Componentes Software para Sistemas Distribuidos de Tiempo Real

Tesis Doctoral

Departamento de Lenguajes y Ciencias de la Computación
Universidad de Málaga, España

Presentada por
Daniel Garrido Márquez

Dirigida por
Dr. Manuel Díaz Rodríguez

D. Manuel Díaz Rodríguez, Titular de Universidad del Departamento de Lenguajes y Ciencias de la Computación de la Universidad de Málaga

Certifica

Que **D. Daniel Garrido Márquez**, Ingeniero en Informática por la Universidad de Málaga, ha realizado en el Departamento de Lenguajes y Ciencias de la Computación de la Universidad de Málaga, bajo su dirección, el trabajo de investigación correspondiente a su Tesis Doctoral titulada

Componentes Software para Sistemas Distribuidos de Tiempo Real

Revisado el presente trabajo, estimo que puede ser presentado al tribunal que ha de juzgarlo, y autorizo la presentación de esta Tesis Doctoral en la Universidad de Málaga.

En Málaga, a 9 de Diciembre de 2005

Fdo: Manuel Díaz Rodríguez
Titular de Universidad
Departamento de Lenguajes y
Ciencias de la Computación
Universidad de Málaga

Agradecimientos

En la realización de este trabajo, no pocas personas merecen mi agradecimiento. En primer lugar, Manuel Díaz, director de la tesis, por toda la ayuda prestada no sólo en la tesis, sino en toda la labor que vengo realizando desde mi llegada a la Universidad. En segundo lugar, todos los miembros del Departamento de Lenguajes y Ciencias de la Computación y del grupo GISUM que me han ayudado de alguna u otra forma en la finalización de este trabajo; y en particular, a José María Troya, por todo el apoyo recibido. De manera muy especial a Paco Rus, Luis Llopis, Enrique Soler y José María Alvarez. En tercer lugar, a Tecnatom S.A. sin cuya ayuda, gran parte de este trabajo no podría haber sido realizado. Y, finalmente, a mi mujer, mis padres y familiares, por el apoyo recibido y la paciencia que han tenido hasta que este trabajo ha sido finalizado.

Índice

Lista de figuras

Lista de tablas

Prólogo

El desarrollo de sistemas distribuidos de tiempo real es una de las tareas más complejas a las que pueda enfrentarse un Ingeniero en Informática. En este tipo de sistemas hay que tener en cuenta tres factores: en primer lugar, los requisitos funcionales del sistema a desarrollar, como en cualquier otra aplicación. En segundo lugar, hay que tener en cuenta el desarrollo distribuido de la aplicación, con las dificultades adicionales que ello conlleva. Y, por último, los requisitos temporales, siendo éste quizás el factor de mayor importancia y dificultad.

La Ingeniería del Software lleva desde hace tiempo intentando aplicar sus diversas técnicas en los sistemas de tiempo real para que estos sistemas dejen de ser un "mundo aparte" en el desarrollo de aplicaciones, beneficiándose así de todas las posibilidades de esta disciplina. Pueden observarse, no obstante, algunas carencias en lenguajes, entornos de desarrollo, herramientas, etc. En esta tesis se presenta la utilización de algunas de las tecnologías más novedosas de la Ingeniería del Software aplicándolas en el campo de los sistemas distribuidos de tiempo real y realizando propuestas que permitan solucionar estas carencias.

La tecnología basada en componentes permite la utilización de módulos software reutilizables que permiten la construcción de un sistema mediante el "ensamblado" de diferentes componentes que pueden ser desarrollados de manera independiente. La tecnología basada en componentes presenta numerosas ventajas con respecto al paradigma basada en objetos, evitando muchos de sus problemas. Sin embargo, los modelos estándares de componentes, no tienen en cuenta las especiales necesidades de los sistemas de tiempo real. En esta tesis se presenta un nuevo modelo de componentes junto con un entorno de desarrollo que tiene en cuenta estas necesidades.

Dentro del ámbito de la programación distribuida, el estándar CORBA se presenta como una poderosa alternativa para la construcción de sistemas distribuidos. CORBA presenta numerosas ventajas, tales como, independencia del lenguaje de desarrollo, sistema operativo o plataforma hardware utilizada. Sin embargo, las implementaciones estándares de CORBA no pueden ser utilizadas en sistemas de tiempo real, al no presentar comportamientos predecibles. La extensión de tiempo real de CORBA permite superar estos problemas y obtener un comportamiento predecible en el desarrollo de aplicaciones distribuidas. En esta tesis, RT-CORBA va a ser utilizado como plataforma de ejecución del modelo de componentes propuesto, de forma que pueda tenerse garantizado un comportamiento temporal predecible de las aplicaciones.

SDL es una Técnica de Descripción Formal que puede ser utilizada para la validación y verificación de sistemas software al tener una semántica bien definida. En esta tesis se propone una metodología de análisis de tiempo real que utiliza SDL para realizar el análisis de planificabilidad de los sistemas construidos. Esta metodología toma como punto de partida el trabajo realizado en [Llopis, 2002], donde se definen unas extensiones a SDL que permiten su utilización para el modelado, simulación y análisis de sistemas de tiempo real monoprocesador. En la presente tesis se va a ampliar la metodología propuesta para su utilización en sistemas distribuidos.

Por último, cualquier propuesta realizada en el campo de la Ingeniería del Software debería poder ser aplicada en sistemas reales. El grupo de investigación GISUM ha colaborado en numerosos proyectos con la empresa Tecnatom S.A. para la realización de simuladores para centrales nucleares. Las propuestas realizadas han sido aplicadas en algunos desarrollos para comprobar su idoneidad.

Las principales aportaciones de esta tesis pueden resumirse pues en:

- Definición de un modelo de componentes para tiempo real.
- Plataforma de desarrollo y ejecución de los componentes sobre RT-CORBA.
- Metodología de análisis de tiempo real basada en SDL.
- Aplicación en simuladores de centrales nucleares.

La estructura de esta memoria es como se muestra a continuación. En el primer capítulo se realiza una introducción a los sistemas distribuidos de tiempo real y se presentan los cuatro pilares básicos de la tesis: sistemas distribuidos, componentes, CORBA y SDL. En el segundo capítulo se detalla el modelo propuesto de componentes

para tiempo real. El tercer capítulo muestra cómo puede realizarse su implementación utilizando RT-CORBA como plataforma de ejecución. El cuarto capítulo trata sobre la metodología de análisis que permite realizar análisis de planificabilidad para los sistemas desarrollados mediante el modelo de componentes y su plataforma de ejecución. Por último, se presenta la aplicación de las técnicas propuestas en el desarrollo de aplicaciones para simuladores de centrales nucleares. El último capítulo presenta las conclusiones y algunos trabajos futuros.

Capítulo 1. Introducción

Los sistemas distribuidos de tiempo real (SDTRs) son ampliamente utilizados en la actualidad en todo tipo de situaciones: sistemas militares, medicina, aplicaciones empotradas, simuladores, etc. Este tipo de sistemas, como sus equivalentes en sistemas no distribuidos (STRs), se caracterizan porque su correcto funcionamiento depende no sólo de las entradas y salidas del mismo, sino porque además se debe dar respuesta a los diferentes eventos en el momento adecuado, pudiendo ser fatal cualquier retraso. En particular, la complejidad de los SDTRs es aún mayor, puesto que tienen que tratar con un nuevo elemento, la red, que debe ser considerado en el análisis de tiempo real.

El desarrollo de los SDTRs viene caracterizándose por soluciones *ad hoc* que no garantizan predecibilidad o que si aportan esta característica, no se han analizado adecuadamente, por lo que en ambos casos las aplicaciones desarrolladas no pueden considerarse como aplicaciones de tiempo real estricto. No obstante, en los últimos años, se vienen realizando intentos por la comunidad de tiempo real para utilizar las nuevas técnicas y metodologías de la Ingeniería del Software en los sistemas de tiempo real, y en particular, en los SDTRs.

Una primera aportación es la utilización de componentes software. Los componentes en el sentido de [Szypersky et al., 2002] son módulos binarios con interfaces bien definidas, de forma que una aplicación puede realizarse en base a la composición de componentes y sus interacciones, aumentando el grado de reutilización del software así como su calidad, al disponer de partes de código ya comprobadas. El principal inconveniente de los modelos de componentes actuales es que no son adecuados para su utilización en sistemas de tiempo real, al no permitir, por ejemplo, la indicación de restricciones temporales.

En el campo de las aplicaciones distribuidas destaca el trabajo del OMG (*Object Management Group*) para la definición de una arquitectura para aplicaciones

distribuidas [OMG, 1999]. El resultado es la arquitectura OMA (*Object Management Architecture*), de la que CORBA (*Common Object Request Broker Architecture*) es su parte central. CORBA permite desarrollar aplicaciones que se comunican independientemente del lenguaje de programación, sistema operativo o plataforma hardware. El principal inconveniente para la utilización de CORBA en SDTRs es que el tiempo de respuesta de las invocaciones no está acotado, por lo que no puede realizarse un análisis de tiempo real.

La solución para la utilización de CORBA en SDTRs pasa por la utilización de la extensión de tiempo real conocida como Real-time CORBA o RT-CORBA [OMG, 2005]. RT-CORBA permite desarrollar aplicaciones distribuidas predecibles gestionando los recursos del procesador, las comunicaciones y la memoria [Schmidt y Kuhns, 2000].

Utilizando componentes software y RT-CORBA como middleware de comunicaciones, pueden desarrollarse eficientemente aplicaciones distribuidas predecibles, pero falta un tercer elemento que permita utilizar estas aplicaciones para tiempo real, como es una metodología de análisis de tiempo real que permita analizar aplicaciones distribuidas. En este sentido, el trabajo presentado en [Alvarez et al., 1999] utiliza SDL (*Specification Description Language*) [ITU SDL, 1994], técnica de descripción formal (TDF) que en ese trabajo es ampliada para permitir el análisis de tiempo real de sistemas diseñados en SDL, aunque centrándose únicamente en sistemas no distribuidos.

El objetivo de la presente tesis es ampliar y fusionar adecuadamente estos tres elementos para poder aplicar las técnicas de la Ingeniería del Software en los sistemas de tiempo real. Para ello, se va a definir un modelo de componentes distribuidos que junto con RT-CORBA como plataforma de comunicaciones y una metodología de análisis basada en la extensión de tiempo real de SDL va a permitir realizar aplicaciones distribuidas de tiempo real analizables.

Mediante la definición de un nuevo modelo de componentes para tiempo real, nuestra propuesta permite superar las carencias que presentan los modelos de componentes convencionales que se centran únicamente en la funcionalidad de los sistemas, sin considerar el comportamiento temporal. El modelo incorpora un lenguaje para la definición de componentes de tiempo real, que junto con un conjunto de herramientas adecuado, permite utilizar la metodología de componentes en sistemas de tiempo real distribuidos.

Para que el modelo de componentes sea predecible, debe utilizar una plataforma de comunicaciones que aporte esta característica. Para ello, RT-CORBA se perfila como el candidato idóneo para la implementación del modelo de componentes manteniendo la predecibilidad. La utilización de RT-CORBA permitirá construir aplicaciones en diferentes plataformas, con diferentes sistemas operativos e incluso con diferentes lenguajes de programación.

Por último, se necesita una metodología de análisis para comprobar que las aplicaciones desarrolladas con el modelo de componentes y sobre la plataforma RT-CORBA son planificables. El modelo extendido de SDL para sistemas distribuidos permitirá realizar un análisis de tiempo real del sistema y lo que es también importante, comprobar desde la fase de diseño posibles problemas de planificabilidad, al poder simularse por completo el sistema incluyendo las comunicaciones. Este modelo debe incluir también el coste de las comunicaciones y, en concreto, el coste de RT-CORBA.

En el resto de este capítulo se verá una descripción de los sistemas distribuidos de tiempo real. Posteriormente se describirán los tres elementos que constituyen la base para esta tesis: componentes, CORBA y su extensión de tiempo real, y finalmente la extensión de tiempo real de SDL.

1.1. Sistemas Distribuidos de Tiempo Real

El desarrollo de sistemas distribuidos predecibles y planificables es un desafío especial para cualquier desarrollador. Más allá de las dependencias con el Sistema Operativo de Tiempo Real (SOTR) utilizado para proporcionar una planificación de hebras predecible, los sistemas distribuidos dependen de factores adicionales además de los contemplados en la planificación en un solo procesador. Estos factores adicionales incluyen las características de rendimiento de otros componentes del sistema tales como la implementación del protocolo de red, el *firmware* del adaptador de red; o el medio físico de la red o bus.

Además de tener que considerar estos factores, las teorías clásicas de planificación, tales como el análisis basado en razón monótona (ARM) [Liu y Layland, 1973] u otras, no consideran la planificación de múltiples procesadores y mucho menos el soporte de múltiples anchos de banda. Los diseñadores de sistemas distribuidos tienen una tarea difícil en determinar la planificación para los procesadores, buses y redes del sistema. Muchas veces, la solución pasa por adoptar soluciones *ad hoc*, que si bien

pueden ser correctas, no son mantenibles o reutilizables, puesto que cualquier pequeño cambio en los requisitos puede hacer que el sistema ya no sea planificable.

1.1.1. Definición de sistema distribuido

Un sistema computacional distribuido está formado por varios elementos de procesamiento autónomos que cooperan en un objetivo común o para lograr una meta común.

Resulta útil clasificar los sistemas distribuidos en *fuertemente acoplados*, en los que los elementos de proceso, o nodos, tienen acceso a una memoria común, y *débilmente acoplados*, que carecen de ella. La importancia de esta clasificación radica en que la sincronización y la comunicación en un sistema fuertemente acoplado pueden efectuarse mediante técnicas basadas en el empleo de variables compartidas, mientras que en un sistema débilmente acoplado se requiere, en último término, paso de mensajes o invocaciones remotas.

En la presente tesis, la expresión "sistema distribuido" hará referencia a una arquitectura débilmente acoplada. Se supondrá además una conectividad completa (sin considerar enrutado de mensajes), y que cada procesador sólo tiene acceso a su propio reloj, el cual se encuentra débilmente sincronizado con el resto.

Partiendo de la variedad de procesadores del sistema, se puede establecer otra clasificación. Un *sistema homogéneo*, es aquél en el que todos los procesadores son del mismo tipo; un *sistema heterogéneo* contiene procesadores de tipos diferentes. Es muy interesante disponer de herramientas, lenguajes de programación, etc. que permitan su utilización en sistemas heterogéneos. En este sentido cabe destacar la importancia de CORBA, que puede ser utilizado independientemente del sistema operativo, plataforma hardware o lenguaje de programación.

En un sistema distribuido hay ciertos factores que cobran especial importancia:

- Soporte del lenguaje: el desarrollo de un programa distribuido se facilita en gran medida si el lenguaje y su entorno de programación soportan el particionado, la configuración, asignación y reconfiguración de la aplicación distribuida, junto a un acceso independiente de la ubicación de los recursos remotos.
- Fiabilidad: disponer de varios procesadores permite que la aplicación sea tolerante a fallos; si bien, la aplicación deberá ser capaz de explotar esta redundancia. El

disponer de varios procesadores también introduce la posibilidad de que aparezcan fallos distintos a los que aparecen en un sistema monoprocesador.

- Algoritmos de control distribuidos: La presencia de paralelismo real en la aplicación, procesadores físicamente distribuidos, y la posibilidad de que fallen los procesadores y los elementos de proceso, implica la necesidad de nuevos algoritmos para el control de los recursos.
- Planificación con tiempos límite (*deadlines*): cuando los procesos son distribuidos, los algoritmos óptimos para un procesador dejan de serlo. Se precisan nuevos algoritmos.

1.1.2. Programación en sistemas distribuidos

En este ámbito podemos distinguir tres paradigmas dentro de los más utilizados: librerías de bajo nivel, procedimientos remotos y objetos distribuidos.

Programación mediante librerías de bajo nivel

La programación mediante mecanismos de bajo nivel es el mecanismo más inmediato, sin embargo, presenta el inconveniente de la dificultad en su programación. El desarrollador es el responsable de todas las tareas: localización de recursos, empaquetamiento de parámetros, protocolos, etc. Ejemplos de estas librerías o APIs son, por ejemplo, las diferentes implementaciones de *sockets* para los protocolos de transporte de red.

Llamada a procedimientos remotos

El objetivo del paradigma de llamada a procedimientos remotos (RPC) [Birrell y Nelson, 1984] es hacer la comunicación distribuida tan simple como sea posible. Usualmente se emplea RPC para comunicar programas escritos en el mismo lenguaje. Entre los programas, uno de ellos, *servidor*, puede ser llamado remotamente por otro denominado *cliente*. A partir de la especificación del servidor, es posible generar automáticamente los denominados procedimientos *stub* y *skeleton.* El *stub* se sitúa en vez del servidor en el lugar donde se origina la llamada remota. El *skeleton* se ubica en el mismo lugar que el procedimiento de servicio. El cometido de estos dos

procedimientos es proveer un enlace transparente entre el cliente y el servidor tal y como puede verse en la Figura 1.1.

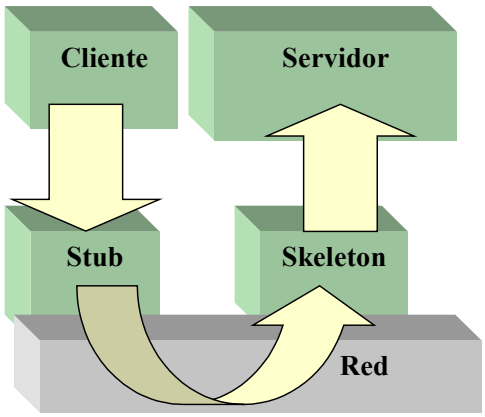


Figura 1.1. Llamada remota a procedimiento

El *stub* realiza las siguientes funciones: se encarga de localizar al *skeleton* del servidor, empaqueta los parámetros de la llamada remota para su transmisión a través de la red, envía la petición al *skeleton* y espera la respuesta del servidor con desempaquetamiento de los parámetros.

El *skeleton*, por su parte, recibe peticiones de los procedimientos *stub* del cliente, desempaqueta los parámetros, invoca el servicio en el servidor y vuelve a transmitir la respuesta al cliente.

Los principales inconvenientes de los RPC pasan por su dificultad de programación, siendo un mecanismo de bajo nivel en comparación con otros. Presentan asimismo dificultades para su utilización en distintas plataformas o lenguajes de programación.

El modelo de objetos distribuido

El término *objetos distribuidos* en su sentido más amplio, permite:

- La creación dinámica de un objeto (en cualquier lenguaje) sobre una máquina remota.
- La identificación de un objeto por determinar y alojado en cualquier máquina.
- La invocación transparente de un método remoto sobre un objeto como si fuera un método local, sin importar el lenguaje en el que esté codificado el objeto.
- El reparto de ejecución (*dispatch*) de la llamada a un método a través de la red.

No todos los sistemas que permiten objetos distribuidos proporcionan mecanismos que aporten toda esta funcionalidad.

Dentro de este paradigma de programación, los denominados *middleware* tienen gran importancia. Un *middleware* es el software que reside entre la aplicación y los sistemas operativos, protocolos y hardware subyacente y cuyo objetivo es permitir que componentes heterogéneos y distribuidos se interconecten y colaboren entre sí. CORBA, por ejemplo, podría verse como un middleware de comunicaciones.

Este paradigma de programación distribuida es uno de los más utilizados hoy día, presentando todas las ventajas de la orientación a objetos, aunque, por el contrario, puede presentar una menor eficiencia debido a la sobrecarga de objetos, métodos, interfaces, etc. pero esta sobrecarga es muchas veces preferible debido a las ventajas.

1.1.3. Planificación en sistemas distribuidos

El desarrollo de esquemas de planificación apropiados para sistemas distribuidos (y multiprocesador) es problemático. Como se verá en capítulos posteriores, en esta tesis se trata también este problema, aportando una solución basada en técnicas de descripción formal.

Graham (1969) [Graham, 1969] mostró que los sistemas multiprocesador pueden comportarse de un modo bastante impredecible en cuanto al comportamiento de temporización que presentan. Empleando asignación dinámica (esto es, procesos que se asignan a procesadores cuando pasan al estado de ejecutables), encontró las siguientes anomalías:

- Disminuir el tiempo de ejecución de un proceso *P*, podría conducir a que tuviera un tiempo de respuesta mayor.
- Incrementar la prioridad de un proceso *P*, podría conducir a que tuviera un tiempo de respuesta mayor.
- Incrementar el número de procesadores podría conducir a que *P* tuviera un tiempo de respuesta mayor.

Por su parte, Mok y Dertouzos (1978) [Mok y Dertouzos, 1978] mostraron que los algoritmos que son óptimos para sistemas monoprocesador no son óptimos para mayores números de procesadores. Otros ejemplos podrían demostrar que la formulación “primero el *deadline* más cercano” (EDF) es igualmente no óptima. En

conclusión, la planificación óptima para sistemas multiprocesador es NP-duro (Graham et al., 1979) [Graham et al., 1979]. Es necesario buscar formas de simplificar el problema y de proporcionar algoritmos que den resultados subóptimos adecuados.

Asignación de procesos

La asignación cubre el proceso real de convertir el sistema configurado en un conjunto de módulos ejecutables, y descargar éstos sobre los elementos de proceso concretos del sistema. Una asignación aparentemente lógica de procesos puede afectar negativamente a la planificabilidad.

Para procesos periódicos, si se usa un despliegue de procesos estático, entonces el algoritmo de tiempo límite monótono, u otros monoprocesador, pueden comprobar la planificabilidad sobre cada procesador. Al efectuar la asignación, los procesos relacionados armónicamente deberían desplegarse juntos (en el mismo procesador), puesto que esto ayudará a incrementar la utilización del sistema.

En el caso de procesos esporádicos podría parecer útil la misma aproximación, con un comportamiento estático. Pero, la desventaja de esta aproximación es que no se obtiene ningún beneficio de la capacidad de reserva de algún procesador cuando algún otro procesador experimenta una sobrecarga transitoria.

Para los sistemas de tiempo real estrictos, es necesario que cada procesador sea capaz de tratar con los peores casos de tiempo de ejecución de sus procesos periódicos, y con los tiempos máximos de llegada y ejecución para su carga esporádica. Para mejorar esta situación en [Stankovic et al., 1985] se proponen algoritmos de planificación de tareas más flexibles.

Bloqueos remotos

Calcular los tiempos de ejecución (en el caso medio y en el peor) requiere conocer los bloqueos potenciales. El bloqueo interno al procesador puede delimitarse mediante protocolos de herencia o de techo de prioridad. Sin embargo, en un sistema multiprocesador existe otra forma de bloqueo: éste ocurre cuando un proceso es retrasado por otro proceso de otro procesador. Este nuevo tipo de bloqueo se denomina *bloqueo remoto*, y no se acota fácilmente.

En un sistema distribuido, el bloqueo remoto puede eliminarse añadiendo procesos que gestionen la distribución de los datos. Por ejemplo, en lugar de estar bloqueado esperando a leer ciertos datos de un lugar remoto, podría añadirse un proceso adicional a dicho lugar remoto cuyo papel fuera enviar los datos donde se precisen. Así, los datos estarían disponibles localmente. Este tipo de modificación sobre el diseño puede hacerse sistemáticamente, pero complica la aplicación (aunque conduce a un modelo de planificación más simple).

1.1.4. El medio de comunicación

La comunicación entre procesos que están en diferentes máquinas en un sistema distribuido requiere la transmisión y la recepción de los mensajes a través del subsistema de comunicación subyacente. En general, estos mensajes deberán competir unos con otros para ganarse el acceso al medio de comunicación (por ejemplo, conmutadores, buses o anillos). Para que los procesos de tiempo real cumplan sus tiempos límites, deberá ser necesario planificar el acceso al subsistema de comunicación de forma que sea consistente con la planificación de los procesos de cada procesador. Si éste no es el caso, entonces puede aparecer una inversión de prioridad cuando un proceso de alta prioridad intenta acceder al enlace de comunicación. Los protocolos estándar como los asociados con *Ethernet* no soportan el tráfico con restricciones de tiempo estrictas, ya que tienden a incluir los mensajes en colas FIFO o a utilizar algoritmos de retractación (*backoff*) no predecibles cuando se detecta una colisión de mensajes.

Aunque el enlace de comunicación es sólo otro recurso, existen al menos cuatro cuestiones que diferencian el problema de la planificación de enlaces del de la planificación de procesadores.

- A diferencia de un procesador, que tiene un único punto de acceso, un canal de comunicación tiene muchos puntos de acceso; uno por cada nodo físicamente conectado.
- Mientras que los algoritmos interrumpibles son adecuados para la planificación de procesos sobre un único procesador, la interrupción durante la transmisión del mensaje hace que necesite retransmitirse el mensaje completo.

- Además de los tiempos límites impuestos por los procesos de la aplicación, habría también que imponer tiempos límite sobre la disponibilidad del buffer. Los contenidos de cada buffer deberán ser retransmitidos antes de que puedan situarse nuevos datos en él.

Si bien se han empleado muchas aproximaciones *ad hoc*, en los sistemas distribuidos hay al menos tres esquemas que permiten un comportamiento predecible.

TDMA

La extensión natural al empleo de un ejecutivo cíclico para la planificación monoprocesador, es emplear una aproximación cíclica para las comunicaciones. Si todos los procesos de la aplicación son periódicos, es posible producir un protocolo de comunicación por franjas de tiempo. Tales protocolos se denominan TDMA (*Time Division Multiple Access*, *acceso múltiple por división del tiempo*).

Cada nodo dispone de un reloj sincronizado con el resto de relojes de los nodos. A cada nodo se le asignan franjas de tiempo en las que puede comunicarse durante cada ciclo de comunicación. Éstas se encuentran sincronizadas con las franjas de ejecución del distribuidor cíclico de cada tiempo. No deberían aparecer colisiones, dado que cada nodo sabe cuándo puede escribir, y además sabe cuándo hay un mensaje disponible que necesita leer.

La dificultad con la aproximación TDMA está en construir la planificación; además, esta dificultad se incrementa exponencialmente con el número de nodos del sistema. Una arquitectura que ha mostrado considerable éxito en el empleo de TDMA es TTA (*Time Triggered Architecture*; *arquitectura disparada por tiempo*) [Kopetz, 1997], que emplea una heurística de reducción de un grafo complejo para construir los planes. La otra desventaja de TDMA es que es difícil planificar cuándo pueden enviarse los mensajes esporádicos.

Esquemas de paso de testigo temporizado

Una aproximación más general a la basada puramente en franjas de tiempo es el empleo de un paso de testigo: cierto mensaje especial (*testigo*) es pasado de un nodo a otro. Los nodos pueden enviar mensajes sólo cuando poseen el testigo, y dado que sólo hay un

testigo, no podrá haber colisiones entre mensajes. Cada nodo posee el testigo sólo durante un tiempo máximo, y por esto el tiempo de rotación del testigo se encuentra limitado.

Cierto número de protocolos emplean esta aproximación, como por ejemplo, el protocolo de fibra óptica FDDI [Ross, 1989] (*Fiber Distributed Data Interface*; Interfaz de Datos Distribuidos de Fibra).

Protocolos basados en prioridad

Este tipo de protocolos intentan aplicar el uso de prioridad en los procesadores a los mensajes. Tales protocolos suelen tener dos fases: en la primera fase, cada nodo indica la prioridad del mensaje que desea transmitir. Al final de esta fase, uno de los nodos habrá ganado el derecho a transmitir su mensaje. En algunos protocolos, podrán solaparse las dos fases (esto es, mientras cierto mensaje está siendo difundido, se modificarán partes del mensaje de modo que se pueda determinar la prioridad del próximo mensaje).

Si bien durante cierto tiempo se han venido definiendo protocolos basados en prioridad, éstos solían tener la desventaja de que los protocolos de comunicación sólo soportaban un pequeño rango de prioridades. Esta limitación ha sido superada hoy día, por ejemplo, en el protocolo CAN (*Controller Area Network*; red de área de controladores) [Tindell et al., 1995].

En CAN 2.0A, cada mensaje de 8 bytes está precedido por un identificador de 11 bits que sirve de prioridad. Al comienzo de la secuencia de transmisión, cada nodo escribe (simultáneamente) al bus de difusión el primer bit de su identificador de mensaje de máxima prioridad. El protocolo CAN actúa como una gran puerta AND; si cualquier nodo escribe un 0, todos los nodos leen un 0. El bit 0 se denomina *dominante*. El protocolo procede como sigue (para cada bit del identificador):

- Si un nodo transmite un 0, continuará con el próximo bit.
- Si un nodo transmite un 1 y lee un 1, continuará con el próximo bit.
- Si un nodo transmite un 1 y lee un 0, se retrae, esto es, no toma parte en esta ronda de transmisión.

Cuanto menor sea el valor del identificador, mayor prioridad tendrá. Como los identificadores son únicos, el protocolo fuerza a que sólo un nodo acabe dominando tras 11 rondas bit a bit.

El principal valor de CAN es que es un genuino protocolo basado en prioridad, por lo que pueden aplicarse técnicas de análisis tradicionales. La desventaja de este tipo de protocolo es que restringe la velocidad de comunicación.

Para que sea posible que todos los nodos escriban sus bits de identificación simultáneamente, y para que todos lean los valores AND subsiguientes (y actúen conforme a este valor antes de enviar o no, el siguiente bit), hay que imponer severos límites en la velocidad de transmisión. Estos límites son, en realidad, función de la longitud del cable empleado para transmitir los bits, y por ello CAN no es adecuado para entornos geográficamente dispersos. Su principal ámbito de utilización, de hecho, es en sistemas de tipo empotrado.

ATM

ATM [McDysan y Spohn, 1998] puede utilizarse tanto en redes de área extensa como en redes de área local. El objetivo es dar soporte a un amplio rango de requisitos de comunicación, como los que se requieren para voz, vídeo o transmisión de datos.

ATM permite la comunicación punto a punto mediante uno o más conmutadores. Habitualmente, se conecta cada computador de la red a un conmutador mediante dos fibras ópticas: una para el tráfico hacia el conmutador, y la otra para el tráfico en sentido contrario.

Cualquier dato transmitido se divide en paquetes de tamaño fijo llamados *células*. Las aplicaciones se comunican entre ellas mediante *canales virtuales*. Los datos que se transmiten en cada canal virtual tienen cierto comportamiento de temporización asociado, como cadencia de bit, periodo o tiempo límite.

Un *nivel de adaptación* proporciona servicios específicos para soportar los datos de clase de usuario concretos; el comportamiento preciso de este nivel varía para adecuarse a las necesidades de transmisión de un sistema concreto. En el nivel de adaptación se efectúa la corrección de error y sincronización de principio a fin, y la segmentación y el reensamblado de los datos de usuario en células ATM.

Una red ATM corriente puede contener muchos conmutadores, cada uno de los cuales lleva asociado una política de entrada/salida de células; en la mayoría de los conmutadores comerciales se empleaba la política FIFO básica, si bien actualmente, algunos proporcionan una política basada en prioridad, siendo esto más adecuado para su utilización en sistemas de tiempo real.

1.2. El paradigma de programación basado en componentes

La Ingeniería del Software evolucionó desde los lenguajes imperativos como COBOL, Pascal, Fortran, etc. hasta lenguajes basados en objetos por las exigencias de las aplicaciones: mayor complejidad, difícil mantenimiento, necesidad de reutilización etc. La programación orientada a objetos parecía la panacea para todos estos problemas y permitía abarcar proyectos más complejos y de una forma más segura.

A pesar del éxito de la programación orientada a objetos, surgieron nuevos problemas o desafíos, como por ejemplo, las posibilidades de reutilización, un objetivo importante de cualquier modelo de objetos. En lenguajes como C++, se utiliza la herencia de implementación, que permite a un objeto heredar (subclase) algunas de las funciones de otro objeto y sobrescribir otras. Sin embargo, este tipo de reutilización no deja claro qué ocurre, cuando se hacen cambios, en el comportamiento de otros objetos: el "contrato" es implícito y ambiguo, o al menos, propenso a errores.

En aplicaciones simples, los "problemas" de la programación orientada a objetos pueden no ser problemáticos, pero en aplicaciones mayores con distintos equipos, la actualización de alguna clase, puede afectar a otras sin que llegue al conocimiento de otros miembros del equipo.

Con el tiempo, un nuevo paradigma basado en componentes ha surgido [Szypersky et al., 2002]. Los componentes software son bloques reutilizables para la construcción de aplicaciones. Difieren de otro tipo de software reutilizable en que pueden ser modificados en tiempo de diseño como ejecutables binarios, sin tener que conocer absolutamente nada sobre su implementación.

En el campo de tiempo real, se han usado fundamentalmente lenguajes como Ada, C y últimamente C++. Ninguno de estos lenguajes está basado en componentes, por lo que adaptar estos, o definir un nuevo lenguaje basado en componentes sería un reto y posiblemente también un paso adelante en el desarrollo de aplicaciones de tiempo real.

Un modelo de componentes define cómo deben ser los componentes y cómo interoperan con otros componentes y con el sistema.

Algunas de las características deseables de los modelos de componentes son las siguientes:

- Estandarizado: el modelo debería ser aprobado por un amplio grupo de profesionales e investigadores.

- Independiente del lenguaje: un modelo de componentes no está desarrollado para un lenguaje de programación específico ni para ninguna característica especial de ningún lenguaje.
- Define interfaces: un modelo de componentes debería proporcionar un mecanismo para definir las interfaces de un componente.
- Facilita el empaquetado: cómo el componente es suministrado al usuario.
- Permite introspección: un modelo de componentes permite la introspección de los componentes en varias fases de desarrollo, desde la fase de diseño hasta la de ejecución.
- Soporte para dinamismo: un modelo de componentes debe permitir al sistema reorganizarse en tiempo de ejecución: los componentes y sus conexiones.

A continuación se describirán los principales modelos actuales para la programación con componentes, entre los que encontramos .NET de Microsoft, CCM de OMG y los *beans* de Java.

1.2.1. La plataforma .NET

Recientemente Microsoft ha dirigido todos sus esfuerzos hacia la tecnología .NET [Microsoft .NET, 2005]. En el estado anterior a .NET, la tecnología basada en los sistemas operativos y lenguajes de Microsoft sufría de algunas deficiencias como puedan ser los continuos "parches" en sistemas operativos; el mantenimiento de la compatibilidad con versiones previas (no se aprovechaban al máximo las capacidades de las nuevas máquinas) y la continua extensión de la interfaz de programación (API) en vez de su reemplazamiento por una nueva, lo que dificultaba enormemente la labor de los programadores.

Todos estos problemas, y algunos más, llevan a Microsoft el replantearse los modelos de programación utilizados, entornos de ejecución, etc. Surge así la plataforma .NET y la amalgama de tecnologías relacionadas a este término, destacando así, por ejemplo, el lenguaje C# [Robinson et al., 2002] diseñado para aprovechar al máximo las características de esta nueva plataforma. Se pueden tener dos visiones de .NET:

- Librería para el desarrollo de aplicaciones: .NET puede verse como una nueva librería o API para el desarrollo de aplicaciones.

- Entorno de ejecución en el que los programas se van a ejecutar, proporcionando un nivel de abstracción sobre el sistema operativo: en esta visión, .NET funciona como una máquina virtual que va a permitir la ejecución de las distintas aplicaciones.

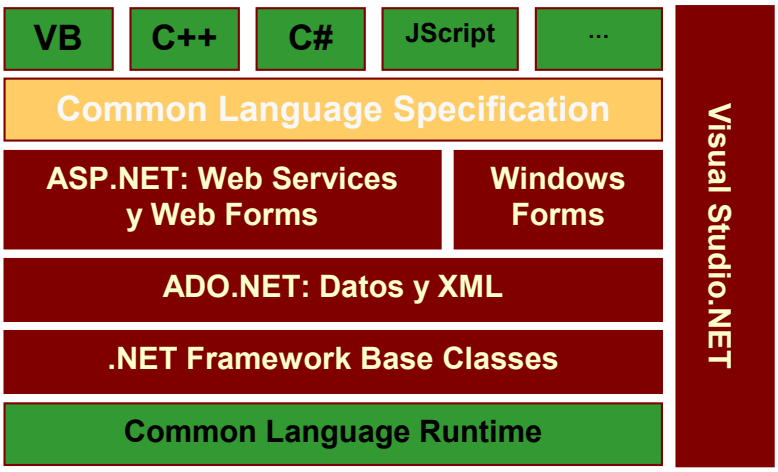


Figura 1.2. La plataforma .NET

Dentro de la plataforma .NET existen distintos lenguajes para el desarrollo de aplicaciones. Dentro de estos lenguajes desempeña un papel destacado el lenguaje C#, el primer lenguaje de la familia C/C++ orientado a componentes y que incluye conceptos de orientación a componentes como conceptos de primera clase en el lenguaje, tales como eventos, propiedades, etc.

Con .NET el modelo COM [Box, 1997] va a ser reemplazado por los denominados *assemblies* (ensamblados), intentando superar los problemas que COM tenía. Este subapartado se centrará en el estudio de los componentes de .NET y su utilización para tiempo real, teniendo la plataforma .NET numerosas características que quedan fuera del ámbito de estudio de la presente tesis.

Los componentes de .NET

En .NET los componentes van a estar contenidos en ensamblados. Un ensamblado es el término .NET para una unidad de configuración y despliegue. Sin embargo, un ensamblado no es un componente en .NET. Para .NET un componente es la forma binaria de una clase y un simple ensamblado puede contener muchos componentes de este tipo.

Un ensamblado consta de metadatos, descripción de tipos exportados y métodos, código intermedio de .NET y recursos. Todas estas partes pueden estar en un solo fichero o en varios.

Las principales características de los ensamblados son las siguientes:

- Autodescriptivos: los ensamblados incluyen metadatos que los describen. Los metadatos incluyen los tipos exportados desde el ensamblado y un *manifiesto*.
- Dependencias: las dependencias están almacenadas en el manifiesto del ensamblado. De esta forma es posible conocer exactamente las versiones exactas de otros ensamblados necesitados por el que se está desarrollando.
- Posibilidad de cargar múltiples versiones de un mismo ensamblado en la misma aplicación. Evita posibles problemas por múltiples dependencias sobre el mismo ensamblado pero con diferentes versiones.
- Aislamiento de aplicaciones a través de los dominios de aplicación. Los fallos de una aplicación no pueden afectar directamente a otras aplicaciones.
- Instalación sencilla: bastaría una simple copia.

Dentro de los ensamblados, una parte importante está formada por el manifiesto, parte de los metadatos. Un manifiesto describe el ensamblado con toda la información que se necesita para referenciarlo y listando todas sus independencias, consta de:

- Identidad.
- Lista de ficheros que pertenecen al ensamblado.
- Lista de ensamblados utilizados (número de versión y clave pública).
- Lista de solicitudes de permisos: permisos necesitados para ejecutar el ensamblado.
- Tipos exportados: no son parte del manifiesto, a menos que los tipos sean incluidos en un módulo (unidad de reutilización). La descripción del tipo es almacenada como metadatos dentro del ensamblado.

.NET y tiempo real

.NET no está diseñado para tiempo real, es una tecnología para sistemas genéricos. Los ensamblados de .NET presentan las mismas carencias para tiempo real que pueda presentar el modelo de componentes COM. A pesar de sus numerosas mejoras con respecto a COM, no se han incluido características necesarias para tiempo real.

No obstante, existen características, apropiadas para el desarrollo de aplicaciones orientadas a componentes y cuya utilización para sistemas en tiempo real puede ser

factible, como, por ejemplo, el modelo de eventos u otras características como los delegados *multicast* (punteros a funciones orientados a objetos). Puede ser interesante la utilización de C# como un lenguaje de base para la implementación de componentes, puesto que proporciona numerosos conceptos de la programación orientada a componentes de forma intrínseca al lenguaje.

También podría ser factible, dado que .NET funciona de forma similar a Java (mediante una máquina virtual), la alteración del entorno de ejecución de .NET para que soportara la construcción de aplicaciones predecibles. Así, por ejemplo, .NET incorpora un recolector de basura de forma similar a Java. Al igual que en Java, el comportamiento no predecible del recolector de basura no es recomendable para aplicaciones de tiempo real, por lo que se podría pensar en una extensión o modificación de .NET para su utilización en sistemas empotrados y/o de tiempo real.

1.2.2. El modelo de componentes de Java

JavaBeans presenta un modelo de componentes independiente de la plataforma, pero dependiente del lenguaje Java, recibiendo los componentes el nombre de *beans*. Además de los JavaBeans, que básicamente son componentes gráficos, Sun ha desarrollado los denominados Enterprise JavaBeans (EJB) [EJB, 2005].

Dentro de las principales características que las interfaces de JavaBeans ofrecen, encontramos las siguientes:

- Propiedades: un *bean* puede definir un número arbitrario de propiedades de cualquier tipo. Una propiedad es un atributo con nombre que puede afectar a la apariencia o comportamiento de un *bean*.

  Además de propiedades con simples y múltiples valores, se pueden definir propiedades de tipos *bound* y *constrained*. Las propiedades de tipo *bound* usan eventos Java para notificar a otros componentes el cambio de valor. Las propiedades de tipo *constrained*, permiten vetar cambios. Este tipo de propiedades no aparecen en muchos sistemas orientados a objetos y permiten a los desarrolladores realizar la lógica de la aplicación en una forma modular y con un mantenimiento fácil.
- Eventos: el mecanismo de eventos que utiliza JavaBeans afecta a tres interfaces relacionadas: *Event*, *EventSource* y *EventListener*.

  Las fuentes de eventos (*EventSources*) notifican a todos los consumidores de eventos (*EventListeners*) los eventos a través de objetos de tipo *Event*.

El modelo de eventos permite utilizar, además, características como *multicast* y los adaptadores de eventos, que permiten ayudar al programador para no tener que escribir manejadores para todos los eventos.

- Introspección: el uso de técnicas de reflexión permite a las herramientas el descubrimiento del soporte de ciertas características en los *beans*. Para ello se utiliza una terminología a la hora de declarar métodos, propiedades, etc.

Por su parte, los EJB se ejecutarán dentro de contenedores que aislarán al *bean* del entorno de ejecución del servidor. El contenedor automáticamente asigna hebras para los componentes y gestiona aspectos como concurrencia, seguridad y actividades relacionadas con transacciones. El entorno servidor de los EJB incluye:

- Protocolo para comunicaciones remotas.
- Servicios de directorios, seguridad, gestión del sistema y transacciones.

Beans y tiempo real

Una ventaja para el desarrollador del componente con respecto a otros modelos, puede ser la independencia de la plataforma, ya que, una vez desarrollado el componente, podría distribuirse y utilizarse en otras plataformas sin necesidad de realizar modificaciones, independientemente de si el componente va a ser utilizado en aplicaciones estándares o de tiempo real, aunque en este último caso habría que verificar si el componente cumple con sus requisitos en la nueva plataforma.

Este modelo de componentes no es adecuado inicialmente para el desarrollo de aplicaciones de tiempo real, ya que, aunque en este caso es independiente de la plataforma, no lo es del lenguaje, al depender de Java, un lenguaje que no contempla aspectos de tiempo real. Sin embargo, si se modificara el propio lenguaje Java, bien a través de la implementación de tiempo real de Java [Bollella et al., 2000], bien a través de una extensión propia; los componentes basados en este modelo sí podrían utilizarse para desarrollar un modelo de componentes que incorporara características de tiempo real.

Algunas características del modelo son interesantes para tiempo real. Por ejemplo, se podría utilizar la característica de introspección para que el entorno pudiera obtener información de los componentes (eventos producidos, eventos consumidos, etc.) y además utilizando una terminología adecuada, se podrían indicar, por ejemplo, distintas calidades de servicio (QoS) que el entorno podría detectar y utilizar.

El modelo de eventos podría ser adecuado, siempre y cuando se extendiera para permitir la incorporación de tiempo o prioridades en los mismos. Las características de *multicast* que incorpora este modelo pueden ser muy útiles para la implementación de canales de tiempo real con múltiples consumidores. Esto podría llevar a la necesidad de un servicio de planificación de eventos que garantizara el cumplimiento de las restricciones de tiempo. Una solución similar ha sido adoptada en TAO [Harrison et al., 1997].

Otra característica muy interesante que incorporan los EJB son los contenedores, que controlan aspectos como la concurrencia y se pueden extender para controlar planificación, gestión de eventos, etc. Los contenedores permiten también realizar planificación en distintos niveles. En cada contenedor, de forma transparente se contará con un planificador (que puede ser configurable) que será el encargado de su nivel, con independencia de niveles superiores o inferiores. Esta misma característica de múltiples niveles puede ser también utilizada para los eventos, de forma que un gestor de eventos por contenedor permita una planificación independiente en cada nivel.

Este modelo de contenedores/niveles permite realizar un estudio más sencillo de planificabilidad que con un solo nivel para todas las aplicaciones del sistema, esto puede ser especialmente adecuado para sistemas dinámicos.

1.2.3. El modelo de componentes CCM

CCM es el modelo de componentes propuesto por OMG (*Object Management Group*) organización que se encarga de la definición de una arquitectura de objetos y principal impulsora de CORBA (que será estudiado en una sección aparte).

Aunque CORBA proporciona a diseñadores y desarrolladores todo lo que necesitan para el desarrollo de aplicaciones, las numerosas políticas POA, transacciones, seguridad, etc. combinadas entre sí dan un elevado número de posibilidades que hacen difícil seleccionar la mejor para una aplicación en concreto.

El Modelo de Componentes CORBA, CCM (*Corba Component Model*) [OMG, 1999][Wang et al., 2001], intenta solventar estas dificultades utilizando un subconjunto de combinaciones útiles para un mejor desarrollo de aplicaciones en la parte del servidor.

CCM está formado por un número de piezas conceptuales, que juntas forman el modelo de programación para servidores. Estas piezas son:

- Los componentes, incluyendo un Modelo de Componentes Abstracto y un Marco de Trabajo de Implementación de Componentes centrado en el nuevo lenguaje para la definición de componentes (CIDL).
- El Modelo de Programación de Componente Contenedor.
- Integración con persistencia, transacciones y eventos.
- Empaquetamiento de componentes y su despliegue.
- Interconexión con EJB 1.1.
- Modelo de Metadatos de Componentes (repositorio de interfaces).

Los componentes de CCM

Un componente es un nuevo meta-tipo en CORBA. En vez de definir el componente con *interface*, los componentes se declararán con la palabra *component*. CCM ha extendido IDL con un conjunto de nuevas palabras claves necesarias para su modelo. En tiempo de ejecución, un componente es representado por una referencia al componente.

Los diversos *stubs* y *skeletons* soportados por los componentes son, en esta ocasión, referidos como ports (puertos) y cuatro de ellos tienen nombres especiales:

- Facetas (*Facets*): son las múltiples potenciales interfaces que un componente ofrece. Cuando se declara un componente, existe una interfaz principal, denominada la interfaz equivalente. Se puede heredar de esta interfaz de la forma tradicional usada en CORBA, obteniendo las denominadas interfaces soportadas. En CCM, las facetas conocidas también como interfaces proporcionadas, y que no están relacionadas con las soportadas por herencia, permiten a los clientes utilizar distintas interfaces.
- Receptáculos (*Receptacles*): son los *stubs* clientes que un componente utiliza para invocar a otros componentes. Debido a que el modelo de programación basado en componentes insiste en la reutilización vía composición, un componente podría delegar algunas de sus operaciones en otros componentes. En estos casos, un componente debe obtener la referencia a una instancia del otro componente.
- Fuentes de eventos (*Event sources*): son los nombres de los puntos de conexión que emiten eventos de un tipo especificado a uno o más consumidores interesados, o a un canal de eventos.
- Sumideros de eventos (*Event sinks*): son los nombres de las conexiones en las que los eventos de un tipo específico deben ser puestos por un suministrador o un canal.

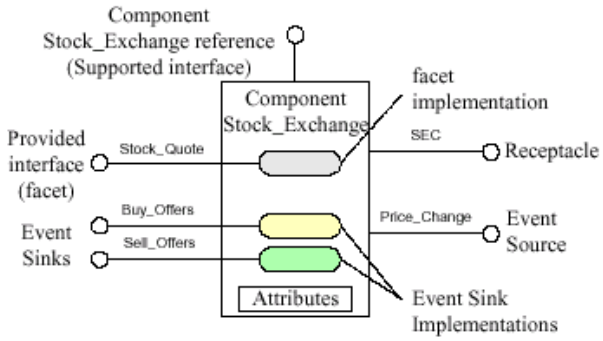


Figura 1.3. Ejemplo de componente CCM

```
interface Sell, Buy;

// Define an equivalent, supported interfaces
component Stock_Exchange supports Sell, Buy {
  provides Stock_Quote;     // Facet

  consumes Buy_Offers;      // Event Sinks
  consumes Sell_Offers;

  publishes Price_Change;   // Event Source
  uses SEC;                 // Receptacle
    ...                     // Other definitions
};
```

Figura 1.4. Definición de componente CCM

Otras nuevas características del modelo incluyen:

- Claves primarias: valores que identifican a los componentes unívocamente.
- Atributos y configuración: expuestos para permitir la configuración del componente.
- Interfaces *Home*: que proporcionan funcionamiento de factoría y operaciones de búsqueda de componentes.

El marco de trabajo para implementación de componentes

CCM proporciona un gran número de interfaces para soportar la estructura y funcionalidad de los componentes. La implementación para muchas de estas interfaces puede ser generada automáticamente. El *CORBA Component Implementation Framework* (CIF) está diseñado para desarrollar estas tareas y que los programadores no tengan que preocuparse de ellas. Por ejemplo, la gestión del ciclo de vida y estados de los componentes podría ser resuelta con este marco de trabajo.

CCM define un lenguaje declarativo, el *Component Implementation Definition Language* (CIDL) que permite describir implementaciones y estados persistentes de los componentes. CIF usa las descripciones CIDL para generar código que automatiza

comportamientos de los componentes tales como navegación, consulta de identidades, activación y gestión del estado.

El modelo de programación basado en contenedores

Un contenedor es el entorno de ejecución que es ofrecido por CCM a los componentes. Los contenedores encapsulan la implementación de un componente y usan un conjunto de APIs para acceder al entorno de ejecución, facilitando así el desarrollo y/o configuración de aplicaciones CORBA. Se proporcionan las siguientes características:

- Activación/Desactivación de implementaciones de componentes para preservar recursos limitados del sistema, tales como memoria principal.
- Proporcionar un nivel de adaptación para cuatro servicios comúnmente usados: Transacciones, Persistencia, Seguridad y Notificación. Este marco libera a los clientes de tener que localizar estos servicios.
- Nivel de adaptación para *callbacks* que el contenedor y el *Object Request Broker* (ORB) utilizan para informar al componente sobre eventos interesantes tales como mensajes de los servicios anteriormente mencionados.
- Gestión de las políticas POA para determinar la creación de referencias a los componentes.

CCM y tiempo real

Al igual que los modelos anteriores, CCM tampoco está preparado inicialmente para aplicaciones de tiempo real, ya que, no incorpora la posibilidad de indicar restricciones temporales (en este caso además es muy probable que los diferentes componentes estén incluso en máquinas distintas, con el grado de impredecibilidad que ello aporta). Y además, presenta los mismos problemas del estándar CORBA, que no permite acotar el tiempo de ejecución de una invocación remota.

Este modelo presenta múltiples interfaces mediante las facetas. Con esta característica se pueden conseguir distintas QoS, interfaces funcionales y especiales para tiempo real, distintas interfaces para una utilización estática o dinámica (por ejemplo, mediante una prueba de aceptación) de los componentes y la posibilidad de implementar un mecanismo de reconfiguración dinámica.

Además se contempla un modelo basado en eventos con fuentes y sumideros, que como en los *beans* de Java, podría ser extendido de forma que incorpore restricciones temporales y/o prioridades en la transmisión y recepción de los eventos.

La característica de activación/desactivación de implementaciones de componentes que incluyen los contenedores de CCM puede utilizarse para tener implementaciones degradadas de los componentes (distintas QoS) e incluso para reconfiguración dinámica. En el modelo actual CCM se suministran niveles de adaptación para cuatro servicios muy comúnmente usados, como son Transacciones, Persistencia, Seguridad y Notificación. Sería aconsejable incorporar un quinto nivel de adaptación para los servicios de tiempo real, de forma que esto facilitara el desarrollo de aplicaciones con CCM y tiempo real.

1.2.4. Tendencias actuales sobre componentes y tiempo real

Existen algunos trabajos intentando relacionar los componentes con tiempo real. Para el estudio de la composicionalidad de componentes con características de tiempo real, el trabajo presentado en [Hooman y Van Roosmalen, 2000] propone un modelo a seguir para el desarrollo de aplicaciones de tiempo real independiente de la plataforma y basándose en el concepto de componentes de una forma genérica, como fragmentos de código software. Hooman propone un modelo de programación en tiempo real con un método de verificación formal de programas con anotaciones de tiempo. La idea básica del modelo de programación ideado es que es posible limitar las restricciones de tiempo programadas a las que aparecen en las especificaciones. Sin embargo, este modelo presenta dos inconvenientes:

- El concepto de componente que presenta se limita a fragmentos de código. Si bien esto puede ser correcto, puede ser algo difícil de utilizar en un sistema completo.
- Las posibilidades que ofrece al programador en caso de que el sistema no sea planificable se limitan a cambiar la plataforma o su programa.

El modelo formal descrito por T.A.Henzinger [Henzinger, 2000] propone una forma estructurada y formal para el desarrollo de software y hardware interactuando a través de componentes en tiempo real, y realizando un estudio de las posibles formas de composición de componentes.

Otro trabajo formal es el desarrollado por [Sifakis, 1999] donde se presenta la construcción de sistemas reactivos con cuestiones acerca de su composicionalidad, y en

particular en la forma en que sistemas sin tiempo bien construidos pueden ser extendidos para su utilización con requisitos temporales.

Dentro del ámbito de los sistemas distribuidos, están comenzando a aparecer los primeros resultados para componentes de tiempo real: herramientas, entornos de ejecución, métodos para expresar restricciones, etc. Los resultados son todavía muy preliminares y no se han establecido aún modelos de referencia como en el caso de los modelos de componentes estándares.

Los trabajos estudiados pueden agruparse en tres grandes grupos: plataformas de ejecución (con componentes u objetos), estudios sobre planificabilidad e indicación de restricciones y finalmente estudios sobre estándares ya consolidados como puedan ser Java o CORBA (con las mejoras que estos estudios supondrían para los modelos de componentes).

En el trabajo desarrollado en [Villela et al., 2001] se presenta un marco de trabajo con un conjunto de herramientas para el desarrollo de componentes distribuidos en tiempo real. Las herramientas incluyen plantillas de componentes, diagramas de despliegue para la configuración de la aplicación en los distintos nodos que forman el sistema, generadores de código, un repositorio de componentes, etc. Sin embargo, no se garantiza el cumplimiento de las restricciones de tiempo real por estar en un estado muy preliminar de desarrollo; se hace un especial énfasis en las herramientas que permitirán desarrollar las aplicaciones distribuidas.

VERTAF es un marco de trabajo para aplicaciones orientadas a objetos en sistemas empotrados de tiempo real presentado en [Hsiung et al., 2002]. Dispone de cinco herramientas: Implementador, Modelador, Planificador, Verificador y Generador: El componente Implementador especifica los objetos de diseño. El componente Modelador pasa los objetos de una aplicación, que se ha diseñado orientada a objetos, a procesos, donde cada uno es una tarea de tiempo real. El componente Planificador comprueba la planificabilidad de las tareas, usando algún algoritmo de planificación. El componente Verificador comprueba que es factible, comprobando si cumple las restricciones de tiempo; y finalmente, el componente Generador genera el código de la aplicación.

VERTAF pretende incorporar la verificación en el proceso de diseño, ahorrando de esta forma costes. Utiliza para ello técnicas de comprobación de modelos (*model cheking*), modelando los sistemas como conjuntos de autómatas con tiempo. Otros aspectos interesantes de VERTAF son la utilización de puertos de entrada, salida y

configuración; además de la utilización de métodos disparados por tiempo y por eventos.

El trabajo desarrollado en [Brinkschulte et al., 2002] presenta la arquitectura OSA+: un middleware con soporte para el diseño de sistemas en tiempo real heterogéneos y que permite el uso de pequeños microprocesadores como nodos (especialmente pensado para sistemas empotrados).

La entidad principal del middleware de OSA+ son los servicios, comunicándose unos con otros a través de trabajos, que están formados por órdenes (qué y cuando) y resultados. El middleware dispone de una parte para servicios básicos y otra, opcional, para extensión de servicios, haciéndolo flexible y escalable.

La planificación es realizada en base a los trabajos. OSA+ puede garantizar el cumplimiento de restricciones sólo si todas las capas existentes por debajo del middleware pueden manejar sus funciones en tiempo real: El hardware debe garantizar el cumplimiento de WCETs y el sistema operativo debe ser de tiempo real.

Se presenta además el proyecto Komodo con la implementación de OSA+ en Windows NT y VxWorks; utilización de microprocesadores PicoJava con cuatro hebras como máximo y manejo, tanto de prioridades como de eventos y la adaptación de una máquina virtual java.

En [Pfeffer et al., 2002] los autores continúan con la arquitectura OSA sobre microprocesadores Java, exponiendo además las posibles formas de planificación que utilizan directamente sobre hardware: RMA, EDF, GP (porcentaje garantizado de CPU).

Finalizando con las plataformas de ejecución, en [Yen et al., 2002] se propone un mecanismo integrado basado en componentes para el desarrollo de software empotrado; haciendo especial énfasis en dos problemas identificados por los autores: identificación de componentes junto con su recuperación, y composición. Sin embargo, este trabajo no está centrado en tiempo real, aunque la sintaxis presentada es interesante y puede ser ampliada para componentes en tiempo real.

En el trabajo presentado en [Yau y Zhou, 2002] se utilizan modelos desde las primeras fases del diseño de una aplicación, hasta su implementación; pensando que la utilización de modelos facilita el desarrollo de aplicaciones de tiempo real frente a la utilización de los resultados de investigaciones teóricas que se centran más en modelos basados en tareas en lugar de en modelos basados en objetos o componentes como suelen ser gran parte de las aplicaciones actuales. Además de los modelos, se utilizan también aspectos para indicar el código de planificación, pudiendo realizarse análisis de

planificabilidad durante el diseño. Este trabajo está centrado pues en planificabilidad y orientación a aspectos.

Para diseñar sistemas distribuidos de tiempo real predecibles se requieren especificaciones de los componentes en el dominio del tiempo, además de las interfaces funcionales. El trabajo desarrollado en [Kopetz y Obermaisser, 2002] trata la especificación temporal de interfaces y establece los principios de la composición.

Se presenta una nueva interfaz añadida a los componentes denominada *Temporal firewall*: interfaz para intercambios unidireccionales de información de estado entre emisor y receptor. Son indicados además los cuatro principios que debe cumplir una arquitectura para que pueda realizarse una composición correcta:

- Desarrollo independiente de nodos.
- Estabilidad de los servicios primordiales.
- El sistema de comunicación debe ser eficiente.
- Las réplicas deben ser deterministas.

La interfaz presentada por los autores soporta composición temporal y garantiza el cumplimiento de los plazos.

El trabajo presentado en [Wellings et al., 2002] intenta demostrar cómo se puede tratar el problema de planificación dinámica en el análisis de peores tiempos de ejecución con un mínimo de anotaciones. Este trabajo puede ser interesante como punto de partida para la realización de herramientas.

Sobre estándares normales de componentes, tanto Java como CORBA están siendo ampliamente utilizados para el desarrollo de sistemas distribuidos, de consumo electrónico, tiempo real, etc.

La utilización de Java podrá aumentar cuando las primeras implementaciones de Java para Tiempo Real sean estables; por su parte, en el ámbito de CORBA podemos destacar a TAO [Levine et al., 1998], una implementación de CORBA diseñada para tiempo real y con una amplia comunidad de usuarios.

Un ejemplo de utilización de Java para sistemas empotrados electrónicos, lo encontramos en un trabajo de la Universidad nacional de Taiwan [Fu et al., 2000]. En este trabajo existen tres componentes por nodo: sistema de tiempo real multitarea, controlador y puentes entre diferentes sistemas. En el trabajo se tratan también temas de planificación, pero centrados en el nivel de un nodo concreto, y no de forma distribuida.

En [Nakamoto et al., 2002] se utiliza CORBA en sistemas empotrados para satélites. Los autores, sin embargo, no consideran RT-CORBA sino que utilizan un CORBA empotrado propio.

Ambos trabajos previos son interesantes en el sentido de que muestran cómo los diseñadores e implementadores de aplicaciones resuelven las limitaciones de los modelos estándares y consiguen utilizar estos modelos en aplicaciones de tiempo real. El inconveniente es que cada uno de estos trabajos utiliza soluciones propias, con las dificultades de reutilización que ello conlleva. De ahí, el interés que supondría contar con un estándar para la realización de aplicaciones con componentes para sistemas de tiempo real empotrados.

Un trabajo muy interesante que mezcla Java y CORBA, es la implementación de CORBA, Zen [Klefstad et al., 2002] donde se presenta un ORB escrito en Real-Time Java, aunque aún en sus primeros estados. Este ORB desarrollado por los autores de TAO puede ser un referente a tener en cuenta en un futuro próximo.

Una última aportación es la realizada por [Singh et al., 2002] donde se presenta una propuesta basada en aspectos para desarrollar código de sincronización para sistemas distribuidos. Se comentan además detalles de implementación, destacando su implementación sobre el servicio de eventos de tiempo real de TAO.

1.3. CORBA y RT-CORBA

Las redes de ordenadores son típicamente heterogéneas. Los orígenes de esta heterogeneidad están debidos a multitud de factores: avances tecnológicos progresivos, factores económicos, necesidades de las aplicaciones, etc. Esta heterogeneidad hace aún más complicado el desarrollo de aplicaciones distribuidas al tener que considerar problemas tales como el empaquetamiento de datos, sistemas operativos y lenguajes de programación diferentes, tecnologías de redes, etc.

CORBA (*Common Object Request Broker Architecture*) nace buscando modelos y abstracciones que sean independientes de la plataforma, que sirvan para resolver una amplia gama de problemas y que oculten tanto como sea posible la complejidad de los sistemas subyacentes al mismo tiempo que se mantiene el rendimiento en la mayor medida posible.

CORBA es una arquitectura para software distribuido desarrollada por el grupo OMG (*Object Management Group*), organización constituida en 1989 por un gran número de empresas (más de 800 actualmente) para la definición de una plataforma sobre la que desarrollar aplicaciones distribuidas cuyos componentes pueden estar escritos en lenguajes diferentes y ejecutarse sobre máquinas de distinto tipo, ofreciendo una capa homogénea para el desarrollo de aplicaciones que se ejecutan en sistemas heterogéneos. El objetivo principal de CORBA es ofrecer una capa **homogénea** para el desarrollo de aplicaciones que se ejecutan en **sistemas distribuidos heterogéneos**.

La predecibilidad temporal es un aspecto esencial en el desarrollo de aplicaciones distribuidas de tiempo real. CORBA, en este sentido, presenta una serie de carencias que imposibilitan su utilización en este tipo de sistemas.

La principal carencia de CORBA para aplicaciones de tiempo real es la imposibilidad de establecer un límite máximo en una invocación: cuando un cliente CORBA realiza una petición, el tiempo máximo que puede tardar esa petición en ser realizada, no está acotado. Esto es debido a múltiples factores: compartición de recursos en el lado del cliente y del servidor, compartición de recursos de red, etc. CORBA estándar no define cómo deben comportarse los ORBs ante clientes o servidores de distintas prioridades, por lo que no hay ninguna garantía en este sentido.

El objetivo de la especificación **Real-time CORBA** (RT-CORBA) es definir qué necesita CORBA para poder ser predecible manteniendo en la mayor medida posible el espíritu de diseño CORBA.

Específicamente, RT-CORBA proporciona políticas y mecanismos para controlar y/o configurar los siguientes elementos:

- Recursos del procesador: manejo de prioridades, hebras, sincronizadores, etc.
- Recursos de comunicación: añade propiedades a los protocolos de comunicación y mecanismos adicionales.
- Recursos de memoria: mecanismos de encolado para peticiones, controlando el uso de memoria.

RT-CORBA no puede hacer *"milagros"*, sino que proporciona unos mecanismos para el desarrollo de aplicaciones predecibles, pero dejando la responsabilidad final sobre el sistema operativo en el que se ejecutan los aplicaciones, por lo que si RT-CORBA se ejecuta en sistemas operativos que no son de tiempo real, el comportamiento de la aplicación tampoco será de tiempo real.

Las versiones iniciales de RT-CORBA presentaban como única restricción el estar ligadas a planificación basada en prioridades fijas. Estas limitaciones son superadas en las últimas versiones que permiten planificación dinámica [OMG, 2005].

1.3.1. Fundamentos de CORBA

CORBA proporciona una infraestructura de comunicación para aplicaciones basadas en objetos distribuidos, análoga a un bus hardware, con objetos comunicándose entre sí (Figura 1.5) [Henning y Vinoski, 2001]. La abstracción para el desarrollador consiste en tener objetos CORBA que exponen métodos que pueden ser invocados, a su vez, por otros objetos CORBA.

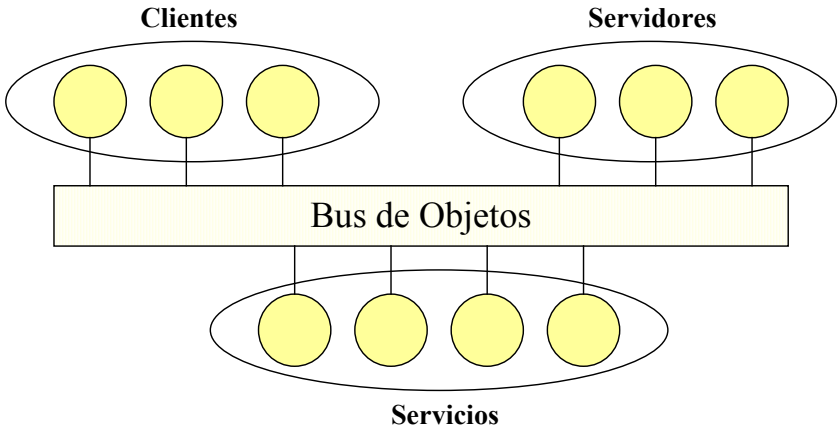


Figura 1.5. Abstracción de objetos de CORBA

La arquitectura OMA

La arquitectura OMA (*Object Management Architecture*) es la arquitectura propuesta por OMG para desarrollar aplicaciones distribuidas, siendo CORBA su núcleo. Esta arquitectura se basa en dos modelos: el modelo de objetos, que define qué es un objeto y el modelo de referencia, que caracteriza las interacciones entre éstos.

El modelo de objetos define a un objeto CORBA como entidad encapsulada que tiene una identidad distintiva inmutable la cual proporciona servicios accesibles por medio de una interfaz bien definida. Los clientes acceden a estos servicios mediante el envío de solicitudes al objeto, quedando los detalles de comunicación ocultos.

El modelo de referencia proporciona un conjunto de categorías de interfaces, relacionadas conceptualmente a través de un ORB (*Object Request Broker*):

- Los *servicios de objetos* (*CORBAservices*) son interfaces independientes del dominio de la aplicación usados por la mayoría de programas, como por ejemplo, el Servicio de Nombres o el Servicio de Eventos.
- Las *interfaces de aplicaciones*, son diseñadas por cada aplicación concreta, no están estandarizadas y OMG no interviene en su desarrollo.
- Las *interfaces de dominios* (servicios verticales) juegan un papel similar al de los servicios de objetos pero orientados a dominios de la aplicación específica, como, por ejemplo, telecomunicaciones o medicina.
- Los *servicios comunes* (servicios horizontales) son servicios aplicables a cualquier dominio de la aplicación como, por ejemplo, formatos para intercambio de documentos, interfaces gráficas, gestión de tareas, gestión de sistemas, etc.

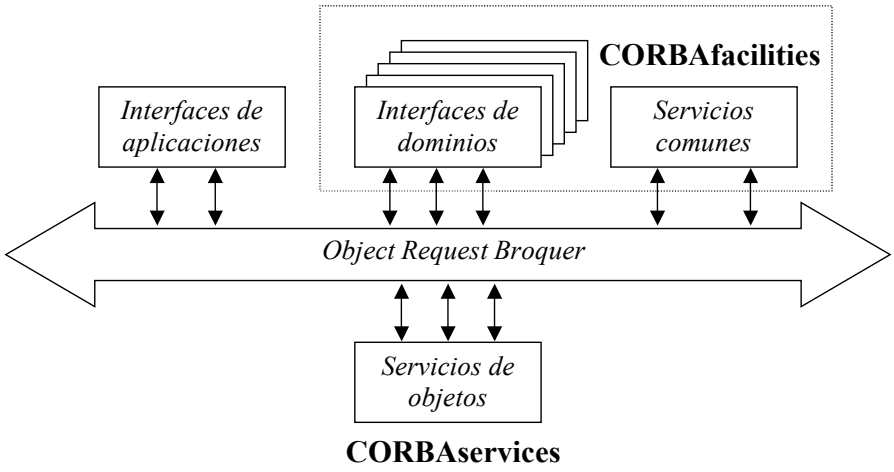


Figura 1.6. Arquitectura OMA

Versiones de CORBA

Desde las primeras versiones de CORBA, se han ido presentando diferentes revisiones. La primera versión, CORBA 1.1, presentada por el OMG en diciembre de 1991 no permitía la interoperabilidad entre ORBs de distintos fabricantes. CORBA 1.2 fue presentado en junio de 1994, definiendo nuevos servicios. En 1996 sale a la luz CORBA 2, una nueva versión que daba la posibilidad de interoperabilidad entre ORBs mediante dos protocolos: *General Inter-ORB Protocol* (GIOP), e *Internet Inter-ORB Protocol* (IIOP), y la revisión de ésta: CORBA 2.2 introdujo el *Portable Object Adapter* (POA). La versión 2.3 presentó a RT-CORBA como una extensión. Actualmente se encuentra en un avanzado estado de desarrollo CORBA 3 que incorpora numerosos

nuevos conceptos como los componentes CCM. No obstante, aún no existen implementaciones que incorporen CORBA 3 por completo, si bien sí añaden algunas de sus partes más importantes.

RT-CORBA 1.0 apareció como un conjunto de extensiones a CORBA 2.3 y a la especificación CORBA de mensajes [OMG, 1998]. Posteriormente, fue combinada con la especificación del estándar CORBA en CORBA 2.5. La versión más actual hasta la fecha es la número 1.2 para planificación estática, presentada en [OMG, 2005], incluyendo el concepto de *hebras distribuidas* (*Distributable Threads*).

Invocación de operaciones

En la Figura 1.7 puede verse el flujo de ejecución básico entre un cliente y un objeto CORBA en una invocación genérica del tipo: `result = object->operation(args).`

El primer paso para realizar la invocación es obtener una referencia del objeto destino. Esta referencia puede ser obtenida de diferentes formas (ej: servicio de nombres, factorías, etc.). Una vez obtenida la referencia, el cliente procede a invocar la operación sobre el objeto destino. Para ello, utiliza una clase intermedia contenida en el *stub* y que representa al objeto destino en el espacio de direcciones del cliente (ofrece una interfaz similar al objeto destino con los mismos métodos). La clase contenida en el *stub* es responsable del empaquetamiento de los parámetros y de la comunicación con el objeto destino, utilizando para ello las funciones que el ORB le proporciona. Los datos son transmitidos al ORB del servidor utilizando algún protocolo que instancie al protocolo de comunicaciones GIOP, como por ejemplo IIOP.

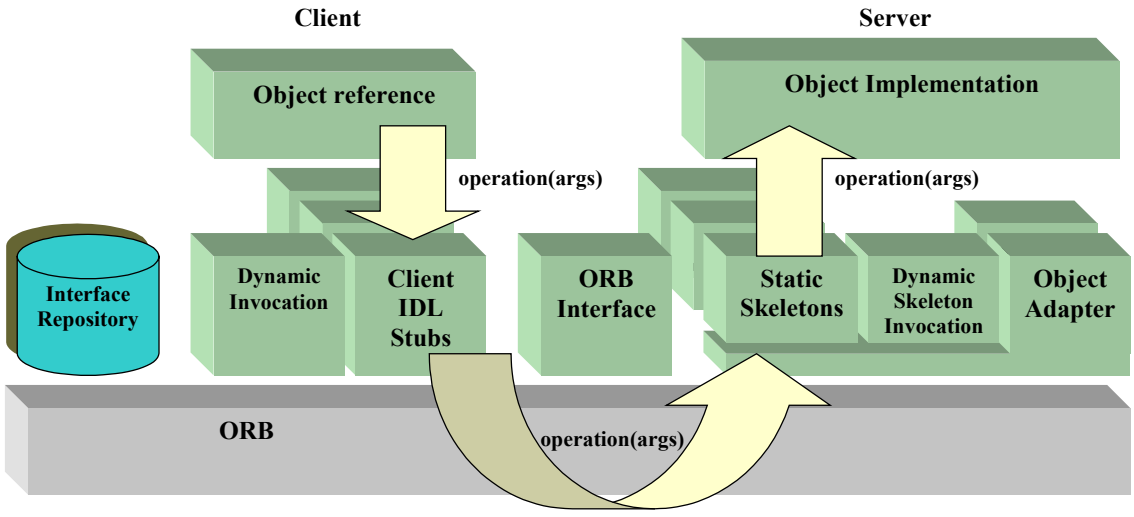


Figura 1.7. Invocación general CORBA

En el espacio de direcciones del servidor, la invocación es capturada por el ORB, que usa la información contenida en la invocación para localizar al adaptador de objetos al que pertenece el objeto destino. El adaptador de objetos es la entidad responsable de conectar objetos CORBA con objetos reales (sirvientes). Finalmente, la invocación llega a la clase sirviente, ejecutándose y transmitiendo resultados al cliente.

Tipos de invocaciones

Existen dos formas de realizar y tratar invocaciones CORBA según se utilice código generado automáticamente o se construyan invocaciones de forma dinámica.

- **Invocación estática:** se forma a partir de la interfaz IDL, generándose los *stubs* y *skeletons*, que serán compilados con las aplicaciones y accediendo así a los objetos remotos. Un *stub* es la función o clase que permite que un cliente invoque una operación remota como si fuese local (*proxy*), y un *skeleton* es la función o clase que permite que una invocación remota recibida por un servidor sea enviada al sirviente adecuado. Los *stubs* y los *skeletons* ocultan a los clientes y a los programadores de los objetos, los detalles de las comunicaciones.
- **Invocación dinámica:** se basa en construir y cursar una invocación en tiempo de ejecución, y no se requiere información procedente de la fase de compilación ya que la creación de las solicitudes en tiempo de ejecución requieren el acceso a los servicios que suministren información sobre las interfaces y los tipos de datos, y esta información se puede obtener del denominado *Repositorio de interfaces* (*Interface Repository*).

Puede establecerse una segunda clasificación de las invocaciones atendiendo a la sincronización:

- **Peticiones síncronas:** el cliente queda bloqueado hasta recibir la respuesta del objeto CORBA.
- **Peticiones asíncronas diferidas:** el cliente realiza la petición y continua su ejecución, preguntando posteriormente por la respuesta.
- **Peticiones *oneway*:** son de sentido único (sin parámetros de salida ni respuesta desde el servidor) y se realizan sin garantías, pudiendo ser descartadas por el ORB sin que el cliente tenga constancia de ello.

Lenguaje de Definición de Interfaces: IDL

Para realizar las diferentes invocaciones de una operación sobre un objeto distribuido, el cliente debe conocer la interfaz ofrecida por el objeto. Ésta se compone de un conjunto de operaciones y de los tipos de datos que reciben y devuelven dichas operaciones. También pueden incluirse atributos cuyos valores se pueden consultar o modificar.

Las interfaces en CORBA se definen mediante el Lenguaje de Definición de Interfaces (IDL), que es un lenguaje declarativo (los objetos y las aplicaciones no se implementan en IDL). El propósito de IDL es simplemente definir las interfaces de los objetos de forma independiente a un lenguaje de programación concreto (aunque IDL está inspirado en C++ y Java), siendo esto fundamental para cumplir el objetivo de heterogeneidad de CORBA.

IDL establece el contrato entre el cliente y el servidor, describiendo los tipos e interfaces de objeto utilizados en una aplicación de manera independiente del lenguaje de implementación, ya que no importa si el cliente y el servidor están escritos en diferentes lenguajes.

El punto de partida para la implementación de un sistema CORBA está formado por interfaces IDL que indican los puntos de interoperabilidad entre sistemas. El compilador IDL convierte las especificaciones independientes en especificaciones de tipos e interfaces en un lenguaje concreto de implementación.

Cada objeto CORBA tiene una única interfaz, aunque puede haber múltiples objetos con la mismo interfaz, equivalente a las instancias, salvo que se pueden encontrar en espacios de direcciones separados.

```
module Control_Temperatura {
    interface termometro {
        typedef Temp       short;
      readonly attribute estado;

      struct Error {
          Temp valor;
          Temp MAX,MIN;
      }

        exception Temp_Erronea {Error detalles};
        void leer_temp(out Temp actual)
          raises(Temp_Erronea);
       void set_alarm(in Temp MAX,in Temp Min)
          raises(Temp_Erronea);
      }
};
```

Correspondencias a lenguajes

IDL no puede utilizarse para escribir directamente aplicaciones. En su lugar, los compiladores de IDL generan código que clientes y servidores utilizan. El código generado presenta las particularidades propias del lenguaje destino, existiendo diferentes traducciones o equivalencia (“mapeos”*)* desde las interfaces al lenguaje destino. Estas traducciones son definidas por OMG y existen, por ejemplo, para lenguajes como C, C++, Java, Ada, Smalltalk o COBOL. Existen además traducciones no oficiales para otros lenguajes como Eiffel, Perl, Tcl, Objective-C y Python.

Adaptadores de objetos

Los adaptadores de objetos constituyen el medio de enlazar los sirvientes con el ORB, siendo una entidad que adapta la interfaz de un objeto a una interfaz diferente, que es la que espera el cliente. A través del mecanismo de delegación se permite que un cliente invoque servicios de un objeto sin conocer cuál es la interfaz real del objeto.

En CORBA, los adaptadores de objetos cumplen tres funciones: crear las referencias a los objetos, asegurar que cada objeto destino está representado por un sirviente y recibir las invocaciones cursadas por el ORB en el lado del servidor y dirigirlas a los sirvientes que encarnan a los objetos destino. Si no existieran los adaptadores de objetos, el ORB debería suministrar servicios adicionales, aumentando su complejidad y limitando el número de posibles formas de implementar los sirvientes.

En C++ los sirvientes son instancias de clases y se definen típicamente a partir de los *skeletons* (clases de C++). La implementación de las operaciones se lleva a cabo redefiniendo las funciones virtuales puras de la clase base del *skeleton*, posteriormente los sirvientes se registran en el adaptador de objetos pudiendo recibir peticiones desde los clientes.

La versión CORBA 2.2 introdujo el denominado *Adaptador de Objetos Portable (Portable Object Adapter*, *POA)* el cual contempla una amplia gama de interacciones entre los objetos CORBA y los sirvientes escritos en lenguajes concretos, manteniendo la portabilidad.

El desarrollador puede crear distintos POA que se adecuen mejor a las características de su aplicación. Estas características denominadas *políticas (policy)* se especifican cuando el POA es creado (no pueden ser cambiadas a posteriori) y son compartidas por todos los objetos del POA.

El protocolo Inter-ORB (GIOP)

En CORBA 2.0 se introdujo una arquitectura para interoperabilidad entre ORBs basada en el *General Inter-ORB Protocol* (GIOP), protocolo abstracto que especifica cómo deben comunicarse los ORBs sobre cualquier medio de protocolo orientado a la conexión. Cualquier fabricante que quiera desarrollar un ORB debe respetar este protocolo si quiere que sus aplicaciones sean compatibles con las de otros fabricantes. Para ello, se crean instancias de este protocolo, siendo una de las más usadas el protocolo *Internet Inter-ORB Protocol* (IIOP), que especifica cómo implementar GIOP sobre TCP/IP.

La interoperabilidad entre ORBs requiere, además que las referencias de los objetos, tengan un formato estándar siendo las referencias opacas para las aplicaciones, pero conteniendo información para el ORB con el fin de poder establecer las comunicaciones entre los clientes y los servidores. Este formato se denomina IOR (*Interoperable Object Reference*), suficientemente flexible como para admitir una amplia gama de protocolos. En el caso de IIOP, una IOR contiene: el nombre de la máquina, el puerto TCP/IP y un identificador del objeto.

La especificación del GIOP consta de los siguientes elementos:

- Suposiciones sobre el transporte: orientado a conexión, conexiones *full-duplex*, conexiones simétricas, transporte fiable, abstracción de flujo de bytes e indicación de pérdida abrupta de una conexión.
- Representación de datos común (*Common Data Representation, CDR)*: GIOP define un formato para el envío por la red de cada tipo de datos IDL, de forma que el emisor y el receptor estén de acuerdo con el formato binario de los datos.
- Formatos de los mensajes: GIOP define ocho tipos de mensajes que utilizan los clientes y los servidores para comunicarse, destacando los mensajes *Request* y *Reply* para el envío y recepción de peticiones.

Servicios CORBA

OMG define una serie de servicios genéricos comúnmente utilizados por los desarrolladores para el desarrollo de aplicaciones distribuidas. Todos estos servicios ofrecen una serie de interfaces al programador, que los utiliza como a cualquier objeto

CORBA normal. La implementación de los servicios no es obligatoria para un ORB, pudiendo cada ORB implementar los que desee. No obstante, al funcionar todos mediante el protocolo IIOP, los servicios de un determinado ORB deben poder ser utilizados por cualquier otro ORB y con aplicaciones escritas en cualquier otro lenguaje.

- **Servicio de Nombres**: El servicio de nombres es uno de los más utilizados en el desarrollo de aplicaciones CORBA. La función principal del servicio de nombres es vincular un nombre a los objetos y permitir localizar el objeto a partir de su nombre. Las dos operaciones básicas del servicio de nombres son la vinculación (asociación de un nombre a un objeto) y la resolución (obtención de una referencia de un objeto a partir de su nombre).

  Usualmente, un servidor CORBA vinculará varios objetos CORBA en el servicio de nombres. Posteriormente, las aplicaciones clientes obtendrán referencias de estos objetos a partir de su nombre, y podrán utilizar los servicios ofertados por los objetos registrados.
- **Servicio de Eventos**: El servicio de eventos proporciona un sistema de comunicación que desacopla a los participantes. Con este servicio las aplicaciones utilizan los roles de consumidor o productor de eventos, siendo un evento un *envío* (asíncrono) por parte de los productores de cualquier tipo definible en una interfaz.

  En este servicio, los productores de eventos crearán nuevos eventos que son *consumidos* por todas aquellas aplicaciones que se hayan registrado como consumidores en el servicio de eventos.
- **Servicio de Notificación**: Similar al servicio de eventos , añadiendo transmisión de evento con nuevas posibilidades: suscripción en eventos determinados, QoS, etc.
- **Servicio de *Trading***: Este servicio es similar al servicio de nombres. Pero en este caso no se especifica un nombre, sino una serie de características del objeto deseado.
- **Servicio de Transacciones:** Proporciona las interfaces necesarias que combinan el paradigma de transacción, y el paradigma de objeto. Conjuntamente se usan para la resolución de problemas de negocios de tipo comercial.
- **Servicio de Ciclo de vida:** Define servicios y convenciones para crear, borrar, copiar y mover objetos CORBA.
- **Servicio de Concurrencia:** Media el acceso concurrente a un objeto.

- **Servicio de Externalización:** Define protocolos y convenciones para la externalización (guardar el estado de un objeto en un *stream* de datos) e internalización de los objetos.
- **Servicio de Licencia:** Define interfaces que soportan la gestión de las licencias del software.
- .**Servicio de Persistencia:** Interfaz que representa la información persistente como almacenes de objetos.
- **Servicio de Propiedad:** Proporciona la capacidad para asociar dinámicamente valores de nombres con objetos sin IDL.
- **Servicio de Relaciones:** Permite a las entidades y relaciones ser representadas explícitamente.
- **Servicio de Seguridad:** Marco de trabajo para la seguridad en CORBA.
- **Servicio de Tiempo:** Permite a un usuario obtener el tiempo actual con un error determinado.

Algunas implementaciones CORBA añaden sus propios servicios, que no son estándares; como, por ejemplo, el servicio de eventos de tiempo real de TAO, que es ampliamente utilizado en sistemas basados en TAO.

Principios del diseño CORBA

Los principios del diseño CORBA radican en la separación de interfaz e implementación, con lo cual los clientes tienen conocimiento sólo de las interfaces. Un segundo punto es la transparencia de localización, ya que la utilización de un servicio de un objeto es ortogonal a la localización del objeto. Y, por último, un tercer punto es la transparencia de acceso, permitiendo, al invocar las operaciones de objetos, la independencia de implementación, arquitectura, sistema operativo, protocolo y red.

En general, el desarrollo de una aplicación CORBA consiste en obtener los ejecutables del *cliente* y del *servidor*.

Los principales pasos a seguir para el desarrollo de una aplicación CORBA son los siguientes:

- Determinación de los objetos de la aplicación y definir sus interfaces en IDL.

- Compilación de las definiciones en IDL para obtener los *stubs* y los *skeletons*. Seguidamente se declaran e implementan los sirvientes que van a encarnar a los objetos CORBA.
- Escritura de los módulos principales del servidor y del cliente.
- Finalmente, se compilan y enlazan los ficheros de implementación del servidor con los *stubs* y los *skeletons* para obtener el fichero ejecutable del servidor o del cliente, pudiendo ya ejecutarse la aplicación.

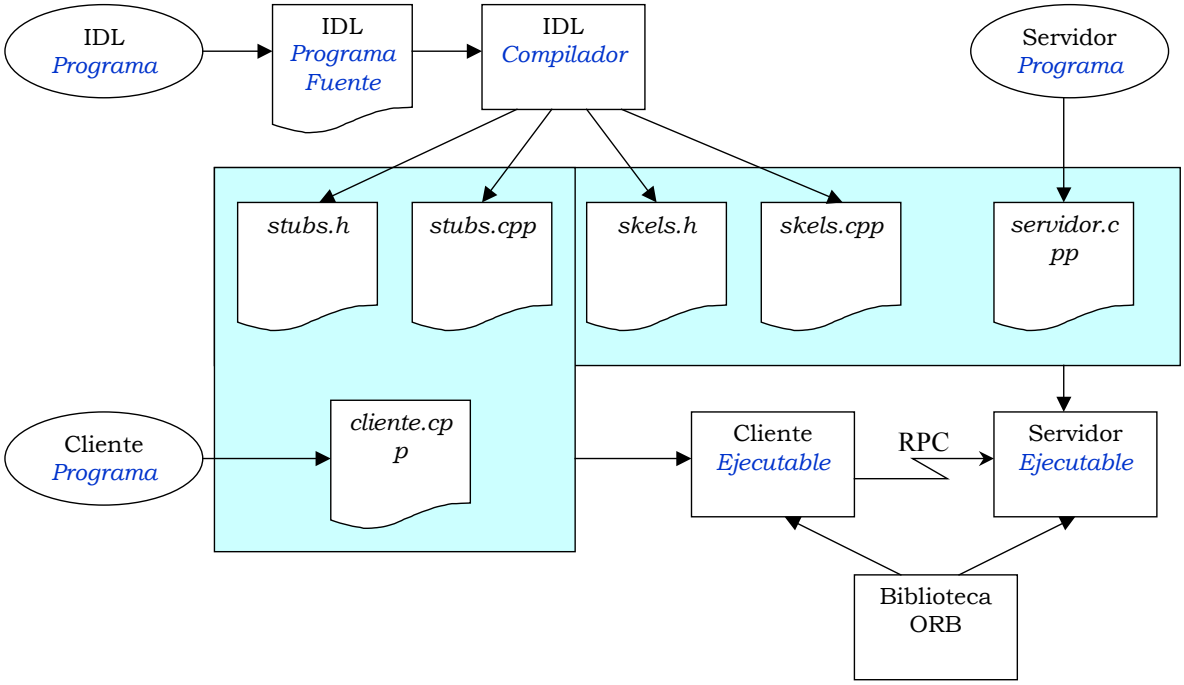


Figura 1.8. Desarrollo de una aplicación CORBA

1.3.2. Arquitectura CORBA de tiempo real

La Figura 1.9 muestra los principales elementos de la arquitectura RT-CORBA encuadrados dentro de CORBA. Las entidades sombreadas de la figura son las extensiones de CORBA para tiempo real. En ellas se pueden distinguir:

- El ORB de tiempo real, el cual aporta las interfaces necesarias para crear y destruir los objetos para la programación de tiempo real.
- La interfaz `PriorityMapping`, utilizada para tener un esquema de prioridad independiente de las plataformas donde residen las aplicaciones.

- La interfaz `RTPOA`, proporciona los métodos necesarios para poder llegar a configurar los POA de tiempo real.
- La interfaz `RTCORBA::Current`, aporta control sobre las prioridades de las hebras.
- El servicio de planificación, opcional en las implementaciones, asegura la planificabilidad de las invocaciones CORBA.

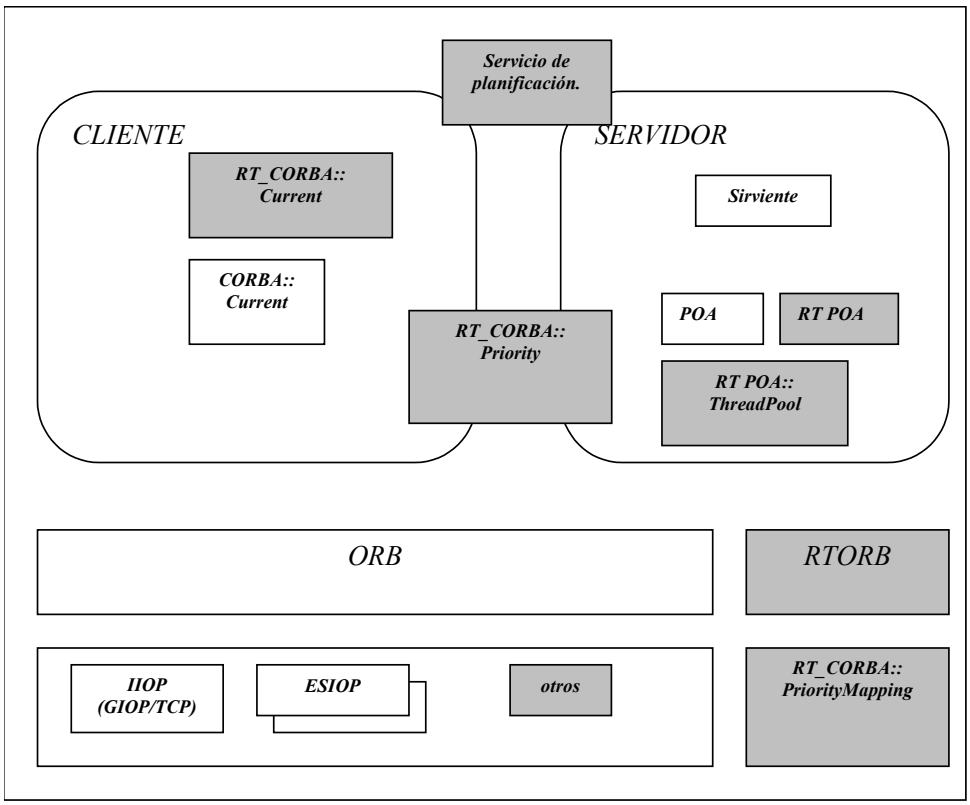


Figura 1.9. Arquitectura RT-CORBA

1.3.3. Gestión de prioridades

Los ORB estándares no definen mecanismos para indicar la prioridad a la que debe ejecutarse una petición. Esta es una característica necesaria para el desarrollo de aplicaciones distribuidas de tiempo real; minimizando la inversión de prioridad, la latencia o el *jitter*. RT-CORBA proporciona mecanismos para manipular las prioridades de las peticiones, tanto en el lado del cliente como en el del servidor, y además permite utilizar las prioridades independientemente del sistema operativo utilizado, con lo que se evitan dificultades al poder centrarse en un esquema de prioridades único y consistente en todo el sistema.

Prioridades nativas y prioridades CORBA

RT-CORBA proporciona dos tipos de prioridades para evitar los problemas relativos a los diferentes esquemas de prioridad en diferentes sistemas operativos. De esta forma se distingue entre:

- Prioridades nativas: son las prioridades utilizadas por el sistema operativo en el que se ejecuta una aplicación.
- Prioridades CORBA: son independientes del sistema operativo, con valores entre 0 y 32768, donde un valor más alto se considera más prioritario.

El programador de aplicaciones CORBA sólo tiene que utilizar prioridades CORBA, consistentes en todo el sistema. Los ORBs deben proporcionar mecanismos para transformar prioridades CORBA en nativas y viceversa.

Modelos de prioridades

Los objetos CORBA pueden seguir dos modelos diferentes de prioridades, según se desee respetar la prioridad del cliente o utilizar otra prioridad fijada en el servidor.

- *Modelo propagado por el cliente (CLIENT_PROPAGATED model)*: en el modelo propagado por el cliente, las invocaciones son ejecutadas en la hebra del sirviente con la misma prioridad CORBA con la que se ejecutó la hebra que hizo la petición. La traducción de la prioridad CORBA a la prioridad nativa del sistema operativo, la lleva a cabo el ORB de manera automática.
- *Modelo declarado por el servidor (SERVER_DECLARED model)*: en este modelo es el servidor el que asigna prioridades de tiempo real de CORBA. Y es a esa prioridad a la que se ejecutará el sirviente cuando le lleguen las peticiones de los clientes, independientemente de las prioridades de las hebras de los clientes. El servidor es responsable de establecer correctamente las prioridades.
- *Transformaciones de prioridades*: si bien los dos modelos anteriores cubren una amplia gama de aplicaciones, puede ocurrir que el desarrollador desee utilizar sus propios modelos de propagación de prioridades, o que haya situaciones más dinámicas que requieran posibilidades adicionales. Las transformaciones de prioridad cumplen este objetivo, al permitir modificar la prioridad de una invocación

en diferentes puntos, básicamente, tras la recepción de la llamada por el ORB de destino, y tras la realización de peticiones por un sirviente.

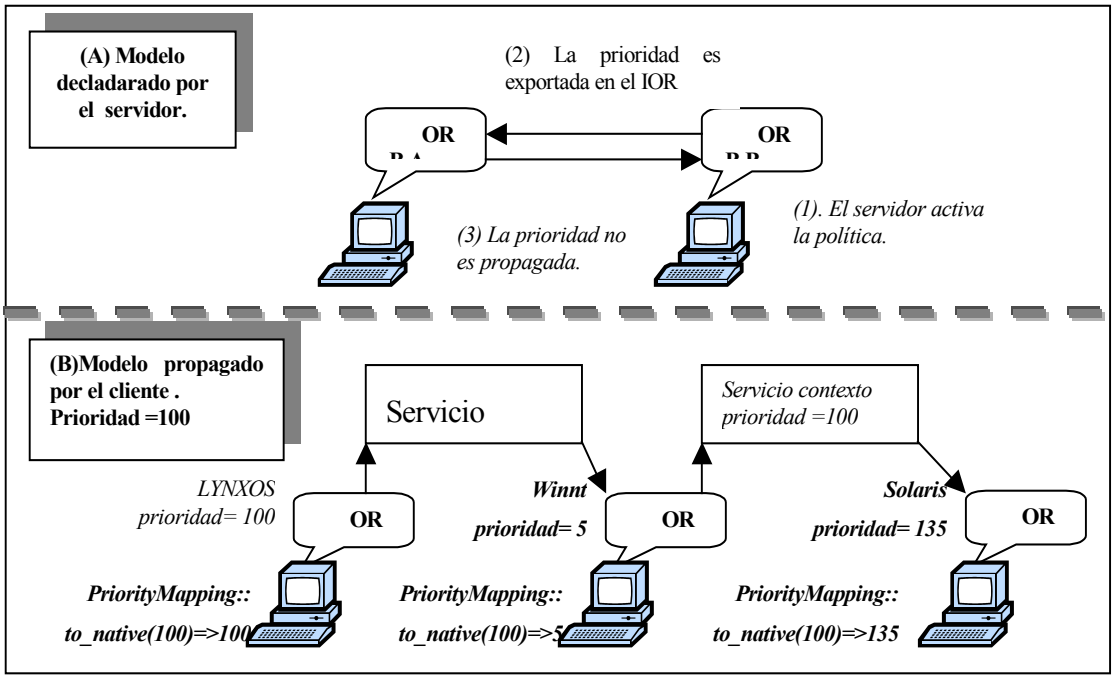


Figura 1.10. Modelos de prioridad en RT-CORBA

La interfaz Current

RT-CORBA define la interfaz `Current` para poder obtener y/o modificar la prioridad CORBA de la hebra actual (proporcionando indirectamente acceso a la prioridad nativa). Contiene un atributo denominado `the_priority` con el valor de la prioridad CORBA de la hebra actual.

1.3.4. Servidores multihebra

La utilización de múltiples hebras o tareas es necesario en el desarrollo de aplicaciones de tiempo real. Tener múltiples hebras permite dar prioridad a los diferentes eventos que pueden ocurrir en nuestro sistema.

La especificación RT-CORBA añade el concepto de *thread pool* o *conjunto de hebras* para permitir disponer de más de una hebra en el servidor. Estas hebras son utilizadas para realizar el tratamiento de las peticiones por parte de los clientes, y permiten eliminar la impredecibilidad asociada a la creación dinámica de hebras o asociar diferentes niveles de prioridad para el tratamiento de las peticiones. Es decir,

aportan predecibilidad facilitando la gestión de los recursos del procesador y de la memoria.

Un *thead pool*, es por lo tanto, un conjunto de hebras creadas para el tratamiento de las peticiones de los clientes. Los *thead pool* son asociados a un POA durante su creación y presentan varias características fundamentales:

- Hebras estáticas: durante la creación de un *thead pool*, se indica un número inicial de hebras estáticas que son creadas conjuntamente con el *thead pool*.
- Hebras dinámicas: un *thead pool* puede permitir la creación de un número máximo de hebras dinámicas para utilizar en caso de que todas las hebras estáticas estén siendo utilizadas.
- Prioridad por defecto: indica la prioridad a la que serán atendidas las peticiones por parte de los clientes. No obstante, se respetarán los mecanismos de propagación de prioridades descritos anteriormente.
- Capacidad de encolamiento: se puede permitir, en caso de que no haya hebras disponibles, encolar las peticiones en espera de que termine alguna petición.

Se distinguen, además, dos tipos de *thead pool*s: con calles y sin calles (*lanes*), según puedan distinguirse diferentes niveles de prioridad dentro de un mismo *thead pool*.

Un *thead pool* con calles permite la creación de distintas *calles* con diferentes niveles de prioridad. Los *thead pool* con calle pueden además configurarse para que las diferentes calles puedan prestarse hebras entre sí. Se realiza una partición de las hebras en diferentes conjuntos, con diferentes prioridades en cada uno de estos conjuntos.

1.3.5. Sincronización

El acceso predecible a los recursos compartidos es fundamental en un sistema de tiempo real. RT-CORBA proporciona mecanismos de sincronización que permiten acotar los tiempos de bloqueo en el acceso a estos recursos.

Una implementación RT-CORBA debe proporcionar una implementación de *Mutex* (`RTCORBA::Mutex`) que implemente alguna forma de herencia de prioridad.

Podrían ser válidas tanto la herencia simple de prioridad o alguna forma del protocolo de techo de prioridad [Rajkumar, 1991][Sha et al., 1990].

Al utilizar este tipo de *mutexes* tanto el ORB como la aplicación, se asegura un esquema de herencia de prioridad consistente en todo el sistema y que además limita la inversión de prioridades.

Los *mutex* de RT-CORBA son objetos con restricciones de localidad, no pueden ser utilizados remotamente. Disponen de una operación para bloquear el acceso al *mutex*, y otra para desbloquearlo, así como otra operación que intenta bloquear el *mutex* con un tiempo máximo de espera.

1.3.6. Configuración de protocolos

La selección de los protocolos de comunicación es crucial para la obtención de calidad de servicio. En los ORBs que no son de tiempo real, el sistema operativo, la red y/o el bus de comunicaciones son considerados como una *caja negra*. Aunque esta aproximación es útil para aplicaciones del tipo *mejor esfuerzo*, es inadecuada para aplicaciones con requisitos más exigentes de calidad de servicio.

La configuración de algunos aspectos de la conexión entre cliente y servidor va a hacer que RT-CORBA pierda *transparencia de localización*, pero hace factible así la optimización de los recursos de red.

Básicamente RT-CORBA proporciona tres mecanismos:

- Preasignación de conexiones: permite establecer una conexión explícita entre cliente y servidor mientras que en CORBA estándar el proceso de establecimiento de las conexiones es responsabilidad del ORB. El pre-establecimiento de las conexiones permite eliminar una fuente muy común de *jitter*.
- Política de prioridad de conexión con bandas: se crean varios bandas de comunicaciones con distintas prioridades. La prioridad de la invocación determina qué banda debe usarse.
- Política de conexión privada: en CORBA estándar las conexiones pueden ser compartidas entre múltiples invocaciones, como consecuencia de ello, se puede producir una pérdida de rendimiento en invocaciones más prioritarias. Para evitar esto, se puede utilizar la política de conexión privada, que garantiza conexiones no compartidas entre el cliente y el servidor.

1.4. SDL y la extensión de tiempo real

Una de las técnicas de descripción formal más extendida y usada es SDL (*Specification and Description Language*). SDL es un lenguaje de especificación gráfico orientado al diseño de sistemas de comunicación distribuidos. SDL es un lenguaje con una base formal donde cada símbolo lleva asociada una semántica exacta, de tal forma que permite eliminar cualquier ambigüedad y garantiza la integridad del sistema. SDL es un estándar de la Unión Internacional de Telecomunicaciones (*International Telecommunication Union,* ITU) y está muy bien soportado en herramientas comerciales; siendo, posiblemente, el método formal más utilizado en la industria.

La Técnica de Descripción Formal SDL fue estandarizada en 1976 en la *Recomendación Z.100* de la ITU. Desde esa fecha hasta la actualidad, la ITU ha actualizado cada cuatro años dicha *Recomendación*. Una de las mayores modificaciones de SDL se produjo en 1992, dando lugar a SDL92, donde el aspecto incorporado más importante fue la orientación a objetos. En la versión de 1996, prácticamente no se hicieron cambios importantes. Posteriormente, en el año 2000 se presentaron nuevas modificaciones importantes. Las partes del trabajo presentado que utilizan SDL se han basado en SDL96.

SDL es un lenguaje gráfico que permite la visualización de los modelos del diseño en vez de utilizar una notación textual, que también es posible generar. Si se especifica un sistema usando SDL existen herramientas que permiten:

- Compilar la especificación para detectar los posibles errores sintácticos.
- Simular y validar la semántica del diseño realizado.
- Generar casos de prueba que permitirán controlar el diseño realizado.
- Generar código para producir ficheros ejecutables sobre diferentes plataformas como VxWorks, POSIX, WIN32, etc...

SDL está basado en máquinas de estados finitos extendidas y, aunque puede ser usado en todas las fases de desarrollo de software, desde la especificación a la implementación, es más adecuado para la fase del diseño del sistema.

La utilización de SDL en la especificación se orienta a describir *qué* debe ofrecer un sistema. El objetivo es establecer claramente la relación entre el estímulo que trata el sistema y la respuesta a éste. En la etapa de diseño, se debe describir *cómo* un sistema debe llevar a cabo sus funciones. El diseño es un proceso incremental, es decir, el sistema debe ser una transformación de la fase de especificación. Por último, el uso de

SDL en la implementación consiste en tomar el diseño y generar un producto ejecutable, además de optimizarlo y añadir información de la plataforma donde se va a ejecutar.

SDL no es adecuado para diseñar aplicaciones de tiempo real debido a que si bien permite modelar características tales como la concurrencia y reactividad, no trata de forma completa aspectos importantes para esta clase de aplicaciones, como la sincronización, las restricciones de tiempo y la predecibilidad. Estos problemas, entre otros, hacen necesario incluir algunas extensiones para el tratamiento de sistemas de tiempo real con SDL. Estas extensiones, junto con técnicas de diseño que solucionen las limitaciones presentadas constituyen parte de la tesis doctoral de Luis M. Llopis [Llopis, 2002], habiéndose utilizado parte de este trabajo.

1.4.1. Fundamentos de SDL

La especificación de un sistema debe describir tanto el comportamiento interno como el externo. La vista externa del sistema es el comportamiento externamente observable, es decir, las secuencias de respuestas y las secuencias de estímulos. Por otra parte, la vista interna es el conjunto de acciones que son ejecutadas por los diferentes elementos del sistema. Las máquinas de estados finitos extendidas comunicantes (MEFEs) se utilizan para la descripción del comportamiento del sistema. Extienden el modelo MEF en varias direcciones como la definición de variables locales, entradas y salidas con valores, transiciones con guardas, etc. Las MEFEs se representan mediante *procesos* y la comunicación entre ellos, o entre procesos y el entorno, se representa con *señales*.

Una MEFE está compuesta de un número finito de *estados* y *transiciones* que conectan los estados, como puede verse, por ejemplo, en la Figura 1.11.

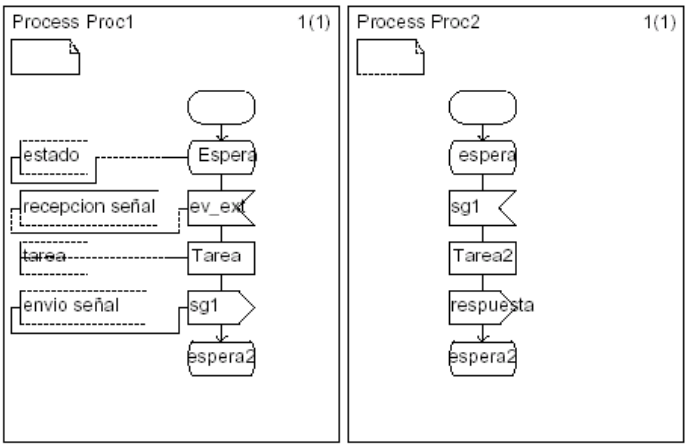


Figura 1.11. Las máquinas de estados extendidas con notación SDL

Procesos SDL

Los diagramas de procesos describen los patrones de comportamiento. Un proceso SDL tiene las siguientes características:

- Tiempo de vida. Comenzará a existir cuando se cree y dejará de existir cuando se ejecute el símbolo `stop`.
- Parámetros. Se pueden pasar en la creación dinámica pero no en la estática.
- Estado. Determina cómo un proceso reacciona con respecto a la entrada de una señal.
- Variables. En combinación con el estado, forman el *estado-espacio* del proceso.
- Puerto de entrada. Recibe y almacena las señales que pueden ser consumidas.

En la Figura 1.12 se pueden ver las dos partes principales de un proceso: la zona de descripción del comportamiento y la zona de definiciones.

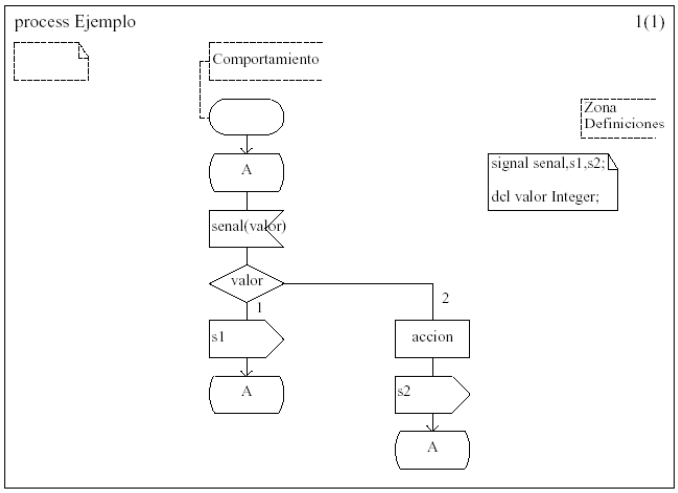


Figura 1.12. Zonas de un proceso SDL

Cuando un proceso se ha creado, la ejecución del cuerpo de éste comienza por la transición que sigue al símbolo `start`. El símbolo `stop` indica la finalización del proceso. No existen mecanismos específicos para controlar la finalización de procesos.

Si un proceso no está ejecutando ninguna acción, entonces deberá estar esperando en un estado a que se produzca un estímulo. Los estímulos a los que responde el proceso pueden ser señales o temporizadores expirados.

Conceptualmente una máquina de estados tiene en cuenta diferentes situaciones como, por ejemplo, esperar la recepción de una llamada si se está modelando un teléfono o esperar la inserción de una moneda si se diseña una máquina de refrescos.

Cada situación se caracteriza por un conjunto de variables asociadas y un manejo de esa situación una vez que se recibe una petición. Cada una de estas situaciones es un estado desde el punto de vista de SDL.

Comunicación entre procesos: señales

La comunicación entre procesos se realiza a través del envío asíncrono de lo que se denominan *señales*. Las señales son eventos de comunicación atómica que aparecen entre el entorno y el sistema o internamente entre los procesos. Una instancia de una señal se crea cuando un proceso ejecuta una operación de *envío de señal* y deja de existir cuando un proceso la consume con una operación de *recepción de señal*.

Los *canales* y *rutas de señales* llevan la instancia de la señal desde el proceso de envío al receptor. SDL asume que una instancia puede ser consumida o creada por el entorno. Cuando una instancia de una señal llega al proceso receptor, ésta se mantiene en lo que se denomina el *puerto de entrada* hasta que el receptor la consume. De esta forma, el puerto de entrada es una cola FIFO ilimitada que almacena instancias pendientes de ser consumidas.

La ventaja de este esquema de comunicación es la falta de acoplamiento entre las diferentes partes del sistema. Un proceso que quiere enviar una señal no se preocupa de que el receptor esté preparado para aceptar esa comunicación y, además, continúa su ejecución normalmente. Esta falta de acoplamiento es una buena forma de construir sistemas de gran tamaño y, por supuesto, de aumentar la concurrencia del sistema.

En la Figura 1.13 se muestra un envío y recepción de una señal.

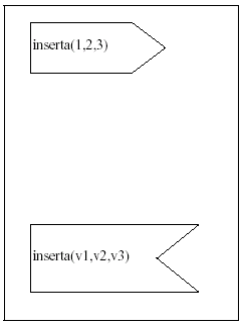


Figura 1.13. Envío y recepción de señales

SDL también soporta otros esquemas de comunicación como la *llamada a procedimiento remoto* (*Remote Procedure Call*, RPC), mecanismo síncrono que se tratará posteriormente.

Elementos de programación en SDL

SDL incluye una serie de elementos que permiten realizar algoritmos como si de un lenguaje de un programación textual se tratase incluyendo la posibilidad de manipular variables, decisiones, procedimientos, etc. La Figura 1.14 muestra el cuerpo de un algoritmo SDL junto con algunos de los símbolos utilizados para representar tareas (ej: modificación de variables) o decisiones.

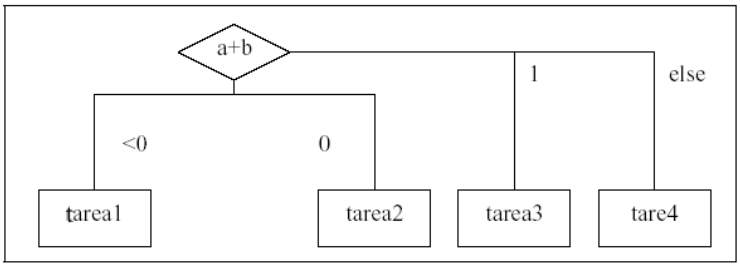


Figura 1.14. Tareas y decisiones en SDL

Procedimientos remotos

El intercambio de señales es el mecanismo básico de comunicación asíncrono. Sin embargo, SDL ofrece otros esquemas, entre los que está la *llamada a procedimiento remoto*, que permite que la comunicación entre procesos sea síncrona.

Un procedimiento remoto es un procedimiento que puede ser llamado por otro proceso diferente a donde está definido. La Figura 1.15 muestra un ejemplo en el que un proceso *p2* contiene una llamada a un procedimiento *p* exportado por el proceso *p1*. Si *p* es llamado cuando *p1* está en el estado *s1*, se realizará una llamada local a *p*, el resultado se devuelve a *p2* y *p1* se mantiene en el mismo estado *s1*. Si *p* se llama cuando *p1* está en el estado *s2*, la llamada local se pospondrá.

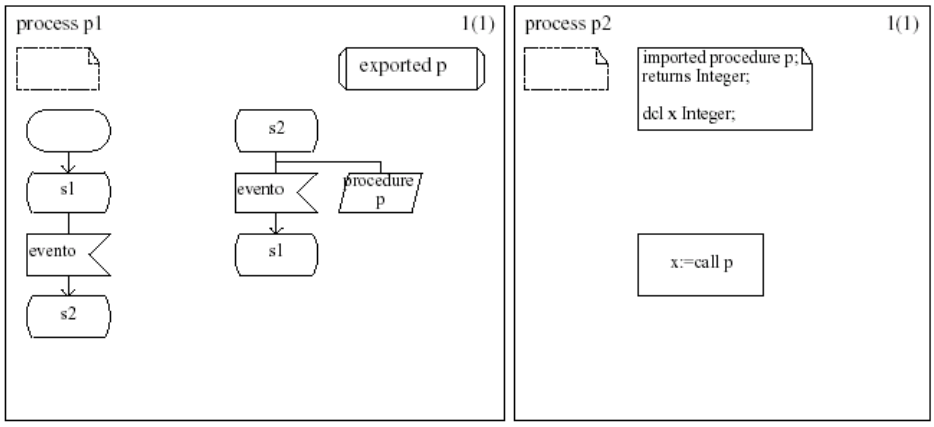


Figura 1.15. Ejemplo de manejo de procedimiento remoto

1.4.2. Carencias de SDL para tiempo real

En las secciones anteriores se han descrito los mecanismos básicos de SDL. Sin embargo, éste método formal posee una serie de carencias en el caso de que sea utilizado para el diseño de STRs. Las limitaciones de SDL son muy diversas; desde cuestiones de diseño detallado (como la dificultad de especificar procedimientos de expiraciones o *timeouts*, y la dificultad de conseguir transiciones atómicas), hasta cuestiones de alto nivel (como la dificultad de modelar el tiempo en situaciones no deterministas).

Limitaciones en la expresión de las restricciones temporales

El manejo del tiempo en las técnicas de descripción formal es fundamental para construir diseños que cumplan los requisitos temporales impuestos por las aplicaciones de tiempo real. SDL presenta dos mecanismos principales para tratar con el tiempo: un reloj global y los temporizadores.

Para la utilización del tiempo desde un punto de vista general estos mecanismos pueden ser suficientes pero, sin embargo, presentan limitaciones en STRs. Por ejemplo, para asegurar que la transición asociada a la señal que se envía, debido a la expiración del temporizador, se ejecuta inmediatamente, el diseñador debe asegurarse que el proceso que la contiene está inactivo cuando se recibe ésta. Si no es así, la señal pasará a la cola de entrada y será atendida cuando llegue su turno. Con esta semántica será imposible activar un proceso SDL exactamente cuando se produzca la expiración.

Otra limitación asociada a la semántica del tiempo definida en SDL es la dificultad de expresar restricciones de tiempo real relacionadas con el envío y recepción de señales. El problema central es que es imposible conocer cuándo una señal ha sido realmente enviada o atendida, por lo que se añade una fuente de impredecibilidad que dificulta el análisis.

En el supuesto que la planificación de los procesos SDL sea interrumpible, aspecto que debe ser tenido en cuenta en el diseño de sistemas de tiempo real para facilitar el cumplimiento de los requisitos temporales, no es posible determinar el tiempo que transcurre entre que la señal comienza a atenderse y la invocación de la función `now`, en el caso de que ese proceso hubiera sido interrumpido por otro más prioritario justo antes de invocar a dicha función.

Limitaciones en la semántica de ejecución de SDL

La semántica de SDL no modela el paso del tiempo correctamente. Por ejemplo, el tiempo que pasa cuando un sistema está suspendido en un estado o ejecuta una acción es indeterminado, es decir, no hay forma de especificar tiempo de ejecución en las acciones.

Si no existe un control completo sobre el tiempo no será posible simular el sistema, ni validarlo completamente. Además, tampoco se podrá incluir un análisis de planificabilidad en la fase de diseño.

Inversión de Prioridades y Acceso a los Recursos Compartidos

La inversión de prioridades no acotada en SDL puede ocurrir cuando diferentes procesos *clientes* intentan acceder a un mismo proceso *servidor* encapsulando a un recurso. En el caso de que se mantuviera la semántica de *primero en entrar, primero en salir* el proceso de mayor prioridad podría estar suspendido un tiempo ilimitado.

1.4.3. Extensiones de tiempo real para SDL

En este apartado se describirán un conjunto de extensiones presentadas en [Llopis, 2002] que permitirán solucionar las limitaciones de los apartados anteriores. El objetivo es introducir el mínimo número de modificaciones y mantener la semántica de SDL lo más coincidente posible a la semántica estándar y, de ese modo, poder seguir utilizando herramientas comunes para los aspectos no relacionados con el tiempo. Las primeras extensiones que se proponen están relacionadas con los mecanismos para expresar restricciones temporales:

Prioridades para las transiciones y los procesos

Aunque en algunos entornos [TAU, 2000], ya se definen extensiones que hacen posible la especificación de prioridades para los procesos, la propuesta es asignarlas a las transiciones de los procesos, dependiendo de los eventos a los que responden. Si se asigna la prioridad a las transiciones, la prioridad del proceso dependerá de las señales en su cola de entrada y la transición que se puede ejecutar en el estado actual. En la Figura 1.16 se muestra cómo pueden especificarse esas prioridades, introduciendo un

símbolo de comentario, asociado a cada transición, con la palabra reservada *with priority* y su prioridad, por lo que, de ese modo, no se altera la sintaxis definida.

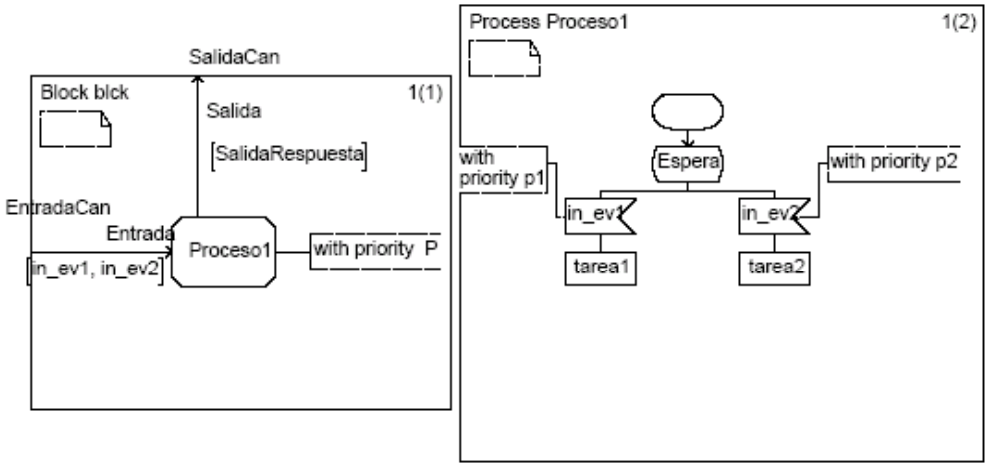


Figura 1.16. Especificación de la prioridad

Marcas de tiempo para las señales.

Cada una de las instancias de las señales almacenará dos marcas de tiempo, una cuando la señal se envió y otra cuando la señal se atiende. Se podrán acceder con dos funciones predefinidas: *tiempo_enviado* y *tiempo_atendido*, mediante las cuales se conseguirán expresar fácilmente restricciones de tiempo real relativas, por ejemplo, a la variación o *jitter* de entrada y de salida.

Un modelo de ejecución analizable

El nuevo modelo de ejecución está basado en una planificación interrumpible con prioridades fijas aunque, sin embargo, las prioridades no se asignarán directamente a los procesos sino a sus transiciones. Las prioridades del proceso variarán de un estado a otro, dependiendo de la transición que se ejecute en el estado actual (teniendo en cuenta la cola de entrada de señales del proceso). Los procesos se planificarán en función de esta prioridad *dinámica*, aunque el análisis de planificabilidad se basará en las prioridades de las transiciones, que son fijas.

El modelo también limita la utilización de algunos aspectos de SDL que pueden ser fuente de no determinismo. Concretamente, no se permite la creación dinámica de procesos, la existencia de señales `save`, ni la utilización de *servicios*.

Las transiciones se ejecutarán en función de su prioridad pero teniendo en cuenta ciertas limitaciones derivadas de la semántica de SDL. Esto quiere decir que las transiciones pueden ser interrumpidas por otras de más alta prioridad que estén preparadas para ejecutar pero, por ejemplo, nunca si éstas pertenecieran al mismo proceso SDL. Es decir, si una transición de un proceso tiene alta prioridad y está disponible para ejecutar mientras otra transición de menor prioridad lo está haciendo, aquélla deberá esperar a que la transición que se está ejecutando actualmente termine, si ambas pertenecen al mismo proceso SDL. Evidentemente, esto puede causar un incremento en el tiempo de respuesta de los eventos, pero esta restricción es necesaria si se quiere mantener la semántica de ejecución de los procesos SDL que no permite que dos transiciones del mismo proceso puedan ejecutarse concurrentemente.

Compartición de recursos

Todos aquellos datos y recursos que son compartidos por varios procesos son encapsulados en una clase especial de proceso. Con objeto de evitar incluir más extensiones que modifiquen la semántica de SDL, estos procesos no se podrán distinguir externamente de los demás pero su comportamiento estará limitado de la siguiente forma:

- Estos procesos actúan como un *servidor pasivo*, es decir, ellos mismos no comienzan ninguna acción (no tienen temporizadores ni realizan envíos autónomos de señales).
- El mecanismo de comunicación con estos procesos es síncrono. Esto se consigue realizando *llamadas remotas a procedimiento*, donde el proceso que realiza la llamada deberá ejecutar el procedimiento asociado antes de continuar con su ejecución.
- Finalmente, el bloqueo durante la ejecución de la transición debe ser limitado para asegurar la predecibilidad.

Para limitar ese bloqueo, cada uno de estos procesos tiene asignado un techo de prioridad, que es el máximo de las prioridades de todas las transiciones que acceden al recurso. De esta forma se evita la inversión de prioridades y se garantiza la exclusión mutua, puesto que todas las transiciones se ejecutarán, en el acceso al recurso, a la más alta prioridad de entre todas las que lo comparten.

Otra ventaja relacionada con la implementación de esta estrategia de diseño es que se puede implementar muy eficientemente. Aunque los datos compartidos se encapsulan en un proceso, este proceso no se traduce a un proceso real en la implementación, sino que pueden ser procedimientos, uno por cada una de las transiciones. Los procesos que acceden a estos recursos compartidos invocarán a estos procedimientos y, además, tendrán que cambiar su prioridad a la prioridad del techo del recurso. Esta implementación es similar a la que define ADA95 con los tipos protegidos.

Variación en la activación

Otro conjunto importante de requisitos de tiempo real son los relacionados con el *jitter de control* o variación en la activación. La variación en la activación, aparece cuando no sólo es suficiente con que la respuesta a un evento deba cumplir un plazo establecido, sino que la acción de entrada o salida se debe realizar en un momento específico del tiempo, con una pequeña variación o tolerancia (*jitter*).

El problema consiste en enviar una señal, *salida*, en un instante de tiempo concreto con respecto a la llegada de la señal, *entrada*, que activó la transición que la contiene. Sin embargo, como se puede observar en la Figura 1.17, la tarea termina antes del momento en el que se debe enviar la señal *salida* y será necesario retrasar ese envío sin sobrecargar el sistema.

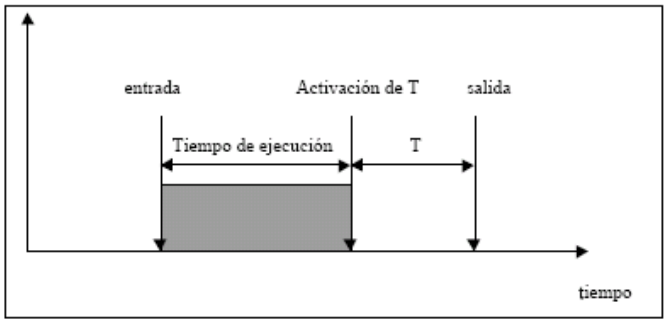


Figura 1.17. Cronograma para el retraso en la activación

El primer requisito para resolver el problema es conocer el instante en el que se produce el evento que activa la transición. Esto se puede conseguir fácilmente guardando las marcas de tiempo con las operaciones predefinidas *tiempo_enviado* y *tiempo_atendido* descritas anteriormente. Además, será necesario asegurar que la salida de la señal ocurre exactamente en el instante requerido. La solución pasa por definir un

temporizador y asignar a la transición que trata la expiración de éste, la máxima prioridad del sistema. La precisión del temporizador y la sobrecarga del planificador serán aspectos a tener en cuenta pues también formarán parte de la variación final.

1.5. Aportaciones

Las aportaciones realizadas por esta tesis están centradas en la aplicación de la Ingeniería del Software a la programación de sistemas de tiempo real. Las distintas fases en el desarrollo de esta tesis han permitido realizar aportaciones en distintos campos dentro de la Ingeniería del Software: paradigma de programación orientado a componentes, *middlewares* de comunicaciones, técnicas formales, etc.

- Modelo de componentes para tiempo real: los modelos de componentes estándares no pueden ser utilizados de forma satisfactoria en los sistemas en tiempo real, y mucho menos en sistemas distribuidos de tiempo real. Sin embargo, la utilización de este paradigma de programación aportaría indudables beneficios a la comunidad de tiempo real. Se ha diseñado un modelo de componentes [Díaz et al., 2004] con todas las características necesarias para su utilización en sistemas de tiempo real distribuidos.

  El modelo de componentes incorpora su propio lenguaje de definición de interfaces basado en el IDL de CORBA, pero con construcciones propias para los sistemas de tiempo real.

  Para facilitar la tarea del programador, se incorporan también al modelo una serie de herramientas que permiten realizar el diseño de los componentes, componer aplicaciones, desplegar los componentes, etc.
- Entorno de ejecución predecible: el modelo de componentes por sí solo, no es suficiente para el desarrollo de aplicaciones distribuidas predecibles, sino que debe apoyarse en una plataforma de ejecución. Para poder disponer de un modelo que pueda ser utilizado en múltiples plataformas, se ha desarrollado un entorno de ejecución basado en RT-CORBA [Díaz et al., 2004b] que permite la ejecución de los componentes del modelo de una forma predecible y en distintas plataformas.

  El entorno permite la ejecución de los componentes de manera que los detalles de comunicación sean transparentes al programador, que sólo tendrá que utilizar las herramientas proporcionadas por el modelo de componentes. La utilización de

CORBA, y de RT-CORBA en particular, aporta numerosas ventajas en la implementación del modelo, al encargarse la implementación elegida de los detalles de comunicaciones tales como establecimiento de conexiones, empaquetamiento y desempaquetamiento de parámetros en las invocaciones, etc.

- Modelo de análisis distribuido: existen numerosos trabajos sobre análisis de planificabilidad. Si bien, relativamente pocos, se centran en los sistemas basados en componentes y distribuidos. En la presente tesis se aporta un modelo de análisis basado en SDL [Díaz et al., 2005] que permite comprobar la planificabilidad de los sistemas desarrollados con el modelo de componentes y sobre el entorno de ejecución basado en RT-CORBA.

  El trabajo presentado en [Llopis, 2002] permitía realizar el análisis de planificabilidad de aplicaciones monoprocesador utilizando la extensión de tiempo real presentada en dicha tesis. En esta tesis, se extiende dicho trabajo en dos direcciones. Por un lado, se realiza el modelado de RT-CORBA en SDL. Hacer este modelado permite la utilización de todas las ventajas de SDL en RT-CORBA. De esta forma pueden realizar numerosos tipos de análisis sobre RT-CORBA o las implementaciones de RT-CORBA utilizadas. En segundo lugar, se incorpora el estudio de la sobrecarga de las invocaciones remotas al modelo de análisis.

- Aplicación concreta: el Grupo de Ingeniería del Software de la Universidad de Málaga (GISUM) ha trabajado en cooperación con la empresa Tecnatom S.A. en numerosos proyectos relacionados con simuladores para centrales nucleares [Díaz y Garrido, 2003][Díaz y Garrido, 2004][Díaz y Garrido, 2004b][Díaz et al., 2005b][Díaz et al, 2005c]. Como una evaluación de las anteriores propuestas, se han utilizado todas ellas para el modelado, diseño e implementación de componentes para este tipo de sistemas.

  Los simuladores para centrales nucleares son un tipo de sistemas donde las restricciones de tiempo cobran especial importancia. Este tipo de simuladores son utilizados para el entrenamiento de los operadores de la central, planteando diferentes situaciones, desde las más normales a las más extraordinarias. El funcionamiento de estos simuladores se basa, internamente, en un conjunto de aplicaciones distribuidas con requisitos de tiempo real que interoperan entre sí, es por ello, un tipo de sistema idóneo al que se pueden aplicar todas las aportaciones anteriores.

Capítulo 2. Un modelo de componentes distribuido para tiempo real

El desarrollo basado en componentes es una tecnología clave para la creación de sistemas en tiempo real. Sin embargo, como ha podido comprobarse, los modelos estándares de componentes y sus herramientas asociadas no son adecuadas para este tipo de sistemas, al no considerar explícitamente cuestiones de tiempo real.

El primer objetivo de la tesis es la creación de un modelo de componentes predecible para sistemas distribuidos, junto con un entorno de desarrollo y ejecución que permita la creación y posterior análisis de aplicaciones distribuidas de tiempo real.

Nuestro modelo de componentes, UM-RTCOM [Díaz et al. 2004], está basado en la definición de componente de [Szypersky et al., 2002]. Desde este punto de vista, un componente es una unidad de composición básica con un conjunto de interfaces y requisitos, que ha sido (probablemente) desarrollado de manera independiente al resto de la aplicación (en tiempo y espacio). Los componentes pueden ser configurados y conectados a otros componentes a través de algún mecanismo de composición específico y se ejecutarán sobre alguna infraestructura tales como, RT-CORBA, CORBA CCM, JavaBeans o .NET. De forma particular, en las aplicaciones de tiempo real, los componentes deben además colaborar para cumplir sus restricciones temporales (*deadlines*, *jitter*, etc.). Estas restricciones son dependientes del hardware, de forma que un componente debe ser adaptado y verificado para cada plataforma hardware destino. Además, el comportamiento temporal debe ser estudiado tanto a nivel de componente como de aplicación.

El modelo enriquece los modelos tradicionales al incorporar construcciones que permiten expresar requisitos temporales, sincronización, calidad de servicio, eventos, etc. Es un modelo jerárquico en el que los componentes actúan, al mismo tiempo,

ofreciendo interfaces para su composición con otros componentes, y como contenedores de otros componentes. Un componente indica qué interfaces proporciona y qué interfaces necesita, de forma que sólo puede ser utilizado cuando se realiza la composición con componentes que ofrecen las interfaces que necesita. Sólo los componentes que no requieren interfaces pueden actuar como aplicaciones.

El modelo incorpora mecanismos para poder hacer análisis de tiempo real a nivel de componente y aplicación, a través de metainformación consistente en un modelo de comportamiento abstracto de los componentes junto con un método para la medida de los peores tiempos de ejecución en la plataforma final de ejecución. La utilización de un metamodelo permite, además, el poder realizar varios tipos de análisis diferentes del análisis de planificabilidad, si bien, en esta tesis nos centraremos en este último.

Mediante la utilización del entorno y las herramientas se pueden desarrollar nuevos componentes predecibles que, al combinarse con otros componentes predecibles permiten la construcción de sistemas que pueden ser a su vez comprobados con las herramientas proporcionadas en cuanto a tiempo real.

Las herramientas de análisis que forman parte del entorno no necesitan acceder al código del componente, viendo éstos como cajas *negras* y haciendo uso solamente del metamodelo asociado al componente. Desde este punto de vista, un componente solamente se considera completamente especificado si incluye esta información. El metamodelo ha sido especificado teniendo en cuenta la extensión de tiempo real de SDL, la cual está basada en análisis de razón monótona. Cada componente será descrito utilizando un conjunto fijo de elementos que permiten expresar su comportamiento temporal para un análisis posterior. Si bien, la extensión de tiempo real de SDL está diseñada para su utilización en sistemas monoprocesador, en la presente tesis, se realizará una ampliación para su utilización en sistemas distribuidos, por lo que podrá ser aplicado con el modelo de componentes propuesto.

A nivel de componente, las herramientas obtienen un modelo SDL abstracto del componente y el desarrollador tiene que realizar anotaciones que le van a permitir obtener los peores tiempos de ejecución en la plataforma final. Una vez que los componentes han sido interconectados, la aplicación resultante debe ser analizada con respecto a las restricciones de tiempo real externas. Con este objetivo, los modelos abstractos de los componentes y la medida de los peores tiempos en la plataforma final permiten realizar el análisis de tiempo real.

2.1. Dos visiones: desarrollador y usuario

En el modelo presentado, se consideran dos diferentes vistas: la vista del desarrollador del componente y la vista del usuario del mismo. Esta aproximación, utilizada también en el modelo CCM de CORBA, aporta como principal ventaja la claridad en la distinción de estos dos roles para el desarrollo basado en componentes. El desarrollo de un componente tiene problemáticas distintas a su utilización, son por ello, contextos diferentes que se resuelven mejor con visiones diferentes.

La vista del desarrollador se centra en los elementos básicos para la construcción de componentes y en cómo éstos pueden ser combinados durante la fase de desarrollo para la creación de nuevos componentes o aplicaciones.

Por su parte, la vista de usuario se centra en la utilización de las interfaces de los componentes, la configuración de los componentes en plataformas concretas y la indicación de restricciones de tiempo real en situaciones concretas. Esta vista, no incluye ningún detalle sobre la implementación del componente y es utilizada por las herramientas de ensamblado y análisis del entorno.

2.1.1. La vista del desarrollador

Desde el punto de vista del desarrollador, las bases para la construcción de un componente son los elementos básicos del modelo y la combinación e interacción de estos elementos para formar un nuevo componente. El modelo debe proporcionar mecanismos para expresar servicios ofertados o requeridos, comunicación síncrona o asíncrona, etc.

Para poder realizar el análisis de tiempo real, en UM-RTCOM se debe además proporcionar un metamodelo del componente consistente en un modelo SDL abstracto con el comportamiento del mismo junto con anotaciones para el cálculo de los peores tiempos de ejecución. El metamodelo es extraído automáticamente por las herramientas del entorno de desarrollo, y las anotaciones deben ser suministradas por el desarrollador del componente.

A través únicamente de estos elementos: “piezas” del modelo y anotaciones, se podrán desarrollar componentes de tiempo real que podrán ser posteriormente desplegados en un sistema distribuido y analizados con los modelos SDL generados de manera semi-automática.

2.1.2. La vista del usuario

La vista del usuario proporciona mecanismos para utilizar los componentes y combinarlos adecuadamente a través de la interconexión de las interfaces de entrada y salida.

En el caso de nuestro modelo habrá, además, que configurar los componentes, pudiendo adaptarlos a diferentes plataformas destino, siendo esto importante para las aplicaciones de tiempo real. Por último, se podrán expresar restricciones de tiempo real tales como *deadlines* sobre las activaciones de métodos o eventos.

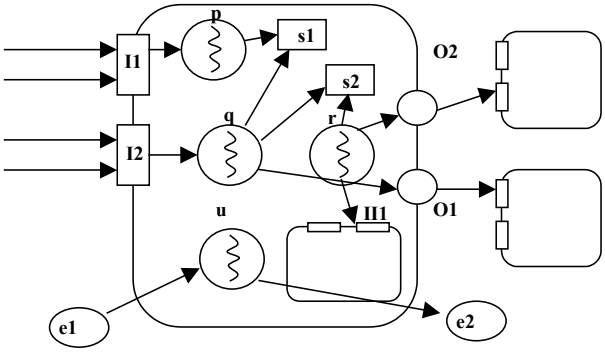
Conociendo únicamente las interfaces, los parámetros de configuración en la plataforma destino y las restricciones de tiempo real, el usuario del componente (que puede ser el desarrollador de un nuevo componente o de una aplicación final), declarará las instancias del componente que vaya a utilizar, dándoles una configuración concreta y utilizándolas mediante las interfaces o los eventos pudiendo analizar la aplicación para los requisitos temporales especificados.

2.2. Tipos de componentes

Hay dos tipos de componentes en el modelo: primitivos y genéricos. Los componentes genéricos (Figura 2.1) actúan como contenedores de componentes primitivos y de, posiblemente, otros componentes genéricos. Son los componentes *genuinos* del modelo. La interacción con estos componentes puede realizarse mediante la invocación de servicios o generando eventos.

Los componentes primitivos son la base para la construcción de estos componentes genéricos; existiendo dos tipos: activos (*Active*) *y* pasivos (*Passive*). Un componente genérico puede tener un número arbitrario de componentes activos y pasivos y utiliza a éstos para implementar el tratamiento de los servicios, eventos y comportamiento.

Los componentes activos son similares a hebras que comparten información a través de componentes pasivos o de otros componentes genéricos. Por su parte, los componentes pasivos son utilizados para encapsular recursos compartidos, proporcionando automáticamente exclusión mutua y mecanismos para herencia de prioridad a través del entorno de ejecución.




```
Component comp_name {
  interface I1 {
      void op1(<parameters>);
      void op2();
      }
  ...
  input     I1,I2;
  output O1 with o1;
  output O2 with o2;
  consumes event_type1;
  publishes event_type2;
}
Component implementation comp_name {
  Tipo_activo p,q,r,u;
  Tipo_pasivo s1,s2;
...
}
event event_type1(<parameters>);
event event_type2(<parameters>);
```

Figura 2.1. Componente genérico

2.2.1. Componentes genéricos

Los componentes genéricos constituyen la base del modelo. Actúan como contenedores de otros componentes, ya sean primitivos o genéricos, y pueden interconectarse con otros componentes para formar componentes mayores o aplicaciones.

Un componente genérico consta de una parte de definiciones, que será pública y visible para los usuarios del componente, y de una parte de implementación privada, y la cual incluye la implementación de todos los servicios ofertados por el componente.

Definición

La sintaxis para la definición de un componente puede verse en la Figura 2.2, incluyendo:

- Nombre del componente
- Definición de interfaces

- Interfaces requeridas y ofertadas
- Eventos consumidos y producidos
- Ranuras de configuración (se detallarán en secciones posteriores)

```
Component <comp_name> '{'
  {interface_definition}
  {input_interface}
  {output_interface}
  {consumed_event}
  {published_event}
'}'

interface_definition ::= interface <interface_name> '{'
                              {service_definition}
                         '}'


service_definition ::= void <service_name> '(' {args} ')' ';'

input_interface ::= input {interface_name} ';'
output_interface ::= output {interface_name} with <name_comp>';'
consumed_event ::= consumed {event_type_name} ';'
published_event ::= published {event_type_name} ';'
```

Figura 2.2. Sintaxis de definición de un componente

A continuación se muestra un ejemplo de definición de componente según la sintaxis indicada.

Ej:

```
Component micomponente {
  interface I1 {
      void op1(in short arg1,out short arg2);
      void op2();
      }
  ...

  interface O1 {
      void op3(in float a);
      void op4();
      }
  ...

  input  I1;
  input  I2;

  output O1 with o1;
  output O2 with o2;

  consumes event_type1;
  publishes event_type2;
}
```

Nombre del componente

Indica únicamente el nombre que el componente va a tener en el sistema, es similar, al nombre de una clase o módulo.

Ej:

```
Component micomponente
```

Interfaces

En un componente pueden escribirse una serie de interfaces que serán ofertadas o requeridas por el componente. En la definición de las interfaces se ha utilizado la misma sintaxis de CORBA-IDL, con las facilidades que ello conlleva para los desarrolladores, al ser CORBA-IDL un lenguaje bastante expresivo y extendido.

Ej:

```
interface I1 {
    void op1(in short arg1,out short arg2);
    void op2();
}
```

Un componente puede ofrecer varias interfaces, con las ventajas que ello conlleva al ofrecer distintas visiones de un mismo componente. Así, por ejemplo, se podrían modelar varias calidades de servicios mediante la utilización de varias interfaces, por ejemplo, para un comportamiento degradado del sistema.

El modelo permite diferenciar entre interfaces de entrada (*input*) e interfaces de salida (*output*). Las interfaces de entrada (funcionalidad proporcionada) constituyen los servicios ofertados por el cliente. Por otra parte, las interfaces de salida (funcionalidad requerida) son aquéllas que el componente actual requiere de otros componentes para poder ser utilizado (funcionalidad requerida). Para que un componente pueda ser utilizado en un sistema mayor, debe interconectarse con otros componentes que ofrezcan las interfaces que necesita.

Ej:

```
input  I1;
input  I2;
output O1 with o1;
output O2 with o2;
```

Por ejemplo, en un sistema de monitorización formado por controladores y sensores, un componente "controlador" podría requerir servicios ofertados por componentes "sensores", y a su vez, el componente "controlador" podría ofertar servicios de control.

Las interfaces requeridas (en el ejemplo anterior `O1` y `O2`) pueden ser posteriormente utilizadas en el código del componente como si ya estuvieran disponibles, con todos los servicios requeridos a través de la utilización de "referencias", que en el ejemplo son `o1` y `o2` para las interfaces `O1` y `O2` respectivamente. Es decir, se puede suponer en el código del componente que `o1` implementa los servicios de la interfaz `O1` y que `o2` implementa los servicios de la interfaz `O2`. Si bien, en tiempo de desarrollo no se dispone de estos componentes (y por ello se requieren), en tiempo de ejecución, el sistema se encargará de realizar la composición con componentes que ofertan estas interfaces.

Eventos

De manera adicional a las interfaces, que en el modelo UM-RTCOM son síncronas, se pueden también indicar eventos en la definición de un componente: qué eventos se producen en el componente y qué eventos se consumen. Los eventos son mecanismos de comunicación asíncronos, tanto para el que produce los eventos, como para el que los consume.

Ej:

```
consumes event_type1;
publishes event_type2;
```

La definición de eventos a consumir y producir se realiza indicando un tipo de evento, cuya definición es similar a la signatura de una operación, que deben respetar tanto el componente consumidor de eventos como el productor del evento.

```
event <event_type_name> '(' <args> ')' ';'
```

Figura 2.3. Sintaxis de definición de tipos de eventos

La definición de los tipos de eventos debe ser visible en la definición de las interfaces, para que los tipos de eventos indicados sean reconocidos como válidos. Los

argumentos del tipo de evento deben ser todos de entrada, puesto que al ser los eventos asíncronos no hay posibilidad de recoger respuesta alguna.

Ej:

```
event event_type1(in short num);
event event_type2(in short a,in short b);
```

Implementación de un componente

La parte de implementación de un componente incluye:

- Implementación de servicios
- Tratamiento de eventos
- Métodos privados y datos miembro
- Componentes utilizados (genéricos, activos y pasivos)
- Composición de interfaces

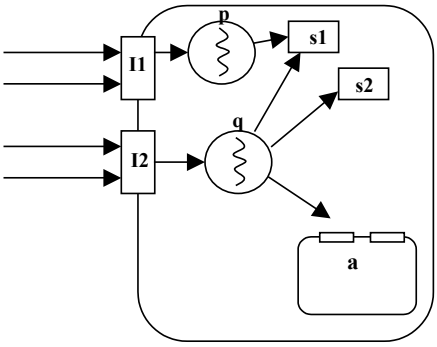
Con respecto al lenguaje de implementación, está basado en la sintaxis de C++, si bien, el modelo es independiente del mismo y podrían realizarse implementaciones de los componentes en otros lenguajes orientados a objetos.

La Figura 2.4 muestra parte de la sintaxis de implementación de un componente:

```
Component implementation <comp_name> '{'
  {component_member_def}
  {active_member_def}
  {pasive_member_def}
  {active_implementation}
  {passive_implementation}
  {data_member}
  {method_member}
  {interface_composition}
 '}'

component_member_def ::= <component_type> <name_member> ';'
active_member_def ::= <active_type> <name_member>
                                    [with period <time>]';'
passive_member_def ::= <component_type> <name_member> ';'


data_member ::= <data_type> <member_name> ';'
method_member ::= <return_type> <member_name> '(' {args} ')' ';'
```

Figura 2.4. Sintaxis de implementación de un componente

La Figura 2.5 muestra un ejemplo concreto de parte de la sintaxis de implementación de un componente.




```
Component implementation comp_name {
  Generic2 a;
  Tipo_activo p;
  Tipo_activo2 q;
  Tipo_pasivo s1;
  Tipo_pasivo2 s2;
...
}
```

Figura 2.5. Parte de implementación de un componente genérico

Implementación de servicios

La implementación de los servicios y el tratamiento de los eventos se realizará a través de los componentes primitivos *Active* como se verá posteriormente. Básicamente se utilizarán primitivas de sincronización que permiten a un componente activo encargarse del tratamiento de un determinado servicio sobre una determinada interfaz .

Tratamiento de eventos

Al igual que en el caso anterior, los eventos consumidos por el componente deben tratarse en rutinas del componente. En nuestro modelo, la utilización de primitivas de sincronización en los componentes activos permite realizar el consumo de los eventos producidos en el sistema.

Métodos privados y datos miembro

El componente puede incluir cualesquiera métodos adicionales así como datos miembro accesibles solamente desde el propio componente y los activos y pasivos contenidos. La sintaxis para estos métodos privados o datos miembro está basada en la de C++.

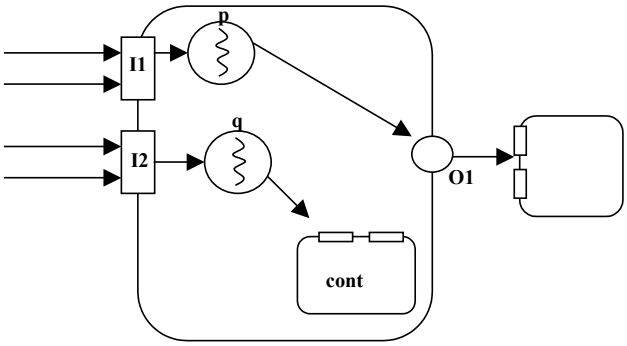
Ej:

```
float m_data;

int metodo1(float temp) {
      m_data = temp;
}
```

Utilización de componentes

Los componentes utilizados o contenidos pueden ser primitivos, que permiten implementar el comportamiento del componente, o bien, pueden ser componentes utilizados en su totalidad y de forma privada en la parte de implementación, es decir, no es necesario interconectar estos componentes con el componente desarrollado (pero posiblemente sí con otros componentes). Se utilizan las interfaces ofertadas, pero por algún motivo, no se desea expresar en la parte de definición que se "requieren" estos componentes. Se trataría pues, de componentes ya existentes en tiempo de desarrollo, mientras que en el caso de "requerir" las interfaces, se trataría de componentes no existentes en tiempo de desarrollo del componente.

La Figura 2.6 muestra un ejemplo de un componente genérico con estas dos situaciones. Se incluye un componente genérico interno y se realiza además la interconexión con un componente externo.




```
Component comp_name {
  interface O1 {
      void op1(<parameters>);
      void op2();
      }
  output O1 with o1;
...
}
Component implementation comp_name {
  Generico_contenido cont;
...
}
```

Figura 2.6. Componentes internos y externos

En el caso de componentes contenidos, el componente contenedor simplemente utiliza al otro componente, existiendo una relación jerárquica contenedor/contenido. En el segundo caso, ambos componentes son compuestos por una "entidad" de orden superior (por ejemplo otro componente) que "pega" los componentes a través de la conexión de las interfaces.

Ej:

```
// Definición de componentes utilizados
  Generic2 a;
  MiActivo p;
  MiPasivo s1;
```

En el caso de interfaces requeridas, en la parte de definición se indicaron "referencias" a "supuestos" componentes que estarán disponibles en tiempo de ejecución con las interfaces requeridas por el componente desarrollado.

Ej:

```
output O1 with o1;      // En la parte de definición
call o1.servicio1(<args>);    // En la parte de implementación
```

En el ejemplo anterior, se requería una interfaz de tipo `O1` (que fue completamente definida en la parte de implementación). En el código del componente (activos o pasivos) se podrá utilizar la referencia `o1` con todos los servicios ofertados en `O1`, como, por ejemplo, `servicio1`. Se ha presentado también una de las primitivas de sincronización (`call`) que será descrita posteriormente.

2.2.2. Componentes activos

Los componentes activos (*Active)* son los elementos primitivos del modelo que permiten expresar "flujos de ejecución" en el interior de un componente. La concurrencia es un factor esencial en los sistemas de tiempo real, de ahí la importancia de utilizar elementos de primer orden en el modelo para poder expresarla

En lugar de utilizar otros mecanismos, como pudieran ser hebras o procesos, el modelo utiliza pues, estos componentes activos para expresar concurrencia en el interior de un componente. Esta elección viene motivada sobre todo por la posterior fase de

análisis, ya que, al sólo poder utilizar este mecanismo de concurrencia, es más fácil su control y, por tanto, su análisis.

La utilización de componentes activos requiere una parte de definición y otra de declaración de instancias de la definición. Cada componente activo debe implementar un método *execute* sin valor de retorno y sin argumentos, que por ejemplo, con sintaxis de C++ sería: `void execute()`. Un componente activo puede incluir además métodos adicionales todos ellos privados. Un componente genérico puede contener cualquier número arbitrario de componentes activos.

Definición

La parte de definición incluye el nombre del tipo del componente activo, métodos adicionales (con la posibilidad de incorporar métodos de inicialización) y el método *execute*. Se pueden incluir también datos miembros privados, pero no así contener a otros componentes, ya sean genéricos, primitivos o pasivos. Sí se puede, sin embargo, acceder a los componentes genéricos o pasivos contenidos por el genérico que contiene a su vez al activo.

```
Active <active_name> '{'
  {data_member}
  {method_member}
  execute_member
 '}'

data_member ::= <data_type> <member_name> ';'
method_member ::= <return_type> <member_name> '(' {args} ')' '{'
                        <body_method>
                  '}'

execute_member ::= void execute '(' ')' '{'
                        <body_method>
                   '}'
```

Figura 2.7. Sintaxis de un componente activo

Ej:

```
  Active Tipo_activo {
      float m_miembro1; // declaración de miembros privados
      …
      void subrutine1(float data) { // subrutinas adicionales
      …
      }
      …
      void execute() {
      …
      }
}
```

La definición de los componentes activos es similar pues, a la definición de una clase en C++ o Java, pero más simple.

Declaración de instancias

Similar a la declaración de instancias de clases, en el interior de un componente genérico se pueden tener varias instancias de componentes activos de un mismo tipo.

Ej:

```
Component implementation generico {
      ...
  Tipo_activo p,q,r,u;
}
```

Ejecución

Los componentes activos son responsables del flujo de ejecución de un componente, interactuando, bien con otros componentes pasivos, bien con otros componentes genéricos, o lo más importante, recibiendo peticiones de servicios o eventos del componente en el que están incluidos (como se verá posteriormente).

Tras la fase de inicialización del componente genérico, donde podrán invocarse métodos del activo (por ejemplo, para inicializar), el método *execute* será invocado automáticamente por el sistema.

Los componentes activos también permiten indicar la ejecución periódica de su método *execute*. Para ello, basta con indicarlo en la declaración de instancia del componente mediante la utilización de las palabras clave *"with period <period>"*. A partir de esa declaración, el entorno de ejecución se encargará de llamar con el periodo indicado al método *execute* del componente activo. En caso de que el método no hubiera terminado su ejecución anterior, no podrá realizarse una nueva invocación (se retrasaría hasta la terminación de la anterior).

Ej:

```
Tipo_activo p with period 30;
Tipo_activo2 q with period 50;
```

En el ejemplo anterior se declaran dos instancias de los componentes activo `Tipo_activo` y `Tipo_activo`2 con periodos de 30 y 50 milisegundos respectivamente.

Existen, no obstante, otros mecanismos para poder tener comportamientos periódicos en los componentes, como se verá posteriormente.

2.2.3. Componentes pasivos

Los componentes pasivos (*Passive*) desempeñan el papel de recursos compartidos en el interior de los componentes. No pueden iniciar ninguna acción por sí mismos y siempre están esperando por la invocación de alguno de sus métodos/servicios. Esta forma de comportamiento viene dada por los recursos compartidos convencionales, de forma, que posteriormente pueda realizarse también el análisis de tiempo real.

Los componentes pasivos proporcionan acceso en exclusión mutua a sus diferentes servicios y además, proporcionan mecanismos de techo de prioridad. Estos mecanismos son automáticamente proporcionados por la plataforma de ejecución del componente de forma transparente.

Un componente genérico puede contener, al igual que con los componentes activos, cualquier número arbitrario de componentes pasivos.

Definición

La parte de declaración incluye el nombre del tipo del componente pasivo y los métodos ofertados por el pasivo. Al igual que en el caso de los componentes activos, se pueden incluir también datos miembros privados, pero no contener a otros componentes, ya sean genéricos, primitivos o pasivos. Sin embargo, se puede acceder a otros componentes genéricos o pasivos contenidos por el genérico que contiene a su vez al pasivo.

```
Passive <pasive_name> '{'
  {data_member}
  {method_member}
'}'

data_member ::= <data_type> <member_name> ';'

method_member ::= <return_type> <member_name> '(' {args} ')'
                  '{'
                       <body_method>
                  '}'
```

Figura 2.8. Sintaxis de un componente pasivo

Ej:

```
  Passive Tipo_passive {
      float m_dato;     // declaración de miembros privados

      void set_dato(float dato) {
            m_dato = dato;
      }

      float get_dato() {
            return m_dato;
      }
}
```

La definición de los componentes pasivos es similar a la definición de componentes activos, pero sin incluir método *execute*.

Declaración de instancias

La declaración de los componentes pasivos es también realizada en la parte de implementación de los componentes genéricos siendo similar a la declaración de variables.

Ej:

```
      Component implementation generico {
            ...
        Tipo_pasivo s1,s2;
      }
```

Ejecución

Los componente pasivos no pueden iniciar ninguna acción de manera independiente. Siempre están esperando por la invocación de alguno de sus métodos para comenzar acciones. Los métodos del componente pasivo podrán ser accedidos desde otros componentes activos, asignando el sistema automáticamente un techo de prioridad y proporcionando exclusión mutua durante la ejecución de alguno de los métodos.

2.3. Interacciones entre componentes

El modelo permite la comunicación entre componentes a través de interfaces y eventos mediante la utilización de tres primitivas de comunicación: `wait`, `call` y `raise`. El

desarrollador únicamente puede utilizar estas tres primitivas para realizar la comunicación entre los componentes, por lo que, se facilita el análisis de tiempo real, ya que, la utilización de estas primitivas "marca" puntos de planificación que pueden ser identificados para, como se verá posteriormente, la extracción del modelo SDL y para la realización del análisis de tiempo real.

Es importante resaltar que un componente solamente puede suspender su ejecución por la utilización de estas primitivas. De hecho, el análisis de planificabilidad va a utilizar los bloques de código secuencial que quedan entre pares `wait/call/raise`. El código de los componentes puede verse por tanto, tal y como se muestra en la siguiente expresión:

```
({wait|call|raise}.seqblk.{wait|call|raise})*
```

La Figura 2.9 muestra un ejemplo de código de un componente activo y la existencia de distintos bloques secuenciales entre las primitivas de sincronización `wait` y `call`.

```
Active Tipo_activo {
…
void execute(){
  …
  seq_block1;
  wait I1.op1(…) {}
  if (cond)
      wait I2.op2(…);
    else
      call o2.op2(…);
  seq_block3;
  …
}
```

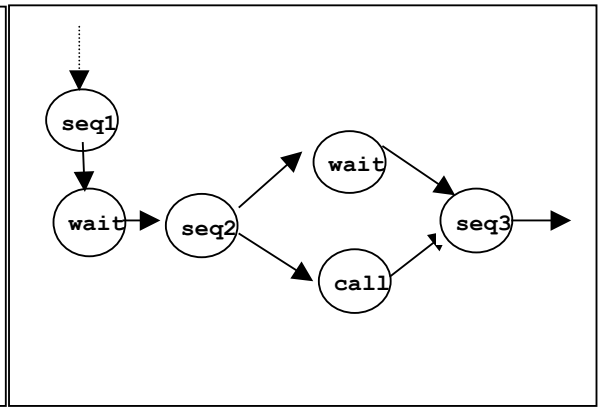


Figura 2.9. Puntos de sincronización

2.3.1. Recepción de peticiones: *wait*

La primitiva *wait* es utilizada para la recepción de peticiones al componente, ya sea en forma de servicios o de eventos. Esta primitiva es utilizada en los componentes primitivos activos, indicando el servicio o servicios sobre los que se está esperando una nueva petición. El componente activo que ejecuta el *wait* queda suspendido al ejecutar la primitiva, despertando cuando se recibe alguna de las peticiones sobre las que está esperando.

En el caso de los eventos el comportamiento es similar, el componente activo queda suspendido hasta que se produce un nuevo evento del tipo indicado. Al utilizar la primitiva, pueden mezclarse indistintamente esperas sobre servicios o sobre eventos.

La sintaxis de la primitiva *wait* es la siguiente:

```
wait{[<interf_name>.<method>|<event>](<args>)
    | Time <time>}{<blk>}}
```

Figura 2.10. Sintaxis de la primitiva *wait*

El componente que utilice la primitiva *wait* se quedará suspendido hasta que se produzca una llamada sobre las interfaces o eventos indicados. Al utilizar la primitiva, se utiliza la signatura de los servicios o eventos, de forma que se dispondrá de los argumentos indicados y podrán utilizarse a continuación en el bloque de código asociado.

Ej:

```
interface I1 {
      void metodo1(in short arg1,
                   in short arg2,
                   in short arg3);
      void metodo2(in short arg1);
      ...
}

Active miactivo {
  void execute() {
    short var_aux;

    wait I1.metodo1(arg1,arg2,arg3) {
                var_aux=arg1+arg2+arg3;
          }
          I1.metodo2(arg1); {
                var_aux=arg1+1;
          }
  }
}
```

En el ejemplo anterior el componente activo espera sobre los métodos `metodo1` y `metodo2` de la interfaz `I1`. El primero de los métodos dispone de tres argumentos de tipo `short`; `arg1, arg2 y arg3`, los cuales están disponibles tras la recepción de la invocación en el bloque asociado.

Existen dos restricciones asociadas a la primitiva *wait* y las interfaces, no así a los eventos. Como restricción sobre CORBA-IDL, las operaciones son todas de tipo `void`, si bien esto no supone ningún problema al poder utilizarse argumentos de tipo

`out` o `inout` de CORBA-IDL. La segunda restricción afecta al modelo de ejecución. El modelo no permite esperar por un mismo servicio utilizando la primitiva *wait* en más de un componente activo a la vez. Es decir, un servicio sólo puede ser tratado en un único componente activo. Esta restricción es importante para simplificar el modelo, evitando confusiones, como por ejemplo con las variables de salida, ¿qué valor deberían tomar si se trata el `wait` en más de un activo al mismo tiempo?

La primitiva *wait* permite también indicar un tiempo opcional en milisegundos, de forma que si transcurre el tiempo indicado sin que se invoque ninguno de los servicios, el componente activo despertará ejecutando el bloque asociado.

Ej:

```
wait I1.metodo1(arg1,arg2,arg3) {
      short var_aux;

      var_aux=arg1+arg2+arg3;
      }

      Time 1000 {
      int timeout=1;
      }
```

La indicación de tiempos en los *wait* puede ser una forma de tener comportamientos periódicos en los componentes activos. Si bien, se verán otros mecanismos más adecuados para conseguir esto.

2.3.2. Invocación de servicios: *call*

La primitiva *call* es utilizada para invocar servicios ofrecidos en alguna interfaz de un componente genérico. Esta primitiva es utilizada en los componentes primitivos indicando el servicio que se desea invocar.

La sintaxis de la primitiva *call* es la siguiente:

```
call <ref>[.<interf_name>].<method>(<parameters>)
     [Time <time>]
```

Figura 2.11. Sintaxis de la primitiva *call*

El desarrollador debe utilizar una referencia a un componente, bien por ser una interfaz requerida por el componente, o bien, por estar incluido el componente llamado en la parte de implementación del componente llamante.

El componente que ejecuta la primitiva puede quedar o no bloqueado dependiendo de la definición del servicio. Es decir, si se utiliza la palabra clave `oneway` de CORBA-IDL el servicio será invocado de forma asíncrona, sin esperar la finalización del mismo. En este caso, los argumentos tienen que ser todos de entrada.

Ej:

```
interface I1 {
      void metodo1(in short arg1,
                   in short arg2,
                   in short arg3);
      oneway void metodo2(in short arg1);
      ...
}


Active miactivo {
  void execute() {
    short arg,arg2,arg3;

    arg1=1;
    arg2=2;
    arg3=3;
    call o1.metodo1(arg1,arg2,arg3);
    call o1.metodo2(arg3);
  }
}
```

En el ejemplo anterior se realizará la llamada al método `metodo1` utilizando la referencia `o1` y se realiza también la llamada al método `metodo2`, que en este caso es asíncrono.

En el caso de tener argumentos de salida, tras la invocación, las variables utilizadas como argumento contendrán los resultados.

La primitiva *call* permite también indicar un tiempo opcional en milisegundos, de forma que si transcurre el tiempo indicado sin que se haya ejecutado el servicio, el componente verá interrumpida la llamada.

En el caso de situaciones erróneas, el sistema proporciona un miembro predefinido **`completed`** para cada componente, que puede consultarse tras la realización de un *call* indicando si la llamada se completó satisfactoriamente. En el caso de operaciones asíncronas, no hay forma de conocer si la llamada se realizó satisfactoriamente.

Ej:

```
call o1.metodo1(arg1,arg2,arg3) Time 1000;

if (this->completed) { ... }
```

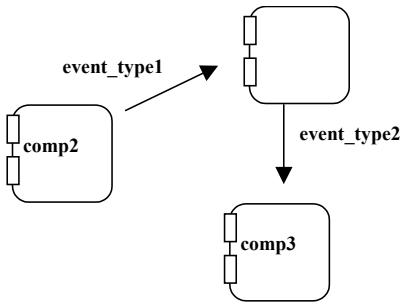
Siguiendo con el ejemplo anterior, se ha vuelto a invocar el primer método con un tiempo límite de 1000 milisegundos.

2.3.3. Generación de eventos: *raise*

Además de la interacción entre componentes mediante interfaces, el modelo proporciona un mecanismo de comunicación basado en eventos asíncronos. De esta forma, en un sistema habrá componentes que producirán eventos y habrá otros componentes que consumirán esos eventos. En el modelo de eventos propuesto, un evento puede ser consumido por todos los componentes que estuvieran esperando por eventos de ese tipo.

Existen diferentes tipos de eventos diferenciados por su nombre y por su signatura. Los componentes indican en su declaración qué tipos de eventos consumen y producen.

La Figura 2.12 muestra un ejemplo donde el componente `comp1` genera eventos de tipo `event_type1`, que son consumidos por el componente `comp2`, que a su vez, también genera eventos del tipo `event_type2` que son consumidos por el componente `comp3`.




```
Component comp1 {
  publishes event_type1;
}

Component comp2 {
  consumes event_type1;
  publishes event_type2;
}

Component comp3 {
  consumes event_type2;
}

event event_type1(<parameters>);
event event_type2(<parameters>);
```

Figura 2.12. Producción y consumo de eventos

El consumo de eventos se realiza a través de la primitiva *wait* como se explicó anteriormente. Por el contrario, para producir eventos hay que utilizar la primitiva *raise* indicando el tipo de evento producido con los parámetros adecuados respetando la signatura del evento. La sintaxis de esta primitiva es la siguiente:

```
raise <event>(<parameters>
```

Figura 2.13. Sintaxis de la primitiva *raise*

En este caso no hay que utilizar referencia de componentes, basta con indicar el nombre del evento a lanzar.

El componente que ejecuta la primitiva no queda bloqueado, pudiendo continuar su ejecución inmediatamente, en este sentido son similares a los servicios asíncronos (`oneway`). No puede conocerse si el evento ha sido consumido y además los argumentos tienen que ser todos de entrada.

Ej:

```
event event_num(in short n);

Active miactivo {
  void execute() {
      ...
      raise event_num(3);
  }
}
```

En el ejemplo anterior se muestra la definición de un evento con argumento numérico, y a continuación se muestra como puede generarse un evento de ese tipo. El consumo del evento puede realizarse como se muestra a continuación:

Ej:

```
wait event_num(n) {
      short temp;
      temp = n+1;
}
```

2.4. Anotaciones WCET

Los componentes quedan suspendidos en primitivas *wait* esperando la invocación del servicio o evento asociado a través de las primitivas *call/raise*. Desde el punto de vista

del análisis de tiempo real, estas primitivas funcionan como puntos de sincronización entre las tareas del sistema. Por otra parte, para la realización del análisis de planificabilidad de un sistema es necesario conocer los peores tiempos de ejecución (WCET) de los constituyentes del mismo.

UM-RTCOM utiliza los bloques secuenciales existentes entre las parejas *wait/call* o *wait/raise* para realizar el cálculo de los WCET de un sistema. Estos bloques son "secuenciales puros" en el sentido de que sólo contienen código secuencial y no puede haber en su interior otras fuentes de bloqueo por utilización de recursos.

El cálculo del peor tiempo de ejecución depende, no obstante, de la plataforma final de ejecución, por lo que no bastaría con realizarlo en una sola plataforma. Hacen falta mecanismos adicionales que permitan la reutilización fácil del componente.

El proceso de encontrar los WCET no es un problema resuelto, especialmente si la información del código fuente, como es el caso de los componentes, no está disponible. La aproximación utilizada es utilizar anotaciones que el desarrollador del componente tiene que proporcionar, permitiendo de esta forma el cálculo de los peores tiempos de ejecución de manera semi-automática en las diferentes plataformas finales.

Según lo descrito anteriormente sobre los bloques secuenciales y las primitivas de sincronización del modelo, las anotaciones se van a incorporar al metamodelo SDL del componente y, en concreto, a los bloques secuenciales, que son las partes de las que se quieren conocer estos WCET.

Las anotaciones van a permitir acotar el tiempo máximo de ejecución de las diferentes sentencias existentes en un programa [Shaw, 1989]: bucles, sentencias selectivas, asignaciones, interacción con el hardware, etc. En general, se elimina cualquier comportamiento temporal no predecible como, por ejemplo, bucles no acotados o sentencias similares.

2.4.1. Anotaciones sobre bloques de código

Las anotaciones más simples son las que se realizan sobre un fragmento de código indicando el peor tiempo de ejecución. Estas anotaciones indican explícitamente un peor tiempo de ejecución sobre un fragmento de código, pudiendo ser útiles en situaciones de prueba y depuración de un sistema o en casos de interacción con el hardware donde se conozca el máximo tiempo que puede durar una interacción con el hardware.

Para realizar estas anotaciones simplemente hay que utilizar la etiqueta WCET indicando el tiempo máximo:

```
 [WCET <time>]
{sentence}
[END WCET]
```

Figura 2.14. Anotación WCET <time>

En el siguiente fragmento de código se indica que el máximo tiempo de ejecución de las sentencias contenidas entre las etiquetas `[WCET]` y `[END-WCET]` es de 1300 milisegundos.

Ej:

```
[WCET 1300]
...// sentencias
[END WCET]
```

2.4.2. Anotaciones sobre sentencias condicionales

Para el cálculo de los WCET en sentencias de tipo condicional, hay que tener en cuenta siempre la peor rama de ejecución. Las anotaciones sobre este tipo de sentencias van a indicar a la herramienta cuál es la peor rama de todas las posibles, y posteriormente, se procederá a calcular el peor tiempo de ejecución para dicha rama.

Para la indicación de la peor rama, basta con escribir una etiqueta `[WCET IF]` antes del código asociado a la peor rama. Esta etiqueta será posteriormente interpretada por las herramientas para la elección de la peor rama.

Ej:

```
if (condicion) {
...
} else [WCET IF] {
      ...
}
```

2.4.3. Anotaciones sobre bucles

En el caso de los bucles, para el cálculo de los WCET hay que indicar el máximo número de iteraciones que un bucle puede dar, no estando permitidos los bucles no acotados en cuanto a número de iteraciones.

La etiqueta `[WCET LOOP <iters>]` permite indicar el número máximo de iteraciones que un bucle puede realizar. Con esta información, las herramientas pueden ahora calcular el tiempo máximo de las sentencias del cuerpo del bucle y aplicar el número máximo de repeticiones que se pueden producir del bucle.

En el siguiente ejemplo se muestra la utilización de esta etiqueta, indicando un máximo de 30 iteraciones:

Ej:

```
[WCET LOOP 30]
while (condicion) {
      ...
}
```

2.4.4. Creación de bloques secuenciales

El sistema permite la creación de bloques secuenciales "puros" artificiales. Esta división, puede ser útil en diversas circunstancias, como por ejemplo, el cálculo de los peores tiempos de ejecución de un conjunto de sentencias determinados.

Para la creación de un nuevo bloque secuencial, basta con utilizar la etiqueta `[WCET NEW]` en el punto en el que se quiera crear un nuevo bloque secuencial.

Ej:

```
wait(...);
...
[WCET NEW]
call(...);
```

2.4.5. El proceso de cálculo de los WCET

Tras la realización de las anotaciones sobre el código, las herramientas podrán obtener un modelo SDL anotado. En la fase final de utilización de un componente, la información contenida en el metamodelo permitirá probar los distintos bloques secuenciales del componente en una plataforma concreta. Para ello, y utilizando mecanismos de reflexión se permitirá la invocación de los distintos bloques secuenciales en las peores situaciones posibles. Desde este punto de vista, el código de los componentes es desplegado y dividido en bloques secuenciales puros donde no hay

llamadas, bucles o sentencias de selección y solamente se pueden realizar acciones secuenciales.

La Figura 2.15 muestra un ejemplo de código de componente activo anotado y el modelo SDL resultante incluyendo también las anotaciones. En la Figura 2.16 se muestra cómo los bloques secuenciales entre las primitivas de sincronización tienen que ser probados para obtener los peores tiempos de ejecución y poder realizar así el análisis.

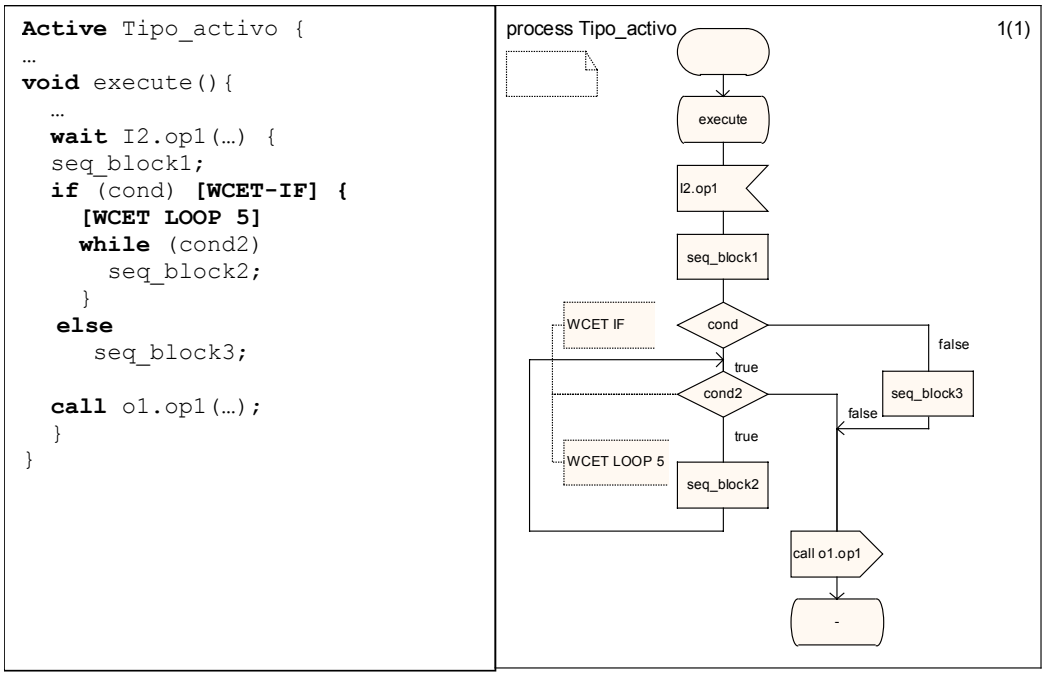


Figura 2.15. Componente activo y modelo SDL equivalente

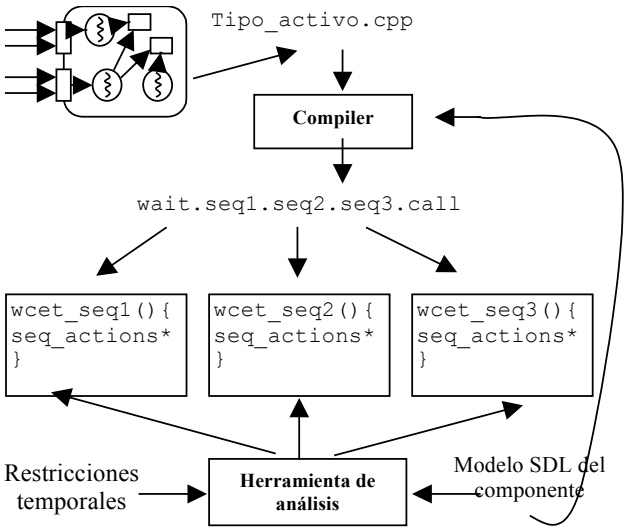


Figura 2.16. Contexto de la obtención de WCET

2.5. Vista del usuario: Instanciación de componentes

La vista del desarrollador de componentes tiene en cuenta la creación de componentes mediante la utilización de elementos primitivos y su combinación. Asimismo, también se encarga de la interacción entre componentes que, posiblemente, pueden no estar disponibles en tiempo de ejecución. Por último, el desarrollador tiene que realizar anotaciones sobre el código de forma que al final de todo este proceso se dispone de un componente independiente que puede ser utilizado por cualquier usuario teniendo en cuenta solamente la definición de los componentes.

El primer paso para la utilización de un componente es la declaración de instancias del mismo. El usuario del componente no dispone del código fuente y sólo utilizará los componentes a través de sus interfaces o eventos. Tras la declaración de las instancias, el usuario tendrá que configurar el componente, indicar restricciones temporales sobre algunos de sus elementos, y finalmente realizar la conexión de las interfaces de entrada y salida con otros componentes. La declaración de instancias se realiza de forma similar a la declaración de variables.

Tras la realización de todos los pasos indicados anteriormente (configuración, restricciones, composición), se podrán utilizar todos los métodos ofertados por las interfaces de entrada del componente a través de la utilización de la primitiva de sincronización `call`.

Ej:

```
Component A {
      interface I1 {
            void metodo1();
      }
}
... // Configuración, restricciones y composición

Component implementation B {

      // Declaración de instancias
      A instancia_a;


      Passive mipasivo() {

        void metodo1() {
            call instancia_a.I1.metodo1();
        }

      }
}
```

2.6. Vista del usuario: Configuración de componentes

Configurar un componente es el proceso de proporcionar la información dependiente del dominio del problema y la plataforma donde se va a usar el componente, de forma que sea posible utilizarlo en un dominio y plataforma concretas.

En el modelo UM-RTCOM, el creador del componente dispone de ranuras de configuración (*configuration slots*) en una sección de la definición del componente (sección visible al usuario, al igual que las interfaces) con todos los aspectos configurables del componente, de forma que el usuario del componente proporciona valores para todos los parámetros indicados, adaptando el componente a una situación concreta.

De esta forma, se consigue un mayor grado de reutilización en los componentes de tiempo real que, tradicionalmente, tenían que ser modificados atendiendo a las diferentes plataformas finales.

Mediante las ranuras de configuración se le ofrece al usuario la posibilidad de usar un mismo componente en muy diferentes situaciones sin tener que realizarle modificaciones.

2.6.1. Definición de ranuras

Las ranuras de configuración se indican en la parte de definición de los componentes, como puede verse en la Figura 2.17:

```
Component <comp_name> '{'

  {interface_definition}
  {input_interface}
  {output_interface}
  {consumed_event}
  {published_event}
  {config_slots}

'}'

config_slots ::= config  '{'
                         {slot_definition}
                         '}'


slot_definition ::= <slot_type> <slot_name>';'
```

Figura 2.17. Definición de ranuras de configuración

Ej:

```
Component Nombre_componente {
   config {
      float slot1;
      int slotn;
      }
}
```

Existe una sección especial (`config`) en la definición de los componentes. En esa sección se indicarán una lista de pares <tipo,nombre> donde el nombre indica el nombre de la ranura de configuración, y el tipo debe ser un tipo primitivo, como por ejemplo: `int`, `float`, `double`, etc.

2.6.2. Utilización de ranuras

Los nombres de los parámetros de configuración pueden ser utilizados en el código del componente como si fueran constantes. El desarrollador del componente es el responsable de la correcta utilización de estas ranuras de configuración en el código del componente, para hacer que éste pueda ser utilizado en diferentes situaciones y plataformas. El usuario del componente únicamente dispone de las ranuras de configuración y de la información que sobre éstas proporciona el desarrollador.

El siguiente ejemplo muestra cómo puede definirse un comportamiento periódico en un activo a través de la utilización de una ranura de configuración.

Ej:

```
Component Controlador {
      config {
            int periodo_sensor;
      }
...
}

Active periodico {
  int mivar;

  void init() {
    mivar=0;
  }

  void execute() {
    wait Time periodo_sensor {
      var++;
      }
  }
}
```

En el ejemplo se definió una ranura de configuración denominada `periodo_sensor`. Esta ranura es posteriormente utilizada en el método *execute* de un componente activo contenido en el componente `Controlador`.

2.6.3. Instanciación de ranuras

Las acciones anteriores (definición y utilización) son realizadas por el desarrollador del usuario. El usuario del componente únicamente tiene que "rellenar" estas ranuras durante la declaración de instancias de los componentes.

El modelo, proporciona un método predefinido *configure*, adecuadamente sobrecargado que permite al usuario del componente suministrar una lista con todos los valores de los parámetros de configuración.

```
Nombre_componente.configure(<slot values>);
```

Figura 2.18. Método *configure* para ranuras de configuración

Siguiendo con el ejemplo anterior, la utilización del método *configure* quedaría como se indica a continuación:

Ej:

```
Controlador micomponente;
micomponente.configure(30);
```

De esta forma, se dispondría de un componente activo con periodo de 30 milisegundos.

Tras la utilización del método *configure,* el entorno de desarrollo/ejecución se encarga de sustituir los valores de las ranuras por los suministrados por los usuarios. Una vez proporcionados estos valores, ya no pueden alterarse.

2.7. Restricciones de tiempo real

La posibilidad de indicar restricciones de tiempo real es una de las principales aportaciones del modelo, no contemplada en los modelos tradicionales de componentes.

En UM-RTCOM, los usuarios de los componentes van a poder indicar restricciones de tiempo real al utilizar los componentes.

Los únicos elementos accesibles para el usuario de un componente son las interfaces y los eventos. Por lo tanto, las restricciones del modelo pueden ser relativas a periodos y *deadlines* sobre la activación de métodos de interfaces o eventos.

La utilización de las restricciones y el mecanismo de ranuras de configuración permiten la propagación de las restricciones temporales desde el nivel más externo del sistema (eventos externos) a los componentes más internos. Esta forma de propagación de las restricciones, va a permitir tener una visión plana del modelo con un conjunto de hebras y recursos que pueden ser analizados de acuerdo con la extensión de tiempo real de SDL. Si bien, como ya se ha comentado, para el caso de sistemas distribuidos se requiere la realización de modificaciones sobre la extensión de tiempo real.

2.7.1. Restricciones sobre métodos de interfaces

El usuario del componente puede indicar restricciones sobre cómo va a ser la utilización de un componente en su sistema. Puede para ello indicar dos restricciones diferentes: el periodo mínimo que puede haber entre dos invocaciones de un mismo método sobre una instancia de un componente, y el tiempo máximo (*deadline*) para la ejecución de esa invocación una vez realizada.

Estas restricciones van a poder ser después aprovechadas por las herramientas de análisis para poder determinar las prioridades de los elementos de un sistema. La indicación de las restricciones no es obligatoria, por lo que en caso de que el usuario no las indicara, el sistema podría requerir alguna acción del usuario para poder calcular las prioridades o realizar el análisis de tiempo real del sistema.

La indicación de las restricciones se hace tras la declaración de las instancias de los componentes y su sintaxis es la siguiente:

```
Instance_name constraints <Interface.Method>
      Period T, Deadline T ';'
```

Figura 2.19. Restricciones sobre métodos de interfaces

Se indica el nombre de la instancia sobre la que se van a indicar las restricciones, a continuación el nombre de la interfaz y el método que van a tener las restricciones, y

ya por ultimo se indican las restricciones, que pueden ser, o bien un periodo mínimo entre invocaciones de la interfaz, o bien, el *deadline* para la ejecución de las invocaciones del método, o ambas restricciones.

Ej:

```
Sensor1 constraints I1.lectura Period 30;
Sensor2 constraints I1.lectura Period 50;
```

En el ejemplo anterior, se indican restricciones sobre dos componentes sensores, indicando que el método `lectura` tiene un periodo de 30 y 50 milisegundos respectivamente.

Si posteriormente, el usuario no cumple estas restricciones, el sistema podría no funcionar correctamente, por la indicación incorrecta de las restricciones. Es decir, el sistema puede realizar un análisis de tiempo real sobre las restricciones indicadas, y en caso de que se indique que el sistema es planificable, se garantiza su funcionamiento. Pero, en caso de ser incorrectamente indicadas las restricciones, no se realiza ninguna garantía sobre el correcto funcionamiento del sistema.

2.7.2. Restricciones sobre eventos

La indicación de restricciones sobre eventos es parecida al caso de los métodos de interfaces. Se pueden indicar para ello dos restricciones diferentes: el periodo mínimo que puede haber entre la producción de dos instancias del mismo evento, y el tiempo máximo (*deadline*) para la ejecución de esa invocación una vez realizada.

Existe una diferencia con respecto a las restricciones de métodos y los eventos. El periodo mínimo para la repetición del evento es global al sistema, al ser los eventos compartidos en todo el sistema. Sin embargo, el tiempo máximo para su tratamiento, puede depender del componente.

La indicación de estas restricciones va a poder ser también aprovechada por las herramientas de análisis, al igual que en el caso de las interfaces, pudiendo asignar prioridades para el tratamiento de los eventos del sistema.

La sintaxis para la indicación de restricciones sobre periodos de eventos es la siguiente:

```
constraints <event_type> Period T ';'
```

Figura 2.20. Restricciones sobre eventos

Se indica el nombre del tipo de evento sobre el que se va a indicar la restricción, y a continuación el periodo mínimo entre dos producciones del mismo evento.

En caso de indicaciones contradictorias por parte del usuario, el sistema utilizaría la opción más restrictiva.

Ej:

```
constraints evento_alarma Period 30;
```

En el ejemplo anterior, se indica que entre dos eventos del tipo `evento_alarma`, debe haber un tiempo mínimo de 30 milisegundos.

La indicación del plazo máximo de ejecución, depende del componente que consuma el evento, se indica pues en la parte de implementación del mismo con la siguiente sintaxis:

```
constraints <event_type> Deadline T ';'
```

Figura 2.21. Plazos de ejecución sobre eventos

Ej:

```
constraints evento_alarma Deadline 10;
```

En el ejemplo anterior, los eventos de tipo `evento_alarma` deben ser tratados antes de 10 milisegundos.

La indicación de *deadlines* en el consumo de eventos va a permitir después a las herramientas de análisis asignar diferentes prioridades a las transiciones que consuman estos eventos en diferentes componentes del sistema.

2.8. Composición de componentes

Uno de los aspectos básicos en todo modelo de componentes es la interconexión de los componentes. Mediante la composición, componentes desarrollados por diferentes desarrolladores van a poder ser "conectados" y utilizados sin mayores problemas.

En el modelo, el usuario de los componentes va a tener que indicar las conexiones de las interfaces de salida con las interfaces de entrada de los diferentes

componentes, de forma que un componente disponga de toda la funcionalidad "ofertada" por otros componentes.

Solamente tras la composición, van a poder ser utilizados los componentes en un sistema más amplio. En un sistema de tiempo real, el comportamiento temporal de los componentes puede variar tras realizar su composición, por lo que hay que realizar análisis temporal a nivel de aplicación.

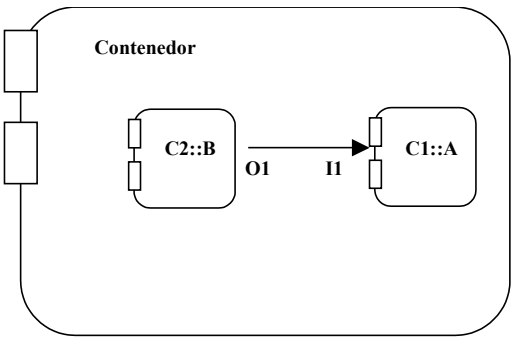
La conexión de las interfaces se realiza tras la declaración de las instancias de los componentes y su sintaxis es la siguiente:

```
<instance1>.<ref_interface1> <-> <instance2>.<interface2>
```

Figura 2.22. Composición de interfaces

Hay que indicar el nombre de la primera instancia y de la referencia de la interfaz de salida a conectar y, posteriormente, el nombre de la segunda instancia junto con la interfaz de entrada para conectar.

En la Figura 2.23 la referencia de la interfaz de salida del componente `B` de tipo `C2` (`o1`) es conectada con la interfaz de entrada del componente `A` de tipo `C1`. Tras la realización de esta composición, en todos aquellos lugares donde se hiciera referencia a la instancia `o1` de tipo de interfaz `O1`, podrán realizarse invocaciones sobre operaciones contenidas en esa interfaz.




```
Component C1 {
  interface I1 {
    void op1(<parameters>);
    void op2();
  }
      ...
  input I1;
}

Component C2 {
  interface O1 {
    void op1(<parameters>);
    void op2();
  }
      ...
  output O1 with o1;
}

Component implementation Contenedor {
      C1 A;
      C2 B;

      B.o1 <-> A.I1;
}
```

Figura 2.23. Interconexión de componentes

Como puede observarse, el nombre de las interfaces no tiene porqué coincidir al realizar la composición, sino que lo importante es el nombre de las operaciones contenidas en esas interfaces, así como su signatura.

2.9. Componentes como aplicaciones

Una aplicación del modelo estará finalmente compuesta por un conjunto de componentes interactuando entre sí a través de invocaciones de servicios o eventos producidos o consumidos.

En UM-RTCOM, una aplicación es también un componente al que se podrá requerir su ejecución desde el entorno de ejecución. Para ello, los componentes-aplicación contienen un método especial *execute*, similar al de los componentes activos. Es desde este método, donde el componente-aplicación podrá comenzar la interacción entre los diferentes componentes de la aplicación o simplemente esperar la finalización de la aplicación.

Antes de la ejecución de este método, el sistema se encargará de la creación de las instancias de componentes necesarias, ya sean componentes activos, pasivos o genéricos. En el caso de componentes distribuidos se procederá a su despliegue en las máquinas remotas. Tras este despliegue se realizará la configuración de los componentes y, finalmente, la composición de las interfaces de entrada y salida. Tras la realización de todos estos pasos, comenzará la ejecución del método *execute*, comenzando la ejecución de la aplicación.

Ej:

```
Component Aplicacion1 {
...
}

Component implementation Aplicacion1 {
      void execute() {
      ...
      }
```

Los componentes-aplicación tienen como única restricción el no tener interfaces "requeridas", disponen pues, de la posibilidad de ejecutarse sin tener que conectarse con ningún otro componente del sistema.

Desde el punto de vista de tiempo real, sólo las aplicaciones son analizables. Un componente-aplicación debe ser visto como un contenedor de componentes, donde el resto de componentes son declarados e interconectados en su interior y donde las restricciones temporales son finalmente especificadas.

2.10. Proceso de desarrollo

El proceso de desarrollo de un componente en el modelo UM-RTCOM consta de una serie de pasos bien definidos: escritura del código del componente, anotación del código para el cálculo de los WCET, extracción del modelo SDL, pruebas de tiempo y, dependiendo del uso que se vaya a hacer del componente, su composición con otros componentes o la realización de los diferentes tipos de análisis sobre el componente a través de su modelo SDL; entre ellos, el análisis de tiempo real.

Código fuente

La escritura del código fuente puede incluir las dos visiones ya comentadas: vista del desarrollador y del usuario.

El desarrollador del componente realiza la definición de las interfaces que el componente va a ofrecer y requerir, así como la indicación de eventos producidos y consumidos.

Tras la escritura de las interfaces y los tipos de eventos necesarios, el desarrollador debe escribir ahora el código para la implementación de los servicios y el consumo de los eventos; para ello, utilizará componentes auxiliares, ya sean activos, pasivos u otros componentes genéricos, actuando en ese caso como usuario de otros componentes.

El editor de componentes permite realizar la creación de componentes con todas sus tareas asociadas: creación de interfaces, escritura de código de servicios y eventos, ranuras de configuración, utilización de componentes, etc. siendo su utilización similar a la de típicos entornos integrados tales como Visual Studio, Netbeans, etc. Esta herramienta es la principal de todas las desarrolladas, ya que permite realizar todo el desarrollo de una aplicación o componente, comenzando con la creación de componentes y finalizando con su análisis.

La Figura 2.24 muestra la edición del código de los componentes activos de un componente. En la parte izquierda de la ventana pueden verse los diferentes elementos de un componente que pueden ser editados. En la parte derecha aparecerá el elemento editado en cada momento, como por ejemplo, el código de los componentes activos.

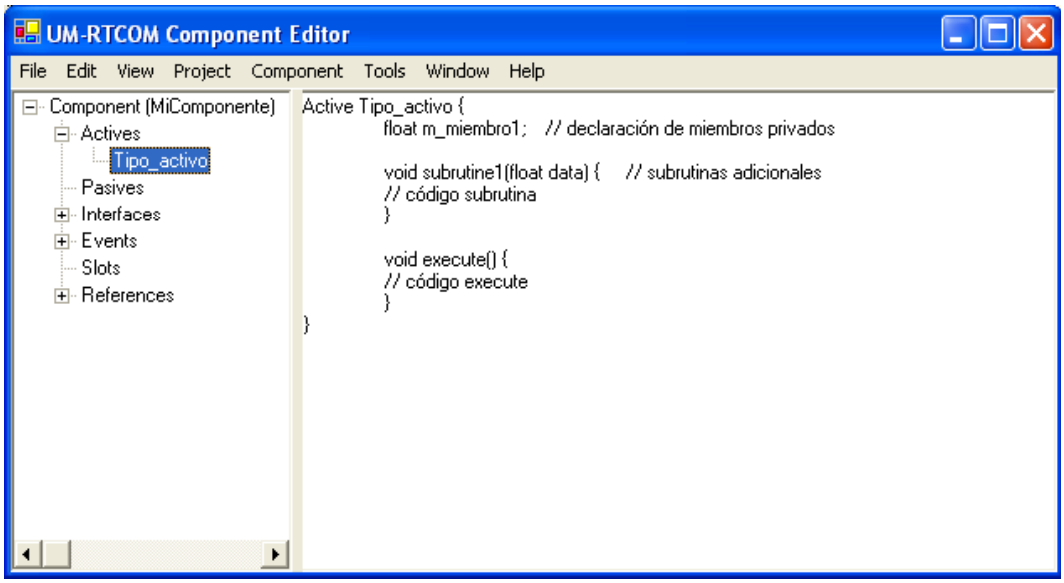


Figura 2.24. Editor de componentes

Tras la creación de los diferentes componentes, el siguiente paso sería su composición en una aplicación mediante la interconexión de las interfaces de entrada y salida, y la configuración de los componentes mediante las ranuras de configuración. En la Figura 2.25 puede verse cómo se realiza la composición de la interfaz de salida `interface1` del componente `a` con la interfaz equivalente del componente `b`.

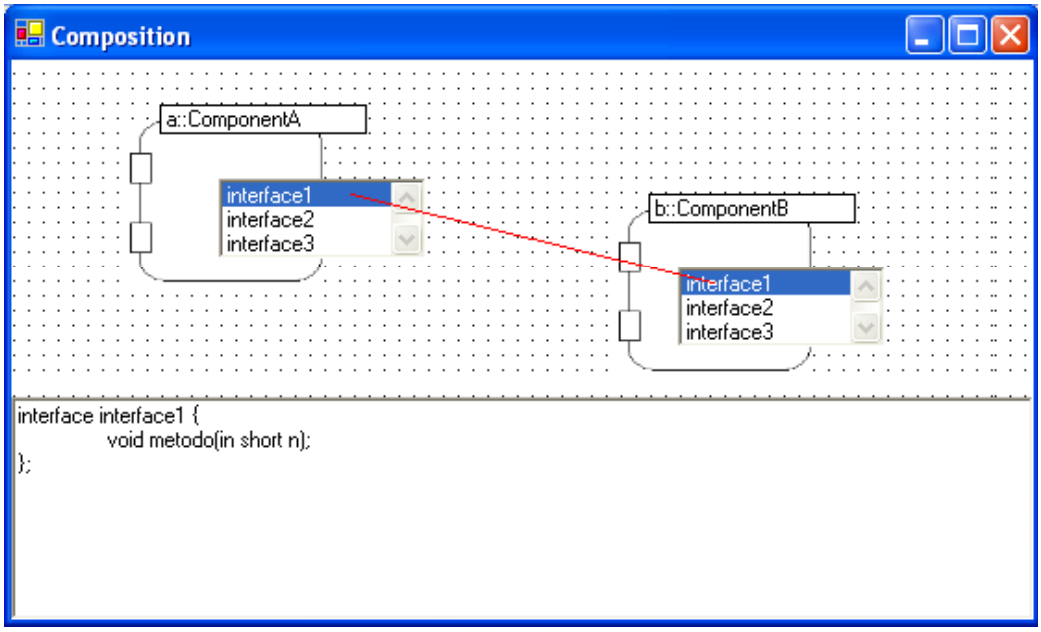


Figura 2.25. Interconexión de componentes

El editor de componentes permite realizar el despliegue de los componentes, para ello el usuario indica en esta fase la ubicación donde van a ejecutarse los diferentes componentes que forman una aplicación. Al realizarse el despliegue de la aplicación, la herramienta se encarga de recopilar información sobre dependencias entre componentes, utilizando esta información para poder establecer prioridades en los distintos componentes activos. Esta información es utilizada posteriormente por la herramienta de análisis.

El despliegue de los componentes depende de la ejecución de procesos *demonios* en las máquinas destino del despliegue. Estos procesos son responsables de distintas tareas como por ejemplo, la transferencia del código, la compilación en la plataforma destino o la realización de pruebas para la obtención de los peores tiempos de ejecución.

Anotación del código fuente

Simultáneamente, o en una fase posterior, el desarrollador del componente debe proporcionar anotaciones que permitan el cálculo de los WCET para que los componentes puedan ser analizables en diferentes plataformas finales.

La anotación del código fuente va a ayudar a las herramientas de análisis a realizar su labor, por lo que es una labor importante. Según las anotaciones realizadas, las herramientas podrán solicitar ayuda del usuario.

La anotación es realizada sobre el mismo código fuente de los componentes, por lo que puede utilizarse el editor de componentes.

Extracción del modelo SDL

Tras la escritura del código del componente y su anotación para el cálculo de los WCET, las herramientas del entorno pueden ahora extraer un modelo SDL que refleja el comportamiento del componente sin dar detalles sobre la implementación del mismo.

La función del generador SDL es la obtención del metamodelo SDL del componente. Para ello, se analiza el código fuente de los componentes, obteniendo los diferentes bloques secuenciales que lo componen, así como los puntos de sincronización basándose en la utilización de las primitivas *wait*, *call* y *raise*. Esta herramienta puede ser utilizada de manera independiente desde la línea de comandos, o bien integrada con el editor de componentes.

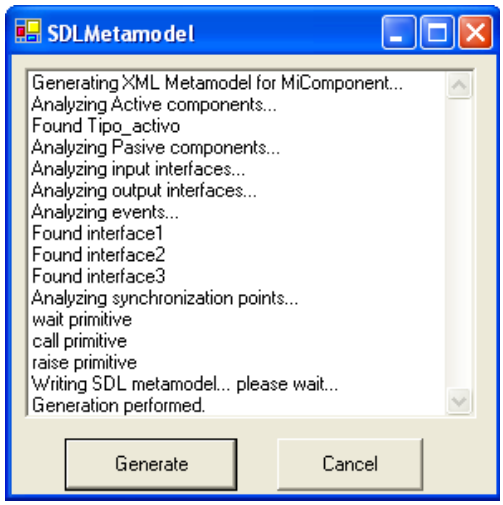


Figura 2.26. Generador SDL

La herramienta podría solicitar la intervención del usuario en caso de situaciones dudosas como, por ejemplo, la no introducción de algunas anotaciones en el código del usuario que pueden ser necesarias para la realización del análisis de tiempo real.

Pruebas temporales

El componente es analizado en la plataforma final. Para ello, y partiendo de las anotaciones hechas sobre el código y de los bloques secuenciales que existen entre las primitivas de sincronización, se calcularán los peores tiempos de ejecución en una plataforma concreta.

La herramienta de pruebas temporales, al igual que el generador del metamodelo SDL, puede ser utilizada de manera independiente desde la línea de comandos, o bien integrada con el editor de componentes.

La información resultante obtenida será almacenada para poder ser después utilizada por la herramienta de análisis. Ambas herramientas podrían haberse unificado, si bien, parece interesante el disponer de esta herramienta de manera independiente para la realización de pruebas previas al análisis.

La realización de las pruebas utiliza como base los bloques secuenciales obtenidos en la fase previa de generación del metamodelo SDL. De esta forma, se realizará la medición de los peores tiempos de ejecución utilizando para ello el modelo SDL anotado por el usuario sobre la plataforma final.

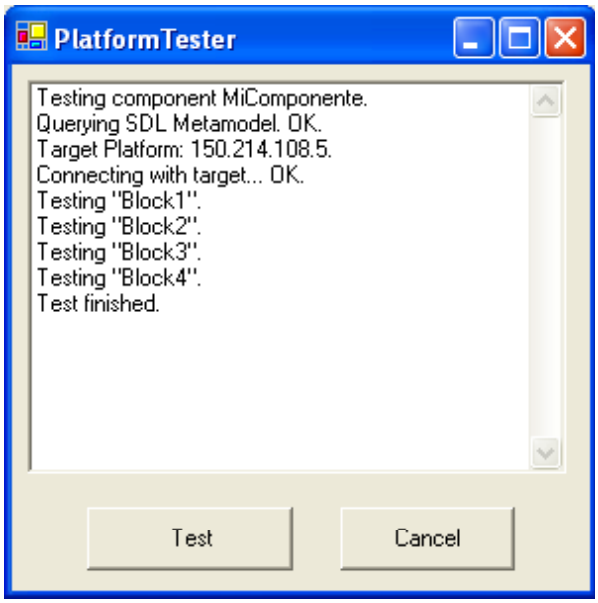


Figura 2.27. Generador de pruebas

Análisis

A través de la utilización de los WCET y utilizando la metodología de análisis que se describirá en secciones posteriores, se podrá saber si una aplicación es planificable con los componentes configurados y con la una distribución de los componentes realizada en las diferentes máquinas del sistema. La Figura 2.28 muestra un ejemplo de utilización de la herramienta de análisis.

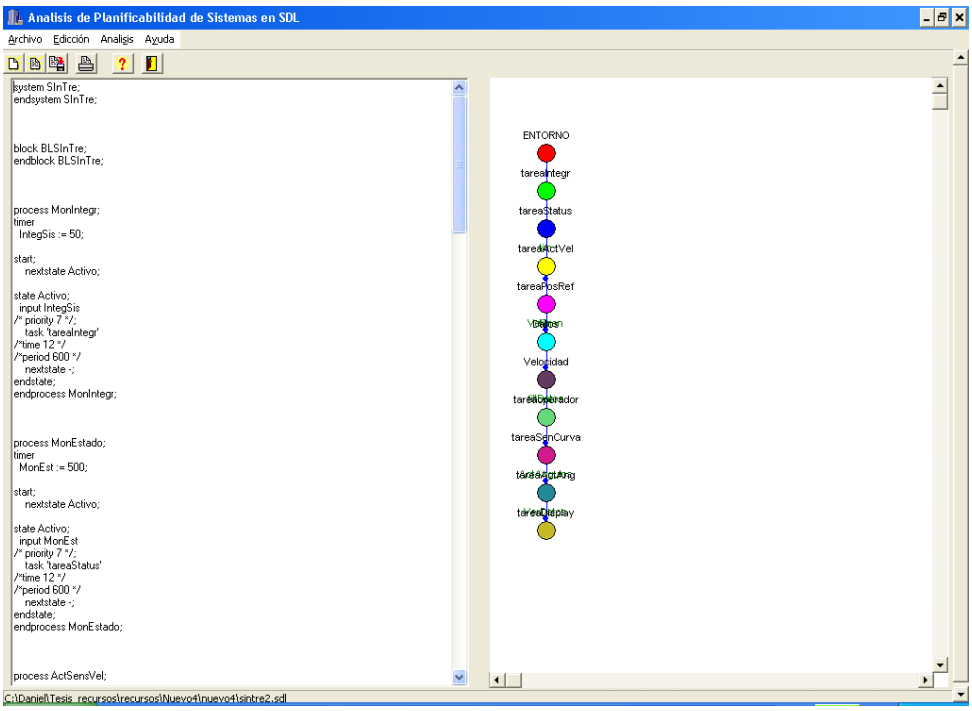


Figura 2.28. Herramienta de análisis

El proceso completo de desarrollo de un componente puede ser visto en la Figura 2.29.

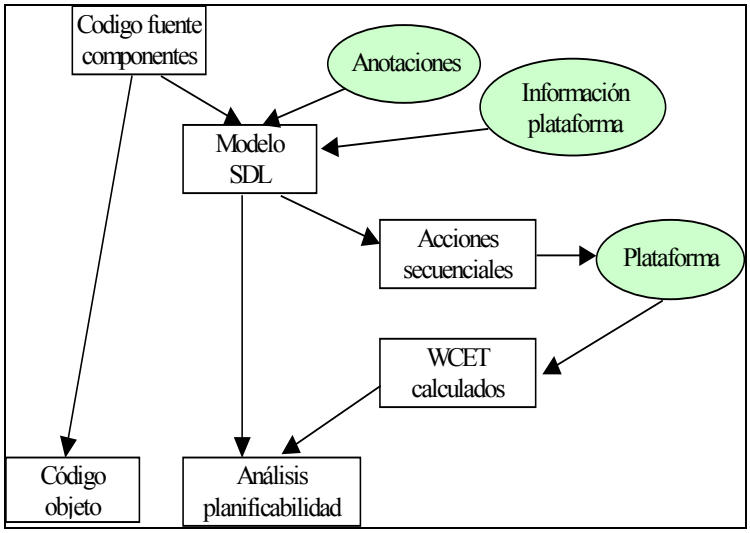


Figura 2.29. Proceso de desarrollo

2.11. Otros modelos de componentes predecibles

Se presentan a continuación algunos trabajos relacionados con componentes y tiempo real. Gran parte de estos trabajos están orientados hacia sistemas empotrados de tiempo real y en concreto a sistemas industriales no siendo factible su utilización en sistemas más genéricos, al contrario que UM-RTCOM, que sí permite esta posibilidad.

2.11.1. El modelo RTCOM

El modelo RTCOM es presentado en el proyecto ACCORD (*Aspectual Component-based Real-time system Development*) [Tesanovic et al., 2004] consistente en un método de diseño para sistemas de tiempo real basado en la descomposición de un sistema en componentes y aspectos. RTCOM es un modelo de componentes que soporta restricciones temporales, gestión de recursos y composicionalidad.

Programación orientada a aspectos

La programación orientada a aspectos (AOSD) ha emergido como una nueva forma de desarrollar software mediante la separación de aspectos que superan el ámbito de un

componente y están "entretejidos" en todo el sistema, como por ejemplo, la gestión de memoria, la seguridad, etc. [Kiczales et al., 1997]. Típicamente, una implementación AOSD incluye los siguientes elementos:

- Componentes, escritos en un lenguaje de componentes.
- Aspectos, escritos en un lenguaje de aspectos [AspectJ, 2005] y que separan los distintos "aspectos" de un sistema.
- El "tejedor" de aspectos, compilador que combina los aspectos y componentes.

Dos conceptos muy importantes en la programación orientada a aspectos son los puntos de corte (*pointcut*) y los avisos (*advices*). Los puntos de corte indican qué parte del código pueden ser modificados por los aspectos (métodos, atributos, etc.) y los avisos indican de qué manera es modificado el código, por ejemplo, insertando código antes de la ejecución de un método, reemplazando código, etc.

Los componentes utilizados en AOSD son componentes de *caja blanca*, sin ocultación de información para los usuarios del mismo. Mediante los aspectos es posible modificar el comportamiento de los componentes.

Características principales

En el modelo RTCOM los componentes son complementados con aspectos, tanto internos (de la aplicación o componente), como externos (tiempo de ejecución y composición). RTCOM es un modelo independiente del lenguaje que consta de las siguientes partes: la parte funcional, la parte dependiente del entorno de ejecución y la parte de composición. Estas diferentes partes motivan también diferentes tipos de interfaces en RTCOM.

Se consideran tres tipos de aspectos: aplicación, ejecución y composición. Los aspectos de aplicación modifican el código de los componentes. Los aspectos de ejecución son utilizados al insertar un componente en un entorno de ejecución y los aspectos de composición son utilizados para proporcionar información sobre cómo componer el componente.

Parte funcional

La parte funcional de los componentes de RTCOM consta de los denominados *mecanismos* y *operaciones*. Los mecanismos son métodos de *grano fino* o funciones de

cada componente y las operaciones son proporcionadas a otros componentes y al sistema, siendo métodos de *grano grueso* y utilizando para su implementación los mecanismos de su componente y a operaciones de otros componentes que forman parte del sistema.

Los denominados *aspectos de aplicación* pueden ser utilizados para modificar el comportamiento temporal manteniendo la posibilidad de realizar análisis temporal de tiempo real. Un ejemplo de este tipo de aspectos es la optimización del uso de memoria, aspecto muy importante en sistemas empotrados.

Los aspectos de aplicación pueden ejecutarse antes o después de una operación, o incluso pueden reemplazar su código. Los aspectos de aplicación de RTCOM son implementados utilizando mecanismos de los componentes. Es decir, en RTCOM, las operaciones pueden variar dependiendo de los aspectos utilizados, sin embargo, los mecanismos son fijos.

La posibilidad de poder modificar las operaciones es vital en el modelo RTCOM para realizar análisis temporal.

Parte dependiente del entorno de ejecución

RTCOM tiene en cuenta el comportamiento temporal de la parte funcional de los componentes al ser insertados en una plataforma de ejecución. Para ello, utiliza los denominados *aspectos de ejecución*. Estos aspectos son expresados como atributos de operaciones, mecanismos y aspectos de aplicación.

Un ejemplo muy importante de este tipo de aspectos en RTCOM es el cálculo de los WCET. Para ello, se expresan una serie de atributos sobre el número de veces que se utilizan mecanismos, tiempo empleado por los mecanismos, etc. Se suponen conocidos los peores tiempos de los mecanismos y de partes de los métodos, y se proporciona una herramienta que según esta información contenida en los aspectos, permite calcular los WCET.

Parte de composición de RTCOM

La parte de composición se refiere ambas partes: funcional y de ejecución y hace referencia a la composicionalidad de los componentes con respecto a diferentes aspectos de aplicación, así como en relación a diferentes componentes.

2.11.2. El modelo SaveCCM

El modelo SaveCCM [Hansson et al., 2004] está centrado en el desarrollo de aplicaciones de control empotradas en sistemas de vehículos. SaveCCM es un modelo simple en el que la flexibilidad está limitada para facilitar el análisis de tiempo real. SaveCCM está siendo desarrollado en el proyecto SAVE (*Component Based Design of Safety Critical Vehicular Systems*). La idea básica es centrarse en la simplicidad y analizabilidad de tiempo real y en los atributos de calidad para proporcionar un soporte eficiente en el diseño e implementación de sistemas de vehículos.

SaveCCM consta de los siguiente elementos principales:

- Componentes: unidades básicas de comportamiento.
- *Switches*: que proporcionan facilidades para cambiar dinámicamente la infraestructura de conexión entre componentes.
- *Assemblies*: creación de componentes agregados de conjuntos de componentes interconectados y *switches*.
- Marco de trabajo de ejecución: proporciona un conjunto de servicios, como por ejemplo la comunicación entre componentes.

Interfaces funcionales

Las interfaces funcionales son definidas en términos de *puertos*, que son puntos de interacción entre el elemento y su entorno exterior. Se distinguen entre puertos de entrada y salida. SaveCCM distingue además entre puertos de datos, de disparos (*triggers)* y mixtos. La composición en el modelo se realiza mediante la conexión de puertos coincidentes en los tipos de datos o de disparos.

Modelo de ejecución

El modelo de ejecución de SaveCCM es bastante restrictivo al ser objetivos prioritarios la predecibilidad y analizabilidad. Utiliza un paradigma de flujo de control (tuberías y filtros) en el que las ejecuciones son disparadas por relojes o eventos externos y donde los componentes tienen un tiempo finito, posiblemente variable, de ejecución.

En el modelo, un componente está siempre esperando para ser activado o ejecutando. Un componente pasa de espera a ejecución cuando todos los puertos de entrada de disparos están activos.

En una primera fase de su ejecución un componente lee todas sus entradas. En la segunda fase, el componente realiza sus computaciones basándose únicamente en sus entradas y en su estado interno. En la tercera fase, el componente genera salidas, volviendo al estado de espera por nuevos disparos.

Otras características

- Entradas/Salidas externas: se accede a sensores y actuadores a través de componentes que los encapsulan.
- Tiempo: se asume un reloj global perfecto que puede ser accedido a través de componentes especiales.
- *Switches*: los *switches* proporcionan los mecanismos de transferencias de datos y/o disparos entre componentes, permitiendo la configuración de *assemblies*. Un *switch* contiene una especificación de conexión que consiste en un conjunto de patrones de conexiones definiendo una forma específica de conectar los puertos de entrada y salida del *switch*.
- *Assemblies*: permiten formar componentes agregados desde componentes y switches. Son encapsulaciones de componentes y *switches* con una interfaz funcional externa. Un *assembly* no es un componente en el modelo SaveCCM.
- Atributos de calidad: se incluye una lista de atributos de calidad y (posiblemente) sus valores en la especificación de componentes y *assemblies*.

2.11.3. VEST

VEST [Stankovic, 2001] proporciona un entorno para la construcción y análisis de sistemas empotrados distribuidos de tiempo real. VEST ayuda a los desarrolladores a seleccionar o crear componentes de software pasivos, integrarlos en algún producto, traducir los componentes pasivos en estructuras activas tales como *threads* y traducir los *threads* en hardware específico. Todo ello realizando comprobaciones de dependencias y análisis no funcionales para ofrecer tantas garantías como sea posible incluyendo rendimiento de tiempo real y fiabilidad.

El entorno VEST está compuesto de cinco librerías, un conjunto de comprobaciones de aspectos y un entorno gráfico para la composición y análisis de productos empotrados.

Librerías de componentes

Las librerías de VEST contienen software y descripciones de componentes hardware y redes. Los componentes de VEST pueden ser *abstractos* o *actuales*. Un componente abstracto es una entidad de diseño que representa los requisitos; por ejemplo, un temporizador con ciertos requisitos o un procesador genérico. Un componente actual es la implementación o descripción de una entidad reutilizable. Por ejemplo, un módulo de un temporizador escrito en C o un procesador concreto.

La información sobre los componentes abstractos incluye sus interfaces y requisitos. La información para los componentes actuales incluye categorías tales como información de enlazado, localización de código fuente, peor tiempo de ejecución, memoria, etc. incluyendo toda la información necesaria para analizar dependencias entre elementos del sistema. La información de ambos tipos de componentes es contenida de manera que puede consultarse de forma reflexiva, siendo ésta una de las principales características de VEST así como su extensibilidad.

Para soportar el diseño completo de los sistemas empotrados, VEST implementa cuatro librerías de componentes: la librería de aplicación, la librería del *middleware*, la librería del sistema operativo y una librería hardware.

Librería de aspectos prescriptivos

Los aspectos prescriptivos son *avisos* en el sentido de la AOSD independientes del lenguaje de programación y reutilizables, los cuales pueden ser aplicados a un diseño. Por ejemplo, un desarrollador puede invocar un conjunto de aspectos prescriptivos en la librería para sumar un cierto mecanismo de seguridad a un producto.

Comprobaciones de aspectos

VEST implementa un conjunto de comprobaciones de aspectos para componentes, tanto en el interior de un componente, como entre varios componentes. Un desarrollador puede aplicar estas comprobaciones a un diseño para descubrir errores causados por dependencias entre componentes.

Un aspecto que puede comprobarse en VEST es el análisis de planificabilidad tanto para sistemas monoprocesador como para sistemas distribuidos empotrados, cubriendo, por lo tanto, una amplia gama de sistemas de tiempo real.

Entorno de composición

VEST proporciona un entorno de desarrollo que permite a los desarrolladores componer sistemas distribuidos empotrados desde los componentes, realizando comprobaciones de dependencias e invocando aspectos prescriptivos sobre el diseño.

2.11.4. Modelos de componentes industriales

En esta sección se presentan algunos modelos de componentes utilizados en entornos industriales. Carecen del carácter genérico de UM-RTCOM y están orientados a un dominio de problema muy concreto, pero algunos de ellos presentan características interesantes desde el punto de vista de tiempo real.

Koala

Koala [Ommering et al., 2000] es un modelo de componentes y un lenguaje de descripción de arquitecturas utilizado satisfactoriamente para el desarrollo de dispositivos electrónicos de consumo. Koala está desarrollado por Philips y es utilizado en televisores, videos, reproductores, etc.

Un componente de Koala es un fragmento de código que puede interactuar con su entorno únicamente a través de interfaces explícitas. El código está basado en C y en jerarquías de directorios.

Las conexiones entre componentes son expresadas en términos de interfaces que son descritas como un conjunto pequeño de funciones relacionadas semánticamente. Koala identifica interfaces requeridas y proporcionadas. Los componentes pueden tener además múltiples interfaces. Koala proporciona además el interesante concepto de interfaces requeridas opcionales que pueden estar conectadas o no a otros componentes que las proporcionan permitiendo a un componente conocer qué ofrece su entorno.

En el diseño es posible indicar la ordenación de las tareas incluyendo relaciones de precedencia y exclusión mutua, pero no hay soporte para propiedades temporales ni para modelar el rendimiento.

La infraestructura de ejecución consiste en una imagen de un proceso simple, construida sobre un núcleo de tiempo real con planificación expulsiva que separa tareas de alta frecuencia y de baja frecuencia.

Rubus

Rubus [Norström et al., 2001] es un Sistema Operativo de Tiempo Real pequeño desarrollado por Arcticus Systems AB. Rubus incluye una parte soportando ejecución controlada por tiempo y otra por eventos. La parte controlada por tiempo puede ser utilizada para aplicaciones de tiempo real duro. Rubus incluye un modelo de componentes y herramientas asociadas que son utilizadas en proyectos con Volvo Construction Equipment Components AB.

Un componente incluye comportamiento, estado persistente, un conjunto de puertos de entrada y salida y una función de entrada que proporciona un método de inicialización.

Una *tarea* proporciona el flujo de ejecución para los componentes. La función de entrada toma como argumento los puertos de entrada, el estado persistente y referencias a los puertos de salida. El sistema automáticamente genera las comunicaciones entre los componentes conectados.

Los atributos de una tarea incluyen el identificador, periodo, tiempo de liberación, deadline y peor tiempo de ejecución. También pueden especificarse relaciones de precedencia y exclusión mutua. Las propiedades no funcionales son expresadas en las interfaces de los componentes.

La planificabilidad es derivada automáticamente de las descripciones de los componentes, utilizando los atributos de las tareas y las relaciones de precedencia. Las tareas disparadas por tiempo son planificadas estáticamente, y las disparadas por eventos son planificadas *on-line* con planificación expulsiva basada en prioridades fijas.

PECOS

El proyecto PECOS [Müller et al., 2001] intenta utilizar software basado en componentes en sistemas empotrados tales como teléfonos o PDAs.

El modelo de componentes incluye interfaces definidas por puertos de entrada y salida, y conectores para conectar los puertos compatibles (con los mismos tipos). Existen componentes activos de forma similar a UM-RTCOM: pasivos (comportamiento sin hebras) y componentes de eventos (disparados por eventos). Los atributos de un componente pueden especificar consumo de memoria, WCET, ciclado o prioridad.

El modelo PECOS es traducido a un sistema multitarea, expulsivo y con prioridades para ejecutar los diferentes componentes. Incluye un lenguaje de composición que traslada a C++ y Java.

El modelo no especifica nada sobre la planificación de los componentes, planificador a usar o cómo comprobar la planificabilidad, simplemente asume que hay un planificador.

Capítulo 3. Una plataforma de ejecución distribuida

El modelo de componentes definido permite realizar aplicaciones distribuidas de tiempo real. Sin embargo, para que las aplicaciones desarrolladas puedan ser utilizadas en entornos de tiempo real y el análisis de tiempo real sea válido, el modelo debe sustentarse en una plataforma de ejecución distribuida y predecible. UM-RTCOM es, en sí mismo, independiente de la plataforma de ejecución o del sistema operativo, bastando únicamente con que la plataforma elegida tenga un comportamiento predecible. De entre las diferentes opciones existentes, la elección final ha sido RT-CORBA [Díaz et al., 2004b]. Como ya se comentó, las características de RT-CORBA en particular y de CORBA en general: (orientación a objetos, distribuido, predecible, etc.) hacen particularmente interesante esta opción frente a otras.

La solución adoptada pasa pues, por desarrollar aplicaciones utilizando el modelo propuesto: activos, pasivos, eventos, ranuras de configuración, etc. Posteriormente, para la utilización de estos elementos en una plataforma concreta, las herramientas del entorno se encargarán de transformar el código y adecuarlo a la plataforma de ejecución elegida, que en este caso ha sido RT-CORBA, si bien podría haber sido cualquier otra que cumpliera con los requisitos de predecibilidad.

Además de realizar la traducción de los elementos del modelo, será necesario también el desarrollo de cierta infraestructura de ejecución. El desarrollador de las aplicaciones no tiene constancia de esta infraestructura así como tampoco de la traducción, encargándose únicamente del desarrollo de sus aplicaciones; siendo las herramientas de desarrollo las encargadas de la implantación en una plataforma de ejecución concreta.

Como un factor adicional a tener en cuenta está el hecho de que la traducción debe respetar los requisitos de la extensión de tiempo real de SDL y, en concreto, el

modelo abstracto SDL del componente. Esto es esencial para que el resultado del análisis de tiempo real sea válido en la plataforma de ejecución.

Se han considerado dos implementaciones gratuitas de RT-CORBA para C++: TAO [Levine et al., 1998] es una implementación muy popular de CORBA para numerosas plataformas contando con versiones gratuitas y soportadas comercialmente. Incluye la mayor parte del estándar CORBA y RT-CORBA e incluso extensiones propias, y está centrada en el desarrollo de aplicaciones de tiempo real. Posteriormente, se consideró también la utilización de ROFES [Lankes et al., 2003], una implementación de RT-CORBA mínima orientada a sistemas empotrados con un consumo mínimo de recursos.

El resto de este capítulo muestra en primer lugar la traducción de los elementos del modelo para su ejecución con RT-CORBA, mostrándose posteriormente detalles de la infraestructura de ejecución en plataformas concretas. El capítulo finaliza mostrando otras plataformas de ejecución distribuidas de tiempo real que también podrían haber sido utilizadas para la implementación del modelo de componentes.

3.1. Modelo de ejecución de UM-RTCOM sobre RT-CORBA

Las aplicaciones desarrolladas con el modelo de componentes UM-RTCOM van a ser ejecutadas utilizando RT-CORBA como plataforma de distribución de los componentes y un sistema operativo de tiempo real como soporte para la implementación de RT-CORBA y para el modelado de algunos de los elementos de UM-RTCOM.

La solución adoptada pasa por traducir los diferentes elementos de UM-RTCOM en elementos de RT-CORBA, de forma que la predecibilidad de las comunicaciones va a ser responsabilidad de la implementación de RT-CORBA. Por otra parte, otros elementos van a ser modelados utilizando directamente el sistema operativo subyacente, recayendo por tanto cierta responsabilidad en el sistema operativo utilizado.

La traducción presentada intenta ser lo más independiente posible de la implementación de RT-CORBA y del sistema operativo utilizado, si bien, existen algunos aspectos (los cuales se comentarán) que deben adecuarse a ambos.

En la fase de análisis, las herramientas deberán también adecuarse a la implementación de RT-CORBA utilizada (ROFES, TAO, ...) así como a la plataforma

de ejecución: sistema operativo, tecnología de red, etc. para analizar el cumplimiento de los requisitos de tiempo real.

3.1.1. Traducción de componentes genéricos

Los elementos principales del modelo son los componentes genéricos. Cada componente genérico es encapsulado en una librería dinámica donde se activan una serie de objetos CORBA que permiten utilizar al componente.

Esta librería dinámica ofrece una serie de métodos en una clase C++ que permite utilizar toda la funcionalidad del componente, tanto de manera local como remota. Es importante resaltar que esta clase no es "visible" para el usuario, en el sentido de que el usuario sólo maneja los componentes del modelo UM-RTCOM. Esta clase, así como las librerías son únicamente facilidades de implementación para implantar el sistema sobre RT-CORBA.

Todos los elementos contenidos en el componente genérico: activos, pasivos, etc. son encapsulados también en esta librería dinámica.

En el siguiente ejemplo (que se utilizará también posteriormente con otros elementos) puede verse la definición de un componente denominado `micomponente`. Dicho componente es traducido a una librería con el mismo nombre y ofreciendo la misma funcionalidad: interfaces `I1` e `I2`, eventos, etc.

Ej:

```
Component micomponente {
  interface I1 {
      void op1(in short arg1,out short arg2);
      void op2();
      }

  ...

  interface O1 {
      void op3(in float a);
      void op4();
      }
  ...

  input I1;
  input I2;

  output O1 with o1;
  output O2 with o2;
}
```

Parte de la clase resultante de la traducción es mostrada a continuación, donde se incluyen métodos para la creación de instancias, obtención de interfaces o composición, todos los cuales se detallarán en secciones posteriores:

Ej:

```
class micomponente {
      void create_instance(char *lpszname);
      void create_remote_instance(char *lpszname,
                                  char *lpszhost,DWORD port);
...
      I1 *get_I1();
      I2 *get_I2();

      void connect(char *lpszref_name,O1 *pconnected);
      void connect(char *lpszref_name,O2 *pconnected);

...
}
```

La Figura 3.1 muestra el proceso de encapsulado de los componentes en el interior de las librerías dinámicas junto con los objetos CORBA expuestos para su utilización por otras librerías dinámicas.

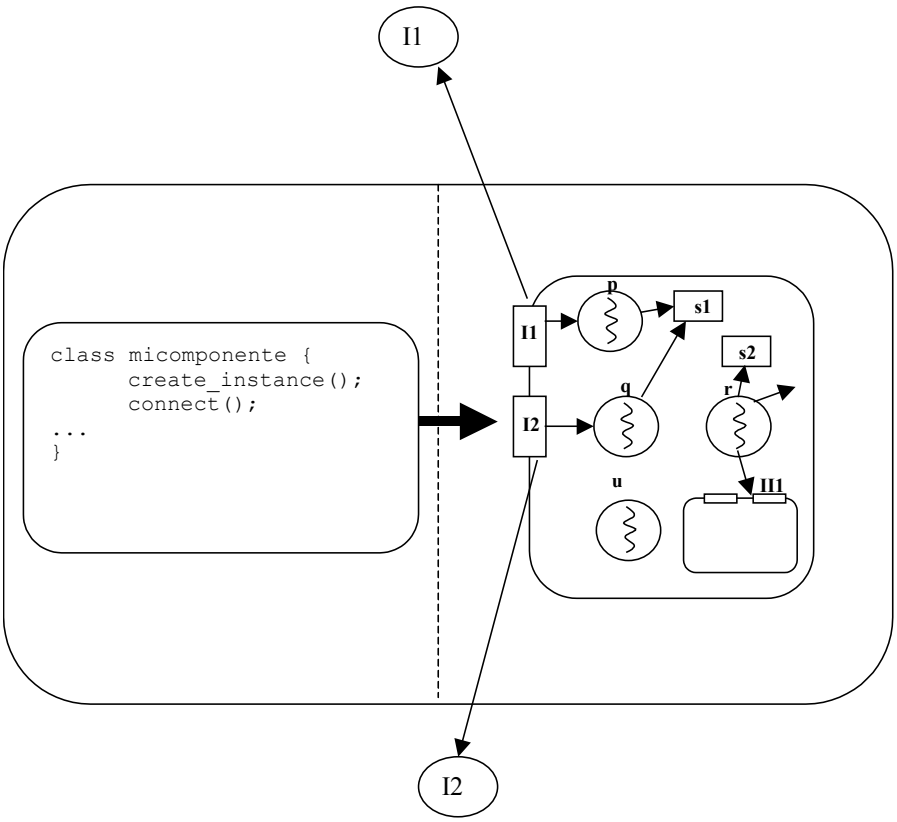
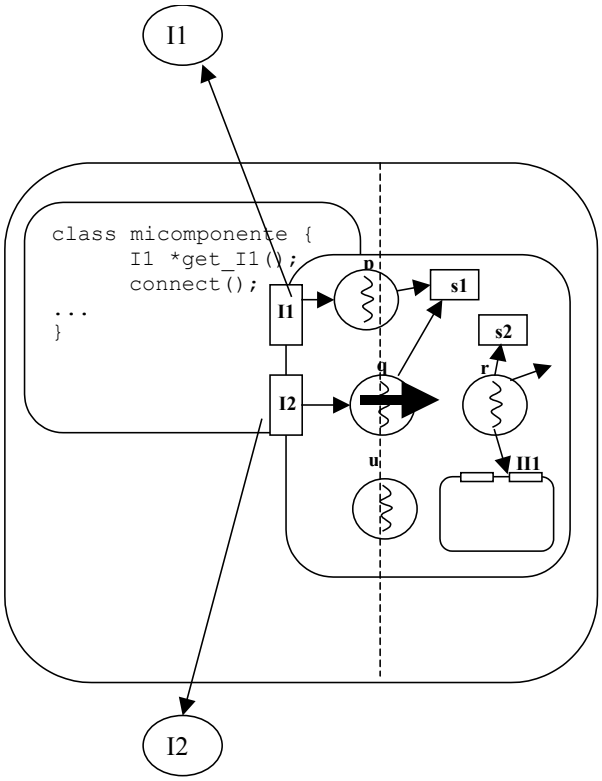


Figura 3.1. Librería dinámicas para componentes genéricos

Interfaces

Las interfaces ofrecen la funcionalidad del componente. Para cada una de las interfaces ofertadas, existe un método en la clase C++ generada del componente que permite obtener una instancia de una clase con el nombre de la interfaz. Dicha clase ofrece los mismos métodos que la interfaz del componente, respetando; no obstante, la traducción C++ de CORBA.

Así por ejemplo, para el caso anterior con interfaz ofertada `I1` se tendría en la clase C++ `micomponente` un método `get_I1`, que para cualquier instancia de un componente tipo `micomponente` devuelve un puntero a una instancia de clase `I1`, la cual ofrece la misma interfaz que la interfaz original. Además, internamente el puntero a dicha instancia de `I1` está asociado a la instancia de `micomponente` de forma que cuando se realiza una invocación a cualquier método de `I1` realmente se está invocando al componente de tipo `micomponente`.




```
Component micomponente {
  interface I1 {
      void op1(in short arg1,
               out short arg2);
      void op2();
   }
   …
}
=>

class micomponente {
…
      I1 *get_I1();
}

class I1 {
      void op1(CORBA::Short arg1,
               CORBA::Short& arg2);
      void op2();
}
```

Figura 3.2. Interfaces y librerías dinámicas

Cuando se invoca algún método de las interfaces, inicialmente a través de la primitiva *call*, se invoca de forma transparente al servicio, utilizando los objetos CORBA activados en el lugar de ejecución del componente así como los tipos de datos de CORBA.

Eventos

Los eventos permiten la comunicación asíncrona entre componentes. La implementación en RT-CORBA de los eventos del modelo presenta un problema. RT-CORBA no dispone de un mecanismo similar. Podrían utilizarse tanto métodos *oneway* como invocaciones AMI [OMG, 2001] de CORBA, pero ninguna de estas posibilidades ofrece lo que los eventos del modelo requieren: métodos asíncronos e inexistencia de interfaces para la interacción entre los componentes.

La solución pasa por utilizar el servicio de eventos de las implementaciones de RT-CORBA utilizadas, por lo que la calidad de la implementación de los eventos en el modelo depende la calidad de la implementación de RT-CORBA sobre la plataforma de ejecución elegida.

La traducción de los eventos consiste en utilizar eventos CORBA para cada uno de los tipos de eventos utilizados. Se van a producir eventos CORBA cuyo tipo de evento contiene una estructura CORBA con diferentes campos, que son los argumentos del tipo de evento.

En el siguiente ejemplo se dispone del tipo de evento `event_type1` que resulta transformado en una estructura CORBA equivalente con dos campos `a` y `b`, que eran los argumentos del tipo de evento del modelo.

Esta estructura va a ser utilizada para producir y consumir eventos utilizando el servicio de eventos de CORBA (en TAO se va a utilizar el servicio de eventos de tiempo real [Harrison et al., 1997], el cual no pertenece al estándar de CORBA,).

Ej:

```
event event_type1(in short a,in short b);
...

=>

struct event_type1 {
      short a;
      short b;
};
```

Tanto los productores de los eventos como los consumidores, se adaptan a esta estructura CORBA, generándose código auxiliar para poder trabajar con los eventos tal y como se realiza en el modelo, por ejemplo, con las primitivas de sincronización, y siendo todo este proceso transparente para el usuario.

3.1.2. Implementación de componentes

La parte de implementación de un componente incluye:

- Implementación de servicios
- Tratamiento de eventos
- Métodos privados y datos miembro
- Componentes utilizados (genéricos, activos y pasivos)
- Composición de interfaces

Implementación de servicios

Los servicios son implementados en el modelo a través de los componentes *Active*. Existe una capa intermedia representada por los objetos CORBA que se encargan de trasladar la petición al componente activo y posteriormente la respuesta al cliente.

Siguiendo con el ejemplo anterior, para la operación `op1` existe un método en el sirviente del objeto CORBA que se encarga de recibir peticiones. Cuando llega una de estas peticiones, es trasladada al componente activo que está esperando por la recepción de estas peticiones.

Ej:

```
class I1_i:public POA_I1 {
  void op1(CORBA::Short arg1,CORBA::Short &arg2);
...
}
void I1_I::op1(CORBA::Short arg1,CORBA::Short &arg2) {
... // código que se explicará con la primitiva wait
}
```

En resumen, por cada servicio ofertado en las interfaces, existe un método equivalente en los objetos CORBA. La petición del cliente es recibida en el objeto CORBA y posteriormente trasladada al componente activo correspondiente esperando también por la respuesta por parte del componente activo.

Tratamiento de eventos

El tratamiento de los eventos consumidos por un componente se va a realizar a través de clases auxiliares encargadas de consumir los diferentes tipos de eventos. Por cada tipo

de evento consumido, se genera una clase encargada de consumir ese tipo de eventos y de dejar los eventos recibidos disponibles para otros elementos del modelo, como por ejemplo los componentes activos esperando en la primitiva de sincronización *wait*.

Ej:

```
class consumer_event_type1:public POA_RtecEventComm::PushConsumer {
      consumer_event_type1();
      void connect(RtecEventChannelAdmin::EventChannel_ptr ev);

      // método de recepción de eventos
      void push(const RtecEventComm::EventSet& data,
                throw (CORBA::SystemException);
      ...
};

      void consumer_event_type1::push(
                 const RtecEventComm::EventSet& data
                throw (CORBA::SystemException) {
      ... // código que se explicará con la primitiva wait
      }
```

Las clases consumidoras generadas se encuentran relacionadas con las clases utilizadas para los objetos CORBA y también con los componentes activos; todo ello para la utilización de las primitivas de sincronización.

Métodos privados y datos miembro

Los métodos privados y datos miembro son trasladados directamente a C++ como métodos y datos de la nueva clase generada.

Ej:

```
      Component micomponente {
            float m_data;

            int metodo1(float temp) {
                  m_data = temp;
            }
      }
```

=>

```
      class micomponente {
            float m_data;
            int metodo1(float temp);
      };

      int micomponente::metodo1(float temp) {
            data = temp;
      }
```

Utilización de componentes

La utilización de componentes en el código de implementación se basa en invocar métodos de dichos componentes. Para los componentes primitivos las llamadas a los métodos son llamadas simples C++. Para el caso de otros componentes genéricos, se realiza la sustitución de la primitiva *call* utilizada, por la utilización de alguna clase de interfaz proporcionada por la librería C++ generada del componente utilizado.

En el caso de los componentes genéricos hay que distinguir dos casos: la declaración de un componente genérico y su utilización directa, o bien la utilización de una interfaz de salida.

En el caso de declarar una instancia de un componente y utilizarlo, las transformaciones necesarias serían la obtención de un puntero a la interfaz de entrada y la sustitución del call por una llamada al método de la interfaz.

Ej:

```
micomponente c;
call c.i1.op1(1,ovar);
=>
micomponente c;
I1 *pI1;

pI1 = c.getI1();

pI1->op1(1,ovar);
```

El caso de las interfaces de salida se dispone ya de un puntero a la interfaz necesitada (el cual ha sido conectado a través de mecanismos que se explicarán en otras secciones), únicamente hay que cambiar el call por la invocación del método.

Ej:

```
output O1 with o1;      // En la parte de definición

...

call o1.op1(1,ovar);    // Utilización del componente
=>
O1 *po1;    // Será conectado adecuadamente por la parte de
            // conexión

...

o1->op1(1,ovar);  // Sustitución del call
                  // de manera automática
```

3.1.3. Traducción de componentes activos

Los componentes activos son transformados en clases C++ junto con hebras del sistema operativo utilizado. La clase C++ contiene los datos miembros y métodos privados del activo, y la hebra encapsulada permite acceder a estos métodos y datos. En este apartado existen pues, detalles dependientes de la plataforma, tales como la creación de la hebra o el establecimiento de su prioridad.

Un aspecto muy importante para el modelo es el de las prioridades de los componentes activos, puesto que constituyen la base, junto con los pasivos de la metodología de análisis de tiempo real que se va a proponer. La solución adoptada pasa por utilizar los mecanismos de gestión de prioridades que ofrece RT-CORBA. En concreto, las herramientas del entorno asignan prioridades a los componentes activos, y se utiliza el modelo de prioridades propagado por el cliente para las invocaciones a servicios de otros componentes.

Las instancias declaradas de los componentes activos son activadas automáticamente junto con el componente genérico en el que están incluidos, incluyendo código para la invocación de posibles métodos de inicialización así como la ejecución automática del método *execute*. Cada una de las instancias del componente activo da lugar a una instancia de esta nueva clase C++.

En el siguiente ejemplo se muestra un componente activo y cómo se realiza la traducción a una clase con hebra encapsulada. Se utiliza sintaxis típica de Windows, si bien podría utilizarse cualquier otra API similar como por ejemplo la ofrecida por POSIX [Aldea y González, 2000].

Ej:

```
  Active Tipo_activo {
      float m_miembro1; // declaración de miembros privados
      …

      void subrutine1(float data) { // subrutinas adicionales
      …
      }
      …
      void execute() {
      …
      }
}

=>
```

```
  class Tipo_activo {
      float m_miembro1; // declaración de miembros privados
      …
      void subrutine1(float data) { // subrutinas adicionales
      …
      }
      …
      void execute() {
      …
      }
      HANDLE m_thread;  // manejador asociado a la hebra
  }
...

      Tipo_activo t;    // Instancia de componente activo
      t.m_thread = CreateThread(...);     // Creación de hebra
      SetThreadPriority(t.m_thread);      // Establecimiento
                                          // de prioridad
```

La implementación de la activación periódica de los componentes activos se basa en la posibilidad de poder usar tareas o hebras periódicas en el sistema operativo destino del componente. Una posible solución para no tener que depender demasiado de dichos sistemas operativos, puede ser optar por APIs que ocultan las diferencias entre sistemas operativos tal y como ofrece ACE [Schmidt, 1994] (que se comentará en el apartado de implementación con TAO).

3.1.4. Traducción de componentes pasivos

Los componentes pasivos son transformados en clases C++ con los métodos y datos miembro que el pasivo proporciona y con exclusión mutua en dichos métodos a través de los *mutexes* de RT-CORBA, que dependen en última instancia de los mecanismos de exclusión mutua proporcionados por el sistema operativo de tiempo real utilizado.

Ej:

```
  Passive Tipo_passive {
      float m_dato;     // declaración de miembros privados

      void set_dato(float dato)
      {
            m_dato = dato;
      }

      float get_dato() {
            return m_dato;
      }
  }
=>
```

```
class Tipo_passive {
    float m_dato;     // declaración de miembros privados
    RTCORBA::Mutex_var mutex;

    Tipo_passive() {
          mutex = rtorb->create_mutex();
    }

    ~Tipo_passive() {
          rtorb->destroy_mutex(mutex);
    }

    void set_dato(float dato)
    {
          mutex->lock();
          m_dato = dato;
          mutex->unlock();
    }

    float get_dato() {
          float temp_resul;

          mutex->lock();
          temp_resul = m_dato;
          mutex->unlock();

          return temp_resul;
    }
}
```

La utilización de los componentes pasivos es pues, similar a la utilización de una clase C++, salvo por el detalle de la exclusión mutua proporcionada por los *mutexes* de RT-CORBA.

3.1.5. Traducción de la primitiva *wait* para servicios

La primitiva *wait* es utilizada en los componentes activos para esperar la invocación de servicios o eventos. En esta sección se explica cómo es la traducción de la primitiva *wait* en su utilización con servicios ofertados en interfaces.

La traducción de la primitiva *wait* afecta tanto al código de los componentes activos como a los objetos CORBA implícitamente creados para la invocación de los servicios.

El funcionamiento es el siguiente: cuando se realiza la invocación de un `wait`, el componente activo queda suspendido esperando la activación de un evento del sistema operativo utilizado. Hay que tener también en cuenta la restricción que impide que un servicio se pueda esperar en componentes activos diferentes.

En los objetos CORBA, cuando se recibe alguna petición en un método de un sirviente, se utiliza una variable auxiliar asociada al componente activo a la que se le asigna un valor que indica qué método ha sido invocado. Adicionalmente, los valores de entrada del método son asignados a variables miembro implícitas del componente activo para poder ser utilizados por éste. Tras estas asignaciones, se lanza el evento por el que estaba esperando el componente activo.

Tras despertar el componente activo, en una sentencia tipo *switch* se realiza la ejecución del código asociado al servicio que ha sido invocado. Tras finalizar la ejecución del bloque de código asociado se lanza otro evento que despierta al sirviente CORBA, el cual copia las variables de salida (contenidas en otras variables miembro implícitas del componente activo) y se finaliza la invocación.

En los objetos RT-CORBA podría ocurrir que llegaran varias invocaciones simultáneamente sobre distintos servicios. Para evitar problemas, se utilizan adecuadamente *mutexes* de RT-CORBA de forma que las peticiones son encoladas atendiendo a su prioridad y de esta forma se evitan además problemas por recibir múltiples invocaciones sobre un mismo componente activo. Es decir, el componente activo va a tratar las peticiones de una en una, y mientras tanto, las peticiones son encoladas atendiendo a su prioridad gracias a la utilización de los *mutexes* de RT-CORBA y de los *thread pools* en caso de que estos últimos hayan sido utilizados. Es necesario realizar pues, análisis del código fuente para poder realizar todas estas transformaciones de código.

En el siguiente ejemplo se muestra una interfaz con dos métodos `metodo1` y `metodo2`, los cuales son utilizados con la primitiva *wait* tal y como se muestra posteriormente.

Se va a utilizar la sintaxis de la API de Windows, aunque podría utilizarse la de cualquier otro sistema operativo.

Ej:

```
// Definición de interfaces y métodos
interface I1 {

      void metodo1(in short arg1,
                   in short arg2,
                   in short arg3);

      void metodo2(out short arg1);

}
```

```
// Componente activo utilizando la primitiva wait
Active a1 {
      void execute() {
            wait I1.metodo1(arg1,arg2,arg3) {
                  short var_aux;

                  var_aux=arg1+arg2+arg3;
                  }
                 I1.metodo2(arg1); {
                  arg1 = 5;
            }
      ...
      }
}
```

En primer lugar, para la identificación del servicio solicitado, existe una tabla para cada componente que permite identificar de manera unívoca a los distintos métodos de las distintas interfaces ofertadas. Para el ejemplo, la tabla sería similar a la siguiente:

Ej:

```
Service_Id_Table  micomponente_Service_Id_Table [] =
            {{"metodo1",1}, {"metodo2",2},…}
```

De esta forma, cada método queda identificado por un valor numérico arbitrario dentro del componente. Adicionalmente, existen métodos que permiten localizar de una manera eficiente tanto el método invocado como su valor numérico.

En el componente activo, aparte de la traducción realizada a todo componente de este tipo, se añade una variable que permite identificar el método que ha sido invocado dentro del *wait*, así como variables auxiliares que permiten rellenar los valores de entrada para la sentencia `wait` y almacenar posteriormente las variables de salida. Los manejadores de los eventos (en caso de utilizar este mecanismo) también son almacenados así como un *mutex* para el acceso al componente activo desde los servicios.

Ej:

```
class a1 {
    public:
          int invoked_method;     // Número de método invocado

          // Variables implícitas para almacenar argumentos
          CORBA::Short I1_metodo1_arg1, I1_metodo1_arg2,
                       I1_metodo1_arg3,I1_metodo2_arg1;

          RTCORBA::Mutex_var mutex_active;    // Mutex del activo

          // Manejadores de eventos implícitos
          HANDLE invocation_event;
          HANDLE response_event;
          };
```

En el código del sirviente, se asigna el valor adecuado a la variable `invoked_method`, se copian los valores de los argumentos de entrada y se despierta al componente activo. Hay además que realizar tareas asociadas con la exclusión mutua para el acceso al componente activo, tales como la creación de los *mutexes* para los distintos servicios o su gestión. Se requieren además métodos para poder asociar el único componente activo de cada servicio.

Tras la finalización del bloque de código en el componente activo se lanza un evento que despierta al sirviente, copiando éste los valores de las variables de salida y finalizando la invocación.

Ej:

```
class I1_i:public POA_I1 {

  a1 *m_pactive_metodo1, *m_pactive_metodo2;
  RTCORBA::Mutex_var mutex_metodo1, mutex_metodo2;

  I1_i() {
      mutex_metodo1 = rtorb->create_mutex();
      mutex_metodo2 = rtorb->create_mutex();
  }

  void set_active_metodo1(a1 *m_pactive) {
      m_pactive_metodo1 = m_pactive;
  }

  void set_active_metodo2(a1 *m_pactive) {
      m_pactive_metodo2 = m_pactive;
  }


  void metodo1(CORBA::Short arg1,
               CORBA::Short arg2,
               CORBA::Short arg3);

  void metodo2(CORBA::Short& arg1);
  ...
}

  void I1_i::metodo1(CORBA::Short arg1,
               CORBA::Short arg2,
               CORBA::Short arg3) {

      mutex_metodo1->lock();
      m_pactive_metodo1->mutex_active->lock();

      // Identificar al método invocado
      m_pactive_metodo1->invoked_method = 1;

      // Copiar argumentos de entrada
      m_pactive_metodo1->I1_metodo1_arg1=arg1;
      m_pactive_metodo1->I1_metodo1_arg2=arg2;
      m_pactive_metodo1->I1_metodo1_arg3=arg3;
```

```
        // Despertar al activo
        PushEvent(m_pactive_metodo1->invocation_event);

        //Esperar terminacion
        WaitForSingleObject(m_pactive_metodo1->response_event);

        // Copiar valores de posibles variables de salida

        // Terminar la invocación
        m_pactive_metodo1->mutex_active->unlock();
        mutex_metodo1->unlock();
    }



    void I1_i::metodo2(CORBA::Short& arg1) {

        mutex_metodo2->lock();
        m_pactive_metodo2->mutex_active->lock();

        // Identificar al método invocado
        m_pactive_metodo1->invoked_method = 2;

        // Copiar argumentos de entrada
        // Despertar al activo
        PushEvent(m_pactive_metodo2->invocation_event);

        //Esperar terminacion
        WaitForSingleObject(m_pactive_metodo2->response_event);

        // Copiar valores de posibles variables de salida
        arg1 = m_pactive_metodo2->I1_metodo2_arg1;

        // Terminar la invocación
        m_pactive_metodo2->mutex_active->unlock();
        mutex_metodo2->unlock();
    }
```

El código del método *execute* quedaría transformado como se muestra a continuación:

Ej:

```
    void execute() {
        WaitForSingleObject(invocation_event);

        switch (invoked_method) {
        case 1:     // metodo1
                    short var_aux;
                    var_aux=  I1_metodo1_arg1+  I1_metodo1_arg2  +
                    I1_metodo1_arg3;
                    break;
        case 2:
                    I1_metodo2_arg1 = 5;
                    break;
        }

        PulseEvent(response_event);
    }
```

En primer lugar, se espera el evento de invocación de algún método. Posteriormente, según el método invocado se ejecuta el bloque de código asociado y finalmente se lanza un nuevo evento para despertar al sirviente.

En el caso de que se haya especificado un tiempo opcional en milisegundos en la primitiva *wait*, debe disponerse en el sistema operativo de algún mecanismo que permita esperar por dicho tiempo opcional y en caso de que no se produzca ninguna invocación esto puede ser detectado en la sentencia *switch* ejecutando el código asociado a dicha condición.

3.1.6. Traducción de la primitiva *wait* para eventos

La traducción de la primitiva *wait* para su utilización con eventos afecta al código de los componentes activos y a las clases consumidoras de eventos generadas automáticamente.

El funcionamiento es el siguiente: cuando se realiza la invocación de un `wait` por un evento, el componente activo queda suspendido esperando la activación de un evento del sistema operativo utilizado tal y como ocurría en el caso de los servicios. En este caso, al contrario que con los servicios, un mismo evento puede estar siendo esperado en más de un componente activo diferente.

En las clases consumidoras de eventos, cuando se recibe un evento hay que realizar dos tareas: despertar a todos los activos que están esperando por ese evento, y copiar los argumentos de entrada en esos activos. Existen diferentes alternativas para poder realizar esta tarea. La solución adoptada ha consistido en disponer de una lista priorizada de los componentes activos que están esperando por ese tipo de eventos. Tras la llegada de un evento, se copian los argumentos en los distintos activos y son despertados lanzando un evento del sistema operativo.

En el componente activo, el procedimiento a seguir es similar al tratamiento de los servicios, utilizándose la misma variable auxiliar del componente activo para saber qué evento ha sido lanzado y con la utilización de las variables intrínsecas donde se han copiado los argumentos de entrada del evento.

Tras despertar el componente activo, en una sentencia tipo *switch* se realiza la ejecución del código asociado al evento que ha sido producido. En este caso, al ser el

evento asíncrono, no hay que realizar ninguna acción tras finalizar la ejecución del bloque de código asociado.

En el siguiente ejemplo se muestra un tipo de evento, el cual es utilizado con la primitiva *wait* tal y como se muestra posteriormente.

Ej:

```
// Definición del tipo de evento
      event evento_num(in short num);

// Componente activo utilizando la primitiva wait
      Active a1 {
            void execute() {
                  wait evento_num(arg1) {
                        short var_aux;

                        var_aux=arg1+1;
                  }
            ...
            }
      }
```

Al igual que ocurría con los servicios, para la identificación del evento producido, se utiliza la misma tabla que permite identificar a servicios y, en esta ocasión, eventos. Para el ejemplo, la tabla podría ser similar a la siguiente:

Ej:

```
   Service_Id_Table  micomponente_Service_Id_Table [] =
                  {{"metodo1",1}, {"metodo2",2}, {"evento_num",3}, …}
```

En el componente activo, se utiliza la misma variable que permite identificar el método o evento que ha sido invocado/producido dentro del `wait`, así como también existen variables auxiliares que permiten rellenar los valores de entrada.

Ej:

```
  class a1 {
      public:
            int invoked_method_event; // Número de método/evento

            // Variables implícitas para almacenar argumentos
            CORBA::Short e1_arg1;

            // Manejadores de eventos implícitos
            HANDLE invocation_event;
      ...
      };
```

En el código de la clase consumidora, se recorre la lista con los activos suscritos al evento, y para cada uno de ellos se asigna el valor adecuado a la variable `invoked_method_event` y se copian los valores de los argumentos de entrada, despertando finalmente al componente activo.

Ej:

```
class consumer_evento_num:public POA_RtecEventComm::PushConsumer {
      ...

      Active_list m_list;     // Lista de componentes activos

      // sumar componente activo para recepción del evento
      void add_active(void *pactive,char *lpszactive_type);

      // método de recepción de eventos
      void push(const RtecEventComm::EventSet& data,
                throw (CORBA::SystemException);
      ...
};

void consumer_evento_num::push(
      const RtecEventComm::EventSet& data
      throw (CORBA::SystemException) {

      // Extraer información del evento
      CORBA::Short num;
      evento_num *pevento;

      ...
      if ((e.data.any_value >>= pevento) != 0) {
            num = pevento->num;
      }

      // Recorrido de la lista de activos
      Active_element *pelement;

      for (…) {
            pelement = m_list->GetCurrent();

            // Convertir al tipo de activo adecuado
            if (active_type(pelement,"a1")) {
                  a1 *pa1 = convert_to_a1(pelement);
                  ...
                  // Identificar al método invocado
                  pa1->invoked_method = 3;

                  // Copiar argumentos de entrada
                  pa1->e1_arg1 = arg1;

                  // Despertar al activo
                  PushEvent(pa1->invocation_event);
                  }
      }
...
}
```

En el código previo puede verse la necesidad de clases o funciones auxiliares tales como por ejemplo, `Active_list` o `Active_element`, para almacenar la lista de componentes activos.

La inserción de los componentes activos en las correspondientes listas es realizada de forma automática por las herramientas del entorno. También es necesario el acceso al componente activo en exclusión mutua.

El código del método *execute* del componente activo quedaría transformado como se muestra a continuación:

Ej:

```
void execute() {

      // Espera la generación del evento
      WaitForSingleObject(invocation_event);

      // Funcionamiento según el evento
      switch (invoked_method) {

      case 3:
            // evento_num
            short var_aux;

            var_aux= e1_arg1+1;

            break;
}
```

El funcionamiento es similar al de los servicios. En primer lugar, se espera el evento de invocación y posteriormente se ejecuta el bloque de código asociado.

3.1.7. Traducción de la primitiva *call*

La primitiva `call` es utilizada para invocar servicios ofrecidos en alguna interfaz de un componente genérico. El componente genérico utilizado puede estar ejecutándose de forma local o remota y la invocación puede ser síncrona o asíncrona dependiendo de cómo se definiera la interfaz del componente, siendo esto transparente para su utilización.

La traducción de la primitiva *call* consiste en sustituirla en los lugares del código donde se utilice, por la invocación al método equivalente de las clases automáticamente generadas.

Ej:

```
Component micomponente {
  interface I1 {
      void op1(in short arg1,out short arg2);
      void op2();
      }

micomponente c;
call c.op1(1,ovar);
```

=>

```
micomponente c;
I1 *pI1;

pI1 = c.getI1();
pI1->op1(1,ovar);
```

En el interior de la clase generada `I1` existen los mecanismos necesarios para poder invocar al objeto CORBA asociado a la interfaz. La clase `I1` almacena una referencia al objeto CORBA asociado, y cuenta con métodos de conexión automáticamente invocados para realizar la conexión con dicho objeto CORBA. En el método asociado a cada servicio, realmente se realiza una invocación al objeto CORBA asociado, el cual podría ser local o remoto.

Ej:

```
class I1 {

      I1_var i1_server;

      void connect(char *lpszname,char *lpszhost,DWORD port);

public:
      void op1(CORBA::Short arg1,CORBA::Short& arg2);

...

}


void I1::connect(CORBA::Short arg1,CORBA::Short& arg2) {

// Procedimiento de conexión con el objeto CORBA
// utilizando el nombre del objeto y su localización
      ...

      i1_server = CORBA::_narrow(i1_object);
}

void I1::op1(CORBA::Short arg1,CORBA::Short& arg2) {

      i1_server->op1(arg1,arg2);
}
```

En el caso de la utilización de un tiempo máximo de ejecución se utilizan los mecanismos de excepción de C++ y de *timeout* de RT-CORBA provocándose una excepción en caso de transcurrir el tiempo máximo y no haberse completado la invocación. En el caso de situaciones erróneas puede utilizarse el miembro predefinido **`completed`** como si de una variable lógica se tratara, indicando si la llamada síncrona se completó satisfactoriamente.

3.1.8. Traducción de la primitiva *raise*

La primitiva *raise* es utilizada para generar eventos de forma asíncrona, siendo capturados estos eventos por todos los componentes que así lo indican en su definición. Los eventos son distribuidos, pudiendo producirse en una máquina y siendo consumidos en otras. La traducción de la primitiva *raise*, se basa en la producción de eventos CORBA, teniendo en cuenta su utilización en la primitiva *wait*, y consiste en la sustitución en los lugares del código donde se utilice por un método auxiliar que se encarga de la producción del evento.

Ej:

```
      event evento_num(in short n);
      raise evento_num(3);
=>
      micomponente c;
      c.raise_evento_num(3);
```

En el interior de la clase generada `micomponente` existen los mecanismos necesarios para poder generar el evento CORBA tales como la conexión con el servicio de eventos, etc. Entre ellos, el método `raise_evento_num` es el encargado de la producción del evento tal y como se muestra a continuación:

Ej:

```
      void raise_evento_num(CORBA::Short num) {
            evento_num ev_data;

            ev_data.num = num;

            RtecEventComm::EventSet event(1);   // Tipos de datos CORBA
            event.data.any_value <<= ev_data;

            // Introducción del evento en el servicio de eventos
            consumer_proxy_->push(event);
      }
```

Como puede verse en el código previo, la generación del evento es altamente dependiente de la implementación de RT-CORBA utilizada, puesto que depende de clases, métodos, etc. que pueden variar en diferentes implementaciones de RT-CORBA.

En el caso de los eventos no es posible conocer si el evento ha sido consumido satisfactoriamente en los componentes consumidores.

3.1.9. Instanciación de componentes

Las instancias de componentes son traducidas a las instancias de las clases C++ generadas.

Para la creación de los componentes, las clases C++ generadas ofertan métodos para dicha creación tales como `create_instance` o `create_remote_instance`, pudiendo crearse los componentes de forma local o remota según el despliegue que se haya realizado de la aplicación.

Ej:

```
Component micomponente {
  ...
}

micomponente c;

=>

class micomponente {
      void create_instance(char *lpszname);
      void create_remote_instance(char *lpszname,
                                  char *lpszhost,DWORD port);
...
};

c.create_remote_instance("instance_name","host",port);
```

3.1.10. Configuración de componentes

La traducción de las ranuras de configuración se basa en la utilización del método *configure*. Para cada componente del modelo existe un método con dicho nombre que puede ser utilizado para proporcionar los valores de las ranuras de configuración. Los valores de las ranuras de configuración son entonces utilizadas como constantes en el código del componente.

Ej:

```
Component Controlador {
      config {
            int periodo_sensor;
      }
...
}

Controlador micomponente;
micomponente.configure(30);
```

=>

```
class Controlador {
      int periodo_sensor;


      void configure(int periodo_sensor);
};

void Controlador::configure(int periodo_sensor) {
      this->periodo_sensor = periodo_sensor;
}
```

3.1.11. Interconexión de componentes

La interconexión de los componentes contempla la conexión de las interfaces de salida de un componente con las interfaces de entrada de otro componente.

En el modelo presentado las interfaces de salida pueden tener múltiples conexiones o referencias, (ej: `O1 with o1, o2, o3`). Cada una de estas referencias estará conectada a un componente y, posteriormente, en el código del componente se utilizan estos nombres para realizar las invocaciones (ej: `call o1.metodo`).

La composición debe contemplar pues estas diferentes conexiones y el cómo conectarlas a otros componentes.

La solución propuesta en RT-CORBA pasa por utilizar las clases generadas para los componentes con los métodos adecuados, junto con el servicio de nombres de RT-CORBA; todo ello en fase de inicialización para no afectar al comportamiento de tiempo real de la aplicación.

Las clases generadas incluyen para cada componente métodos *connect* que permiten conectar las interfaces de salida del componente con otros componentes, para ello basta con indicar el nombre de la interfaz de salida (en el ejemplo podrían ser `o1`, `o2` u `o3`) y suministrar un puntero a la interfaz deseada del componente con el que se va a conectar. De esta forma y de manera interna, la interfaz de salida queda conectada al

componente indicado. Todo este proceso es automáticamente realizado por las herramientas del entorno.

Ej:

```
Component A {
  interface I1 {
      void op1(<parameters>);
      void op2();
      }
  ...
  input I1;
}

Component B {
  interface O1 {
      void op1(<parameters>);
      void op2();
      }
  ...
  output O1 with o1;
}


A instance_a;
B instance_b;

instance_b.o1 <-> instance_a.I1;
```

=>

```
class B {
      O1 *m_po1;  // Puntero a ref. de interfaz de salida o1
      void connect(char *lpszref_name,O1 *connected);
};


void B::connect(char *lpszref_name,O1 *pconnected) {
      ...
      m_po1 = pconnected;     // Conexión establecida
}

A instance_a;
B instance_b;

b.connect("o1",instance_a.get_I1());
```

En la Figura 3.3 puede verse la composición de los dos componentes contenidos en librerías dinámicas. Los objetos CORBA permiten la utilización de componentes de forma local o remota a través de la utilización de las interfaces CORBA.

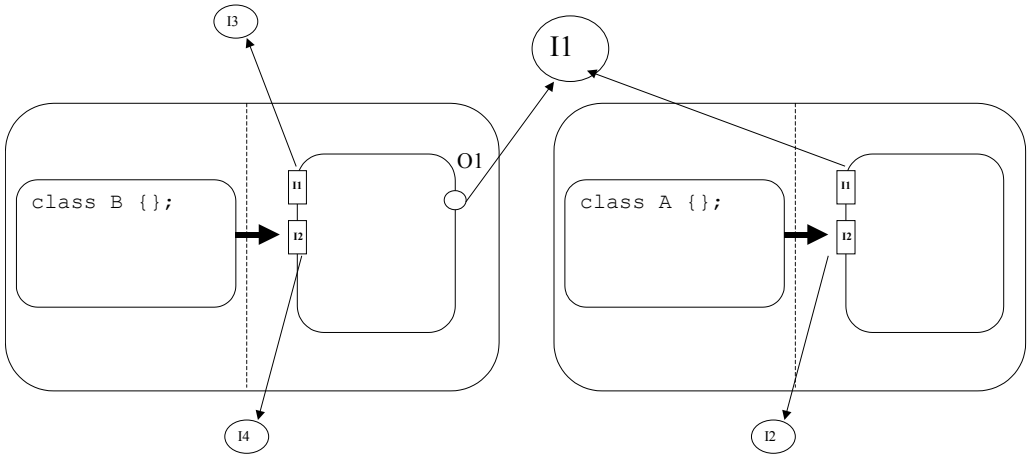


Figura 3.3. Composición de componentes

3.1.12. Componentes como aplicaciones

Los componentes-aplicaciones son transformados en aplicaciones C++ del sistema operativo utilizado. En el módulo principal del programa, se realiza la traducción del método *execute*, donde se declaran, por ejemplo, las instancias de los componentes utilizados y se realiza su inicialización, comenzando así la ejecución de la aplicación.

Una aplicación puede contener componentes ejecutándose en otros nodos, para ello el sistema proporcionará en la implementación concreta, módulos contenedores que puedan lanzar y utilizar cualesquiera componentes.

Las aplicaciones son desplegadas en plataformas concretas de ejecución, teniendo que tomar en cuenta el sistema operativo o la red de comunicaciones. Las herramientas del entorno se encargan aquí de seleccionar las mejores características para el despliegue de la aplicación, tales como por ejemplo, la utilización de conexiones privadas o no multiplexadas de RT-CORBA, configuración de los protocolos de comunicación utilizados, etc.

3.2. Infraestructura de ejecución

Se han diseñado prototipos del modelo y su traducción en dos implementaciones de RT-CORBA. Los prototipos realizados no pretenden ser implementaciones finales del modelo, sino que son utilizados para comprobar la idoneidad del modelo propuesto y su traducción en RT-CORBA. Por un lado, TAO es una implementación de RT-CORBA ampliamente difundida y utilizada en todo tipo de sistemas. TAO tiene como inconveniente su complejidad; es por ello, que se ha utilizado también ROFES, otra

implementación de RT-CORBA especialmente diseñada para sistemas empotrados y que presenta como principal ventaja su simplicidad.

3.2.1. The ACE ORB

TAO (*The ACE ORB*) es una implementación gratuita de CORBA realizada por el grupo Distributed Object Computing (DOC) de la Universidad Washington en San Luis y de California (Irvine) para el desarrollo de aplicaciones distribuidas de gran tamaño.

Los desafíos principales con los que TAO quiere tratar son la portabilidad entre sistemas operativos, gestión de la conexión e inicialización de servicios, demultiplexación de eventos y su manejo, multitarea y sincronización, detección y tolerancia a fallos y cuestiones relativas a calidad de servicio.

TAO es un proyecto de gran envergadura y de alta calidad, siendo ampliamente utilizado en todo tipo de proyectos de tiempo real, por lo que puede ser muy adecuado para la implementación del modelo de componentes.

ACE. Un entorno de comunicaciones adaptativo

TAO está construido sobre ACE (The Adaptive Communication Environment), un marco de trabajo en C++ basado en patrones de diseños con clases para hebras, sincronización, comunicaciones, etc. ofreciendo una API independiente del sistema operativo (similar a POSIX). La arquitectura de ACE puede verse en la Figura 3.4.

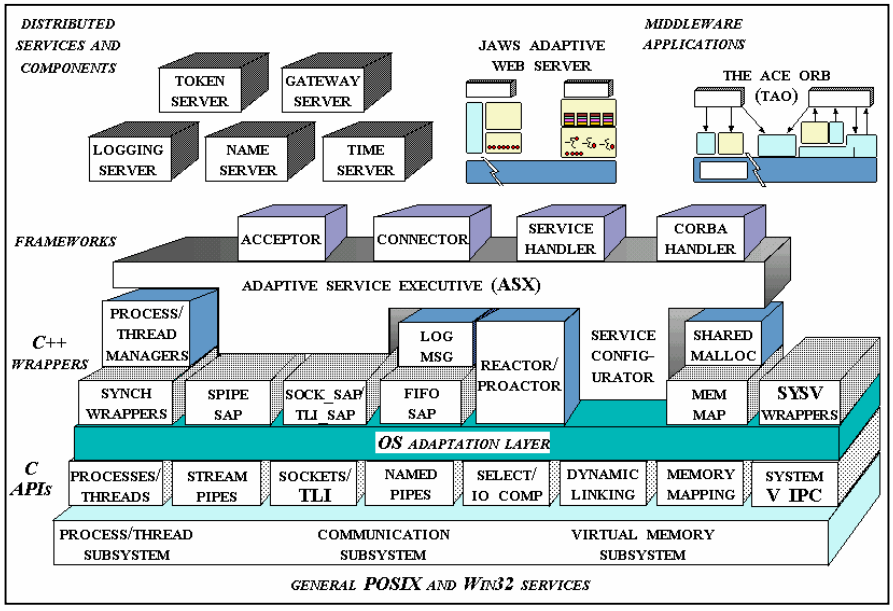


Figura 3.4. Arquitectura de ACE

- El nivel adaptador de sistema operativos de ACE: Ofrece una adaptación de POSIX que permite a otros niveles y componentes de ACE adaptarse a la API del sistema operativo. Este nivel comprende aspectos tales como concurrencia y sincronización, comunicación inter-procesos (IPC) y memoria compartida, sistemas de ficheros, etc.
- El nivel C++ para interfaces de sistemas operativos: ofrece un nivel de adaptación en C++ ofreciendo muchas de las mismas características del nivel anterior
- El nivel del marco de trabajo: ofrece un marco de trabajo para el desarrollo de aplicaciones de redes de alto nivel. Sus componentes incluyen servicios para demultiplexación de eventos, inicialización, configuración, enlace jerárquico de capas y componentes de adaptador de ORB (CORBA).
- Componentes y servicios distribuidos: librería estándar de servicios distribuidos muy comunes en aplicaciones distribuidas, como por ejemplo, un servicio de *login* o de tiempo.

Características de TAO

TAO es un ORB C++ compatible con la mayor parte de las características y servicios definidos en la especificación CORBA 3.X, incluyendo la especificación de RT-CORBA. TAO incluye también una implementación del modelo CCM llamada CIAO.
TAO ha sido implementando en numerosos sistemas operativos incluyendo diferentes versiones de Windows e UNIX y sistemas operativos de tiempo real tales como LynxOS, VxWorks, QnX, Neutrino, OS9 y ChorusOS. La Figura 3.5 muestra los principales componentes de TAO:

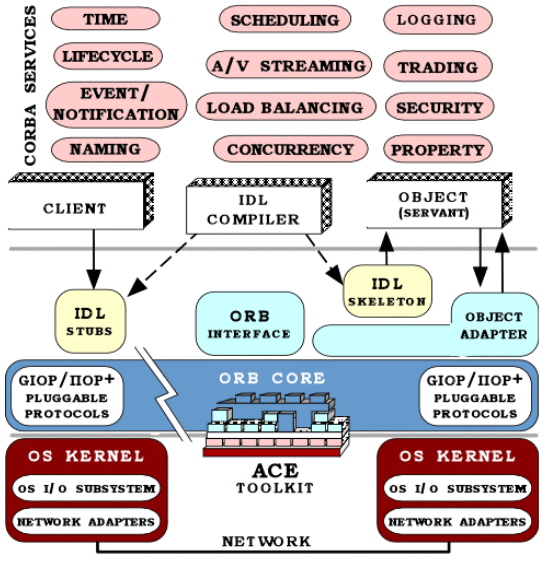


Figura 3.5. Componentes de TAO

- Compilador IDL: basado en la versión mejorada del compilador SunSoft IDL e incluyendo algunas de las últimas características de CORBA IDL tales como la sintaxis de CCM, objetos por valor, especificación de mensajes, etc.
- Protocolo Inter-ORB: TAO implementa los protocolos GIOP/IIOP versiones 1.0, 1.1 y 2.2. con la posibilidad de utilizar otros mecanismos/protocolos tales como memoria compartida, UDP, SSL, etc.
- Núcleo del ORB: el núcleo del ORB proporciona mecanismos síncronos y asíncronos con comportamiento predecible y de alto rendimiento. Proporciona diferentes modelos de concurrencia: reactivo, *thead* por conexión, *thead pool* (incluyendo los *thead pool* de RT-CORBA) y reactor por prioridad.
- Adaptador de objetos portable: el POA de TAO está diseñado con estrategias de demultiplexación optimizadas para identificar a los sirvientes.
- Repositorio de implementación: para lanzar automáticamente servidores en respuesta a solicitudes de clientes.
- Repositorio de interfaces: proporciona información en tiempo de ejecución sobre interfaces IDL.

TAO proporciona muchos de los servicios CORBA estándares incluyendo:

- Servicio de *streaming* para audio y video: implementa la especificación *"Control and Management of Audio/Video Streams Specification"* para aplicaciones que necesiten adaptarse a cambios en la disponibilidad de recursos.
- Servicio de concurrencia: soporta un subconjunto de este servicio, la parte no transaccional.
- Servicio de eventos.
- Servicio de ciclo de vida.
- Servicio de *login*.
- Servicio de nombres: soporta también el servicio de nombres interoperable indicando una manera estándar para clientes y servidores de localizar al servicio de nombres.
- Servicio de notificación.
- Servicio de estado persistente.
- Servicio de propiedades.
- Servicio de seguridad.
- Servicio de tiempo.

- Servicio de *trading*.

Adicionalmente, TAO proporciona los siguientes servicios no estándares:

- Servicio de balance de carga: implementa algoritmos de *round robin* y dispersión mínima para ayudar a balancear la carga en un grupo de máquinas.
- Servicio de eventos de tiempo real: mejora del servicio de eventos proporcionando filtrado, correlación de eventos, despacho de tiempo real y comunicación *multicast*.
- Servicio de planificación [Gill et al., 2001]: soporta planificación de razón monótona, y *primero el más urgente* para asignar prioridades y comprobar la planificabilidad. Está integrado con el servicio de eventos de tiempo real, si bien los implementadores de TAO están planteándose integrarlo con el ORB así como soportar el servicio de planificación definido en la especificación de RT-CORBA.

Arquitectura de TAO

La arquitectura de TAO incluye la gestión de la interfaz de red, el subsistema de I/O, los protocolos de transporte y los componentes CORBA. En contraposición a otros ORBs no diseñados para tiempo real, TAO incluye la gestión de la interfaz de red para obtener un comportamiento predecible. Un esquema con la arquitectura de TAO puede verse en la Figura 3.6:

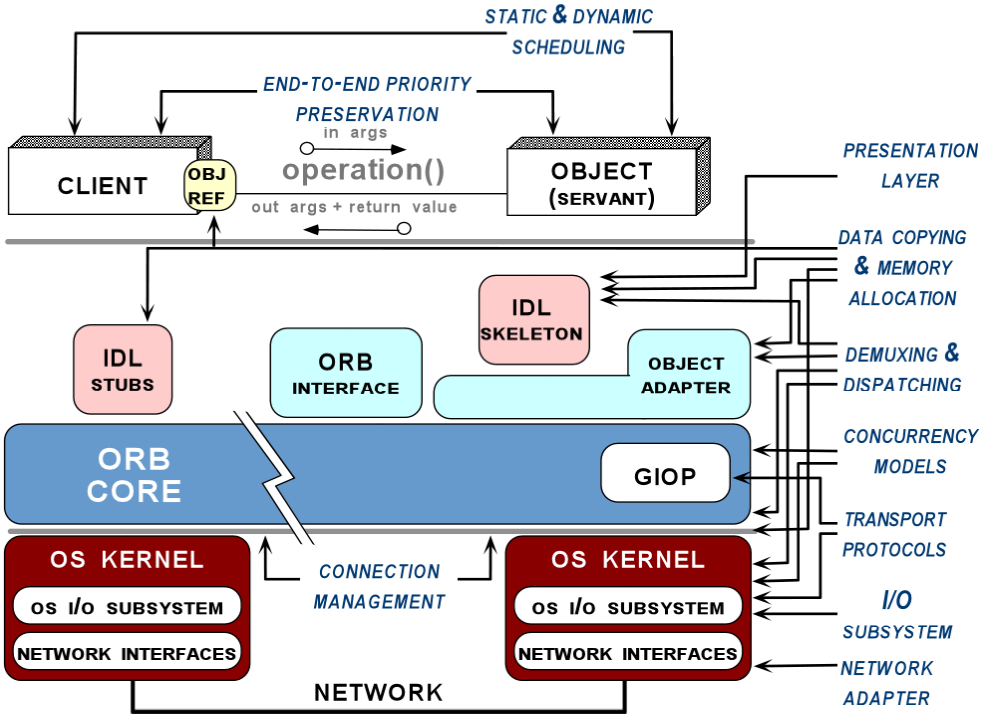


Figura 3.6. Arquitectura de TAO

Subsistema I/O: El subsistema I/O es responsable de la mediación entre el ORB y el acceso de las aplicaciones a la red de bajo nivel y a recursos del sistema operativo. Algunas responsabilidades de este subsistema son las siguientes:

- Forzar la garantía de los requisitos de QoS, minimizar la inversión de prioridad y el no-determinismo.
- Facilitar la especificación por parte de las aplicaciones de los requisitos de QoS.
- Posibilitar la influencia del ORB en la QoS de la red y el sistema operativo.

La consecución de estos objetivos depende, lógicamente, de las posibilidades del sistema operativo utilizado.

Núcleo de ORB de tiempo real: El núcleo del ORB maneja las conexiones de transporte, dirige las peticiones de clientes al adaptador de objetos y devuelve la respuesta si existe. Implementa el modelo de concurrencia de peticiones y es el punto final de demultiplexación de transporte del ORB. El núcleo del ORB de TAO está basado en componentes de ACE tales como *aceptadores* (*acceptors*), *conectores* (*connectors*), *reactores* (*reactors*) y tareas.

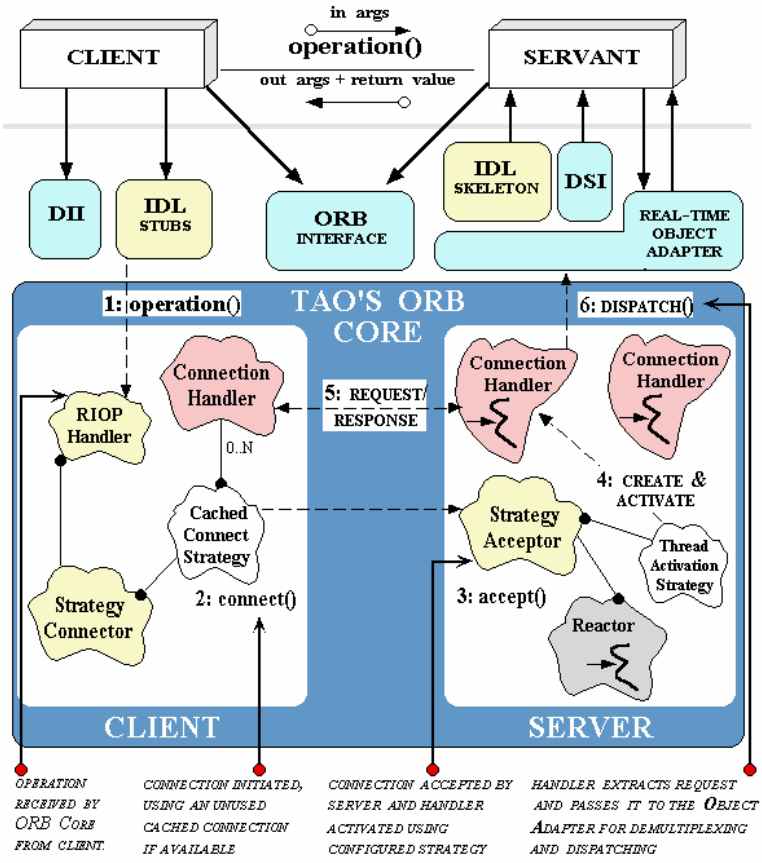


Figura 3.7. Componentes del ORB de TAO

La Figura 3.7 muestra cómo en la parte del cliente se usa un conector para manejar las conexiones con el servidor (mecanismo de cache) ahorrando tiempo de configuración y minimizando las latencias entre la invocación y la ejecución. En la parte del servidor hay un *aceptador* junto con un *reactor* para aceptar conexiones. El *aceptador* delega la activación del manejador de la conexión a alguna estrategia de TAO. El manejador de la conexión extrae la solicitud e interopera con el adaptador de objetos.

Adaptador de Objetos: TAO implementa el adaptador de objetos POA. Las políticas de despacho y demultiplexación son las que hacen que el adaptador de objetos sea eficiente y predecible. En este sentido, el adaptador de objetos de TAO incorpora técnicas de *hashing* que permiten identificar al sirviente y a su operación en un tiempo O(1) en el peor caso.

Especificación de QoS: Las aplicaciones de tiempo real que utilicen TAO pueden utilizar el servicio de planificación de TAO para especificar sus requisitos de planificación. La calidad de servicio es proporcionada a través de las operaciones.

TAO proporciona las estructuras `RT_Operation` y `RT_Info` para proporcionar información de planificación, incluyendo el peor tiempo de ejecución, periodo, importancia y dependencias de datos. Sin embargo, no queda claro cómo se realiza el análisis de tiempo real (se menciona la utilización de ARM), y además su utilización es incómoda para el desarrollador. Por estos motivos, este servicio no ha sido utilizado en esta tesis. Actualmente, los desarrolladores de TAO están planteándose la realización de herramientas para facilitar todas estas tareas.

Marco de presentación: El compilador de TAO IDL incluye optimizaciones en los *stubs* y *skeletons* relativas a la gestión de la memoria:

- Uso reducido de memoria dinámica (impredecible).
- Copia de datos reducida.
- Sobrecarga de llamadas a función reducida.

3.2.2. Infraestructura de ejecución sobre TAO

Los prototipos de implementación del modelo con TAO han sido desarrollados bajo el sistema operativo Solaris 5.8 [Solaris, 2005] y, al estar ampliamente difundido, también

bajo Windows. Si bien este último no puede ser utilizado en aplicaciones de tiempo real duro, sí puede ser utilizado para aplicaciones tiempo real blando, como se comprobará en posteriores capítulos.

TAO es un proyecto en constante evolución, con la consiguiente aparición de nuevas versiones. Si bien, la traducción de RT-CORBA es independiente de la versión de TAO, se ha utilizado la versión 1.3 del mismo con carácter preferente.

TAO se adaptó bien a la implementación de los prototipos, al disponer de numerosas características e implementar la mayor parte de RT-CORBA así como del estándar CORBA.

Características a tener en cuenta de TAO son, por ejemplo, sus múltiples modelos de concurrencia independientemente de los *thread pools* de RT-CORBA. TAO puede ser configurado para tratar las invocaciones secuencialmente, con múltiples hebras, etc. Todas estas características son independientes de la traducción, habiendo sido utilizadas en cada ocasión las consideradas más idóneas.

Por ejemplo, para el caso de la gestión de prioridades de RT-CORBA, TAO ofrece tres diferentes traducciones de las prioridades CORBA a las prioridades nativas: *directa*, *lineal* y *continua*. Así, por ejemplo, en el modo lineal se puede utilizar el rango completo de prioridades de CORBA, transformándose en la prioridad más adecuada en el sistema operativo destino utilizado.

TAO proporciona también la posibilidad de utilizar diferentes políticas de planificación en el sistema operativo: *FIFO* (la hebra de prioridad más alta es ejecutada primero), *round-robin* (porciones fijas de tiempo) y *SCHED_OTHER* (políticas de planificación del sistema operativo). En este caso, la elección fue utilizar la planificación del sistema operativo utilizado.

Para la implementación de los eventos del modelo UM-RTCOM, se ha utilizado el servicio de eventos de tiempo real. Este servicio permite la planificación de los eventos teniendo en cuenta características de calidad de servicio y presentando características adicionales al servicio de eventos estándar de CORBA como la posibilidad de realizar filtrados, recibiendo solamente los eventos en los que se está interesado. Esta característica es idónea para los eventos del modelo UM-RTCOM, y de hecho se tuvo presente prácticamente desde el inicio, la posibilidad de implementarlos utilizando esta característica de TAO.

TAO permite también utilizar el servicio de eventos de tiempo real combinado con el servicio de planificación que él mismo incluye. Este servicio se encarga de

asignar prioridades, elegir políticas, etc. pero su estado de implementación no parecía suficientemente acabado, por lo que se descartó su utilización.

Otra posible ventaja de utilizar TAO es que está construido sobre ACE por lo que es posible aprovechar todas las características que este marco de trabajo ofrece para eliminar diferencias entre sistemas operativos. Así, por ejemplo, en el siguiente fragmento de código se muestra como puede realizarse la traducción de la activación periódica de los componentes activos utilizando ACE.

Ej:

```
class Activo1: public ACE_Event_Handler
{

public:

   // Método a ejecutar
   void execute() {
        cout << ACE_OS::gettimeofday().sec() << endl;
   }


   // Manejador de eventos
   virtual int handle_timeout (const ACE_Time_Value &,
                               const void *arg)
   {
        execute();

        return 0;
   }
};



class Activo2: public ACE_Event_Handler
{
public:

   // Método a ejecutar
   void execute() {
        cout << ACE_OS::gettimeofday().sec() << endl;
   }


   // Manejador de eventos
   virtual int handle_timeout (const ACE_Time_Value &,
        const void *arg)
   {
        execute();

        return 0;
   }
};
...
```

```
ACE_Time_Value activation;
ACE_Time_Value period1,period2;
Thread_Timer_Queue timer;
Activo1 a1;
Activo2 a2;

...
period1 = ACE_Time_Value(5,0);
period2 = ACE_Time_Value(8,0);

timer.schedule(&a1,0,activation,period1);
timer.schedule(&a2,0,activation,period2);
```

En el fragmento de código previo se muestran dos componentes activos con periodos de 5 y 8 segundos. En esta ocasión los componentes activos deben heredar de una clase de ACE denominada `ACE_Event_Handler` y sobrecargar ciertos métodos activados automáticamente con los periodos indicados, teniendo también que utilizar algunos de los mecanismos para temporizadores que ofrece ACE. La ventaja de esta aproximación, es que este código funciona sin modificaciones en los distintos sistemas operativos que soportan ACE+TAO.

3.2.3. ROFES

El proyecto ROFES (Universidad Técnica de Aquisgrán) ofrece una implementación de RT-CORBA especialmente diseñada para sistemas empotrados con pocos recursos de memoria o CPU. ROFES (*Real-Time CORBA for embedded Systems*) se apoya en la especificación de CORBA mínimo de OMG (*Minimum CORBA*) [OMG, 2004]. Esta especificación es una versión recortada de CORBA pensada para su utilización en sistemas empotrados.

ROFES se basa en una arquitectura de *microkernel*. El ORB está separado en varios componentes que pueden ser dinámicamente utilizados. De esta forma, en memoria sólo se encuentran los componentes requeridos obteniendo un consumo muy bajo de memoria.

ROFES puede utilizarse tanto en sistemas Windows XP/CE, Unix, Real-Time Linux o LynxOS. ROFES soporta también diversas arquitecturas de red además de *Ethernet*, incluyendo la utilización de la red CAN de tiempo real u otras arquitecturas como la denominada *Scalable Coherent Interface* (SCI) [Lankes et al., 2001] o el

protocolo *time-triggered ethernet* para Real-Time Linux [Lankes et al., 2002] [Yodaiken y Barabanov, 1997].

El proyecto ROFES es un proyecto modesto en comparación con TAO, por lo que no existe mucha más información sobre su arquitectura interna. Sin embargo, ofrece la ventaja de un código fuente mucho más sencillo de entender que el ofertado por TAO, por lo que puede ser factible para el estudio y desarrollo de técnicas de análisis de tiempo real.

ROFES incluye un compilador de IDL, pero no así ningún servicio CORBA, teniendo que utilizar los ofertados por otros ORB, como por ejemplo el servicio de nombres de TAO.

ROFES soporta actualmente GIOP versiones 1.0, 1.1 y 1.2. con la posibilidad de utilizar instancias de este protocolo bajo TCP/IP (IIOP), CAN (CANIOP) y SCI (SCIOP).

3.2.4. Infraestructura de ejecución sobre ROFES

La utilización del proyecto ROFES se justifica en la posibilidad de desarrollar aplicaciones en sistemas empotrados o mediante comunicaciones inalámbricas utilizando CORBA. ROFES puede ser utilizado perfectamente en dispositivos tipo *Asistentes Digitales Personales*, más comúnmente conocidos como PDAs. La posibilidad de poder utilizar este tipo de dispositivos en aplicaciones distribuidas abre un amplio abanico de posibilidades que pareció interesante estudiar. En concreto, se pensó en la utilización de este tipo de dispositivos para el desarrollo de herramientas de simulación como las comentadas posteriormente.

El principal inconveniente para la utilización de ROFES es su grado de desarrollo. Al contrario que TAO, que lleva ya cierto tiempo siendo utilizado, ROFES es un proyecto en sus primeras fases con un grupo de desarrollo considerablemente más reducido que el de TAO, por lo que es una implementación que puede presentar errores, falta de documentación, etc. Por el contrario, su simplicidad es alta, con lo que puede ser objeto de estudios, etc. Además, ofrece la interesante característica de utilizar la red de comunicaciones CAN, red predecible que puede ser utilizada en sistemas de tiempo real. En cualquier caso, el grado de desarrollo de ROFES es suficiente para el desarrollo de prototipos del modelo UM-RTCOM.

En concreto, se ha utilizado ROFES sobre el sistema operativo Windows CE [Grattan y Brain, 2000] y con dispositivos Pocket PC, como se ha comentado, principalmente para el desarrollo de herramientas de simulación.

En este caso no se dispone de las amplias posibilidades que ofrece TAO como diferentes modelos de concurrencia, planificación, etc. La implementación tuvo que ceñirse a los elementos básicos de RT-CORBA junto con lo ofrecido por el sistema operativo utilizado.

El desarrollo bajo Windows CE se realizó utilizando Microsoft Embedded Visual C++ 4.0 [EVC, 2005], interesante entorno para el desarrollo bajo Windows CE ofreciendo numerosas herramientas que facilitan el desarrollo de aplicaciones, tales como emuladores de los dispositivos PDA, herramientas de monitorización del *kernel* del sistema operativo, etc.

3.3. Otras plataformas de ejecución

En las siguientes secciones se mostrarán distintas plataformas-modelos de ejecución para aplicaciones de tiempo real que también han sido consideradas para la implementación del modelo UM-RTCOM. Las principales motivaciones para utilizar CORBA y RT-CORBA quedaron detalladas en secciones anteriores.

3.3.1. ARMADA

El proyecto ARMADA [Abdelzaher et al., 1997] es un proyecto entre el Real-Time Computing Laboratory (RTCL) y la Universidad de Michigan para el desarrollo de un conjunto de servicios de comunicaciones en forma de middleware que proporciona soporte para tolerancia a fallos y garantías de tiempo real en aplicaciones distribuidas. ARMADA comprende tres áreas principalmente:

1. Las aplicaciones distribuidas de tiempo real dependen fuertemente del sistema de comunicaciones a bajo nivel. Esto motivaría el desarrollo de un soporte de comunicaciones para tiempo real.
2. ARMADA pretende proporcionar tolerancia a fallos y garantías en la transmisión de mensajes. Para ello ayuda al programador de aplicaciones con un conjunto de

servicios de middleware para comunicación de grupos y tolerancia a fallos con restricciones de tiempo.

3. El tercer área en el proyecto ARMADA es el desarrollo de herramientas para la prueba y validación de los servicios desarrollados en las otras dos áreas.

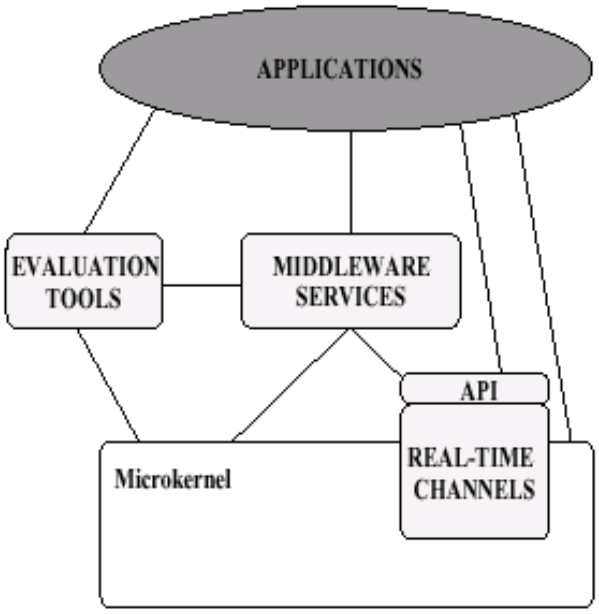


Figura 3.8. Visión general de ARMADA

Plataforma

ARMADA se basa en las capacidades de un *microkernel* de tiempo real introduciendo un conjunto de servicios de comunicación, tolerancia a fallos y herramientas de prueba que proporcionan un marco de trabajo integrado para el desarrollo y ejecución de aplicaciones de tiempo real.

El subsistema de comunicaciones en tiempo real de ARMADA se diseñó sobre IP o UDP, no utilizando TCP por la sobrecarga que introduciría en la transmisión de mensajes entre las aplicaciones.

Para la comunicación entre aplicaciones se utilizaron servicios diseñados como servidores multihebra a nivel de usuario (aunque estos servicios pueden ser posteriormente sumados al *kernel* de forma selectiva eliminando así, por ejemplo, cambios de contexto). Los clientes se comunican con los servidores a través de una librería de usuario.

Arquitectura de comunicación en tiempo real de ARMADA

ARMADA proporciona aplicaciones con una arquitectura de comunicaciones y servicios que permite garantizar calidad de servicio entre dos *hosts* conectados. Para lograr esto, se consideran tres aspectos:

1. El comportamiento de un componente no debe afectar o sobrecargar al resto.
2. Diferenciación de servicios: asignando prioridades a clases de conexiones.
3. Degradación suave en presencia de sobrecarga.

Se desarrolló una librería para mecanismos de gestión de recursos que cumplía los requisitos anteriores con la abstracción de puntos finales de comunicación con flujo garantizado. Esta librería, denominada CLIPS (*Communication Library for Implementing Priority Semantics*) utiliza esta abstracción de punto final (CLIP) para garantizar un flujo en términos del número de paquetes enviados a través de él por periodo, e implementando un buffer configurable. Entre las características más destacables de los *clips* encontramos las siguientes:

- Uno o más sockets pueden estar conectados al mismo *clip*, en cuyo caso el *clip* obtiene recursos suficientes para todos.
- Diferentes *clips* pueden tener diferentes prioridades, para permitir al tráfico de mayor prioridad ser procesado primero. Cada *clip* tiene asociado un *deadline*, que especifica el tiempo máximo de respuesta.

Real-time Communication Service

Se utilizó CLIPS para implementar un servicio de comunicaciones con calidad de servicio asegurada denominado el canal de tiempo real). Un canal de tiempo real es una conexión *unicast* virtual entre una fuente y un *host* destino con garantías de rendimiento sobre el retraso de mensajes y la disponibilidad de ancho de banda. Este sistema satisface tres requisitos para el desarrollo de comunicaciones: garantías por conexión, protección de sobrecarga y justicia al tráfico de "mejor-esfuerzo".

Interfaz de aplicación RTC

RTC es la interfaz de programación que permite utilizar las rutinas para el establecimiento de conexiones, transmisión de mensajes y recepción, etc. Entre sus principales características se encuentran las siguientes:

- Rutinas para el establecimiento de conexiones de forma similar a la utilización de *sockets*: las aplicaciones se registran con nombres de servicios y los clientes deben localizan esos servicios.
- La aplicación que recibe una petición de utilización, aprueba explícitamente el establecimiento de una conexión. Se aprueba el establecimiento de las mismas según los recursos disponibles.

RTCAST

RTCAST es un componente de ARMADA que permite canales *multicast* basándose en el paradigma de *grupos de procesos*. En el paradigma mencionado, un sistema es estructurado como un grupo de procesos cooperativos que dan servicios a la aplicación. Se incluyen entre otros servicios y facilidades: un *multicast* atómico, servicio de miembros del grupo y de control de admisión.

Herramientas de evaluación

El proyecto ARMADA incluye un conjunto de herramientas para la validación y evaluación de las capacidades de los sistemas generados:

- ORCHESTRA: entorno de ejecución de inyección de fallos que puede ser utilizado para evaluar los protocolos de comunicación y las aplicaciones distribuidas.
- COGENT: generador de cargas para la red de forma controlada, que permite evaluar el sistema y el rendimiento de la red de una forma controlada.

3.3.2. El modelo Time-triggered Message-triggered Object

Un entorno de programación ideal para aplicaciones de tiempo real sería aquél basado en un lenguaje de alto nivel común y fácil de ser usado por programadores como por ejemplo C++ o Java. Ese es el propósito de TMO, una forma simple y natural sintácticamente y al mismo tiempo una extensión semánticamente poderosa.

TMO (*Time-triggered Message-triggered Object*) desarrollado por K.H. Kane y otros en la Universidad de California [Kim, 1999], presenta una forma distinta a la tradicional basada en prioridades y utilizadas en gran número de sistemas de tiempo real.

Para TMO el objetivo es trasladar de una forma natural los requisitos de la aplicación de tiempo real al sistema implementado, de forma que el programador no tenga que preocuparse por establecer prioridades. Se basan para ello en cuatro principios:

1. Una de las mayores metas en la realización de programación estructurada distribuida de tiempo real es reducir la complejidad en el diseño de aplicaciones de sistemas de gran tamaño. La reducción de complejidad es lograda tratando con mayores niveles de abstracción, como objetos distribuidos, dejando los detalles a herramientas y entornos de ejecución.
2. Las prioridades fijas son atributos que pueden ser fácilmente observados por el motor de ejecución de bajo nivel. Sin embargo, es una forma poco natural de expresar la *urgencia* de los procesos de las aplicaciones.
3. Si hay requisitos de tiempo inherentes al sistema, es mejor para los diseñadores de objetos distribuidos, que se expresen en la forma más clara, simple y más fácilmente analizable en el diseño de objetos de red de alto nivel.
4. Dadas expresiones naturales de requisitos de tiempo que el diseñador ha introducido en los objetos distribuidos, debería ser un trabajo del *middleware*, sistema operativo, o herramientas de diseño, el traducir las especificaciones de alto nivel a las necesidades del motor de ejecución de bajo nivel.

Características principales

Cualquier aplicación distribuida de tiempo real debería incorporar las siguientes facilidades para permitir una especificación eficiente:

- Formas de referenciar pasado, presente y futuro.
- Invocación uniforme de métodos locales y remotos.
- Imposición de *deadlines* para la llegado de resultados de los métodos invocados.
- Acciones disparadas por tiempo.
- Ejecución concurrente de métodos de objetos
- Invocación no bloqueante de métodos de objetos.

Mientras que en C++ los objetos son unidades pasivas, los denominados objetos TMO son activos, distribuidos y de tiempo real. Más aún, son capaces de adaptarse a cambios dinámicos en la configuración de la red. TMO extiende los objetos convencionales con varias características que incluyen:

- Métodos espontáneos (SpM): Los métodos espontáneos son ejecutados cuando el reloj de tiempo real alcanza valores específicos determinados en tiempo de diseño.
- Métodos de servicio (SvM): Los métodos o servicio son ejecutados por solicitudes de mensajes desde los clientes.
- Capacidades para acceder a otros objetos TMO y al entorno de red, incluyendo canales *multicast* lógicos y dispositivos de entrada/salida.
- Restricción básica de concurrencia (BCC): bajo esta regla los métodos de servicio (SvM's) no pueden afectar la ejecución de los métodos espontáneos (SpM's) facilitando así los esfuerzos del diseñador para garantizar las capacidades de tiempo del objeto TMO. Básicamente, la activación de un SvM disparada por un mensaje desde un cliente externo es permitida únicamente cuando no existen conflictos potenciales con la ejecución de SpMs.
- Ventana de tiempo impuesta sobre cada acción de salida de un método: se indican los tiempos que cada método tarda en ejecutarse. El diseñador del objeto servidor debe estar seguro de que el objeto puede ejecutarse en los tiempos especificados. En la parte del cliente, éste impondrá un *deadline* para los resultados retornados.

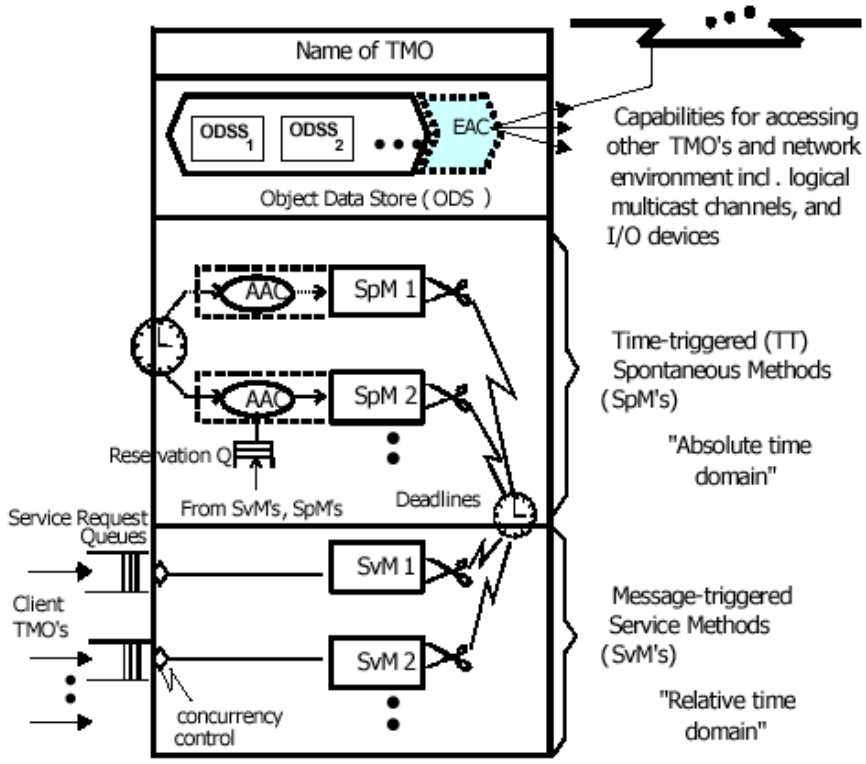


Figura 3.9. Visión general de TMO

Interacción entre objetos de tiempo real

Cualquier lenguaje de tiempo real debe soportar múltiples tipos de solicitudes de servicios. TMO incluye los siguientes:

- Llamadas bloqueantes: después de efectuar una llamada a un SvM, el cliente espera hasta que el SvM retorna un resultado. En este tipo de llamadas TMO, permite especificar *deadlines* para obtener los resultados. Si los resultados no se obtienen para el *deadline* especificado, se indicará permitiendo generar excepciones.
- Llamadas no bloqueantes: después de llamar un SvM, el cliente puede seguir ejecutándose sin tener que esperar los resultados. Más adelante, el cliente podrá consultar si ya se ha completado la operación y en tal caso recoger los resultados.
- Llamada de cliente transferidas: en TMO, un SvM puede pasar una solicitud de un cliente a otro utilizando el denominado *client-transfer call.* Este proceso se puede repetir en una cadena hasta que el último SvM retorna los resultados al cliente.
- Canales *multicast*: esta característica permite la creación dinámica de canales lógicos *multicast* compartidos entre los procesos distribuidos para su interacción de una manera sencilla.

3.3.3. CRL y su entorno de ejecución

A.Stoyen, T. Marlowe y M. Younis presentan en [Stoyen et al., 1999] un entorno de desarrollo para aplicaciones distribuidas complejas de tiempo real basándose en el desarrollo de dos nuevas áreas en la comunidad de tiempo real:

- Soporte de compiladores para sistemas de tiempo real.
- Sistemas de computador complejos: grandes aplicaciones con dependencias complejas en plataformas heterogéneas.

Como motivación para su trabajo encuentran la falta de entornos para el desarrollo de aplicaciones de tiempo real complejas y distribuidas que dificultan la labor de los programadores.

Con lenguajes específicos para tiempo real, los programadores se beneficiarían al igual que otros lenguajes como COBOL o Fortran beneficiaron a ámbitos concretos de la computación.

Otro factor a tener en cuenta es la necesidad de realizar un análisis de planificabilidad que no puede ser realizado en los lenguajes tradicionales de

programación por la falta de soporte sintáctico y semántico, lo que imposibilita la ejecución de los peores tiempos de ejecución.

La necesidad de lenguajes de tiempo real específicos, motiva a los autores a realizar un entorno completo de desarrollo de aplicaciones de tiempo real basándose en un lenguaje de su propia construcción: CRL (*Complex Real-Time Systems*). Este lenguaje se basa en otro anterior denominado Real-Time Euclid que fue el primero en incluir un analizador completo de planificabilidad independientemente de la metodología de planificación y en rechazar la compilación de programas con segmentos cuyos tiempos de ejecución no fueran acotados. Para la realización de este entorno, además de CRL, lenguaje específico de tiempo real; los autores utilizan trabajos de compiladores y entornos de desarrollo, teniendo en mente la necesidad de poder expresar restricciones de granularidad fina, a un nivel distinto del de los procesos, como por ejemplo, entre cualquier par de instrucciones.

Características del entorno

Entre las principales características del entorno desarrollado, se encuentran las siguientes:

- Utilización de un lenguaje para aplicaciones complejas de tiempo real: CRL.
- Realización de análisis de planificabilidad, verificación de restricciones, pruebas de ejecución y advertencias.
- Entorno de ejecución con soporte para comunicaciones y planificación dinámica.
- Interfaz de usuario para la monitorización del sistema.
- Herramientas para las siguientes tareas:
  - Realización segura de transformaciones automáticas en los programas.
  - Monitorización, *profiling*, depuración, pruebas y evaluación.
  - Asignación de módulos del sistema a procesadores.
  - Generador de cargas.

Herramientas suministradas

Se incluyen numerosas herramientas para ayudar en el desarrollo de las aplicaciones:

- *Timing tool*: es una herramienta para la estimación estática y segura de tiempos de ejecución, que desde el código fuente e información sobre la plataforma de

ejecución, genera un código intermedio anotado con los tiempos de ejecución (peores tiempos por método y tiempos por sentencia ejecutable). Estas anotaciones serán utilizadas por otras herramientas y por el entorno de ejecución

- Herramientas para transformaciones y análisis: mediante la utilización de grafos de dependencia de datos, grafo de llamadas, etc. tienen como propósito la mejora del código y facilitar el posterior análisis. Estas transformaciones se realizan como una secuencia de pasos, algunos de los cuales pueden repetirse y tienen dos efectos:

- Cambios en el código de acuerdo a la transformación aplicada.
- Posibilidad de relajar/fortalecer restricciones a través de la interacción con el usuario.

- Analizador de planificabilidad: la entrada a este proceso es el resultado de las transformaciones previas, puede realizar análisis exhaustivo o heurísticos con posible generación de planificaciones parciales estáticas. Los autores ven el análisis de planificabilidad como una técnica de verificación en tiempo de compilación, y no sólo como la construcción de planificaciones estáticas. El análisis de planificabilidad consistiría en la exploración de todas las posibles combinaciones de rutas de ejecución de tareas que pueden ser planificadas por un planificador dinámico sin violar restricciones temporales. Este tipo de análisis tiene un coste exponencial, por lo que incluye la posibilidad de retroceso al motor de transformaciones para intentar lograr una planificación factible, siendo necesario reducir al máximo el espacio del problema.

Entorno de ejecución

El entorno de ejecución consiste en la emulación de una plataforma homogénea interconectada por una red con retrasos predecibles en las comunicaciones. Este entorno, estaría formado por los siguientes componentes:

- *Kernel*: abstracción de un sistema distribuido.
- Simulador de red: proporciona retrasos de tiempo debido a la distribución de los objetos.
- Interfaz de usuario: muestra medidas y estadísticas sobre transformaciones y efectos sobre rendimiento, *deadlines*, utilización del procesador, etc.

3.3.4. Sistemas Operativos de Tiempo Real

En el ámbito de los Sistemas Operativos de Tiempo Real existen numerosos trabajos realizados, mostrándose en este apartado algunos de los más relevantes y desarrollándose algo más las características de los relacionados más directamente con tecnologías actuales de componentes, Java, etc. tales como Jbed o Portos.

Chimera [Chimera, 2005] es un RTOS multiprocesador desarrollado en Carnegie Mellon, pensado para soportar el desarrollo de software reconfigurable dinámicamente para robots.

Industrial TRON [Industrial, 2005] es una especificación de RTOS para sistemas empotrados. Es la especificación de SO estándar de-facto en Japón para aplicaciones de consumo.

LynxOs [Lynx, 2005] es un SO estilo UNIX, tanto desde el punto de vista del usuario como del programador. Es una reimplementación de UNIX teniendo en cuenta y pensando en el tiempo real y ofreciendo POSIX 1.003.1

Jbed

Jbed [Mattsson et al., 1999] es un sistema operativo de tiempo real con una máquina virtual integrada para sistemas empotrados y aplicaciones de Internet. Entre las características más destacables de Jbed figuran su pequeño tamaño y su velocidad lo que, según sus autores, lo hacen apto para la creación de aplicaciones para sistemas empotrados utilizando únicamente Java, incluso para aplicaciones duras de tiempo real.

Sin utilizar la especificación de tiempo real para Java [Bollella et al., 2000], se han modificado aspectos originales de la máquina virtual Java como puedan ser el recolector de basura (sustituido por uno interrumpible) o la creación de nuevas clases como, por ejemplo, la clase *Task,* que representa a la tarea típica de planificación de lenguajes tales como Ada y que permite expresar *deadlines*, tiempos máximos de ejecución, etc.

Jbed es un sistema de los denominados *time-triggered* (disparado por tiempo): las aplicaciones reciben pequeños "fragmentos" de tiempo para realizar sus tareas. En lugar de utilizar prioridades, las rutinas disparadas por el paso del tiempo tienen *deadlines* que son usualmente conocidos a priori.

Jbed garantiza que el sistema puede ser planificado si las duraciones de las tareas de tiempo real fueron dadas correctamente. También se permite la integración de componentes en sistemas ya en ejecución. Para ello se realizan pruebas de admisión comprobando si el sistema completo seguirá siendo planificable tras insertar un nuevo componente.

Portos

Portos [Szypersky et al., 2002] es un sistema operativo de tiempo real orientado a componentes que permite indicar requisitos de tiempo y añadir componentes en tiempo de ejecución. Puede ser utilizado con dos de los estándares más comunes en el mercado como son COM de Microsoft y Java.

Como características interesantes incluye dos novedades con respecto a otros sistemas operativos de tiempo real:

- Planificación por plazos de una manera similar a TMO, en lugar de indicar prioridades, se indican los instantes de tiempo en los que tienen que ejecutarse las distintas tareas.
- Posibilidad de añadir nuevos componentes en tiempo de ejecución sin perjudicar el cumplimiento de las aplicaciones que se estén ejecutando. Al crearse una tarea deben especificarse la duración de la misma y un plazo. Si la tarea consume demasiado tiempo de computación, entonces será abortada y se generará una excepción. El planificador garantiza, por su parte, a la tarea admitida que tiene suficiente tiempo para completarse antes de que se cumpla su plazo.

PORTOS incluye también un entorno de desarrollo para crear aplicaciones denominado Denia ,que puede ser utilizado para transferir código a sistemas empotrados donde se ejecute PORTOS.

VxWorks

VxWorks [VxWorks, 2005] es un sistema operativo de tiempo real válido tanto para aplicaciones complejas como para empotradas. Es muy utilizado en aplicaciones industriales y espaciales, por lo que tiene un entorno de desarrollo muy completo, llamado Tornado, que permite su uso e integración de forma muy sencilla.

VxWorks incluye tres componentes integrados que se relacionan entre sí para ofrecer un comportamiento predecible:

- Un sistema operativo de tiempo real escalable y de alto rendimiento, que se ejecuta sobre un procesador dedicado.
- Un conjunto de potentes herramientas de desarrollo, que se usan en un puesto de trabajo general.
- Una gran variedad de opciones de comunicaciones.

Sus características principales son:

- Dominios de protección, que permiten a los desarrolladores crear contenedores lógicos que aíslen y protejan las aplicaciones que se ejecuten en distintos dominios.
- Manejo de recursos.
- Manejo de multitarea eficiente, sin límite en el número de tareas. Planificación interrumpible y *round-robin*, con 256 niveles de prioridad. Cambios de contexto rápidos y deterministas.
- Comunicaciones entre tareas flexibles. Semáforos con herencia de prioridades; colas de mensajes; posibilidad de usar planificación, tuberías, semáforos, señales y hebras POSIX; memoria compartida; etc.
- Enlazado y carga de componentes incremental, con posibilidad de actualizar aplicaciones o componentes sin parar el sistema.
- Manejo de interrupciones y excepciones.
- Manejo de memoria dinámica.
- Reloj del sistema y facilidades de gestión del tiempo.
- Implantado sobre una amplia variedad de plataformas, como son: la gama de procesadores Intel Pentium, Motorola/IBM PowerPC, ARM, MIPS, etc. ofreciendo por tanto una alta posibilidad de utilización.
- Sistema I/O y sistema de ficheros rápido y flexible, con soporte de: entrada/salida asíncrona y manejo de directorios POSIX; SCSI; PCMCIA; sistema de ficheros compatible con MS-DOS; sistema de ficheros en CD-ROM ISO 9960; y manejo opcional de sistema de ficheros sobre flash (TrueFFS).
- Gran variedad de soporte de red: TCP/IP, UDP, sockets, FTP, TFTP, DNS, telnet, rsh, NFS, RPC, ATM, ISDN, etc.

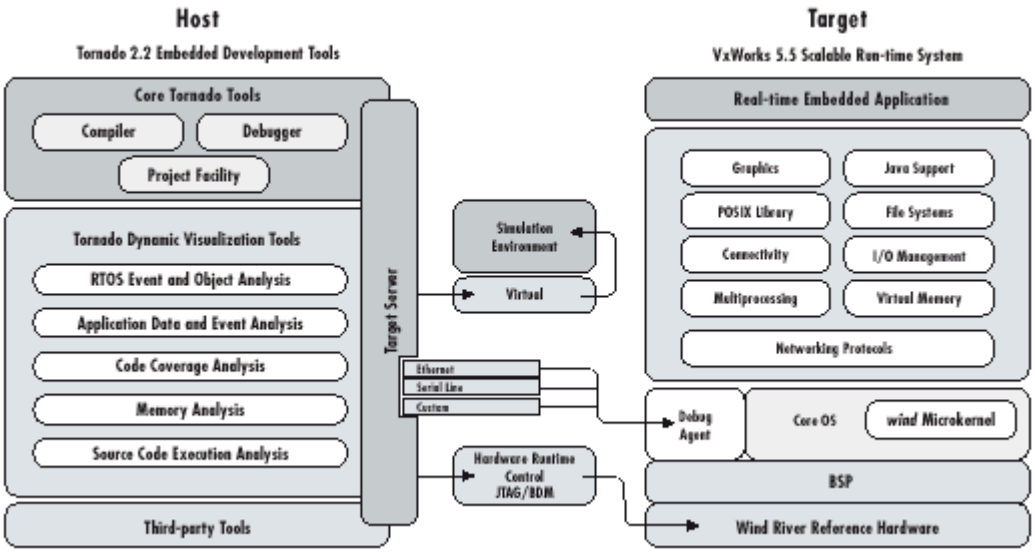


Figura 3.10. Desarrollo bajo VxWorks

Para acelerar el aprendizaje, configuración, desarrollo y depurado de aplicaciones sobre VxWorks, está el entorno de desarrollo Tornado, que consiste en un conjunto integrado de herramientas y utilidades, cuyas características más destacadas son:

- Compatibilidad total con ANSI C y algunas características añadidas de C++, para manejo de excepciones y soporte de *templates*.
- Compatibilidad con POSIX 1003.1/.1b/.1c
- Depurador simbólico.
- Herramienta de monitorización de alto rendimiento.
- Utilidades para tareas, núcleo e información del sistema.
- Librerías con más de 200 rutinas de utilidades.
- Posibilidad de arranque desde ROM, disco local o red.
- Diseño escalable, lo que permite el desarrollo de gran variedad de aplicaciones, tanto empotradas como de más alto nivel.

MaRTE OS

MaRTE OS [Aldea y González, 2000] es un núcleo de tiempo real para aplicaciones empotradas que requieran el subconjunto *Minimal Real-Time POSIX.13*. El código está escrito en ADA en su mayor parte. Permite el desarrollo en ADA y C, utilizando los compiladores de GNU. Además es posible hacer una depuración remota, usando gdb.

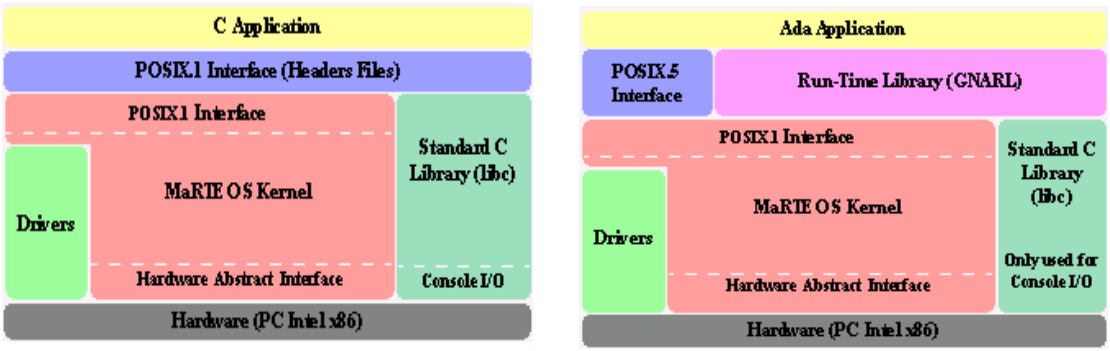


Figura 3.11. Arquitectura de MaRTE OS

El entorno de desarrollo consta de un puesto de trabajo ejecutando Linux más un PC simple basado en un 386, ambos conectados por *Ethernet* para realizar el arranque de las aplicaciones y por una línea serie, que permite la depuración, e incluso emulación sin necesidad de disponer de los dos equipos, utilizando únicamente un emulador donde se ejecuta la aplicación desarrollada.

Las características fundamentales que implementa son las siguientes:

- Manejo de hebras *pthread*.
- Planificación con prioridades y distintas políticas: FIFO, *round-robin*, etc. Además permite definir políticas propias.
- Ofrece *mutex* y variables de condición para sincronización.
- Relojes y *timers*
- Servicios relacionados con el tiempo, como son la suspensión de hebras y los retrasos, tanto absolutos como relativos.
- Entrada/Salida por consola.
- Manejo de memoria dinámica, aunque muy simplificado.
- Manejo de interrupciones a nivel de aplicación.

C5 y Jaluna

Jaluna-1 [Jaluna, 2005] es un entorno de desarrollo para software de tiempo real basado en componentes y que utiliza el sistema operativo C5 (ChorusOS) para la ejecución de aplicaciones.

La utilización de Jaluna junto con C5 está orientada especialmente a los sistemas empotrados: teléfonos, televisores, automóviles, etc. ofreciendo un interesante entorno de trabajo con su código fuente disponible para los desarrolladores.

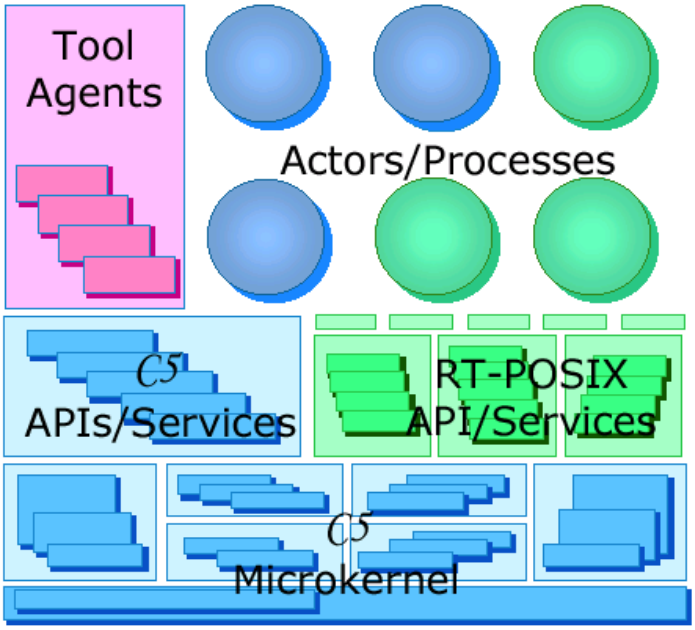


Figura 3.12. Visión general de Jaluna

Entre las principales características de Jaluna destacadas por sus creadores se encuentran las siguientes:

- Utilización del *microkernel* C5 con planificador modular, gestión de memoria, comunicación, sincronización, servicios de tiempo, gestión de interrupciones, etc.
- Utilización de RT-POSIX con mejoras para sistemas empotrados.
- Sistema basado en *host target* similar a los empleados en otros sistemas operativos como, por ejemplo, MaRTE OS.

El sistema operativo C5 proporciona un conjunto fino de componentes ejecutándose sobre un *microkernel*. Estos componentes pueden ser reemplazados por otros similares para adaptarse a las necesidades específicas del hardware o de las aplicaciones. Podemos encontrar componentes planificadores, para gestión de la memoria, comunicaciones, sincronización, servicios de tiempo, etc.

Capítulo 4. Un modelo de análisis

El modelo de componentes desarrollado, con su plataforma de ejecución basada en RT-CORBA, requiere técnicas de análisis para poder ser utilizado en sistemas de tiempo real duros. En esta sección se muestra una extensión a las técnicas de análisis basadas en SDL descritas en [Llopis, 2002], de forma que puedan aplicarse al modelo UM-RTCOM sobre RT-CORBA. De esta forma, dichas técnicas pueden utilizarse en sistemas distribuidos considerando además la sobrecarga de las comunicaciones generadas por RT-CORBA [Díaz et al., 2005].

Hay que considerar dos aspectos para poder realizar el análisis. Por un lado, hay que disponer del modelo SDL de las aplicaciones. Este metamodelo del código de los componentes puede ser construido de forma semi-automática con las herramientas del entorno. Por otra parte, hay que considerar las comunicaciones y la sobrecarga del *middleware*. Para ello se ha desarrollado un modelo SDL del subconjunto de RT-CORBA utilizado.

El modelo de ejecución del modelo de componentes está basado en la extensión de tiempo real de SDL. La utilización de SDL está motivada por su facilidad para expresar interacciones entre procesos en diferentes niveles de abstracción, siendo esto aplicable al modelo de componentes. Existen aproximaciones similares basadas en el perfil de tiempo real de UML [Yau y Zhou, 2002], pero la falta de una semántica formal y su complejidad para expresar interacciones dinámicas entre procesos hacen que estos problemas tengan que ser resueltos antes de poder utilizarlos.

La utilización de la extensión de tiempo real de SDL va a permitir aplicar posteriormente la metodología de análisis descrita en [Alvarez et al., 1999] y además extenderla a aplicaciones distribuidas.

La incorporación del análisis de planificabilidad basado en SDL para RT-CORBA permite obtener tres importantes resultados:

- Es posible incluir el comportamiento del middleware de comunicaciones en el diseño de las aplicaciones, y de esta forma puede realizarse la simulación del sistema completo. Para realizar esto es necesario incluir además información sobre la plataforma de comunicaciones. Se propone para ello, el diseño de un modelo del protocolo de comunicaciones CAN.
- La fase de implementación se simplifica, ya que, la integración del middleware RT-CORBA permite generar código directamente desde el diseño.
- Puede realizarse el análisis de planificabilidad del sistema de tiempo real distribuido, garantizando así las restricciones temporales.

En el resto de este capítulo se verán aspectos de modelado SDL de las aplicaciones. Posteriormente se presentará el modelado de las principales características de RT-CORBA: los modelos de prioridad (propagado por el cliente y declarado en el servidor), *thread pools*, exclusión mutua y la gestión de la conexión. Finalmente, se verá una técnica de análisis de planificabilidad que puede ser aplicada a los modelos SDL descritos y aplicada al modelo UM-RTCOM bajo RT-CORBA.

4.1. Modelado de componentes con SDL

Para la utilización de SDL con el modelo de componentes, es necesario establecer una equivalencia entre los elementos del modelo a elementos SDL, cumpliendo además con los requisitos que la extensión de tiempo real de SDL establece. De esta forma, cada componente genérico va a ser convertido en un bloque SDL. Este bloque incluye un proceso por cada componente activo y un objeto pasivo por cada componente pasivo.

La interconexión entre componentes se realiza mediante la interconexión de señales de SDL y los mecanismos de interacción dependerán de la definición de la interfaz del componente (ej: señales para llamadas asíncronas y eventos, y RPC para llamadas síncronas).

4.1.1. Modelado de componentes activos

El papel de los componentes activos es central para el modelo de análisis que se desarrollará posteriormente. Cada componente activo será convertido en un proceso

SDL de la siguiente forma: el proceso tendrá una transición por cada entrada que el componente activo pueda tener sobre cada interfaz y sobre cada evento.

Siguiendo la semántica de SDL, cada transición está compuesta por una recepción de señal, un conjunto de bloques secuenciales o llamadas síncronas y un conjunto de envío de señales (llamadas asíncronas). En este sentido, se usarán estados de los procesos para modelar el comportamiento del componente (recepción condicional o envío de señales, construcciones iterativas, etc.). El modelo debe reflejar el comportamiento del peor caso del componente en cada estado, incluyendo los bloques secuenciales de código entre puntos de planificación (*wait*, *call* o *raise*) y la interacción con otros componentes.

El siguiente ejemplo muestra un componente con tres tipos de componentes activos (`Tipo_activo1`, `Tipo_activo2` y `Tipo_activo3`), detallándose el código de `Tipo_activo1`. En la Figura 4.1 puede verse el modelo SDL resultante para el componente con tres procesos SDL para las tres instancias de componentes activos. En la parte derecha se muestra el diagrama del proceso equivalente para la instancia `activo1` del tipo de componente `Tipo_activo1`.

Ej:

```
Component sistema {
  interface I2 {
      void op1(in short arg1,
               in short arg2,
               in short arg3);
      void op2();
      void op3();
}

Component implementation sistema {

  Active Tipo_activo1 {
  …
  void execute(){
    wait I2.op1(…) { seq_block1; }
         I2.op2(…) { seq_block2; }
         I2.op3(…) { seq_block3; }
    call o1.op1(…);
    }
  }

  Active Tipo_activo2 {…}
  Active Tipo_activo3 {…}
  Tipo_activo1 activo1;
  Tipo_activo1 activo2;
  Tipo_activo1 activo3;
}
```

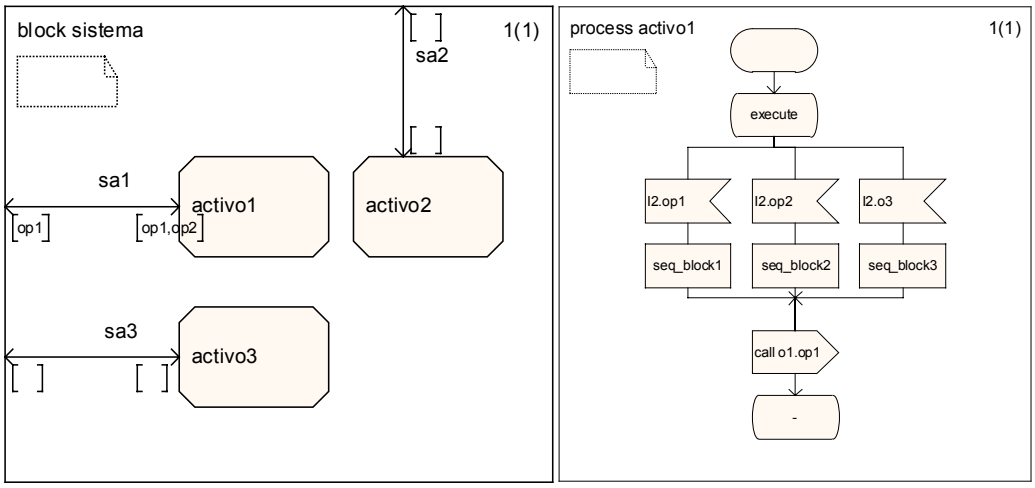


Figura 4.1. Traducción SDL de componentes activos

El modelo debe ser también consistente con el grafo de ejecución del componente activo (es decir, cada transición debe corresponder a una secuencia en ese grafo) y los nombres de las tareas en las transiciones deben ser consistentes con las asignadas por el compilador a los bloques de prueba que tienen que ser instanciados por el desarrollador del componente.

4.1.2. Modelado de componentes pasivos

SDL'2000 [Doldi, 2001] proporciona un mecanismo para modelar datos compartidos, pero, sin embargo, no es predecible. En [Llopis, 2002] se propone un mecanismo para implementar datos compartidos de una manera predecible.

Los componentes pasivos son modelados a través de este mecanismo encapsulando datos compartidos y recursos en una clase especial de procesos. Estos procesos son externamente como cualquier otro SDL, pero su comportamiento está limitado de la siguiente manera: actúan como procesos servidores pasivos, es decir, no inician ninguna acción por ellos mismos, utilizando sólo RPC como mecanismo de comunicaciones, y esperando siempre por recibir RPCs de otros procesos que deseen invocar sus servicios.

El siguiente ejemplo muestra la definición de un componente pasivo `DatoComp` de tipo `Tipo_DatoComp` con dos métodos `read` y `write`. La traducción SDL resultante puede verse en la Figura 4.2.

Ej:

```
Component implementation sistema {

  Passive Tipo_DatoComp {
        void read() {…}
        void write() {…}
  }

  Tipo_DatoComp DatoComp;
}
```

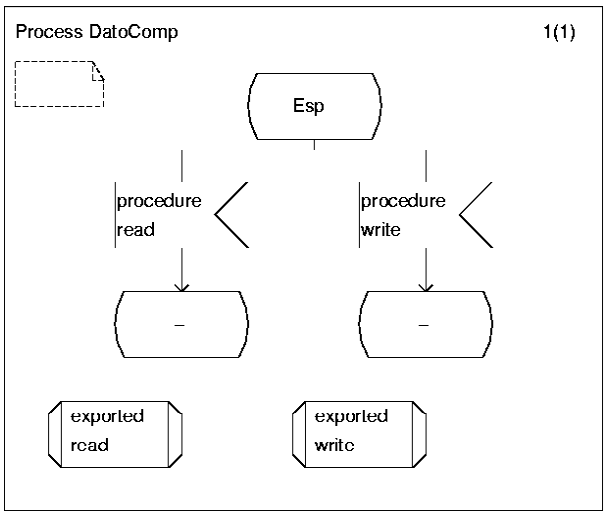


Figura 4.2. Traducción SDL de componentes pasivos

Una restricción adicional para que el modelo sea predecible es que el tiempo de bloqueo durante cada transición debe estar acotado. Cada uno de estos procesos tiene un techo de prioridad asignado, que es el máximo de las prioridades entre todas las transiciones de otros procesos donde se accede al recurso. De esta forma, se evitan posibles fuentes de inversión de prioridad en el acceso a datos y el tiempo de bloqueo es predecible.

4.1.3. Asignación de prioridades

El resultado de la traducción desde el modelo de componentes a SDL permite aplicar el análisis de planificabilidad propuesto en [Alvarez et al., 1999] para sistemas monoprocesador y basado en el peor tiempo de respuesta presentado por Joseph y Pandya [Joseph y Pandya, 1986]. Para integrar este análisis hay que considerar la extensión de tiempo real de SDL donde cada transición tiene una prioridad asociada y el procesador puede ser apropiado por transiciones de más alta prioridad del sistema. Para

cada evento del sistema, el análisis calculará su peor tiempo de respuesta sumando los tiempos de respuesta de las transiciones que toman parte en la respuesta al evento.

Tomando en cuenta las extensiones de tiempo real propuestas, cada transición debe tener una prioridad asociada que es determinada por las relaciones de precedencia entre ellas y los requisitos de tiempo externos. La prioridad será determinada utilizando razón monótona o tiempos límites.

En la Figura 4.3 puede verse un ejemplo de asignación de prioridad con relaciones de precedencia entre transiciones y la propagación de eventos entre componentes. El componente *A* tiene dos componentes activos, *A1* y *A2*, que están esperando por los eventos *E1* y *E2* y, adicionalmente, utilizan interfaces de salida que invocan servicios de *B*. Estos servicios activan a su vez a los componentes *B1* y *B2* (activos) los cuales acceden al componente *P* (pasivo). El evento *E1* tiene un periodo de 30 ms. Y el evento *E2* tiene un periodo de 50 ms.

La traducción SDL determina un proceso para cada componente activo y pasivo. La prioridad de cada transición está determinada por las relaciones de precedencia entre los componentes. Así, por ejemplo, la prioridad de la transición existente en el proceso *B1* está determinada por la prioridad del proceso *A1*, que es inicialmente asignada tomando en cuenta los requisitos temporales del evento externo *E1*. En el caso del componente pasivo *P* (con techo de prioridad), la prioridad de la transición es el máximo de las prioridades de los procesos donde el recurso es accedido (*B1* y *B2*).

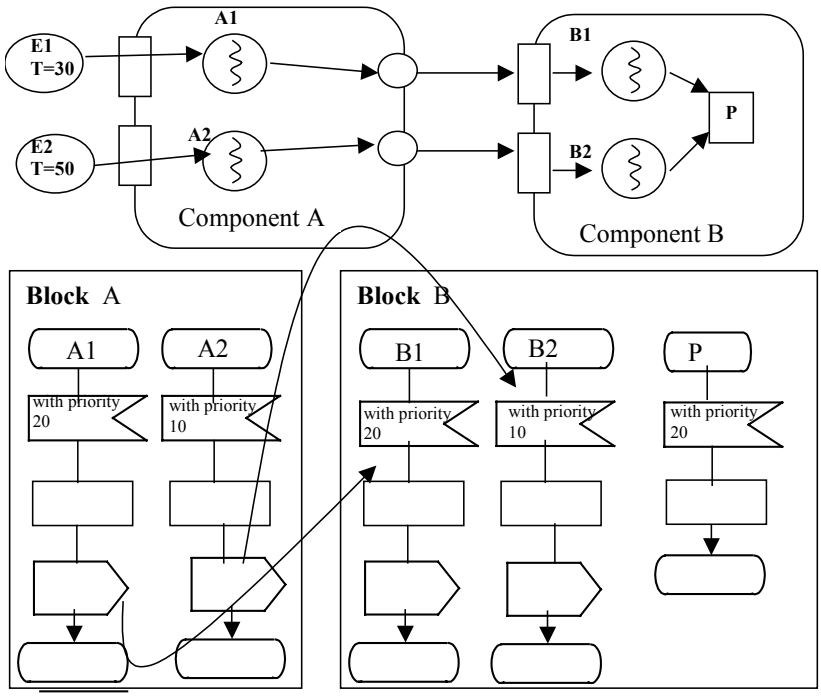


Figura 4.3. Asignación de prioridades y cadenas de eventos

4.2. Combinación de RT-CORBA y SDL

La solución para el problema del análisis de sistemas basados en RT-CORBA se basa en crear un modelo SDL para RT-CORBA que pueda ser utilizado por el usuario y combinado junto con el modelo SDL de las aplicaciones del usuario y el modelo de la plataforma de comunicaciones utilizada. De esta forma, la aplicación completa puede ser analizada en las primeras fases del diseño e incluso se puede generar código utilizando las herramientas adecuadas.

En la Figura 4.4 puede verse el modelo SDL para una aplicación CORBA genérica. Este modelo representa un sistema básico con un objeto CORBA (contenido en la aplicación `Apl2`) y un cliente (aplicación `Apl1`). Se han incluido bloques SDL adiciones representado el *stub* del cliente (`ClientORB`) y el código del servidor (`ServerORB`) incluyendo el adaptador de objetos y el *skeleton*. Finalmente, la plataforma de comunicaciones y el protocolo GIOP se han representado con otros bloques SDL.

Todos los bloques están interconectados con señales tales como *methodX* para los diferentes servicios o los pares `SendMessage`/`ReceiveMessage` y `SendReply`/`NewMessage`, que están basados en el protocolo GIOP de CORBA. De esta forma, se puede representar un sistema basado en CORBA con SDL, beneficiándose de todas las características de SDL. En particular, se pueden modelar las características de RT-CORBA, obteniendo sistemas RT-CORBA analizables.

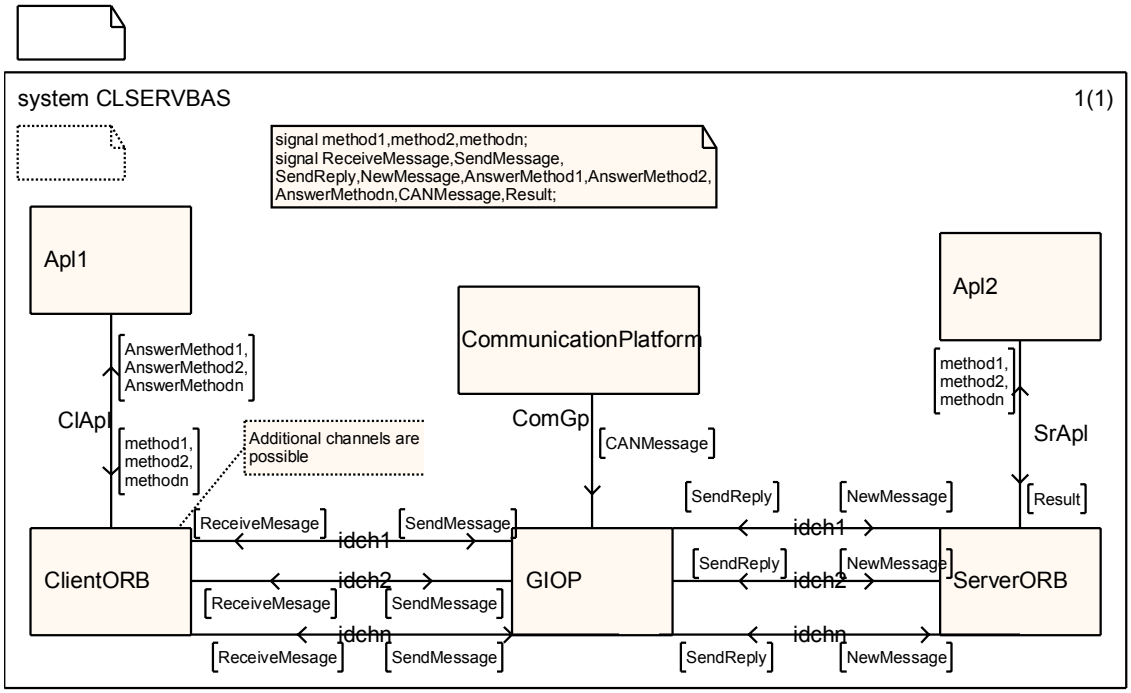


Figura 4.4. RT-CORBA modelado con SDL

Supóngase que se realiza una invocación remota desde `Apl1` (por ejemplo, `method1`). La aplicación enviará una señal `method1`, que es capturada en el bloque `ClientORB`. Este bloque se encarga del empaquetamiento de los parámetros, generando un mensaje GIOP equivalente que es enviado mediante una señal `SendMessage`.

El bloque `GIOP` se responsabiliza ahora de la transmisión de los datos entre los nodos de la aplicación. En particular, es responsable de la interacción con la plataforma de comunicaciones (bloque `CommunicationPlatform`).

La invocación es transmitida por la plataforma de comunicaciones, llegando finalmente al espacio de direcciones del servidor, bloque `ServerORB` con la señal SDL `NewMessage`. En este bloque se realiza el desempaquetamiento de los parámetros y se realiza la transmisión a `Apl2`. En el caso de que la aplicación tenga una respuesta para `Apl1`, se tiene que seguir un camino inverso, desde `Apl2` a `ServerORB` y desde `ServerORB` al bloque `GIOP` con la señal `SendReply`. Esta señal será capturada por el bloque `ClientORB` y la respuesta será transmitida finalmente a `Apl1`.

Para una aplicación en concreto, solamente hay que modificar algunos de los bloques genéricos proporcionados usando reglas simples que pueden ser seguidas por el usuario o por alguna herramienta automática, obteniendo finalmente una representación SDL del sistema. Estas modificaciones incluyen:

- Bloque `GIOP`, que tiene que ser modificado de acuerdo con la implementación RT-CORBA utilizada (ej: TAO o ROFES).
- Plataforma de comunicaciones, que también tiene que ser sustituida por una representación SDL de la utilizada finalmente (ej: CANIOP [Lankes et al., 2003] o ATM).
- Bloques `ClientORB` y `ServerORB`, que tienen que ser adaptados a los bloques reales cuyo código es automáticamente generado por el compilador IDL, siendo también este código dependiente de la implementación CORBA utilizada y de las interfaces IDL de las aplicaciones.

Hay que hacer notar que el usuario o la herramienta sólo tienen que realizar pequeñas modificaciones al modelo básico propuesto tales como el código del *stub* o el nombre de los métodos. Otros bloques son proporcionados por el entorno o las herramientas, tales como la implementación RT-CORBA. Otros aspectos, tales como la interconexión entre bloques, señales, etc. permanecen sin cambios.

En las siguientes secciones se mostrará cómo pueden ser modeladas la principales características de RT-CORBA, tales como modelos de prioridad, *thread pools*, *mutex*, etc.

4.3. Modelado de las políticas de prioridad

Un problema primario relacionado a las aplicaciones de tiempo real distribuidas está relacionado con los diferentes esquemas de prioridad en diferentes sistemas operativos, prioridades no respetadas en el servidor, etc. Como se ha visto, RT-CORBA proporciona mecanismos para superar estos inconvenientes gracias a los modelos de prioridades propagadas por el cliente o declaradas en el servidor. En esta sección se presenta el modelado de estas características en la extensión de tiempo real de SDL. Una parte considerable del modelo SDL para ambos es similar, así que en primer lugar se presentará la parte común y posteriormente se explicarán las diferencias entre ambas.

4.3.1. Visión del cliente

En la parte del cliente, la interacción entre la aplicación y RT-CORBA es realizada en el bloque `ClientORB`. En este bloque, se ha utilizado un proceso SDL para modelar cada servicio CORBA (es decir, métodos de la interfaz IDL). De esta forma, cuando la aplicación realiza una petición para un servicio, la solicitud es capturada por el correspondiente proceso SDL a través de la señal *methodX*. El resultado de esta interacción es la creación de una nueva señal SDL denominada `SendMessage`, que es enviada fuera del bloque `ClientORB`. La señal `SendMessage` representa un mensaje CORBA genérico, *Request*, utilizado por el protocolo GIOP de CORBA para comenzar nuevas solicitudes. Finalmente, es muy posible que las solicitudes tengan una respuesta asociada. Para ello, los servicios tienen que procesar la señal `ReceiveMessage`, que contiene la respuesta a la solicitud. La respuesta es entonces procesada y transferida a la aplicación a través de la señal *AnswerMethodX*.

La Figura 4.5 muestra servicios RT-CORBA genéricos transformados en procesos SDL. En una aplicación concreta, los nombres genéricos tienen que ser cambiados (por alguna herramienta automática) mostrando los nombres reales. Esto incluye los nombres de los servicios y de señales tales como *methodX* y

AnswerMethodX. Por el contrario, las señales `NewMessage` y `ReceiveMessage` son genéricas y están relacionadas con el protocolo GIOP.

En el interior de los procesos representando a los servicios (`Service1`, ..., `Servicen`) hay que modelar otras características del *middleware* RT-CORBA , tales como el establecimiento de la conexión o el empaquetamiento de parámetros. También hay que delegar la realización de la solicitud al bloque `GIOP` (usando la señal `SendMessage`). Y por último, el proceso tiene que estar preparado para recibir la respuesta a la solicitud (a través de la señal `ReceiveMessage`).

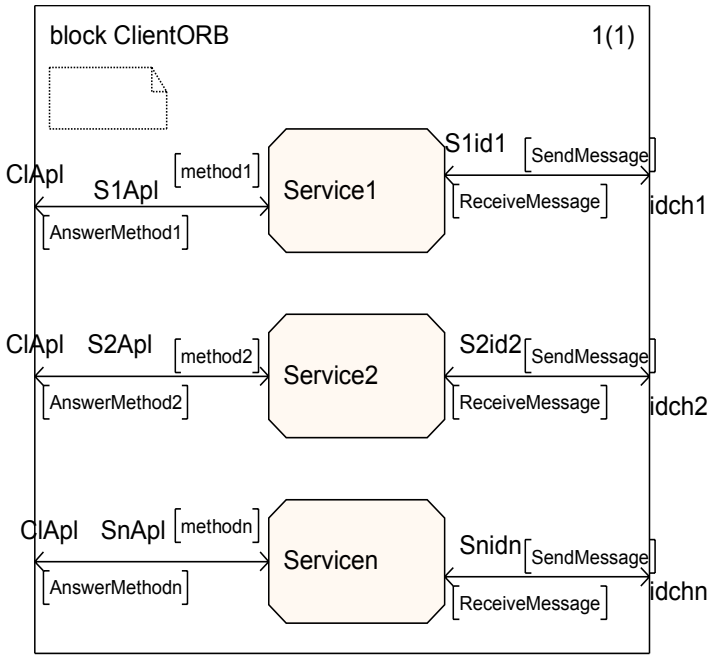


Figura 4.5. Traducción de servicios

La Figura 4.6 muestra el interior de uno de estos servicios. El proceso servicio está esperando por dos señales: `Method1` y `ReceiveMessage`. `Method1` es recibido cuando la aplicación realiza una nueva solicitud. La otra señal, `ReceiveMessage`, es recibida cuando una respuesta es transferida desde el servidor.

Cuando una nueva solicitud es iniciada (se recibe la señal `Method1`), el *stub* es responsable del establecimiento de la conexión con el servidor. Este procedimiento será diferente dependiendo de la implementación RT-CORBA utilizada. Después de esto, los datos relacionados con la invocación son empaquetados y finalmente se genera una nueva señal, `SendMessage`, con toda la información requerida para la realización de la solicitud.

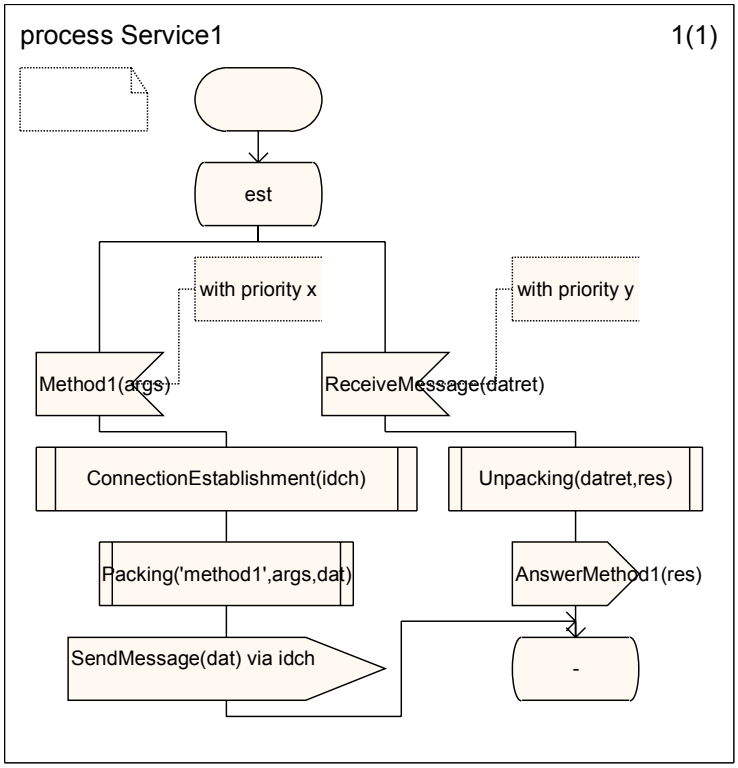


Figura 4.6. Servicio: parte del cliente

Los servicios pueden también esperar la señal `ReceiveMessage`. Esta señal es recibida como respuesta a un envío previo de `SendMessage`. Los pasos son los opuestos a los de la señal `Method1`. La respuesta es desempaquetada y enviada a la aplicación a través de la señal `AnswerMethod1`.

4.3.2. Visión del servidor

La parte del cliente es más compleja que la del servidor, debido a que hay que considerar factores adicionales tales como el adaptador de objetos (POA) y su interacción con el *skeleton* y el sirviente. En este caso, hay un proceso por cada POA existente en la aplicación. Los POAs están conectados a los *skeletons* y los *skeletons* están conectados, a su vez, con los sirvientes que representan a los objetos CORBA (contenidos en la aplicación). La Figura 4.7 muestra estos elementos y la interacción entre ellos.

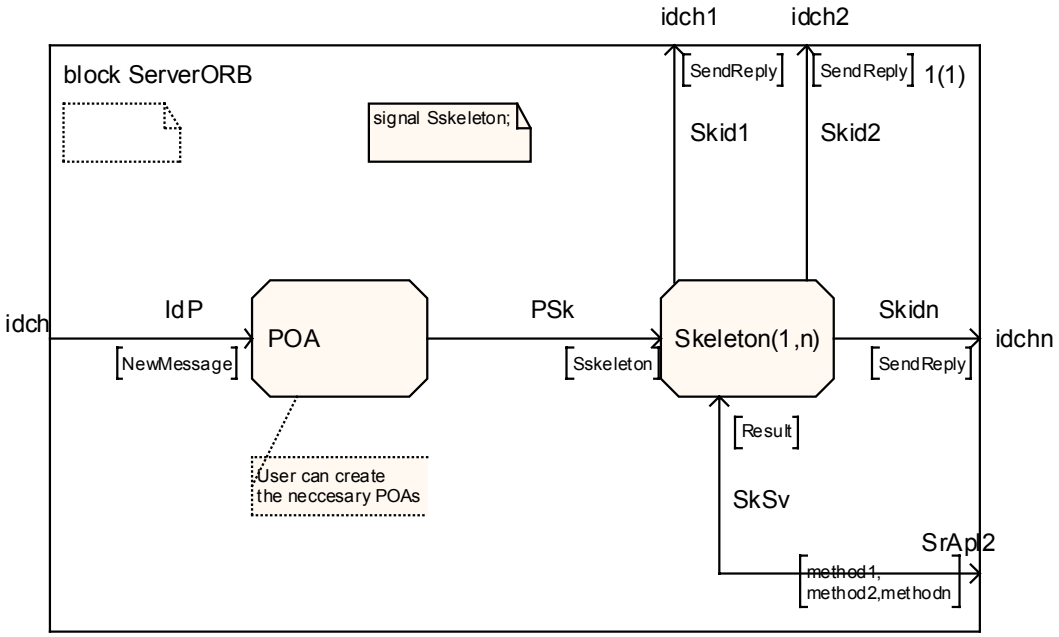


Figura 4.7. Bloques de la parte del servidor

El POA está esperando a señales del tipo `NewMessage` (recibidas desde los procesos servicio). Tras la llegada de una de estas señales, el POA identifica el objeto destino, transfiriendo la solicitud al proceso *skeleton* correspondiente, el cual invoca el método en el servidor.

Si la solicitud tiene una respuesta asociada, ésta es transmitida de vuelta al proceso servicio a través de la señal `SendReply`. En este modelado se utiliza para ello al proceso *skeleton* (el modelado propuesto divide el papel del POA entre dos procesos SDL).

Hasta este punto no hay diferencias entre los modelos de prioridades propagados por el cliente o declarados en el servidor. El POA va a establecer las diferencias entre los dos modelos y sus traducciones en SDL:

- Modelo *server declared*: las invocaciones en este modelo son ejecutadas con la prioridad CORBA del servidor. Este modelo está implícitamente cubierto por la extensión de tiempo real de SDL y no hay que hacer modificaciones o extensiones al modelado propuesto. Las transiciones SDL asociadas al objeto son ejecutadas con la prioridad del objeto.
- Modelo *client propagated*: en este modelo la prioridad de los clientes debe ser respetada en la parte del servidor. Este modelo de prioridad implica cambios dinámicos en las prioridades desde el punto de vista de SDL y no está cubierto por

los trabajos previos. Las transiciones asociadas a los diferentes métodos del sirviente pueden ser ejecutadas con diferentes prioridades dependiendo del cliente que haga la solicitud.

De esta forma, hay que extender el modelo SDL para el modelo *client propagated*. En particular, se propone una nueva primitiva: ***ThePriority***, que cuando es invocada, cambia la prioridad asociada a una señal, lo que permite que la transición asociada a un método se establezca al valor deseado. Esta primitiva es equivalente al atributo `Current::the_priority` de RT-CORBA y de esta forma no se requiere ningún cambio para el código generado desde las herramientas de SDL siguiendo el modelo de programación de RT-CORBA.

La Figura 4.8 muestra el proceso SDL asociado a un POA. El cambio del modelo de prioridad utilizado es realizado en la creación del POA con el procedimiento `CreatePriorityModelPolicy`. De la misma forma se pueden crear políticas adicionales CORBA con métodos equivalentes.

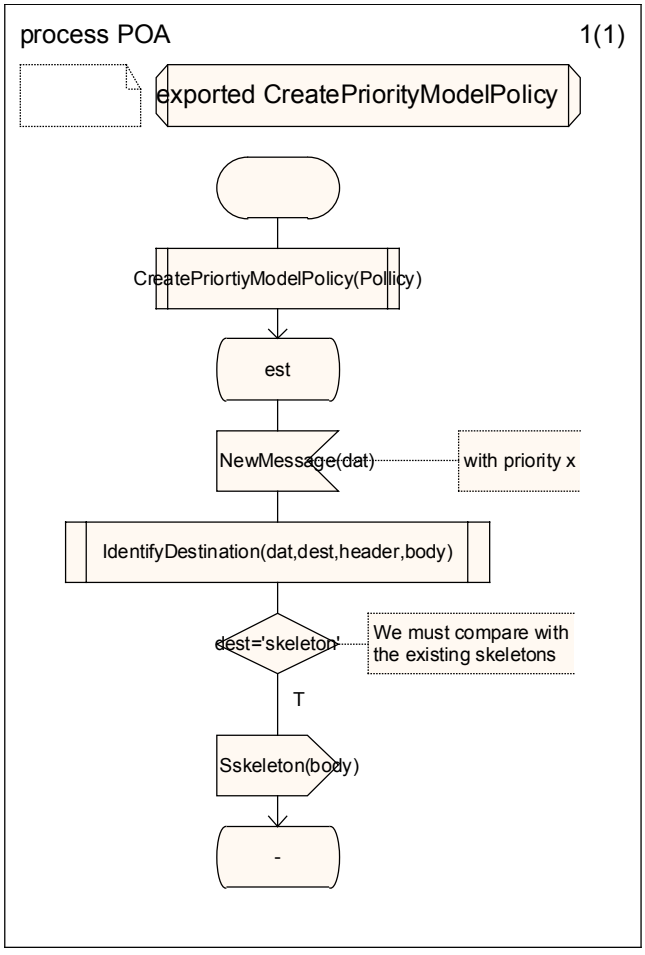


Figura 4.8. Adaptador de objetos

Después de esto, el POA queda esperando por nuevos mensajes (a través de la señal `NewMessage`) desde el bloque GIOP y cuando alguno de estos mensajes llega, tiene que localizar al *skeleton* destino, delegando la solicitud al mismo.

La identificación del objetivo es realizada en el procedimiento `IdentifyDestination`, siendo éste, muy dependiente de la implementación utilizada. De hecho, un factor de calidad de la implementación, puede encontrarse aquí. De esta forma, las buenas implementaciones identifican a los objetos destino en tiempo constante, mientras que otras implementaciones peores pueden requerir tiempos polinomiales.

Tras la invocación del objetivo, se genera una señal para el *skeleton* objetivo. Hay que hacer notar aquí, que el orden de las señales `NewMessage` no es responsabilidad del POA. En su lugar, otros elementos (ej: GIOP o la plataforma de comunicaciones) deben ordenar las invocaciones adecuadamente atendiendo a la prioridad de las solicitudes. El adaptador de objetos lo único que hace es recuperar estas señales ordenadas y transmitirlas a los *skeletons*. Así, en este punto se pueden tener inversiones de prioridad limitadas por la calidad de la implementación RT-CORBA y su capacidad de ordenación de las solicitudes

Finalmente, se envía la respuesta al cliente, habiendo modelado esto con la utilización del proceso *skeleton*. Los *skeletons* (simétricos a los *stubs*) son representados mediante procesos esperando nuevas solicitudes.

La Figura 4.9 muestra un *skeleton* genérico con dos transiciones. La primera transición (`Sskeleton`) está relacionada con nuevas solicitudes sobre un sirviente. Cuando una nueva solicitud es recibida, el *skeleton* tiene que desempaquetar los datos. Para ello, utiliza el procedimiento `Unpacking` (dependiente de la implementación RT-CORBA) obteniendo información tales como el método a ser invocado, los parámetros e información adicional (ej: la prioridad del cliente). El siguiente paso es, precisamente, comprobar el modelo de prioridad usado.

La Figura 4.10 muestra el procedimiento `CheckPriorityModel`. Este procedimiento determina el modelo de prioridad utilizado: s*erver declared* o *client propagated*. Si el modelo deseado es este último, hay que utilizar la primitiva `ThePriority` para establecer la prioridad de la transición. Para esto, hay que utilizar el nombre del método y la prioridad del cliente que ha sido extraída en el procedimiento `ExtractPriority`. Esta primitiva es equivalente al atributo `Current::the_priority`, de forma que se mantiene el modelo de programación de RT-CORBA.

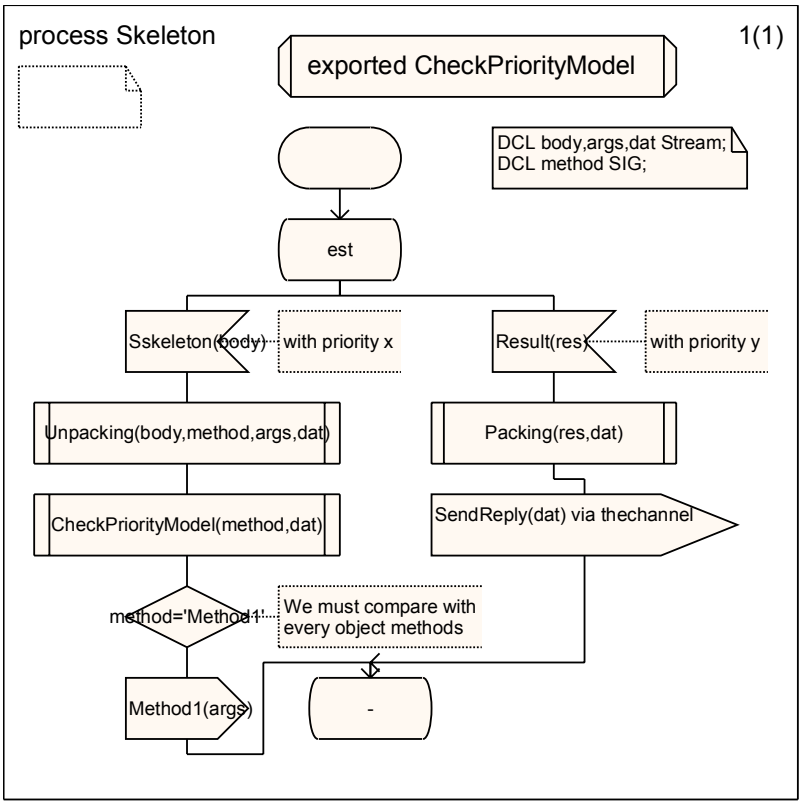


Figura 4.9. Proceso *skeleton*

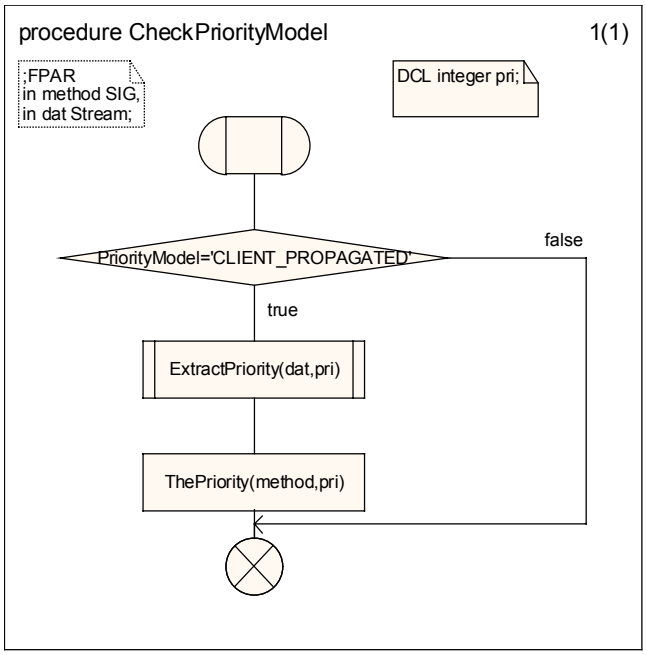


Figura 4.10. Procedimiento `CheckPriorityModel`

En el caso del modelo *server declared*, no se necesitan acciones adicionales, ya que, este modelo está implícito en la extensión de tiempo real de SDL. Para el modelo, el procedimiento `CheckPriorityModel` es la única diferencia entre las dos políticas de

prioridad de RT-CORBA. Después de comprobar el modelo de prioridad, el *skeleton* selecciona el método objetivo invocándolo con los parámetros extraídos. La invocación es realizada a través de una señal con el nombre del método.

La respuesta en el *skeleton* es capturada por la señal `Result`. Esta señal es transmitida por el proceso sirviente después de la ejecución del método. Los pasos siguientes son el empaquetamiento de la respuesta con el procedimiento `Packing` y la generación de una nueva señal para la respuesta: `SendReply`, que es transmitida al bloque `GIOP` y desde el bloque `GIOP` al cliente.

La Figura 4.11 muestra un proceso SDL representando un sirviente CORBA. El proceso sirviente tiene una transición por cada método de la interfaz CORBA. Estas transiciones utilizan la extensión *"with priority"* del modelo, ejecutando las transiciones con la prioridad previamente establecida (*server declared* o *client propagated*). El siguiente y más importante paso es la ejecución del método, donde realmente se ejecuta el servicio.

Para propósitos de análisis, es suficiente con indicar el tiempo del peor caso de ejecución del método aunque es también posible modelar completamente el método. Después de la ejecución del método, los resultados son enviados al *skeleton* generando una señal `Result`.

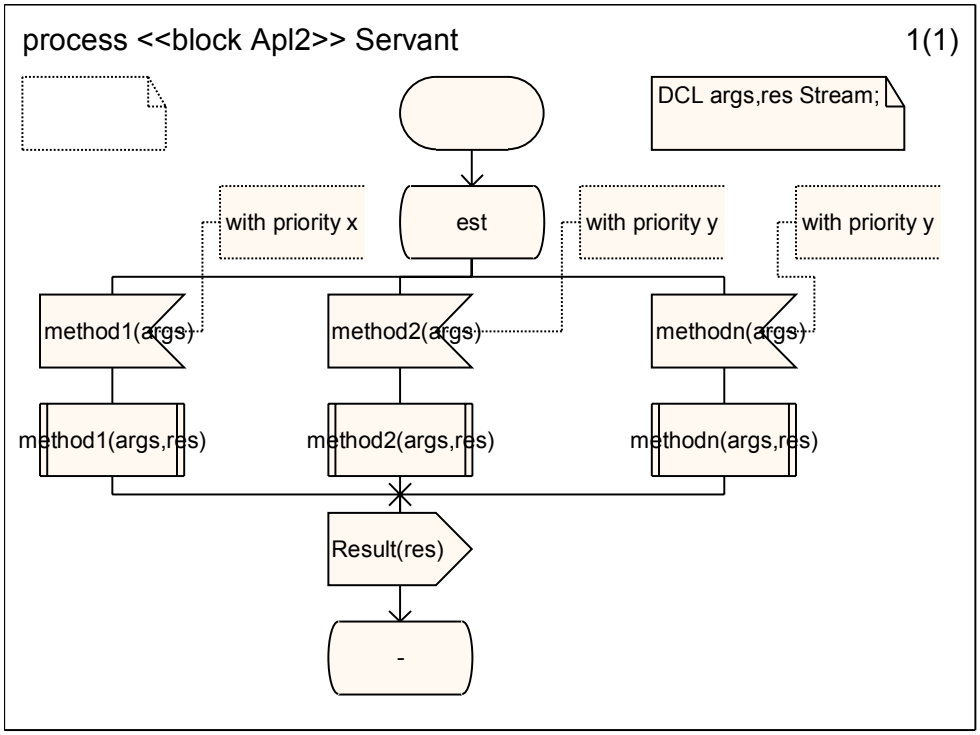


Figura 4.11. Proceso sirviente

4.4. Servidores multihebra

En las secciones previas se describió el diseño en SDL de los modelos de prioridad de RT-CORBA sin tener en cuenta la concurrencia del sistema: múltiples hebras o recursos compartidos. El disponer de múltiples hebras o procesos es una característica necesaria para los sistemas distribuidos de tiempo real. Aún más, es posible evitar algunas fuentes de inversión de prioridad ejecutando solicitudes de prioridad más alta en lugar de tener que esperar por la finalización de solicitudes de prioridad más baja. Por otra parte, la gestión de los recursos compartidos, es también vital en los sistemas de tiempo real.

Los ORBs estándar de CORBA siguen varias estrategias acerca de la multitarea; desde ignorarla a considerarla, pero recayendo sobre el ORB la responsabilidad de crear las hebras, prioridades sobre las hebras, etc. [Pyarali et al, 2003] RT-CORBA va más allá con los *thread pools*, con los que el usuario puede controlar completamente el comportamiento multitarea de la aplicación.

Junto con la multitarea, hay que considerar también la exclusión mutua en el acceso a recursos compartidos. Una vez más, RT-CORBA ofrece *mutexes*, que permiten obtener exclusión mutua en diferentes partes de la aplicación con inversión de prioridad limitada.

En el modelo presentado, la multitarea es realizada con procesos SDL. En la parte del servidor se indican el número máximo de solicitudes simultáneas que un *skeleton* puede tener. La Figura 4.7 mostró los procesos asociados a los *skeletons*. De esta forma, se pueden tener solicitudes simultáneas desde 1 a n. El modelo SDL no requiere nada más para modelar la concurrencia excepto lo descrito en las secciones previas.

La segundo cuestión con la concurrencia está relacionada con los recursos compartidos. RT-CORBA proporciona *mutexes* con la semántica de POSIX, proporcionando tres operaciones: `lock`, `unlock` y `try_lock`. Para ser modelados en SDL, los recursos compartidos serán encapsulados en una clase especial de procesos, tal y como se describió en la sección 4.1. Estos procesos actúan como servidores que sólo utilizan llamadas remotas a procedimiento como mecanismo de comunicación, es decir, siempre están esperando por recepción de RPCs desde otros procesos.

Cada uno de estos procesos tiene un techo de prioridad asignado, que es el máximo de las prioridades de las transiciones de todos los procesos donde se accede al recurso. De esta forma, se evitan posibles inversiones de prioridad y el tiempo de

bloqueo es predecible. La exclusión mutua está también garantizada, ya que, todas las transiciones son ejecutadas a la prioridad más alta entre todos los procesos que acceden al recurso.

4.5. La plataforma de comunicaciones

El resto de características de RT-CORBA están relacionadas con la gestión de la conexión entre el cliente y el servidor: propiedades de protocolos, establecimiento de conexión, conexiones privadas, etc. Estas características no son tenidas en cuenta, ni en el cliente, ni el servidor. En su lugar, son modeladas en el bloque de la plataforma de comunicaciones, dejando incluso al bloque `GIOP`, que representa al protocolo abstracto del mismo nombre, independiente de estos detalles.

Para obtener aplicaciones distribuidas predecibles de tiempo real, también es necesario que la plataforma de comunicaciones sea predecible. En este sentido, se presenta un diseño SDL para el protocolo CAN. Entre otras razones, actualmente hay implementaciones de RT-CORBA con instancias para el protocolo GIOP sobre CAN. Específicamente, ROFES es una implementación de RT-CORBA que incluye el protocolo denominado CANIOP, que utiliza RT-CORBA sobre CAN.

CAN es un bus de comunicaciones tipo *broadcast* donde hay varios procesadores conectados al bus a través de una interfaz. Los datos son transmitidos como mensajes de entre 1 y 8 bytes. A cada mensaje se le asigna un identificador único, representado como un número de 11 bits. Este identificador sirve para dos propósitos: filtrar mensajes en la recepción y asignar prioridades a los mensajes.

El uso de identificador como prioridad es la parte más importante de CAN en relación con tiempo real. El bus CAN actúa como una gran puerta lógica AND, donde cada estación es capaz de ver la salida de la puerta. Este comportamiento es utilizado para resolver colisiones en la utilización del bus. Cada estación espera hasta que el bus esté sin ocupar y comienza a transmitir el mensaje de mayor prioridad que exista en la cola de su procesador. Éste será finalmente transmitido si es el mensaje con mayor prioridad en el sistema, siendo de más prioridad los identificadores con valores más bajos (bit 0 dominante).

Se pueden extraer dos aspectos que deben ser incluidos en la especificación SDL para diseñar un protocolo CAN:

- El bus CAN es un medio compartido para transmitir y el mecanismo de comunicación es a través de *broadcast*.
- La comunicación en el bus CAN es a través de mensajes con prioridad.

El diseño propuesto de CAN en SDL cumple las siguientes características:

- El bus CAN es encapsulado en un proceso, `busCAN`, y el acceso al bus es través de un RPC.
- Como CAN es un recurso global compartido, utiliza el protocolo de techo de prioridad.
- Cuando el mensaje de mayor prioridad es seleccionado, el bus CAN lo envía, y cada procesador analiza si el mensaje debe ser considerado.

En la Figura 4.12 se muestra el bloque con el proceso especificando el protocolo CAN y las señales que debe manejar.

En la Figura 4.13, el proceso `CANbus` está esperando por llamadas remotas del procedimiento `Send`, siendo ejecutado con un techo de prioridad para garantizar que podrá enviar el mensaje sin interrupciones.

El procedimiento `Send` debe construir el mensaje con el formato CAN adecuado, y enviarlo a los procesadores (Figura 4.14).

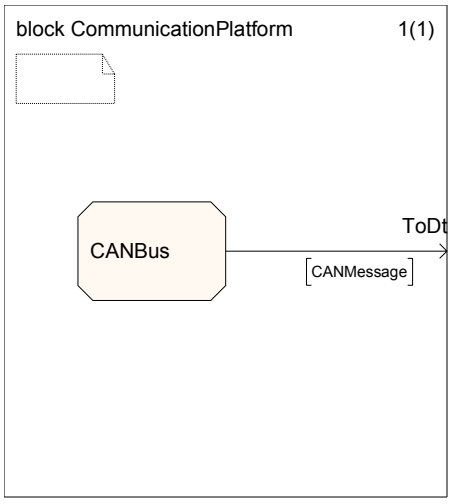


Figura 4.12. Bloque para el protocolo CAN

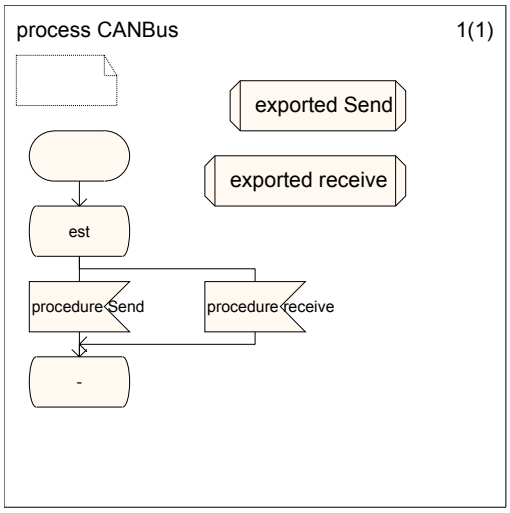


Figura 4.13. Proceso CANBus

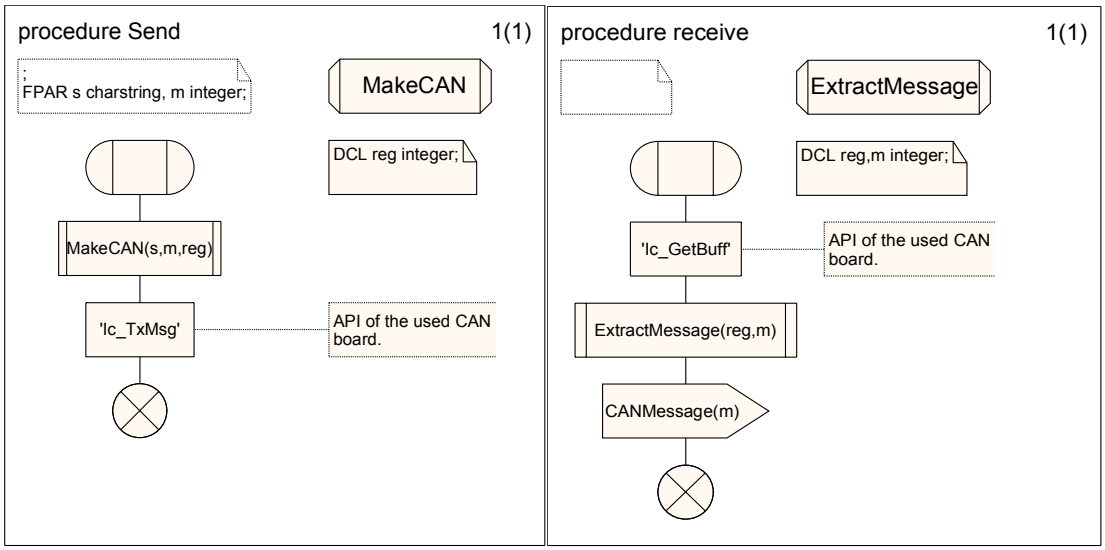


Figura 4.14. Procedimientos de envío y recepción

4.6. Análisis temporal

Los métodos de análisis permiten tener un conocimiento a priori sobre el cumplimiento de los requisitos temporales de los sistemas de tiempo real. La predecibilidad de los modelos SDL propuestos permite integrar estos métodos en el diseño de tales sistemas. En este sentido, se puede considerar como una fase adicional en el diseño que ayudará a los diseñadores a estudiar si las aplicaciones cumplen los requisitos temporales.

El análisis propuesto está basado en el trabajo de Joseph y Pandya, pero tiene en cuenta las características del modelo de ejecución de tiempo real descrito en [Alvarez et al., 1999] y [Llopis, 2002], añadiendo la sobrecarga de RT-CORBA en el cálculo de los peores tiempos de ejecución de los eventos.

En los sistemas distribuidos, la respuesta a un evento puede incluir transiciones pertenecientes a diferentes procesadores. En este sentido, consideramos los eventos como una composición de eventos en diferentes procesadores. Distinguimos dos tipos de eventos, según sus transiciones se ejecuten todas en el mismo procesador o en distintos procesadores. Así pues, a los eventos que se ejecutan en un único procesador los denominamos *eventos locales*, denominando *eventos distribuidos* a aquellos que incluyen transiciones en diferentes procesadores. Los eventos locales están contemplados en el modelo presentado en [Llopis, 2002]. En este trabajo, se amplía el análisis para contemplar los eventos distribuidos.

SDL tiene además algunas características semánticas que influyen en el tiempo de respuesta:

- Existe una relación de precedencia entre las transiciones que responden a un evento. Si se toma esto en cuenta, se puede reducir el número de transiciones que participan en la interferencia para el caso de eventos locales.
- En el caso de eventos distribuidos será necesario tener en cuenta el procesador donde se ejecutan las transiciones, puesto que las interferencias a incluir son diferentes con respecto al caso anterior.
- La semántica de ejecución de SDL influye en el tiempo de respuesta, generando una nueva fuente de bloqueo entre diferentes transiciones en el sistema.

El análisis presentado se basa en el modelo plano SDL de una aplicación basada en componentes, y que es complementada con el modelo SDL de las comunicaciones, y tiene las siguientes particularidades:

- La asignación de prioridades viene determinada por las herramientas del entorno, las cuales analizan las relaciones de precedencia existentes y los requisitos temporales, asignando prioridades.
- El despliegue de los componentes es realizado a través de las herramientas del entorno, considerando los tiempos de ejecución en la plataforma final.
- El plazo de ejecución de los eventos y sus tareas es menor que el periodo.

4.6.1. Notación

El sistema a ser diseñado debe responder a un conjunto de eventos. Denotamos *Ev* como el conjunto de eventos (internos y externos) a los que el sistema responde.

$Ev = \{E_1, ..., E_n\}$

La respuesta a un evento está compuesta por una secuencia de señales que activan sus transiciones. De esta forma, la función σ denota la secuencia de transiciones que responden a un evento:

$\sigma : Ev \rightarrow Trn^*$

donde *Trn* es el conjunto de transiciones en el sistema.

Cada transición $t \in Trn$ tiene la función asociada $\varepsilon(t)$ que retorna los estados de procesos SDL donde *t* puede ejecutarse:

$\varepsilon : Trn \rightarrow Est$

Finalmente, cada transición tiene una prioridad ϕ

$\phi : Trn \rightarrow N$

Se representarán secuencias de transiciones utilizando los símbolos < y >. Por ejemplo, la secuencia $<t_1,t_2>$ indica que la transición t_1 se ejecuta y, cuando finaliza, activa a la transición t_2, que comienza su ejecución.

4.6.2. Relaciones de precedencia

Existen relaciones de precedencia entre las diferentes transiciones que forman parte de la respuesta a un evento. Sin embargo, hay que tener en cuenta que puede haber diferencias en el tiempo de respuesta si el evento incluye transiciones remotas. En el resto de esta sección se explica de qué modo afectan al análisis ambos tipos de relaciones de precedencia.

Relaciones de precedencia en eventos locales

Supongamos un evento externo E_i compuesto por la secuencia de transiciones $<t_{i1}, ..., t_{in}>$ que responden a este evento. Si queremos calcular la interferencia de las tareas del evento E_i sobre la transición t_{ab} que pertenece al evento E_a, debemos tomar en cuenta todas las transiciones que pertenecen al evento E_i con mayor o igual prioridad que la prioridad de la transición t_{ab}.

Sin embargo, las transiciones en el evento E_i, tienen un orden en la ejecución debido a las relaciones de precedencia existentes entre ellas. Cuando la ejecución de la transición t_{i1} finaliza, la transición t_{i2} comienza y así sucesivamente con el resto de transiciones en el evento. Supongamos que existe una transición, t_{ij}, en el evento E_i que tiene una prioridad más baja que la tarea t_{ab}. En esta situación, t_{ab} puede ser solamente interrumpido por la secuencia $t_{i1}, ..., t_{ij-1}$ o por la secuencia $t_{ij+1}, ..., t_{in}$ pero nunca por t_{ij}. Habrá que analizar la secuencia de transiciones que da el peor tiempo de interferencia.

Considérese la situación mostrada en la Figura 4.15 donde se quiere calcular la interferencia del evento E_i con respecto a la transición t_{ab}.

La altura de las cajas indica las prioridades de las transiciones y la línea horizontal indica la prioridad de la transición a analizar (t_{ab}). Hay dos secuencias de transiciones que pueden potencialmente interrumpir a t_{ab}, pero como mucho sólo una de ellas será capaz de interrumpirla en un momento dado, pudiendo mejorarse el análisis realizado. Por lo tanto, se incluirá la secuencia con el peor tiempo de ejecución en la interferencia de la transición t_{ab}.

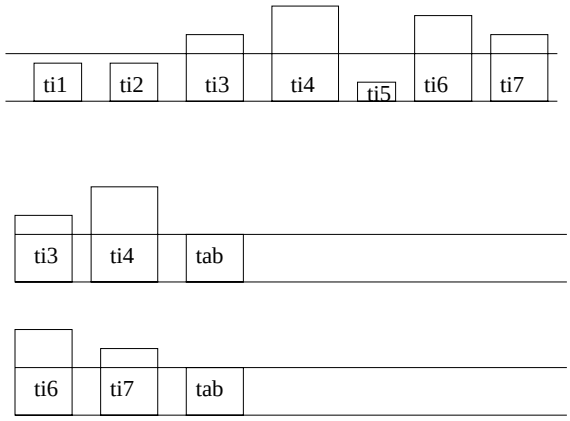


Figura 4.15. Interferencia para la tarea t_{ab}

Relaciones de precedencia en eventos distribuidos

En el caso de relaciones de precedencia entre transiciones incluyendo la ejecución remota de alguna de ellas, no pueden mantenerse las mismas suposiciones.

Supóngase la misma secuencia de transiciones $<t_{i1}, \ldots, t_{in}>$ que responden al evento E_i. Nuevamente, se quiere calcular la interferencia de las tareas del evento E_i sobre la transición t_{ab} que pertenece al evento E_a, y en concreto se quiere tener en cuenta la transición t_{ij} con una prioridad menor que $t_{ab,}$ pero, con la diferencia de que en esta ocasión la transición t_{ij} se ejecuta de manera remota. En el caso anterior, t_{ab} solamente podía ser interrumpida por la secuencia $t_{i1}, \ldots, t_{ij-1}$ o por la secuencia $t_{ij+1}, \ldots, t_{in}$. Sin embargo, en esta ocasión, al poder ejecutarse t_{ij} de forma remota, no va a verse interrumpida por la transición t_{ab}. De esta forma, t_{ij} puede concluir su ejecución en la máquina remota, continuando su ejecución la secuencia $t_{ij+1}, \ldots, t_{in}$. En este caso, si t_{ij} acaba antes que t_{ab}, la secuencia $t_{ij+1}, \ldots, t_{in}$ todavía puede interrumpir a t_{ab}, por lo que

hay que considerar la interferencia de ambas secuencias, t_{i1}, ..., t_{ij-1} y t_{ij+1}, ..., t_{in} en el cálculo del tiempo de respuesta.

Considérese nuevamente la situación mostrada en la Figura 4.15 donde se quiere calcular la interferencia del evento E_i con respecto a la transición t_{ab}, donde t_{i5} se ejecuta de forma remota. En este caso las dos secuencias de transiciones pueden potencialmente interrumpir a t_{ab} , por lo que habrá que incluir a ambas secuencias en la interferencia de la transición t_{ab}.

4.6.3. Fuentes de bloqueo

La semántica de SDL no permite que dos transiciones pertenecientes al mismo proceso SDL sean ejecutadas de forma concurrente.

Supongamos que hay dos transiciones, t_{ij} y t_{kj}, que pertenecen al mismo proceso SDL pero que toman parte en diferentes respuestas a eventos externos, E_i y E_k, siendo la prioridad de t_{ij} mayor que la de t_{kj}. Si se está calculando el tiempo de respuesta de la transición t_{kj}, la transición t_{ij} será incluida en la expresión de la interferencia. Sin embargo, t_{ij} nunca será capaz de interrumpir la ejecución de t_{kj} y no debería ser considerada. Este hecho, reduce el número de transiciones que pueden interrumpir a otras transiciones, pero como consecuencia, hay que considerar un factor adicional de bloqueo denominado *bloqueo hasta finalización*.

En el caso de transiciones ejecutándose en distintos procesadores, hay que considerar una nueva fuente de bloqueo adicional, que no aparecía en los sistemas monoprocesador considerados en [Llopis, 2002], denominada *bloqueo remoto*. El bloqueo remoto se produce al realizar llamadas síncronas entre componentes u objetos CORBA, de forma que el *llamante* queda bloqueado hasta la ejecución del servicio en el *llamado*.

Bloqueo hasta finalización

En la Figura 4.16 hay un evento externo E_i y la secuencia de transiciones $<t_{i1}, t_{i2}, \ldots, t_{i5}>$ que responden a este evento, y se quiere calcular la interferencia para t_{ab}. Como puede verse en la figura, cada transición tiene su prioridad fija.

En la Figura 4.17 puede verse la relación entre las prioridades de la transición del evento E_i y la transición t_{ab}. La línea horizontal discontinua indica la prioridad de t_{ab}.

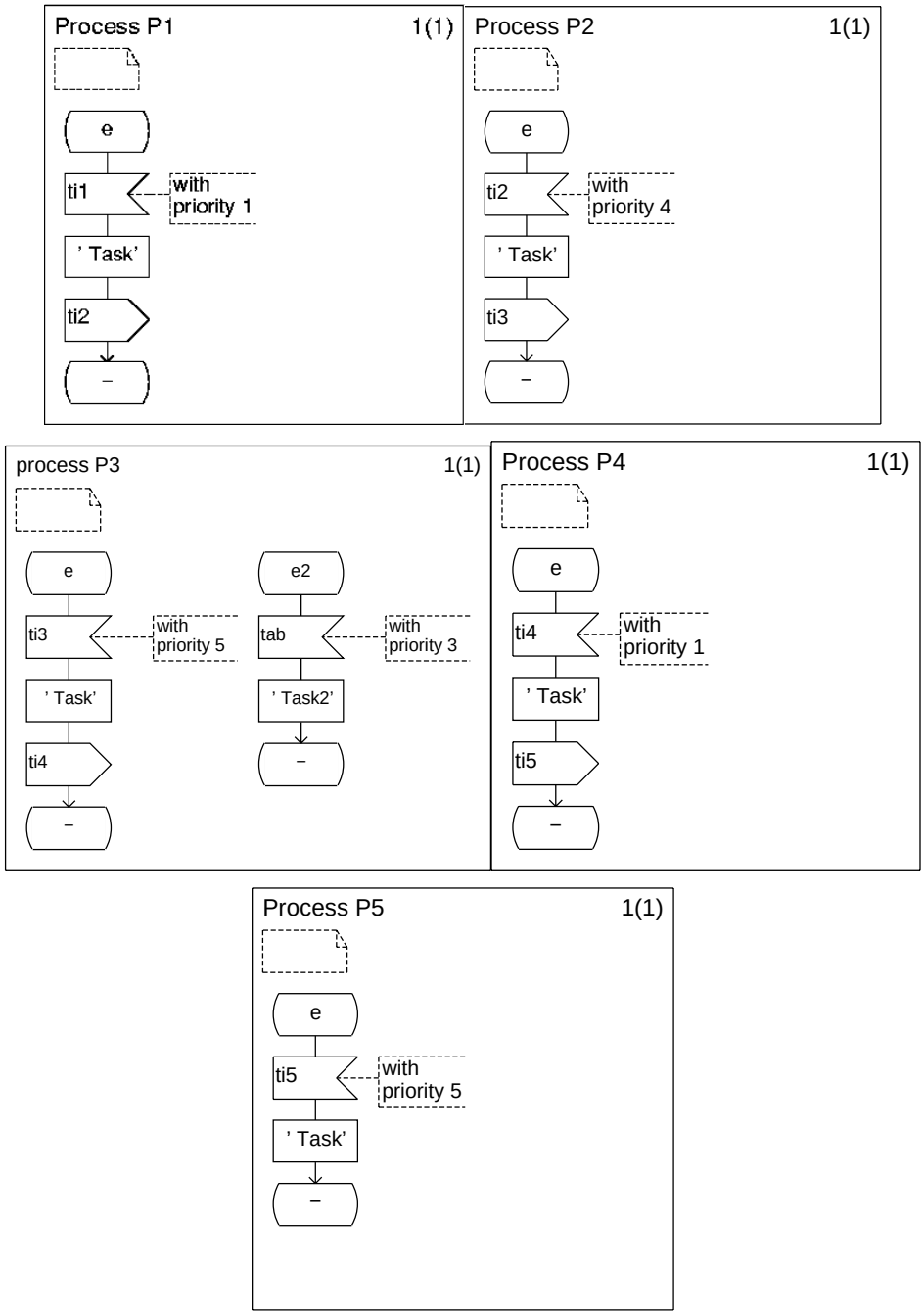


Figura 4.16. Calculando la interferencia a t_{ab} en SDL

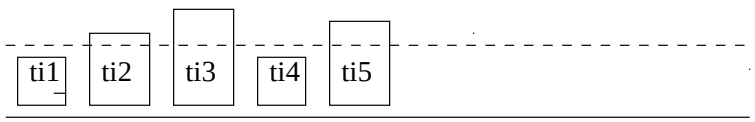


Figura 4.17. Relación entre la prioridad de t_{ab} y las tareas en E_i

Las transiciones t_{i3} y t_{ab} pertenecen al mismo proceso SDL y de esta forma, t_{i3} no puede interrumpir a t_{ab}. Si se toman en cuenta las relaciones de precedencia, entonces hay que seleccionar el peor tiempo de ejecución entre las secuencias $\{t_{i2}, t_{i3}\}$ y $\{t_{i5}\}$ suponiendo que se ejecutan de forma local. Sin embargo, si se toma en cuenta la semántica de SDL se seleccionará entre las secuencias t_{i2} y t_{i5} porque t_{i3} no puede interrumpir a t_{ab}, ya que comparten el mismo proceso SDL.

Aunque las situaciones previas pueden reducir el número de transiciones que pueden interrumpir a otras transiciones en sistemas monoprocesador, el tiempo de bloqueo puede ser incrementado sumando el bloqueo hasta fin de ejecución.

Supóngase que se está calculando el tiempo de bloqueo de la transición t_{ab} y, como puede verse en la Figura 4.18, t_{i1} y t_{ab} pertenecen al mismo proceso SDL y la prioridad de t_{i1} es más baja que la prioridad de t_{ab}. Inicialmente, t_{i1} no toma parte en el tiempo de respuesta de t_{ab} pero si se integra este tipo de análisis en SDL, el tiempo de ejecución de t_{i1} tiene que ser considerado como bloqueo hasta el final de ejecución para la transición t_{ab}.

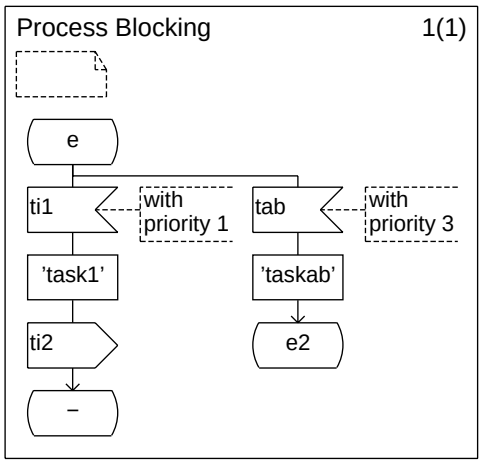


Figura 4.18. Calculando el tiempo de bloqueo de t_{ab}

Bloqueo por interacción con componentes

Este nuevo tipo de bloqueo se produce por la invocación de un servicio de un componente (local o remoto), esto es, la realización de un `call`. Como se ha visto en secciones anteriores (3.1.7. y 4.2.) una invocación mediante un `call` queda finalmente transformada en una llamada CORBA que da lugar finalmente a una nueva transición SDL, que además puede estar localizada en otro procesador y que requiere, la

consideración de transiciones adicionales relacionadas con la utilización del middleware.

Estas nuevas transiciones generan nuevas relaciones de precedencia que deben ser consideradas. La Figura 4.19 muestra las diferentes transiciones implicadas en una llamada a un interfaz de un componente. En concreto, hay que considerar el papel del POA, *Skeleton* y la plataforma de comunicaciones.

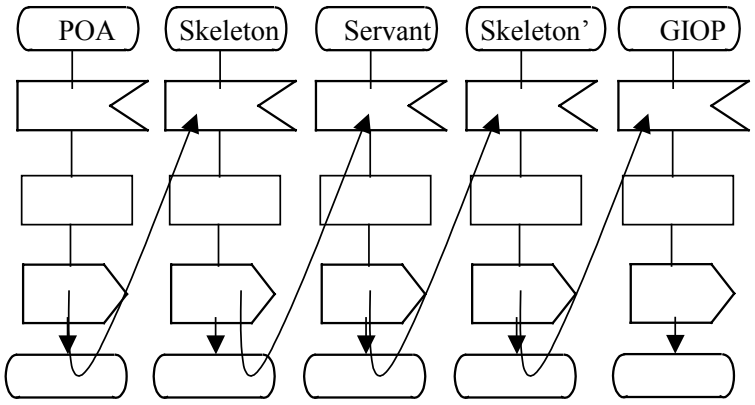


Figura 4.19. Relación de precedencia en un `call`

El componente invocado puede encontrarse localizado en el mismo procesador o en un procesador remoto. Desde el punto de vista del análisis del tiempo de respuesta, estas dos situaciones presentan problemáticas, en principio, distintas, aunque se ha proporcionado un esquema de solución lo más homogéneo posible.

En el caso de que el componente se encuentre en otro procesador, se estará tratando con un evento distribuido. Para el cálculo del tiempo de respuesta de estos eventos, hay que añadir, por una parte, la sobrecarga de la propia plataforma física de comunicaciones y, por otra, el tiempo de respuesta de las transiciones en un procesador remoto. Durante el intervalo de realización del `call`, la CPU del componente *llamante*, queda libre para su utilización por otras transiciones. En secciones posteriores, se detallará cómo la división de una transición remota, permite analizar esta liberación de la CPU.

En el caso de las invocaciones locales hay que seguir manteniendo las transiciones descritas en la Figura 4.19, puesto que las implementaciones estándares de RT-CORBA realizan el mismo tratamiento, tanto si el objeto destino de la invocación está en el mismo procesador, como si está en otro distinto. La distinción en este caso, queda oculta por el modelo de implementación de la plataforma física. Por ejemplo, si la

invocación es local, el modelo de red simplemente reenviaría la señal al proceso local adecuado, mientras que si se trata de una llamada remota, se llevarían a cabo el resto de los procesos de tratamiento correspondientes al protocolo de red implementado (protocolo de transporte, empaquetamiento, control de acceso al medio, etc.).

4.6.4. Cálculo del tiempo de respuesta para eventos

En esta sección se definen nuevos términos y se detalla el cálculo del tiempo de respuesta.

La función π_{sdl} devuelve el proceso SDL al que una transición pertenece.

$$\pi_{sdl} = Trn \rightarrow \Pr c$$

Dado un evento E, $\sigma_{pq,F}(E)$ indica la subsecuencia de transiciones que pertenecen a E y que se ejecutan en el procesador F que comienzan en la transición *p-ésima* y finalizan en la transición *q-ésima*, esto es, si $\sigma(E) =< t_1,...,t_m >$ entonces:

$$\sigma_{pq,F}(E) =< t_p ...,t_q > \quad donde \quad 1 \le p \le q \le m$$

Dada una transición t' que pertenece a un evento E', el conjunto de secuencias de transiciones con mayor prioridad que t', $HP_{t',F}(E)$, es indicado como:

$$HP_{t',F}(E) = \left\{ \begin{matrix} \sigma_{pq,F}(E) : \forall t \in \sigma_{pq,F}(E) \ \ \phi(t') < \phi(t) \wedge \pi_{sdl}(t) \neq \pi_{sdl}(t') \vee (\pi_{sdl}(t) = \pi_{sdl}(t') \wedge \varepsilon(t) = \varepsilon(t')) \\ 1 \le p \le q \le m \end{matrix} \right\}$$

Este conjunto incluye las secuencias de transiciones (ejecutadas en el procesador F) que responden a un evento E que tiene mayor prioridad que la transición t'. Sin embargo, estas transiciones de prioridad más alta deben cumplir una de estas condiciones:

- Los procesos SDL de estas transiciones no pueden coincidir con el proceso SDL de la transición t'.
- Si pertenecen al mismo proceso, deben ejecutarse en el mismo estado.

Al incluir este conjunto a secuencias que están incluidas en secuencias más largas, se define $HP_{t',F}(E)$ para seleccionar la secuencia más larga que pertenece a $HP_{t',F}(E)$ y comienza en p:

$$HP^{P}_{t',F}(E) = \begin{cases} \sigma_{pq,F}(E) & si\ \sigma_{pq,F}(E) \in HP_{t',F}(E) \wedge \sigma_{p-1q,F}(E) \notin HP_{t',F}(E) \wedge \sigma_{pq+1,F}(E) \notin HP_{t',F}(E) \\ <> & enotrocaso \end{cases}$$

donde <> representa la secuencia vacía.

Para calcular el tiempo de interferencia del peor caso de una transición *t'* ejecutada en el procesador *F* que pertenece al evento *E'* hay que realizar los siguientes pasos:

- Seleccionar las secuencias de transiciones que pertenecen a $HP_{t',F}(E)$ para cada evento *E* en el sistema. Varias secuencias pueden aparecer en cada evento.
- Para cada evento *E*, se considera su secuencia más larga $HP^{p}_{t',F}(E)$.

Estas ideas pueden formalizarse en la siguiente expresión:

$$I(t') = \sum_{i=1}^{n} \sum_{j=1}^{m_i} I(t', HP^{j}_{t',F}(E_i))$$

Para aquellos eventos que participan en la interferencia de la transición analizada y que no incluyan transiciones remotas (eventos locales), se puede seguir aplicando las fórmulas presentadas en [Llopis, 2002], y en concreto la expresión de la interferencia quedaría como:

$$I(t') = \sum_{i=1}^{n} I(t', HP^{1}_{t'}(E_i)) + \sum_{i=1}^{n} \max_{j=2..m_i} \{\omega(HP^{j}_{t'}(E_i))\}$$

La ventaja de utilizar esta segunda fórmula es lograr cotas más bajas en los tiempos de respuesta.

Finalmente, la interferencia de una secuencia de transición es denotada como:

$$I(t', \sigma_{pq,F}(E)) = \sum_{t \in \sigma_{pq,F}(E)} \left\lceil \frac{R_{t'}}{T_t} \right\rceil * C_t$$

Para calcular el tiempo de bloqueo en el análisis de planificabilidad hay que considerar tres posibles fuentes de bloqueo:

La primera fuente de bloqueo está relacionada con el acceso a recursos compartidos. Estos recursos son encapsulados en procesos pasivos accedidos a través de llamadas remotas a procedimiento (RPC). Usando el protocolo de techo de prioridad, este tiempo de bloqueo, denominado B_{sh}, está acotado.

La segunda posible fuente es debida al bloqueo hasta fin de ejecución, B_{rtc}. La expresión de B_{rtc} para la transición t_{ab} es la siguiente:

$$B_{rtc}(t) = \max\{C_{t'} : \phi(t') \leq \phi(t) \wedge \pi_{sdl}(t') = \pi_{sdl}(t) \wedge \varepsilon(t') = \varepsilon(t) \wedge t' \notin \sigma(E)\}$$

La expresión del tiempo de respuesta de la transición t, R_t, que participa en la respuesta de un evento externo E es la siguiente:

$$R_t = C_t + WI(t) + \max\{B_{sh}(t), B_{rtc}(t)\}$$

Dado un evento E, el tiempo de respuesta del peor caso puede ser definido como:

$$R_E = \sum_{t \in \sigma(E)} R_t$$

La tercera y última fuente de bloqueo viene dada por las invocaciones remotas, teniendo que añadir la sobrecarga relativa a las comunicaciones y a las nuevas transiciones relacionadas con RT-CORBA, que serán analizadas en la siguiente sección.

4.6.5. Cálculo del tiempo de respuesta para invocaciones remotas

En esta sección se integra en las ecuaciones de análisis la sobrecarga de RT-CORBA (incluyendo la plataforma de comunicaciones) para el caso de alguna transición que realice invocaciones CORBA.

Sea un evento E, que activa una transición interna en la aplicación *Apl1*. Para calcular el tiempo de respuesta de este evento, hay que considerar el tiempo de respuesta de la secuencia de transiciones $\sigma(E)$. Supongamos que *Apl1* tiene una transición t_{remote}, que invoca a un método remoto *Method1*.

La Figura 4.20 muestra un ejemplo de esta situación con los elementos implicados.

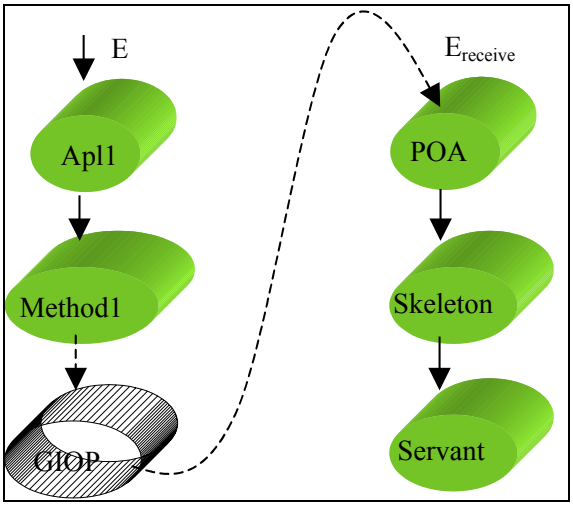


Figura 4.20. Cadena de eventos en una llamada

Para calcular R_E, hay conocer el tiempo de respuesta de t_{remote}. Para hacer esto, se puede considerar que t_{remote} está compuesto de tres partes. La primera parte incluiría el envío de la invocación remota al espacio de direcciones del servidor. La segunda parte ocurre en el servidor, donde se realiza la invocación. Finalmente, la tercera parte incluye la respuesta a la invocación del método. Estas tres partes ocurren en diferentes espacios de direcciones, y, además, existe una relación de precedencia entre ellas. Para facilitar el análisis, estas tres partes se pueden considerar como eventos locales y de esta forma, se puede utilizar la metodología de análisis descrita en la sección 4.6.4.

De esta forma, se pueden considerar tres eventos con la relación de precedencia $<E_{send}, E_{receive}, E_{response}>$, siendo el tiempo de respuesta de t_{remote}:

$$R_{t_{remote}} = \sum_{t \in \sigma_{E_{send}}} R_t + \sum_{t \in \sigma_{E_{receive}}} R_t + \sum_{t \in \sigma_{E_{response}}} R_t$$

El primer evento, E_{send}, incluye todas las transiciones desde la activación de t_{remote} hasta el envío de la ejecución a través de GIOP. Para invocar a *Method1*, t_{remote} usará al *stub* representando al objeto remoto. El *stub*, por su parte, activa una nueva transición en el bloque de comunicaciones, `GIOP`. La siguiente parte a considerar es el tiempo de respuesta de las comunicaciones, que puede ser analizado como se muestra en [Tindell et al., 1995]. De esta forma, se obtiene la secuencia $\sigma(E_{send}) = <t_{Stub}, t_{GIOP}>$.

El segundo evento, $E_{receive}$, es activado en un espacio de direcciones diferente, el servidor. Nuevamente existe una cadena de transiciones, activando el evento una primera transición en el adaptador de objetos. Después de esto, el adaptador de objetos crea una transición en el proceso `Skeleton`, que finalmente invocará el método en el sirviente, activando una nueva transición.

También hay que tener en cuenta las acciones de la respuesta realizadas en el espacio de direcciones del servidor, activándose transiciones adicionales. Cuando el sirviente envía la respuesta, esta es capturada por el *skeleton*, que a su vez, la transmite al bloque `GIOP`. El bloque `GIOP` es responsable de la transmisión al espacio de direcciones del cliente. La respuesta es finalmente recibida por el cliente a través de una señal `AnswerMethod`. De esta forma, se obtiene la secuencia $\sigma(E_{receive}) = <t_{Ereceive}, t_{POA}, t_{Skeleton}, t_{Servant}, t_{Skeleton'}, t_{GIOP'} >$.

En el último evento, hay que considerar la respuesta en el espacio de direcciones del cliente, a través del evento $E_{response}$, con la secuencia de transiciones $\sigma(E_{response}) = <t_{Eresponse}, t_{Stub'}, t_{Apl1'}>$.

Desde este punto de vista, hay tres eventos diferentes en dos procesadores diferentes y de esta forma, se pueden aplicar las técnicas de análisis ya descritas para cada evento, ya que no hay interferencias entre los mismos gracias a la relación de precedencia. Tomando en cuenta esta relación, se puede calcular la sobrecarga de RT-CORBA para t_{remote}, con la suma del peor tiempo de respuesta en el cliente y en el servidor, incluyendo también la sobrecarga de las comunicaciones.

La primera parte de esta suma incluye las transiciones desde la parte del cliente al servidor. La Figura 4.21 muestra estas transiciones.

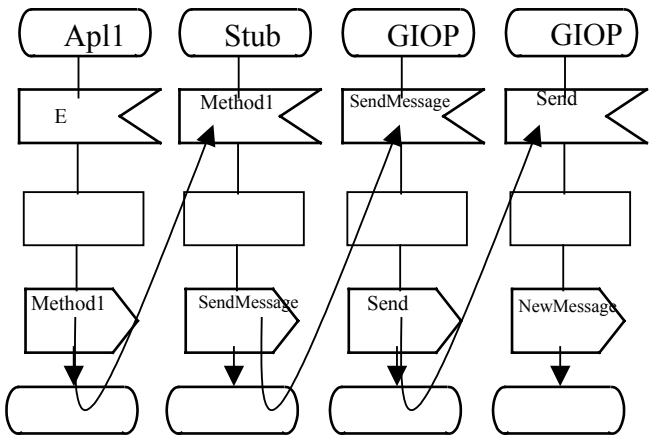


Figura 4.21. Transiciones desde el cliente al servidor

El segundo término de esta ecuación incluye las transiciones en la parte del servidor y el envío de la respuesta al cliente.

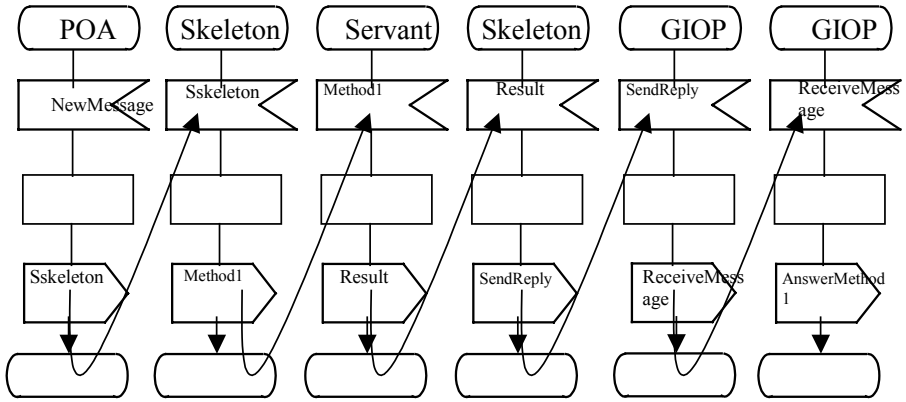


Figura 4.22. Transiciones en la parte del servidor

Finalmente, el tercer término incluye las transiciones en la parte del cliente asociadas a la respuesta.

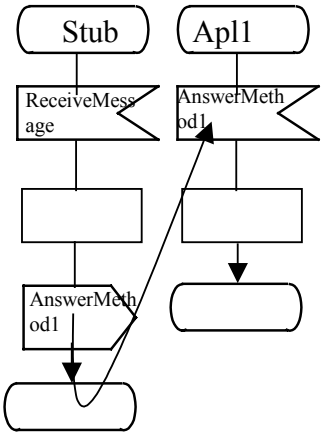


Figura 4.23. Transiciones en la parte del cliente durante la respuesta

En conclusión, la sobrecarga RT-CORBA en la transición t_{remote} es:

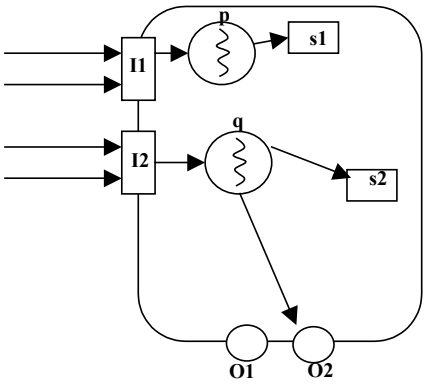
$$R_{tremote} = R_{Stub} + R_{GIOP} + R_{GIOP'} +$$
$$R_{POA} + R_{Skeleton} + R_{Servant} + R_{Skeleton'} + R_{GIOP''} + R_{Stub'} + R_{Apl'}$$

En la fórmula anterior se recogen todas las transiciones que intervienen en el proceso de una invocación remota.

4.7. Análisis de una aplicación basada en componentes

La realización del análisis de una aplicación tiene varias fases que comienzan prácticamente junto con la creación de los componentes. La primera fase es la anotación del código fuente de los componentes para facilitar la labor de las herramientas. Posteriormente, se realiza la obtención de los metamodelos SDL teniendo en cuenta los grafos de ejecución de los componentes y, finalmente, se realizan las pruebas para la obtención de los WCET sobre la plataforma final de despliegue de los componentes. Tras la realización de estas fases en los diferentes componentes que constituyen una aplicación, puede utilizarse la herramienta de análisis que el entorno proporciona, determinando si una aplicación es o no planificable.

La Figura 4.24 muestra un componente genérico de nombre `micomponente` con dos componentes activos (`p` y `q`) que utilizan a su vez a dos componentes pasivos (`s1` y `s2`). El componente dispone además de dos interfaces de entrada y dos interfaces de salida que son tratadas y utilizadas respectivamente en los componentes activos.



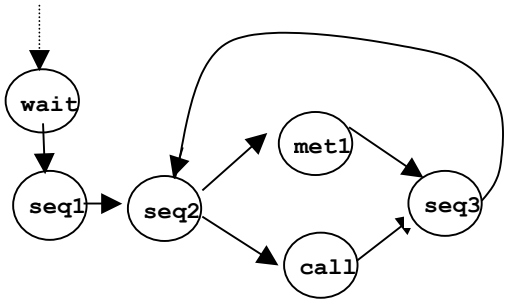

```
Component micomponente {
…
  interface I2 {
    void op1();
    void op2();
  }
…
  interface O2 {
    void op1();
    void op2();
  }
…
  input  I1,I2;
  output O1 with o1;
  output O2 with o2;
}

Component implementation micomponente
{
  Active Tipo_activo { …
  }
  Passive Tipo_pasivo {
      void met1() {… }
  }
  Tipo_activo p,q;
  Tipo_pasivo s1,s2;
}
```

Figura 4.24. Definición de componente genérico para análisis

La primera tarea a realizar por el desarrollador es la anotación del código fuente. Este proceso debe realizarse en todos los componentes activos y pasivos de los componentes. La Figura 4.25 muestra el código anotado del tipo de componente activo

`Tipo_activo` y su grafo de ejecución. Las anotaciones se han realizado sobre el bucle `while` y la sentencia `if`, quedando delimitados tres bloques secuenciales sobre los que se realizarán las medidas de tiempo teniendo.

```
Active Tipo_activo {
…
void execute(){
  wait I2.op1(…) {
  seq_block1;
  [WCET LOOP 10]
  while (cond) {
  seq_block2;
    if (cond) [WCET-IF]
      s1.met1(…);
    else
      call o2.op2(…);
    seq_block3;
    }
  }
}
```



Rutas de ejecución

`wait.seq1.seq2.call.seq3`
`wait.seq1.seq2.met1.seq3`

Figura 4.25. Componente activo y grafo de ejecución

Después de esta fase, se puede ya realizar la obtención del metamodelo SDL siguiendo las reglas de la sección 4.1. Así, por ejemplo, el modelo SDL del componente `Tipo_activo` puede verse en la Figura 4.26.

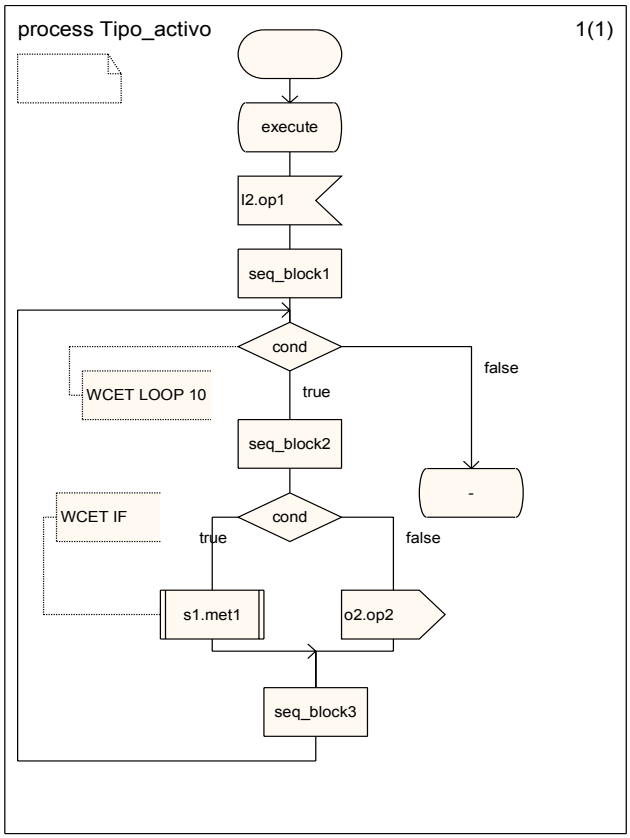


Figura 4.26. Modelo SDL del componente `Tipo_activo`

Este proceso de obtención del modelo SDL debe realizarse para todos los componentes que forman una aplicación, de forma que al final puede obtenerse un modelo SDL plano de todo el sistema. En este modelo final habrá también que insertar el modelado de la infraestructura de comunicaciones.

La Figura 4.27 muestra una visión plana SDL de un componente aplicación donde se utilizan instancias del tipo de componente genérico `micomponente` conectadas a otros dos componentes de tipos `componenteA` y `componenteB`, de tal forma que la instancia de `componenteA` utiliza el servicio `op1` de la interfaz `I2` de `micomponente`, y en el componente activo `Tipo_activo` se utiliza la interfaz de salida con métodos ofertados por la instancia de `componenteB`.

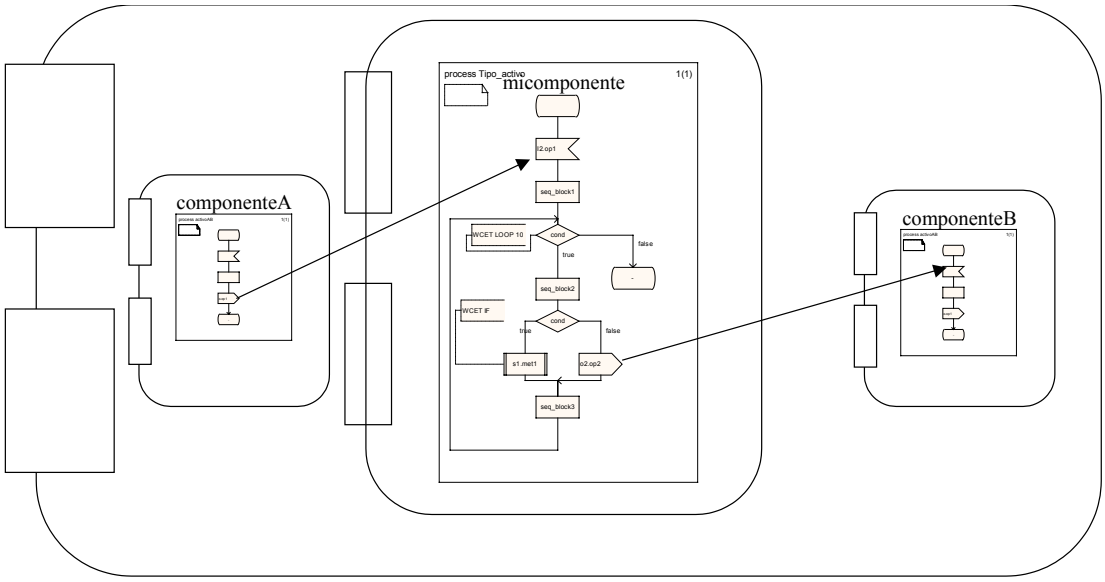


Figura 4.27. Visión plana del modelo SDL de una aplicación

En la visión plana, falta pues, añadir el modelado de la infraestructura de comunicaciones, que en el modelo UM-RTCOM está basada en RT-CORBA. Para ello, se utilizan las técnicas descritas en este capítulo. Así, por ejemplo, en la Figura 4.28 y la Figura 4.29 se muestra cómo quedaría el modelo SDL en el ejemplo descrito, si suponemos que la aplicación se despliega en un nodo de la red junto con `componenteA`, y los componentes `micomponente`, y `componenteB` son desplegados en nodos diferentes.

En la Figura 4.28, los modelos SDL de los componentes se han encapsulado en un bloque SDL para cada componente, incluyendo además un bloque SDL para la

propia aplicación. Existe además, un bloque denominado RTCORBA donde se encapsulan todos los elementos relativos a las comunicaciones. El contenido de este bloque puede verse en la Figura 4.29; incluyendo los *stubs* relativos a las interfaces de los componentes, servidores para el tratamiento de las peticiones y, finalmente, un bloque donde se encapsula la plataforma de comunicaciones.

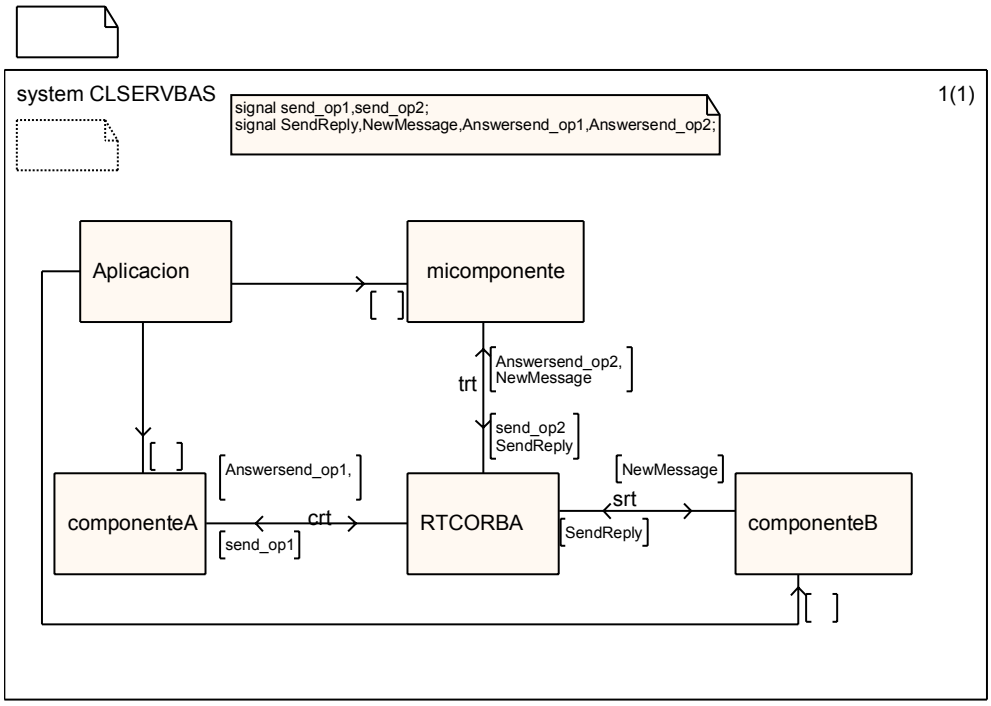


Figura 4.28. Visión plana del modelo SDL de una aplicación

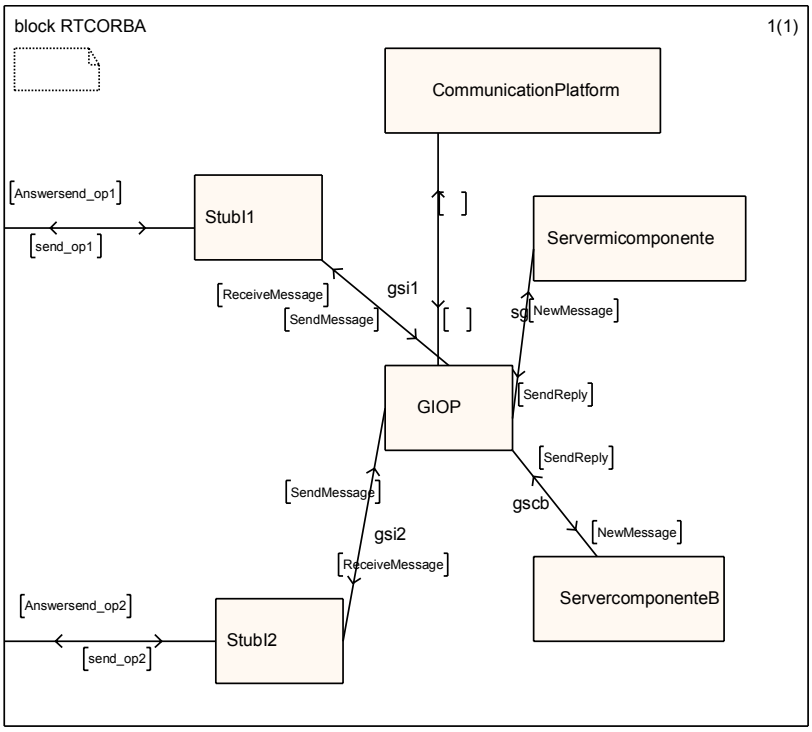


Figura 4.29. Bloque RT-CORBA del modelo plano final

Este modelo SDL final es combinado con la información obtenida en las pruebas temporales sobre la plataforma final, de forma que la herramienta de análisis puede ya realizar el análisis de la aplicación al disponer de un modelo SDL donde puede estudiar las relaciones de precedencia entre los componentes, y la información temporal.

De esta forma, la transición existente en el interior de `componenteA` y que invoca a la operación `I2.op1` de `micomponente`, tiene que considerar como tiempo de respuesta tanto su tiempo local, como la sobrecarga remota. Es decir,

$$R_{ta} = R_{local} + R_{I2.op1}$$

La Figura 4.30 muestra un ejemplo de utilización de la herramienta de análisis del entorno. Esta herramienta está basada en la herramienta SDLTR presentada en [Llopis, 2002], habiendo sido ampliada para la realización del análisis distribuido. Para realizar el análisis del peor tiempo de respuesta, la herramienta debe extraer los eventos que pueden producirse en el sistema, realizándose el análisis del tiempo de respuesta de estos eventos.

Tras la realización del análisis de planificabilidad, y si este es satisfactorio, la aplicación podrá ejecutarse en la configuración elegida teniendo garantizado el cumplimiento de los requisitos de tiempo real.

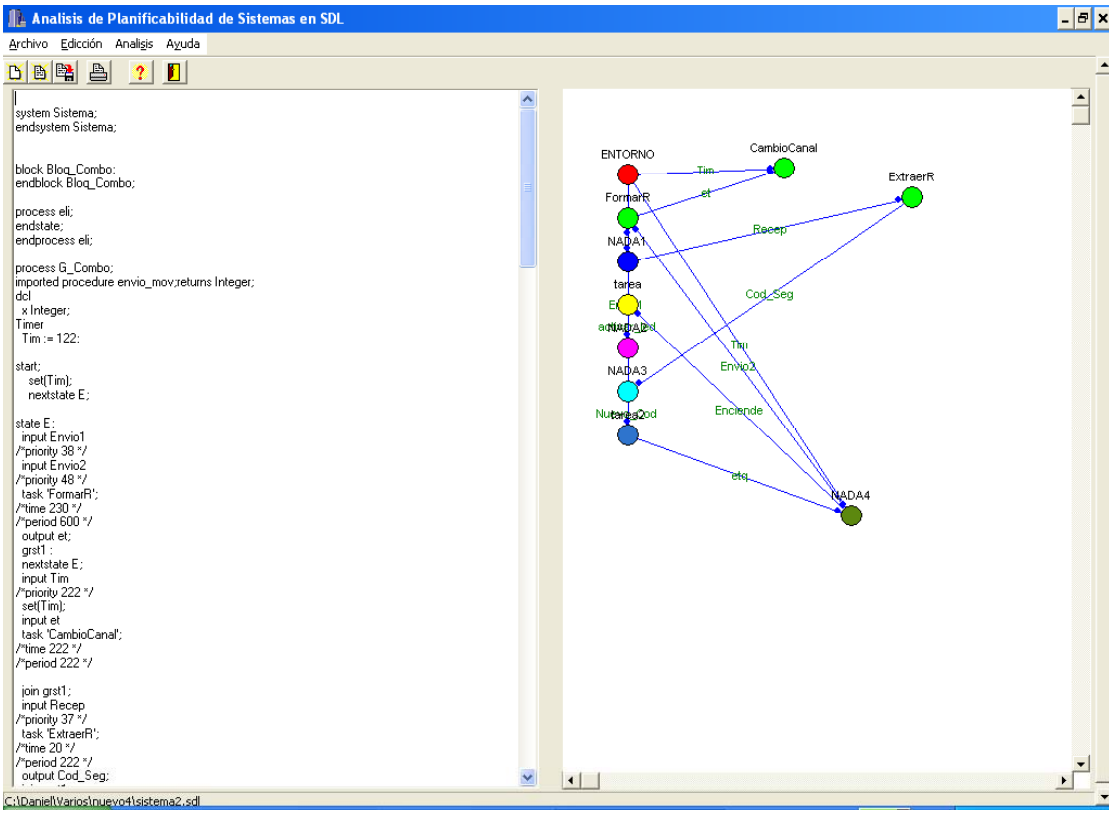


Figura 4.30. Herramienta de análisis

Capítulo 5. Aplicación: Simuladores para Centrales Nucleares

La aplicación de nuevas tecnologías y herramientas de desarrollo representa un desafío y un riesgo económico para las empresas, que no todas están preparadas para asumir. Las aportaciones realizadas en esta tesis: modelo de componentes, plataforma de ejecución, análisis de tiempo real han sido utilizadas de forma experimental en un campo en el que el Grupo de Ingeniería del Software de la Universidad de Málaga (GISUM) tiene una larga experiencia, como es la simulación para centrales nucleares.

Dentro de este campo, se han venido realizando diferentes aplicaciones, herramientas, motores de simulación, etc. [Díaz y Garrido, 2003][Díaz y Garrido, 2004][Díaz y Garrido, 2004b][Díaz et al., 2005b] Estas aplicaciones suelen ser de una alta complejidad, son distribuidas y presentan requisitos temporales, por lo que pareció idóneo la aplicación de todos los trabajos realizados en esta tesis a la reingeniería de ciertas partes de los simuladores.

Los simuladores son réplicas exactas de las salas de control de las centrales, teniendo en cuenta todo tipo de detalles, desde mobiliario, paneles de control, etc. al software, simulando las aplicaciones ejecutándose en la central.

El núcleo de los simuladores está formado por modelos de simulación con restricciones de tiempo real, los cuales proporcionan los valores de las distintas señales y variables necesitadas por el resto de componentes hardware y software.

El propósito principal de los simuladores es el entrenamiento de los operadores de la central de una manera segura, permitiéndoles practicar diferentes situaciones, desde las más normales como, por ejemplo, la monitorización de la temperatura, manipulación de válvulas, etc. hasta situaciones de emergencia. La duración de las sesiones de simulación puede estar en el orden de unos pocos días, de forma que el

software desarrollado y la plataforma hardware tienen que proporcionar calidad, estabilidad, robustez, etc.

El trabajo realizado en esta tesis se ha centrado en una parte de estos simuladores, pero con la perspectiva de poder integrarse en simuladores ya en funcionamiento o por desarrollar. En concreto, el trabajo desarrollado se ha centrado en las partes de las comunicaciones entre el núcleo del simulador, los modelos de simulación y las aplicaciones. Es en esta parte de los simuladores donde se realiza la transmisión de datos entre unos *componentes* del sistema y otros, siendo una parte vital para el correcto funcionamiento de los simuladores.

UM-RTCOM debería ser suficientemente expresivo como para no tener que utilizar mecanismos externos explícitamente, como por ejemplo características de RT-CORBA. De esta forma, los modelos de simulación, por ejemplo, deberían tener por norma general, más prioridad que el resto de aplicaciones. También tendrían que poder realizarse invocaciones simultáneas por parte de las aplicaciones para obtener variables de simulación o, disponer de exclusión mutua en algunas acciones sobre los simuladores. Todas estas características van a ser satisfechas con el modelado hecho en UM-RTCOM. Así, por ejemplo, las herramientas del modelo examinarán los requisitos temporales de los modelos de simulación expresados en UM-RTCOM, obteniendo así las prioridades de dichos modelos. Estas herramientas permitirán también desplegar los componentes con capacidad para poder recibir más de una invocación simultánea y, por último, la exclusión mutua puede ser lograda a través de los componentes pasivos. Las principales ventajas de utilizar UM-RTCOM son la utilización de la tecnología de componentes, y la posibilidad de aplicar las técnicas de análisis descritas.

5.1. Arquitectura del sistema

Los entornos de simulación que nos ocupan, usualmente incluyen dos tipos diferentes de simuladores que influyen en la arquitectura hardware y en las infraestructuras físicas. El primer tipo de simulador es el denominado *Simulador Gráfico Interactivo (SGI)*, el cual utiliza aplicaciones gráficas para realizar el entrenamiento de los operadores. El segundo tipo de simulador es el denominado *Simulador de Alcance Total (SAT)*, siendo éste el más difícil de construir al ser una réplica exacta de la sala de control de la central nuclear simulada (ver Figura 5.1).

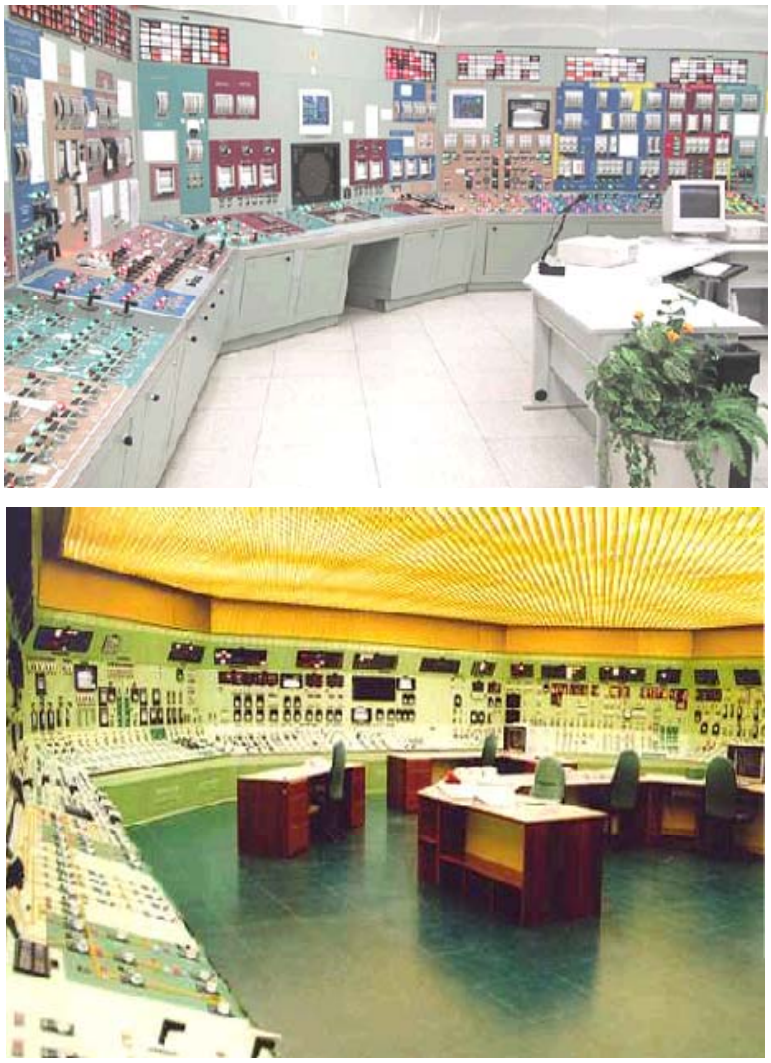


Figura 5.1. Simuladores de Alcance Total

Los principales elementos hardware de SAT y SGI son los siguientes:

- **Ordenadores de simulación**: estos ordenadores son los responsables del proceso de simulación, llevando a cabo la ejecución de los modelos de simulación y proporcionando datos al resto de componentes software y hardware. Constituyen los elementos principales de los simuladores.
- **Consola del instructor**: la consola del instructor existe únicamente en el contexto de los simuladores. A través de la consola del instructor se crean situaciones que tiene que ser resueltas por los estudiantes.
- **Paneles físicos**: los paneles físicos son réplicas exactas de los existentes en la sala de control. Son utilizados únicamente en simuladores tipo SAT y proporcionan al operador la sensación de estar en la sala de control real. Están situados en grandes habitaciones (ej:17x18 metros) donde los operadores realizan sus acciones. Los paneles tienen una cantidad considerable de indicadores,

teclados hardware, válvulas, etc. Todos los elementos de los paneles físicos (hardware) tienen que ser también controlados por el software en la simulación.

- **Subsistemas:** dependiendo de la planta de energía a simular, puede haber varios subsistemas que tienen que ser simulados. Estos subsistemas, normalmente hardware, pueden ser muy diferentes entre sí y, como consecuencia, la reutilización de código es un aspecto fundamental. Un simple subsistema puede incluir elementos software y hardware, tales como ordenadores, teclados, etc.
- **Puestos de estudiante**: los simuladores SGI incluyen adicionalmente el hardware requerido para los *puestos de estudiantes* del simulador. Básicamente, un puesto de estudiante permite la práctica de cualquier aspecto de la simulación de la sala de control, pero de una manera confortable, donde cada estudiante dispone de un equipo con aplicaciones gráficas y varios monitores.

Un esquema del equipamiento hardware de los simuladores y de las instalaciones físicas puede ser visto en la Figura 5.2). La habitación del instructor está situada en la parte superior de la figura. En esta sala, el instructor manipula la sesión de simulación utilizando la consola del instructor. Junto a esta habitación están situados los ordenadores de simulación en una habitación independiente. Finalmente, dependiendo del tipo de simulador utilizado (SGI o SAT), puede haber varios puestos de estudiante en los simuladores SGI, y para los simuladores SAT existe además una habitación, réplica de la sala de control con los paneles físicos y subsistemas.

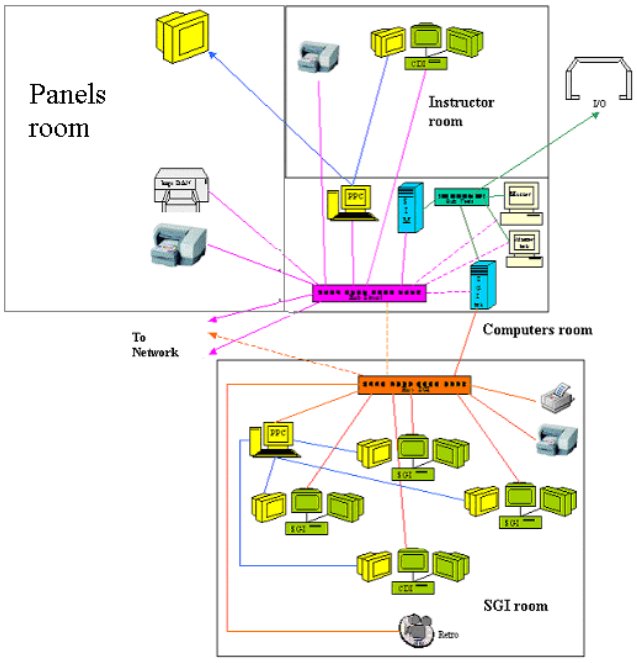


Figura 5.2. Esquema de los simuladores

5.2. Arquitectura software

Hay dos partes bien diferenciadas en la arquitectura software. La primera parte comprende los componentes que actúan como un "servidor de simulación", ofreciendo distintos servicios de simulación a las otras herramientas y aplicaciones. Precisamente, estas herramientas y aplicaciones utilizadas por el instructor y los estudiantes, forman la segunda parte del sistema. Todos estos "componentes" son distribuidos, pudiendo ser ejecutados sobre cualquier nodo de la red. Un diagrama de componentes de alto nivel junto con sus interacciones puede verse en la Figura 5.3.

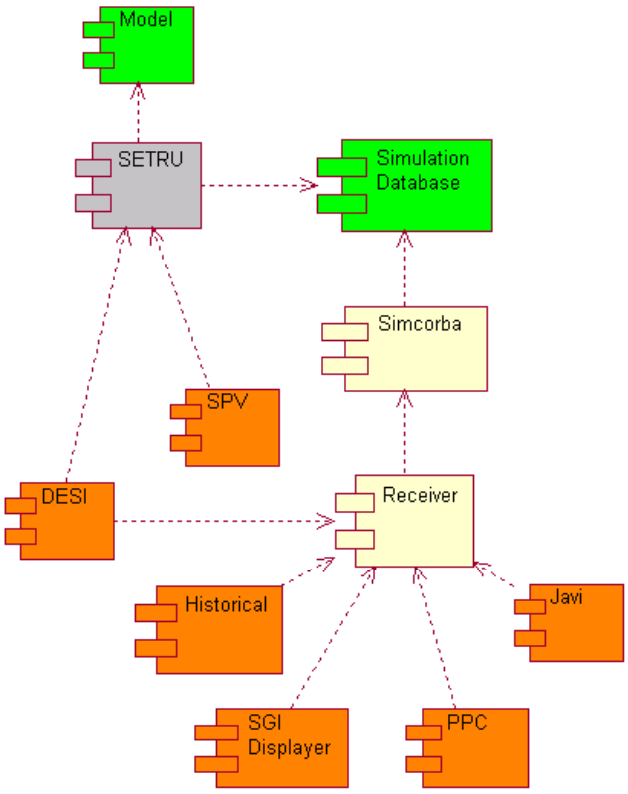


Figura 5.3. Componentes de alto nivel de los simuladores

A continuación se presenta una breve descripción de algunos de estos componentes:

- *SETRU*: el denominado *SETRU* es el componente software principal de los simuladores, constituyendo el motor de simulación responsable de la ejecución de los modelos de simulación.
- *Simulation Database*: base de datos de simulación. Un simulador incluye una gran cantidad de variables, en torno a varias decenas de miles, por lo que se

requiere una base de datos con información sobre todas las variables de simulación: ubicación, descripción, tipos de datos, etc.

- *Model*: los componentes *Model* representan a cada uno de los modelos de simulación del sistema. Los modelos de simulación son una de las partes principales de los simuladores, ya que, cada uno de ellos se encarga de modelar el comportamiento de alguna parte de la planta.
- *Simcorba*: ***Sim****ulator* ***Corba*** es uno de los principales componentes de comunicaciones del sistema. Actúa como "el Servidor de Simulación" para un gran número de aplicaciones clientes, ofreciendo un conjunto de servicios que incluyen el envío periódico de variables, acciones de usuario sobre el simulador, etc.
- *Receiver*: es la parte equivalente a *Simcorba* en la parte de los clientes. Básicamente, es un contenedor de datos que permite a las aplicaciones obtener variables de simulación. Hay un componente *Receiver* por cada aplicación que necesita obtener datos del núcleo de simulación a través de *Simcorba.*

El resto de componentes de la parte cliente son aplicaciones y subsistemas utilizando el componente *Receiver*. Algunos ejemplos son los siguientes:

- *DESI*: Depurador de variables. Es una aplicación que permite depurar el proceso de simulación. Los usuarios pueden modificar variables, consultar variables, mostrar gráficos, etc.
- *SPV*: Supervisor. Aplicación que permite gestionar muchos aspectos de la simulación, tales como el tiempo de ciclado, gestión de los modelos de simulación, etc.
- *SGI Displayer*: Aplicación utilizada en los puestos de estudiante de los simuladores SGI. Utiliza láminas gráficas que permiten examinar y manipular diferentes componentes de la central.
- *Javi*: Herramienta para la visualización del estado de la planta. Utiliza también láminas que permiten observar de manera global el funcionamiento de toda la planta. El usuario obtiene una visión global del sistema de manera instantánea.
- *Historical*: Aplicación para almacenar información de la solicitud en tiempo real. Básicamente almacena las variables de simulación, de forma que posteriormente se pueda reproducir paso a paso el desarrollo de una sesión de simulación pudiendo examinar las acciones de los usuarios.

Debido a la heterogeneidad existente en estos simuladores, se han utilizado entornos de ejecución basados tanto en Unix como en Windows. Unix ha sido utilizado principalmente en el núcleo del simulador. Las herramientas gráficas han sido desarrolladas en su mayor parte en Windows o en Java.

La utilización del modelo de componentes junto con la plataforma de ejecución ha permitido de forma satisfactoria el desarrollo de componentes reutilizables, con todos los beneficios que ello conlleva, además de garantizar las restricciones temporales en las partes críticas desarrolladas con el modelo de componentes propuesto. De esta forma, los componentes desarrollados podrán ser reutilizados en posteriores proyectos de simulación, sin realizar modificaciones a los mismos y facilitando enormemente la labor de los desarrolladores.

5.3. Núcleo del simulador

SETRU es el núcleo de los diferentes simuladores de Tecnatom, proporcionando un entorno de ejecución para los modelos de simulación utilizados en un simulador concreto. La utilización de *SETRU* facilita la ejecución, modificación, inserción, etc. de nuevos modelos de simulación.

Los modelos de simulación son los responsables de la precisa simulación de componentes físicos reales tales como válvulas, sensores, actuadores, etc. proporcionando a las aplicaciones un conjunto de variables de simulación que representan a los componentes físicos y que pueden ser consultadas, modificadas, etc. Los modelos son representados a través de componentes software del modelo UM-RTCOM estando distribuidos en diferentes máquinas del sistema.

Hay dos modos de ejecución del núcleo. El primer modo, denominado "*SETRU"*, permite simulaciones en tiempo real, donde el tiempo interno de la simulación se corresponde con tiempo real y los modelos de simulación deben ejecutarse en tiempo real. Esto significa, que un segundo en la vida real es igual a un segundo en el tiempo de simulación de los modelos de simulación. Si los modelos de simulación no pueden ser ejecutados con la frecuencia deseada, la sesión de simulación se vería afectada al no coincidir el tiempo real con el tiempo de simulación.

Cuando el tiempo real no es una prioridad, existe un segundo modo de simulación denominado "*URTES"*. En este modo, y aunque parezca contradictorio, la

simulación se ejecuta a la mayor velocidad posible, sin embargo, no existen garantías de tiempo real, y de esta forma los modelos de simulación o el núcleo de simulación competirían por los recursos del sistema en igualdad de condiciones con las demás aplicaciones.

Un aspecto esencial para comprender el funcionamiento de los simuladores es la estructuración de la planta en un conjunto de componentes donde cada componente se divide a su vez en varias celdas. Un *componente* podría ser, por ejemplo, una tubería, y, dicha tubería podría ser dividida en varias secciones, las cuales serían las *celdas*. La Figura 5.4 muestra un ejemplo de componente con sus celdas. El papel de los modelos de simulación es actualizar los valores de diferentes variables (presión, velocidad del líquido, temperatura, etc.) para cada una de las celdas de los diferentes componentes que forman la planta.

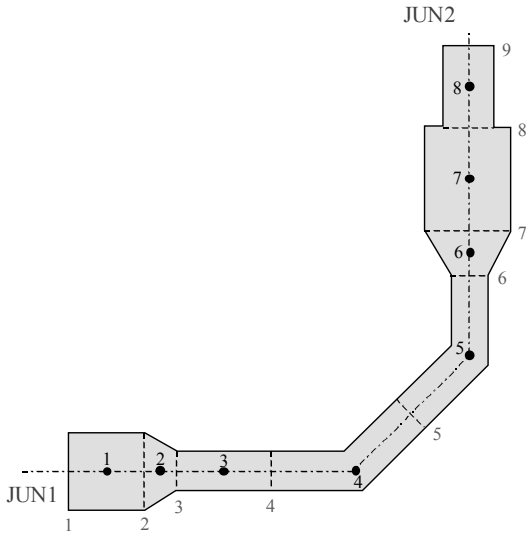


Figura 5.4. División de un componente en celdas

Los modelos de simulación son ejecutados en diferentes máquinas de la red, y sus resultados son actualizados en el núcleo del simulador (*SETRU*), permitiendo realizar a las aplicaciones de comunicaciones la actualización de las variables de simulación en las diferentes herramientas.

En los simuladores hay una gran cantidad de modelos de simulación muy diferentes, desde algunos que permiten simular partes muy pequeñas de la central, como por ejemplo el funcionamiento de una válvula, hasta otros mucho más complejos que simulan el modelo termohidráulico de la planta (TRAC [Alvarez et al, 1996]) o el modelo neutrónico (NEMO), pudiendo haber dependencias de variables entre diferentes

modelos y pudiendo tener diferentes requisitos de tiempo real y consecuentemente diferentes prioridades.

Adicionalmente, el núcleo del simulador puede ser controlado remotamente a través de diferentes aplicaciones como *DESI* o *SPV*, por lo que también tendría que ser considerada su influencia sobre la ejecución de los modelos.

5.3.1. Nueva arquitectura

Para la utilización del modelo UM-RTCOM, tanto *SETRU* como los modelos de simulación, han sido "convertidos" en componentes. *SETRU* se ha convertido en un componente/aplicación, y los modelos de simulación han sido desplegados en los ordenadores de simulación. La comunicación del núcleo con el resto de aplicaciones se realiza a través de *Simcorba*, que también ha sido componentizado. La nueva arquitectura del núcleo del simulador basada en componentes puede ser vista en la Figura 5.5.

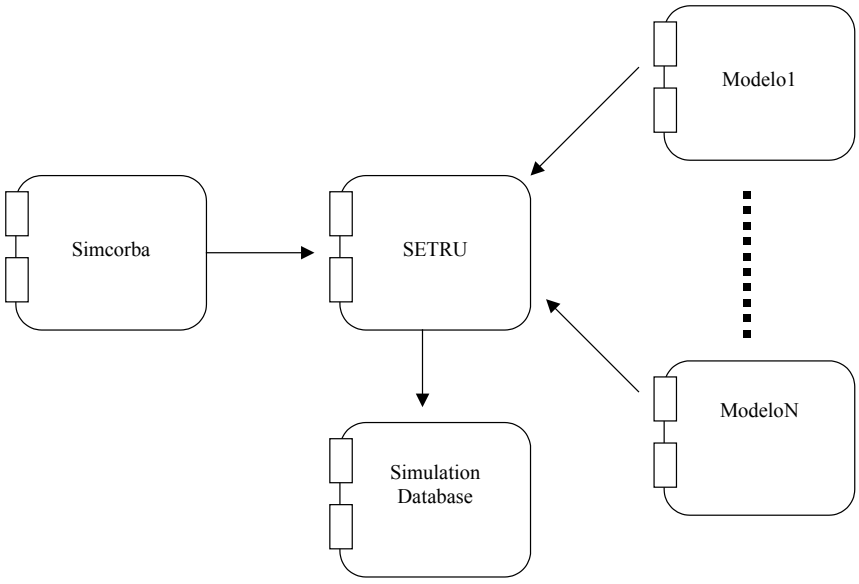


Figura 5.5. Nueva arquitectura del núcleo del simulador

El funcionamiento del núcleo del simulador se basa en la ejecución de *"pasos de simulación"*. En cada paso, se realiza la ejecución de los diferentes modelos de simulación, teniendo que esperar por la finalización de todos los modelos para comenzar un nuevo paso de simulación. Por su parte, las aplicaciones pueden solicitar datos al simulador en todo momento, independientemente de que se esté llevando a

cabo un paso de simulación. El componente *SETRU* es el motor encargado de llevar a cabo todos estos procesos.

La Figura 5.6 muestra un diagrama de secuencia UML con el siguiente comportamiento: en primer lugar, el componente *SETRU* genera un evento (`nextFrame`) indicando el comienzo del siguiente paso de ejecución. Este evento es consumido por todos los modelos de simulación, los cuales comienzan un paso de simulación. Durante la ejecución de un paso, los modelos pueden requerir variables de otros modelos, por lo que las variables requeridas son solicitadas (`send_var_monitor`) a *SETRU*. Cuando los modelos finalizan su paso, realizan la actualización de las variables, utilizando para ello otro servicio de *SETRU* (`send_var_single_update`). Finalmente, lanzan otro evento (`endFrame`) indicando la finalización del paso. Cuando *SETRU* tiene constancia de que todos los modelos han finalizado, espera hasta que tenga que comenzar el siguiente paso. Durante todo este proceso es posible que lleguen también solicitudes de aplicaciones que también tienen que ser tratadas por *SETRU*.

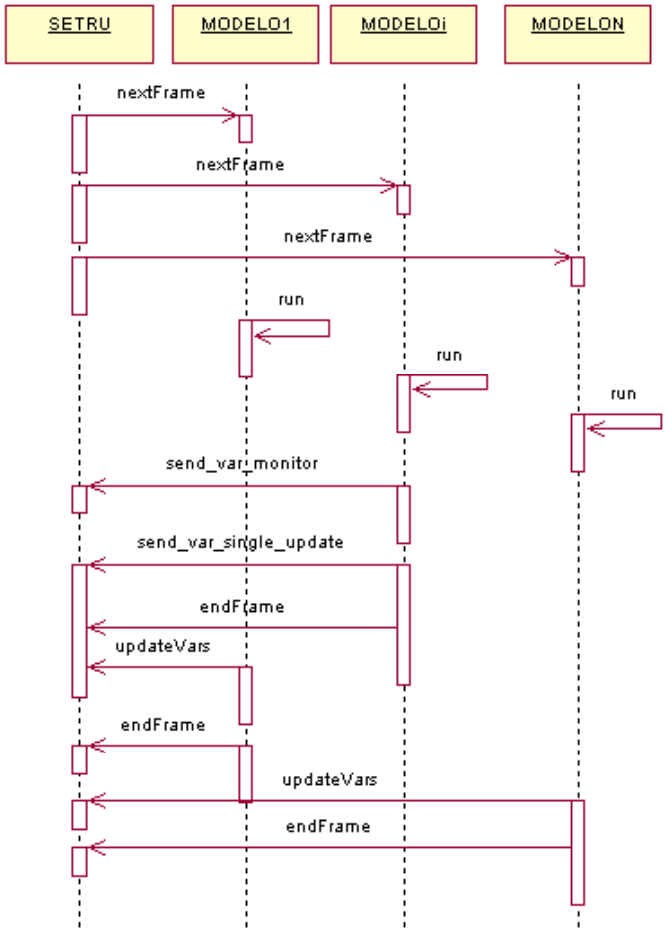


Figura 5.6. Ejecución de pasos de simulación

5.3.2. Nuevos componentes del núcleo

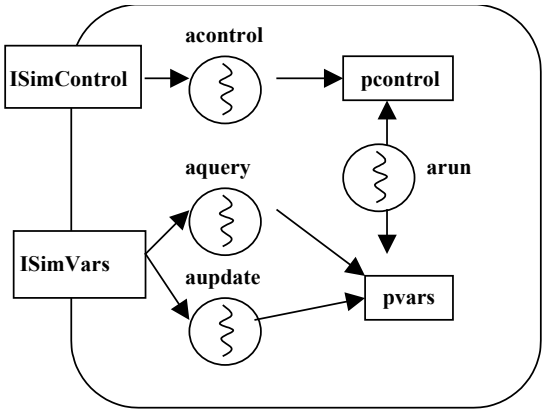
El siguiente fragmento de código muestra partes de la definición del componente *SETRU* en relación al control del simulador: inicio/parada de simulación, control de simulación, condiciones iniciales de simulación, etc. El componente *SETRU* utiliza la posibilidad de disponer de múltiples interfaces, disponiendo de una interfaz `ISimControl` para proporcionar servicios de control de la simulación. Se incluyen también los eventos producidos y consumidos así como ranuras de configuración que permiten, por ejemplo, especificar el periodo de los pasos de simulación. Esta interfaz es utilizada principalmente por las aplicaciones *DESI* y *SPV* para el control de la simulación.

```
Component SETRU {
      interface ISimControl {
            // start-stop methods
            void send_run(out string errMsg);
            void send_freeze(out string errMsg);
            void get_state(out t_Sim_State state,
                           out string errMsg);

            // Generic command
            void send_command(in string command,in string params,
                           out string errMsg);

            // simulation control services
            void send_slowtime(in long slow, out string errMsg);
            void send_normal_time(out string errMsg);
            void send_fast_time(in long fast, out string errMsg);
            void send_compute_n(in long nsteps, out string errMsg);
            void send_step_n(in long n_step, out string errMsg);
            void send_step_cont(out string errMsg);
            }

            // initial condition services
            void send_load_ic(in short icnumber, out string errMsg);
            void send_save_ic(in short icnumber, out string errMsg);
            void send_backtrack(in long num, out string errMsg);
            }

        input ISimControl;
        publishes nextFrame;
        consumes endFrame;


        config {
            int frame_period; // Periodo de los pasos de simulación
            ...
        }
}

event nextFrame();
event endFrame(in string modelName);
```

El mantenimiento de las variables de simulación es la otra parte importante en SETRU. Se ofrece otra interfaz, `ISimVars`, con varios métodos para la consulta y actualización de variables de simulación, de los cuales se muestran sólo algunos por claridad. Esta interfaz es utilizada principalmente tanto por los modelos de simulación como por el componente *Simcorba*.

```
Component SETRU {
      ...

      interface ISimVars {
            ...

                // Query of simulation variables

                void send_var_query(in string varName,
                                    inout char varType,
                                    in short cell,
                                    in short comp,
                                    out double value,
                                    out double max,
                                    out double min,
                                    out string des,
                                    out string uni,
                                    out short dim1,
                                    out short dim2,
                                    out short dim3,
                                    out string errMsg );

                // Updating of simulation variables

                void send_var_single_update(in string varName,
                                            in short cell,
                                            in short comp,
                                            in double value,
                                            out string errMsg);

            }

}
```

La Figura 5.7 muestra los activos requeridos para el control de la simulación (`acontrol`), peticiones de variables (`aquery`) y actualización de variables (`aupdate`), junto con el componente activo central de *SETRU* (`arun`) que se encarga de llevar a cabo el control de los pasos de simulación, y en general de la simulación. También son requeridos dos componentes pasivos que representan zonas de memoria para el control de la simulación (`pcontrol`) y para mantener los valores de las variables de simulación (`pvars`).



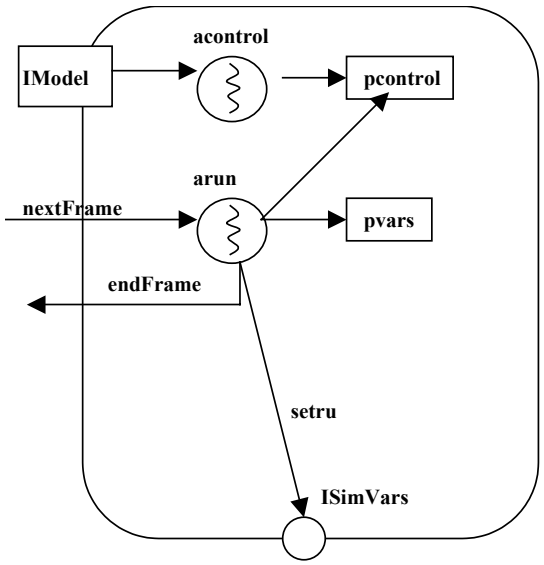

```
Component SETRU {
  interface ISimControl {}
  interface ISimVars {}
  input  ISimControl,ISimVars;
}

Component implementation SETRU {
  Active AControl {… }
  Active AQueryVars {…}
  Active AUpdateVars {…}
  Active ARunSetru {…}

  Passive PMemory {…}


  AControl acontrol;
  AQueryVars aquery;
  AUpdateVars aupdate;
  ARunSetru arun;
  PMemory pcontrol,pvars;
}
```

Figura 5.7. Componente SETRU

Los modelos de simulación están basados en la utilización de un componente *Model*, el cual puede ser personalizado para su utilización con diferentes modelos a través de los métodos de inicialización. También dispone de métodos de consulta, actualización, etc. al margen de los ofrecidos por *SETRU*, pudiendo ser así reutilizados en otros entornos diferentes.

```
Component Model {
      ...
      interface IModel {
            ...
            void init(in string modelName,in string configInfo);

            void nextFrame();


            // Query of simulation variables
            void send_var_query(in string varName,
              inout char varType, in short cell,in short comp,
              out double value,out double max,out double min,
              out string des,out string uni,out short dim1,
              out short dim2,out short dim3,out string errMsg );

            // Updating of simulation variables
            void send_var_single_update(in string varName,
              in short cell,in short comp,in double value,
              out string errMsg);
      }

      input IModel;
      output ISimVars with setru;   // Interfaz requerida
      consumes nextFrame;
      publishes endFrame;
}
```

La parte central de un componente `IModel` está constituida por un componente activo (`arun`) esperando por peticiones de nuevas invocaciones del modelo. Este componente activo consume el evento `nextFrame` y produce el evento `endFrame` cuando acaba la ejecución de un paso de simulación. También existe otro activo (`acontrol`) para tratar las peticiones que se puedan realizar al modelo. Existen dos componentes pasivos para almacenar información de control (`pcontrol`) y las variables requeridas y ofertadas por el modelo (`pvars`). Por último, los modelos deben interconectarse con el componente *SETRU* de forma que puedan consultarse o modificarse variables, para ello se indica una referencia (`setru`) para la interfaz de salida `ISimVars` declarada en el componente *SETRU*.




```
Component Model {
  interface IModel {}
  input IModel;
  output ISimVars with setru;
  consumes nextFrame;
  publishes endFrame;
}

Component implementation Model {
  Active AControl {… }
  Active ARunModel {…}

  Passive PMemory {…}


  AControl acontrol;
  ARunModel arun;
  PMemory pcontrol,pvars;
}
```

Figura 5.8. Componente `Model`

5.3.3. Características de tiempo real

La nueva versión de los simuladores tiene como principal ventaja con respecto a las anteriores, la disminución en la carga de los procesadores y la posibilidad de disminuir el tiempo de los pasos de simulación. En las versiones previas, los diferentes modelos de simulación se ejecutaban de forma secuencial. En esta nueva versión, los modelos pueden ser ejecutados de forma paralela. Así, por ejemplo, si tuviéramos tres modelos A, B y C con peores tiempos de ejecución de 40, 50 y 60 ms. respectivamente, en la versión antigua, el tiempo de un paso de simulación sería como mínimo la suma de todos los tiempos, 150 ms. Por el contrario, en la nueva versión, el tiempo de ejecución de un paso de simulación sería el del peor modelo, en este caso 60 ms.

Las herramientas de UM-RTCOM son responsables de comprobar las precedencias existentes entre los diferentes modelos y aplicaciones para poder establecer correctamente las prioridades de los diferentes componentes activos existentes tanto en *SETRU* como en los modelos de simulación. Se dispone, asimismo de suficientes hebras en la traducción RT-CORBA para poder atender peticiones simultáneas. Esto es posible gracias al conocimiento del número máximo de modelos y de aplicaciones que pueden realizar peticiones simultáneamente. Por otra parte, la utilización del modelo propagado por el cliente de RT-CORBA permite minimizar las inversiones de prioridad y de esta forma los modelos de simulación se ejecutarán y tendrán acceso a los diferentes servicios de *SETRU* con una mayor prioridad que por ejemplo las aplicaciones. La Figura 5.9 muestra un ejemplo donde se están ejecutando los modelos de simulación TRAC y NEMO junto con *DESI* y *SPV*. En el ejemplo, la prioridad CORBA de los modelos de simulación es mayor que la de las aplicaciones.

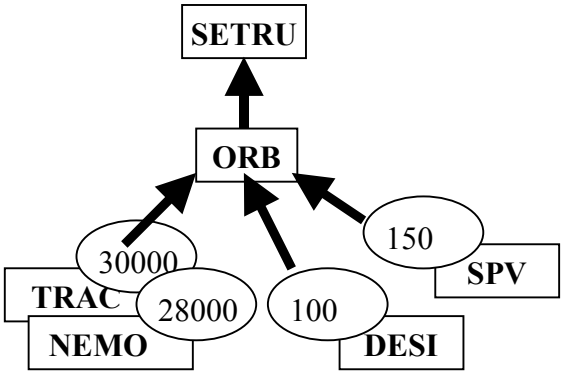


Figura 5.9. Prioridades en modelos y herramientas

Los modelos de simulación tienen todos el mismo tiempo límite de ejecución, que es el tiempo máximo para la ejecución de un paso de simulación, y que es establecido por los desarrolladores de los simuladores. Estas restricciones de tiempo real pueden indicarse utilizando las construcciones de UM-RTCOM. Así, por ejemplo, el periodo mínimo entre la creación de dos eventos `nextFrame` puede ser indicado como se indica a continuación, tanto en el componente *SETRU*, como en los modelos de simulación.

Ej:

```
// Indicado en SETRU
constraints nextFrame Period 125;

// Indicado en los modelos
constrainsts nextFrame Deadline 125;
```

La sincronización en *SETRU* es un aspecto fundamental. Existen diferentes partes del código que tienen que ejecutarse en exclusión mutua. Todas estas partes han sido agrupadas adecuadamente utilizando componentes pasivos como medio para lograr la exclusión mutua y para evitar o minimizar inversiones de prioridad. Así por ejemplo, se evitan situaciones como la ejecución simultánea de comandos contradictorios como arrancar o parar la simulación.

5.4. Comunicaciones: Simcorba/Receiver

La pareja *Simcorba/Receiver* desempeña un papel fundamental en la arquitectura de las comunicaciones de los simuladores. La antigua aplicación *Simcorba* (nombre derivado de ***Sim****ulador* ***Cor****ba*) era la responsable de la transmisión de los datos de simulación al resto de aplicaciones clientes. Este papel va ahora a ser desempeñado por el nuevo componente *Simcorba* contenido en el núcleo del simulador. *Simcorba* está estrechamente relacionado con el componente *Receiver* utilizado por muchas aplicaciones, siendo responsable de la recepción de los datos enviados por *Simcorba* y haciéndolos accesibles a las aplicaciones.

Hay dos versiones de esta pareja: la primera versión utiliza principalmente servicios de interfaces. La segunda versión está basada en la utilización de eventos. La utilización de las versiones es transparente para las aplicaciones que utilizan el componente receptor.

5.4.1. *Simcorba/Receiver* basado en servicios

En la versión basada en servicios, los componentes receptores solicitan periódicamente datos a *Simcorba*. Los datos enviados por *Simcorba* están estrechamente relacionados al tipo de cliente o aplicación, el cual es indicado en fase de inicialización. Existen diferentes tipos para diferentes tipos de aplicaciones como *PPC, SGI, CDI,* etc. Cada uno de estos tipos de aplicaciones pueden tener además distintos requisitos de tiempo real, por lo que la indicación de las restricciones de tiempo real será necesaria para poder asignar prioridades.

Simcorba tiene la información necesaria para conocer las variables requeridas por cada tipo en una manera flexible que permite la incorporación de nuevos tipos sin

tener que cambiar el código de las aplicaciones. Cuando llegan nuevas peticiones de variables a *Simcorba*, éste tiene que interactuar con el resto del núcleo del simulador para obtener la información requerida.

Si bien las aplicaciones podrían haber interactuado directamente con el componente *SETRU* para obtener datos de simulación, se ha optado por disponer de una capa intermedia representada por el componente *Simcorba*. De esta forma, por ejemplo, las aplicaciones podrían ser utilizadas en entornos de simulación diferentes a los desarrollados por Tecnatom S.A.

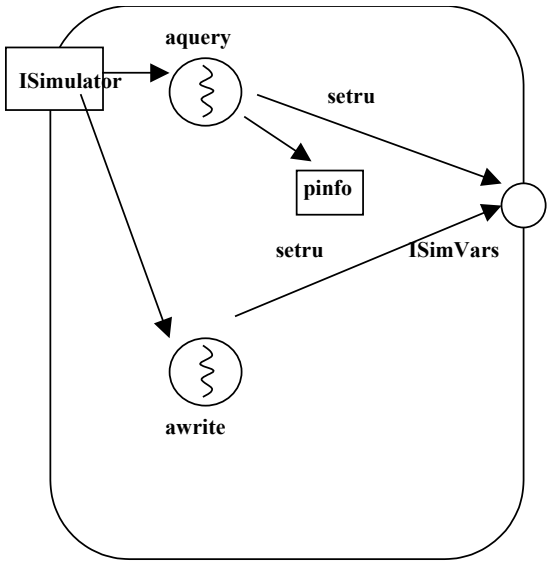
Los siguientes fragmentos de código muestran parte de la definición de los tipos de datos utilizados por los simuladores. Se consideran dos tipos de variables: analógicas (valores en punto flotante) y digitales (valores lógicos). Hay también que considerar listas de variables de estos tipos (ej: `listVblesAnalog`) e incluso se han añadido secuencias (`secDigitalValues`), por motivos que posteriormente se detallarán. También se han definido estructuras (ej: `StructChangedValues`) que permiten almacenar listas de variables.

```
  // single variables
  struct analogVble {
      float value;      long id; };
  struct digitalVble {
      unsigned short value;
      long id;};
  struct inputVble  {
      string name; double value; };

  // lists of variables
  typedef sequence<analogVble>
      listVblesAnalog;
  typedef sequence<digitalVble>
      listVblesDigital;
  typedef sequence<float>
      secAnalogValues;
  typedef sequence<unsigned short>
      secDigitalValues;
  typedef sequence<inputVble>
      secInputVbles;

  // all the variables
  struct StructAllValues {
      secAnalogValues analogues;
      secDigitalValues digitals;
      };

  // changed variables
  struct StructChangedValues {
      listVblesAnalog analogues;
      listVblesDigital digitals;
};
```

El siguiente fragmento de código muestra parte de las definiciones del componente *Simcorba*. Se ofrece una interfaz, `ISimulator`, con métodos para el envío de las variables de simulación a las aplicaciones o para actualizar variables en el simulador. También se requiere interactuar con el resto del núcleo del simulador, para ello se utiliza una referencia a la interfaz `ISimVars`, que será compuesta con el componente *SETRU*.

```
Component Simcorba {
  interface ISimulator {
      // Send all variables
      void sendAllValues(in string type,out StructAllValues values);

      // Send changed variables
      void sendChangedValues(in string type,
                             out StructChangedValues values);

      // Update simulation variables
      void writeValues(in secInputVbles s);
  }

  input ISimulator;
  output ISimVars with setru; // Interfaz requerida
}
```

El componente *Simcorba* tiene dos componentes activos encargados de la petición de variables (`AQuery`) y actualización de variables (`AUpdate`) utilizando el componente *SETRU*. Se dispone además de un componente pasivo (`PInfoVars`) encargado de almacenar información sobre los diferentes tipos de componentes receptores.




```
Component Simcorba {
  interface ISimulator {}

  input ISimulator;
  output ISimVars with setru;
}

Component implementation Simcorba {
  Active AQuery {…}
  Active Awrite {…}

  Passive PInfoVars {…}

  AQuery aquery;
  AWrite awrite;

  PInfoVars pinfo;
}
```

Figura 5.10. Componente `Simcorba`

Los componentes *Simcorba* y *Receiver* tienen como principal desafío el gran número de variables existente en los simuladores (en torno a varias decenas de miles) y su actualización en las aplicaciones (aproximadamente 4 veces por segundo). Estos requisitos influyeron en la definición de los servicios ofertados por *Simcorba*. De esta forma, hay dos alternativas:

La primera posibilidad consiste en transferir todas las variables en cada actualización. En la segunda posibilidad sólo se realiza la transferencia de las variables cambiadas. El envío de todas las variables tiene como principal ventaja la simplicidad en la implementación del código. Por el contrario, el gran volumen de datos hace utilizar preferentemente la segunda opción. En cualquier caso, *Simcorba* ofrece los servicios `sendAllValues` y `sendChangedValues` para transferir todas las variables o sólo las modificadas, respectivamente. Ambas opciones pueden ser utilizadas actualmente.

Finalmente, también se ofrece el servicio `writeValues` para la modificación de variables en el simulador, pudiendo simular, por ejemplo, acciones de usuario.

5.4.2. *Simcorba/Receiver* basado en eventos

La segunda versión de la pareja *Simcorba/Receiver* está basada en la utilización de los eventos de UM-RTCOM. En este caso hay una mayor flexibilidad al no tener que interconectar a los componentes entre sí. Como inconveniente, está el hecho de que esta versión depende de la robustez y estabilidad del servicio de eventos subyacente, como por ejemplo pueda ser el servicio de eventos de tiempo real de TAO.

Los diferentes tipos de cliente (*PPC*, *SGI*, CDI, etc.) siguieron siendo utilizados, pero readaptados para su utilización como tipos de eventos. En cualquier caso, *Simcorba* siguió ofreciendo algunos servicios para su utilización de manera independiente al servicio de eventos o para situaciones especiales, como por ejemplo, nuevos clientes conectados que quieran recuperar todas las variables o un caso más usual, la modificación de variables en el simulador.

Los cambios necesarios en el código pasan por introducir un nuevo tipo de eventos, que será el utilizado para transmitir los datos, y la indicación en los componentes *Simcorba* y *Receiver* de que ese evento va a ser producido y consumido respectivamente.

```
event event_changed_values(in string type,in StructChangedValues);

Component Simcorba {
...

  input ISimulator;
  output ISimVars with setru; // Interfaz requerida

  publishes event_changed_values;
}

Component Receiver {
...
  consumes event_changed_values;
}
```

Cuando un evento se produce, se indica el tipo de evento en él mismo, de forma que los clientes no interesados pueden filtrarlos. En una versión más refinada de estos componentes, podrían tenerse distintos tipos de eventos en vez de un único `event_changed_values`, de esta forma, el filtrado no tendría que hacerlo el desarrollador del componente, sino que este sería realizado automáticamente por el servicio de eventos. La Figura 5.11 muestra este caso con un tipo de eventos *PPC* y varios consumidores del mismo. La transmisión se realiza utilizando *multicast* gracias al servicio de eventos de TAO.

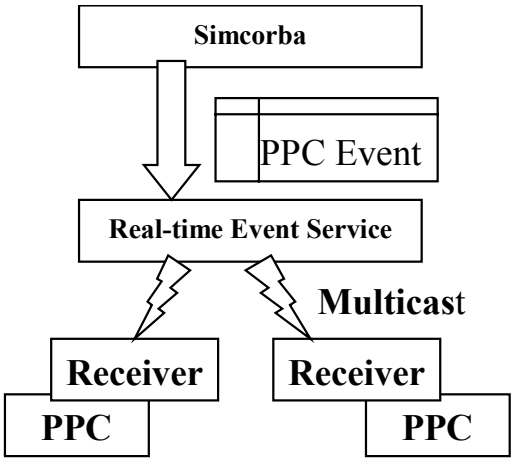


Figura 5.11. Simcorba y diferentes tipos de eventos

5.4.3. El componente *Receiver*

Los componentes *Receiver* tienen como principales objetivos la reutilización y la transparencia de las comunicaciones para las aplicaciones. De esta forma, las aplicaciones que utilizan el componente *Receiver* no conocen nada acerca de la implementación interna de sus componentes receptores. Desde el punto de vista del

cliente, el comportamiento del componente *Receiver* es como un contenedor de datos pasivo con *arrays* que pueden ser consultados y automáticamente actualizados con datos de simulación. Además, el componente *Receiver* ofrece una serie de servicios adicionales como, por ejemplo, la modificación de variables del simulador.

Hay un aspecto importante a resaltar en la utilización del componente *Receiver*. En este caso, dicho componente va a ser compuesto con componentes que se están ejecutando en otras aplicaciones y, además, utilizado directamente en aplicaciones gráficas que no utilizan el modelo UM-RTCOM. Es por ello, que en este caso el componente *Receiver* va a ser utilizado en aplicaciones de tiempo real blando. La Figura 5.12 muestra dos aplicaciones diferentes utilizando el componente *Receiver*.

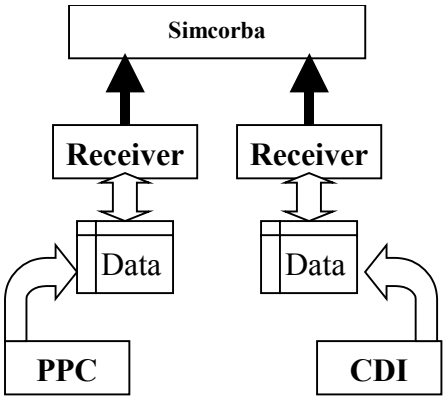


Figura 5.12. Utilización del componente *Receiver*

Fue necesario desarrollar una clase empaquetadora (*wrapper*) para la utilización del componente en aplicaciones gráficas Windows. De esta forma, se ofrece a las aplicaciones una API simple con *arrays* representando a las variables de simulación:

```
class  CReceiver {
  public:
    void initialize(char *Receiver_type,...);
    bool thereisSession();
    void requestallValues();

    float *getAnalogueVariables(long &num);
    unsigned char * getDigitalVariables(long &num);

    void setAnalogueVariables(float *vars,long num);
    void setDigitalVariables(unsigned char *vars,long num);

    void setValues(char *pvalues_list);
};
```

La primera llamada que debe realizarse es `initialize()`, donde se realizan varias tareas relacionadas con la inicialización del componente UM-RTCOM utilizado. Tras la finalización de este método el cliente tiene a su disposición dos *arrays* para las

variables analógicas y las digitales. Estos *arrays* pueden ser manipulados como cualquier otro *array* C/C++ a través de punteros: `float *` para el *array* analógico y `char *` para el *array* digital. Todas las variables necesitadas por los diferentes tipos de clientes están contenidas en este *array*. Adicionalmente, los clientes deben tener información sobre qué variable está contenida en cada posición (un mapa de variables). El siguiente ejemplo muestra de manera básica cómo obtener la variable analógica número 0, que podría representar, por ejemplo, el tiempo de simulación.

```
CReceiver theReceiver;
float *analog_array;
long num_anlgs;

theReceiver.initialize(...);
analog_array = theReceiver.getAnalogueVariables(num_anlgs);

cout << "Current time: " << analog_array[0] << endl;
```

Como puede observarse, los detalles de comunicación permanecen ocultos para la aplicación cliente , que sólo tiene que utilizar *arrays* C/C++ comunes.

5.4.4. Características de tiempo real

En ambas versiones el número de clientes estáticos puede ser conocido cuando *Simcorba* es iniciado. De esta forma, el entorno, con la ayuda del usuario, puede crear recursos suficientes para la gestión de las peticiones, es decir, se pueden crear *thread pools* con un número suficiente de hebras para no penalizar las peticiones de los clientes. Dependiendo también de la plataforma de ejecución, las herramientas podrían aprovechar algunas características de RT-CORBA tales como las conexiones privadas o no multiplexadas.

La versión basada en eventos podría no beneficiarse de algunas de las características anteriores pero, por el contrario, es más fácil de mantener y gran parte del comportamiento temporal es responsabilidad del servicio de eventos utilizado. Por ejemplo, en TAO puede tomar ventaja de las características de *multicast* que ofrece el servicio de eventos de tiempo real, incrementando así el rendimiento de las aplicaciones.

La utilización implícita del modelo propagado por el cliente de RT-CORBA permite que aplicaciones con diferentes prioridades puedan también tener prioridad en

las comunicaciones. Si bien en entornos Windows esto podría tener menos importancia, no obstante, el modelo lo permite, pues podría haber por ejemplo, aplicaciones Unix o Windows CE. La Figura 5.13 muestra un ejemplo de esta situación, donde el tipo *CDI* representa a la *consola del instructor*, cuyas órdenes son más importantes que la actualización de variables en los visualizadores SGI y, por lo tanto, deberían tener una prioridad CORBA superior.

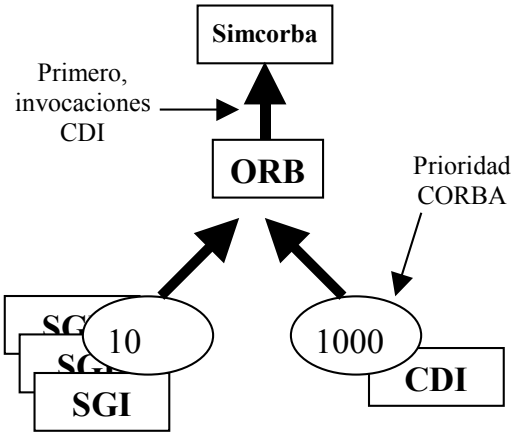


Figura 5.13. *Simcorba/Receiver* y prioridades

Los siguientes gráficos muestran pruebas realizadas con los componentes *Simcorba/Receiver* con la versión basada en servicios. Las pruebas han sido realizadas sobre Irix 6.5 con un cliente CORBA de alta prioridad (32.000) y un cliente CORBA de baja prioridad (10) y con clientes adicionales de baja prioridad, utilizando todos ellos el componente *Receiver*. Los tiempos de respuesta obtenidos muestran la estabilidad en el comportamiento de ambos componentes, *Simcorba* y *Receiver*.

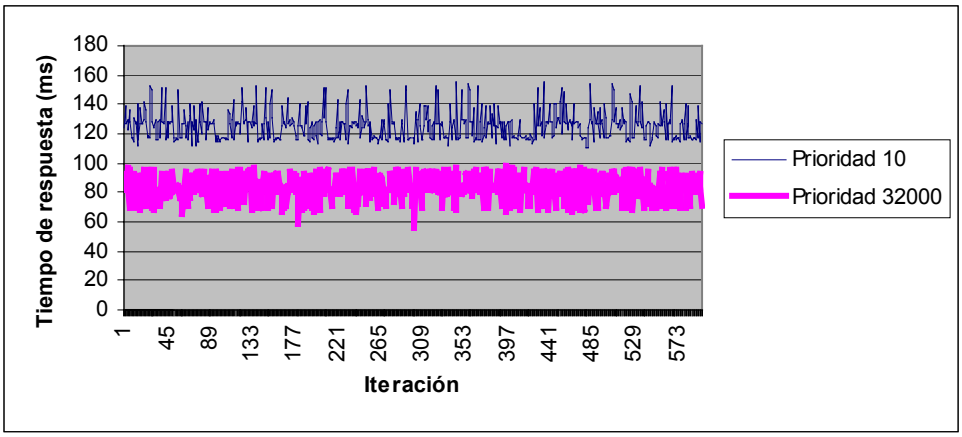


Figura 5.14. Comparación de tiempos de respuesta

El primer gráfico muestra el tiempo de respuesta de dos clientes compitiendo con 9 clientes adicionales con baja prioridad. El gráfico muestra claramente que el tiempo de respuesta del cliente de alta prioridad es más bajo (como cabía esperar), y además muestra la estabilidad de los tiempos de respuesta.

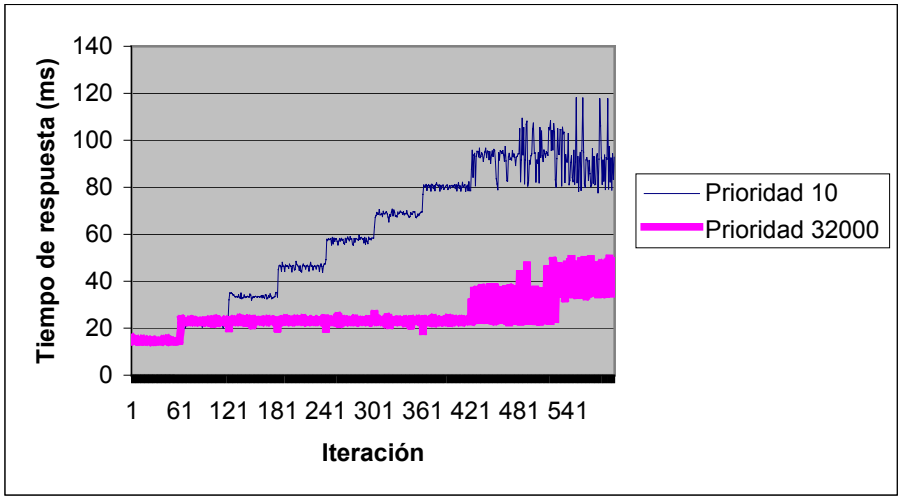


Figura 5.15. Clientes de baja prioridad progresivos

El segundo gráfico muestra la evolución del tiempo de respuesta de ambos clientes cuando se va incrementando el número de clientes de forma progresiva. De esta forma, la prueba comienza con un solo cliente de prioridad CORBA 10 o 32000 y progresivamente se van iniciando nuevos clientes.

Como puede observarse, el tiempo de respuesta del cliente de alta prioridad se mantiene, en oposición al tiempo del cliente de baja prioridad cuyo tiempo de respuesta crece de manera lineal como cabía esperar.

5.5. Aplicaciones y herramientas

Las aplicaciones, herramientas y subsistemas son diferentes en cada simulador y su integración podría ser difícil. Además, puede surgir durante el desarrollo de un proyecto, la necesidad de desarrollar nuevas herramientas no contempladas inicialmente. La utilización del componente *Receiver* facilita la tarea de integración de aplicaciones.

En esta sección se muestran algunos ejemplos de aplicaciones y subsistemas junto con cuestiones relacionadas con tiempo real.

5.5.1. Visualizadores SGI

Los *visualizadores SGI (SGI Displayer)* permiten el entrenamiento de los operadores en una sala aislada del simulador de alcance total. Un visualizador *SGI* permite inspeccionar láminas gráficas con los diferentes componentes existentes en la sala de control. Así, por ejemplo, los estudiantes pueden realizar acciones tales como apertura o cierre de válvulas, reconocimiento de alarmas, etc. Además, las acciones realizadas por los estudiantes tienen su repercusión en el estado global de la sesión de simulación, afectando a otros estudiantes.

En la Figura 5.16 puede verse un ejemplo del tipo de aplicaciones gráficas que pueden utilizarse en estos visualizadores, con la manipulación de una lámina gráfica representando elementos físicos.

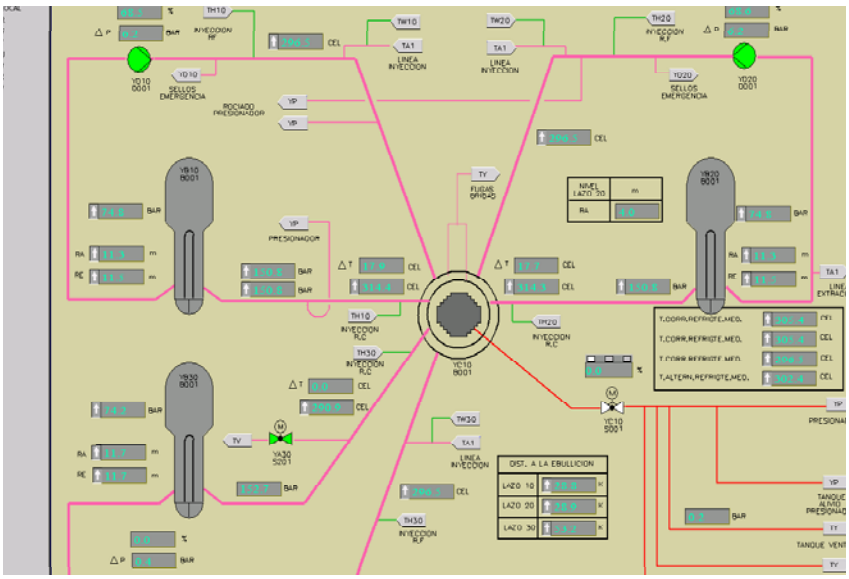


Figura 5.16. Ejemplo de aplicación SGI

Los visualizadores están organizados en una jerarquía instructor/estudiantes, donde el instructor puede realizar las mismas acciones que los estudiantes y, adicionalmente, otras acciones especiales relativas la gestión de la sesión e simulación. Ejemplos de tales acciones son el control de estudiantes conectados, modificación de variables especiales, etc.

La Figura 5.17 muestra un ejemplo de sesión de simulación con los puestos *SGI* utilizando el componente *Receiver*. Cada puesto *SGI* utiliza un componente *Receiver* para la comunicación con el simulador.

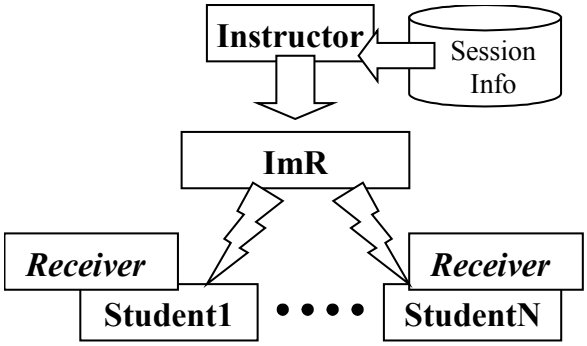


Figura 5.17. Visualizadores SGI

El software de comunicaciones de los visualizadores está organizado en librerías dinámicas que utilizan de forma transparente el componente *Receiver*. El instructor ofrece servicios adicionales tales como la conexión a sesiones de simulación iniciadas, información sobre láminas, etc.

Las características de tiempo real en este caso están relacionadas con el componente *Receiver*, el cual tiene que recibir datos cada 250 ms. No es crucial la recepción en ese tiempo límite, pero si esto no ocurriera así, la aplicación no sería cómoda de utilizar, con retardos en la visualización, las acciones de usuario, etc., llevando incluso a cometer errores a los usuarios.

5.5.2. Computador de Procesos de Planta

El denominado *PPC, Plant Process Computer* o *Computador de Procesos de Planta* es un ejemplo de subsistema propio de un simulador específico. Es además, ejemplo de un gran subsistema integrado en un simulador.

El principal propósito del *PPC* es informar sobre el estado de la planta en la sala de control a través de alarmas, informes impresos, gráficos, etc. Algunos componentes del PPC han sido replicados en hardware, otros en software, y algunos tanto en hardware como en software.

El subsistema *PPC* está compuesto de varios paneles físicos, ordenadores controlando el estado de la planta (cada ordenador es responsable de diferentes subsistemas de la planta), pantallas (al menos 10) y teclados asociados a las pantallas. Cada pantalla puede ser utilizada en diferentes modos de ejecución que muestran alarmas, gráficos de barras, valores de variables, etc.

La Figura 5.18 muestra un ejemplo del tipo de componentes físicos con los que tiene que tratar el *PPC*. Concretamente, es un teclado software replicando completamente el funcionamiento de un teclado hardware existente en los paneles físicos.

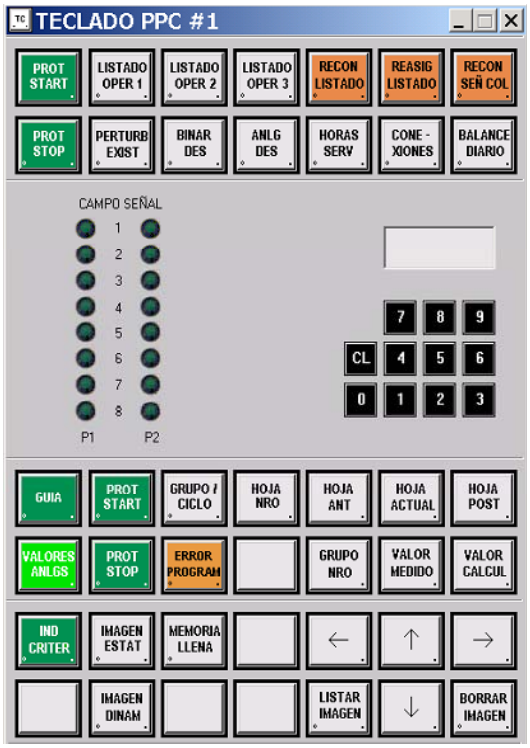


Figura 5.18. Teclado PPC

La Figura 5.19 muestra parte de la arquitectura de las comunicaciones del *PPC* con los ordenadores simulados que utilizan componentes receptores para recibir los valores de las variables de simulación, las "pantallas" recuperando datos de los ordenadores, los teclados y el núcleo del PPC dividido en diferentes subsistemas.

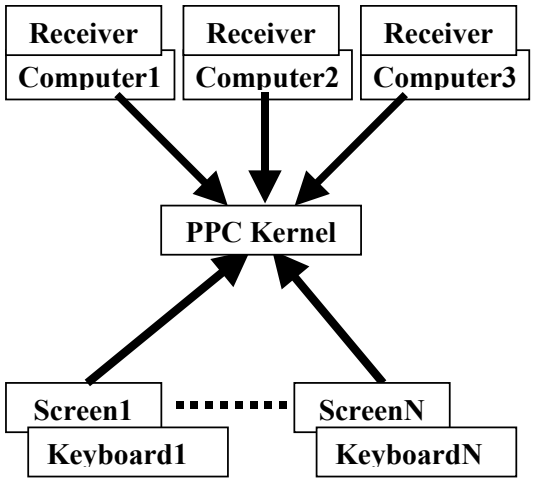


Figura 5.19. *Plant Process Computer*

En esta aplicación sí han podido ser utilizados otros componentes UM-RTCOM además de los componentes *Receiver*. En concreto, las comunicaciones entre los ordenadores, el núcleo del PPC y las pantallas/teclados han sido también desarrolladas con componentes de UM-RTCOM encargados de la comunicación entre todos estos elementos del *PPC*.

Se han utilizado *thread pools* para manejar las peticiones de estos elementos y la mayor prioridad se ha dado a los ordenadores simulados, puesto que de estos depende el buen funcionamiento del resto de elementos del PPC. La prioridad más baja ha sido dada a los componentes asociados a las pantallas.

Periódicamente (al menos cada 250 ms.) los ordenadores comprueban la existencia de nuevas alarmas, proporcionándolas al núcleo del PPC. Las alarmas, como el resto de elementos de los simuladores, fueron transmitidas a través de variables de simulación, y en concreto, a través de variables digitales.

El núcleo del PPC es responsable de procesar estas alarmas, informando de ello a las pantallas en un formato adecuado para ellas. Las pantallas recuperan las alarmas y, dependiendo del modo de visualización, los datos son mostrados de manera específica.

Hay que tratar también acciones aperiódicas (eventos) representadas por las acciones de usuario. Estas acciones son procesadas y recibidas en el núcleo del PPC. Para evitar interferencias hay que utilizar también zonas en exclusión mutua (componentes pasivos), por ejemplo, para la gestión de las listas de alarmas.

5.5.3. MINIDESI

Los dispositivos tipo PDA (*Personal Digital Assistants*) presentan grandes posibilidades de utilización sólo limitadas por la imaginación de los desarrolladores. Son dispositivos pequeños, baratos, con una cantidad de recursos aceptable, y dotados de las últimas tecnologías tanto en hardware como en software. Si a todo ello, añadimos su utilización en redes inalámbricas, tenemos la posibilidad de generar toda una nueva gama de aplicaciones.

MINIDESI [Díaz et al, 2005b] es una aplicación utilizada para facilitar la labor de docencia del instructor de las sesiones de simulación al facilitarle el desplazamiento y actuación en los simuladores de alcance total. MINIDESI es una versión reducida de DESI que incorpora las funcionalidades más importantes, tales como, monitorización de variables o funciones de control.

La aplicación desarrollada ha utilizado el modelo UM-RTCOM, integrándose fácilmente en el resto del entorno de simulación. Si en los casos anteriores, las aplicaciones eran bastante "pesadas", en esta ocasión se trata de una aplicación implementada sobre dispositivos con pocos recursos, por lo que se ha utilizado internamente la implementación basada en ROFES junto con el sistema operativo Windows CE.

La principal motivación para la realización de la herramienta MINIDESI es la de facilitar la labor de docencia del instructor, ya que va a permitir al instructor el no estar en un lugar en concreto como es la CDI, sino que, al utilizar redes inalámbricas va a poder desplazarse cómodamente por todo el simulador, actuando de esta forma directamente en los puestos de los estudiantes, manipulando variables, controlando la simulación, etc.

Entorno de desarrollo

El entorno de desarrollo para Pocket PC que se ha utilizado es Embedded Visual Studio de Microsoft, entorno de desarrollo similar a Visual Studio pero adaptado a PDAs. En Embedded Visual C++ es posible desarrollar aplicaciones documento/vista, basadas en diálogos, MFC, controles ActiveX. Además, está soportado por herramientas que, por ejemplo, en el caso de sistemas en tiempo real, permiten monitorizar las acciones realizadas por el núcleo del sistema operativo. Con respecto al sistema operativo, se ha utilizado Windows CE, el cual tiene características por las que puede considerarse realmente como un sistema operativo de tiempo real, con suficientes niveles de prioridad, control de inversión de prioridades, etc.

Descripción de la herramienta

MINIDESI es fundamentalmente un depurador de variables que además incorpora funciones de control de la simulación. El aspecto de MINIDESI es el de un diálogo con capacidad para monitorizar 10 variables y 6 páginas. Es decir, se pueden monitorizar simultáneamente hasta 60 variables. La Figura 5.20 muestra la ventana principal de MINIDESI monitorizando variables. El usuario puede escribir el nombre de variables a consultar/modificar. El valor de estas variables es consultado al núcleo del simulador y actualizado periódicamente (4 veces por segundo) en MINIDESI.

Si el usuario así lo desea, también podrá realizar modificaciones sobre variables. Estas modificaciones son enviadas al núcleo del simulador y tienen su repercusión sobre la simulación global y, por ejemplo, pueden ser observadas por DESI. Acciones tales como la rotura de tuberías, pueden ser simuladas fácilmente con la modificación de unas cuantas variables.

MINIDESI también puede ser utilizado para controlar la sesión de simulación tanto para su inicio/parada (botón Run/Freeze) como para modificar el funcionamiento de la propia simulación (ej.: velocidad, ejecución continua o por pasos, etc.).

Figura 5.20. Aplicación MINIDESI

Detalles de las comunicaciones

La aplicación MINIDESI enlaza dinámicamente con una librería dinámica generada por UM-RTCOM. La comunicación de MINIDESI con el resto del entorno de simulación se basa en la misma interfaz que es compartida con el núcleo del simulador.

El desarrollo de las comunicaciones utilizando ROFES y Pocket PC no ha tenido mayores inconvenientes que los que pueda tener una aplicación en Visual C++ normal. Los mayores problemas han venido dados por el propio ROFES, encontrando algunos fallos en el código generado que producían problemas de compilación.

La interoperabilidad entre ROFES y TAO en el servidor de simulación se efectuó sin mayores problemas, pudiendo pues integrar a MINIDESI como una herramienta más.

5.5.4. Nuevas aplicaciones

Uno de los principales objetivos del paradigma de programación basado en componentes es la reutilización. En este sentido, surgió la necesidad de hacer nuevas herramientas en los simuladores que no se habían contemplado inicialmente. La utilización del modelo UM-RTCOM permitió la incorporación de estas herramientas utilizando los componentes ya desarrollados, sin tener que realizar modificaciones en el resto del entorno de simulación. Los componentes *Simcorba* y *Receiver* cumplieron su propósito, permitiendo que las nuevas herramientas se conectaran con el núcleo del simulador, pudiendo tanto recibir variables como modificar sus valores. La Figura 5.21 y la Figura 5.22 muestran la aplicación *Javi* en diferentes situaciones de uso.

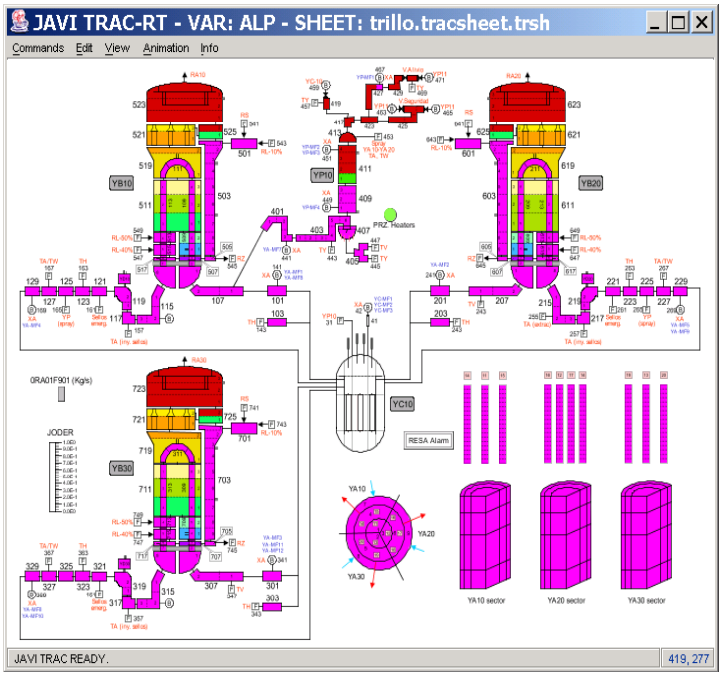


Figura 5.21. Visualización de una lámina en *Javi*

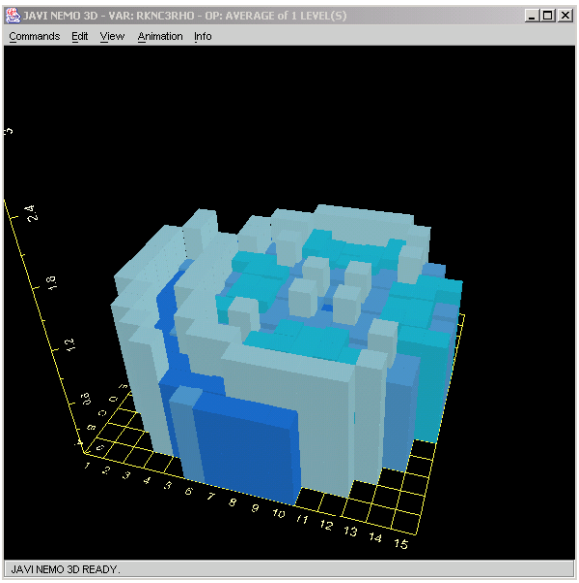


Figura 5.22. Gráficos 3D en *Javi*

La aplicación *Javi* se encarga de visualizar el estado global de la planta mediante láminas gráficas con un código de colores que permite tener una visión rápida y global de los valores de las variables de simulación, permitiendo detectar anomalías.

5.6. Análisis de tiempo real

Las técnicas de análisis propuestas han sido aplicadas a un subconjunto de las comunicaciones del simulador. En concreto, los elementos analizados incluyen la comunicación de *SETRUSimcorba* con los modelos de simulación (en el ejemplo, el modelo TRAC) y algunas aplicaciones como, por ejemplo, la consola del instructor y los visualizadores *SGI* (se han utilizado 3 visualizadores).

Utilizando términos CORBA, en los que se basa el análisis, *SETRUSimcorba* es un servidor conteniendo a dos objetos CORBA (*SETRU* y *Simcorba)*, proporcionando las interfaces mostradas en secciones anteriores (5.3.2. y 5.4.1.) con métodos para consultar las variables de simulación, actualizarlas, enviar comandos al simulador y recuperar el estado del mismo. Las otras aplicaciones contienen clientes CORBA utilizando estos servicios. En estos clientes, la prioridad más alta es para los modelos de simulación, responsables de actualizar las variables de simulación. La consola del instructor tiene una prioridad intermedia para el envío de comandos, y la prioridad más baja es para las aplicaciones visualizadoras.

La realización del análisis comienza con la obtención del modelado en SDL de las aplicaciones. Gracias a las herramientas del entorno, se puede extraer dicho modelo y complementarlo con el modelado SDL de las partes con RT-CORBA de las diferentes aplicaciones. Estos dos elementos modelados en SDL, aplicaciones y RT-CORBA son completados con la obtención de los peores tiempos de ejecución de los diferentes bloques secuenciales que componen las aplicaciones. Posteriormente, la herramienta de análisis obtiene los tiempos de respuesta de los diferentes eventos que componen el sistema, utilizando estos modelos SDL junto con la información temporal.

5.6.1. Modelo SDL

Siguiendo las reglas de la transformación a SDL, la Figura 5.23 muestra una visión global del modelo SDL del sistema. Existe un bloque SDL por cada aplicación, con el modelo SDL extraído por las herramientas, y otro bloque RT-CORBA adicional para las comunicaciones entre las diferentes partes del sistema.

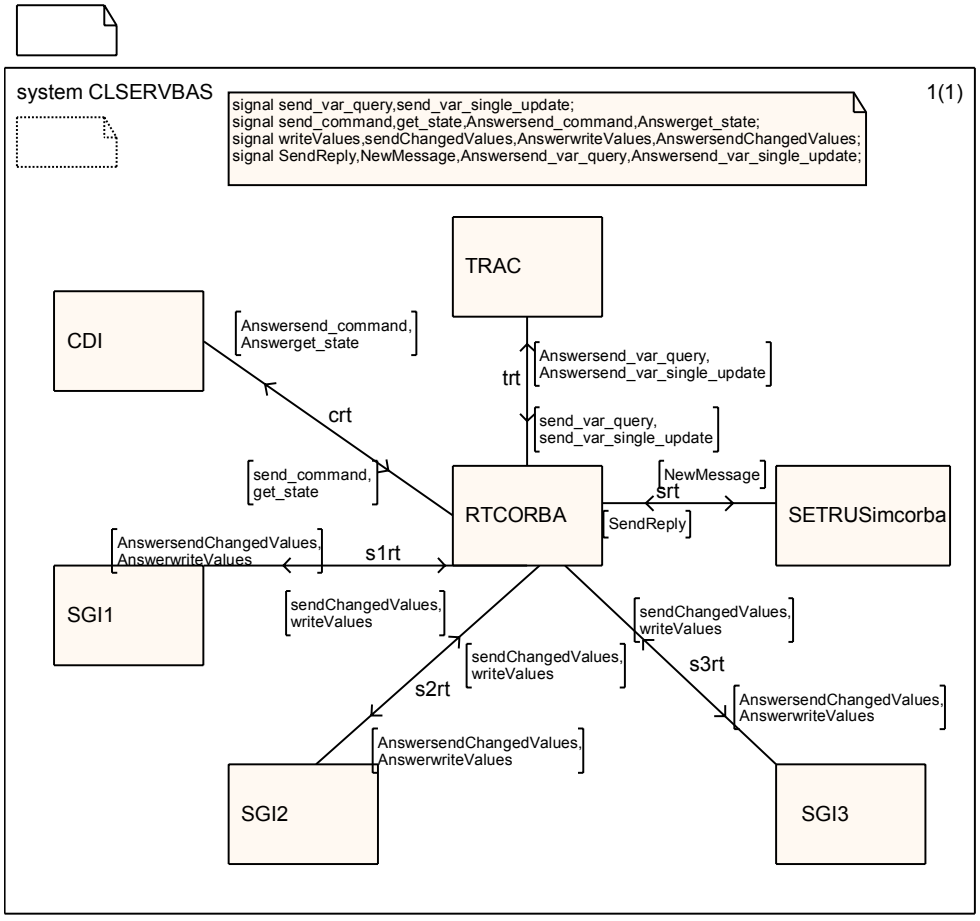


Figura 5.23. Modelado en SDL del simulador

La Figura 5.24 muestra parte del bloque `TRAC` con los diferentes procesos resultantes de la transformación de los activos y pasivos del componente `TRAC`.

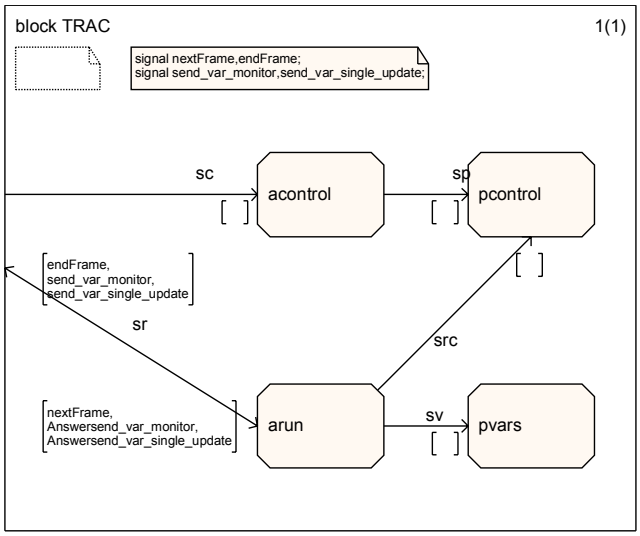


Figura 5.24. Modelado en SDL de TRAC

En la Figura 5.25 puede verse parte del diagrama SDL relativo al proceso `acontrol` con las transiciones necesarias para esperar la generación del evento `nextFrame`, solicitar variables a *SETRU*, ejecutar el paso de simulación y finalmente actualizar variables.

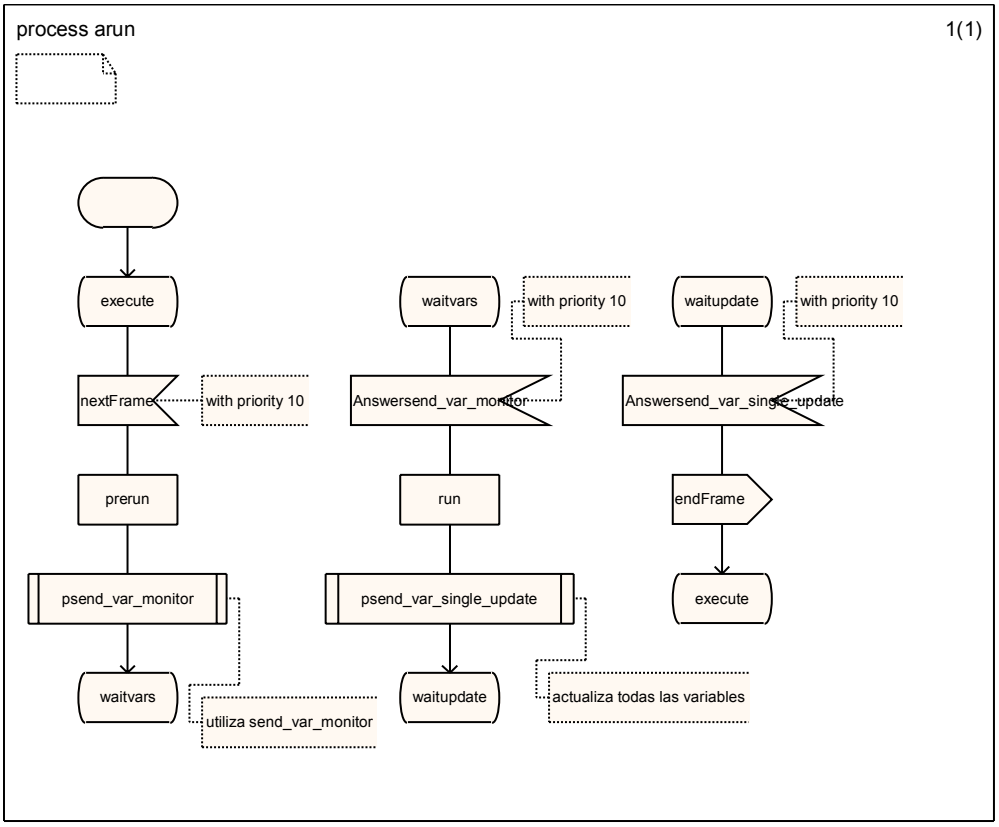


Figura 5.25. Proceso SDL `acontrol`

La Figura 5.26 muestra el contenido del bloque RT-CORBA con los diferentes *stubs* de las aplicaciones, el bloque del servidor `SETRUSimcorba`, el bloque GIOP y la plataforma de comunicaciones.

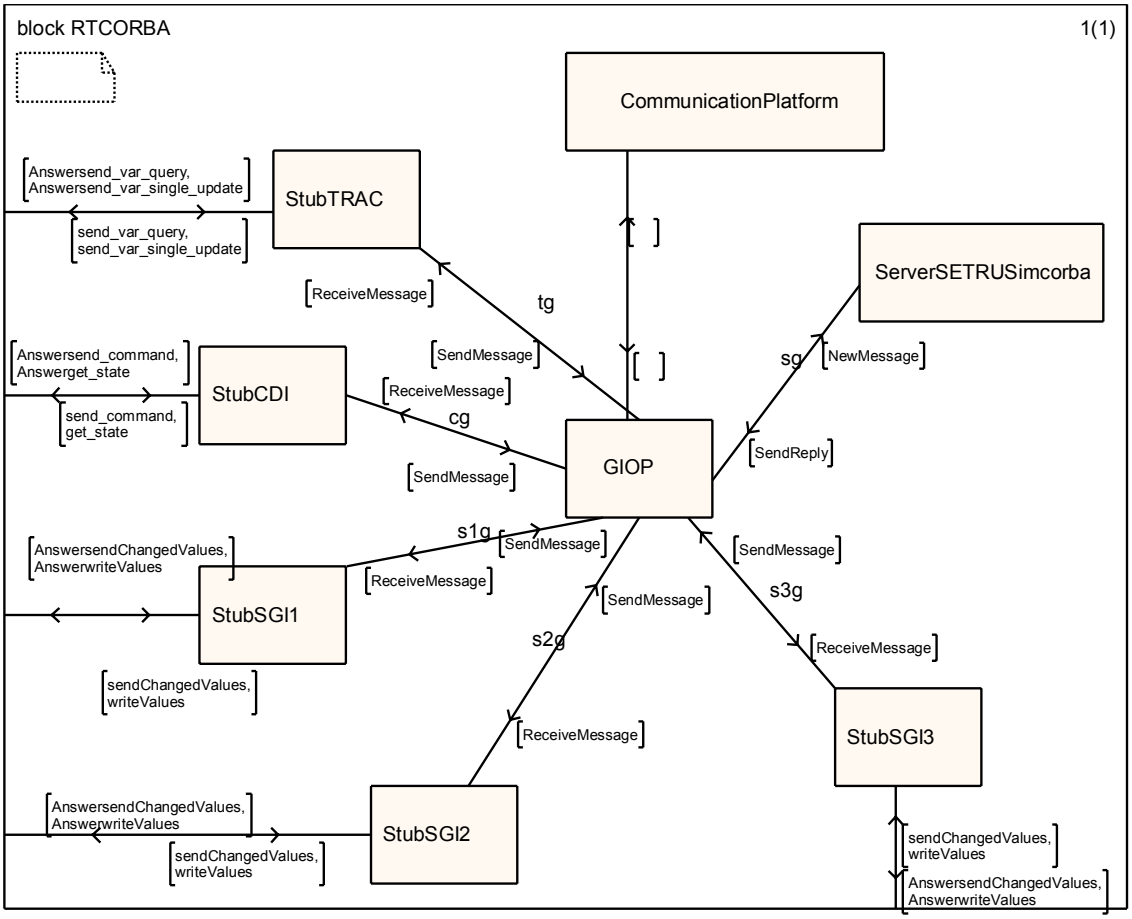


Figura 5.26. Modelado SDL del bloque RTCORBA

La Figura 5.27 muestra los servicios utilizados por el visualizador 1 con los dos procesos contenidos en su *stub*: `ServicesendChangedValues` y `ServicewriteValues`.

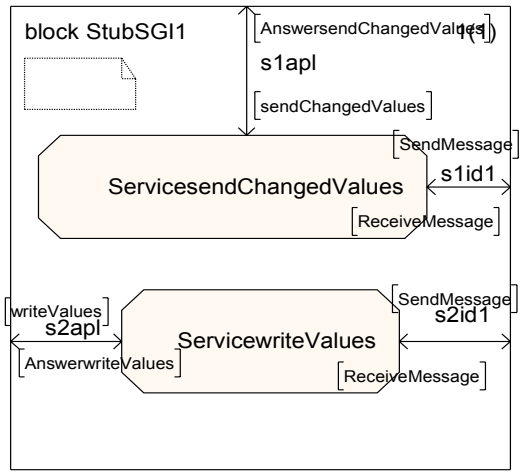


Figura 5.27. *Stub* de SGI1

La Figura 5.28 muestra el comportamiento del proceso `ServicesendChangedValues` con la recepción de peticiones y el envío de respuestas a los clientes.

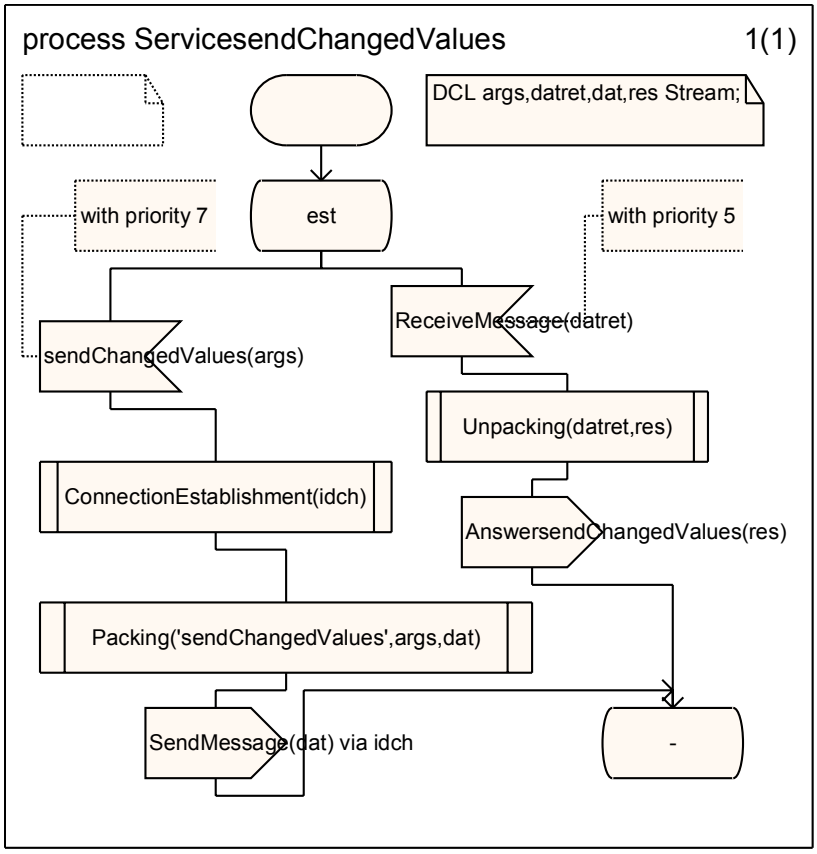


Figura 5.28. Proceso `ServicesendChangedValues`

Finalmente, todos estos bloques, procesos, señales, etc. van a poder ser combinados en un modelo "plano" SDL que puede ser analizado por las herramientas de análisis obteniendo los peores tiempos de respuesta de los diferentes eventos que pueden ocurrir en el sistema.

5.6.2. Eventos, tareas y tiempos de ejecución

Para realizar el análisis del peor tiempo de respuesta hay que considerar los eventos que se van a producir en el sistema. En el sistema analizado, el modelo TRAC puede consultar y modificar variables. La consola del instructor puede enviar comandos al simulador y recuperar el estado de la simulación, y los visualizadores SGI, interactuando a través de *Simcorba*, puede obtener las variables de simulación y

actualizar valores de las mismas. Los eventos que pueden producirse en el sistema son los siguientes:

- Eventos asociados a TRAC: el evento `nextFrame` desencadena la consulta de las variables necesitadas, la ejecución de un paso y la actualización de las variables.
- Eventos asociados a la CDI: el instructor de la sesión de simulación puede lanzar de forma aperiódica comandos que deben ser tratados por el simulador. Asimismo, periódicamente se consulta el estado del simulador.
- Eventos asociados a los visualizadores *SGI*: las pantallas de la aplicación son actualizadas periódicamente mediante la consulta de variables. Asimismo y de forma aperiódica, el usuario puede realizar acciones que modifican el valor de algunas variables de simulación.

La siguiente tabla muestra estos eventos junto con sus requisitos temporales (*deadlines*). En los visualizadores *SGI*, los eventos son diferenciados para cada uno.

Evento	Deadline (ms)
ev_nextFrame	125
ev_get_state	250
ev_command	350
ev_update_screen1	250
ev_user_action1	350
ev_update_screen2	250
ev_user_action2	350
ev_update_screen3	250
ev_user_action3	350

Tabla 1 Eventos del simulador

La siguiente tabla muestra los métodos que son utilizados por cada uno de los componentes/aplicaciones.

Componente/Aplicación	Métodos utilizados
TRAC	`SETRU::ISimVars::send_var_query` `SETRU::ISimVars::send_var_single_update`
CDI	`SETRU::ISimControl::send_command` `SETRU::ISimControl::get_state`
SGI1	`Simcorba::ISimulator::sendChangedValues` `Simcorba::Isimulator::writeValues`
SGI2	`Simcorba::ISimulator::sendChangedValues` `Simcorba::Isimulator::writeValues`
SGI3	`Simcorba::ISimulator::sendChangedValues` `Simcorba::Isimulator::writeValues`

Tabla 2 Métodos CORBA utilizados

A continuación se muestran los peores tiempos de ejecución de las diferentes tareas asociadas a los eventos obtenidos tras la utilización de las herramientas de prueba del sistema. *Simcorba/SETRU* ha sido ejecutado sobre el sistema operativo Irix 6.5 con aplicaciones sobre Windows y redes dedicadas, siendo ésta la plataforma real de ejecución de los simuladores.

Las tablas muestran las diferentes tareas para cada evento, el tiempo local empleado así como el tiempo de ejecución remoto en el servidor. En estos dos últimos casos, los tiempos incluyen también la sobrecarga debida a los diferentes elementos de RT-CORBA en cada uno de ellos, es decir, *stub*, GIOP, POA, *skeleton* y la respuesta. Todos los tiempos son indicados en milisegundos. Las tablas para los tres visualizadores *SGI* han sido unificadas.

ev_nextFrame	T. local (ms)	T. remoto (ms)
Preparación paso de simulación	2	-
Consulta variables: `send_var_monitor`	2.21	2.29
Ejecución de paso de simulación: `run`	50	-
Actualización de variables en SETRU: `send_var_single_update`	21.3	21.7

Tabla 3 Tareas y tiempos para ev_nextFrame

ev_get_state	T. local (ms)	T. remoto (ms)
Obtención estado del simulador: `get_`state	0.12	0.13
Actualización estado aplicación	12	-

Tabla 4 Tareas y tiempos para ev_get_state

ev_command	T. local (ms)	T. remoto (ms)
Realización comando	5	-
Envío comando al simulador: `send_`command	0.23	0.37

Tabla 5 Tareas y tiempos para ev_command

ev_update_screenX	T. local (ms)	T. remoto (ms)
Preparación actualización	5	-
Envío comando al simulador: `sendChangedValues`	1.8	4.2
Actualización pantalla	15	-

Tabla 6 Tareas y tiempos para ev_update_screenX

ev_user_actionX	T. local (ms)	T. remoto (ms)
Realización acción de usuario	6	-
Envío comando al simulador: `writeValues`	0.22	0.23
Actualización pantalla	15	-

Tabla 7 Tareas y tiempos para ev_user_actionX

5.6.3. Tiempos de respuesta

Los modelos SDL anotados con la información temporal pueden ser utilizados por la herramienta de análisis para la obtención de los peores tiempos de respuesta de los diferentes eventos del sistema, comprobando así su planificabilidad.

La siguiente tabla muestra la prioridad de los diferentes eventos del sistema. La prioridad de las tareas asociadas a cada evento va a ser la misma que la de dicho evento.

Evento	Prioridad
ev_nextFrame	10
ev_get_state	5
ev_command	8
ev_update_screen1	5
ev_user_action1	7
ev_update_screen2	5
ev_user_action2	7
ev_update_screen3	5
ev_user_action3	7

Tabla 8 Prioridad de los eventos del simulador

La Figura 5.29 muestra la utilización de la herramienta de análisis con los eventos y tareas descritos anteriormente. En la parte izquierda puede verse parte del

modelado SDL etiquetado con tiempos, periodos y prioridades. En la parte derecha pueden verse los eventos y las relaciones de precedencia entre las diferentes tareas.

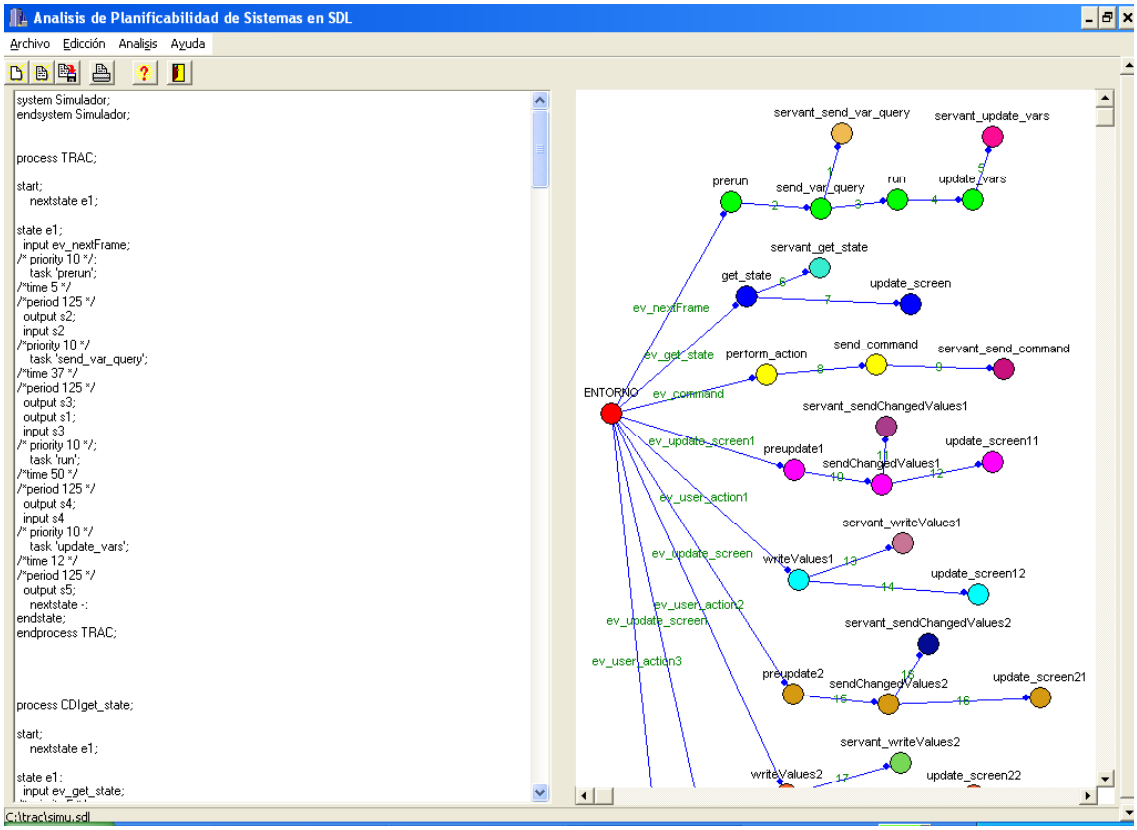


Figura 5.29. Herramienta de análisis con el simulador

Evento	Deadline (ms)	R_{local} (ms)	R_{remoto} (ms)	R_{total} (ms)
ev_nextFrame	125	75.51	47.98	123.49
ev_get_state	250	22.58	37.78	60.36
ev_command	350	5.23	24.36	29.59
ev_update_screen1	250	40.46	37.78	78.24
ev_user_action1	350	21.22	25.05	46.27
ev_update_screen2	250	40.46	37.78	78.24
ev_user_action2	350	21.22	25.05	46.27
ev_update_screen3	250	40.46	37.78	78.24
ev_user_action3	350	21.22	25.05	46.27

Tabla 9 Tiempos de respuesta del simulador

Tras la utilización de la herramienta de análisis, se obtienen los siguientes resultados de la Tabla 9, mostrando la planificabilidad del sistema.

Capítulo 6. Conclusiones

El principal objetivo de esta tesis es la creación de un entorno completo para el desarrollo, despliegue y análisis de sistemas distribuidos de tiempo real. En todo momento, las aportaciones realizadas han intentado estar en línea con las aportaciones más recientes de la Ingeniería del Software.

Dentro de las últimas novedades en el campo de los lenguajes de programación y sus paradigmas, se decidió utilizar la tecnología basada en componentes como base para la implementación de sistemas de tiempo real. La tecnología de componentes es uno de los últimos paradigmas (junto con el de los "aspectos") utilizado para el desarrollo de sistemas software y que intenta superar las limitaciones de paradigmas anteriores como, por ejemplo, el paradigma basado en objetos. La utilización de los componentes parece una alternativa bastante interesante por las propiedades que éstos tienen: independencia, reutilización, dinamismo, etc.

Los modelos convencionales de componentes presentan dificultades para su utilización en sistemas de tiempo real. Por ello, se ha presentado un nuevo modelo, UM-RTCOM, que permite la creación de componentes software reutilizables para sistemas de tiempo real. Las principales aportaciones de este modelo de componentes están relacionadas con este tipo de sistemas y sus necesidades: múltiples interfaces, ranuras de configuración, comportamientos activos y pasivos, gestión de recursos, posibilidad de configuración, etc. El modelo debe también poder ser utilizado conjuntamente con la metodología de análisis propuesta.

El desarrollo del modelo de componentes conlleva también la utilización de una serie de herramientas, tales como un editor de componentes, herramientas de prueba, despliegue, etc. Las herramientas y el modelo de componentes están muy interrelacionados entre sí, y también con las otras fases en el ciclo de vida de las

aplicaciones. Así, por ejemplo, los resultados obtenidos por la herramienta de prueba, pueden ser utilizados por la herramienta de análisis.

Para la infraestructura de ejecución, se escogió la utilización de RT-CORBA. CORBA, por sí mismo, es una poderosa tecnología para el desarrollo de todo tipo de aplicaciones distribuidas con independencia de lenguaje, sistema operativo y plataforma. No obstante, y como se ha comentado, ha sido necesario utilizar la extensión de tiempo real de CORBA. También habría sido muy interesante la utilización del modelo de componentes CCM de CORBA, pero, el estado de desarrollo actual desaconsejó su utilización, puesto que se querían utilizar las propuestas realizadas en aplicaciones reales y que debían tener cierta robustez. En concreto, se han utilizado dos implementaciones de RT-CORBA, TAO y ROFES, adecuadas para las aplicaciones realizadas. TAO ha sido utilizado para las aplicaciones con mayores requisitos y donde la estabilidad debía ser mayor. ROFES, por su parte, ha sido utilizado en aplicaciones no críticas en sistemas empotrados. La utilización de RT-CORBA para la implementación del modelo de componentes permite beneficiarse de todas las ventajas de CORBA y además obtener la predecibilidad que RT-CORBA aporta, permitiendo así la realización posterior del análisis de tiempo real.

La implementación del modelo de componentes en RT-CORBA pasa por la realización de una transformación desde el modelo a RT-CORBA, teniendo en cuenta también la metodología de análisis que se va a utilizar.

Todos los elementos anteriores podrían no ser útiles en los sistemas de tiempo real si se carece de una metodología que permita estudiar el comportamiento temporal de estos sistemas y saber si un sistema puede ser planificable. En este sentido, se decidió la utilización de los trabajos previos realizados en la tesis de Luis Llopis en el grupo GISUM de la Universidad de Málaga. Las propuestas realizadas en esta tesis utilizaban la Técnica de Descripción Formal SDL, como la base para la realización de análisis de planificabilidad para sistemas monoprocesador. En la presente tesis se ha realizado una ampliación de las propuestas realizadas para permitir su utilización en sistemas distribuidos con RT-CORBA como plataforma de ejecución. Se han presentado las ecuaciones que permiten la obtención del tiempo de respuesta en presencia de invocaciones remotas.

Por último, todas las propuestas realizadas se han utilizado en el marco de sistemas software complejos como son los simuladores para centrales nucleares. El grupo GISUM ha venido colaborando desde hace años con la empresa Tecnatom S.A.

en la realización de software para estos simuladores. Este tipo de simuladores presenta características de tiempo real y, dada la experiencia previa del grupo en su desarrollo, pareció un campo de especial interés donde aplicar las propuestas. En concreto, se realizó un estudio sobre la parte de comunicaciones de los simuladores, realizando la componentización de ciertos elementos, con la utilización de RT-CORBA y la metodología de análisis propuesta.

Para que un modelo de componentes sea realmente útil, los componentes desarrollados deben ser reutilizables. En este sentido, el modelo UM-RTCOM ha cumplido las expectativas. Tal y como se ha mostrado con la realización de los simuladores, surgió la necesidad de realizar nuevas aplicaciones no contempladas inicialmente. El modelo permitió la reutilización de los componentes ya desarrollados, integrándose las nuevas aplicaciones sin mayores dificultades.

La utilización de las propuestas realizadas ha sido satisfactoria. Se ha definido un modelo de componentes, ejecutándose sobre una plataforma predecible y con una metodología de análisis para tiempo real, por lo que el principal objetivo de la tesis está cubierto. Como trabajos futuros, aparecen dos nuevos puntos de referencia: la utilización del paradigma orientado a aspectos y los sistemas *peer-to-peer*.

Los aspectos software intentan extraer el código que va más allá de un componente y que aparece "mezclado" en los diferentes componentes que forman parte de un sistema. De esta forma, se obtiene un código más limpio y fácil de mantener, disminuyendo también los errores. Pueden existir muy diferentes aspectos en un sistema. Así, por ejemplo, para los sistemas de tiempo real, la "planificación" es algo que va más allá de un simple componente, siendo algo compartido por todos los componentes que forman una aplicación. Sería interesante el estudio de la posible aplicación de los aspectos en el campo de los sistemas de tiempo real y en combinación con los componentes del modelo UM-RTCOM. No obstante, los aspectos suelen estar asociados a código dinámico, aspecto este que puede dificultar la predecibilidad y consecuentemente, el análisis de tiempo real.

El segundo campo de estudio puede ser el de los sistemas *peer-to-peer (P2P)*. En este tipo de sistemas, muy en auge hoy en día gracias a Internet, el concepto de *clientes* y *servidores* existentes en otros paradigmas desaparece. En los sistemas P2P todos los elementos están al mismo nivel jerárquico, cobrando esto especial importancia en el campo de las redes *ad hoc*. Este tipo de redes carecen de una infraestructura y son establecidas de manera dinámica con los elementos existentes en un momento dado.

Estos elementos, sin ayuda externa tienen que ponerse de acuerdo en muy diferentes aspectos: seguridad, identificación, recursos, etc. Sería muy interesante como trabajo futuro, estudiar la posible implementación del modelo UM-RTCOM sobre este tipo de redes como plataforma de ejecución, en vez de la utilización de RT-CORBA. Sin embargo, las peculiaridades de este tipo de sistemas hacen especialmente difícil la realización del análisis de tiempo real.

Bibliografía

[Abdelzaher et al., 1997] T. Abdelzaher, S. Dawson, W. Feng, S. Ghosh, F. Jahanian, S. Johnson, A. Mehra, T. Mitton, J. Norton, A. Shaikh, K. Shin, V. Vaidyan, Z. Wang y H. Zou. "ARMADA Middleware Suite". Proceedings of the Workshop on Middleware for Real-Time Systems and Services, IEEE, 1997.

[Aldea y González, 2000] M. Aldea y M. González. "Early Experience with an Implementation of the POSIX.13 Minimal Real-Time Operating System for Embedded Applications". Proceedings Workshop on Real-Time Programming, pp. 37–45, Elsevier, 2000.

[Alvarez et al, 1996] J.M. Alvarez, M. Díaz, L. Llopis, F. Rus y E. Soler. "Practical parallelization strategies of a thermohydraulic code". Proceedings of Euroconference in Supercomputation in Non Linear and Disordered Systems, pp. 254-258, 1996.

[Alvarez et al., 1999] J.M Alvarez, M. Díaz, L. Llopis, E. Pimentel y J.M. Troya. "Integrating Schedulability Analysis and SDL in an Object-Oriented Methodology". Proceedings 9th SDL Forum, pp. 241-256, 1999.

[AspectJ, 2005] The AspectJ Programming Guide, Xerox Corporation, http://eclipse.org/aspectj/doc/released/progguide/index.html, 2005.

[Birrell y Nelson, 1984] A. Birrell y B.J. Nelson. "Implementing remote procedure" calls". ACM Transactions on Computer Systems, 2(1), pp. 39-59, 1984.

[Bollella et al., 2000] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin y M. Turnbull. Real-time Specification for Java. Addisson Wesley, 2000.

[Box, 1997] D. Box. Essential COM. Addison-Wesley, Reading, 1997.

[Brinkschulte et al., 2002] U. Brinkschulte, A. Bechina, F. Picioroaga y E. Schneider."Distributed Real-Time Computing for Microcontrollers - the OSA+ Approach". Proceedings of the 5th IEEE International Symposium on Object-oriented Real-time Distributed Computing, pp. 169-172, 2002.

[Chimera, 2005] Sistema Operativo Chimera.
http://www-2.cs.cmu.edu/~aml/chimera/chimera.html, 2005.

[Díaz y Garrido, 2003] M. Díaz y D. Garrido. "SGI-SAT Trillo: A Full Scope Simulator for Nuclear Power Plants". 5th International Symposium on Distributed Objects and

Applications, Lecture Notes on Computer Science 2889, Springer Verlag, pp. 7-10, 2003.

[Díaz et al., 2004] M. Díaz, D. Garrido, L. Llopis, F. Rus y J.M. Troya. "Integrating Real-time Analysis in a Component Model for Embedded Systems". Proceedings 30th EUROMICRO Conference (IEEE), pp. 14-21, 2004..

[Díaz et al., 2004b] M. Díaz, D. Garrido y J.M. Troya. "Real-time Training Simulators Based on Distributed Reusable Components". International Journal of Simulation Systems, Science & Technology, vol. 5, num. 3-4, pp. 1-11, 2004.

[Díaz y Garrido, 2004] M. Díaz y D. Garrido. "Applying RT-CORBA in Nuclear Power Plant Simulators". Proceedings of the 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, pp. 7-14, 2004.

[Díaz y Garrido, 2004b] M. Díaz y D. Garrido. "A Simulation Environment for Nuclear Power Plants". Proceedings 8th IEEE International Symposium on Distributed Simulation and Real Time Applications, pp. 98-105, 2004.

[Díaz et al., 2005] M. Díaz, D. Garrido, L. Llopis y J.M. Troya. "Integrating RT-CORBA in SDL". 12th International SDL Forum, Lecture Notes on Computer Science 3530, Springer Verlag, pp. 47-67, 2005.

[Díaz et al, 2005b] M. Díaz, D. Garrido y E. Soler. "A Distributed Simulation Tool on PDA". Proceedings 14th IASTED International Conference on Applied Simulation and Modelling, pp. 376-381, IASTED, 2005.

[Díaz et al., 2005c] M. Díaz, D. Garrido, S. Romero, B. Rubio, E. Soler y J.M. Troya. "A CCA-compliant Nuclear Power Plant Simulator Kernel". Eighth International SIGSOFT Symposium on Component-Based Software Engineering (CBSE), Lecture Notes on Computer Science 3489, Springer Verlag, pp. 283-297, 2005.

[Doldi, 2001] L. Doldi. SDL Illustrated - Visually design executable models, 2001.

[EJB, 2005] Enterprise JavaBeans Specification 2.1, Sun Microsystems http://java.sun.com/products/ejb/, 2005.

[EVC, 2005] Embedded Visual C++, Microsoft, http://msdn.microsoft.com/mobility/othertech/eVisualc/default.aspx, 2005.

[Fu et al., 2000] J. Fu, W. See, P. Hsiung, J. Chao y S. Chen. "A Java-Based Distributed System Framework for Real-Time Development". ICDCS International Workshop on Distributed Real-Time Systems, pp. B31-B36, 2000.

[Gill et al., 2001] C. D. Gill, D. L. Levine y D. C. Schmidt. "The Design and Performance of a Real-Time CORBA Scheduling Service". Real-Time Systems, vol. 20, num. 2, pp. 117-154, 2001.

[Graham, 1969] R.L. Graham. "Bounds on multiprocessing timing anomalies". SIAM J. Appl. Math., vol. 17, num. 2, pp. 416-429, 1969.

[Graham et al., 1979] R.L. Graham, E.L. Lawler, J.K. Lenstra y AHGR Kan. "Optimization and approximation in deterministic sequencing and scheduling: A Survey". Annals of Discrete Mathematics. 5, pp. 287-326, 1979.

[Grattan y Brain, 2000] N. Grattan y M. Brain. Windows CE 3.0 Application Programming. Prentice-Hall, 2000.

[Hansson et al., 2004] H. Hansson, M. Åkerholm1, I. Crnkovic y M. Törngren. "SaveCCM – a component model for safety-critical real-time systems". Proceedings 30th EUROMICRO Conference (IEEE), pp. 14-21, 2004.

[Harrison et al., 1997] T.H. Harrison, D.L. Levine y D.C. Schmidt, "The Design and Performance of a Real-Time CORBA Event Service", Proceedings of the OOPSLA '97 conference, pp. 184-200, 1997.

[Henning y Vinoski, 2001] M. Henning y S. Vinoski. Programación Avanzada en CORBA con C++. Addison Wesley, 2001.

[Henzinger, 2000] T.A. Henzinger. "Masaccio: A Formal Model for Embedded Componentes". Proceedings of the First IFIP International Conference on Theoretical Computer Science (TCS), Lecture Notes in Computer Science 1872, Springer Verlag , pp. 549-563, 2000.

[Hooman y Van Roosmalen, 2000] J. Hooman y O.Van Roosmalen. "An Approach to Platform Independent Real-Time Programming". Real-Time Systems, vol. 19, num 1, pp. 61-85, 2000.

[Hsiung et al., 2002] P. Hsiung, T. Lee, W. See, J. Fu y S. Chen. "VERTAF: An Object-Oriented Application Framework for Embedded Real-Time Systems", Proceedings OF the 5th IEEE International Symposium on Object-oriented Real-time Distributed Computing, pp. 322-329, 2002.

[Industrial, 2005] Industrial TRON. http://tron.um.u-tokyo.ac.jp/

[ITU SDL, 1994] ITU Recommendation Z.100. Specification and Description Language (SDL), 1994.

[Jaluna, 2005] Jaluna Real-time Component Suite.
http://sourceforge.net/projects/jaluna , 2005.

[Joseph y Pandya, 1986] M. Joseph y P. Pandya. "Finding Response Time in a Real-Time System". The Computer Journal, vol. 29, num. 5, pp. 390-395, 1986.

[Kiczales et al., 1997] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.M. Loingtier y J. Irwin, "Aspect-oriented programming". Proceeding of the ECOOP, ser. Lecture Notes in Computer Science, vol. 1241, Springer Verlag, pp. 220-242, 1997.

[Kim, 1999] K.H. Kim. “Real-Time Object-Oriented Distributed Software Engineering and the TMO Scheme”. International Journal of Software Engineering and Knowledge Engineering, vol 9., num 2., pp. 251-276, 1999.

[Klefstad et al., 2002] R. Klefstad, D.C. Schmidt, C. O’Ryan. “Towards Highly Configurable Real-time Object Request Brokers”, Proceedings of the 5th IEEE International Symposium on Object-oriented Real-time Distributed Computing, pp. 437-447, 2002.

[Kopetz, 1997] H. Kopetz. Real-Time Systems: Design Principles for Distributed Embedded Applications. Kluwer International Series in Engineering and Computer Science.

[Kopetz y Obermaisser, 2002] H. Kopetz y R. Obermaisser. “Temporal Composability”. Computing & Control Engineering Journal, vol. 13, pp. 156-162, 2002.

[Kopetz y Ochsenreiter, 1987] H. Kopetz y W. Ochsenreiter. “Cock synchronization in distributed real-time systems”. IEEE Transactions on Computers, vol. 36, num. 8, pp. 933-940, 1987.

[Lamport, 1978] L. Lamport. “Time, clocks, and the ordering of events in a distributed system”. Communications of the ACM, 21(7), pp. 558-565, 1978.

[Lankes et al., 2001] S. Lankes, M. Pfeiffer y T. Bemmerl. “Design and Implementation of a SCI-based Real-Time CORBA”. Proceedings of the 4th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, pp. 23-30, 2001.

[Lankes et al., 2002] S. Lankes, M. Reke y A. Jabs. “A Time-Triggered Ethernet Protocol for Real-Time CORBA”. Proceedings of the 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, pp. 215-222, 2002.

[Lankes et al., 2003] S. Lankes, A. Jabs y T. Bemmerl. “Integration of a CAN-based Connection-oriented Communication Model into Real-Time CORBA”. Proceedings IEEE International Parallel and Distributed Processing Symposium, p. 121, 2003.

[Levine et al., 1998] D. L. Levine, S. Mungee y D.C. Schmidt, “The Design of the TAO Real-Time Object Request Broker”. Computer Communications 21, pp. 294-324, 1998.

[Liu y Layland, 1973] L. Liu y J.W.Layland. “Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment”. Journal of ACM, vol. 20, num. 1, pp. 46-61, 1973.

[Llopis, 2002] Luis Manuel Llopis Torres. Tesis doctoral titulada “Integración de Análisis de Tiempo Real en la Técnica de Descripción Formal SDL”. Universidad de Málaga, 2002.

[Lynx, 2005] Sistema Operativo Lynx Os. http://www.lynuxworks.com/, 2005.

[Mattsson et al., 1999] T. Mattsson, H. Heeb y J. Tryggvesson. “Jbed: Java for Real-Time Systems”. Dr. Dobb´s Journal, Nov., 1999.

[McDysan y Spohn, 1998] D.E. McDysan y D.L. Spohn. ATM Theory and Application. McGraw-Hill, 1998.

[Microsoft .NET, 2005] Microsoft .NET home page. http://www.microsoft.com/net/default.mspx, 2005.

[Mok y Dertouzos, 1978] A. Mok y M. Dertouzos. "Multiprocessor scheduling in a hard real-time environment". Proceedings of the 7th Texas Conference on Computing Systems, pp. 5-1/5-12, 1978.

[Müller et al., 2001] P. Müller, C. Stich y C. Zeidler. "Components @ Work: Component Technology for Embedded Systems". 27th International Workshop on Component-Based Software Engineering, EUROMICRO, pp. 64-71, 2001.

[Nakamoto et al., 2002] Y. Nakamoto, N. Iga, H. Hihara. "Embedded CORBA Development and Its Applications to In-Satellite Network Software". Proceedings of the 5th IEEE International Symposium on Object-oriented Real-time distributed Computing, pp. 315-321, 2002.

[Norström et al., 2001] C. Norström, M. Gustafsson, K. Sandström, J. Mäki-Turja y N.-E. Bånkestad. "Experiences from Introducing State-of-the art Realtime Techniques in the Automotive Industry". Proceedings 8th IEEE International Conference on Engineering of Computer-based Systems, pp. 111-118, 2001.

[OMG, 1998] Object Management Group. "CORBA Messaging", orbos/98-05-05, 1998.

[OMG, 1999] Object Management Group. "CORBA Component Model Joint Revised Submission", 1999.

[OMG, 2001] Object Management Group. "The Common Object Request Broker: Architecture and Specification", versión 2.6, Diciembre 2001.

[OMG, 2004] Object Management Group, "Minimum CORBA Update", realtime/2004-06-01, 2004.

[OMG, 2005] Object Management Group. "RealTime-CORBA Specification", versión 1.2 Static Scheduling, formal/05-01-04, 2005.

[Ommering et al., 2000] R. van Ommering, F. van der Linden y J. Kramer. "The Koala component model for consumer electronics software". IEEE Computer, 33(3), pp. 78–85, 2000.

[Pfeffer et al., 2002] M. Pfeffer, S. Uhrig, Th. Ungerer y U. Brinkschulte, "A Real-Time Java System on a Multithreaded Java Microcontroller". Proceedings of the 5th IEEE International Symposium on Object-oriented Real-time distributed Computing, pp. 34-41, 2002.

[Pyarali et al, 2003] I. Pyarali, D.C. Schmidt y R.K. Cytron. "Techniques for Enhancing Real-time CORBA Quality of Service". Proceedings of the IEEE, vol. 91, num. 7, pp. 1070-1085, 2003.

[Rajkumar, 1991] R. Rajkumar. Synchronization in Real-Time Systems. A Priority Inheritance Approach. Kluwer Academic Publishers, 1991.

[Robinson et al., 2002] S. Robinson, B. Harvey, C. Nagel, O. Cornes, K. Watson, M. Skinner, J. Glynn, Z. Greenvoss y S. Allen. Professional C# 2nd Edition. Wrox Press, 2002.

[Ross, 1989] F.E. Ross. "An Overview of FDDI: The Fiber Distributed Data Interface". IEEE Journal on Selected Areas in Communications, vol. 7, num. 7, pp. 1043-1051, 1989.

[Schmidt, 1994] D. C. Schmidt, "ACE: an Object-Oriented Framework for Developing Distributed Applications". Proceedings of the 6th USENIX C++ Technical Conference, USENIX Association, 1994.

[Schmidt y Kuhns, 2000] D.C. Schmidt y F. Kuhns, "An overview of the Real-time CORBA Specification". IEEE Computer special issue on Object-Oriented Real-time Distributed Computing, vol. 33, num. 6, pp. 56-63, 2000.

[Sha et al., 1990] L. Sha, R. Rajkumar y J.P. Lehoczky. "Priority Inheritance Protocols: An Approach to Real-Time Synchronization". IEEE Transations on Computers, vol. 39, num. 9, pp. 1156-1174, 1990.

[Shaw, 1989] A.C. Shaw. "Reasoning about time in higher-level language software. IEEE Transactions on Software Engineering, SE-15(7), pp. 875-889, 1989.

[Sifakis, 1999] J. Sifakis. "The Compositional Specification of Timed Systems, A Tutorial". Computer Aided Verification, Lecture Notes in Computer Science 1633, Springer Verlarg, pp. 2-7, 1999.

[Singh et al., 2002] G. Singh, B. Maddula y Q. Zeng. "Enhancing Real-Time Event Service for Synchronization in Object Oriented Distributed Systems". Proceedings of the 5th IEEE International Symposium on Object-oriented Real-time distributed Computing, pp. 233-240, 2002.

[Solaris, 2005] Sistema Operativo Solaris. http://www.sun.com, 2005.

[Stankovic et al., 1985] J. Stankovic, K. Ramamritham y S. Cheng. "Evaluation of a flexible task scheduling algorithm for distributed hard real-time systems", IEEE Transactions on Computing, 34(12), pp. 1130-1143, 1985.

[Stankovic, 2001] J. Stankovic. "VEST: a toolset for constructing and analyzing component based operating systems for embedded and real-time systems,". Proceedings of the Embedded Software, First International Workshop, Lecture Notes in Computer Science 2211, Springer Verlag, pp. 390–402, 2001.

[Stoyen et al., 1999] A.D. Stoyen, T.J. Marlowe, M.F. Younis y P.V. Petrov. "A Development Environment for Complex Distributed Real-Time Applications". IEEE Transactions On Software Engineering, vol. 25, num. 1, pp. 50-74, 1999.

[Szypersky et al., 2002] C. Szypersky, D. Gruntz y S. Murer. Component Software. Beyond Object-Oriented Programming. 2nd Edition. ACM Press, 2002.

[TAU, 2000] Telelogic. SDT 4.0 Manuals, 2000.

[Tecnatom, 2005] Tecnatom S.A. http://www.tecnatom.es , 2005.

[Tesanovic et al., 2004] A. Tesanovic, D. Nyström, J. Hansson y C. Norström. "Aspects and Components in Real-Time System Development: Towards Reconfigurable and Reusable Software". Journal of Embedded Computing, vol. 1, num. 1, pp. 1-16, 2004.

[Tindell et al., 1995] K. Tindell, A. Burns y A.J. Wellings. "Calculating controller area network (CAN) message response times". Control Engineering Practice 3(8), pp. 1163-1169, 1995.

[Villela et al., 2001] C. Villela, L.B. Becker y C.E. Pereira. "Framework for Component-Based Development of Distributed Real-Time Systems". Sixth International Workshop on Object-Oriented Real-Time Dependable Systems, pp. 85-90, 2001.

[VxWorks, 2005] Sistema Operativo VxWorks 5.X.
http://cdn.windriver.com/products/device_technologies/os/vxworks5 , 2005.

[Wang et al., 2001] N. Wang, D.C. Schmidt y C. O'Ryan. "Overview of the CORBA Component Model", capítulo del libro "Component-based software engineering: putting the pieces together", Addison-Wesley Longman Publishing Co., 2001.

[Wellings et al., 2002] A. Wellings, G. Bernat y E. Yu-Shing Hu. "Addressing Dynamic Dispatching Issues in WCET Analysis for Object-Oriented Hard Real-Time Systems". Proceedings of the 5th IEEE International Symposium on Object-oriented Real-time distributed Computing, pp. 109-116, 2002.

[Yau y Zhou, 2002] S.S. Yau y X. Zhou. "Schedulability in Model-based Software Development for Distributed Real-time Systems". Proceedings 7th International Workshop on Object-Oriented Real-time Dependable Systems, pp. 45-52, 2002.

[Yen et al., 2002] I. Yen, J. Goluguri, F. Bastani, L. Khan. "A Component-based Approach for Embedded Software Development". Proceedings of the 5th IEEE International Symposium on Object-oriented Real-time Distributed Computing, pp. 402-410, 2002.

[Yodaiken y Barabanov, 1997] V. Yodaiken y M. Barabanov. "A real-time linux", http://www.fsmlabs.com/images/stories/pdf/archive/usenix.pdf, 1997.