# Middleware de distribución y modelo transaccional en sistemas de tiempo real *

J. Javier Gutiérrez, Michael González Harbour y Juan López Campos
Departamento de Electrónica y Computadores
Universidad de Cantabria
39005-Santander, SPAIN
{gutierjj,mgh,lopezju}@unican.es

**Resumen**

En este trabajo se analiza la problemática asociada al diseño e implementación de *middleware* (o software de intermediación) para distribución en aplicaciones de tiempo real. Se identifican los cuellos de botella y las principales decisiones de diseño de este middleware en función de las capacidades de las que se le quiera dotar para soportar el modelo de transacciones, ampliamente reconocido en el análisis de planificabilidad. Estas capacidades varían entre una mayor expresividad y control de los parámetros de tiempo real por parte del middleware en detrimento de cierta facilidad en el uso, y la mayor transparencia de uso con la pérdida de control por parte de la aplicación. Se utilizan como referentes dos modelos de distribución bien conocidos: el modelo de CORBA y el del lenguaje de programación Ada 95.

## 1. Introducción

En la actualidad, cada vez son más frecuentes las aplicaciones que requieren el uso de varios computadores para realizar tareas que necesitan cooperar o sincronizarse, y que tienen algún tipo de restricción temporal que deben cumplir. Este es el campo de los sistemas multiprocesadores y distribuidos de tiempo real. La identificación de estos sistemas no es nueva (el software lleva volando muchos años en aviones y satélites por ejemplo), pero sí que se va renovando la perspectiva con la que se aborda el diseño e implementación de estos sistemas y las herramientas de las que se dispone para realizar estas labores.

Una de estas herramientas es el middleware de distribución. Este middleware como tal aparece bajo distintos formatos y soportando diferentes paradigmas de distribución. Así por ejemplo, CORBA [14] soporta un modelo de distribución de objetos, mientras que el lenguaje Ada 95 [20], aunque también soporta la orientación a objetos, se basa en las llamadas a procedimientos remotos (RPCs, *Remote Procedure Calls*). La ventaja de este middleware de distribución es que proporciona una capa abstracta que permite en principio la realización de las aplicaciones sin la preocupación de si va a ejecutar en uno o varios procesadores. De este modo el programador de la aplicación se preocupa de resolver su problema con independencia de donde va a ser ejecutado. En ausencia de middleware de distribución se pueden utilizar otros paradigmas como el de paso de mensajes o el uso directo de las redes de comunicación. Esto hace que el código de la aplicación deba recoger todas estas llamadas dependientes del modo en el que se va a ejecutar la aplicación.

Para que el middleware pueda ser utilizado en aplicaciones de tiempo real además se deben tener en cuenta otras consideraciones relativas todas a ellas a producir unos tiempos de ejecución deterministas. Así se deben tener en cuenta por ejemplo: la creación dinámica de tareas, las estrategias de planificación y las inversiones en el orden de ejecución de las mismas, o el uso de protocolos y redes de comunicaciones capaces de garantizar el cumplimiento de plazos. Con estas consideraciones aparece por ejemplo el CORBA de tiempo real RT-CORBA [15]. En el caso de Ada 95 el manual no

* Este trabajo ha sido financiado por el Ministerio de Educación y Ciencia del Gobierno de España dentro del proyecto TRECOM (ref. TIC2002-04123-C03-02)

**Actas de las XIII Jornadas de Concurrencia y Sistemas Distribuidos, pp.119-130**
**ISBN: 84-9732-432-3** 

hace una consideración especial para las aplicaciones de tiempo real distribuido [20], pero debido a las características de este lenguaje se puede realizar una implementación del anexo de sistemas distribuidos (DSA, *Distributed Systems Annex*) con características de tiempo real [18][9]. Estos dos middleware de distribución para tiempo real se basan en planificación por prioridades fijas.

Por otra parte, las técnicas de análisis de planificabilidad aplicables a los sistemas distribuidos de tiempo real se encuentran en un nivel de desarrollo importante, tanto para prioridades fijas, como para sistemas planificados por plazo (EDF). Estas técnicas operan sobre modelos bastante precisos de los sistemas reales, y progresivamente van mejorando en la reducción del pesimismo de los tiempos de respuesta que obtienen o en la extensión de los modelos sobre los que operan [16][17][5]. En nuestro grupo se ha desarrollado MAST [4][13], que tomando como eje el modelo de sistema de tiempo real, reúne un conjunto de herramientas de modelado y análisis de planificabilidad aplicable a sistemas distribuidos.

Uno de los modelos más extendido entre las técnicas de análisis es el modelo transaccional [7][8], en el que los eventos van activando la ejecución de actividades (ejecución de tareas en los procesadores o envío de mensajes por las redes de comunicaciones) y éstas a su vez pueden generar nuevos eventos. En MAST por ejemplo se recoge un rico conjunto de patrones de combinaciones de eventos (*Event Handlers*) [4]. Sin embargo, a pesar de que los modelos responden a la descripción del software de aplicaciones reales, cuando se trata de sistematizar el diseño y la implementación de sistemas distribuidos encontramos que la capacidad del middleware de distribución carece de la expresividad incluida en estos modelos analizables, de manera que las aplicaciones de tiempo real que los usan tienen que buscar alternativas al margen del propio middleware que fuerzan a modificar el código propio de la aplicación.

Por último, en el proyecto europeo FIRST [2] se ha desarrollado una arquitectura en el nivel del sistema operativo con planificación flexible que soporta calidad de servicio y tiempo real. Esta arquitectura se basa en la especificación de una API que permite a las aplicaciones realizar reservas de recursos mediante contrato. Fundamentalmente, si un contrato es aceptado se garantiza la disponibilidad de un mínimo de los recursos reservados, y una vez que se cumplen las necesidades mínimas de los contratos establecidos, el resto de los recursos sobrantes se reparten en función de ciertos parámetros que también se especifican en el contrato, para obtener una mejor calidad de los resultados. El modelo de contrato FIRST soporta sistemas distribuidos, pudiendo establecerse los contratos tanto en los procesadores como en las redes de comunicaciones. Esto significa por un lado que las aplicaciones tienen garantía de que se van a cumplir los requerimientos de calidad de servicio y de tiempo real que han firmado en el contrato, y por otro, que se van a utilizar el resto de los recursos disponibles para dar un mejor servicio a las aplicaciones según lo hayan requerido. Esta planificación ha sido implementada y probada en los sistemas operativos MaRTE OS [12] y Shark [19].

El objetivo de este trabajo es analizar los aspectos del middleware de distribución que están implicados en la implementación del modelo de transacciones y proponer algunas alternativas para su diseño de manera que pueda integrar también los aspectos de tiempo real y flexibilidad comentados.

El documento está organizado de la siguiente manera. El apartado 2 está dedicado a presentar el modelo de transacciones que proviene del análisis de planificabilidad. En el apartado 3 se hace un análisis de las características del middleware de distribución actual para implementar el modelo de transacciones. Algunas alternativas al diseño de este middleware para que incorpore los conceptos de transacción y de tiempo real flexible se presentan en el apartado 4. Finalmente, el apartado 5 plantea las conclusiones y el trabajo futuro.

## 2. Modelo de sistema distribuido de tiempo real basado en transacciones

En este apartado se va a describir el modelo de transacciones que incluye MAST [4], que además del tradicional modelo lineal gobernado por eventos incluye un conjunto de patrones de combinaciones de eventos que permite modelar un abanico más amplio de situaciones que se dan en las aplicaciones. El modelo MAST se ha diseñado para

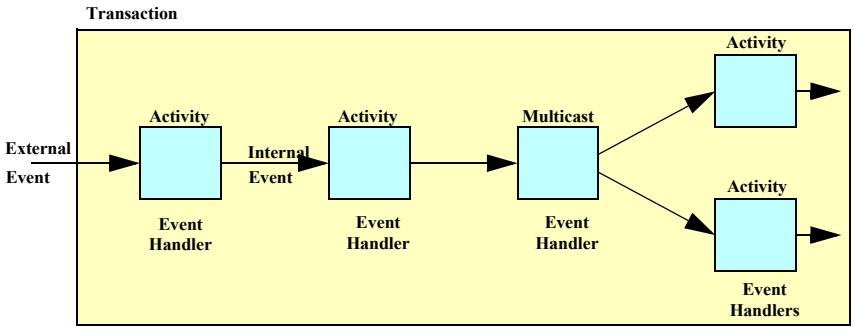


**Figura 1**. Modelo de transacción en MAST

representar tanto sistemas monoprocesadores como multiprocesadores o distribuidos, con énfasis en la descripción de sistemas gobernados por eventos en los que cada tarea puede condicionalmente generar varios eventos a su finalización, o también puede ser activada por uno o más eventos. Los eventos externos que llegan al sistema pueden ser periódicos, aperiódicos, aperiódicos no acotados, esporádicos, a ráfagas, o simples (llegan sólo una vez).

La metodología de modelado facilita la descripción independiente de los parámetros de sobrecarga en los procesadores, en las redes y en los *drivers* de las redes. El modelo soporta además la especificación de requisitos temporales tanto estrictos (plazos, o máximo *jitter* de salida), como no estrictos (plazos laxos, o razones máximas de plazos perdidos). El conjunto de herramientas MAST [13] incluye tanto herramientas de análisis de planificabilidad basado en prioridades fijas, como herramientas de asignación de prioridades. Actualmente se está extendiendo para incluir análisis EDF y asignación de plazos. Se ofrece como código abierto (bajo licencia GNU) y es totalmente extensible.

A continuación se describe brevemente la estructura del modelo de MAST que debe ser considerada por el middleware de distribución.

### 2.1. La transacción

Un sistema de tiempo real se modela como un conjunto de transacciones cada una de las cuales se activa por la llegada de uno o más eventos externos, y representa un conjunto de actividades que serán ejecutadas por el sistema. Las actividades generan eventos que son internos a la transacción y que pueden activar a otras actividades de la transacción. El modelo contiene estructuras especiales capaces de manejar los eventos de diferentes modos. Los eventos internos pueden tener asociados requisitos temporales. La Figura 1 muestra un ejemplo de transacción representada por un grafo que muestra el flujo de los eventos entre los diferentes manejadores de eventos (*Event Handlers*), representados como cajas en el grafo. Esta transacción en particular es activada por un evento externo, después ejecuta dos actividades (*Activities*), y se usa un manejador de eventos de tipo *Multicast* que activa las dos últimas actividades en paralelo.

### 2.2. Los manejadores de eventos

En MAST se consideran dos tipos de manejadores de eventos:

- Actividades. Representan la ejecución de una operación, es decir, un procedimiento o una función en un procesador, o la transmisión de un mensaje por la red.
- Manejadores estructurales. Se limitan a manipular eventos y no consumen recursos o tiempo de ejecución.

Una actividad es activada por un evento de entrada y genera un evento a la salida. Además, ejecuta una operación que representa un trozo de código a ser ejecutado en un procesador, o un mensaje a ser enviado por la red. Entre otras cosas, la actividad lleva asociado el servidor de planificación que la ejecuta junto con sus parámetros de planificación (para prioridades fijas será la prioridad).

Los manejadores estructurales se activan por la llegada de uno o más eventos y pueden generar

además uno o más eventos a la salida. Estos manejadores junto con la actividad están representados en la Figura 2 y se describen brevemente a continuación:

- *Rate Divisor*. Genera un evento de salida cuando han llegado un número de eventos de entrada igual a un número dado.
- *Delay*. Genera el evento de salida cuando ha transcurrido un intervalo de tiempo dado desde la llegada del evento de entrada.
- *Offset*. Genera el evento de salida una vez que ha llegado el evento de entrada y ha transcurrido un determinado tiempo referido a la llegada de algún evento previo. Si el intervalo de tiempo ha pasado, el evento se genera inmediatamente.
- *Concentrator*. Genera un evento a la salida cuando llega cualquiera de los eventos de su entrada.
- *Barrier*. Genera el evento de salida cuando todos los eventos de entrada han llegado.
- *Delivery Server*. Genera un evento en una sola de sus salidas cada vez que llega el evento de entrada. El camino de salida se elige en el instante de generación del evento de acuerdo con una política de elección (por ejemplo cíclica, o aleatoria).
- *Query Server*. Como el anterior genera un evento en una sola de sus salidas cada vez que llega el evento de entrada, pero el camino de salida es elegido en el instante de consumo del evento por una de las actividades que está conectada al evento de salida. En este caso también es necesaria una política de asignación para el caso en el que haya varios requerimientos pendientes de consumo del evento. Se pueden considerar las políticas cíclica, por prioridad, *FIFO*, o *LIFO*.
- *Multicast*. Genera un evento en cada una de sus salidas cada vez que llega el evento de entrada.

Por último, vamos a resaltar otro punto de vista sobre los manejadores de eventos que aparecen en la Figura 2, y que los clasifica en:

- Manejadores lineales de eventos. Tienen un solo evento a la entrada y uno sólo también a la salida.
- Manejadores no lineales de eventos. Tienen más de un evento a la entrada o a la salida.

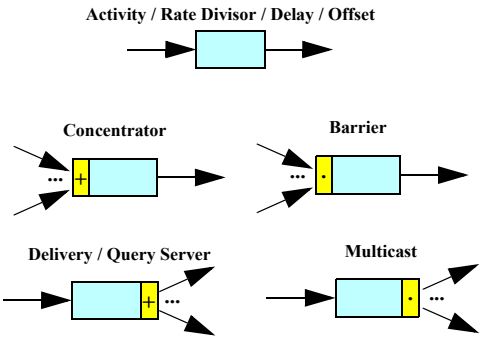


**Figura 2**. Manejadores de eventos del modelo MAST

En resumen, la transacción aparece como un grafo (constituido por eventos y manejadores de eventos) que representa la ejecución de actividades en el sistema. Sería deseable que este modelo estuviera representado en el middleware de distribución para permitir el desarrollo de aplicaciones distribuidas de tiempo real que además de ser analizables, eliminaran del código propio de la aplicación aspectos que corresponden al mero hecho de la distribución. La representación de esta transacción en el actual middleware de distribución se analiza en el siguiente apartado.

## 3. La transacción de tiempo real en el middleware de distribución

Como acabamos de ver en el apartado anterior es fácil pensar en un modelo transaccional para la ejecución de actividades. Naturalmente este modelo sólo representa parte de la realidad, es decir, sólo si adaptamos la realidad a ese modelo la podremos analizar. Pues aún así, y con las restricciones del modelo, el middleware de distribución todavía es más restrictivo porque coarta por ejemplo la libre expresión de uno aspectos más importantes de un sistema de tiempo real: los parámetros de planificación (para sistemas planificados por prioridades fijas, las prioridades).

En [9] se presenta RT-GLADE como modificación de GLADE [18] que es la implementación del anexo de sistemas distribuidos de Ada 95 de GNAT (compilador Ada de código libre de ACT [1]). RT-GLADE solventa, entre otros aspectos, el de la libre asignación de prioridades en las ejecuciones

remotas, eliminando la restricción de los modelos de más rígidos que se habían implementado en GLADE. Estos modelos de propagación de prioridades estaban basados en los de RT-CORBA [15]: *Client Propagated* (el objeto o procedimiento remoto invocado hereda la prioridad del que lo invoca) y *Server Declared* (el objeto o procedimiento remoto invocado se ejecuta siempre a una prioridad previamente establecida). Además RT-GLADE incorpora protocolos de comunicaciones basados en prioridades (RT-EP [11] y CAN-RT-TOP [10]) de manera que a los mensajes generados en las invocaciones remotas también se les pueda asignar la prioridad libremente. RT-GLADE ejecuta sobre el sistema operativo MaRTE OS [12] que implementa el perfil mínimo de POSIX [3]. La libre asignación de prioridades permite optimizar los tiempos de respuesta de las tareas y por tanto mejorar las características de tiempo real de un sistema. Adicionalmente RT-CORBA dispone de un mecanismo de transformación de prioridades a la salida o a la entrada de los servidores que elimina parte de las limitaciones que imponen las dos políticas que propone.

En cualquier caso, RT-CORBA es una aproximación de CORBA al tiempo real, por lo que mantiene el modelo cliente/servidor, que se aleja del modelo transaccional más adecuado para modelar aplicaciones de tiempo real distribuidas. El objetivo de CORBA está más encaminado a la simplificación del desarrollo de las aplicaciones distribuidas proporcionando una visión uniforme, ocultando las capas de red y de sistema operativo, y soportando también varios lenguajes de programación (C, Ada o Java).

Ada 95 sin embargo se basa en el modelo de llamada a procedimiento remoto (RPC) que en principio es más adecuado al modelo transaccional. La ventaja de ser un middleware integrado en el propio lenguaje es que la aplicación distribuida no se diferencia de la que no lo es. De hecho las aplicaciones se pueden concebir sin más, con independencia de donde se vayan a ejecutar, y después si son distribuidas se añaden ciertos parámetros de configuración. Estos parámetros categorizan el código para expresar si puede ser llamado remotamente o no y lo agrupan en particiones que después se podrían ejecutar en un sólo procesador o en varios. La principal virtud es que el código de la aplicación prácticamente no hay que tocarlo.

Cuando añadimos el tiempo real al middleware de distribución, esta pulcritud y benevolencia de las que hace gala se ve seriamente perjudicada, haciendo que aparezcan los efectos de la distribución en medio del código de la aplicación.

Con la idea del CORBA original de simplificación del desarrollo y de ocultamiento de la capa de comunicaciones, y con la concepción de aplicación de Ada 95 vamos a analizar las transformaciones que debe sufrir una sencilla aplicación monoprocesadora de tiempo real que sigue el modelo transaccional al convertirse en distribuida.

### 3.1. Aplicación monoprocesadora de tiempo real

Como ejemplo de aplicación inicialmente monoprocesadora vamos a pensar en una sola transacción tal como la que muestra la Figura 3. Esta es una transacción lineal que podría corresponder a una aplicación que debe activar la transacción, por ejemplo, debido a la detección del paso de un objeto que pasa por una cinta transportadora. La transacción consta de tres actividades:

- La captura de una imagen del objeto.
- El procesado de la imagen en busca de la identificación del objeto.
- La actuación sobre el objeto una vez que se ha identificado.

Para ilustrar el ejemplo vamos a presentar el pseudocódigo en estilo Ada de esta aplicación.

```
task Captura_Imagen is
   pragma Priority (10);
end Captura_Imagen;

task body Captura_Imagen is
begin
   ...
   Procesa_Imagen();
   ...
end Captura_Imagen;

procedure Procesa_Imagen() is
begin
   ...
   Actua ();
   ...
end Procesa_Imagen;
```

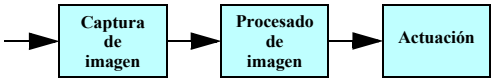


**Figura 3**. Transacción lineal monoprocesadora


**Figura 4**. Transacción lineal distribuida con APCs

```
procedure Actua () is
begin
   ...
end Actua;
```

El código del ejemplo puede constar de una tarea `Captura_Imagen` que estaría esperando, por ejemplo, la sincroniación con un procedimiento de atención de una interrupción provocada por la detección del objeto. Por simplicidad suponemos que la tarea realiza la captura de la imagen y después llama al procedimiento `Procesa_Imagen` que a su vez llamaría a `Actua`. El propio lenguaje Ada 95 permite especificar incluso la prioridad a la que queremos que se ejecute (en este caso 10).

### 3.2. Versión distribuida de la aplicación

Para la versión distribuida de la transacción lineal vamos a suponer que cada actividad de procesado se realiza en un procesador diferente. Además, para simplificar el ejemplo suponemos que las llamadas remotas no retornan ningún valor. Este caso se corresponde con las APCs (*Asynchronous Procedure Calls*) de Ada 95. En esta situación, tal como muestra la Figura 4, la transacción se ve incrementada con dos nuevas actividades que son los mensajes de ida que circulan por la red debido a realización de las llamadas remotas.

En este ejemplo la versión distribuida del código Ada 95 tendría tres particiones, una por procesador, y habría que escribir además un pequeño código de configuración. El código de las particiones quedaría como sigue:

- Partición 1: contiene la tarea `Captura_Imagen` que no cambia.
- Partición 2: contiene el procedimiento `Procesa_Imagen`. Habría que categorizar el paquete que lo contuviera con el pragma `Remote_Call_Interface`, y a él mismo con el pragma `Asynchronous` para indicar que es una APC.
- Partición 3: contiene el procedimiento `Actua` y se categorizaría al igual que `Procesa_Imagen`.

El resultado es concluyente: no ha habido que tocar el código de la aplicación monoprocesadora. Con un proceso de configuración adecuado y el conveniente empaquetado del código, el middleware se encargaría de realizar convenientemente las llamadas remotas.

### 3.3. Versión distribuida de tiempo real de la aplicación

El hecho de añadir requisitos temporales a la ejecución de las actividades implica tener que establecer explícitamente los parámetros de planificación. La arquitectura de la aplicación no cambia, pero incluso con un middleware que soporte la asignación libre de prioridades es necesario modificar el código de la primitiva aplicación monoprocesadora. Las instrucciones necesarias para establecer las prioridades con las que se van a ejecutar las llamadas remotas, y con las que se enviarán los mensajes necesarios por la red, aparecerán infiltradas en el código de la aplicación. Teniendo en cuenta que mantenemos el mismo esquema de tres particiones, mostramos el pseudocódigo inicial con las modificaciones necesarias (en negrita):

```
task Captura_Imagen is
   pragma Priority (10);
end Captura_Imagen;

task body Captura_Imagen is
begin
   ...
   Establece prioridad del mensaje y
   de la ejecución de Procesa_Imagen;
   Procesa_Imagen();
   ...
end Captura_Imagen;

procedure Procesa_Imagen() is
begin
   ...
   Establece prioridad del mensaje y
   de la ejecución de Actua;
   Actua ();
   ...
end Procesa_Imagen;

procedure Actua () is
begin
   ...
end Actua;
```

En cualquier caso, esta situación todavía es programable haciendo uso de los recursos que propor-

ciona el middleware para el establecimiento de las prioridades.

### 3.4. Aplicación monoprocesadora modificada

En este caso modificamos la aplicación monoprocesadora original para hacer algo tan inocente como la reutilización de código. Suponemos que añadimos una nueva transacción que hace lo mismo que la que teníamos, por ejemplo, atiende a una segunda cinta transportadora. La aplicación tendrá una nueva tarea (`Captura_Imagen_Bis`) que realiza la detección y captura de la imagen del objeto, y llamará al procedimiento `Procesa_Imagen`. El pseudocódigo de la nueva tarea de esta aplicación será:

```
task Captura_Imagen_Bis is
   pragma Priority (15);
end Captura_Imagen_Bis;

task body Captura_Imagen_Bis is
begin
   ...
   Procesa_Imagen();
   ...
end Captura_Imagen_Bis;
```

La ampliación de la aplicación no requiere la modificación del código de los procedimientos `Procesa_Imagen` y `Actua`, que previamente se han desarrollado y que se utilizan como un servicio.

### 3.5. Versión distribuida de tiempo real de la aplicación modificada

En la distribución de esta nueva aplicación con dos transacciones, el middleware ya no nos puede ofrecer una solución. La solución habría que buscarla dentro de la propia aplicación. Si continuamos haciendo que las actividades de las transacciones se distribuyan en tres procesadores no es suficiente con modificar el código de la aplicación que sería:

```
task Captura_Imagen is
   pragma Priority (10);
end Captura_Imagen;

task body Captura_Imagen is
begin
   ...
   Establece prioridad del mensaje y
   de la ejecución de Procesa_Imagen;
   Procesa_Imagen();
   ...
end Captura_Imagen;

task Captura_Imagen_Bis is
   pragma Priority (15);
end Captura_Imagen_Bis;

task body Captura_Imagen_Bis is
begin
   ...
   Establece prioridad del mensaje y
   de la ejecución de Procesa_Imagen;
   Procesa_Imagen();
   ...
end Captura_Imagen_Bis;

procedure Procesa_Imagen() is
begin
   ...
   ¿Establece prioridad del mensaje y
   de la ejecución de Actua?;
   Actua ();
   ...
end Procesa_Imagen;

procedure Actua () is
begin
   ...
end Actua;
```

El problema se plantea si necesitamos asignar diferentes prioridades a todas las actividades de las transacciones, porque es el esquema que hace la aplicación planificable. La llamada a `Procesa_Imagen` se realiza correctamente y a la prioridad indicada por cada tarea, pero en la llamada a `Actua` desde `Procesa_Imagen` ya no se pueden establecer las prioridades porque este código puede ser invocado con prioridades diferentes.

Por tanto, en cuanto aparece el anidamiento de llamadas remotas el middleware ya no resuelve el problema. Para solucionar el problema desde la aplicación, podemos proponer dos sencillas alternativas:

- Replicar el código del procedimiento que hace la llamada anidada (`Procesa_Imagen`) para que se puedan establecer las prioridades correctas en las dos llamadas a `Actua`.

```
procedure Procesa_Imagen() is
begin
   ...
   Establece prioridades para llamada
   a Actua desde Captura_Imagen;
   Actua ();
   ...
end Procesa_Imagen;

procedure Procesa_Imagen_Bis() is
begin
   ...
   Establece prioridades para llamada
   a Actua desde Captura_Imagen_Bis;
   Actua ();
   ...
end Procesa_Imagen_Bis;
```

- Modificar `Procesa_Imagen` para pasar un código que permita distinguir las prioridades con las que se llamará a `Actua`.

```
procedure Procesa_Imagen(codigo) is
begin
   ...
   if codigo then
     Establece prioridades para
     llamada desde Captura_Imagen;
   else
     Establece prioridades para lla-
     mada desde Captura_Imagen_Bis;
   end if;
   Actua ();
   ...
end Procesa_Imagen;
```

En cualquier caso podemos imaginar que para una aplicación real con varias transacciones y varias actividades en cada transacción, las alternativas propuestas no son lo más deseable.

### 3.6. Aplicaciones no lineales

Los manejadores de eventos no lineales (vistos en el apartado 2) no están soportados completamente por el middleware de distribución. En este caso se puede plantear la discusión sobre si este middleware debe darles soporte o no.

Vamos a fijar la atención por un momento en los sistemas distribuidos basados en el paso de mensajes. En [6] se plantea el uso de un prototipo basado en colas de mensajes con prioridad que responden a patrones con combinaciones de eventos como las de los manejadores de eventos no lineales. En este caso la aplicación usa directamente las colas de mensajes por lo que la lógica de la aplicación se puede realizar directamente siguiendo estos patrones. Así pues, la sincronización de los eventos de entrada o de salida está implementada en parte por la propia naturaleza de las colas de mensajes.

Cuando se utiliza un middleware de distribución se podría plantear la construcción de estas estructuras de sincronización de eventos en un nivel superior al de comunicaciones, ofreciendo una interfaz de patrones de sincronización. Esta opción en principio parece un poco rígida, y daría lugar a un middleware con capacidad para hacer aplicaciones demasiado encorsetadas quizá, si se quiere hacer un middleware no demasiado complejo. Si por el contrario se quieren ampliar más las posibilidades estructurales del middleware, podría llegar a ser difícil de utilizar.

La otra alternativa es permitir que la aplicación siga implementando directamente su lógica, y que el middleware proporcione mecanismos sencillos que permitan que esta sincronización de eventos tenga lugar (o al menos que no lo impida). Vamos a mostrar un ejemplo de traslación de un manejador de tipo *Barrier* de la versión monoprocesadora a la distribuida en Ada 95:

- La versión monoprocesadora se puede implementar con una tarea que contiene dos *accepts* que se llaman desde dos flujos distintos del programa.

```
task Barrier is
   entry Sucede1();
   entry Sucede2();
end Barrier;

task body Barrier is
begin
  loop
    accept Sucede1();
    accept Sucede2();
    Trabajo útil;
  end loop;
end Barrier;
```

- Una posible versión distribuida, que permitiría que los puntos de entrada de esta tarea pudieran ser llamados remotamente, necesitaría dos procedimientos intermediarios categorizados como remotos y localizados en la misma partición que la tarea.

```
procedure Intermediario_Sucede1() is
begin
  Sucede1();
end Intermediario_Sucede1;

procedure Intermediario_Sucede2() is
begin
  Sucede2();
end Intermediario_Sucede2;
```

Esta alternativa de permitir que la aplicación elija libremente su lógica con los mecanismos que le proporciona el propio lenguaje de programación parece más acertada.

En cualquier caso, para los manejadores de eventos no lineales sigue vigente el problema de anidamiento de llamadas puesto de manifiesto en el apartado anterior (3.5), pero agravado además por el hecho de que las llamadas a un determinado código pueden pertenecer a la misma o a diferentes transacciones.

### 3.7. Aplicaciones con planificación flexible

La aparición de la planificación flexible (como la propuesta en [2]) abre un nuevo mundo al middleware de distribución. El hecho de que todas las actividades del sistema deban ejecutarse bajo los auspicios de un contrato para garantizar los requisitos temporales y de calidad de servicio, hace que la tecnología del middleware tenga que evolu-

cionar para poder hacer uso de los nuevos servicios que proporciona el sistema operativo.

El modelo de planificación diseñado en el proyecto FIRST [2] sustituye los parámetros de planificación básicos del sistema operativo (prioridades o plazos) por un modelo de contrato en el que se establecen parámetros relativos a:

- garantizar un tiempo de procesado mínimo (durante un periodo máximo),
- compartir la capacidad sobrante del procesador (hasta una ejecución máxima en un periodo mínimo con criterios de calidad de servicio),
- declarar el acceso a recursos compartidos,
- reclamar dinámicamente capacidad o devolver la que sobra en un periodo,
- la utilización de la planificación jerárquica (en dos niveles,), y finalmente,
- la distribución.

Todos los puntos excepto el último afectan sólo al sistema operativo, que ofrece una API para su utilización. El punto de mayor interés, dejando los detalles aparte, es que todo el software que ejecute el sistema operativo debe tener un contrato (también el middleware). El contrato asocia sus parámetros a un servidor de planificación (que finalmente está soportado por el sistema operativo subyacente, planificado por prioridades o por plazos). Si el contrato es aceptado se garantiza la ejecución contratada.

La distribución afecta a los mecanismos de planificación de las redes de comunicación que se basa en contratos de red asociados al envío de mensajes. Para ello la red debe implementar los *communication endpoints*, que son una especie de *sockets* a los que se asocia el contrato de red. Se distinguen dos tipos de *endpoints*: de envío (se asocia al servidor de red con los parámetros de planificación de la red) y de recepción (proporciona el mecanismo de almacenamiento de mensajes y espera a los mismos).

Las aplicaciones que usan los sistemas operativos y redes con planificación flexible poseen, aunque pueda parecer paradójico, una estructura muy rígida que es la que marca el contrato (son flexibles en el marco de su contrato). Esto condiciona fuertemente el diseño del middleware hasta sus capas más internas. Hay dos aspectos fundamentales del middleware que se ven afectados:

- El conjunto de tareas (*pool*) que se encargan de proporcionar concurrencia en el nudo destino a las peticiones remotas. Estas tareas tanto en el modelo de RT-CORBA como en el de Ada 95 son anónimas, y para aplicaciones de tiempo real basta con crear un número fijo suficiente en la inicialización para evitar que se creen dinámicamente. En el esquema de planificación flexible las tareas no pueden ejecutar sin contrato, por lo que las tareas no pueden ser anónimas y el middleware debe proporcionar los mecanismos de creación explícita de las mismas y vinculación a un contrato para ejecutar un código concreto.
- En la gestión de las comunicaciones por parte del middleware, al igual que para las tareas que eran anónimas, se deben proporcionar mecanismos que vinculen los contratos con los *communication endpoints* y con las tareas que los utilizan.

En definitiva los puntos que se ven afectados en el middleware al introducir la planificación flexible establecen una línea de unión entre tareas, mensajes, y ejecuciones remotas que estructuralmente se parecen mucho a las transacciones.

## 4. Alternativas de diseño para los nuevos retos del middleware de distribución

El objetivo principal en el diseño sería mantener separada la lógica de las aplicaciones de los parámetros que las caracterizan como aplicaciones de tiempo real, o de calidad de servicio.

Después del análisis realizado en el apartado 3 de este trabajo vamos a apuntar las líneas de actuación de alto nivel que nos permitirán integrar el modelo de transacciones y la planificación flexible en el middleware de distribución. Para ello responderemos a las siguientes cuestiones:

- Hasta qué punto se deben incluir todos los manejadores de eventos en el modelo de transacción soportado por este middleware.
  De acuerdo con lo presentado en apartado 3.6 consideramos tres posibles soluciones:
  - La aplicación implementa la transacción. La aplicación crearía y manejaría su propia

estructura de datos y podría utilizar el actual middleware.
  - El middleware implementa la transacción completa. Esto daría lugar a la creación de patrones de eventos en el middleware que serían ofrecidos a la aplicación. Podría dar lugar a aplicaciones muy rígidas o muy difíciles de programar.
  - El middleware proporciona soporte para un tipo básico de transacción (lineal) y es la aplicación la encargada de implementar el resto de manejadores de eventos. Esta solución es la que más nos gusta ya que los manejadores de eventos no lineales provienen de la propia lógica de la aplicación, y no del hecho de que ésta sea distribuida. Lo que sí debe proporcionar el middleware es el mecanismo básico de identificación de una transacción.
- Cómo se debe gestionar la concurrencia en el nudo destino de las llamadas remotas.
  Las transacciones con un mecanismo de identificación básico se podrían ejecutar como hasta ahora con tareas anónimas. En cambio, la gestión de los contratos por parte del middleware necesita una API para la vinculación de los contratos a las tareas y al código que deben ejecutar. Por tanto, las tareas no serán anónimas, sino que se crearán en la inicialización de la aplicación, como el resto de los contratos para el código que no se ejecuta remotamente.
- Cómo se deben gestionar las comunicaciones.
  Los contratos para sistemas distribuidos (apartado 3.7) se gestionan a través de *communication endpoints* que establecen una relación entre una tarea que envía un mensaje, el propio mensaje que lleva asociado su contrato, y la tarea que lo va a recibir. Al igual que las tareas del *pool* no pueden ser anónimas, estos puntos de comunicación que pertenecen a la red deben ser manejados por el middleware para que sepa reconducir las llamadas remotas. Por tanto, habrá que proporcionar un mecanismo similar y ligado también al mecanismo básico de identificación de la transacción que tiene el mismo objetivo de seguir la línea por la que debe fluir una secuencia de llamadas remotas.
- Hasta qué punto se puede respetar/vulnerar la separación entre la lógica de la aplicación y los requisitos temporales o de calidad de servicio.
  El planteamiento de los puntos anteriores obedece a razones de origen técnico; el de éste obedece a razones prácticas. Aunque en principio no se debe renunciar incluso a proponer soluciones que impliquen el cambio de los estándares RT-CORBA o DSA de Ada 95, como primer paso optaremos por la búsqueda de soluciones a los problemas planteados dentro de la tecnología que más conocemos, Ada 95, manteniéndonos dentro del estándar, y experimentando dichas soluciones en RT-GLADE.

En las decisiones de diseño tomadas en los puntos anteriores aparece un hilo conductor que es la necesidad de crear un identificador de la transacción que se pueda utilizar a lo largo de las sucesivas llamadas remotas de que conste.

Así pues, en esta primera aproximación abordaremos la inclusión de transacciones lineales y de los contratos del FIRST, para lo que proponemos la creación del *identificador de evento*, que será usado como enlace de estructuras similares a los *endpoints* del planificador FIRST. La aplicación debe crear estos *endpoints* de envío (uno para la llamada y otro para la respuesta) y recepción (uno en el que espera respuesta la tarea llamadora y otro en el que espera la llamada la tarea remota) asociados al identificador de evento. En la llamada a una operación remota la tarea llamadora establece mediante una operación especial el evento con el que llama, que es el que permitirá al middleware decidir la secuencia completa de llamada, y respuesta si la hay. Con el identificador del evento, los *endpoints* y la asignación fija de tarea (pérdida del anonimato) para la ejecución remota, las transacciones lineales y el modelo de contrato se pueden soportar de una manera uniforme.

La implementación de los manejadores de eventos no lineales se debe hacer en el código de la aplicación siguiendo su propia lógica. Además se puede consultar en cualquier momento el identificador de evento con el que ejecuta un determinado código para tomar decisiones en función de su valor.

Con esta aproximación se consigue implementar un middleware con un nivel de abstracción bas-

tante alto, que permite encapsular casi sin cambios el código de la aplicación, aunque es necesaria una labor de configuración importante en la que hay que establecer parámetros de planificación o contratos. Desde un punto de vista de la ingeniería de software estas soluciones permiten que el experto en una aplicación concreta se dedique a realizar el código de la aplicación, y luego ésta pueda ser sintonizada por el experto en tiempo real o calidad de servicio, que en muchos casos podrá estar asistido por herramientas que automaticen labores como la extracción de parámetros temporales, el análisis de planificabilidad, o el cálculo de prioridades o de parámetros de calidad de servicio.

## 5. Conclusiones y trabajo futuro

En el trabajo presentado se ha realizado un análisis del middleware de distribución desde el punto de vista de su adecuación a la implementación de sistemas de tiempo real por un lado, y a nuevas necesidades de flexibilidad o de calidad de servicio de las aplicaciones por otro. Se ha puesto de manifiesto que el middleware existente no cumple como tal para este tipo de aplicaciones, es decir, no presenta un nivel de abstracción suficiente.

En particular se ha analizado el modelo transaccional perfectamente establecido en el análisis de planificabilidad, y también la incorporación de esquemas de planificación basados en contratos tanto para los procesadores como para las redes de comunicación.

Se han planteado algunas soluciones de diseño como el hecho de que el middleware sólo soporte internamente las transacciones lineales, o la necesidad de crear una API en el middleware para soportar los modelos de transacción y de contrato. Además se ha identificado al evento como la piedra angular del middleware para soportar el modelo transaccional y la planificación flexible basada en contrato.

La conclusión fundamental es que el trabajo no ha hecho más que empezar, y que la nueva generación de middlewares de distribución de tiempo real incorporará de un modo u otro los paradigmas aquí presentados. Nuestro trabajo continuará en la línea expuesta, llevando a la práctica el desarrollo del modelo transaccional en RT-GLADE.

## Referencias


[1] Ada-Core Technologies, Ada 95 GNAT Pro http://www.gnat.com/

[2] FIRST web page. IST Programme of the European Commission project IST-2001-34820. http://www.idt.mdh.se/salsart/first/

[3] ISO/IEC 9945-1:1998. Information Technology - Portable Operating System Interface (POSIX)- Part 1: System Application Program Interface (API) [C Language]. The Institute of Electrical and Electronics Engineers, 1996.

[4] González Harbour M., Gutiérrez J.J., Palencia J.C., y Drake J.M.. MAST: Modeling and Analysis Suite for Real-Time Applications. Proc. of the Euromicro Conference on Real-Time Systems, Delft, The Netherlands, Junio 2001.

[5] Gutiérrez J.J., Palencia J.C., y González M.. Schedulability Analysis of Distributed Hard Real-Time Systems with Multiple-Event Synchronization. Proc. of the 12th Euromicro Conference on Real-Time Systems, Stockholm, Sweden, 2000.

[6] Gutiérrez J.J., y González Harbour M.. A Framework for Developing Distributed Hard Real-Time Applications. Proc. of 25th IFAC Workshop on Real-Time Programming, WRTP'2000, Elsevier Science, 2000.

[7] Klein M., Ralya T., Pollak B., Obenza R., y González M.. A Practitioner's Handbook for Real-Time Systems Analysis. Kluwer Academic Pub., 1993.

[8] Liu J.. Real-Time Systems. Prentice Hall, 2000.

[9] López Campos J., Gutiérrez J.J., y Michael González Harbour. The Chance for Ada to Support Distribution and Real Time in Embedded Systems. Proc. of the International Conference on Reliable Software Technologies, Palma de Mallorca, Spain, in LNCS, Vol. 3063, Springer, Junio 2004.

[10] López Campos J., Gutiérrez J.J., y González Harbour M.. CAN-RT-TOP: Real-Time Task-Oriented Protocol over CAN for Analyzable Distributed Applications. Proc. of the 3rd International Workshop on Real-Time Networks, Catania (Italy), Julio 2004.

[11] Martínez J.M., González Harbour M., y Gutiérrez J.J.. RT-EP: Real-Time Ethermet Protocol for Analyzable Distributed Applications on a Minimum Real-Time POSIX Kernel. Proc. of the 2nd International Workshop on Real-Time LANs in the Internet Age, Porto (Portugal), Julio 2003.

[12] MaRTE OS web page. http://marte.unican.es/

[13] MAST web page. http://mast.unican.es/

[14] Object Management Group. CORBA Core Specification. OMG Document, v3.0 formal/02-06-01, Julio 2003.

[15] Object Management Group. Realtime CORBA Specification. OMG Document, v2.0 formal/03-11-01, Noviembre 2003.

[16] Palencia J.C., y González Harbour M.. Exploiting Precedence Relations in the Schedulability Analysis of Distributed Real-Time Systems. Proc. of the 20th IEEE Real-Time Systems Symposium, 1999.

[17] Palencia J.C., y González Harbour M.. Offset-Based Response Time Analysis of Distributed Systems Scheduled under EDF. Proc. of the 15th Euromicro Conference on Real-Time Systems, ECRTS, Porto, Portugal, pp. 3-12, Julio 2003.

[18] Pautet L. y Tardieu S.. GLADE: a Framework for Building Large Object-Oriented Real-Time Distributed Systems. Proc. of the 3rd IEEE Intl. Symposium on Object-Oriented Real-Time Distributed Computing, (ISORC'00), Newport Beach, USA, Marzo 2000.

[19] Shark web page. http://shark.sssup.it/

[20] Tucker Taft S., y Duff R.A. (Eds.). Ada 95 Reference Manual. Language and Standard Libraries. International Standard ISO/IEC 8652:1995(E), in LNCS 1246, Springer, 1997.