_Leer is ...
7:          task type Suplente_Escribir is ...
8:          Error_de_Estado:Exception;
9:          Error_de_Modo:Exception;
10:         ...
11:     end Servidor_Ficheros;
12:
13:     task Agente is
14:         entry Servidor_Ficheros_Leer(In_Param_1: in...; Out_Param_1: out ...;
15:                                      Reencolada: out Boolean);
16:         entry Servidor_Ficheros_Fin_Leer(In_Param_1: in...; Out_Param_1: in...;
17:                                      Reencolada: in Boolean);
18:         entry Servidor_Ficheros_Escribir(...);
19:         entry Servidor_Ficheros_Fin_Escribir(...);
20:     end Agente;
21:
22:     package body Entorno_Ejecucion is
23:         ...
24:
25:     package body Servidor_Ficheros is
26:         ...
27:
28:     Task body Agente is
29:         – Código del servidor
30:         ...
31:     end Agente;
32:
```

```
33: begin
34:       null;
35: end Miembro_Servidor_Ficheros;
```

El identificador del procedimiento coincide con el identificador del agente (línea 1). Este procedimiento contiene internamente el paquete *Entorno_Ejecucion* (líneas 3..4, 22..23), un paquete con los suplentes del grupo *Servidor_Ficheros* (líneas 5..11, 25..26) y la tarea *Agente* (líneas 13..20, 28..31).

La especificación del paquete *Servidor_Ficheros* contiene todas las declaraciones existentes en la especificación del grupo (líneas 8..10), así como la declaración de un tipo tarea por cada uno de los puntos de entrada remotos declarados en la especificación del grupo (líneas 6..7).

La especificación de la tarea *Agente* contiene la declaración de todos los puntos de entrada declarados en la especificación del grupo (líneas 14..15 y 18), así como un punto de entrada adicional (por cada punto de entrada declarado en la especificación del grupo), que se utiliza para recibir los parámetros de entrada y salida de cada cita remota (líneas 16..17 y 19)[14]. El identificador de cada uno de los puntos de entrada está formado por el identificador del grupo y el identificador de cada uno de los puntos de entrada remotos[15]. El perfil de los parámetros de los puntos de entrada coincide con el perfil de los parámetros de los puntos de entrada remotos, aunque contienen además el parámetro de salida *Reencolada* que indica si la cita es reencolada.

Código Ada 83 de una tarea suplente de un cliente remoto

Continuando con el ejemplo anterior, la especificación de la tarea suplente de los clientes remotos del punto de entrada *Leer* es:

```
1: task type Suplente_Leer is
2:     entry Recibir_Peticion(Id: in Entorno_Ejecucion.Id_Cita_Remota;
3:                            M: in Entorno_Ejecucion.Mensaje);
4: end Suplente_Leer;
```

Contiene la declaración del punto de entrada *Recibir_Peticion* (línea 2), que se utiliza para que la tarea suplente reciba el identificador asociado a la cita remota (necesario para que pueda enviar al cliente remoto el mensaje con los parámetros de salida), y el mensaje que contiene los parámetros de entrada. El cuerpo de esta tarea es el siguiente:

[14] En principio solamente es necesario pasar los parámetros de salida, pero si pasamos de nuevo los parámetros de entrada se simplifica la implementación de *Agente*.

[15] Esto es necesario porque un agente puede pertenecer a varios grupos que tengan puntos de entrada con el mismo identificador.

```
1:  task body Suplente_Leer is
2:      Id_Cita: Entorno_Ejecucion.Id_Cita_Remota;
3:      In_Param_1:...
4:          ...
5:      Out_Param_1:...
6:          ...
7:      Reencolada:Boolean;
8:      Men_Respuesta:Entorno_Ejecucion.Mensaje;
9:      type Excepciones_Remotas is (Error_de_estado,...,Otra_excepcion,Ninguna);
10:     procedure Desaplanar_parametros_In(M:in Entorno_Ejecucion.Mensaje;In_Param_1:out ... ) is ...
11:     procedure Aplanar_parametros_out(M:out Entorno_Ejecucion.Mensaje;Out_Param_1:in ...;
12:                                      Excepcion_Remota:in Excepciones_Remotas:=Ninguna) is ...
13:
14: begin
15:     accept Recibir_Peticion(Id:in Entorno_Ejecucion.Id_Cita_Remota;
16:                                      M:in Entorno_Ejecucion.Mensaje) do
17:         Id_Cita:=Id;
18:         Desaplanar_parametros_in(Men,In_Param_1,...);
19:     end Recibir_Peticion;
20:     Agente.Servidor_Ficheros_Leer(In_Param_1,..,Out_Param_1,...,Reencolada);
21:     if not(Reencolada) then
22:         Aplanar_parametros_out(Men_Respuesta,Out_Param_1,...);
23:         Entorno_Ejecucion.Enviar_respuesta(Id_Cita,Men_Respuesta);
24:     end if;
25:     Agente.Servidor_Ficheros_Fin_Leer(I  Cita,In_Param_1,..,Out_Param_1,..,Reencolada);
26: exception
27:     when Error_de_estado =>
28:         Aplanar_parametros_out(Men_Res  esta,Out_Param_1,..,Error_de_Estado);
29:         Entorno_Ejecucion.Enviar_respues.  Id_Cita,Men_Respuesta);
30:     when ...
31:     ...
32: end Suplente_Leer
```

Veamos en detalle la sección de declaraciones del cuerpo de la tarea:

- La línea 2 contiene la declaración de la variable *Id_Cita*, que se utiliza para almacenar el identificador asociado a la cita remota.
- Las líneas 3..6 contienen la declaración de variables auxiliares, que se utilizan para almacenar una copia de los parámetros de entrada y de salida de la cita remota.
- La línea 7 contiene la declaración de la variable *Reencolada*, necesaria para recibir un parámetro de salida adicional asociado a los puntos de entrada remotos que indica si la cita fué reencolada.
- La línea 8 contiene la declaración de la variable *Men_Respuesta*, que se utiliza para crear el mensaje con los parámetros de salida.

- La línea 9 declara el tipo enumerado *Excepciones_Remotas*, que contiene un literal por cada una de las posibles excepciones remotas a propagar por este suplente.

- Las líneas 10..12 contienen los procedimientos de aplanado y desaplanado de los mensajes.

Veamos ahora el bloque de sentencias del cuerpo de la tarea suplente:

- En la línea 15 espera en una cita el identificador asociado a la cita y el mensaje que contiene los parámetros de entrada: la línea 17 almacena el identificador de la cita en la variable auxiliar *Id_Cita*; la línea 18 llama al procedimiento que realiza el desaplanado de los parámetros de entrada.

- En la línea 20 realiza una cita local con la tarea que implementa el código del agente (en el punto de entrada especificado por el cliente remoto), pasando como parámetros de entrada los parámetros de entrada contenidos en el mensaje. La implementación de los puntos de entrada remotos contiene un parámetro de salida adicional (*Reencolada*), que avisa a la tarea suplente si la cita fue reencolada o no.

- En las líneas 22..23, si la cita no fue reencolada, se llama a la función de aplanado de los parámetros de salida en el mensaje *Men_Respuesta*, y se envía al cliente remoto utilizando el identificador de la cita *Id_Cita*.

- En la línea 25 se realiza una nueva cita con *Agente*, pasándo el valor actual de los parámetros de entrada y salida (tal y como se describió en el apartado 5.2.3).

- Finalmente, en las líneas 27..31 se tratan las excepciones. Si la tarea *Agente* elevó una excepción que debe propagarse al cliente remoto, se pasa el literal correspondiente al procedimiento de aplanado (en el parámetro de entrada *Excepcion_Remota* —línea 27—), y se envía el mensaje al cliente remoto (línea 28).

5.3 Traducción de las sentencias de Drago

Todo el código del usuario que contiene un agente se traduce a sentencias Ada 83 dentro del cuerpo de la tarea *Agente*. En este capítulo se muestra un posible esquema de traducción de las siguientes sentencias de Drago relacionadas con la comunicación remota con grupos:

- Traducción de la sentencia de cita con un grupo.

- Traducción de una sentencia de aceptación de cita remota (con o sin alternativas).

- Traducción de la sentencia de reencolamiento.

- Traducción de una sentencia de cita remota con reencolamiento.

5.3.1 Cita remota con un grupo.

El código que se genera al traducir la sentencia Drago de cita remota con un grupo debe llamar al procedimiento suplente del grupo (tal y como muestra la figura 5.4). La sintaxis de la sentencia Drago de cita remota con un grupo de Drago coincide con la notación punto que se utiliza en Ada-83 para cualificar la llamada a subprogramas contenidos dentro de un paquete. Por lo tanto, dado que el suplente del grupo se genera dentro de un paquete local, cuyo identificador coincide con el identificador del grupo remoto y dentro de un procedimiento con el mismo identificador que el punto de entrada remoto (descrito en el apartado 5.2.2), no es necesario traducir la sentencia Drago de cita remota con un grupo.

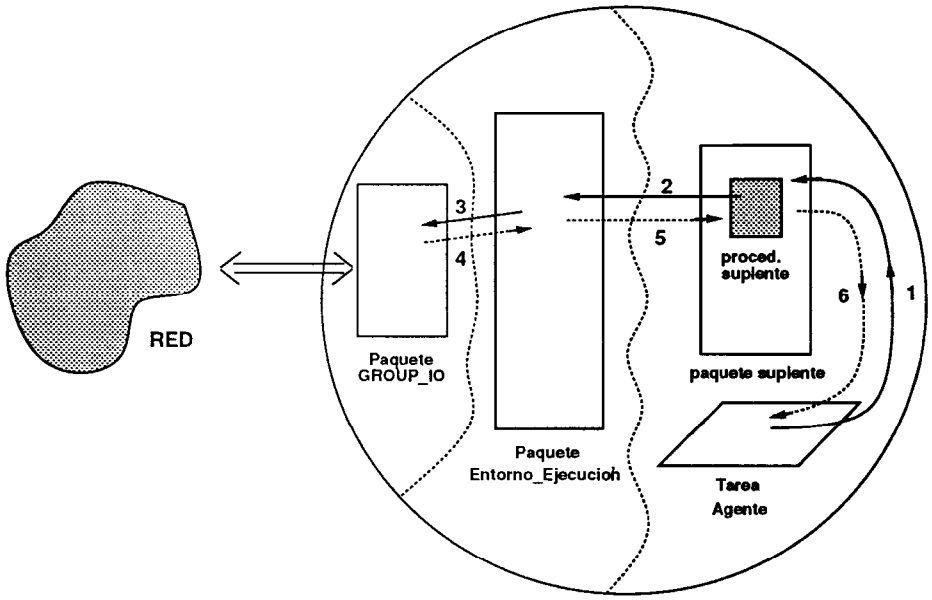


Figura 5.4: Secuencia de una cita remota.

Ejemplo

Supongamos que un cliente escrito en Drago contiene el siguiente código:

```
with group Servidor_Ficheros;
replicated agent Cliente is
begin
    Servidor_Ficheros.Leer(....);
end Cliente;
```

El código que se genera al traducir este agente a Ada 83 es el siguiente:

```
procedure Cliente is

    package Servidor_Ficheros is
        procedure Leer(....);
        procedure Escribir(....);
        Status_Error: Exception;
        Mode_Error : Exception;
        ...
    end Servidor_Ficheros;

    package body Servidor_Ficheros is
        ...

    task Agente;
    Task body Agente is
    begin
        Servidor_Ficheros.Leer(....);
    end Agente;


begin
    null;
end Cliente;
```

Aunque la sintaxis es la misma, el código de la llamada al punto de entrada remoto está realmente realizando una llamada al procedimiento *Leer* del paquete local *Servidor_Ficheros* que contiene el suplente del grupo.

5.3.2 Sentencia *select*

La sentencia **select** de Drago es sintácticamente muy parecida a la sentencia **select** de Ada 83 (aunque carece de las alternativas alternativas *delay*, *else* y *terminate* de Ada 83). La diferencia principal es que la sentencia **select** de Drago es determinista, mientras que la sentencia **select** de Ada 83 es no-determinista. Por lo tanto, para traducir la sentencia **select** de Drago a una sentencia **select** de Ada 83, es necesario generar código adicional que proporcione la semántica del **select** de Drago. Este código adicional hace lo siguiente:

1. Evalua todas las condiciones especificadas por el usuario que tengan algún mensaje encolado.

2. Mientras no haya un mensaje encolado en alguna de las alternativas especificadas en la sentencia *select* cuya condición sea cierta, solicita al paquete *Entorno_Ejecucion* que pase

el mensaje de petición de cita remota más antiguo a la cola correspondiente, actualizando su atributo **count**.

3. Pide al paquete **Entorno_Ejecucion** que extraiga de la cola correspondiente el mensaje más antiguo cuya condición sea cierta y cree la tarea suplente del cliente remoto.
4. Ejecuta una sentencia **select** de Ada 83 que contiene todas las alternativas de la sentencia **select** de Drago, pero sin las condiciones[16].

De esta forma, aunque la sentencia **select** de Ada-83 es no-determinista, el agente tiene un comportamiento determinista, porque solamente hay una tarea intentando realizar la cita con una de las alternativas que tienen su condición abierta: la tarea suplente que acaba de crear el paquete *Entorno_Ejecucion*. La figura 5.5 muestra el flujo del código Ada 83 que implementa la sentencia **select** de Drago.

La sentencia **accept** es un caso particular de la sentencia **select** y se implementa como ésta.

[16]No son necesarias porque el código anterior acaba de evaluarlas.

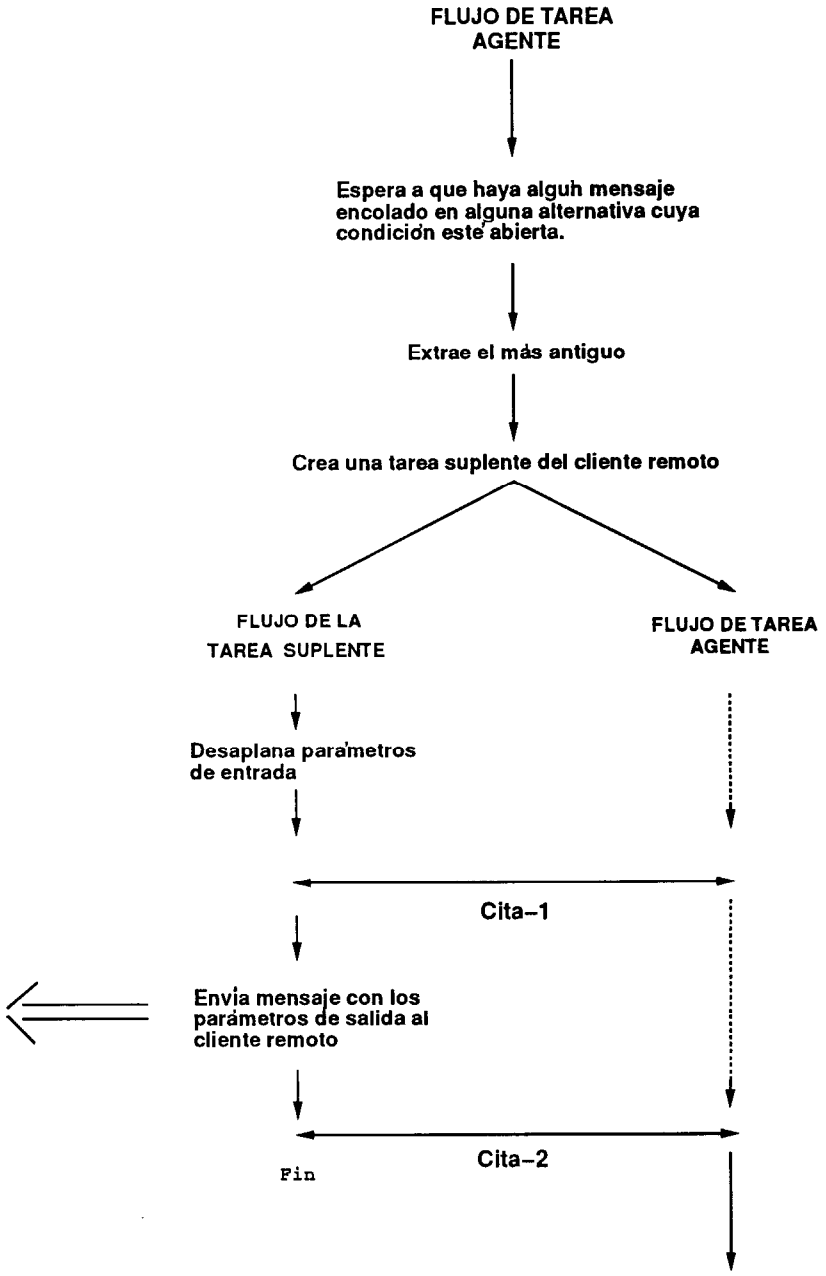


Figura 5.5: Flujo del código Ada 83 que implementa la sentencia **select** de Drago.

Código Ada 83 de una sentencia *select*

Siguiendo con el ejemplo anterior, supongamos la siguiente sentencia **select** de Drago dentro del bloque de sentencias un agente que es miembro del grupo *Servidor_Ficheros*:

```
select
     when <Condicion_1>=>
     accept Leer(....) do
          ...
     end;
or
     when <Condicion_2>=>
     accept Escribir(....) do
          ...
     end;
or
     ...
end select;
```

La traducción de esta sentencia en código Ada 83 es la siguiente:

```
1:  declare
2:      Alguna_alternativa_disponible:Boolean:=False;
3:
4:      Procedure Evaluar_condiciones(Alguna_alternativa_disponible:out Boolean) is
5:          Valor_condicion:Boolean:=False;
6:          Alguna_alt_disp:Boolean:=False;
7:          Alguna_condicion_abierta:Boolean:=False;
8:      begin
9:          Valor_condicion:=<Condicion_1>;
10:         Alguna_condicion_abierta:= Alguna_condicion_abierta OR Valor_condicion;
11:         if Entorno_Ejecucion.Count(Servidor_Ficheros_Leer)>0 then
12:             Entorno_Ejecucion.Actualizar_Condicion(Servidor_Ficheros_Leer,Valor_condicion);
13:             Alguna_alt_disp:=Alguna_alt_disp OR Valor_condicion;
14:         end if;
15:         Valor_condicion:=<Condicion_2>;
16:         Alguna_condicion_abierta:= Alguna_condicion_abierta OR Valor_condicion;
17:         if Entorno_Ejecucion.Count(Servidor_Ficheros_Escribir)>0 then
18:             Entorno_Ejecucion.Actualizar_Condicion(Servidor_Ficheros_Escribir,Valor_condicion);
19:             Alguna_alt_disp:=Alguna_alt_disp OR Valor_condicion;
20:         end if;
21:         ...
22:         if not(Alguna_condicion_abierta) then
23:             raise PROGRAM_ERROR;
24:         end if;
```

```
25:         Alguna_Alternativa_disponible:=Alguna_alt_disp;
26:     end Evaluar_condiciones:
27:
28: begin
29:     loop
30:         Evaluar_Condiciones(Alguna_alternativa_disponible);
31:         exit when Alguna_alternativa_disponible;
32:         Entorno_Ejecucion.Encolar_siguiente_peticion;
33:     end loop;
34:     Entorno_Ejecucion.Crear_Suplente;
35: end;
36: select
37:     accept Leer(...) do
38:         ...
39:     end;
40: or
41:     accept Escribir(...)
42:         ...
43:     end;
44: or
45: ...
46: end select;
```

Las alternativas de una sentencia **select** pueden especificar condiciones que habiliten o deshabiliten la aceptación de las alternativas. Estas condiciones son expresiones lógicas que pueden utilizar cualquier variable y/o función que sea visible en el punto donde se encuentra la instrucción **select**. Por lo tanto, es necesario generar el código de evaluación de las condiciones exactamente en el punto donde se encuentra la instrucción **select**[17]. Para ello creamos un nuevo ámbito mediante la instrucción **declare** de Ada 83 (línea 1).

La línea 2 contiene la declaración de la variable auxiliar *Alguna_alternativa_disponible*. Las líneas 4..26 contienen la declaración de un procedimiento que evalúa todas las alternativas del **select** que tienen algún mensaje de petición de cita remota encolado. Devuelve en el parámetro de salida *Alguna_alternativa_disponible* un valor lógico, en el que indica si hay algún mensaje de petición de cita remota encolado en alguna de las alternativas especificadas en el **select** cuya condición esté abierta. Las líneas 28..35 contienen el código adicional que proporciona la semántica del **select** de Drago de la siguiente forma:

1. Evalúa todas las condiciones de las alternativas especificadas en el **select** (línea 30).

2. Si hay alguna petición de cita remota encolada en alguna alternativa abierta, crea la tarea suplente del cliente remoto (línea 34).

[17] Si Ada 83 tuviese punteros a funciones no sería necesario crear una función de evaluación cada vez que se traduce una instrucción **select**. Podrían generarse funciones con el código de las condiciones y pasarlas mediante punteros a una función de evaluación.

3. Si no hay ninguna petición de cita remota encolada disponible, deja pasar un mensaje de petición de cita remota del nivel de comunicación con grupos al nivel de gestión de colas remotas y lo encola en la cola correspondiente actualizando el atributo **count** (línea 32). A continuación vuelve a evaluar todas las condiciones asociadas a las alternativas.

5.3.3 Sentencia *requeue*

La sentencia de reencolamiento (requeue) solamente está permitida dentro del bloque de sentencias de una sentencia accept. De acuerdo con lo expuesto en el apartado 5.3.2, el código Ada 83 que implementa la sentencia accept crea una tarea suplente del cliente remoto. Esta tarea desaplana los parámetros de la cita remota y realiza una cita con la tarea *Agente* (ver **A** en figura 5.6). En el instante del reencolamiento, la tarea suplente del cliente remoto está bloqueada en la cita, mientras que la tarea *Agente* está ejecutando la sentencia de reencolamiento dentro del bloque de sentencias asociado al accept (ver **B** en figura 5.6).

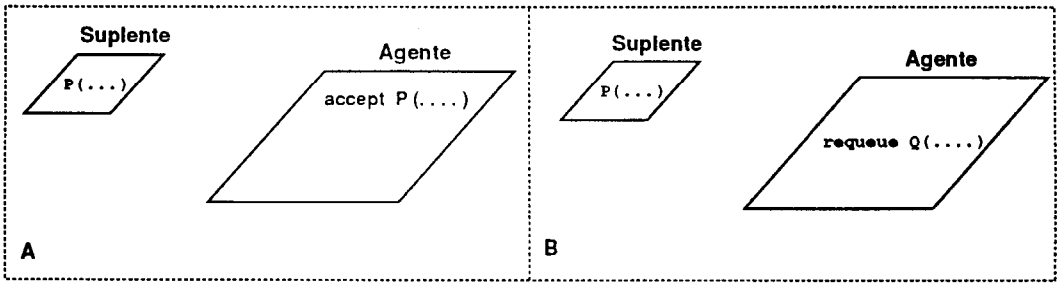


Figura 5.6: Estado del programa Ada 83 en el instante del reencolamiento.

A partir de este instante, el código Ada 83 que implementa la llamada con reencolamiento debe hace lo siguiente:

- Crea un mensaje *M* con el contenido de los parámetros de entrada recibidos en el mensaje de petición de cita remota, el contenido actual de los parámetros de salida y los parámetros de entrada adicionales.

- Llama al paquete *Entorno_Ejecucion* para encolar *M* en la cola correspondiente (ver **C** en figura 5.7).

- Finaliza la ejecución de la sentencia accept más interna, y avisa a la tarea suplente del cliente remoto (mediante un parámetro de salida adicional) que la cita ha sido reencolada (ver **D** en figura 5.7).

Cuando la tarea suplente del cliente remoto recibe una notificación de reencolamiento simplemente finaliza su ejecución (sin enviar al cliente remoto ningún mensaje —ver **E** en figura 5.7—).

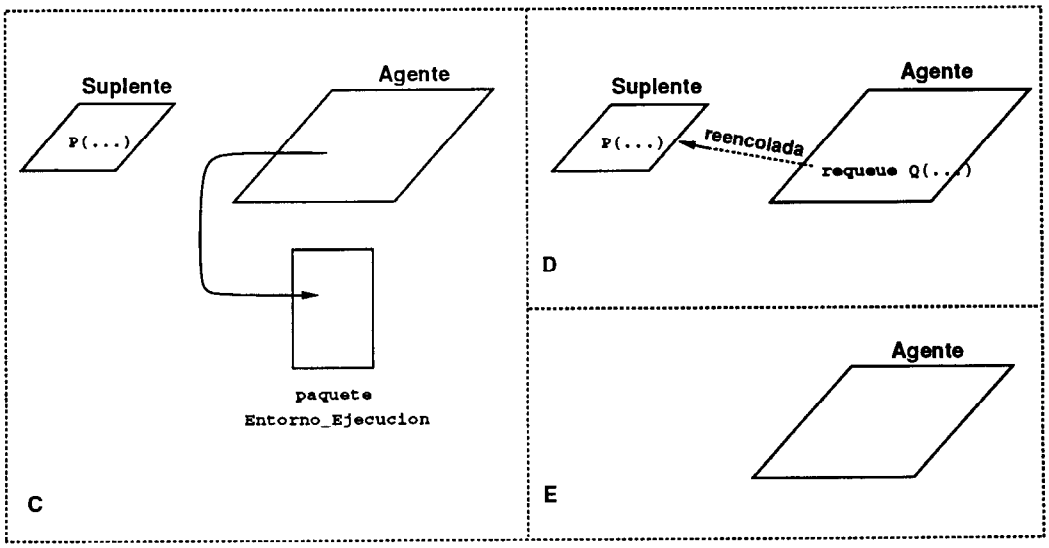


Figura 5.7: La tarea *Agente* gestiona el reencolamiento y lo notifica al suplente.

Obtención de los parámetros de salida

En el instante del reencolamiento, el código Ada 83 puede acceder a los parámetros de entrada que se deben pasar al punto de reencolamiento. Sin embargo, es necesario realizar algún truco para conseguir el valor de los parámetros de salida[18]. La solución propuesta consiste en realizar una cita adicional con la tarea suplente del cliente remoto, para recibir el valor de los parámetros de salida. Por lo tanto, el código Ada 83 que implementa la sentencia de reencolamiento hace lo siguiente:

- Guarda el contenido actual de los parámetros de entrada adicionales en variables auxiliares.
- Finaliza la ejecución de la sentencia **accept** más interna, y avisa a la tarea suplente del cliente remoto (mediante un parámetro de salida adicional) que hay un reencolamiento (ver **C** en figura 5.8).
- Realiza una cita adicional con la tarea suplente del cliente remoto para recibir el valor de los parámetros de salida (ver **D** en figura 5.8).
- Crea un mensaje *M* con el identificador de cita *Id_Cita* y el contenido actual de los parámetros de entrada y salida, y llama al paquete *Entorno_Ejecucion* para encolarlo en la cola correspondiente (ver **E** en figura 5.8).
- La tarea suplente del cliente remoto finaliza su ejecución (ver **F** en figura 5.8).

[18]Recordemos que Ada 83 no deja que el bloque de sentencias de una sentencia **accept** consulte el valor de los parámetros de salida.

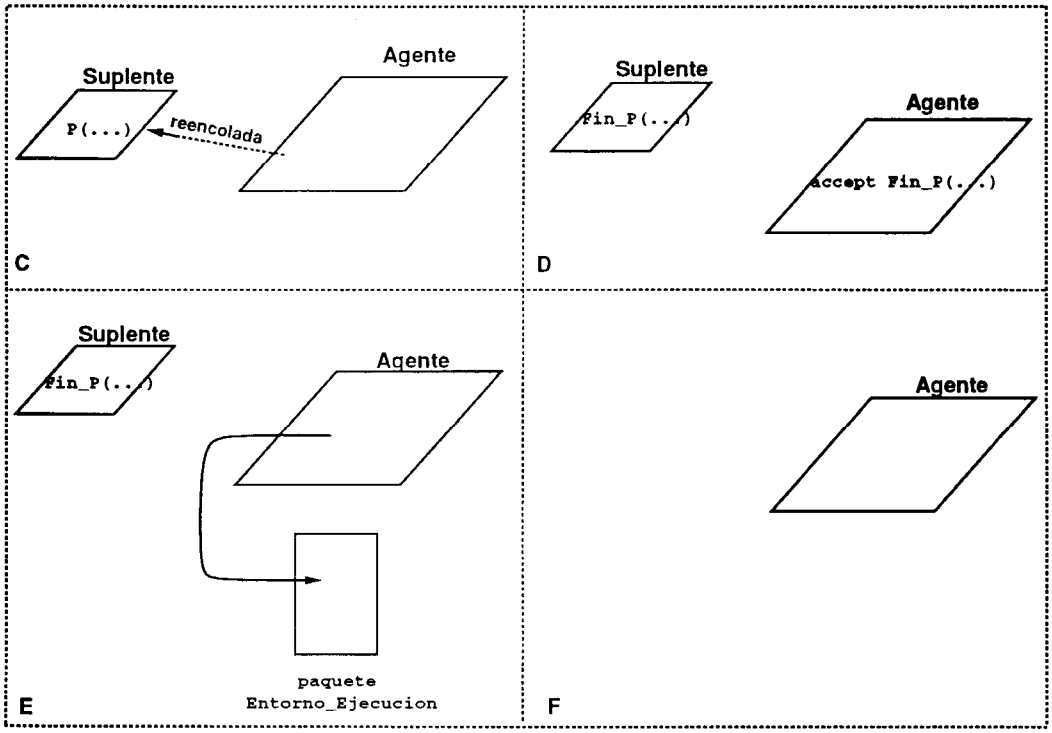


Figura 5.8: Visión detallada del reencolamiento.

Código Ada 83 de una sentencia *requeue*

Supongamos la siguiente secuencia de sentencias Drago:

```
accept Leer(In_Param_1: in ...; Out_Param_1: out ... ) do
    ...
    requeue Esperar(<Expresión>);
    ...
end Leer;
```

La traducción de esta sentencia en código Ada 83 es la siguiente:

```
1:  declare
2:     Men_Reencolamiento:Entorno_Ejecucion.Mensaje;
3:     Parametro_adicional:...
4:  begin
5:     accept Leer(In_Param_1:in ...;Out_Param_1:out ...;Reencolada:out Boolean) do
6:        ...
7:        Parametro_adicional:=<Expresión>;
8:        ...
9:        Reencolada:=True;
10:       return;
11:       ...
12:       Reencolada:=False;
13:    end Leer;
14:    accept Fin_Leer(Id_Cita,In_Param_1:in ...; Out_Param_1:in ...;Reencolada: in Boolean) do
15:       if Reencolada then
16:          Aplanar(Men_Reencolamiento,Id_Cita,In_Param_1,..,Out_Param_1,..,Parametro_adicional);
17:          Entorno_Ejecucion.Reencolar(Esperar,Mensaje_Reencolamiento);
18:       end if;
19:    end Fin_Leer;
20: end;
```

La sección de declaraciones solamente contiene la declaración de la variable *Mensaje_Reencolamiento* que se utiliza para crear el mensaje de reencolamiento (línea 2), así como una variable auxiliar para almacenar el valor del parámetro de entrada adicional de la sentencia de reencolamiento (línea 3).

Veamos ahora el contenido del cuerpo. La instrucción de aceptación de cita remota (línea 5) contiene el parámetro de salida adicional *Reencolada* que indica que se está ejecutando una instrucción de reencolamiento.

La instrucción de reencolamiento de Drago se traduce mediante dos fragmentos de código Ada 83 (líneas 5..13 y líneas 14..19). En el primer fragmento se guarda, en variables temporales, el valor de los parámetros de entrada adicionales de la sentencia de reencolamiento (línea 7), se avisa (mediante el parámetro de salida *Reencolada*) que la cita está siendo reencolada (línea 9) y se finaliza el bloque de sentencias del **accept** más interno (línea 10). En el segundo fragmento se espera, en una cita adicional, el valor de los parámetros de entrada y de salida de la cita que acaba de completarse[19] (línea 14), y, si la cita ha sido reencolada, se realiza el aplanado de los parámetros sobre el mensaje *Men_Reencolamiento* (línea 16) y se reencola en la cola correspondiente (línea 17).

[19] Solamente serían necesarios los parámetros de salida (ya que la tarea *Agente* podría almacenar los parámetros de entrada en variables locales en el punto de reencolamiento), pero, repetir el paso de los parámetros de entrada, no supone ninguna complicación adicional a la tarea suplente del cliente remoto, y simplifica el trabajo que debe hacer la tarea *Agente*.

5.3.4 Cita remota con reencolamiento

Al igual que ocurre con la sentencia **requeue** (descrita en el apartado anterior), la sentencia de cita remota con reencolamiento solamente está permitida dentro del bloque de sentencias de una sentencia **accept**. Por lo tanto, en el instante de la cita remota con reencolamiento la tarea suplente del cliente remoto está bloqueada en la cita, mientras que la tarea *Agente* está ejecutando el bloque de sentencias asociado al **accept** (ver **B** en figura 5.9).

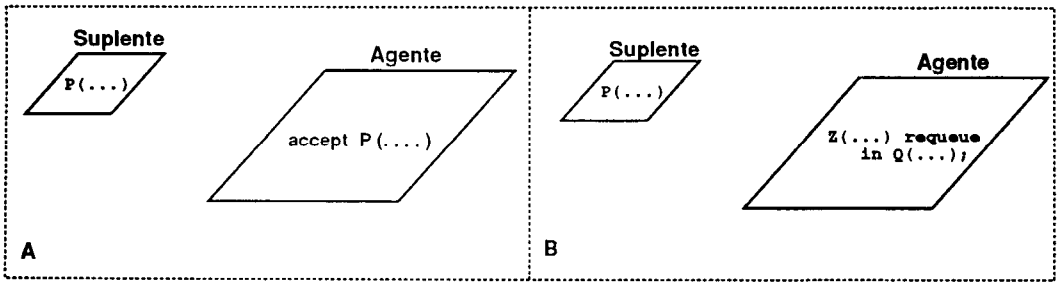


Figura 5.9: Estado del programa Ada 83 en el instante de una cita con reencolamiento.

Veamos una primera aproximación a la implementación de la cita remota con reencolamiento. Supongamos que tenemos punteros a procedimientos y que en el punto de la cita con reencolamiento podemos acceder a los parámetros de salida[20]. El código que implementa la llamada con reencolamiento hace lo siguiente:

- Aplana los parámetros de entrada de la cita remota en un mensaje *M* y lo envía al grupo sin esperar las respuestas (ver **C** en figura 5.10).
- Crea un mensaje *M'* con el contenido actual de los parámetros para el reencolamiento.
- Registra en el paquete *Entorno_Ejecucion* el identificador de la cita remota, el mensaje de reencolamiento *M'* y un puntero a un procedimiento que se encarga de recibir los mensajes de respuesta y actualizar el mensaje de reencolamiento con el valor de los parámetros de salida recibidos (ver **D** en figura 5.10)[21].
- Finaliza la ejecución de la sentencia **accept** más interna y avisa a la tarea suplente del cliente remoto (mediante el parámetro de salida adicional *Reencolada*) que la cita va a ser reencolada (ver **E** en figura 5.10).
- Cuando la tarea suplente del cliente recibe la notificación del reencolamiento simplemente finaliza su ejecución[22] (ver **F** en figura 5.10).

Toda esta secuencia es completamente determinista, y por lo tanto se repite exactamente igual en todas las réplicas (asegurando la consistencia de las réplicas). La gestión de las respuestas también es determinista, ya que el paquete *Entorno_Ejecucion* es el único que llama al procedimiento registrado mediante el puntero. Esto sólo ocurre en los siguientes casos:

[20] En el apartado anterior acabamos de ver que pueden obtenerse fácilmente mediante una cita adicional.

[21] Ada 83 no posee punteros a procedimiento. Sin embargo, podemos conseguir una funcionalidad equivalente mediante los tipos tarea (tal y como se describe en el siguiente apartado).

[22] Igual que ocurre en el caso de una instrucción de reencolamiento simple.

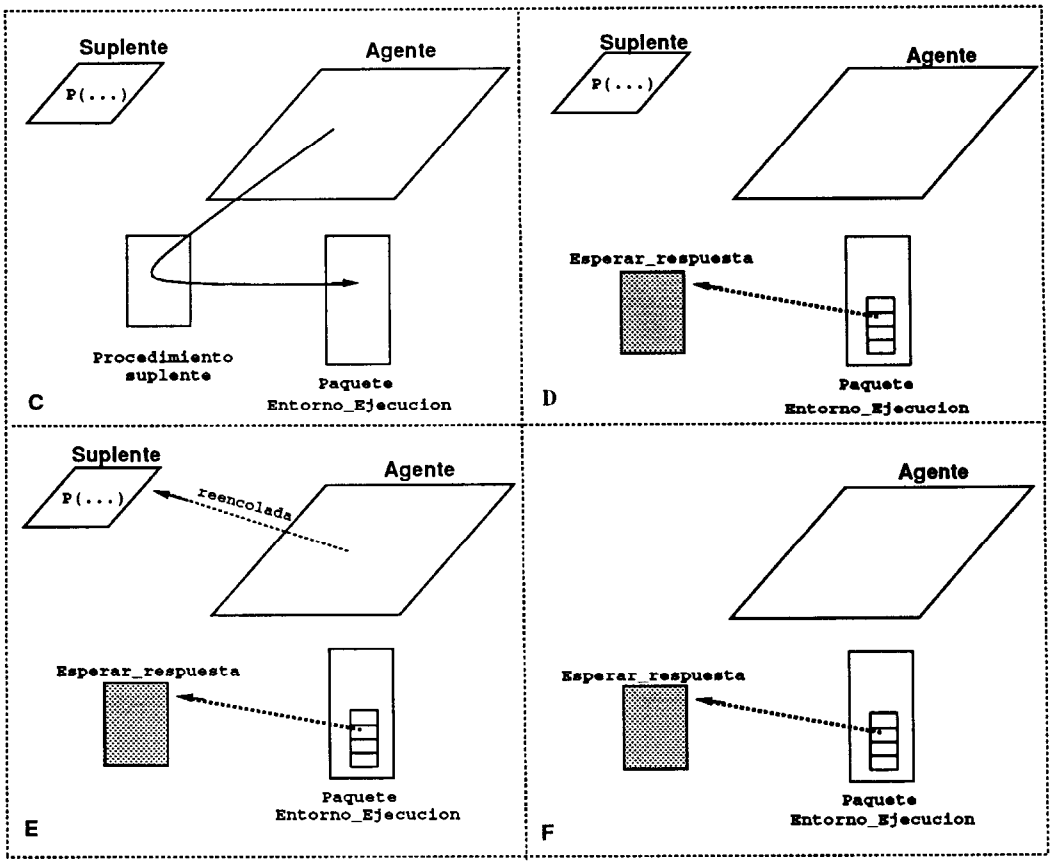


Figura 5.10: Cita remota con reencolamiento (I).

- Cuando el código del agente acepta una cita remota (con o sin alternativas), y no hay ningún mensaje encolado, en alguno de los puntos de entrada especificados, cuya condición esté abierta (ver **G** en figura 5.11).

- Cuando el código del agente necesita el siguiente mensaje de respuesta de una cita remota.

En ambos casos la tarea *Agente* llama al paquete *Entorno_Ejecucion* (ver **H** en figura 5.11) y éste solicita al paquete *GROUP_IO* el siguiente mensaje (ver **I** en figura 5.11). Si es un mensaje de respuesta para una cita remota con reencolamiento que está registrada, *Entorno_Ejecucion* llama (mediante el puntero) al procedimiento que trata el mensaje de respuesta (ver **J** en figura 5.11). Este procedimiento desaplana los parámetros de salida del mensaje de respuesta, actualiza el mensaje de reencolamiento *M'* y en caso de que sea el último mensaje de respuesta lo reencola en la cola correspondiente (actualizando el atributo **count**).

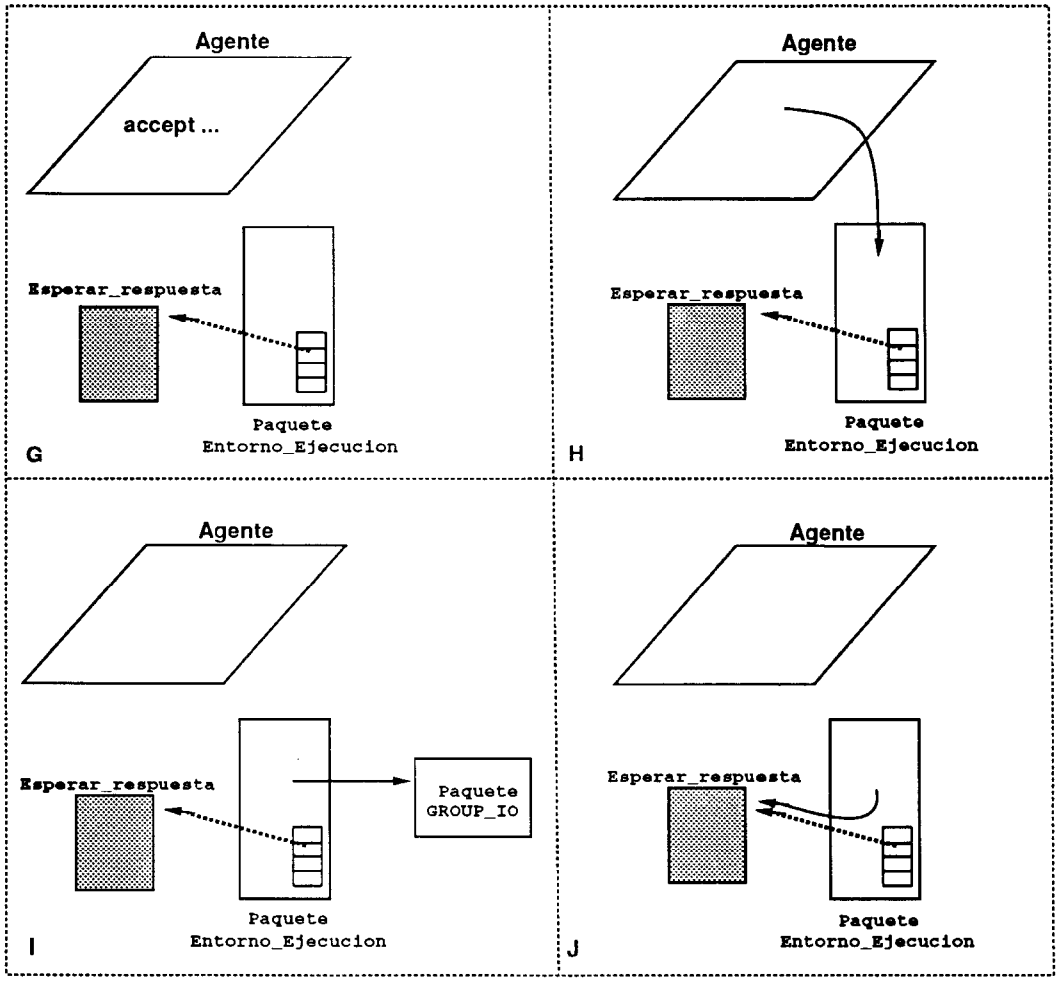


Figura 5.11: Cita remota con reencolamiento (II).

Código Ada 83 de una sentencia de cita remota con reencolamiento

El problema principal para implementar en Ada 83 el modelo presentado anteriormente es que Ada 83 no posee punteros a procedimiento[23]. Sin embargo, podemos conseguir una funcionalidad equivalente mediante los tipos tarea. Para ello creamos una tarea mediante un puntero y hacemos que la tarea bloquee al llamador (en este caso *Entorno_Ejecucion*) hasta que finalice de ejecutar todo el bloque de sentencias del procedimiento que sustituye). Esta tarea puede utilizar su ámbito para almacenar el valor de los parámetros de reencolamiento, evitando así que *Entorno_Ejecucion* almacene el mensaje de reencolamiento. Veámoslo en detalle a continuación.

El código que implementa la llamada con reencolamiento hace lo siguiente:

- Llama a un procedimiento suplente del grupo, que aplana los parámetros de entrada de la cita remota en un mensaje *M* y lo envía al grupo sin esperar las respuestas (ver **C** en figura 5.12).
- Guarda el contenido de los parámetros de entrada adicionales en variables temporales.
- Finaliza la ejecución de la sentencia **accept** más interna, y avisa a la tarea suplente del cliente remoto (mediante el parámetro de salida *Reencolada*) que hay un reencolamiento (ver **D** en figura 5.12).
- Realiza una cita adicional con la tarea suplente del cliente remoto para recibir el valor de los parámetros de salida (ver **E** en figura 5.12).
- Crea mediante un puntero una tarea que se encarga de esperar los mensajes con los parámetros de salida de la cita remota y realizar el reencolamiento.
- Sincroniza con la tarea recien creada, para pasarle el valor de los parámetros de reencolamiento (ver **F** en figura 5.12).
- Registra en el paquete *Entorno_Ejecucion* el identificador de la cita remota y el puntero a la tarea recien creada (ver **G** en figura 5.12).
- Cuando la tarea suplente del cliente recibe la notificación del reencolamiento, simplemente finaliza su ejecución[24] (ver **G** en figura 5.12).

[23] Los parámetros de salida podemos obtenerlos mediante una cita adicional, al igual que ocurre en la implementación del reencolamiento (ver apartado 5.3.3).

[24] Igual que ocurre en el caso de una instrucción de reencolamiento simple.

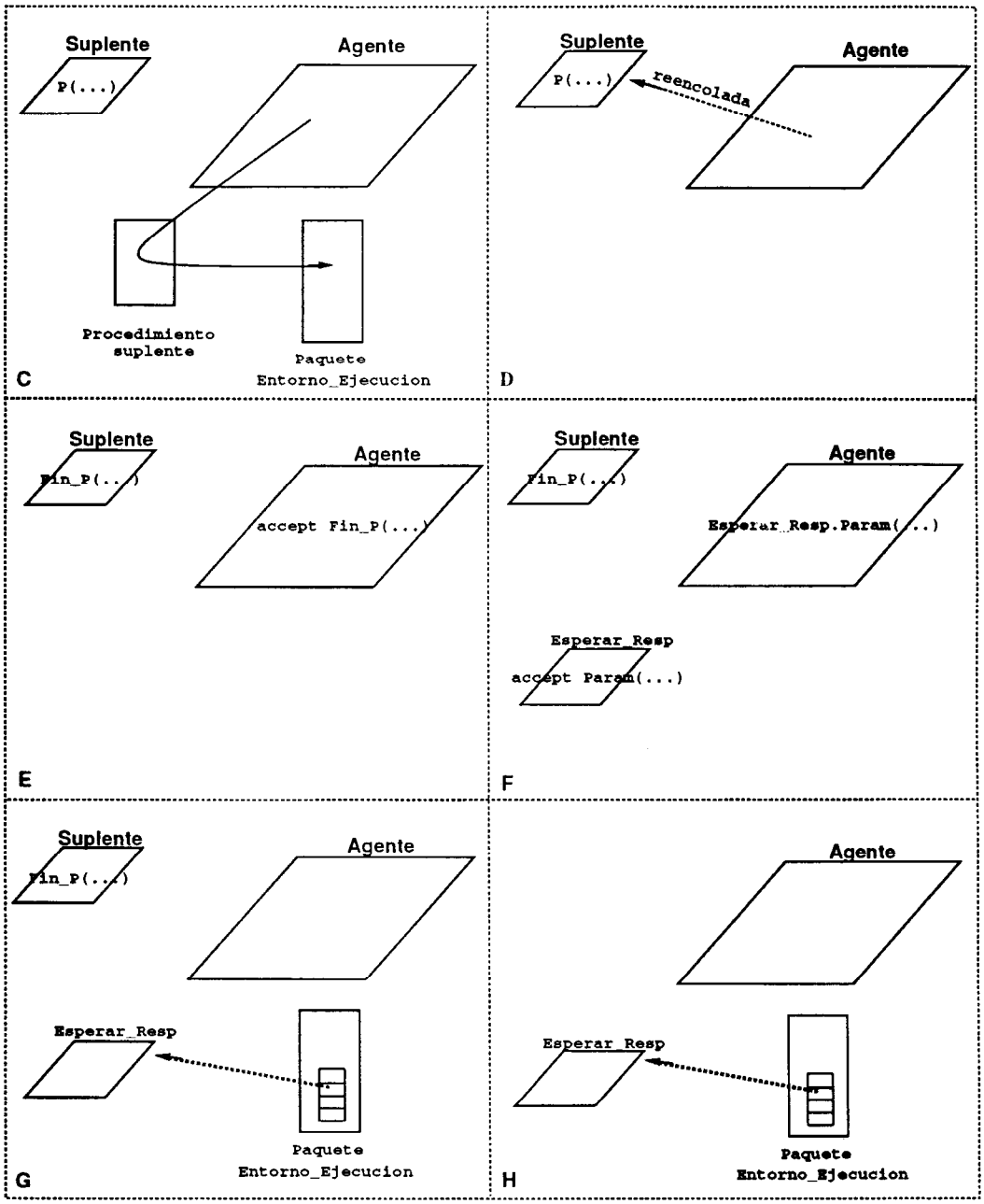


Figura 5.12: Cita remota con reencolamiento (III).

Ejemplo

Supongamos un servidor de disco remoto implementado mediante un grupo replicado con la siguiente especificación de grupo:

```
replicated group specification Servidor_Disco is
   type Contenido_Fichero is ...
   entry Leer_Fichero(Nombre_Fichero: in String; Respuesta: out Contenido_Fichero);
end Servidor_Disco;
```

Supongamos además la siguiente secuencia de sentencias de un agente que es miembro del grupo *Servidor_Ficheros* y a la vez cliente del grupo *Servidor_Disco*:

```
1:  ...
2:  Tmp:Servidor_Disco.Contenido_Fichero;
3:  ...
4:  accept Leer(In_Param_1:in ... ;Out_Param_N:out ...) do
5:        ...
6:        Servidor_Disco.Leer_Fichero(In_Param_1,Tmp)
7:                                          requeue in Esperar(Tmp);
8:  ...
9:  end Leer;
```

Este agente acepta una cita remota al punto de entrada *Leer* (línea 4), y en el bloque de sentencias del **accept** realiza una cita remota con reencolamiento con el grupo *Servidor_Disco*. Para ello ha declarado la variable auxiliar *Tmp* (línea 2) que almacenará el parámetro de salida de la cita remota *Leer_Fichero* (línea 6) y se pasará como parámetro de entrada al punto de entrada de reencolamiento *Esperar* (línea 7).

El resultado de traducir las líneas 4..9 de este agente a código Ada 83 es el siguiente:

```
1:  declare
2:      P:Puntero_a_Esperar_Respuestas;
3:      Id_Cita:Entorno_Ejecucion.Id_Cita_Remota;
4:  begin
5:      accept Leer(In_Param_1: in ...;Out_Param_1: out ...; Reencolada: out Boolean) do
6:          ...
7:          Servidor_Disco.Leer_Fichero(In_Param_1,Id_Cita);
8:          Tmp:=<Valor recibido en parámetro Respuesta>;
9:          Reencolada:=true;
10:         return;
11:         ...
12:         Reencolada:=false;
13:     end;
14:     accept Fin_Leer(In_Param_1: in ...; Out_Param_1: in ...; Reencolada: in Boolean) do
15:         if Reencolada then
16:             P:=NEW Esperar_Respuestas;
17:             P.Toma_Parametros(In_Param_1,..,Out_Param_N,Tmp);
18:             Entorno_Ejecucion.Registrar(P,Id_Cita);
19:         end if;
20:     end Fin_Leer;
21: end;
```

Este código sigue exactamente los pasos descritos anteriormente:

- En la línea 7 llama a un procedimiento suplente del grupo remoto que aplana los parámetros de entrada y envía al grupo un mensaje con la petición de cita remota sin esperar por las respuestas (devuelve en un parámetro de salida el identificador asociado a la cita remota).
- En la línea 8 almacena en una variable temporal el valor actual del parámetro de entrada adicional *Respuesta* que se utiliza en el reencolamiento.
- En las líneas 9..10 finaliza la ejecución de la sentencia **accept** más interna y avisa a la tarea suplente del cliente remoto (mediante el parámetro de salida adicional *Reencolada*) que la cita va a ser reencolada.
- En la línea 14 realiza una segunda cita para recibir el valor actual de los parámetros de entrada y de salida de la cita remota que debe reencolarse.
- En la línea 16 crea (mediante un puntero) la tarea *Esperar_Respuesta* que va a esperar por las respuestas y realizar el reencolamiento.
- En las línea 17 sincroniza con la tarea *Esperar_Respuesta* para pasarle el valor actual de los parámetros necesarios para el reencolamiento.
- En la línea 18 llama al paquete *Entorno_Ejecucion* para registrar en una tabla el identificador de la cita remota y el puntero a la tarea recién creada.

La declaración del tipo tarea *Esperar_Respuestas* asociado a esta cita remota con reencolamiento es:

```
1: task type Esperar_Respuestas is
2:     entry Recibir_parametros(I_Param_1: in ... ; O_Param_1: in ...;
3:                              I_Respuesta: in Servidor_Disco.Contenido_Fichero);
4:     entry Siguiente_Respuesta(M: in Entorno_Ejecucion.Mensaje;
5:                               No_hay_respuesta: in Boolean);
6: end Esperar_Respuestas;
```

El punto de entrada *Recibir_Parametros* se utiliza para recibir los parámetros necesarios para realizar el reencolamiento. El punto de sentrada *Siguiente_Respuesta* se utiliza para recibir el siguiente mensaje de respuesta (en caso de no haber más respuestas se recibe el parámetro de entrada *No_hay_respuesta* a cierto). El cuerpo de esta tarea es el siguiente:

```
1:  task body Esperar_Respuestas is
2:      In_Param_1:...
3:      ...
4:      respuesta:...
5:      Out_Param_1:...
6:      ...
7:      type Excepciones_Remotas is (...);
8:      Excepcion_Remota:Excepciones_Remotas;
9:      Id_Cita:Entorno_Ejecucion.Id_Cita;
10:     Men_Reencolamiento:Entorno_Ejecucion.Mensaje;
11:
12:     procedure Desaplanar_Respuesta(M: in Entorno_Ejecucion.Mensaje;
13:                                    Respuesta: out Servidor_Disco.Contenido_Fichero;
14:                                    Excepcion_Remota: out Excepciones_Remotas) is ...
15:     procedure Aplanar_Parametros(M: out Entorno_Ejecucion.Message;
16:                                    Respuesta: out Disk_Server.File_Content;
17:                                    Excepcion_Remota: in Excepciones_Remotas) is ...
18:
19: begin
20:     accept Recibir_parametros(I_Param_1:in ... ; O_Param_1:in ...;
21:                                    I_Respuesta :in Servidor_Disco.Contenido_Fichero) do
22:         In_Param_1:=I_Param_1;
23:         ...
24:         Out_Param_1:=O_Param_1;
25:         ...
26:         Respuesta:=I_Respuesta;
27:     end Recibir_parametros;
28:     accept Siguiente_Respuesta(M:Entorno_Ejecucion.Mensaje;No_hay_respuesta:in Boolean) do
29:         if No_hay_respuesta then
30              Aplanar_parametros(Men_Reencolamiento,...,Group_Error);
31:         else
32:             Desaplanar_Respuesta(M,Respuesta,Excepcion_Remota);
33:             Aplanar_parametros(Men_Reencolamiento,In_Param_1,..,Out_Param_N,
34:                                    Respuesta,Excepcion_Remota);
35:         end if;
36:         Entorno_Ejecucion.Reencolar(Esperar,Men_Reencolamiento);
37:     end Siguiente_Respuesta;
38: end Esperar_Respuestas;
```

En las líneas 20..27 espera los parámetros de la sentencia de reencolamiento y los almacena en variables locales. En la línea 28 espera el mensaje con la respuesta de la cita remota con reencolamiento[25] (el paquete *Entorno_Ejecucion* sincronizará en este punto cuando desee

[25]En este caso espera solamente un mensaje de respuesta porque se realizó la cita remota con un grupo replicado. En caso de realizar una cita remota con reencolamiento con un grupo cooperativo la tarea realizaría un bucle de espera por todas las respuestas.

pasar un mensaje de respuesta). En caso de que haya muerto el último miembro del grupo recibe el parámetro de entrada *No_hay_respuesta* a cierto y pasa, como parámetro, el literal *Group_Error* al punto de reencolamiento. En la línea 32 desaplana el mensaje de respuesta, y en las líneas 33..34 crea el mensaje de reencolamiento. Finalmente en la línea 36 llama al paquete *Entorno_Ejecucion* para reencolar el mensaje y finaliza la cita.

5.4 Estructura interna del preprocesador de Drago

En los capítulos anteriores se ha descrito un posible esquema de traducción de Drago a Ada 83. En este capítulo se describe la estructura interna de un preprocesador de Drago.

5.4.1 Herramientas utilizadas

El preprocesador de Drago se ha escrito utilizando la herramienta Aflex [Sel90] (para generar el analizador léxico de Drago —ver figura 5.13—) y la herramienta Ayacc [TTR88] (para generar el analizador sintáctico —ver figura 5.14—). Aflex es una versión avanzada de Lex [Les75] para Ada 83; AYacc es una versión de la herramienta [Joh75] para Ada 83.

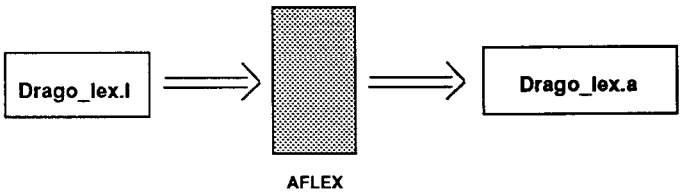


Figura 5.13: Aflex genera el analizador léxico.

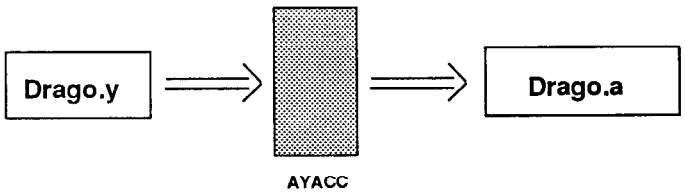


Figura 5.14: Ayacc genera el analizador sintáctico.

5.4.2 Estructura interna

El preprocesador de Drago ha sido estructurado en varios paquetes Ada 83 que realizan las siguientes funciones:

- Analizador léxico (generado mediante Aflex).
- Analizador sintáctico y el analizador semántico (generado mediante Ayacc), implementado mediante código entrelazado.
- Tabla de símbolos, que almacena las declaraciones existentes en el programa fuente que son necesarias para la generación de código.
- Generador de código, necesario para traducir a Ada 83 las sentencias Drago.
- *Buffer* de caracteres ASCII que se utiliza para generar el código intermedio.

La figura 5.15 muestra la estructura general del preprocesador de Drago mostrando (mediante flechas) las dependencias existentes entre los paquetes que lo componen.

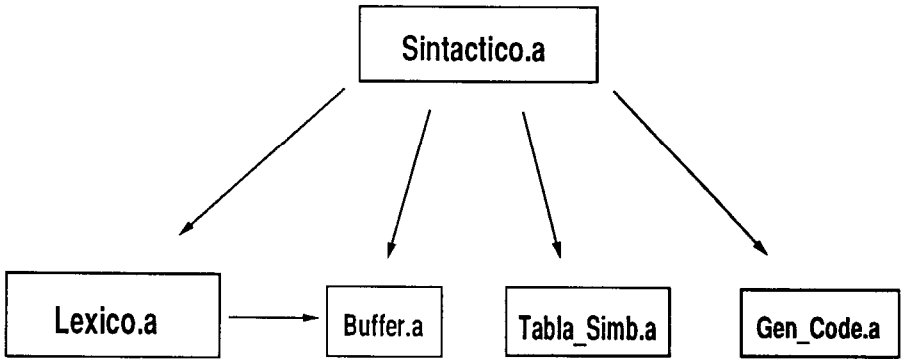


Figura 5.15: Estructura interna del preprocesador.

5.4.3 Funcionamiento del preprocesador

El funcionamiento del preprocesador es el siguiente:

1. El analizador sintáctico solicita al analizador léxico los tokens del programa fuente Drago.
2. El analizador léxico retorna al analizador sintáctico el siguiente token y simultáneamente lo vuelca en el *buffer* de código intermedio. De esta forma, el preprocesador no necesita generar las instrucciones Ada 83 existentes en el programa Drago (ya que pasan directamente al buffer de código intermedio).
3. Cuando el analizador sintáctico detecta el final de una sentencia Drago, llama al analizador semántico (implementado mediante código entrelazado con el analizador sintáctico, que es la forma usual de hacerlo mediante Yacc).

4. El analizador semántico realiza el análisis semántico de la sentencia y, en caso necesario:

   - Almacena información en la tabla de símbolos
   - Extrae la sentencia del buffer y llama al generador de código para generar el código Ada 83 equivalente.

5.5 Resumen

En este capítulo se ha descrito la estructura general de un posible esquema de traducción de código Drago a código Ada 83. Las especificaciones de grupo no necesitan traducción. Los agentes se traducen a programas Ada 83, estructurados mediante procedimientos que constan de: varios paquetes locales con los suplentes de los grupos de los cuales es cliente y/o miembro cada agente; el paquete local *Entorno_Ejecucion*; y la tarea *Agente* que contiene la traducción de todas las sentencias del programa Drago.

Por cada grupo del cual es cliente y/o miembro un agente, se genera un paquete con el identificador del grupo. La especificación de este paquete contiene todas las declaraciones contenidas en la especificación del grupo que son visibles al agente, así como los suplentes necesarios para implementar la comunicación remota. Los suplentes de grupo se implementan mediante procedimientos, mientras que los suplentes de los clientes remotos se implementan mediante tareas. Se ha descrito la estructura interna de un preprocesador de Drago que genera código Ada 83 de acuerdo con este modelo de traducción.

Capítulo 6

Conclusiones y trabajos futuros

Esta tesis ha conseguido el objetivo propuesto: el diseño e implementación de un lenguaje de programación para construir aplicaciones distribuidas tolerantes a fallos y cooperativas. Nos planteamos este objetivo como resultado de la experiencia con sistemas de programación tolerantes a fallos de más bajo nivel conceptual, en especial Isis. Pensamos que, para mejorar la práctica de la programación de este tipo de aplicaciones, era fundamental tener un soporte linguístico adecuado, y no sólo conjuntos de subrutinas, sin la posibilidad de disponer de las comprobaciones automáticas que realizan los compiladores de lenguajes modernos. No tendría mucho sentido embarcarse en la construcción de aplicaciones tolerantes a fallos partiendo de un soporte linguístico básico poco fiable.

Si bien nuestro interés estuvo inicialmente centrado en la programación tolerante a fallos mediante replicación de módulos software, muy pronto vimos la necesidad adicional de abordar de manera complementaria la programación con módulos cooperativos, no replicados. Este segundo enfoque se entendió importante para obtener otra de las principales ventajas de la programación distribuida, aparte de la tolerancia a fallos: el incremento de la capacidad de procesamiento.

Para simplificar la definición del lenguaje y su implementación, se hizo un esfuerzo importante para diseñar ambos tipos de módulos, replicados y cooperativos, de forma que fueran lo más parecidos posibles y, en aquellos puntos en que debían ser diferentes, las diferencias resultaran intuitivamente fáciles de comprender.

Decidimos desarrollar Drago como una extensión de un lenguaje existente que estuviese bien definido y fuese popular, modular y fuertemente tipado. De esta forma centramos nuestro esfuerzo en el desarrollo de las nuevas características del lenguaje (distribución, comunicación y sincronización remota, etc.).

Drago se desarrolló como una extensión del lenguaje de programación Ada 83. La elección de Ada 83 estuvo influida por su semántica de parámetros formales (parámetros de modo *in*, *out*, e *in out*), que se adapta muy bien a la comunicación remota, su soporte de excepciones (mecanismo conveniente para notificar fallos en la comunicación remota), y la existencia de compiladores Ada para numerosos sistemas operativos y procesadores.

Se ha realizado un esfuerzo importante para conseguir que la semántica de ejecución de

Drago sea lo más cercana posible a la semántica de Ada 83. De esta forma los programadores pueden utilizar Drago para obtener, con poco esfuerzo, la versión distribuida tolerante a fallos, de aplicaciones escritas en Ada 83.

Decidimos realizar la implementación de Drago para conseguir dos objetivos importantes: completar su diseño y desarrollar una tecnología de la que no disponíamos y que considerábamos necesaria para realizar posteriores investigaciones.

6.1 Aportaciones de esta tesis

Esta tesis ha realizado las siguientes aportaciones concretas:

1. Se ha proporcionado soporte linguístico a la abstracción de grupo, necesario para implementar aplicaciones distribuidas mediante el paradigma de comunicación con grupos. Una aportación importante de esta tesis ha sido el soporte para programar grupos cooperativos. Hemos encontrado en la literatura lenguajes que proporcionan soporte para programar aplicaciones mediante grupos replicados (por ejemplo, C-concurrente tolerante a fallos). Sin embargo, no hemos encontrado lenguajes que proporcionen soporte para aplicaciones cooperativas.
2. Se ha diseñado un mecanismo de comunicación remota uniforme con grupos (la cita remota con un grupo), así como sentencias de aceptación determinista de citas remotas (necesario para implementar aplicaciones replicadas sin necesidad de comunicación adicional entre las réplicas). Estos mecanismos facilitan la programación de aplicaciones tolerantes a fallos.
3. Se ha diseñado un mecanismo de notificación de fallos consistente, que permite a los miembros de grupos cooperativos gestionar fallos de manera coordinada.
4. Se ha diseñado un nuevo mecanismo de llamada remota asíncrona, la cita remota con reencolamiento. La llamada remota asíncrona que proporciona Ada 95 no permite parámetros de salida, mientras que la cita remota con reencolamiento, sí. Además, este nuevo mecanismo permite un cierto grado de paralelismo en servidores replicados sin necesidad de utilizar tareas adicionales.
5. Se ha implementado Drago mediante un preprocesador escrito con Alex y Ayacc, que genera código Ada 83. Esto permite instalar rápidamente Drago en diferentes sistemas operativos y/o redes de comunicación.

6.2 Líneas futuras de trabajo

Dentro de las posibles líneas de continuación de este trabajo, se pueden considerar de interés, tanto teórico como práctico:

1. Utilizar Drago para realizar prototipos de aplicaciones distribuidas reales, y con ello obtener mejor evaluación de sus características.

2. Reutilizar las aplicaciones escritas con Drago para analizar nuevos protocolos de comunicación con grupos. Esto es posible porque el subsistema de protocolos de comunicación de Drago está claramente separado del resto de la implementación del lenguaje. Lo cuál permite escribir nuevos protocolos y disponer así de implementaciones diversas de Drago con protocolos diferentes. De esa manera, puede evaluarse el resultado de ejecutar una misma aplicación con diferentes protocolos.

3. Ampliar la definición de Drago para tratar con grupos dinámicos (la versión actual solamente soporta grupos estáticos). En particular, es necesario analizar cómo puede pasarse estado a los nuevos miembros de grupos replicados.

4. Analizar qué soporte debe proporcionar Drago para tratar excepciones remotas de grupos cooperativos. En particular, para cuando cada miembro del grupo cooperativo eleva una excepción remota diferente. Para ello podrían analizarse mecanismos tales como grupos de excepciones, jerarquía de excepciones, prioridades entre las excepciones, etc. También debe analizarse cómo implementar excepciones remotas globales para conseguir una semántica de excepciones remotas más cercana a la que da Ada.

5. Incorporar en Drago nuevos mecanismos de tolerancia a fallos software. Aunque Drago proporciona soporte suficiente para implementar aplicaciones tolerantes a fallos software mediante la técnica de n-versiones (descrita en el capítulo 4), la disponibilidad del preprocesador desarrollado para Drago permite incorporar rápidamente nuevos mecanismos específicos para soportar fallos software.

6. Diseñar un lenguaje de configuración para las aplicaciones distribuidas Drago.

Apéndice A

Manual de referencia técnica de Drago

Este apéndice contiene el manual de referencia técnica de Drago. Todos los párrafos de este manual están numerados para facilitar su referencia.

A.1 Notación sintáctica

Para describir la sintaxis de Drago se ha utilizado la notación propuesta en el manual de referencia de Ada 83 [ANS83] (que es un formalismo extendido de la notación de Backus-Naur), que resumimos a continuación: 1

- Se utilizan palabras en minúsculas (algunas conteniendo el carácter de subrayado _ para denotar categorias sintácticas. 2
- Si el nombre de la categoría sintáctica comienza en letra *cursiva*, es equivalente al nombre de la categoría sintáctica sin la parte en letra cursiva. La parte en letra cursiva simplemente expresa alguna información semántica. Por ejemplo, *type*_name y *task*_name son ambas equivalentes a name. 3
- Las palabras reservadas se representan en **negrita**. 4
- Los corchetes [y] indican opcionalidad. 5
- Las llaves { y } denotan repetición (incluyendo ninguna vez). 6
- La barra vertical | separa posibles alternativas. 7

A.2 Modelo de sistema distribuido

Los programas Drago se ejecutan en un sistema de computación distribuido compuesto de múltiples procesadores autónomos sin memoria compartida. Los procesadores cooperan mediante el envío de mensajes a través de una red de comunicación. Cada procesador ejecuta su 1

propia secuencia de instrucciones y utiliza sus propios datos locales, ambos almacenados en su memoria local.

Se han considerado las siguientes suposiciones del sistema distribuido:

- Los procesadores son homogéneos[1].
- Los procesadores son de tipo fallo silencioso (*fail-silent*) [PBS+88] —si ocurre un fallo en un procesador, éste simplemente se detiene silenciosamente sin notificar el fallo—.

No hacemos ninguna suposición sobre la topología de la red; por ejemplo, la red puede ser una red de area local o puede ser un conjunto de redes de area local conectadas mediante una red troncal.

A.3 Grupos

Un grupo Drago es un conjunto de agentes (uno o más) que comparten una semántica de aplicación común. Todo grupo es visto desde el exterior como una entidad lógica individual que no permite a sus clientes ver su estructura interna ni las interacciones entre sus miembros. Los grupos se utilizan generalmente para:

1. **Abstraer** las características comunes de sus miembros y los servicios que proporcionan.
2. **Encapsular** el estado interno y esconder las interacciones entre los miembros del grupo así como proporcionar una interfaz uniforme al mundo exterior.
3. Construir objetos mayores (utilizando el grupo como un **bloque básico** de construcción)

La comunicación entre los clientes externos y un grupo servidor se denomina comunicación **intergrupo**, mientras que la comunicación interna entre los miembros del grupo se denomina comunicación **intragrupo**.

Se dice que un grupo es **cerrado** cuando solamente posee comunicación intragrupo (solamente sus miembros pueden comunicarse); en caso contrario se dice que el grupo es **abierto**. El modelo abierto es más general y se corresponde con el modelo cliente-servidor utilizado en los sistemas operativos distribuidos. Este modelo puede extenderse fácilmente a comunicación con grupos, haciendo que el cliente y el servidor sean grupos[2].

Podemos clasificar los grupos en cuatro categorias:

1. **Grupo homogéneo en datos y operaciones:** Cada miembro mantiene una réplica completa de los objetos del grupo e implementa un conjunto idéntico de operaciones sobre dichos objetos.

2. **Grupo homogéneo en operaciones:** El espacio de objetos se divide entre los miembros del grupo. Cada miembro mantiene solamente parte del estado global.

3. **Grupo homogéneo en datos:** Los miembros comparten un conjunto de objetos, pero cada miembro soporta un conjunto de operaciones sobre el mismo objeto.

4. **Grupo Heterogéneo:** Los objetos y operaciones que implementa y mantiene cada miembro pueden ser heterogéneos. Puede o no haber cooperación entre los miembros del grupo, y sus estados internos pueden ser completamente independientes.

A.3.1 Cita remota con un grupo

La cita remota con un grupo (CRG) es una extensión del mecanismo de cita de Ada83 (con algunas restricciones[3]. Es el único mecanismo de sincronización y comunicación remota que proporciona Drago.

La cita remota con un grupo se inicia nombrando un grupo y un punto de entrada declarado en una especificación de grupo (un *punto de entrada remoto*). Todos los miembros del grupo reciben las mismas peticiones de citas remotas en el mismo orden y las aceptan mediante una extensión de la sentencia **accept** de Ada83. La parte que inicia la cita se queda bloqueada hasta que todos los miembros vivos del grupo han aceptado y completado la cita remota.

La cita remota con grupos tiene las siguientes propiedades (una extensión de las propiedades del entorno virtualmente síncrono de Isis [GAAM93])[4]:

- **Ordenación global:** Todos los miembros de un grupo observan las mismas peticiones de citas remotas en el mismo orden.

- **Causalidad:** Una petición de cita remota con grupo grc_2 que es causada por una acción que ocurrió despues de aceptar alguna petición de cita remota anterior grc_1, es observada por todos los miembros después que grc_1.

- **Atomicidad:** La petición de cita remota con grupo es recibida por todos los miembros del grupo o por ninguno. En particular, si ocurren fallos mientras se está realizando una cita remota con un grupo, Drago garantiza que la petición de cita remota será recibida por todos o ninguno de los miembros *supervivientes* del grupo destino (incluso si el peticionario falla).

- **Uniformidad:** La cita remota es uniforme: esto significa que si cualquier miembro del grupo, fallido o no, recibe una petición de cita remota grc_1, entonces todos los miembros no fallidos reciben grc_1[5].

La sincronía virtual permite a los programadores diseñar sus aplicaciones distribuidas como si fuesen a ejecutarse en un entorno completamente síncrono. Sin embargo, la implementación real se ejecuta de forma asíncrona, explotando así el paralelismo inherente de la arquitectura distribuida [BCJ+90].

Los puntos de entrada remotos pueden tener parámetros de entrada (modo **in**) y salida (modo **out**). La tupla [punto de entrada, parámetros de entrada] constituye la petición de cita remota que se envía al grupo cuando se inicia una cita remota con un grupo; al completarse la cita remota, el grupo envía al cliente remoto la tupla [parámetros de salida][6].

A.3.2 Abstracciones de grupo

El modelo de programación de Drago se basa en grupos abiertos. Drago proporciona dos abstracciones de grupo:

- Abstracción de grupo **replicado**, que implementa un grupo homogéneo en datos y operaciones.
- Abstracción de grupo **cooperativo**, que puede implementar las otras tres categorías de grupos descritas anteriormente.

En las siguientes secciones se definen estas abstracciones de grupo en términos de sus interacciones con otros grupos.

Grupo replicado

Un grupo replicado se comporta como un único miembro. Los miembros de un grupo replicado no se comunican entre ellos ni se conocen (cada réplica se comporta como si no hubiesen más réplicas[7]).

Todos los miembros de un grupo replicado deben ser *réplicas* idénticas de un autómata determinista para asegurar *consistencia de réplicas*[8].

Cuando un grupo replicado recibe una petición de cita remota se comporta de la siguiente forma:

1. Radia la cita remota a todos sus miembros (ver *A* en figura A.1).

[1]Sin embargo, no se impide que haya alguna implementación de Drago que soporte la configuración de un programa Drago en un sistema distribuido heterogéneo.

[2]Para un artículo completo sobre grupos y comunicación con grupos ver [LCN90].

[3]No se permite la cita remota temporizada y tampoco puede abortarse una cita remota.

[4]Isis define la sincronía virtual en términos de ordenación de eventos (envío/recepción de mensajes). Los eventos considerados en Drago son los asociados con la cita remota.

[5]Sin uniformidad, puede alcanzarse un estado inconsistente si un miembro del grupo modifica el entorno después de recibir una cita remota y después falla (el resto de los miembros podría no recibir la petición de cita remota). En general, un protocolo uniforme evita la necesidad de considerar las peticiones de citas remotas iniciadas por los miembros antes de fallar.

[6]La implementación de Drago gestiona automáticamente la construcción y envío de los mensajes de llamada/respuesta por la red.

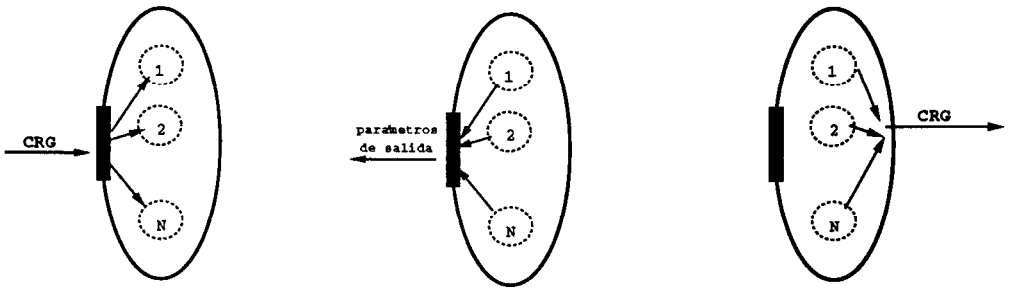
[7]De esta forma el grado de replicación de un grupo replicado puede variarse sin necesidad de recompilación o reenlazado.

[8]La consistencia de réplicas es un requisito fuerte, pero no puede evitarse sin conocimiento de la semántica de los objetos a replicar. En ausencia de conocimiento específico de la aplicación, la consistencia de réplicas es una condición necesaria y suficiente para la transparencia de replicación [Coo85].

2. Cada miembro ejecuta el bloque de sentencias asociado a la sentencia **accept** correspondiente —suponiendo que todas las réplicas aceptan la cita remota—. 6

3. Selecciona el valor de los parámetros de salida de uno cualquiera de los miembros no-fallidos y los envía al cliente remoto (ver *B* en figura A.1). 7

Cuando todas las réplicas inician una cita remota con otro grupo, se elige sólo una de las peticiones de cita remota correspondientes y se envía al grupo servidor[9] (ver *C* en figura A.1). 8



A) La petición de cita remota se difunde a todos los miembros.

B) El valor de los parámetros de salida de una de las réplicas se envía al cliente remoto.

C) Cuando todas las réplicas inician una cita remota solo se envía una.

Figura A.1: Interacciones de un grupo replicado

Un grupo replicado puede tolerar fallos hardware si se replica el agente replicado correspondiente y se ejecuta cada réplica en hardware que tenga independencia de fallo. Puesto que cada réplica que ejecuta un procesador no-fallido comienza en el mismo estado inicial y acepta la misma secuencia de citas remotas en el mismo orden (si el código de usuario que contiene el agente replicado es determinista), todas las réplicas hacen exactamente el mismo trabajo y producen exactamente la misma respuesta. 9

Grupo cooperativo

Un grupo cooperativo es un conjunto de miembros que cooperan para conseguir un objetivo común definido por la semántica de la aplicación. 1

Cuando un grupo cooperativo recibe una petición de cita remota, el grupo se comporta de la siguiente forma: 2

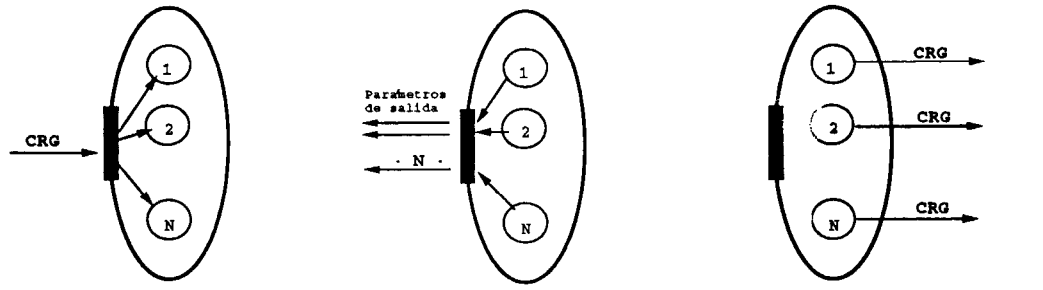
1. Radia la petición de cita remota a todos sus miembros (ver *A* en figura A.2). 3

2. Cada miembro ejecuta de forma independiente el bloque de sentencias asociado a la cita remota (suponiendo que todos los miembros aceptan la cita remota) y envía al cliente remoto el valor de los parámetros de salida correspondientes (ver *B* en figura A.2). 4

Cuando cualquiera de sus miembros inicia una cita remota con un grupo, esta cita remota 5

[9]Se supone que todas las réplicas inician exactamente la misma cita remota, puesto que todas ellas tienen el mismo código determinista.

se realiza de forma totalmente independiente del resto de los miembros (los otros miembros no ven la cita remota ni los parámetros de salida correspondientes —ver *C* en figura A.2—).



A) La petición de cita remota se radia a todos los miembros del grupo.

B) Cada miembro envía al cliente remoto el valor de sus parámetros de salida.

C) Las citas remotas iniciadas por los miembros se realizan de forma independiente.

Figura A.2: Interaciones de un grupo cooperativo

6 Los miembros de un grupo cooperativo pueden comunicarse entre ellos (para cooperar) mediante la **comunicación intragrupo** (ver figura A.3). La comunicación intragrupo cumple todas las propiedades de la cita remota con grupos descritas en el apartado A.3.1.

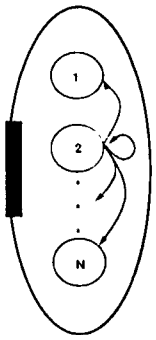


Figura A.3: Comunicación intragrupo en un grupo cooperativo

Interacciones entre grupos

1 Todo grupo Drago puede ser cliente o servidor de otros grupos (de acuerdo con el modelo cliente-servidor). Por lo tanto, pueden existir cuatro tipos de interacciones[10]

CLIENTE	SERVIDOR
Grupo replicado	Grupo replicado
Grupo replicado	Grupo cooperativo
Grupo cooperativo	Grupo replicado
Grupo cooperativo	Grupo cooperativo

[10] Al igual que Isis, Drago no proporciona ningún mecanismo para que un agente interactúe directamente con otro agente. En este caso debe utilizarse un grupo con un único miembro (el receptor).

A continuación analizaremos en detalle los cuatro casos. Supongamos que el grupo cliente tiene N miembros y que el grupo servidor tiene M miembros. Supongamos también que todos los miembros del grupo servidor aceptan la cita remota.

1. **Grupo replicado — grupo replicado:** Cuando todos los miembros de un grupo replicado inician una cita remota con otro grupo replicado, se elige solamente una de las peticiones de cita remota correspondientes y se envía al grupo servidor. Todas las réplicas del grupo servidor aceptan y ejecutan la cita remota, y finalmente los parámetros de salida de uno cualquiera de los miembros de este grupo se envían al grupo cliente. Este valor de los parámetros de salida se radia a todos los miembros del grupo cliente. De esta forma, todas las réplicas del grupo cliente reciben el mismo valor de los parámetros de salida. Ver figura A.4.

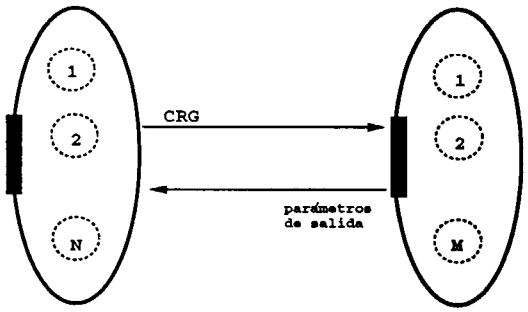


Figura A.4: Grupo replicado — grupo replicado

2. **Grupo replicado — grupo cooperativo:** Cuando todos los miembros de un grupo replicado inician una cita remota, Drago elige solamente una de las peticiones de cita remota y la envía al grupo servidor. Todos los miembros del grupo servidor aceptan la cita remota, la ejecutan y envían al grupo cliente el valor de los parámetros de salida correspondientes. Estos valores se radian a todos los miembros del grupo cliente. De esta forma, todos los miembros del grupo cliente reciben los mismos M valores por cada uno de los parámetros de salida. Ver figura A.5.

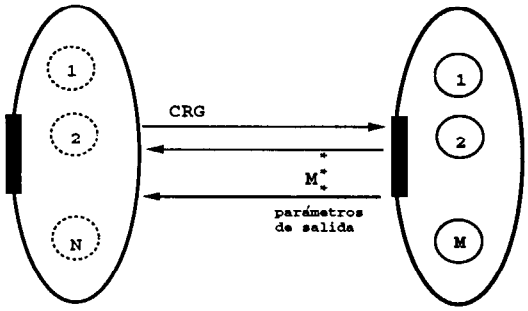


Figura A.5: Grupo replicado – grupo cooperativo

3. **Grupo cooperativo — grupo replicado:** Toda cita remota iniciada por cualquier miembro del grupo cliente se realiza de forma independiente con el grupo servidor. Todas las réplicas del grupo servidor la aceptan, la ejecutan, y finalmente el grupo replicado envía

CLIENTE	*SERVIDOR*	*CRGs enviadas*	*Respuestas/CRG*
Grupo replicado	Grupo replicado	1	1
Grupo replicado	Grupo cooperativo	1	M
Grupo cooperativo	Grupo replicado	N	1
Grupo cooperativo	Grupo cooperativo	N	M

Tabla A.1: Tabla resumen de las interacciones entre grupos.

el valor de los parámetros de salida de una de las réplicas al miembro que inició la cita remota. De esta forma, el miembro del grupo cliente que inicia una cita remota recibe una respuesta. Ver figura A.6.

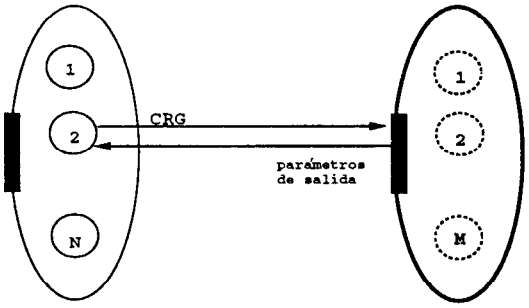


Figura A.6: Grupo cooperativo — grupo replicado

6 4. **Grupo cooperativo — grupo cooperativo:** Toda cita remota iniciada por cualquier miembro del grupo cliente se realiza de forma independiente con el grupo servidor. Todos los miembros del grupo servidor aceptan la cita remota, la ejecutan y envían el valor de sus parámetros de salida al miembro que inició la cita remota. De esta forma, el miembro del grupo cliente que inicia una cita remota recibe *M* respuestas. Ver figura A.7.

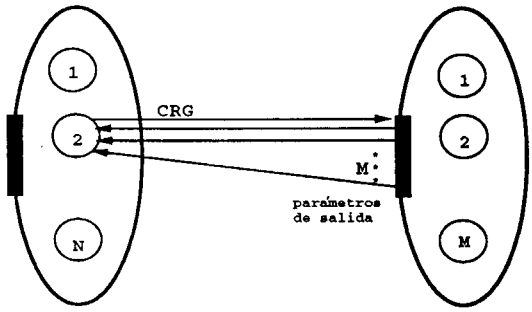


Figura A.7: Grupo cooperativo — grupo cooperativo

A.3.3 Especificación de grupo

Drago proporciona la abstracción de grupo mediante una unidad de compilación denominada especificación de grupo (**group specification**). Las otras unidades de compilación de Drago son los agentes, los paquetes, los genéricos y los subprogramas[11]. 1

Una especificación de grupo permite especificar entes distribuidos que están lógicamente relacionados, y proporciona una interfaz (compartida) a los miembros del grupo (cuando un agente comparte una especificación de grupo G se dice que es miembro del grupo G —ver figura A.8—). 2

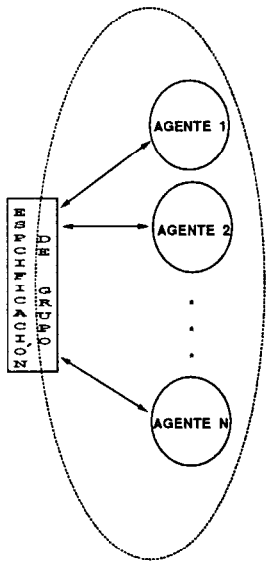


Figura A.8: La especificación de grupo proporciona la abstracción de grupo.

Una especificación de grupo puede contener declaraciones de constantes, tipos (incluyendo los tipos privados y privados limitados de Ada83), excepciones y puntos de entrada remotos[12]. 3

La sintaxis de una especificación de grupo se basa en la sintaxis de los paquetes Ada83. 4

```
group_specification ::=
    [ replicated ] group specification group_simple_name is
        intergroup_basic_declarative_item
    [ intragroup
        intragroup_basic_declarative_item ]
    [ private
        private_basic_declarative_item]
    end [group_simple_name];
```

Los grupos replicados se indican mediante la palabra reservada **replicated** en la cabecera 5

[11] Los paquetes, genéricos y subprogramas son unidades de compilación heredadas directamente de Ada83.

[12] El bloque de sentencias asociado a cada punto de entrada remoto se implementa dentro de los agentes (ver

de la especificación del grupo correspondiente. Los grupos cooperativos no requieren ninguna palabra reservada específica porque son la especificación de grupo por defecto.

6 Si *group_simple_name* aparece al final de una especificación de grupo, debe coincidir con el identificador del grupo.

7 La primera lista de declaraciones de una especificación de grupo se denomina *sección intergrupo* y contiene toda la información que pueden conocer los clientes remotos acerca de este grupo.

8 La lista opcional de declaraciones después de la palabra reservada **intragroup** se denomina *sección intragrupo* y contiene información que solamente pueden conocer los miembros del grupo. Esta sección solamente existe en las especificaciones de grupos cooperativos[13].

9 La lista opcional de declaraciones después de la palabra reservada **private** se denomina **sección privada** del grupo y proporciona a los grupos la misma funcionalidad que la parte privada de los paquetes Ada83 (ver apartado 7.4 en [ANS83]).

10 Todas las partes de una especificación de grupo pueden contener declaraciones de constantes, tipos y excepciones. Los puntos de entrada remotos solamente pueden declararse en las secciones intergrupo e intragrupo. No se permiten declaraciones de variables y de tareas en una especificación de grupo.

11 Todo agente que pertenece a un grupo tiene visible todas las declaraciones contenidas en la especificación de grupo correspondiente y debe atender todos los puntos de entrada declarados en dicha especificación.

A.3.4 Notificación de fallo

1 Los grupos cooperativos pueden especificar en la sección intragrupo un punto de entrada remoto especial que es llamado automáticamente por el entorno de tiempo de ejecución de Drago en todos los miembros vivos del grupo cuando detecta el fallo de un miembro del grupo. Este mecanismo se denomina *notificación de fallo* y tiene la siguiente sintaxis:

```
failure_entry ::=
    failure entry identifier [(identifier: in Member_Identifier)]
```

2 Un *punto de entrada de fallo* puede opcionalmente contener un parámetro de entrada del tipo privado **Member_Identifier**[14]. Este parámetro de entrada contiene el identificador del miembro que ha fallado.

---

siguiente capítulo).

[13]Las citas remotas con los puntos de entrada declarados en la sección intragrupo se realizan exactamente igual que las citas remotas con puntos de entrada declarados en la sección intergrupo. Los grupos replicados no tienen esta sección porque se supone que son réplicas de un autómata determinista y que por ello no necesitan pasarse información de estado (todas las réplicas tienen exactamente el mismo estado).

[14]El entorno de tiempo de ejecución de Drago utiliza el tipo privado **Member_Identifier** para identificar los miembros que pertenecen a cada grupo.

El mecanismo de notificación de fallos de Drago cumple las propiedades de la cita remota con grupos de Drago (descritas en el apartado A.3.1)[15]. En caso de múltiples fallos, Drago serializa las notificaciones[16]. 3

La notificación de fallos de Drago puede utilizarse, por ejemplo, para redistribuir la carga de trabajo cuando muere un miembro de un grupo cooperativo[17]. 4

A.4 Agentes

La unidad de distribución de Drago es el *agente*. Un agente es un tipo especial de objeto abstracto (tiene estado interno que no es accesible directamente desde el exterior del agente). La comunicación y sincronización entre los agentes se consigue mediante la *cita remota con un grupo* (descrita en el apartado A.3.1). Un agente puede tener también operaciones locales (procedimientos y funciones) y un código de inicialización (ver figura A.9). 1

La comunicación remota entre los agentes (mediante la *cita remota con un grupo*) se implementa mediante un mecanismo de comunicación basado en intercambio de mensajes. Los argumentos se pasan por valor[18]. La implementación de Drago gestiona todos los detalles de construcción y envío de los mensajes. 2

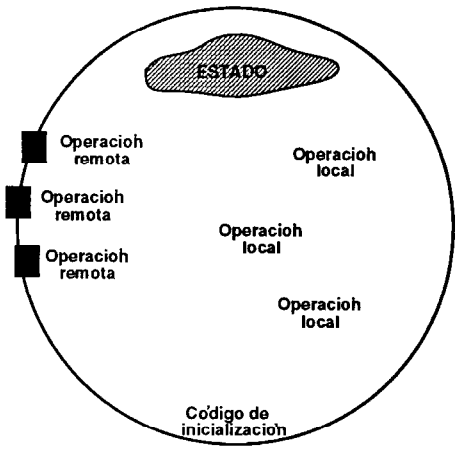


Figura A.9: El agente

Cada agente reside en un nodo de la red, aunque varios agentes pueden residir en el mismo nodo. Un programa distribuido Drago consta de varios agentes que residen en varios nodos de la red. 3

[15]Para un análisis detallado de las ventajas de un aviso consistente de fallos en sistemas de comunicación fiable ver el artículo [BG93].

[16]Si k miembros fallan simultáneamente, el entorno de tiempo de ejecución de Drago realiza k notificaciones de fallo en los miembros vivos.

[17]Los grupos replicados no tienen la comunicación intragrupo ni la notificación de fallo porque se supone que todos los miembros de un grupo replicado son réplicas de un autómata determinista y por ello no necesitan pasarse información de estado ni redistribuir su carga de trabajo (todos los miembros realizan el mismo trabajo independientemente del número de miembros vivos).

[18]Asegurando así que el estado interno del agente no puede ser accedido directamente por otros agentes

4 Drago proporciona dos tipos de agentes: **agentes replicados y agentes cooperativos**. La principal diferencia entre ambos es que los agentes replicados no pueden tener tareas internas[19], mientras que los agentes cooperativos sí.

A.4.1 Sintaxis de los agentes

1 La sintaxis de los agentes se basa en la sintaxis del cuerpo de las tareas de Ada83:

```
agent_unit ::=
    [ replicated ] agent agent_simple_name is
        [declarative_part]
    [begin
        sequence_of_statements
    [ exception
        exception_handler
        {exception_handler}]]
    end [agent_simple_name];
```

2 La palabra reservada **replicated** en la cabecera de un agente especifica que es un agente replicado. Si *agent_*simple_name aparece al final del agente, debe coincidir con el especificado en la cabecera del agente.

3 En la elaboración de un agente, se elabora primero su parte declarativa y a continuación se ejecuta su secuencia de sentencias (si la hay[20]). El manejador de excepciones opcional al final del agente trata las excepciones elevadas durante la ejecución de la secuencia de sentencias del agente.

4 De forma similar a las tareas Ada83, decimos que un agente ha *completado* su ejecución cuando ha finalizado la ejecución de su secuencia de sentencias. agente no tiene tareas que dependan de él, su *terminación* ocurre cuando ha completado su ejecución. Tras su terminación decimos que el agente ha *terminado* su ejecución. Si un agente tiene tareas que dependen de él, su terminación ocurre cuando su secuencia de sentencias se ha completado y todas las tareas que dependen de él han terminado su ejecución.

A.4.2 Cláusulas de contexto de grupo

1 Las cláusulas de contexto de grupo se utilizan para especificar los grupos con los que está relacionado un agente. Un agente puede ser *miembro* y/o *cliente* de varios grupos. Para ello Drago proporciona dos cláusulas de contexto de grupo: cláusula **with group** y cláusula **for group**.

[19] Los agentes replicados deben implementar autómatas deterministas.

[20] Los agentes replicados deben tener siempre una secuencia de sentencias porque sólo tienen un flujo de control. Los agentes cooperativos pueden estar compuestos solamente de tareas internas que realizan citas remotas con grupos.

with_group_clause ::=
with group *group*_simple_name { , *group*_simple_name } ;

for_group_clause ::=
for group *group*_simple_name { , *group*_simple_name } ;

Los nombres que aparecen en una cláusula de contexto de grupo deben ser nombres simples de *especificaciones de grupo*. 2

Cuando un agente especifica una cláusula **with group**, se convierte en *cliente* de todos los grupos especificados y tiene visibles todas las declaraciones contenidas en la sección intergrupo de las especificaciones de grupo correspondientes. 3

Cuando un agente especifica una cláusula **for group**, se convierte en *miembro* de todos los grupos especificados. Un *miembro* de un grupo tiene visibles todas las declaraciones contenidas en la especificación de grupo correspondiente y debe proporcionar todos los servicios remotos especificados en ella (mediante puntos de entrada remotos). Los agentes replicados solamente pueden ser miembros de grupos replicados. Los agentes cooperativos solamente pueden ser miembros de grupos cooperativos. 4

A.4.3 Colas asociadas a los puntos de entrada remotos

Todo agente que es miembro de un grupo tiene internamente dos niveles de colas asociadas a los puntos de entrada remotos (ver figura A.10): 1

- **Nivel de peticiones de cita remota pendientes**: Este nivel mantiene una única cola con todas las peticiones de cita remota que cumplen las propiedades de la cita remota de Drago (descritas en el apartado A.3.1). 2

- **Nivel de peticiones de cita remota encoladas**: Este nivel mantiene una cola de peticiones de cita remota asociada a cada punto de entrada remoto del agente (puntos de entrada declarados en una especificación de grupo). 3

En los apartados A.4.4 y A.4.5 se describe cómo se pasan los mensajes del nivel de peticiones de cita remota pendientes al nivel de peticiones de cita remota encoladas. 4

El atributo **Count** de Ada83 se ha extendido para proporcionar su funcionalidad a los puntos de entrada remotos de Drago. Cuando se aplica a un punto de entrada remoto devuelve un valor de tipo *universal_integer* con el número de peticiones de cita remota encoladas en el punto de entrada especificado (en el nivel de peticiones de cita remota encoladas). **Count** solamente puede utilizarse dentro del bloque de sentencias de un agente. 5

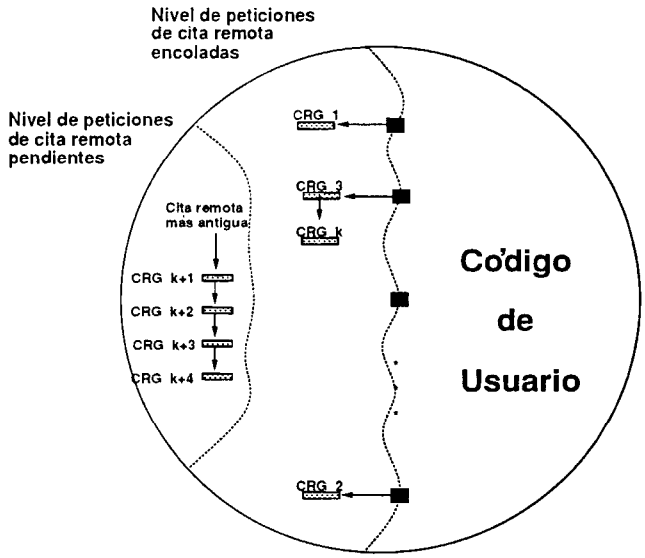


Figura A.10: Colas de citas remotas asociadas a cada agente

A.4.4 Sentencia *accept*

1 El agente puede aceptar una cita remota mediante una sentencia **accept** (una extensión de la sentencia **accept** de Ada83). Su sintaxis es la siguiente:

```
agent_accept_statement ::=
    accept [group_simple_name.] entry_simple_name [formal_part] [ do
        sequence_of_statements
    [ exception
        exception_handler
        {,exception_handler} ]
    end [entry_simple_name]];
```

2 La parte formal de una sentencia **accept** debe ser conformante con la parte formal de la declaración del punto de entrada correspondiente (en la especificación de grupo). Si *entry_simple_name* aparece al final de la sentencia **accept**, debe coincidir con el especificado al principio del **accept**.

3 Las acciones a realizar cuando se acepta una cita remota se especifican mediante la secuencia de sentencias. Dicha secuencia de sentencias puede llamar a subprogramas que realicen citas remotas. Una sentencia **accept** puede no tener asociada ninguna secuencia de sentencias, incluso si el punto de entrada correspondiente no tiene parámetros. La secuencia de sentencias de una sentencia **accept** puede contener sentencias **return**. Si el agente no quiere enviar ningún valor en los parámetros de salida al cliente remoto, puede utilizar una sentencia **return null**.

4 Si se eleva una excepción en la secuencia de sentencias de una sentencia **accept** y la excepción no es tratada localmente, al completar la cita remota la excepción se propaga

simultáneamente al llamador remoto (ver apartado A.4.10).

Una sentencia **accept** asociada a un punto de entrada remoto solamente está permitida dentro de la secuencia de sentencias de un agente. 5

Semantica de ejecución de la sentencia *accept*

Cuando el agente ejecuta una sentencia **accept** y tiene varias peticiones de cita remota encoladas en la cola asociada al punto de entrada especificado (en el nivel de peticiones de cita remota encoladas), Drago acepta la petición de cita remota encolada más antigua. 6

Cuando el agente ejecuta una sentencia **accept** y no tiene ninguna petición de cita remota encolada en la cola asociada al punto de entrada especificado (en el nivel de peticiones de cita remota encoladas), Drago pasa la petición de cita remota más antigua del nivel de peticiones de cita remota pendientes a la cola correspondiente del nivel de peticiones de cita remota encoladas. Este proceso se repite hasta que el agente tenga una petición de cita remota encolada en la cola asociada al punto de entrada especificado y finalmente acepte dicha cita remota. 7

A.4.5 Sentencia *select*

La sentencia **select** permite al agente la espera selectiva de una o más alternativas. De forma similar a la sentencia **select** de las tareas Ada83, la elección puede depender de condiciones asociadas a cada alternativa de la espera selectiva. 1

La sintaxis de la espera selectiva de Drago se basa en la sintaxis de la espera selectiva de las tareas de Ada83. Las principales diferencias entre la espera selectiva de citas remotas y la espera selectiva de citas entre tareas son que la espera selectiva de citas remotas es *determinista* y que no existen aquí la alternativa *terminate* ni la alternativa *delay*. 2

```
agent_selective_wait ::=
    select
        agent_select_alternative
    or
        agent_select_alternative
    end select;

agent_select_alternative ::=
    [ when condition => ]
        agent_accept_statement [sequence_of_statements]
```

Decimos que una alternativa de una sentencia **select** está *abierta* si no comienza con **when** y una condición, o si el resultado de evaluar la condición es el valor lógico *True*. En caso contrario decimos que está *cerrada*. 3

La condición de una alternativa de una sentencia **select** no puede referenciar ningún paráme- 4

tro formal del punto de entrada, pero puede referenciar cualquier otra cosa visible en ese punto[21].

Semántica de ejecución de la sentencia *select*

5 La sentencia **select** evalúa las condiciones asociadas a las alternativas del **select** que tienen alguna petición de cita remota en el nivel de peticiones de cita remota encoladas.

6 Si hay varias alternativas abiertas, el agente acepta la petición de cita remota encolada *más antigua* y ejecuta el bloque de sentencias asociado a dicha alternativa (así como la posible secuencia de sentencias posterior, si la hay).

7 Si no hay ninguna petición de cita remota encolada en alguna de las alternativas abiertas, Drago pasa la petición de cita remota *más antigua* del nivel de peticiones de cita remota pendientes a la cola correspondiente del nivel de peticiones de cita remota encoladas (incrementando el valor del atributo **Count** correspondiente), y a continuación reevalúa las condiciones[22]. El agente repite este proceso hasta que tiene alguna petición de cita remota encolada en algunas de las alternativas abiertas. En caso de que tenga peticiones de cita encoladas en varias alternativas abiertas, elige la más antigua. A continuación el agente acepta la cita remota y ejecuta la sentencia de instrucciones asociada y la posible secuencia de sentencias posterior —si la hay—.

8 Si todas las alternativas de la sentencia **select** están cerradas, se eleva la excepción **Program_Error**.

A.4.6 Sentencia *requeue*

1 Los agentes permiten reencolar una cita remota aceptada redirigiendo la cita remota a otro punto de entrada. La sintaxis de la sentencia de reencolamiento de Drago está basada en la sintaxis de la sentencia de reencolamiento de Ada9X ([mt93], apartado 9.5.4)[23].

```
requeue_statement ::=
   requeue [ group_simple_name.] entry_simple_name [(Formal_Part)];
```

2 La sentencia *requeue* de Drago tiene las siguientes características:

3 • La sentencia **requeue** solamente está permitida dentro de una sentencia de aceptación de cita remota (**accept**) y está asociada al **accept** más interno[24].

[21] Esta restricción asegura que todas las citas remotas en la misma cola ven el mismo valor asociado a la condición.

[22] Este comportamiento determinista asegura que todas las réplicas de un grupo replicado actualizan de forma consistente el valor de los atributos **Count** asociados a los puntos de entrada remotos.

[23] Sin embargo existen algunas diferencias: la sentencia **requeue** de Drago no posee la sección *with abort* de Ada9X; la sentencia *requeue* de Ada9X no permite pasar parámetros de entrada adicionales al punto de entrada del reencolamiento.

[24] De forma similar a Ada9X, no se permite utilizar la sentencia de reencolamiento dentro del cuerpo de una unidad que esté anidada dentro del **accept**.

- El punto de entrada nombrado por la sentencia **requeue** debe tener un perfil de parámetros conformante con el perfil de la sentencia **accept** más interna, aunque puede contener parámetros de entrada adicionales[25]. 4
- La cita remota reencolada simplemente se añade al final de la cola de peticiones de cita remota encoladas del punto de entrada nombrado, para ser seleccionada posteriormente. 5
- Una vez realizado el reencolamiento se completa la ejecución de la sentencia **accept** más interna. 6
- Los parámetros formales de un reencolamiento se gestionan de la siguiente forma: 7
  - Si el nuevo punto de entrada tiene parámetros formales, entonces los valores de estos parámetros se inicializan con los valores de los parámetros reales de la sentencia **accept** más interna. Si la sentencia de reencolamiento especifica parámetros de entrada adicionales, los valores de estos parámetros se evalúan en el punto del reencolamiento. 8
  - Si el nuevo punto de entrada no tiene parámetros formales, entonces los valores de los parámetros formales de modo **in out** o modo **out** de la sentencia **accept** más interna se almacenan para enviarlos posteriormente como parámetros de salida al completar el nuevo punto de entrada. 8

A.4.7 Cita remota con un grupo

La *cita remota con un grupo* permite a los agentes realizar citas remotas con todos los agentes que son miembros del grupo (cumpliendo las propiedades descritas en el apartado A.3.1)[26]. La sintaxis de una cita remota con un grupo es la siguiente: 1

```
group_remote_call_statement ::=
    group_name.entry_name [actual_parameter_part]
```

La cita remota con un grupo debe nombrar un grupo y un punto de entrada remoto declarado en la especificación de grupo correspondiente. Al realizar una cita remota con un grupo el entorno de tiempo de ejecución de Drago crea un mensaje de petición de cita remota[27] y la envía al grupo remoto. 3

Si se intenta realizar una cita remota con un grupo que no tiene ningún miembro vivo se eleva la excepción **Group_Error** en el punto de la llamada (ver apartado A.4.10). 4

Drago no permite realizar citas remotas condicionales ni citas remotas temporizadas. 5

[25] Estos parámetros adicionales pueden utilizarse para pasar información de contexto adicional al punto de entrada de reencolamiento.

[26] Sun proporciona una llamada remota a procedimiento que permite realizar una llamada remota con radiado [Cor91], pero su semántica (al menos una vez) es más debil que la semántica de la cita remota de Drago.

[27] Este mensaje de petición de cita remota contiene el valor de los parámetros de entrada que deben enviarse al grupo remoto.

Cita remota con un grupo replicado

7 Cuando se realiza una cita remota con un grupo replicado, el llamador se queda bloqueado hasta que el grupo replicado ha completado la cita remota (y ha enviado los parámetros de salida, si los hay).

Cita remota con un grupo cooperativo

8 Cuando se realiza una cita remota con un grupo cooperativo, el llamador se queda bloqueado hasta que todos los miembros vivos del grupo han completado la cita remota.

9 Dependiendo de la semántica del servicio que ofrece el grupo cooperativo, puede ser necesario que el llamador reciba los parámetros de salida de todos los miembros del grupo o no. Drago soporta ambas alternativas:

10 • Si la especificación del servicio remoto es tal que los clientes remotos solamente necesitan recibir el valor de los parámetros de salida de uno de los miembros vivos del grupo cooperativo, el punto de entrada remoto asociado al servicio remoto debe especificarse como si el servicio lo proporcionase solamente un miembro del grupo cooperativo.

```
Ejemplo:    group specification Servicio_1 is
               entry Hacer(Parametro: in  Integer;
                           Resultado: out Integer);
            end Servicio_1;
```

11 La semántica de la cita remota correspondiente es la siguiente: cuando todos los miembros del grupo cooperativo han completado la cita remota, se entrega al llamador el valor de los parámetros de salida de uno de los miembros que no haya completado la cita con una sentencia **return null**. Si todos los miembros han completado la cita con la sentencia **return null**, se eleva la excepción **Group_Null_Answer** en el punto de la llamada.

12 • Si la especificación del servicio remoto es tal que los clientes remotos deben recibir el valor de los parámetros de salida de varios miembros del grupo cooperativo, todos los parámetros de salida de este servicio remoto deben declararse como formaciones irrestringidas unidimensionales[28].

```
Ejemplo:    group Servicio_2 is
               type Tipo_Respuesta is
                    array(Positive range <>) of Integer;
               entry Hacer(Parametro: in  Integer;
                           Resultado: out Tipo_Respuesta);
            end Servicio_2;
```

13 Las siguientes reglas se aplican a las variables asociadas a los parámetros formales de tipo formación irrestringida unidimensional en el instante de iniciar la cita remota:

14 – Su dimensión especifica el número de respuestas que desea recibir el llamador (no es necesario ningún parámetro adicional).

[28] Los grupos replicados no pueden especificar parámetros de salida de tipo formación irrestringida.

```
Ejemplo:    with group Servicio_2;
            Agent Cliente_del_servicio_2;
               Respuestas:Servicio_2.Tipo_Respuesta(1..3);
               -- Quiero recibir tres respuestas.
            begin
               Servicio_2.Hacer(a => 12, b => Respuestas);
            end Cliente_del_servicio_2;
```

- Si el número de respuestas especificadas es mayor que el número de miembros vivos del grupo, se eleva la excepción **Group_Error** en el punto de la llamada. 15
- Todas las variables de tipo formación irrestringida asociadas a los parámetros de salida deben tener la misma dimensión. 16

La semántica de la cita remota correspondiente es la siguiente: el llamador permanece bloqueado hasta que todos los miembros del grupo cooperativo han completado la cita remota. En este instante se elige arbitrariamente el número de miembros especificado (que no haya completado la cita mediante la sentencia **return null**) y se entrega al llamador el valor de todos los parámetros de salida correspondientes[29]. Si el número de miembros que ha completado la cita remota con la sentencia **return null** impide proporcionar al cliente el número de respuestas solicitada, se eleva la excepción **Group_Null_Answers** en el punto de la llamada. 17

En tiempo de ejecución Drago proporciona los atributos **Num_Members** y **Range** (aplicables solamente a identificadores de grupo), que pueden ser utilizados por los clientes de grupos cooperativos para declarar las variables donde almacenan las respuestas de todos los miembros vivos (ver apartado A.5). 18

A.4.8 Cita remota con reencolamiento

Cuando un agente replicado realiza una cita remota con un grupo, se queda bloqueado hasta que todos los miembros vivos han completado la cita remota. Esto supone una gran pérdida de paralelismo cuando un grupo replicado acepta una cita remota y para completarla necesita realizar una cita remota con otro grupo (el grupo replicado es simultáneamente servidor y cliente). Drago proporciona una cita remota con reencolamiento que permite al agente replicado aceptar citas remotas posteriores mientras espera a que se complete su propia cita remota en curso. 1

requeued_remote_call ::=
group_remote_call_statement **requeue in** entry_call_statement

La semántica de ejecución de la cita remota con reencolamiento es la siguiente: cuando todos los miembros vivos del grupo invocado tienen la petición de cita remota almacenada en el nivel de citas remotas pendientes, el llamador almacena esta cita remota como *cita remota* 2

[29] El entorno de tiempo de ejecución de Drago descarta automáticamente el resto de las respuestas.

pendiente y finaliza la sentencia **accept** más interna (pudiendo así atender otras peticiones). Cuando finalmente se completa la cita remota pendiente, se reencola en el punto de entrada especificado.

3 La cita remota con reencolamiento solamente está permitida dentro de una sentencia **accept** y está asociada a la sentencia **accept** más interna.

4 Si la cita remota con reencolamiento recibe una excepción, la excepción se eleva en el punto de entrada donde se realizó el reencolamiento (cuando sea posteriormente aceptada).

A.4.9 Puntos de entrada locales

1 Los agentes pueden especificar puntos de entrada locales que no son visibles al resto de los agentes. Los puntos de entrada locales deben declararse en la sección de declaración del agente.

2 Los puntos de entrada locales proporcionan colas locales que pueden utilizarse para el reencolamiento (en la sentencia de reencolamiento y en la sentencia de cita remota con reencolamiento).

3 Las tareas locales de los agentes cooperativos pueden realizar llamadas a los puntos de entrada locales. La llamada simplemente se encola en la cola correspondiente del nivel de peticiones encoladas.

4 El atributo **Count** de Ada83 ha sido extendido para proporcionar también su funcionalidad a los puntos de entrada locales (ver apartado A.5).

A.4.10 Excepciones remotas

1 La especificación de grupo permite al programador declarar excepciones remotas (excepciones que pueden elevarse remotamente).

2 Si se eleva una excepción durante la ejecución del bloque de sentencias de un **accept** y la excepción no es tratada antes de finalizar la ejecución del **accept**, la excepción se propaga simultáneamente al llamador.

3 Si se realiza una cita remota con un grupo que no tiene miembros vivos, se eleva la excepción predefinida **Group_Error** en el punto de la llamada.

4 Las excepciones remotas se tratan siguiendo la forma usual de Ada83 (mediante un manejador de excepciones). Por ello, si una excepción remota no es visible al llamador, puede tratarla mediante la alternativa **others**.

Excepciones remotas de grupos replicados

5 Si falla la última réplica viva de un grupo replicado, se eleva la excepción **Group_Error** en el

punto de la llamada[30].

Si los miembros de un grupo replicado propagan una excepción E, cuando se completa la cita remota se eleva la excepción E en el punto de la llamada[31]. 6

Excepciones remotas de grupos cooperativos

Si algún miembro de un grupo cooperativo falla antes de completar la cita remota, cuando el resto de los miembros vivos del grupo haya completado la cita remota se elevará la excepción **Group_Error** en el punto de la llamada. 7

Si un miembro de un grupo cooperativo propaga una excepción remota E, cuando el resto de los miembros vivos haya completado la cita remota se eleva la excepción E en el punto de la llamada[32]. 8

Si varios miembros de un grupo cooperativo propagan diferentes excepciones remotas $(E1, E2, ..., En)$, cuando el resto de los miembros vivos haya completado la cita remota se eleva la excepción **Several_Exceptions** en el punto de la llamada. 9

Si el número de respuestas especificadas por un cliente de un grupo cooperativo es mayor que el número de miembros vivos del grupo, se eleva la excepción *Group_Error* en el punto de la llamada. 10

Si el número de respuestas especificadas por un cliente de un grupo cooperativo no puede proporcionarse porque miembros del grupo cooperativo completaron la cita con una sentencia **return null**, se eleva la excepción **Group_Null_Answer** en el punto de la llamada. 11

A.5 Atributos

Además de los atributos propios de Ada83, Drago proporciona tres atributos adicionales: **range**, **count** y **member_identifier**. Los dos primeros son una extensión de los atributos correspondientes de Ada83.

A.5.1 Range

Al aplicarlo a un identificador de grupo G, proporciona el rango 1..*Número de miembros vivos en G*.

[30] Cuando falla una réplica que no es la última réplica viva no se eleva la excepción **Group_Error** porque todavía quedan réplicas que proporcionan el servicio.

[31] Se supone que todas las réplicas implementa el mismo código determinista y por ello elevan la misma excepción.

[32] De esta forma, cuando finalmente se eleva la excepción, el llamador sabe que todos los miembros vivos que pudieron completar la cita remota correctamente, lo hicieron.

A.5.2 Count

Al aplicarlo a un identificador de un punto de entrada remoto o local devuelve un valor de tipo *universal_integer* que informa sobre el número de peticiones de cita remota actualmente encoladas en el punto de entrada especificado (en el nivel de peticiones de cita remota encoladas del agente).

A.5.3 Member_Identifier

Al aplicarlo a un identificador de grupo G devuelve un valor del tipo privado de Drago *Member_Identifier*, con el identificador del agente en el grupo G.

A.6 Resumen

1 El modelo de programación de Drago se basa en el paradigma de programación con grupos. Un grupo Drago es un conjunto de agentes que comparten una semántica de aplicación común. Cada grupo es visto desde el exterior como una entidad individual que no permite a los clientes remotos ver su estructura interna ni las interacciones entre sus miembros.

2 Drago proporciona dos abstracciones de grupo: abstracción de **grupo replicado** (un grupo de réplicas deterministas) y abstracción de **grupo cooperativo** (un conjunto de miembros que cooperan para conseguir un objetivo común). Drago proporciona estas abstracciones mediante la **especificación de grupo**.

3 Una especificación de grupo contiene declaraciones de constantes, tipos, excepciones y puntos de entrada remotos. De esta forma, una especificación de grupo proporciona una interfaz que es compartida por todos los miembros del grupo (cuando un agente comparte una especificación de grupo decimos que es un *miembro* del grupo). Todo miembro de un grupo tiene visible todas las declaraciones contenidas en la especificación de dicho grupo y debe proporcionar todos los servicios remotos especificados en ella (mediante **puntos de entrada remotos**).

4 La *cita remota con un grupo* es el único mecanismo de comunicación remota y sincronización remota que proporciona Drago. Es una extensión al mecanismo de cita de Ada83 que debe ser aceptada por todos los miembros vivos del grupo invocado. Drago permite que los miembros de un grupo cooperativo se comuniquen de forma transparente a los clientes remotos (mediante la *comunicación intragrupo*) y reciban notificaciones de fallo de los miembros del grupo que fallen.

5 El *agente* es la unidad de distribución de Drago. Un agente es un tipo de objeto abstracto: tiene estado interno (no accesible directamente desde el exterior) y operaciones remotas especiales (especificadas mediante los puntos de entrada remotos) que pueden ser llamadas remotamente desde otros agentes mediante *citas remotas con grupos*. El agente puede tener operaciones locales (procedimientos, funciones y puntos de entrada locales) y código de inicialización.

Cada agente reside en un nodo de la red, aunque varios agentes pueden residir en el mismo nodo. Un programa distribuido Drago consta de varios agentes que residen en varios nodos de la red. 6

Drago proporciona dos tipos de agentes: *agentes replicados* y *agentes cooperativos*. La principal diferencia entre ambos es que los agentes replicados no pueden tener tareas internas, mientras que los agentes cooperativos sí. Los agentes replicados solamente pueden ser miembros de grupos replicados. Los agentes cooperativos solamente pueden ser miembros de grupos cooperativos. 7

Los agentes pueden tener puntos de entrada locales y remotos; pueden realizar citas remotas con grupos, así como aceptar y/o reencolar citas remotas. 8

Bibliografía

[ACD86] S. Ahuja, N. Carriero, and Gelernter D. Linda and friends. *IEEE Computer*, 19(8):26–34, Agosto 1986.

[AG] S. Arevalo and N.H. Gehani. Distributed programming examples using Fault Tolerant concurrent C. Technical report, AT&T Bell Laboratories.

[AMN88] C. Atkinson, T. Moreton, and A. Natali. *Ada for distributed systems*. Cambridge University Press, 1988.

[And81] G.R. Andrews. Synchronizing Resources. *ACM Transactions on Programming Languages and Systems*, 3(4):405–430, Octubre 1981.

[And91] G.R. Andrews. Paradigms for process interaction in distributed programs. *ACM Computing Surveys*, 23(1), Marzo 1991.

[ANS83] ANSI. *Reference Manual for the Ada Programming Language*. MIL-STD 1815A. 1983.

[AS83] G.R. Andrews and F.B. Schneider. Concepts and notations for concurrent programming. *ACM Computing Surveys*, 15(1):3–43, Marzo 1983.

[Avi85] A. Avizienis. The N-version approach to fault tolerant software. *IEEE Transactions on Software Engineering*, pages 1491–1501, Diciembre 1985.

[BA82] M. Ben-Ari. *Principles of concurrent programming*. Prentice-Hall, Englewood Cliffs, NJ., 1982.

[BA90] M. Ben-Ari. *Principles of concurrent and distributed programming*. Prentice-Hall, Englewood Cliffs, NJ., 1990.

[Bal90] H.E. Bal. *Programming distributed systems*. Silicon Press, 1990.

[Bal91] H.E. Bal. A comparative study of five parallel programming languages. *EurOpen Spring 1991 Conference on Open Distributed Systems*, pages 209–228, Mayo 1991.

[Bar87] J. Barnes. *Programación en Ada*. Diaz de Santos, 1987.

[Bar93] J. Barnes. Highlights of Ada 9x. Technical report, Enero 1993.

[BCJ+90] K. Birman, R. Cooper, T. Joseph, K. Marzullo, M. Makpangou, K. Kane, F. Schmuck, and M. Wood. The Isis system manual. version 2.1. Septiembre 1990.

[BG93] K. Birman and B. Glade. Consistent failure reporting in reliable communication systems. Technical Report TR93-1349, Cornell University. Departament of Computer Science, Mayo 1993.

[Bir93a] K. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):36–53, Diciembre 1993.

[Bir93b] K. Birman. A response to Cheriton and Skeen's criticism of causal and totally ordered communication. Technical report, Cornell University. Departament of Computer Science, Octubre 1993.

[BJ87] K.P. Birman and T.A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computing Systems*, 5(1):47–76, Febrero 1987.

[BKT92a] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, Marzo 1992.

[BKT92b] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. Replication techniques for speeding up parallel applications on distributed systems. *Concurrency—Practice and Experience*, 4(5):337–355, Agosto 1992.

[BMST92] N. Budhiraja, K. Marzullo, F.B. Schneider, and S. Toueg. Primary-backup protocols: lower bounds and optimal implementations. Technical Report TR-92-1265, Departament of Computer Science. Cornell University, Ithaca NY 14853, USA, Enero 1992.

[BN84] A.D. Birrell and B.J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computing Systems*, 2(1):39–59, Febrero 1984.

[BS94] O. Babaglou and A. Schiper. On group communication in large-scale distributed systems. Technical Report UBLCS-94-19, Laboratory of Computer Science. University of Bologna, Italy, Julio 1994.

[BST89] H.E. Bal, J.G. Steiner, and A.S. Tanenbaun. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3), Septiembre 1989.

[BT88] H.E. Bal and A.S. Tanenbaun. Distributed programming with shared data. *Proceedings of the IEEE CS 1988 International Conference on Computer Languages*, pages 82–91, Octubre 1988.

[CCH+88] C.D. Callahan, K.D. Cooper, R.T. Hood, K. Kenedy, and L. Torczon. ParaScope: A parallel programming environment. *International Journal of Supercomputer Applications*, 2(4):84–99, 1988.

[CEB92] J. Chen, A.K. Elmagarmid, and O. Bukhres. The InterBase parallel language: supporting distributed transaction applications. Technical report, Departament of Computer Science. Purdue University, West Lafayette, IN 47907, 1992.

[CG86] K.L Clark and S. Gregory. PARLOG: Parallel programming in logic. *ACM Transactions on Programming Languages and Systems*, 8(1):1–49, Enero 1986.

[CGR] R.F. Cmelik, N.H. Gehani, and W.D. Roome. Fault tolerant concurrent C: A tool for writing fault-tolerant distributed programs. Technical report, AT&T Bell Laboratories.

[Chr91] F. Christian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):57–78, Febrero 1991.

[Coo80] R.P. Cook. *mod – a language for distributed programming. *IEEE Transaction on software engineering*, 6(6):563–571, Noviembre 1980.

[Coo85] E.C Cooper. Replicated distributed programs. *Operating Systems Review*, 19(5):63–78, 1985.

[Cor91] J.R. Corbin. *The art of Distributed Applications: Programming Techniques for Remote Procedure Calls*. Springer-Verlag, 1991.

[CS91] R.S. Chin and Chanson. S.T. Distributed object-based programming systems. *ACM Computing Surveys*, 23(1), Marzo 1991.

[CZ85] D. Cheriton and Zwaenepoel. Distributed process groups in the V-kernel. *ACM Transactions on Computing Systems*, 3(2), Mayo 1985.

[D.85] Gelernter D. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, Enero 1985.

[Dij68] E.W Dijkstra. *Co-operating sequential processes.* Programming Languages. Academic Press, New York, 1968.

[Dij71] E.W Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1:115–138, 1971.

[Dol93] D. Dolev, D. Malki. On distributed algorithms in a broadcast domain. *20th International conference on automata, languages and programming (ICALP)*, Julio 1993.

[GAAM93] F. Guerra, S. Arevalo, A. Alvarez, and J. Miranda. A distributed consensus protocol with a coordinator. *IFIP. International conference on decentralized and distributed systems ICDDS'93.*, Septiembre 1993.

[GBRR92] B. Glade, K. Birman, Cooper R., and Renesse R. Light-weight process groups in the Isis system. *Proceedings of the OpenForum'92 technical conference*, pages 323–336, Noviembre 1992.

[GC85] N. Gammage and L. Casey. Xms: A rendezvous-based distributed system architecture. *IEEE-Software*, pages 9–19, May 1985.

[Geh84] N.H. Gehani. Broadcasting sequential processes (bsp). *Transactions on Software Engineering*, 10(4):343–351, 1984.

[GR89] N.H. Gehani and W.D. Roome. *The concurrent C programming language.* Silicon Press, Summit, N.J., 1989.

[Han73] P.B. Hansen. *Operating Systems principles.* Prentice-Hall, Englewood Cliffs, NJ., 1973.

[Han78] P.B. Hansen. Distributed processes: A concurrent programming concept. *Communications of the ACM*, 21(11):934–940, Noviembre 1978.

[Hoa74] C.A.R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, pages 594–557, Octubre 1974.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Distributed Computing Systems. Prentice-Hall, Englewood Cliffs, NJ., 1985.

[HW89] T.P. Hopkins and M.I. Wolczko. Writing concurrent object-oriented programs using Smalltalk-80. Technical report, Department of Computer Science. University of Manchester, Oxford Road, Manchester, M13 9PL, U.K., Octubre 1989.

[Jal94] P. Jalote. *Fault tolerance in distributed systems.* ISBN 0-13-301367-7. Prentice-Hall, Englewood Cliffs, NJ., 1994.

[JK86] T.A. Joseph and Birman K.P. Low cost management of replicated data in fault-tolerant distributed systems. *ACM Transactions on Computing Systems*, 4(1):54–70, Febrero 1986.

[Joh75] 1975 Johnson, S.C. Yacc - yet another compiler compiler. Technical Report 32, AT&T Bell Laboratories, Murray Hill, N.J., Mayo 1975.

[KMBT92] M.F. Kaashoek, R. Michiels, H.E. Bal, and A.S. Tanenbaum. Transparent fault-tolerance in parallel Orca programs. *Symposium on Experiences with Distributed and Multiprocessor Systems III. Newport beach*, pages 297–312, Marzo 1992.

[KT94] M.F. Kaashoek and A.S. Tanenbaum. Efficient reliable group communication for distributed systems. *(Submited for publication)*, 1994.

[KTV93a] M.F. Kaashoek, A.S. Tanenbaum, and K. Verstoep. Group communication in Amoeba and its applications. *Distributed Systems Engineering Journal*, 1:48–58, Julio 1993.

[KTV93b] M.F. Kaashoek, A.S. Tanenbaum, and K. Verstoep. Using group communication to implement a fault-tolerant directory service. *Proceedings of the thirteenth International Conference on Distributed Computing Systems. IEEE.*, pages 130–139, 1993.

[Lam81] B.W. Lampson. *Atomic transactions. Distributed Systems - Architecture and implementation.* Springer-Verlag, New York, 1981.

[LC85] T.J. LeBlanc and R.P. Cook. High-level broadcast communication for local area networks. *IEEE Software*, Mayo 1985.

[LCN90] L. Liang, S.T. Chanson, and G.W. Neufeld. Process groups and group communications: classification and requirements. *IEEE Computer*, Febrero 1990.

[Les75] M.E. Lesk. Lex - a lexical analyzer generator. Technical Report 39, AT&T Bell Laboratories, Murray Hill, N.J., Mayo 1975.

[Lis88] B. Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312, Marzo 1988.

[LKBT92] W.G. Levelt, M.F. Kaashoek, H.E. Bal, and A.S. Tanenbaum. A comparison of two paradigms for distributed shared memory. *Software Practice and Experience*, 22:985–1010, Noviembre 1992.

[LM94] M.C. Little and D.L. McCue. The replica management system: a scheme for flexible and dinamic replication. *Proceedings of the Second Workshop on Configurable distributed Systems*, Marzo 1994.

[LR80] B.W. Lampson and D.D. Redell. Experiences with processes and monitors in mesa. *Communications of the ACM*, 23(2), Febrero 1980.

[LS83] B. Liskov and R. Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, Julio 1983.

[May83] D. May. Occam. *SIGPLAN Notices*, 18(4):69–79, Abril 1983.

[mt93] Ada9X mapping/revision team. *(Proposed) ANSI Reference Manual for the Ada9X Programming Language*. Septiembre 1993.

[NL91] B. Nitzberg and V. Lo. Distributed shared memory: a survey of issues and algorithms. *IEEE Computer*, pages 52–60, Agosto 1991.

[PBS+88] D. Powell, G. Bonn, D.T. Seaton, P. Verissimo, and F. Waeselynck. The delta-4 approach to dependability in open distributed computing systems. *Digest of papers. FTCS-18. Tokyo*, Junio 1988.

[Per87] R.H. Perrott. *Parallel programming*. Addison-Wesley, 1987.

[RA81] G. Ricart and A.K. Agravala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, pages 9–17, Enero 1981.

[RBC+93] R. Renesse, K. Birman, R. Cooper, B. Glade, and P. Stephenson. The Horus system. Technical report, Cornell, Julio 1993.

[RBG92] M. Reiter, K. Birman, and L. Gong. Integrating security in a group oriented distributed system. *Proceedings of the 1992 IEEE Symposium on research in security and privacy*, Septiembre 1992.

[RG92] M. Reiter and B. Glade. How to securely replicate services. Technical Report TR92-1287, Cornell University. Departament of Computer Science, Mayo 1992.

[RTB93] R. Renesse, M.H. Takako, and K. Birman. Design and performance of Horus: a lightweight group communication system. Technical report, Cornell, 1993.

[RTL+91] K.R. Raj, E. Tem H.M. Levy, A.P. Black, N.C. Hutchinson, and E. Jul. Emerald: A gen rpose programming language. *Software Practice and Experience*, 21(1):9 118, Enero 1991.

[Sch80] F.B. Schneider. Ensuring consistency in a distributed database system by use of distributed semaphores. *Proceedings of International Symposium on Distributed Databases (Versailles, France)*, pages 183–189, Marzo 1980.

[Sch90] F.B. Schneider. Implementing fault tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4), Diciembre 1990.

[Sco91] M.L. Scott. The Lynx distributed programming language: motivation, design and experience. *COMPLANG*, 16:209–233, 1991.

[Sel90] J. Self. *Aflex: An Ada lexical analyzer generator V1.1*. Arcadia Environment Research Project. Department of Information and Computer Science, Universidad de California, Irvine, Mayo 1990. UCI-90-18.

[SS83] R.D. Schlichting and F.B. Schneider. Fail stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computing Systems*, 1(3):222–238, Agosto 1983.

[SY83] R.E. Strom and S. Yemini. NIL: An integrated language and system for distributed programming. *SIGPLAN Notices*, 18(6):73–82, Junio 1983.

[SY85] R.E. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computing Systems*, 3(3):204–226, Agosto 1985.

[SZ90] M. Stumm and S. Zhou. Algorithms implementing distributed shared memory. *IEEE Computer*, pages 54–64, Mayo 1990.

[Tan92] A.S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, 1992.

[TKB92] A.S. Tanenbaum, M.F. Kaash and H.E. Bal. Parallel programming using shared objects and broadcasti *EE Computer*, 25(8):10–19, Agosto 1992.

[TTR88] D. Taback, D. Tolani, and Scl R. Ayacc v1.0. Technical Report UCI-88-16, Departartment of Information and Computer Science. University of California, Irvine, Mayo 1988.

[TVR85] A.S. Tanenbaum and R. Van Renesse. Distributed operating systems. *ACM Computing Surveys*, 17(4):419–470, Diciembre 1985.

[VR92] P. Verissimo and L. Rodrigue. Group orientation: a paradigm for distributed systems of the nineties. *Proceedings of the 3rd IEEE Workshop on Future trends of distributed computing systems*, April 1992.

[VRV92] P. Verissimo, L. Rodriguez, and W. Vogels. Group orientation: a paradigm for modern distributed systems. *Proceedings of the ACM SIGOPS 1992 Workshop*, 1992.

[Wan89] K. Wang. *Arquitectura de computadores y procesamiento paralelo*. McGraw-Hill, 1989.

[Wir77] N. Wirth. Modula: A language for modular multiprogramming. *Software Practice and Experience*, 7(1):3–35, Jan 1977.

[Wir85] N. Wirth. *Programming in Modula-2*. Springer-Verlag, 3rd edition, 1985.

[Xer85] Xerox. *Mesa Language Manual*. Septiembre 1985.

[xmt93] Ada 9x mapping/revision team. *Ada9X rationale*. Septiembre 1993.