PicoCORBA: Implementación mínima de objetos CORBA para dispositivos empotrados

D. Villa, F. Moya, y J.C. López

Departamento de Informática, Universidad de Castilla-La Mancha,
Escuela Superior de Informática, 13071 Ciudad Real, España
email: {David.Villa, Francisco.Moya, JuanCarlos.Lopez}@uclm.es

Resumen

El presente artículo plantea una nueva aproximación a la implementación de sistemas CORBA empotrados para dispositivos de cómputo de muy escasos recursos. Se desarrolla la idea de reducir la implementación de objetos CORBA a la generación automática de máquinas de estados que se encargan del reconocimiento de los mensajes necesarios. La estrategia subyacente es determinar la funcionalidad del objeto más pequeño que pueda considerarse CORBA y, a partir de ahí, añadirle capacidades en función de las posibilidades y recursos que ofrezca el dispositivo sobre el que se quiere implementar. Con este enfoque, es posible conseguir implementaciones que pueden ser ubicadas en pequeños microcontroladores de 8 bits, logrando plena interoperabilidad con CORBA.

Palabras clave: sistemas empotrados, middleware, CORBA

1. INTRODUCCIÓN

Los nuevos sistemas relacionados con el paradigma de la computación ubicua requieren de innumerables dispositivos colocados en la vivienda, el automóvil e incluso en las personas. Estos dispositivos deben estar conectados al resto del sistema, para de ese modo poder crear servicios valiosos para los usuarios. Cuanto mayor sea la cantidad y variedad de estos dispositivos, mayor será la calidad y utilidad de la información que se pueda obtener de ellos.

Al mismo tiempo, resulta muy conveniente que estos dispositivos gocen de cierto grado de autonomía. Ello permite que el sistema sea mucho más robusto, fiable y tolerante a fallos. Es decir, se puede prescindir de elementos intermediarios, tales como la pasarela residencial típica en los sistemas domóticos (OSGi[4]).

Para lograr que este conjunto de dispositivos, potencialmente heterogéneos, pueda interoperar se necesita de un elemento “unificador”: el *middleware* de comunicaciones. Por tanto, parece claro que los sistemas de objetos distribuidos son la opción más adecuada. Sin embargo, las implementaciones actuales para COM, Jini [3] o CORBA [2] requieren de una cantidad de recursos considerable.

En este sentido, existen varias iniciativas. La especificación minimumCORBA[5] de OMG [1] describe cómo recortar las características más costosas del ORB (*Object Request Broker*). Estos recortes están relacionados en su mayor parte con la creación e invocación dinámica de objetos. De ese modo se facilita la implementación sobre dispositivos empotrados y, al mismo tiempo, se mantiene la interoperabilidad con el resto de la arquitectura CORBA.

A pesar de ello, un ORB conforme a minimumCORBA puede resultar demasiado pesado. Un caso interesante es el de dynamicTAO [7] y sus sucesores: legORB [8] y UIC-CORBA [6]; legORB es un ORB modular y configurable dinámicamente en el que sólo se instancian los componentes que se solicitan. Con ello se consiguen implementaciones muy pequeñas. Según [9] se pueden construir ORB’s (sólo el lado del cliente) sobre un HP Jornada 680 con Windows CE en 20Kb, y sobre PalmPilot con PalmOS 3.0 en 6Kb.

Un producto comercial muy similar es OpenFusion e*ORB [10], un ORB modular con características de tiempo real que puede funcionar sobre una HP iPAQ o un DSP TMS320C64X de Texas Instruments.

Otros desarrollos interesantes son TINIORB [11], un ORB conforme a minimumCORBA para el dispositivo TINI [13] (*Tiny Internet Interface*) de Dallas Semiconductors; y MicroQoSCORBA [12] que propone generar ORB’s específicos para servidores y clientes dependiendo de las características del dispositivo concreto (probado en SaJe [14] y también con TINI).

Estos datos constituyen una diferencia sensible con otras implementaciones de CORBA o minimumCORBA. Sin embargo, no es suficientemente pequeño. Utilizar una máquina con un procesador SH3 y un sistema operativo como PalmOS, un DSP o un dispositivo TINI (cuyo precio ronda los 60€) resulta prohibitivo si lo que pretendemos es integrar en el sistema ubicuo absolutamente todos los dispositivos del entorno.

Pretendemos que cualquier bombilla, interruptor o pulsador presente en una casa u oficina sea un elemento más del sistema. Para lograrlo necesitamos una implementación aún más pequeña y autocontenida. El objetivo es disponer de objetos CORBA (servidores) que puedan implementarse en microcontroladores de 8 bits con menos de 512 palabras de memoria (con un precio aproximado de 0.5€), completamente autónomos y sin necesidad de ningún sistema operativo o de supervisión del dispositivo.

2. PICOCORBA

Desde el momento en que se definió GIOP (*General Inter-ORB Protocol*) como medio para comunicar ORB’s, los obje-

tos CORBA pueden verse desde el exterior usando dos puntos de vista:

- Cada objeto remoto *es como* un objeto local, y se puede utilizar invocando los métodos del adaptador de objeto del ORB. El ORB y el POA materializan de forma automática una implementación portable del patrón *remote proxy* [15]. Este punto de vista proporciona total aislamiento de la localización física y lógica, lo cual resulta muy positivo en entornos dinámicos.
- Por otro lado, un objeto puede verse desde la red de comunicaciones como un servidor que implementa un pequeño protocolo: GIOP. Esto no tiene particular interés desde el lado de los clientes, pero sí desde el lado del propio objeto.

Este segundo enfoque resulta especialmente interesante si se pretende *aligerar* aún más la implementación. Implica que el objeto puede deshacerse de todo el ORB y sus interfaces de programación, implementando exclusivamente el protocolo que le afecta. Ésta es, de hecho, la única alternativa cuando el dispositivo de cómputo es tan pequeño que no es posible implementar en él un ORB, ni siquiera acogiéndose a minimumCORBA.

Si el objeto conoce el lenguaje que utiliza el ORB para comunicarse con otros ORB's y puede utilizarlo directamente, la cantidad de recursos necesarios es muy inferior; ese lenguaje del que hablamos es GIOP. Desde ese momento, los objetos que implementan GIOP son pequeños servidores del mismo. Cualquier cliente que invoque a un objeto de este tipo no notará diferencia alguna con cualquier otro ORB.

En resumen, cuando la aplicación es lo suficientemente simple, se puede implementar un pequeño ORB "a medida", de manera que sea capaz de responder a unas cuantas peticiones muy concretas, obviando el resto de los servicios.

PicoCORBA materializa este enfoque. Los objetos picoCORBA no se sirven de un ORB para comunicarse con el resto de los objetos conectados al bus. En lugar de ello, cada servidor picoCORBA implementa ad hoc la funcionalidad que los objetos residentes necesitarían del ORB. Es decir, picoCORBA no es una librería, ni siquiera una plantilla. Se trata de un conjunto de herramientas (*compiladores*) que generan código para sustituir al ORB. Evidentemente esto no tendría sentido si el código generado fuese siempre el mismo, aunque variase entre plataformas diferentes.

Esta estrategia tiene sentido precisamente porque no se genera un ORB completo, sino sólo la parte de ORB imprescindible para cubrir las necesidades de los objetos residentes, siendo conforme además a las restricciones de la plataforma objetivo. Dicho de otro modo, el código generado para cada servidor es diferente al de los demás. Por ejemplo, para un servidor que no albergue ningún objeto no se genera nada de código.

Este planteamiento implica que el ORB generado por picoCORBA (*pico-ORB*) no se puede considerar un ORB como tal, pues no está completo. En picoCORBA no tiene cabida (en principio) la implementación de servicios exclusivos del ORB, como pueden ser las peticiones de localización de objetos. PicoCORBA está "orientado a los objetos", más concretamente, a los sirvientes de los objetos. Solamente se atenderán peticiones dirigidas explícitamente a un objeto residente. Si un ORB cliente invoca un servicio que sólo atañe al hipotético ORB local, esa petición será completamente ignorada.

Según la especificación CORBA, se puede decir que un ORB tiene dos interfaces diferentes:

- **A nivel de programa:** Ofrece una API al programador. Esta API se puede usar, desde el punto de vista del cliente, para proporcionar acceso a los objetos remotos; o bien, desde el servidor, para registrar y poner objetos al servicio de la red.
- **A nivel de bus:** Dispone del lenguaje (GIOP) que le permite interactuar con otros ORB's.

Desde el punto de vista de un objeto que reside en un servidor picoCORBA, no existe ORB alguno, no hay ningún componente en el dispositivo que le esté dando ningún tipo de servicio, ni facilitándole en acceso al bus ni al subsistema de comunicaciones. Por tanto, este *pico-ORB* del que se habla no dispone de una interfaz a nivel de programa, no existe realmente. Sin embargo, para los clientes, el servidor se comporta como si tuviese un ORB, sí que dispone de la interfaz a nivel de bus. Por ello, se dice que el *pico-ORB* generado es un *ORB virtual*.

Es importante remarcar que picoCORBA está orientado exclusivamente a la implementación de servidores. Como se ha visto, se trata de crear objetos CORBA lo más pequeños posible, por tanto, eso no tiene ninguna utilidad para los clientes.

CORBA ofrece una arquitectura en la que se pueden añadir nuevos elementos software sin necesidad de realizar ninguna modificación en el resto de la arquitectura. El único requisito es que los nuevos componentes sean capaces de comunicarse con el resto. Para eso se utilizan las interfaces, que se definen en IDL. Gracias a eso, tanto minimumCORBA como picoCORBA pueden interactuar con el resto del sistema CORBA como si se trataran de implementaciones completas.

3. MÁS PEQUEÑO QUE MINIMUMCORBA

Las características de minimumCORBA son, principalmente, *portabilidad*, *interoperabilidad* y soportar la *especificación completa del IDL*. Para llegar hasta estos objetivos, minimumCORBA elimina componentes dinámicos o demasiado complejos. En definitiva, sigue siendo una implementación de CORBA, pero con capacidades reducidas.

Los objetos CORBA habituales están conectados a algún ORB que les proporciona ciertos servicios. Lo mismo ocurre con los objetos minimumCORBA. Se programan y comportan como cualquier otro objeto CORBA que use sólo las interfaces y servicios soportados por minimumCORBA. En cambio,

los objetos picoCORBA deben proveerse ellos mismos de todos los servicios que necesiten, dando así lugar al "*ORB virtual*".

En picoCORBA tampoco hay un *adaptador de objetos* explícito. Los objetos no tienen que registrarse para ser accesibles ni necesitan informar de su existencia a un ORB. Todo esto implica ahorro de recursos, sobre todo, si hay una cantidad considerable de objetos, lo cual supone una ventaja sobre minimumCORBA. En la figura 1 se representan las diferencias entre una implementación CORBA convencional, minimumCORBA y picoCORBA.

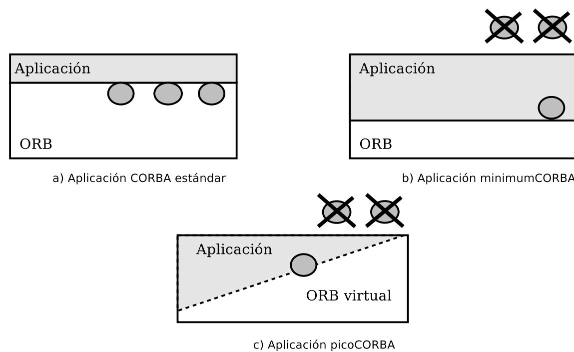

Figura 1: Representación gráfica de las diferencias entre CORBA, minimumCORBA y picoCORBA. Los círculos representan servicios del ORB.

PicoCORBA cumple la mayoría de los objetivos de minimumCORBA, aunque con ciertos condicionantes:

- **Portabilidad**: El código escrito para minimumCORBA puede utilizarse sobre cualquier otro ORB, ya sea minimumCORBA o no. Sin embargo, el código generado por picoCORBA no tiene ningún tipo de portabilidad respecto a otros ORB's ya que no utiliza la API definida para ello. Es un código autocontenido que, además, puede ser ensamblador o cualquier lenguaje para el que la OMG ni siquiera ha definido un *mapping* y, evidentemente, eso impide por definición cualquier grado de portabilidad. Es el precio a pagar para conseguir una implementación extremadamente reducida.

- **Interoperabilidad**: Los objetos picoCORBA interactúan directamente con el ORB del cliente. Si el objeto implementa la funcionalidad mínima de acuerdo al protocolo GIOP, la interoperabilidad está garantizada.

- **Soporte completo de IDL**: El soporte de IDL depende a su vez en gran medida del soporte de CDR (*Common Data Representacion*). Tanto los tipos primitivos de CDR como los operadores de composición están soportados.

De forma complementaria, otros objetivos de picoCORBA son:

- Definir la **funcionalidad esencial** que permite a un objeto tener un comportamiento estándar y que garantiza unos requisitos mínimos de interoperabilidad.

- Definir y construir mecanismos que permitan **dimensionar** el conjunto de funcionalidades soportado por el servidor, conforme a las restricciones impuestas por la plataforma.

En definitiva, un servidor picoCORBA se comporta como un autómata capaz de generar respuestas determinadas ante estímulos concretos. La interfaz de los objetos residentes determina qué mensajes reconocerá ese autómata y cuál será su formato (su gramática).

Bien es cierto, que en tiempo de compilación, no es posible conocer qué mensajes de respuesta tendrá que generar el autómata, ya que depende en gran medida de los valores de los *tokens* identificados en el mensaje de petición procedente del cliente.

Sin embargo, el valor de los campos que conforman la respuesta se puede obtener en tiempo de ejecución a partir del mensaje de petición (p.ej: `request_id`) y de la ejecución del procedimiento correspondiente. De modo que, aunque con ciertas restricciones, se puede abordar la generación automática del *ORB virtual* e, incluso, del esqueleto de implementación del sirviente.

4. FUNCIONALIDAD MÍNIMA

Uno de los objetivos principales de picoCORBA es que se pueda dimensionar y adaptar la funcionalidad del código generado en base a las posibilidades y restricciones de la plataforma destino. Se pretende que esa plataforma pueda ser desde un pequeño microcontrolador hasta un PC.

Con esto en mente, el diseño de picoCORBA permite que, en plataformas con mayores recursos, se puedan añadir características convenientes, pero no esenciales.

Por ejemplo: Un mensaje de petición GIOP determinado llega a un objeto alojado en un ORB estándar. Si dicho mensaje invoca una operación que el objeto no soporta (no está en su interfaz IDL), el ORB devuelve al cliente un mensaje de respuesta indicando la excepción `SYSTEM_EXCEPTION:BAD_OPERATION`. Sin embargo, en PicoCORBA se pueden tomar diferentes alternativas dependiendo de la cantidad de recursos disponibles en la arquitectura sobre la que se va a implementar:

- Ignorar el mensaje y nunca responder.

- Devolver un mensaje de tipo *MessageError* (ver 5.7.).

- Devolver un mensaje de respuesta genérico con la excepción `SYSTEM_EXCEPTION`, pero sin `minor_code`.

- Construir y devolver un mensaje de respuesta completo con la excepción `SYSTEM_EXCEPTION` y un `minor_code_value` con valor `BAD_OPERATION`.

Claramente, para el caso en el que se persigue la funciona-

lidad esencial se debe tomar la primera opción.

Aunque el modelo que pretende picoCORBA permita dimensionar la implementación a cualquier nivel, otro de los objetivos (el más importante) es crear implementaciones pequeñas. Es decir, el esfuerzo de desarrollo va encaminado a dispositivos con prestaciones limitadas. La posibilidad de generar un racimo de objetos en un PC que materializa un **ORB completamente funcional** no presenta ventajas significativas respecto a un ORB estándar.

Sin embargo, incluso disponiendo de una gran cantidad de recursos, cabe la posibilidad de que otros condicionantes como disponibilidad, seguridad, fiabilidad o restricciones de tiempo real aconsejen implementar objetos con la filosofía de picoCORBA. Claramente, no se trata de un requisito prioritario, pero no se descarta su utilización para este fin. Sobre todo, teniendo en cuenta que en el entorno de pruebas para el prototipo, se utilizan habitualmente implementaciones sobre PC.

En el otro extremo del ORB completamente funcional, está la funcionalidad esencial y esa sí es una cuestión relevante. Se trata de definir cuál es el conjunto de características mínimo que debe soportar una implementación creada con picoCORBA para que sea de alguna utilidad.

A continuación, se describen en detalle algunos aspectos significativos para esa funcionalidad esencial.

4.1. Versión de GIOP

La especificación GIOP obliga a que todo cliente "hable" con el servidor en una versión de GIOP igual o menor a la que aparezca en la referencia publicada por el servidor para ese objeto (su IOR). Esta especificación también dice que todo servidor que soporte una versión de GIOP posterior a la 1.0 debe soportar también las anteriores.

Dado que picoCORBA pretende ser lo más pequeño y simple posible, atenerse únicamente a la especificación de GIOP 1.0 garantiza la interoperabilidad con cualquier cliente que sea conforme al estándar y, además, no obliga a implementar nada más.

Las extensiones que OMG ha añadido a GIOP en versiones posteriores a la 1.0 tienen que ver con: direccionamiento indirecto de objetos en otros ORB's, fragmentación de mensajes y GIOP bi-direccional. Todas estas cuestiones quedan fuera de las pretensiones iniciales de picoCORBA. De hecho, resulta muy conveniente evitar estas características, pues su implementación podría resultar muy costosa.

En el caso de plantearse la necesidad de incluir estas características no hay inconveniente en extender picoCORBA para soportar versiones posteriores de GIOP, aunque como se ha explicado, ello obliga a soportar también las versiones anteriores.

4.2. Ordenamiento (*byte order*)

La especificación GIOP determina que quien inicia una conexión (el cliente en el caso de GIOP 1.0) determina cuál será el ordenamiento de los mensajes. El servidor es el encargado de realizar la conversión de ordenamiento cuando sea necesario.

Dado que los prototipos iniciales de picoCORBA están muy orientados a la construcción de agentes en redes domóticas, se considera un entorno lo suficientemente acotado como para asumir un ordenamiento fijo en todas las comunicaciones. En estas circunstancias, los proxies de objetos o las pasarelas de protocolo son los encargados de realizar las conversiones en el ordenamiento. De ese modo, ni los dispositivos picoCORBA ni los clientes deberán preocuparse por ello.

4.3. Interfaz *CORBA::Object*

La norma CORBA dice que todo objeto, por el hecho de serlo, implementa la interfaz *CORBA::Object*. En el listado 1 aparecen las operaciones más significativas de esta interfaz.

```
interface Object {
  InterfaceDef get_interface ();

  boolean is_nil ();

  Object duplicate ();

  void release ();

  boolean is_a (
    in string logical_type_id
  );

  boolean non_existent();

  boolean is_equivalent (
    in Object other_object
  );

  unsigned long hash(
    in unsigned long maximum
  );
};
```

Listado 1: Interfaz *CORBA::Object*

En realidad, no son métodos del objeto, no son operaciones que deba tratar el sirviente. Es el ORB quien se encarga de evaluar y responder estas operaciones.

Algunas de ellas, como `duplicate()`, `release()` o `is_nil()`, son atendidas directamente por el ORB cliente y nunca llegan al servidor. Otras, como `is_a()` o `non_existent()` pueden implicar que el ORB cliente pregunte al ORB remoto.

Como la implementación de un objeto picoCORBA no dispone de ORB, en este caso sí es el objeto quien debe responder a las operaciones necesarias. En cualquier caso, sólo deben preocupar las operaciones que realmente involucran un acceso al objeto remoto. Para picoCORBA, se considera que las

únicas esenciales son `is_a()` y `non_existent()`.

Resumiendo, se garantiza una funcionalidad mínima, que implica que un servidor picoCORBA: a) Ignora todos los mensajes no dirigidos explícitamente a uno de sus objetos; b) Asume un ordenamiento de bytes concreto e ignora los mensajes con ordenamiento distinto; c) Sólo implementa las operaciones `is_a()` y `non_existent()` de la interfaz *CORBA::Object*.

5. CIOP

De la misma manera que GIOP comunica un ORB con otros, definimos CIOP (*Customizable Inter-ORB Protocol*), como un protocolo que determina el modo en que un servidor PicoCORBA debe comunicarse directamente con otros ORB's.

Como se muestra a continuación, el protocolo GIOP utiliza distintos tipos de mensajes en sus comunicaciones (ver tabla I).

Mensaje	Origen	Valor	Versión GIOP
Request	Cliente	0	1.0, 1.1, 1.2
Reply	Server	1	1.0, 1.1, 1.2
CancelRequest	Cliente	2	1.0, 1.1, 1.2
LocateRequest	Cliente	3	1.0, 1.1, 1.2
LocateReply	Servidor	4	1.0, 1.1, 1.2
CloseConnection	Servidor	5	1.0, 1.1, 1.2
MessageError	Ambos	6	1.0, 1.1, 1.2
Fragment	Ambos	7	1.1, 1.2

Tabla I: Tipos de mensajes en GIOP.

Los objetivos de CIOP son muy similares a los de GIOP: *disponibilidad*, *escalabilidad*, *generalidad*, *independencia de arquitectura*, *simplicidad* y *bajo coste*; aunque se enfatizan las dos últimas.

Y también del mismo modo que GIOP, CIOP puede encapsularse sobre cualquier protocolo de transporte que contemple ciertas consideraciones.

Como CIOP impone ciertas restricciones a GIOP, se puede decir que es un subconjunto de éste y, por ello, ambos pueden interactuar sin problemas. Por otra parte, GIOP es independiente de la funcionalidad del ORB, mientras que CIOP no lo es.

A continuación se definen las restricciones aplicables, qué mensajes se soportan y en qué condiciones; y cuáles ni siquiera se consideran. Todas las restricciones se refieren a la funcionalidad mínima explicada en la sección 4..

Sólo se nombran los campos de los mensajes sobre los que se aplica alguna restricción. Los campos y características que no se mencionan siguen las normas que marca GIOP. Para una mejor comprensión, se aconseja consultar paralelamente el capítulo 15, titulado *General Inter-ORB Protocol* del estándar CORBA [2].

5.1. Cabecera CIOP

Es idéntica a la de GIOP con dos puntualizaciones:

- `GIOP_version` tiene el valor constante `1,0`.
- `byte_order` es contante y determinado por la implementación.

5.2. Mensajes de petición (*Request*)

Corresponden a invocaciones a métodos de objetos CORBA. Se toman las siguientes consideraciones:

- el campo `service_context` corresponde a información en el contexto del servicio, por tanto, se ignora.
- `response_expected` se ignora, el servidor picoCORBA siempre devolverá una respuesta aunque el cliente no la solicite. Es un comportamiento conforme al estándar ya que el cliente ignora una respuesta no solicitada.
- `object_key` identifica al objeto sobre el que se invoca la operación. El estándar garantiza que el cliente colocará aquí la cadena que aparezca en la IOR, sin modificación alguna. Aprovechando esta restricción, picoCORBA utiliza "claves de objeto" de tamaño constante. Esto facilita el reconocimiento del mensaje.
- `requesting_principal` está anticuado (*deprecated*) y se ignora.

5.3. Mensajes de respuesta (*Reply*)

Los mensajes de respuesta se devuelven al cliente como resultado de la ejecución de una operación. Un servidor picoCORBA siempre genera un mensaje de respuesta si considera válido el mensaje de petición, en caso contrario no devuelve nada; la implementación más restrictiva de picoCORBA no genera excepciones. El contenido del mensaje se determina a partir de la petición y de los datos aportados por la ejecución del procedimiento de usuario.

Consideraciones sobre los campos de la respuesta:

- `service_context` - se copia el valor del campo homónimo del mensaje de petición.
- `request_id` - conforme al estándar, se copia también el valor del campo homónimo del mensaje de petición.
- `reply_status` - su valor es siempre `NO_EXCEPTION`, ya que picoCORBA no considera excepciones (ni de sistema ni de usuario) ni tampoco información de redirección de localización (*location forward*).

5.4. Mensaje de cancelación de petición (*CancelRequest*)

Según la especificación, los servidores pueden hacer caso omiso a una cancelación de petición y, a pesar de recibirla, optar por procesar y enviar la respuesta de todos modos.

Por tanto, un servidor picoCORBA puede ignorar los mensajes de cancelación de petición y, sin embargo, sigue siendo un comportamiento completamente conforme al estándar.

5.5. Mensajes de localización (*LocateRequest/LocateReply*)

Lo utilizan los clientes para localizar objetos. En su versión mínima, picoCORBA no soporta ningún tipo de información de localización e ignora este tipo de mensajes; evidentemente tampoco genera respuestas *LocateReply*.

5.6. Mensajes de finalización de conexión (*CloseConnection*)

Notifica la desconexión del objeto. Dado que los objetos picoCORBA se implementan normalmente en *hardware*, se consideran "eternos" (*allways on*). Por tanto, un objeto picoCORBA jamás generará un mensaje de este tipo.

5.7. Mensaje de error (*MessageError*)

Los mensajes de error sirven para hacer saber a un emisor que el mensaje que ha enviado está mal formado y no sigue el formato establecido. También lo puede utilizar un servidor para indicar a un cliente que las peticiones que le está enviando corresponden a una versión de GIOP que el servidor no entiende. Estos mensajes pueden ser enviados tanto por el cliente como por el servidor.

PicoCORBA ignora los mensajes procedentes del cliente que estén mal formados o no correspondan a la versión GIOP 1.0 y, por tanto, no informa de este tipo de errores al cliente.

PicoCORBA se utiliza sólo en el servidor, por lo que la versión de GIOP la decide él. Tampoco puede generar mensajes mal formados pues los fabrica un autómata. Si llega a ocurrir, significa que la implementación es errónea y el objeto es inservible. En cualquier caso, no serviría de nada atender estos mensajes pues un servidor picoCORBA es incapaz de recuperarse de un fallo de esta índole.

Por tanto, los objetos picoCORBA ignoran los mensajes de error y nunca los generan.

6. Resultados experimentales

El prototipo realizado supone que el dispositivo sobre el que ubicar la implementación es capaz de obtener un flujo de bytes procedente de la red y devolver una respuesta, independientemente del protocolo de transporte que se utilice (TCP, SLIP, SNAP, etc).

El servidor picoCORBA dispone de un conjunto de procedimientos de usuario (los métodos de los objetos) y su misión es determinar cuál de ellos debe ejecutar en base al mensaje de petición entrante. Un dispositivo picoCORBA en el que sólo reside un objeto no necesita elegir el destinatario del mensaje, únicamente debe comprobar que la petición entrante se refiere realmente al objeto en cuestión.

El prototipo define un conjunto de operaciones para aplicar sobre el flujo de datos entrante. Haciendo uso de estas operaciones, se determina a qué objeto se refiere la petición y cuál es la operación que el cliente pretende invocar. Se trata de un pequeño conjunto de operaciones genéricas que se pueden reutilizar en varios puntos del flujo.

Con estas operaciones simples se puede analizar un mensaje GIOP y determinar la operación que se debe realizar.

- **Comparar**: Permite comparar dos cadenas para comprobar su igualdad. En caso de que no sea así, el mensaje se descarta inmediatamente.
- **Saltar**: Se aplica a conjuntos de bytes cuyo valor es irrelevante para la comprensión del mensaje y no se requiere ningún tipo de tratamiento. Por ejemplo: el campo `service_context`.
- **Comprobar/Decidir**: Determina un punto del flujo en el que deben cumplirse ciertas condiciones. Ante condiciones excluyentes, ese punto puede representar una bifurcación en el árbol de decisión. Representa un estado del autómata del que parten varias transiciones correctas.
- **Almacenar**: Almacena un conjunto de bytes para procesamiento posterior o para su uso en el mensaje de respuesta.

También es necesario disponer de una especificación de los objetos que debe implementar el dispositivo para poder generar la gramática correspondiente.

A partir de la gramática que define el conjunto de mensajes, es posible generar implementaciones del autómata reconocedor en varios lenguajes con idea de poder ubicar el servidor en plataformas muy distintas. En definitiva, y tal como muestra la figura 2, el compilador picoCORBA toma como entrada: a) El IDL que define la interfaz de comunicación; b) La especificación de los objetos residentes; b) Condicionantes de la plataforma destino y del entorno de ejecución.

Y con esta información genera una máquina de estados capaz de identificar a qué método de qué objeto se dirige cada mensaje de petición. La implementación generada incluye esqueletos para los procedimientos de usuario y funciones simples para la manipulación del flujo entrante y para la generación del mensaje de respuesta.

La figura 3 muestra cómo es un mensaje de petición GIOP desde el punto de vista de un ORB convencional y, como contrapartida, la figura 4 representa como ve picoCORBA el mis-

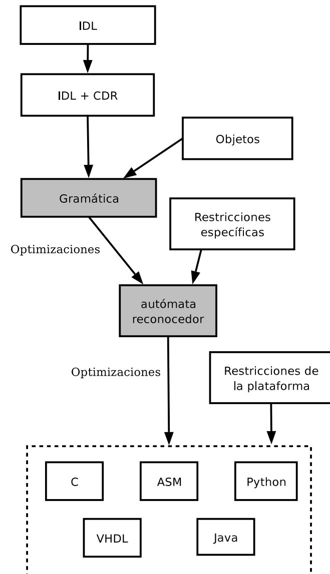

Figura 2: Entradas y salidas del generador de implementaciones

mo mensaje.

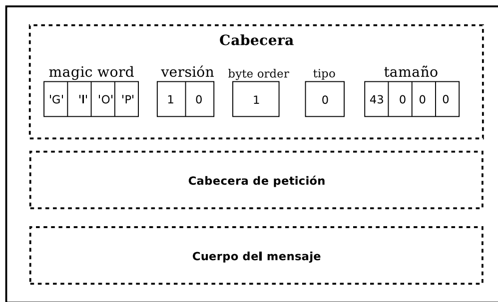

Figura 3: Interpretación de un mensaje de petición GIOP por un ORB convencional

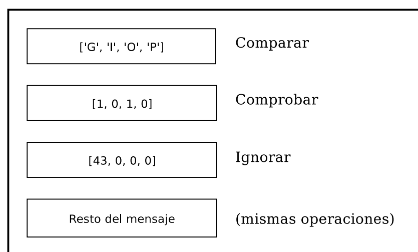

Figura 4: Interpretación de un mensaje de petición GIOP por picoCORBA

Cuando un mensaje ha sido reconocido y la máquina de estados llega a una hoja del árbol de decisión, se ejecuta el procedimiento de usuario correspondiente (un método del objeto).

Ver los mensajes de entrada y salida como flujos resulta muy conveniente para dispositivos con graves limitaciones de memoria. De ese modo, no es necesario almacenar prácticamente nada. El mensaje de petición se analiza "al vuelo" y algunas partes del mensaje de respuesta se pueden obtener copiando simplemente partes del mensaje de entrada. Las partes del mensaje que no se pueden "copiar" se obtienen ejecutando el procedimiento de usuario.

6.1. COMPILADOR DE IDL

El compilador IDL de picoCORBA contrasta fuertemente con los compiladores de ORB's convencionales. En picoCORBA, el código generado es autónomo y autocontenido, no hace uso de ninguna librería y además genera solamente el código imprescindible. Este código incluye además todo lo necesario para encapsulamiento de datos en CDR, tamaño y alineamiento de tipos (todo aquello que en un ORB convencional estaría en una librería).

El código generado no se atiene en absoluto a las especificaciones del *mapping* de OMG; de hecho, algunos de los lenguajes utilizados son ensambladores para distintos procesadores o microcontroladores. Por la misma razón, el código escrito por el programador para la implementación de los métodos de los objetos no es en absoluto portable, a menos que se vaya a utilizar en la misma plataforma.

Tal como se muestra en la figura 5, el compilador picoCORBA es responsable de:

- Convertir la especificación de la interfaz IDL a una representación intermedia.
- Añadir las operaciones propias de la interfaz *CORBA::Object* que se vayan a implementar.
- Evaluar todos los mensajes posibles una vez conocidos los objetos residentes y las interfaces que implementan.
- Obtener la gramática completa para el servidor (teniendo en cuenta todos los objetos), y optimizarla.
- Generar el autómata reconocedor para esa gramática, y optimizarlo.
- Generar la implementación del autómata en un lenguaje concreto teniendo en un cuenta un conjunto de restricciones. Dependiendo del lenguaje y plataforma también es posible realizar optimización sobre el código.
- Generar esqueletos de implementación para los procedimientos de usuario (métodos de los objetos).
- Proveer de funciones y macros auxiliares para facilitar al programador el acceso a los argumentos de las peticiones y para la generación de los mensajes de respuesta.

El proceso se divide en distintas etapas, generando código intermedio en cada una de ellas. Se ha elegido XML como lenguaje intermedio por su versatilidad y facilidad de tratamiento con múltiples herramientas.

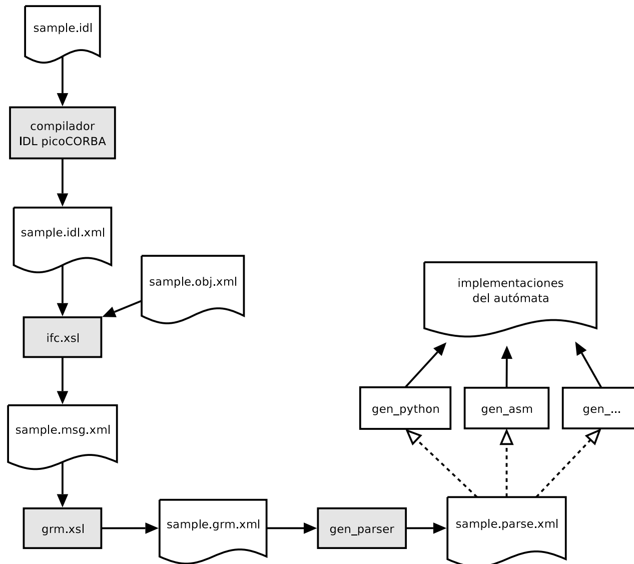

Figura 5: Esquema de las fases de transformación en la compilación con picoCORBA

```
module sampleModule {
  interface simpleIfc {
    void simple (in octet valor);
  };
  interface complexIfc {
    typedef sequence <octet> octetsec;
    short complex (inout octetsec data);
  };
  interface notUsedIfc {
         void notUsed(in octet data);
  };
};
```

Listado 2: `sample.idl`: Definición de las interfaces IDL para los objetos del ejemplo 4

En la primera fase se transforma el IDL original en XML añadiendo toda la información de tipos que corresponde a CDR (la representación de los tipos de datos de CORBA). El listado 2 es un fichero IDL de ejemplo y el 3 muestra parte del resultado.

En la segunda fase se combina lo anterior con la especificación de los objetos (listado 4) y se obtiene como resultado una colección de los mensajes soportados por cada objeto, tal como muestra el listado 5. Después se añade toda la información relativa al formato de los mensajes según lo explicado en la sección 5. y se genera un nuevo fichero en el que se describen detalladamente cada componente de todos los mensajes que debe reconocer el servidor, es decir, una gramática. Por último, a partir de la gramática se genera la implementación del autómata reconocedor correspondiente.

```
<idl stubs="1" skels="0" common="0"
  filename="sample.idl">
<gen_info module="orbit-idl-xml.c"/>
<list>
 <module>
  <ident name="sampleModule" NSQ="sampleModule"
         RID="IDL:sampleModule:1.0"/>
   <definition_list>
    <list><interface>
     <ident name="simpleIfc"
            RID="IDL:sampleModule/simpleIfc:1.0"/>
      <body><list>
       <op DCL="normal">
         <ident name="simple"
     RID="IDL:sampleModule/simpleIfc/simple:1.0"/>
           <parameter_dcls><list>
            <param DCL="(in)">
             <type value="octet"/><simple_declarator>
               <ident name="valor"
  RID="IDL:sampleModule/simpleIfc/simple/valor:1.0"/>
```

Listado 3: Fragmento de `sample.idl.xml`: Resultado de compilar el fichero `sample.idl` (listado 2)

```
<objects>
   <obj id="primero"
      interface="sampleModule::simpleIfc"/>
   <obj id="segundo"
      interface="sampleModule::complexIfc"/>
</objects>
```

Listado 4: `sample.obj.xml`: Ejemplo de especificación de los objetos de un servidor

```
<communication>
 <obj id="primero">
  <msg id="simple"
   RID="IDL:sampleModule/simpleIfc/simple:1.0">
   <param DCL="in" type="octet" name="valor"/>
  </msg>
 </obj>
 <obj id="segundo">
  <dcls based_on_type="sequence">
   <type name="octetsec"
    RID="IDL:sampleModule/complexIfc/octetsec:1.0"/>
  </dcls>
  <msg id="complex"
      RID="IDL:sampleModule/complexIfc/complex:1.0">
   <return_value type="short"/>
   <param DCL="inout" type="octetsec" name="data"/>
  </msg>
 </obj>
</communication>
```

Listado 5: `sample.msg.xml`: Mensajes válidos para los objetos especificados en el listado 4

7. Conclusiones y trabajos futuros

En este artículo se ha descrito una nueva metodología para el desarrollo de objetos CORBA en dispositivos de muy escasos recursos. Aunque el prototipo se centra en la implementación sobre microcontroladores de 8 bits, el modelo es fácilmente extendible a la generación automática de lógica programable, lo que permite utilizar picoCORBA para dispositivos

con fuertes demandas de ancho de banda (cámaras de ví deo, reproductores de audio, micrófonos de ambiente, etc.).

Los objetos picoCORBA se implementan como una simple máquina de estados, lo que garantiza un comportamiento muy determinista con respecto al tiempo. Esto hace que también sea una alternativa a considerar en el diseño de sistemas de tiempo real basados en componentes.

A pesar de la flexibilidad y bajo coste de esta alternativa, picoCORBA mantiene un alto grado de interoperabilidad con otros objetos CORBA desarrollados con ORB's estándar.

La metodologí a propuesta se ha desarrollado en el contexto del proyecto MIDAS2-TI/MARISA (TIN2004-07948-C05-02) para su aplicación a redes domóticas de nueva generación.

---

[1] OMG (*Object Management Group*), http://www.omg.org/
[2] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, edición 2.3, Junio 1999. Disponible en http://www.omg.org/, document id: 98-12-01.
[3] Sun Microsystems, *Jini Architecture Specification*, edición 1.2, Disponible en http://www.sun.com/,
[4] Open Services Gateway Initiative, *OSGi Service Platform*, edición 2.0, Octubre 2001, disponible en http://www.osgi.org/.
[5] Object Management Group, *Minimum CORBA Specification*, edición 2.3, Agosto 2002, Disponible en http://www.omg.org/, document id: 02-08-01
[6] M. Román, Fabio Kon, Roy H. Campbell, *Reflective Middleware: From Your Desk to Your Hand*, 2001.
[7] Fabio Kon, F. Costa, G. Blair, Roy Campbell. *The Case for Reflective Middleware*.
[8] Manuel Roman, M. Dennis, Mickunas, Fabio Kon and Roy Campell. *LegORB and Ubiquitous CORBA*, Febrero 2000.
[9] LegORB, disponible en http://choices.cs.uiuc.edu/2k/LegORB/.
[10] OpenFusion e*ORB, disponible en http://www.prismtechnologies.com/.
[11] J. Morena, F. Moya, J.C. López. *Implementación de un ORB para Dispositivos Empotrados*, Septiembre 2002.
[12] Haugan, Olav. *Configuration and Code Generation Tools for Middleware Targeting Small, Embedded Devices*, M.S. Thesis, December 2001.
[13] Tiny Internet Interface. Disponible en http://www.ibutton.com/TINI/index.html
[14] SaJe, Real-Time Native Java Execution. Disponible en http://saje.systronix.com/.
[15] E. Gamma, R.H., R. Johnson, J. Vlissides, *Design Pattens, Elements of Object-Oriented Software*. 1995, Addison-Wesley.