DISEÑO Y PROTOTIPO DE MIDDLEWARE BASADO EN CORBA PARA EL BUS CAN

TESIS DE GRADO EN INGENIERÍA INFORMÁTICA
ORIENTACIÓN EN SISTEMAS DISTRIBUIDOS

FACULTAD DE INGENIERÍA
UNIVERSIDAD DE BUENOS AIRES

TESISTA: **Sr. Martín ROUAUX**

DIRECTORA: **Prof. CC. María FELDGEN**

OCTUBRE 2005

Direcciones de contacto

Martín ROUAUX
martin.rouaux@redmondsoftware.com

María FELDGEN
maria.feldgen@gmail.com

Resumen

Este trabajo propone una nueva capa de aplicación para el bus CAN, basada en el middleware CORBA. Se construye un prototipo de ORB que, respetando la filosofía original y manteniendo las ventajas de CORBA, es capaz de satisfacer los requerimientos de un bus de campo y de las aplicaciones que sobre estos se construyen. El prototipo soporta un modelo tradicional de comunicación peer-to-peer pero también comunicaciones master / slave, mediante la definición de grupos de objetos. En ambos modelos se explotan las características especiales del bus CAN referentes a arbitración del bus, multicast / broadcast y message filtering. Por último, se compara en forma cualitativa la nueva capa propuesta con otras capas de aplicación existentes para el bus CAN.

Palabras clave: *buses de campo, CAN, capa de aplicación, sistemas distribuidos, middleware, ORB, CORBA.*

Abstract

This work proposes a new CORBA-based application layer for the CAN field bus. An ORB prototype is built following the original CORBA philosophy and conserving its main advantages. The prototype is capable of fulfilling the requirements for both, field buses and applications built on top of them. Support for a traditional peer-to-peer communication model is offered. A master / slave model is also supported through the definition of object groups. In both models, CAN features related to bus arbitration, multicast / broadcast and message filtering are exploited. Finally, a qualitative comparison is carried out between the layer proposed here and existing application layers for the CAN bus.

Keywords: *field buses, CAN, application layer, distributed systems, middleware, ORB, CORBA.*

Índice de Contenidos

1. Introducción

1.1 Objetivo

Los sistemas modernos de control distribuido para plantas de manufactura, aeronaves, automóviles, etc. requieren de redes comunicaciones de tiempo real para intercambiar información del sistema y señales de control entre los distintos componentes físicos del sistema (sensores, actuadores, controladores, etc.) que generalmente se encuentran distribuidos físicamente. Estos sistemas llamados NCS (Networked Control Systems) reemplazaron los sistemas de control con cableados punto a punto tradicionales de décadas anteriores, por la creciente demanda de modularidad, descentralización del control, diagnostico integrado, mantenimiento fácil y bajo costo. Una arquitectura NCS requiere cableado reducido y provee procesamiento distribuido. De este modo, se utiliza el poder de procesamiento de cada nodo permitiendo modularizar las funciones y usar interfases estándares para intercambiabilidad e interoperabilidad, usando modelos de dispositivos basados en objetos que separan claramente las funcionalidades genéricas y las especificas del tipo de dispositivo, de las especificas del fabricante [Lian 01]. Estos sistemas deben cumplir, además, con requerimientos especiales de confiabilidad y de cumplimiento de restricciones de tiempo [Kaiser 98]. Estas restricciones se refieren a la capacidad que deben tener los sistemas de generar resultados dentro de ciertos intervalos de tiempo, los cuales determinan su corrección y validez [Wainer 97].

Los sistemas de control distribuido son sistemas distribuidos que contienen una gran cantidad de dispositivos interconectados para realizar las operaciones deseadas. Al igual que un sistema distribuido tradicional, es una colección de máquinas (computadoras o dispositivos con capacidad de procesamiento) independientes y heterogéneas interconectadas por redes, que aparece ante los usuarios como una única máquina homogénea y coherente [Tanenbaum 96] [Tanenbaum 02]. En particular, para los sistemas de control distribuido, ciertas ventajas de los sistemas distribuidos tradicionales resultan especialmente relevantes. En primer lugar, un sistema distribuido, al estar compuesto por múltiples máquinas, permite evitar puntos únicos de falla, facilitándose así el cumplimiento del requerimiento de confiabilidad. En segundo lugar, los procesos normalmente requieren el control de múltiples dispositivos electromecánicos distribuidos físicamente. Un sistema distribuido presenta una arquitectura que se acomoda fácilmente a la distribución propia del proceso correspondiente. Finalmente, cada dispositivo puede ser controlado por un microcontrolador o microprocesador de menor capacidad de procesamiento y por lo tanto, de menor costo. Así se puede reducir el costo económico global de los sistemas [Kaiser 98].

Para los sistemas de control se usan redes de datos y redes de control dependiendo de la información que deben intercambiar. Las redes de datos se caracterizan por grandes paquetes, que se transmiten a altas velocidades, que generalmente no tienen restricciones de tiempo real, y en general, corresponden a la interacción entre las aplicaciones generales de la empresa e información de los sistemas de control. Las redes de control en cambio, que se usan para el control y monitoreo de los dispositivos, tienen pequeños paquetes, muy frecuentes entre una gran cantidad de nodos y que son críticos en el tiempo. La diferencia fundamental entre ambos tipos de redes, es la capacidad de soportar aplicaciones en tiempo real y críticas en el tiempo. Las métricas de los sistemas de redes que tienen impacto sobre sistemas de control incluyen demora en el acceso, tiempo de transmisión, tiempo de respuesta, demora del mensaje, colisión de mensajes (porcentaje de colisiones), porcentaje de mensajes descartados, tamaño del mensaje, utilización de la red y otros limites determinísticos.

Las redes pueden introducir niveles de servicios no confiables y dependientes del tiempo en términos de demoras, jitter o pérdidas. La calidad del servicio solicitada (Quality-of-Service o QoS) para la transmisión sobre la red puede mejorar el comportamiento en tiempo real de la red, pero su comportamiento seguirá sujeto a interferencia, a problemas pasajeros de ruteo, a flujos agresivos. Estas situaciones arbitrarias de la red pueden comprometer la estabilidad, seguridad y rendimiento de las unidades en un ambiente físico. El logro del control integrado con algoritmos de comunicaciones que compensen las arbitrariedades de los servicios de las redes, siguen siendo temas activos de investigación [ICELab].

Por consiguiente, las redes que interconectan los dispositivos que son controlados por el sistema de control deben cumplir con los siguientes dos criterios principales: demoras acotadas y transmisión garantizada, lo que significa que un mensaje debe transmitirse exitosamente dentro de un periodo acotado. La transmisión fallida o demora excesiva de uno o mas mensajes entre un sensor y un actuador, por ejemplo, puede deteriorar el rendimiento total del sistema o hacerlo inestable. Los protocolos de estas redes deber ser responsables de satisfacer los requerimientos de las respuestas criticas en el tiempo y de tiempo real sobre la red y de la calidad y confiabilidad de la comunicación entre los dispositivos de la red [Lian 01]. Las redes que actualmente cumplen con estas restricciones de periodicidad, de tiempo de respuesta, de jitter, etc. [Thomesse 99] por la utilización de técnicas para controlar el acceso al medio compartido, para la arbitración del mismo y para la priorización del tráfico entre dispositivos [Babiuch 00] son los buses de campo o buses industriales. Estos son protocolos determinísticos optimizados para mensajes cortos. Algunos ejemplos de buses industriales son BATIBUS, EIBUS, EHS, LON, CAN, P-Net, FIP, WorldFIP, Interbus-S, ASI, Profibus-PA, etc.

Dentro de la selección de buses industriales mencionados, CAN resulta una buena opción. En primer lugar, tiene una especificación estándar del bus [CAN 91]. Es aplicable en un amplio rango de sistemas de control distribuido, usados en vehículos, robots o bien en automatización industrial. También hay una amplia variedad de microcontroladores con adaptadores CAN incorporados, lo cual facilita la conexión a la red de los dispositivos involucrados en el proceso [Kaiser 98]. Además, utiliza un mecanismo de arbitración y control de acceso al medio para multicast y broadcast de mensajes en forma confiable y que garantiza que mensajes con alta prioridad siempre ganen el acceso el medio durante la arbitración y que su demora esté acotada. Por último, la especificación de CAN no define las capas superiores del sistema de comunicación [Etschberger 97]. Esto resulta conveniente a la hora de construir un sistema distribuido sobre CAN ya que hay libertad de elección de los protocolos a usar en capas superiores. Hay múltiples propuestas para estas capas, como por ejemplo, DeviceNet, CAL, CANopen, SDS, etc. [Etschberger 97].

Un sistema de control distribuido se puede construir según dos enfoques. El primero de ellos implica desarrollar únicamente los protocolos de comunicación entre los distintos dispositivos. Este enfoque no ofrece una visión única y coherente del sistema, o sea las características de cada dispositivo son visibles y deben tomarse en cuenta al implementar las aplicaciones. El otro enfoque consiste en usar una capa adicional de software sobre los protocolos de comunicaciones. Esta capa, que recibe el nombre de middleware [Tanenbaum 02], provee a las aplicaciones una visión uniforme y coherente de los elementos heterogéneos subyacentes (máquinas, sistemas operativos, sistemas de comunicación, etc.) [Sun 04]. Las principales ventajas de este tipo de software son interoperabilidad, transparencia, confiabilidad, disponibilidad, escalabilidad y abstracción [Bray 97]. Resulta de especial relevancia la transparencia de distribución que consiste en el desconocimiento por parte de las aplicaciones de la ubicación física real de los recursos que estas usan [Tanenbaum 02]. Dado que los dispositivos, en estos sistemas, se modelan como objetos [Kaiser 98], surge que un middleware orientado a objetos es el más conveniente para la modelización y construcción de estos sistemas. Un middleware de este tipo recibe el nombre de Object Request Broker (ORB).

Existen muchos tipos de ORB como por ejemplo, Common Object Request Broker Architecture (CORBA), Distributed Component Object Model (COM/DCOM) o Java Remote Method Invocation (RMI). Dentro de estas opciones, la más adecuada resulta CORBA por los siguientes motivos: existe una especificación estándar [Corba Core 02], no es una solución propietaria, ofrece interoperabilidad entre implementaciones inclusive de fabricantes distintos, disponibilidad de implementaciones de código abierto para plataformas diversas y soporte para múltiples lenguajes de implementación de objetos. DCOM requiere exclusivamente plataforma de sistemas operativos Microsoft, la cual no es habitualmente la plataforma de operación de muchos de los dispositivos que requieren de sistemas operativos en tiempo real. RMI es multiplataforma pero fue diseñado para operar sobre redes de datos con protocolos TCP/IP, que no cumplen con los requisitos de una red de control.

El objetivo del presente trabajo es, partiendo de un middleware existente que usa un sistema de comunicación estándar, adaptarlo para que funcione sobre el sistema de

comunicación de un bus industrial, permitiendo así la construcción de sistemas de control distribuido. Se diseñó un middleware basado en la especificación de CORBA, capaz de funcionar sobre el bus CAN. Se construyó un simulador de un bus CAN. Se implementó un prototipo del diseño en cuestión, sobre un simulador de bus CAN, partiendo de una implementación existente de CORBA de código abierto (ORBit versión 0.5.13 [ORBit 04] para Linux).

1.2 Organización del trabajo

El trabajo se encuentra organizado de la siguiente forma:

- En el capítulo 2 se describen los buses de campo o buses industriales. Primero, se define y caracteriza un bus industrial, se describe en detalle el bus CAN y, se analizan dos capas superiores existentes para construir un sistema basado en CAN, como CANopen y DeviceNet.

- En el capítulo 3 se define el middleware y se caracterizan los distintos tipos, incluyendo las ORBs. Se describe en detalle el funcionamiento y las características de CORBA.

- En el capítulo 4 se plantean los requerimientos que debe cumplir el diseño de un middleware basado en CORBA que pueda funcionar sobre una red CAN.

- El capítulo 5 describe el diseño propuesto, con los protocolos adaptados para cada una de las capas y sus respectivas características de implementación.

- El capítulo 6 contiene una comparación del middleware propuesto con la especificación original de CORBA y con las capas superiores para CAN presentadas en el capítulo 2, CANopen y DeviceNet.

- En el capítulo 7 se presentan las conclusiones del trabajo y se evalúa en que medida el prototipo cumple con los requerimientos planteados en el capítulo 4.

- El apéndice A incluye una descripción del código fuente del prototipo, su organización y el método usado para compilar los ejemplos presentados en el capítulo 6.

2. Buses Industriales

Resumen del capítulo

En este capitulo se describen las características de un bus de campo, sus funciones y los requerimientos que debe cumplir. El bus seleccionado para el presente trabajo es CAN por lo cual se describe este bus en detalle.

Por último, el capitulo describe las características que debería presentar una capa superior para el bus CAN, presentando algunas implementaciones existentes:

- **CANopen**: desarrollada originalmente como un proyecto Esprit (financiado por la Unión Europea), bajo el auspicio de Bosch. Actualmente, la especificación está a cargo de CAN in Automation (CiA), organismo internacional e independiente de usuarios y fabricantes dedicado a promover el uso del bus CAN, mediante la definición y estandarización de capas superiores para el mismo.

- **DeviceNet**: desarrollada originalmente por Allen-Bradley, hoy en día es mantenida por un organismo independiente llamado Open DeviceNet Vendor Association (ODVA).

2.1 Características

2.1.1 Definición de bus industrial

Un bus industrial, bus de campo o fieldbus es un sistema de comunicación o red en tiempo real, basado en una estructura en capas derivada del modelo de referencia ISO/OSI, que permite conectar dispositivos de campo como pueden ser PLCs, reguladores, sensores, actuadores, etc. [Thomesse 99].

A la hora de diseñar los distintos buses industriales, se adoptaron dos posturas contrapuestas:

- Un bus de campo es visto sólo como una red para simplificar el cableado entre dispositivos. Este enfoque termina reduciendo el bus a un mecanismo de control de acceso al medio de transmisión compartido y la aplicación, a un conjunto de procesos aislados e independientes que se comunican por medio del bus. En los desarrollos iniciales, esta era la postura adoptada.

- Un bus de campo es visto como la base sobre la que se construyen sistemas distribuidos de tiempo real. Los requerimientos de las aplicaciones permiten definir los servicios que debería ofrecer el bus para una implementación distribuida adecuada. Normalmente los servicios definidos son lectura y escritura de variables o listas de variables con coherencia temporal y consistencia espacial. El objetivo de este enfoque es dar, a todas las entidades que forman la aplicación, la misma visión del sistema en un momento dado, independientemente de su ubicación.

Un bus industrial está basado en una estructura en capas derivada del modelo de referencia ISO/OSI. La mayoría de los buses se describen en base a un modelo simplificado de sólo 3 capas [Thomesse 99]:

- **Capa Física**

  Esta capa provee transmisión transparente de cadenas de bits entre entidades de la capa data-link. Existen distintas clases de servicio a nivel de la capa física, en lo referente a tipo de transmisión (sincrónica o asincrónica), modo de operación (full-duplex, half-duplex y simplex), topología (punto a punto o multipunto) y bit rate (constante o variable) [ISO7498]. Esta capa incluye el Medium Attachment Unit (MAU) y la Physical Layer Signalling (PLS).

- **Capa Data-link**

  Esta capa provee independencia de la capa física subyacente, transferencia de datos transparente (sin restricciones sobre contenido, formato o codificación) y confiable (maneja errores, perdidas, reordenamiento, etc.), selección de la calidad de servicio y direccionamiento local (dentro de la misma configuración data-link) [ISO7498].

  La capa ofrece dos tipos de servicios: orientado a conexión y sin conexión. Está formada por dos subcapas: el Medium Access Control (MAC) y el Logical Link Control (LLC).

- **Capa de Aplicación**

  Dado que es la capa superior del modelo, no se la puede describir en términos del servicio que ofrece a capas superiores. Por este motivo, el estándar plantea una descripción alternativa en cual, los aspectos de un proceso de aplicación que resultan relevantes para el modelo ISO/OSI se agrupan en un elemento abstracto llamado entidad de aplicación.

  Luego, los procesos intercambian información a través de las entidades de aplicación que los componen. Estas entidades ofrecen servicios orientados a conexión y sin conexión que incluyen facilidades como identificación, sincronización, negociación de calidad de servicio y responsabilidades de las contrapartes, selección de modo de diálogo, etc. [ISO7498].

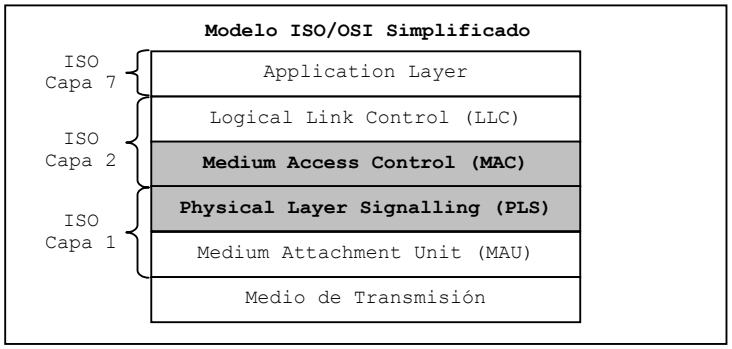


Ilustración 1 – Modelo de referencia ISO/OSI simplificado para buses industriales

En cada una de estas capas existen muchas variantes a nivel de servicios y de protocolos. Las opciones seleccionadas en cada una de ellas terminan definiendo un perfil o tipo de bus. Algunos ejemplos de perfiles son FIP, WorldFIP, Profibus, CAN, P-Net, etc.

2.1.2 Requerimientos de un bus industrial

Desde el punto de visto de un desarrollador de aplicaciones distribuidas, el bus en sí no es importante sino el sistema que se construye sobre éste. La solución con un bus debe ser más barata y funcionar mejor que una solución clásica. Para conseguir esto, el bus debería cumplir, inicialmente al menos, con los siguientes requerimientos [Thomesse 99] [Babiuch 00]:

- **Seguridad, disponibilidad y dependabilidad**

  Este requerimiento apunta al temor existente que, en caso de una falla, se produzca una pérdida de control. Podría hablarse, en forma general, de confiabilidad.

  La dependabilidad es una propiedad cualitativa de un sistema computacional que permite describir o medir la confianza que se puede tener sobre el servicio que el sistema provee [Barbacci 95]. Se suele decir que un sistema computacional es dependable, si, con alta probabilidad, exhibe un comportamiento de acuerdo a su especificación [Veríssimo 01].

  Hay que distinguir entre la dependabilidad del bus en sí y del sistema construido sobre éste. La primera está dada por la capacidad de detección y recuperación de errores de transmisión y la capacidad de cumplir restricciones de tiempo. La segunda, tiene que ver con el perfil del bus en uso y la capacidad de éste de notificar los errores a los procesos de aplicación.

- **Mantenibilidad y flexibilidad**

  Los buses deben hacer más simple el mantenimiento de los sistemas. Mantenimiento involucra actividades como detección y reparación de fallas, y configuración de dispositivos y componentes. Respecto al requerimiento de flexibilidad, se apunta a que el sistema permita la introducción de cambios con relativa facilidad.

- **Modularidad y capacidad de evolución**

  Un bus debe mejorar la capacidad de evolución de una aplicación. El cableado se simplifica con lo cual es más fácil agregar nuevos dispositivos y modificar los existentes. Se relaciona también con el requerimiento anterior de flexibilidad.

- **Interoperabilidad e intercambiabilidad**

  Para construir sistemas es necesario incorporar componentes de diversos fabricantes. El requerimiento de interoperabilidad apunta a que estos sean compatibles, proveyendo exactamente los mismo protocolos, con los mismos mensajes y la misma semántica.

  Por otro lado, la intercambiabilidad se refiere a permitir el reemplazo de dispositivos y componentes fallados por otros similares, inclusive de fabricantes distintos.

- **Mejor rendimiento a menores costos**

  La solución provista por un bus industrial debe ser más barata y tener mejor rendimiento que una solución clásica. Sino, un desarrollador de aplicaciones distribuidas no tendría motivos para adoptarla.

Estos requerimientos son generales y aplicables a la mayoría de los sistemas, redes o máquinas. Un bus industrial, al ser una red en tiempo real, tiene que cumplir además con una serie de restricciones o requerimientos temporales:

- **Periodicidad**

  La aplicación requiere que se transmitan datos identificados (como por ejemplo, variables de estado) con cierta periodicidad que debe ser respetada por el bus de campo.

- **Variación del período**

  El período puede o no variar entre un límite inferior y un límite superior. Esto también se conoce como jitter.

- **Tiempo de respuesta**

  Tiempo o retraso entre un pedido (request) y su respuesta (response). Se puede medir del lado del proceso cliente o bien, del servidor, como se indica en el siguiente esquema:

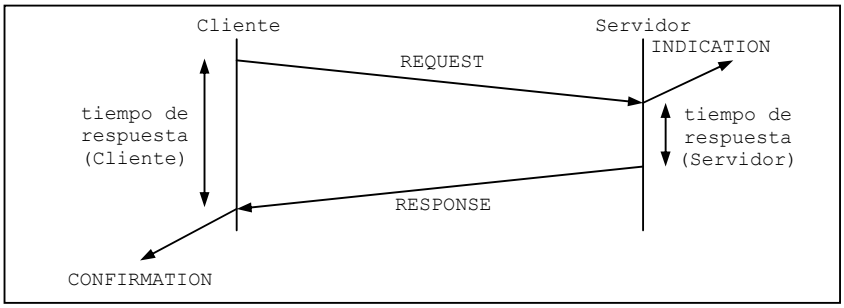


Ilustración 2 – Tiempos de respuesta en cliente y servidor

- **Frescura y rapidez**

  Califica el valor de una variable. Indica que el valor ha sido producido, transmitido y recibido dentro de una ventana de tiempo requerida por la aplicación.

- **Coherencia temporal**

  Propiedad relacionada con dos o más ocurrencias de un evento, las cuales, deben darse dentro de una ventana de tiempo (producción de variables, transmisión de las mismas, etc.).

Un bus debe transmitir datos entre sensores, actuadores, controladores y otros dispositivos. Estos intercambios son conocidos e identificados al momento de especificar la aplicación. Por este motivo, reciben el nombre de datos identificados y son normalmente manejados bajo modelos de comunicaciones cliente / servidor, productor / consumidor o master/slave.

De igual forma, en cualquier aplicación existen ciertos intercambios que no son conocidos durante la especificación. Normalmente, estos corresponden a actividades de configuración, mantenimiento, downloading y uploading de programas o firmware, etc.

Dentro de los intercambios de datos identificados, se distinguen dos tipos de tráfico que un bus debería soportar:

- **Tráfico periódico**

  La mayoría de los datos identificados están involucrados en la entrada o salida de algoritmos de control. Normalmente, son transmitidos periódicamente con o sin jitter (en sistemas centralizados clásicos, se hace lo mismo pero con polling). Los sistemas que presentan este tipo de tráfico son llamados time-triggered.

- **Tráfico aperiódico**

  Si todos los datos de la aplicación son transmitidos en forma periódica, la carga global del bus puede ser elevada. El tráfico aperiódico tiene precedencia (valores que no cambian periódicamente y que sólo son transmitidos cuando se produce un cambio). Los sistemas que presentan este tipo de tráfico son llamados event-triggered.

2.1.3 Perfiles de los buses industriales

El perfil o tipo de bus está dado por las características que el diseñador le da a los servicios y protocolos de cada una de sus capas.

La simplificación del modelo ISO/OSI restringe las posibilidades del bus. Por ejemplo, sin capa de presentación no es posible convenir representaciones independientes de los datos intercambiados y conseguir interoperabilidad, uno de los requerimientos mencionados previamente. De igual forma, está restringida la posibilidad de ruteo y de control de flujo end-to-end, dadas las ausencias de una capa de red y de transporte respectivamente. En algunos buses, las funciones de estas capas existen aunque no se distinguen claramente.

En esta sección se presentan las opciones existentes para los buses para cada una de las capas del modelo simplificado ISO/OSI [Thomesse 99].

- **Capa Física**

  Dependiendo de la aplicación distintos protocolos físicos pueden ser usados (distancias, bit rates, cableado y topología). Hay algunos esfuerzos internacionales de estandarización de esta capa por parte de la International Electrotechnical Commission (IEC) en la norma IS 1158.2 de 1993. Otro estándar usado para esta capa es el ISO 11898 [CiA 04] [CANopen 99].

  Vale aclarar que el presente trabajo no incluye descripciones exhaustivas de capas físicas para buses industriales. Los conceptos planteados a lo largo del trabajo, así como las soluciones propuestas, son aplicables independientemente del estándar de capa física con el que se trabaje. En fin, el middleware propuesto es capaz de operar sobre cualquier capa física para el bus CAN.

- **Capa Data-link**

  Esta capa se subdivide en las subcapas Medium Access Control (MAC) y Logical Link Control (LLC). Los protocolos de la subcapa MAC se basan en alguna de las siguientes clases:

  - **Acceso al medio sin control**: se usa Carrier Sense Multiple Access (CSMA) en BATIBUS, EIBUS, EHS, LON y CAN / VAN en redes embebidas para automóviles. Normalmente se usa CSMA-CA (Collision Avoidance). Se desarrollaron otras variantes de CSMA para poner un límite superior a los frames que colisionaron.

  - **Acceso al medio controlado**: son los usados comúnmente en fieldbuses grandes. El control puede ser centralizado o descentralizado como en Profibus-FMS, que utiliza pasaje de token. Se puede construir una topología de anillo sobre el bus lineal que permite cumplir con la periodicidad si todas las estaciones pasan el token en el tiempo requerido. En un esquema de control descentralizado una estación está a cargo de la administración del bus, la cual no es una configuración robusta. Algunos ejemplos son P-Net, WorldFIP, Interbus-S, ASI y Profibus-PA.

  En la subcapa MAC también se define el direccionamiento de los mensajes. Este puede basarse en una dirección asignada a cada componente o dispositivo conectado al bus o bien, en un identificador asociado a cada dato intercambiado. Este último enfoque es usado en los buses WorldFIP, CAN y Batibus.

Respecto a la subcapa LLC, es común que en los buses no se la distinga claramente de la capa MAC. Esta capa ofrece los siguientes servicios:

- **Tipo 1**: ofrece un servicio orientado a conexión, sin confirmación (acknowledgement) y con detección de fallas de transmisión en el receptor.

- **Tipo 2**: ofrece un servicio orientado a conexión, con acknowledgement y con detección de fallas de transmisión en el emisor.

- **Tipo 3**: ofrece un servicio sin conexión pero con acknowledgement. Es el tipo más común en los buses de campo.

- **Capa de Aplicación**

  En esta capa, el diseñador dispone de dos modelos de comunicación distintos. En primer lugar, el modelo cliente / servidor que implementa comunicación punto a punto (point-to-point) y que da preferencia al tráfico que no es de tiempo real. En segundo lugar, está el modelo productor / consumidor que da preferencia a los datos identificados y al tráfico de tiempo real.

2.2 Controller Area Network (CAN)

2.2.1 Introducción

Controller Area Network (CAN) es un protocolo de comunicaciones seriales diseñado para soportar, en forma eficiente, sistemas de control distribuido con un alto nivel de seguridad. Fue desarrollado originalmente por la empresa Robert Bosch GmbH de Alemania a principios de la década del 90 para su uso en la industria automotriz aunque hoy en día, también es usado como un bus industrial general.

Su arquitectura abierta y la posibilidad que se le da al usuario de hacer ciertas elecciones sobre los protocolos de la capa de aplicación, le dan una gran flexibilidad. Puede transmitir a altas velocidades, a 1Mbit/s, en distancias cortas, del orden de los 40m. De igual forma, puede funcionar a bajas velocidades, 5Kbits/s, sobre distancias mayores, de hasta 10km.

De acuerdo a los perfiles de bus industrial [Thomesse 99] descriptos en la sección anterior, CAN ofrece en la subcapa MAC, acceso al medio sin control mediante un mecanismo similar a CSMA. En la subcapa LLC, ofrece un servicio Tipo 3 o sea, sin conexión pero con acknowledgement y detección de fallas.

La especificación de CAN [CAN 91] se divide en dos partes llamadas “A” y “B”. La primera, contiene la especificación de mensajes estándar. La segunda, describe los mensajes estándar y los extendidos. Este último tipo de mensajes evita a los diseñadores tener que asumir compromisos a la hora de definir esquemas de nombres estructurados (por limitaciones de tamaño).

2.2.2 Características básicas

CAN también adopta un modelo de capas y se limita a especificar parte de la capa data-link y de la capa física [CAN 91] [DevNet1 99]:

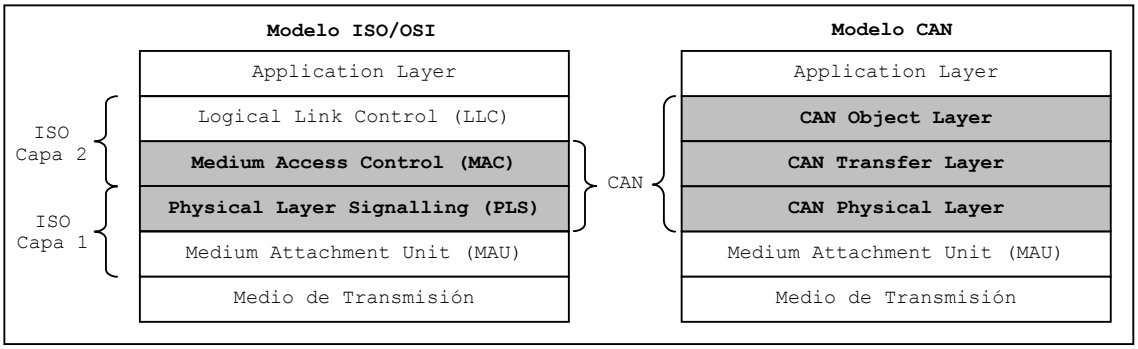


Ilustración 3 - Comparación entre ISO/OSI y CAN

El objetivo de la especificación es asegurar la compatibilidad (a nivel eléctrico e interpretación de datos transferidos) entre distintas implementaciones del protocolo. Por este motivo, CAN define las capas PLS, para obtener compatibilidad a nivel eléctrico, y MAC, para la compatibilidad de los datos transferidos. Dentro de estas dos capas del modelo ISO/OSI, CAN plantea su propio modelo de capas:

Capa	Funcionalidad
CAN Object Layer	• Determinar que mensajes se deben transmitir. • Decidir que mensajes recibidos por la capa inferior son usables (Messsage Filtering). • Proveer una interfase al hardware relacionado con la aplicación.
CAN Transfer Layer	• Control del framing. • Arbitración del medio compartido. • Control de errores. • Señalización de errores. • Confinamiento de fallas.
Physical Layer	• Transferencia de los bits entre los diferentes nodos en lo que respecta a todas las propiedades eléctricas.

Tabla 1 - Modelo de capas del protocolo CAN

La especificación se concentra en la CAN Transfer Layer e incluye algunas observaciones sobre como ésta repercute sobre las capas superiores e inferiores. El usuario del protocolo tiene libertad para hacer cambios en la CAN Object Layer así como en la Physical Layer lo cual, la da al protocolo un amplio grado de flexibilidad. Se pueden usar distintos medios de transmisión, diferentes criterios de selección de mensajes, etc. Se puede decir que CAN se limita a especificar la capa data-link, dejando libertad de elección en la capa física y en la capa de aplicación.

Respecto a la capa física, vale aclarar que los conceptos planteados a lo largo del trabajo, así como las soluciones propuestas para el middleware, son aplicables independientemente de la Physical Layer sobre la cual se implemente el bus CAN.

A continuación se incluye una lista de las características principales del protocolo y de las redes que con él se construyen:

- **Bus CAN**

  Un bus CAN consiste en un único canal que transporta bits en forma serial al cual pueden conectarse varios nodos. El límite de nodos conectados no está dado por el protocolo sino por la tolerancia a demoras (delays) y las limitaciones eléctricas provenientes del medio de transmisión usado.

  Respecto al medio de transmisión, el estándar no fija cuál se debe usar ni cómo se implementa. Existen múltiples opciones como un cable único, cables diferenciales, fibra óptica, etc.

  Para garantizar la compatibilidad a nivel eléctrico el estándar incluye información sobre la señalización en el bus. Este, en un momento dado, puede tener uno de dos valores lógicos complementarios: “dominante” o “recesivo”. Durante la transmisión simultánea de bits distintos, gana en el bus el valor “dominante”. Los valores físicos que representan estos valores lógicos no se definen ya que dependen de la implementación del bus.

Los nodos transmisores controlan constantemente que el valor que se ve en el bus sea el que ellos transmitieron. O sea, no puede suceder que se haya enviado un bit "recesivo" y se lea del bus uno "dominante". Esto sólo sería posible si hay más de un nodo transmitiendo en el mismo momento.

- **Mensajes y ruteo**

La información se envía por el bus en mensajes de formato fijo de longitud variable pero limitada. Cada mensaje incluye un campo IDENTIFIER, el cual define una prioridad estática al momento de acceder al bus y competir con otros mensajes.

Un nodo CAN no hace uso de ninguna información sobre la configuración del sistema, como podrían ser las direcciones de los nodos, cuando tiene que enviar un mensaje. Esto tiene las siguientes consecuencias:

- Flexibilidad del sistema: se pueden agregar nodos a una red CAN sin necesidad de cambios en software o hardware (configuración de direcciones).

- Ruteo de mensajes: el contenido de un mensaje es nombrado mediante el campo IDENTIFIER el cual no indica el destino del mensaje pero si el significado de los datos contenidos. Entonces, todos los nodos pueden decidir si aceptan o no un mensaje mediante Message Filtering (funcionalidad de la capa Object Layer).

- Multicast y broadcast confiables: en base a la técnica anterior, cualquier cantidad de nodos pueden recibir un mismo mensaje. Aparte, en una red CAN está garantizado que un mensaje es aceptado por todos los nodos o por ninguno, por el control de errores provisto por la Transfer Layer.

- Acknowledgement: todos los receptores controlan la consistencia del mensaje que se está recibiendo y generan un acknowledgement para los mensajes consistentes o bien, levantan un flag para los que no lo son. Si algún receptor no genera el acknowledgement, todos los otros receptores descartan el mensaje. Esto también ayuda a implementar el multicast y broadcast confiables.

- **Arbitración del medio y priorización de los mensajes**

Un bus CAN es multimaster. Esto quiere decir que cuando el bus está libre cualquier nodo puede comenzar a transmitir un mensaje. Si dos nodos comienzan exactamente al mismo tiempo, se usa un mecanismo de arbitración bitwise sobre el campo IDENTIFIER (a nivel de bits) que garantiza que no se pierde ni tiempo ni información.

Los transmisores controlan los bits que envían con los que se ven en el bus. Cuando el mismo valor se detecta, el nodo puede seguir transmitiendo. Cuando se envía un valor "recesivo" pero se ve uno "dominante" en el bus, el nodo perdió la arbitración y debe dejar de transmitir. De esta forma, el nodo con el mensaje con mayor prioridad gana el control del bus y se transforma en el único master.

- **Seguridad y confiabilidad**

- Detección de Errores: para dotar de mayor seguridad y confiabilidad al bus se tomaron las siguientes medidas: monitoreo (transmisores controlan bits a transmitir con bits leídos en el bus), CRC (Cyclic Redundancy Check), Bit Stuffing y Message Frame Check.

- Señalamiento de Errores y Tiempo de Recuperación: cualquier nodo que detecte un mensaje corrupto lo indica al resto levantando un flag en el mismo mensaje. Hay retransmisión automática por lo cual, el tiempo de recuperación de errores está acotado.

- Confinamiento de Fallas: los nodos pueden distinguir entre fallas temporales y permanentes. Los nodos que se auto detectan como defectuosos pueden desconectarse del bus.

2.2.3 Transferencia de mensajes

Un nodo transmisor o emisor es aquel que origina un mensaje. Cuando el nodo decide transmitir, adquiere esta condición y la mantiene hasta que el bus vuelva a estar libre o hasta que pierda la arbitración. Un nodo es receptor de un mensaje si no es el transmisor del mismo y el bus no está libre.

Emisores y receptores pueden intercambiar cuatro tipos distintos de frames o mensajes en CAN. Además, se define la secuencia de bits que se intercala entre los mensajes anteriores para indicar a los nodos que el bus se encuentra libre.

Los posibles tipos de frame son:

- DATA FRAME
- REMOTE FRAME
- ERROR FRAME
- OVERLOAD FRAME

A continuación se incluye una descripción en detalle de cada uno de los tipos de frame antes presentados:

- **DATA FRAME**

  Contiene datos que van de un emisor a un receptor. Un frame de este tipo se separa de un REMOTE FRAME mediante un INTERFRAME SPACE. Está formado por siete campos de bits diferentes:

  - START OF FRAME: marca el comienzo del frame. Consiste en un único bit “dominante”. Un nodo puede comenzar a transmitir solo cuando el bus está libre. Todos los nodos se sincronizan con el flanco ascendente del START OF FRAME de la estación que transmite primero.

  - ARBITRATION FIELD: en un mensaje de la Parte A, este campo de arbitración está formado por los subcampos IDENTIFIER y RTR-BIT. La longitud del IDENTIFIER es de 11 bits en un mensaje de la Parte A.

- IDENTIFIER: los bits se definen en el orden ID-10 hasta ID-0 o sea, el bit menos significativo es ID-0. Los bits ID-10 a ID-4 (los 7 mas significativos) deben ser todos no “recesivos”. Como “0” es el valor dominante y “1” el recesivo, los valores más chicos en este campo de arbitración tienen mayor prioridad que los más grandes. Así, un mensaje con este campo en “11111110000” le gana siempre el bus a otro con “11111110001” y así sucesivamente.

- RTR-BIT (Remote Transmission Request bit): en un DATA FRAME debe ser “dominante” (al contrario de un REMOTE FRAME).

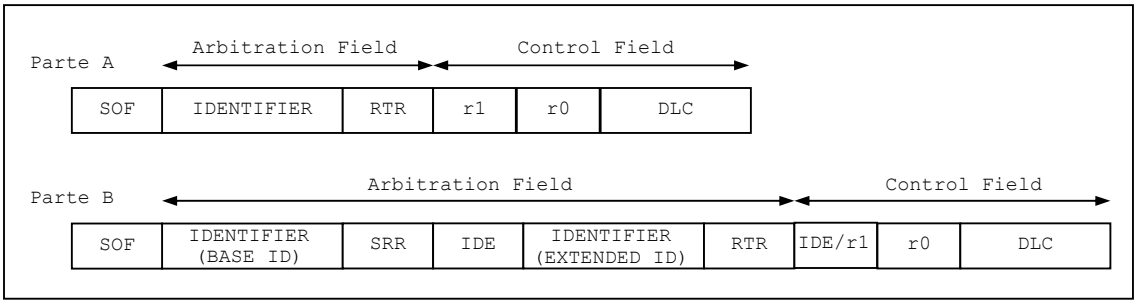


Ilustración 4 - Campo de arbitración de CAN (Partes A y B)

En un mensaje de la Parte B, este campo de arbitración está formado por los subcampos IDENTIFIER, SRR-BIT, IDE-BIT y RTR-BIT. La longitud del IDENTIFIER es de 29 bits. Los bits se definen en el orden ID-28 hasta ID-0 o sea, el bit menos significativo es ID-0.

- IDENTIFIER (Parte A): de ID-28 hasta ID-18 corresponden a los bits del IDENTIFIER de la Parte A siendo ID-18 el menos significativo. Los bits ID-28 a ID-22 (los 7 mas significativos) deben ser todos no “recesivos”.

- IDENTIFIER (Parte B): a diferencia del identificador estándar, este está formado por 29 bits separados en dos secciones: BASE ID (coincide con el IDENTIFIER de la Parte A) y EXTENDED ID (los siguientes 18 bits numerados de ID-17 a ID-0).

- RTR-BIT (Remote Transmission Request bit): en un DATA FRAME debe ser “dominante” (al contrario de un REMOTE FRAME).

- SRR-BIT (Substitute Remote Request bit): debe ser un bit “recesivo”. Reemplaza al RTR-BIT de los frames de formato estándar en los de formato extendido. Por este motivo, si hay una colisión entre el IDENTIFIER de un frame estándar y el BASE ID de un frame extendido, la arbitración la gana siempre el estándar (el RTR-BIT es siempre “dominante” mientras que el SRR-BIT es siempre “recesivo”).

- IDE-BIT (Identifier Extension bit): en los frames estándar, este bit se transmite como “dominante” mientras que en los extendidos, se transmite como “recesivo”. Permite saber si se trata de un mensaje de la Parte A o de la B.

- CONTROL FIELD: consiste de 6 bits. Contiene el DATA LENGTH CODE y dos bits reservados para uso futuro (ambos deben ser “dominantes”). El DATA LENGTH CODE contiene la longitud en bytes del DATA FIELD cuyos valores van desde 0, codificado como “dddd”, hasta 8, codificado como “rddd”.

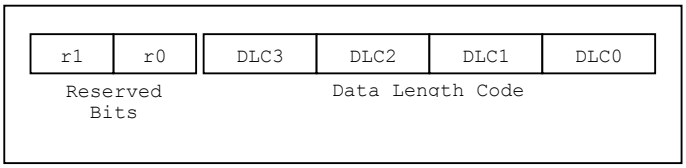


Ilustración 5 - Campo de control de CAN (Partes A y B)

En los mensajes de la Parte B de CAN, el bit r1 se usa para el IDE-BIT (Identifier Extension Bit) ya explicado en el campo de arbitración.

- DATA FIELD: este campo contiene los datos a ser transferidos en el DATA FRAME. Puede contener de 0 a 8 bytes de información comenzando por el MSB.

- CRC FIELD: contiene el código de redundancia cíclico usado para detectar errores en los mensajes CAN. Está formado por los subcampos CRC SEQUENCE y CRC DELIMITER.

  - CRC SEQUENCE: la secuencia se deriva del código BCH, un código de redundancia cíclica apto para frames de menos de 127 bits. Los coeficientes del polinomio a dividir se obtienen de la secuencia formada por START OF FRAME, ARBITRATION FIELD, CONTROL FIELD y DATA FIELD sin bit stuffing y los 15 coeficientes de menor grado son 0. El polinomio generador (el divisor) es:

$$G(x) = x^{15} + x^{14} + x^{10} + x^8 + x^7 + x^4 + x^3 + 1$$

Ecuación 1 – Polinomio generador de CRC en CAN

  El resto de la división es el valor de CRC SEQUENCE.

  - CRC DELIMITER: es un único bit “recesivo” que marca el fin de la secuencia de CRC.

- ACK FIELD: está formado por dos bits, el ACK SLOT y el ACK DELIMITER. El nodo transmisor envía en este campo dos bits “recesivos”. El receptor que obtiene un mensaje válido envía el ACK al transmisor poniendo un bit “recesivo” en el ACK SLOT. El ACK DELIMITER siempre es un bit recesivo (el ACK SLOT está rodeado por bits “recesivos”).

- END OF FRAME: este campo está formado por 7 bits recesivos.

- **REMOTE FRAME**

  Es enviado por un nodo conectado al bus requiriendo la transmisión de un DATA FRAME con el mismo IDENTIFIER. Está formado por 6 campos similares a los del DATA FRAME:

  - START OF FRAME
  - ARBITRATION FIELD
  - CONTROL FIELD
  - CRC FIELD
  - ACK FIELD
  - END OF FRAME.

  Al contrario de los DATA FRAMES, en estos frames el RTR-BIT del campo ARBITRATION FIELD debe ser un bit “recesivo” (así se distinguen ambos tipos de frames uno del otro). No es necesario el campo DATA FIELD ya que se está solicitando un dato, no enviando uno.

- **ERROR FRAME**

  Es transmitido por cualquier nodo cuando se detecta un error en el bus. Consiste únicamente de dos campos: ERROR FLAG (superposición de flags de error de los diferentes nodos) y ERROR DELIMITER. El campo ERROR FLAG puede tener dos formas:

  - ACTIVE ERROR FLAG: consiste en 6 bits “dominantes” consecutivos. Este flag es transmitido por nodos cuyo estado es ERROR ACTIVE.
  - PASSIVE ERROR FLAG: consiste en 6 bits “recesivos” consecutivos a menos que un nodo sobrescriba con un bit “dominante”. Este flag es transmitido por nodos cuyo estado es ERROR PASSIVE. Más adelante se detallarán los posibles estados de un nodo.

  La existencia de 6 bits consecutivos de igual polaridad viola la regla de bit stuffing usada en la codificación del bus CAN. Por esto, todas las otras estaciones también detectan el error y comienzan a transmitir un ERROR FLAG por su cuenta. De esta forma, la longitud de este campo puede variar entre 6 y 12 bits. El campo ERROR DELIMITER consiste en 8 bits “recesivos” consecutivos.

- **OVERLOAD FRAME**

  Es usado para insertar un delay adicional entre un mensaje DATA FRAME o REMOTE FRAME y el siguiente. Está formado únicamente por dos campos: OVERLOAD FLAG y OVERLOAD DELIMITER. Existen dos tipos distintos de condición de overload en el bus que llevan a la transmisión de un OVERLOAD FLAG:

- El estado interno de un nodo receptor que requiere un delay adicional antes del próximo DATA FRAME o REMOTE FRAME. Frente a esta situación, un frame de este tipo solo puede comenzar en el primer bit time de un INTERMISSION extendido. Hasta 2 OVERLOAD FRAMES consecutivos se permiten para demorar un DATA FRAME o un REMOTE FRAME.

- Detección de un bit "dominante" durante el periodo entre frames llamado INTERMISSION. Un frame de este tipo frente a esta condición puede comenzar en el siguiente bit time al bit "dominante" detectado.

El campo OVERLOAD FLAG consiste en 6 bits "dominantes" consecutivos (tiene la misma forma que el campo ACTIVE ERROR FLAG). El formato de este campo destruye el formato del INTERMISSION. Por lo tanto, todos los otros nodos también detectan la condición de overload sobre el bus (lo mismo que sucede con los ERROR FRAMES) y comienzan a transmitir un OVERLOAD FRAME. El campo OVERLOAD DELIMITER consiste en 8 bits "recesivos" (igual que el ERROR DELIMITER).

- **INTERFRAME SPACING**

Todos los DATA FRAMES y REMOTE FRAMES son separados entre sí por secuencia de bits con formato especial llamada INTERFRAME SPACE. A diferencia, los OVERLOAD FRAMES y los ERROR FRAMES no son precedidos por un INTERFRAME SPACE. Este campo está formado por otros dos llamados INTERMISSION y BUS IDLE (o para las estaciones que acaban de transmitir el último mensaje, un campo adicional entre estos dos llamado SUSPEND TRANSMISSION).

- INTERMISSION: consiste en 3 bits "recesivos" consecutivos. Ningún nodo puede empezar a transmitir un DATA FRAME o un REMOTE FRAME durante un INTERMISSION (solo se puede señalar una condición de overload o de error).

- SUSPEND TRANSMISSION: un nodo "error passive", después de haber transmitido un mensaje, envía 8 bits "recesivos" consecutivos después del campo INTERMISSION antes de transmitir un nuevo mensaje o reconocer el bus como libre. Si otra estación comienza a transmitir, el nodo que fue transmisor pasa a ser receptor.

- BUS IDLE: este campo tiene una longitud arbitraria. Todas las estaciones ven el bus como libre y pueden empezar a transmitir cuando lo deseen. En este campo, un bit "dominante" se interpreta como un START OF FRAME de un DATA FRAME o de un REMOTE FRAME.

2.2.4 Codificación y validación de mensajes

Los campos START OF FRAME, ARBITRATION FIELD, CONTROL FIELD, DATA FIELD y CRC SEQUENCE se codifican mediante una regla de bit stuffing. Siempre que el transmisor detecta 5 bits consecutivos con el mismo valor lógico, inserta un bit complementario en el stream de bits que se está transmitiendo. Los campos restantes de los DATA FRAMES y de los REMOTE FRAMES (CRC DELIMITER, ACK FIELD y END OF FRAME) son de formato fijo y no se aplica la técnica de bit stuffing. De igual forma, los ERROR FRAMES y los OVERLOAD FRAMES también son de formato fijo por lo cual no se aplica bit stuffing.

Una vez completado el bit stuffing, siempre que sea requerido, el stream a transmitir se codifica en NRZ (Non-Return-to-Zero).

El instante en que se decide que un mensaje es válido, es distinto para un transmisor que para un receptor. Para el primero, un mensaje es válido si no se presenta ningún error hasta el END OF FRAME. Si se detecta un mensaje corrupto, se producirá una retransmisión automática del mismo, la cual comenzará cuando el bus se encuentre libre nuevamente (para volver a competir con otro nodos en el proceso de arbitración). Para un receptor, un mensaje es válido si no se presenta ningún error hasta el anteúltimo bit de END OF FRAME.

2.2.5 Manejo de errores y confinamiento de fallas

En un bus CAN pueden darse 5 tipos diferentes de error, no excluyentes entre sí. Los errores posibles son:

- **BIT ERROR**

  Un nodo que envía un bit por el bus también monitorea el valor en el bus. Un error de este tipo, se detecta en ese momento si el valor monitoreado es distinto del valor enviado.

- **STUFF ERROR**

  Se detecta cuando aparece un sexto bit consecutivo con igual polaridad que los cinco anteriores. Sólo en los campos que se codifican mediante la técnica de bit stuffing.

- **CRC ERROR**

  El valor del campo con el código de redundancia cíclica resulta del cálculo realizado por el transmisor. El receptor calcula nuevamente ese valor, de igual forma que el transmisor, y lo compara con el que recibió. Si hay alguna diferencia, entonces se produjo algún error en la transmisión del mensaje.

- **FORM ERROR**

  Se produce cuando algún campo de formato fijo presenta uno o más bits no permitidos.

- **ACKNOWLEDGEMENT ERROR**

  Se da cuando el transmisor se da cuenta que no se hizo el acknowledgement del mensaje. En el campo ACKNOWLEDGEMENT SLOT el transmisor pone “11” o sea, dos bits recesivos. Si algún nodo no puede recibir el mensaje, no hace el ACK poniendo “01” en ese slot. Así, todos los nodos ven esto y descartan el mensaje. Esto es justamente lo que permite el broadcast y el multicast confiables.

Para minimizar el efecto de estos errores en el funcionamiento general del bus CAN, la especificación propone un mecanismo de confinamiento de fallas. Este se implementa mediante una máquina de estados. Cada nodo del bus puede encontrarse en alguno de los siguientes estados:

- **ERROR ACTIVE**

  Un nodo en este estado puede tomar parte en cualquier comunicación y transmitir un ACTIVE ERROR FLAG cuando se detecta algún error. Un flag de este tipo consiste de 6 bits dominantes consecutivos. Al ser todos bits dominantes, le gana a cualquier otra comunicación en curso. Al llegar a los nodos receptores, se detecta un STUFF ERROR y el frame de error es descartado.

- **ERROR PASSIVE**

  Un nodo en este estado también puede tomar parte en cualquier comunicación pero frente a un error, debe transmitir un PASSIVE ERROR FLAG, debiendo esperar para comenzar una nueva transmisión. Un flag de este tipo consiste de 6 bits recesivos consecutivos. Al ser todos bits recesivos, pierde con cualquier otra comunicación en curso.

  Un nodo se encuentra en este estado porque sospecha que él es el causante de los problemas que se están dando en el bus [DevNet1 99].

- **BUS OFF**

  Un nodo en este estado no tiene permitido participar en ninguna comunicación o influir en el bus ya que se encuentra desconectado lógicamente del mismo. Un nodo se encuentra en este estado porque asume que él es el causante de los problemas y por ese motivo, se desconecta [DevNet1 99].

Las transiciones entre estos estados, para un nodo en particular, se dan cuando los contadores de errores del mismo (uno para transmisión y otro para recepción) superan ciertos límites (Thr_{EP} y Thr_{BO} en la Ilustración 6). De igual forma, se establecen un conjunto de reglas que determinan cuando se debe incrementar un contador y cuando se lo puede decrementar. Entonces, las transiciones posibles serían:

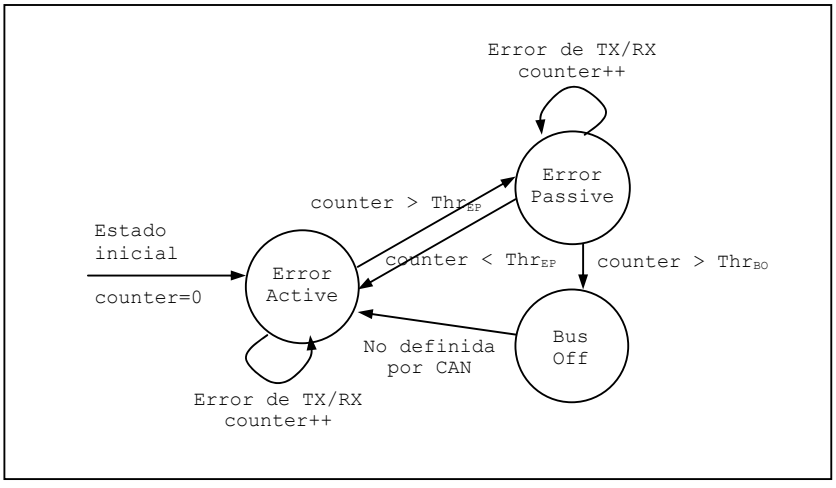


Ilustración 6 - Máquina de estados para confinamiento de fallas en CAN

2.2.6 Aspectos de implementación

Uno de los aspectos más importantes de CAN es que los nodos pueden decidir si aceptan o no un mensaje en base al valor de un campo del mismo. Esta funcionalidad se conoce con el nombre de Message Filtering. Normalmente, se presenta en los controladores CAN en dos posibles formas [Babiuch 00]:

- **BasicCAN**: el filtro de aceptación consiste en un par de registros llamados Receiver Message Identifier Mask y Receiver Message Identifier Selector. El primero contiene una máscara que se aplica sobre el ARBITRATION FIELD de los mensajes recibidos. Los bits que pasan la máscara son comparados con el valor del segundo registro. Si son iguales, el mensaje debe ser aceptado y se copia a los buffers de recepción. Es común referirse al par de registros directamente como registro mask-and-match.

  Es posible encontrar controladores CAN que incluyen más de un par de registros (filtrado doble, filtrado cuádruple, etc.).

- **FullCAN**: en este caso, no se incluye un registro Receiver Message Identifier Mask. O sea, la máscara es transparente y se tienen en cuenta todos los bits del ARBITRATION FIELD de los mensajes recibidos. Se incluyen múltiples Receiver Message Identifier Selectors, también llamados mailboxes, con lo cual es posible seleccionar mensajes en forma individual. Si un mensaje no es aceptado por ninguno de los mailboxes, se le sigue aplicando la técnica de filtrado de BasicCAN.

  Antiguamente era común tener 16 mailboxes en el controlador pero hoy en día, se trabaja con 32 o 64.

Este aspecto de implementación se incluye en el trabajo porque tendrá relevancia al momento de construir el prototipo y de definir un mecanismo de administración de los posibles valores del ARBITRATION FIELD de los frames CAN.

2.3 Capas de aplicación para el bus CAN

2.3.1 Necesidad de una capa de aplicación

Al desarrollar aplicaciones distribuidas basadas en el bus CAN, inmediatamente surge la necesidad de funcionalidad adicional, más allá de la ofrecida por la capa data-link. La funcionalidad adicional puede involucrar, por ejemplo, transmitir bloques de datos mayores a 8 bytes, transferencia con confirmación o acknowledgement, inicialización y supervisión de la red, asignación de identificadores de arbitración CAN, etc.

Esta funcionalidad soporta directamente los procesos de aplicación desarrollados por el usuario. Por este motivo, en el ambiente de buses industriales, es identificada como capa de aplicación [Etschberger 97].

Muchas de las funciones antes mencionadas se asocian con otras capas en el modelo de referencia ISO/OSI, como la capa de transporte, sesión o presentación. Las funciones de dichas capas están presentes en los buses industriales aunque no se distinguen con claridad.

La estandarización en la capa de aplicación permite cumplir con los requerimientos básicos de los buses industriales como ser interoperabilidad, intercambiabilidad de dispositivos y mantenibilidad de los sistemas. Adicionalmente a los estándares para capas de aplicación, es necesario definir modelos o tipos de dispositivo estándar con interfases comunes para asegurar la intercambiabilidad [Etschberger 97].

2.3.2 Requerimientos

Una capa de aplicación para el bus CAN en particular debería cumplir con los siguientes requerimientos [Etschberger 97]:

- **Asignación de identificadores**: una capa de aplicación para el bus CAN debería definir algún mecanismo de asignación de los identificadores a usar en el ARBITRATION FIELD de los frames CAN. Podría usarse algún esquema estático, reservando identificadores para funciones específicas, o dejando que los dispositivos los administren en forma dinámica.

- **Método de comunicación peer-to-peer**: el modelo de comunicaciones propuesto por la capa debe permitir comunicaciones entre dispositivos en una modalidad peer-to-peer, usando algún protocolo de solicitudes y respuestas.

- **Método de intercambio de datos del proceso**: por datos del proceso se entiende que son datos leídos que corresponden al proceso real, el cual se está controlando con el sistema construido sobre el bus CAN. Este requerimiento apunta a que el modelo de comunicaciones permita comunicaciones entre dispositivos en una modalidad productor / consumidor o master / slave.

- **Administración de la red**: se deben prever los medios necesarios para configurar dispositivos, recuperar los que están fallando, monitoreo y supervisión del funcionamiento del bus, etc.

- **Principios de modelado de dispositivos y definición de perfiles**: para garantizar la interoperabilidad e intercambiabilidad de dispositivos similares, es necesario que

implementen la misma interfase. O sea, que tengan los mismos parámetros de configuración, igual comportamiento, que ofrezcan servicios en común, etc.

La capa de aplicación tiene que definir alguna metodología para modelar dispositivos y definir los perfiles de los mismos.

2.3.3 Algunas implementaciones existentes

A continuación se presentan algunas implementaciones existentes de capas de aplicación para el bus CAN. La mayoría son de propósito general mientras que algunas apuntan a ciertas áreas y tipos de aplicación en particular.

- **CAN Application Layer (CAL)**: fue una de las primeras especificaciones producidas por CiA, basada en un protocolo existente desarrollado originalmente por Philips Medical Systems. Ofrece un ambiente orientado a objetos para el desarrollo de aplicaciones distribuidas de cualquier tipo, basadas en CAN. No incluye mecanismos de configuración ni una metodología estandarizada de modelado de objetos [CAL 96].

- **OSEK/VDX**: es un proyecto conjunto de la industria automotriz que busca definir un estándar y una arquitectura abierta para el desarrollo de sistemas de control distribuido en autos y vehículos de transporte. El proyecto incluye un sistema operativo de tiempo real y un protocolo de comunicación sobre el bus CAN [Etschberger 97].

- **SAE J1939**: definido por la Society of Automotive Engineers (SAE) Heavy Truck and Bus Division y aprobado por la CiA. Usado principalmente como sistema de interconexión de dispositivos electrónicos en vehículos pesados y ligeros, acoplados, maquinaria agrícola y de construcción [CiA 04].

- **CANopen**: desarrollada originalmente como un proyecto financiado por la Unión Europea. En 1995, la especificación se entregó a CiA. Utiliza un subconjunto de CAL para las comunicaciones y la administración de la red, agregando un diccionario de objetos para describir y modelar los dispositivos [CANopen 99]. Este enfoque corresponde al usado en otros buses, como Interbus-S y Profibus, para describir dispositivos [Etschberger 97].

- **DeviceNet**: especificación muy sofisticada, desarrollada originalmente por Allen-Bradley y aprobada por CiA. Está definida en términos de un modelo abstracto de objetos, que representa el comportamiento externo de un dispositivo DeviceNet. Hoy en día es mantenida por un organismo independiente llamado Open DeviceNet Vendor Association (ODVA), encargado de promover el uso de la misma [DevNet1 99] [DevNet2 99].

- **Smart Distribution Systems (SDS)**: estándar abierto desarrollado por Honeywell Micro Switch. Plantea un modelo de dispositivo jerárquico y orientado a objetos que garantiza la interoperabilidad entre dispositivos SDS. Está pensado especialmente para sensores y actuadores binarios [Etschberger 97].

- **CAN Kingdom**: desarrollado por la empresa sueca KVaser AB, aprobado por CiA y mantenido por un organismo independiente llamado CAN Kingdom International. Pensado principalmente para control de máquinas como robots industriales o sistemas hidráulicos, más que para automatización industrial. Se lo puede definir como un meta-

protocolo más que un protocolo en si, por las libertades que da al diseñador del sistema [CiA 04].

A continuación se describen con más detalle las especificaciones de CANopen y DeviceNet, ya que son implementaciones que permiten desarrollar aplicaciones de múltiples áreas y que cumplen con la totalidad de los requerimientos de las capas de aplicación antes presentados. CAL se descarta porque no incluye mecanismos de configuración ni una metodología estandarizada de modelado de objetos.

2.4 CANopen

2.4.1 Introducción

CANopen es un bus de campo desarrollado originalmente como un proyecto Esprit financiado por la Unión Europea, bajo el auspicio de Bosch. A partir de 1995, CiA se hizo cargo del mantenimiento de la especificación.

CANopen es un protocolo de aplicación para el bus CAN, desarrollado como una red estandarizada para sistemas embebidos con una gran flexibilidad de configuración. Originalmente se usaba en sistemas de control distribuido que involucrasen movimiento aunque luego se expandió a otros campos como equipamiento médico, vehículos de transporte, maquinaria de construcción, etc.

Como la especificación de CAN no define la capa de aplicación ni tampoco detalles de la capa física como por ejemplo, conectores, cableado, alimentación de los dispositivos, etc. el estándar de CANopen define su propia capa física de acuerdo a la norma ISO 11898.

2.4.2 Modelo de objetos

CANopen plantea una metodología de descripción de dispositivos basada en la definición de perfiles. Cada perfil define un dispositivo estándar el cual, especifica la funcionalidad requerida para todo dispositivo que pertenezca a dicho perfil.

Además de esta funcionalidad, un perfil de dispositivo incluye funcionalidad opcional. El fabricante puede optar por no incluirla pero si lo hace, debe implementarla de acuerdo a lo especificado por el perfil correspondiente. Así, la funcionalidad entre dispositivos de distintos fabricantes se mantiene consistente.

Por último, los perfiles incluyen un mecanismo por el cual los fabricantes pueden incluir funcionalidad propia, extendiendo el perfil en cuestión. Esto garantiza que los perfiles estándar no se vuelvan obsoletos y que puedan evolucionar.

De esta forma, esta metodología permite garantizar, en los sistemas construidos en base a CANopen, interoperabilidad, intercambiabilidad y capacidad de evolución, algunos de los requerimientos de los buses industriales enunciados [Thomesse 99].

La parte más importante del perfil de un dispositivo es la descripción del diccionario de objetos. Básicamente, un diccionario consiste en un agrupamiento de objetos, único para cada perfil de dispositivo, accesible a través del bus CAN en una forma ordenada y predefinida. Constituye la verdadera interfase entre las aplicaciones y el bus CAN, tanto desde el punto de vista de los datos como de la configuración de los dispositivos.

[CANopen 99] incluye una definición de un diccionario estándar que responde a la siguiente estructura:

Índice	Objeto	Descripción
0x0000	No usado	
0x0001 - 0x001F	Tipos de dato estáticos	Contiene definiciones de tipos de dato estándar como boolean, enteros, string, punto flotante, etc. Se incluyen como referencia, no pueden ser leídas ni escritas.
0x0020 - 0x003F	Tipos de dato complejos	Contiene definiciones de estructuras predefinidas, compuestas de tipos estáticos, comunes a todos los dispositivos.

0x0040 – 0x005F	Tipos de dato específicos del fabricante	Contiene definiciones de estructuras predefinidas, compuestas de tipos estáticos, específicas de un dispositivo en particular.
0x0060 – 0x007F	Tipos de dato estáticos específicos del perfil del dispositivo	Definiciones de tipos de dato básicos específicos para el perfil del dispositivo.
0x0080 – 0x009F	Tipos de dato complejos específicos del perfil del dispositivo	Definiciones de estructuras específicas para el perfil del dispositivo.
0x00A0 – 0x0FFF	Reservado	
0x1000 – 0x1FFF	Rango para el perfil de comunicaciones	Contiene parámetros de configuración del bus CAN. Estas entradas del diccionario son comunes a todos los dispositivos.
0x2000 – 0x5FFF	Rango para el perfil específico del fabricante	Contiene las extensiones al perfil estándar, realizadas por el fabricante.
0x6000 – 0x9FFF	Rango para perfiles de dispositivo estandarizados	Contiene todos los objetos de datos comunes a un tipo de perfil que pueden ser leídos o escritos desde la red. Algunas de las entradas son obligatorias (funcionalidad requerida) mientras que otras son opcionales (funcionalidad opcional).
0xA000 – 0xFFFF	Reservado	

Tabla 2 - Estructura de un diccionario de objetos estándar en CANopen

El direccionamiento de las entradas del diccionario se realiza, en primer lugar, mediante un índice de 16 bits. Si la entrada corresponde a una variable estática, el índice referencia directamente el valor de la misma. Por el contrario, si la entrada corresponde a una variable compleja, el índice referencia la totalidad de la estructura y es necesario un subíndice de 8 bits para acceder a un miembro en particular. Por ejemplo, supóngase la siguiente estructura, correspondiente a los parámetros de comunicación de una interfase RS232:

Índice	Subíndice	Variable	Tipo de dato
0x6092	**0x00**	Número de miembros	Unsigned8
	0x01	Baud Rate	Unsigned16
	0x02	Número de Data Bits	Unsigned8
	0x03	Número de Stop Bits	Unsigned8
	0x04	Paridad	Unsigned8

Tabla 3 - Ejemplo de variable compleja en CANopen

Para leer o escribir el número de stop bits es necesario acceder a la entrada con índice 0x6092 y luego, al subíndice 3. En el subíndice 0 se incluye siempre la cantidad de miembros de la estructura.

Un perfil de dispositivo, aparte de la descripción del diccionario de objetos, debe incluir:

- Los parámetros para la transmisión de datos del proceso controlado.

- Un mapeo de los objetos de aplicación (presentes en el diccionario de objetos) a los mensajes involucrados en la transmisión.

2.4.3 Modelo de comunicaciones

CAN ofrece un servicio sin conexión en la capa data-link, un servicio de LLC tipo 3 [Thomesse 99]. CANopen y su sistema de comunicaciones basado en CAL respetan esto y definen un modelo de comunicaciones sin conexión, basado completamente en el intercambio de mensajes.

La especificación de CANopen responde al siguiente modelo de referencia, basado también en una simplificación del modelo ISO/OSI [ISO7498]:

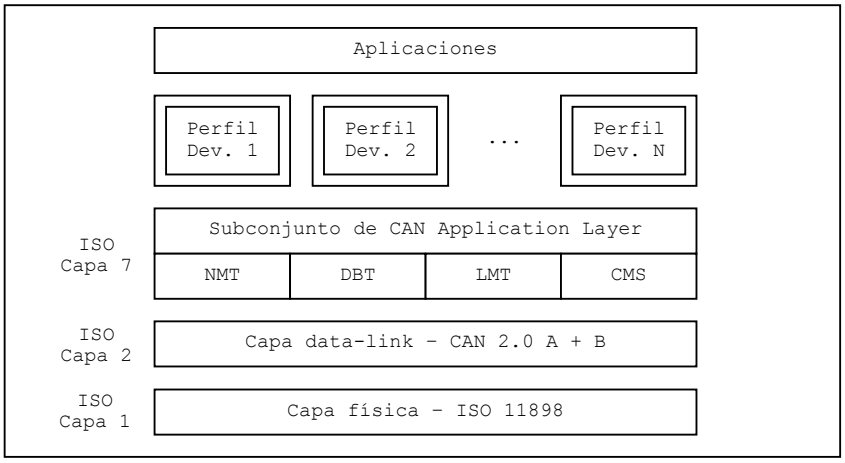


Ilustración 7 - Modelo de referencia de CANopen

Como ya se dijo, el modelo de comunicaciones es orientado a mensajes. La especificación CANopen se refiere a estos como objetos de comunicación. Se entiende por objeto de comunicación o communication object (COB) una unidad de transporte en un bus CAN o sea, un mensaje CAN de la parte A. Estos objetos se identifican mediante un COB-ID único que se corresponde con el valor del ARBITRATION FIELD del frame CAN.

Como se puede ver en la ilustración anterior, los perfiles de los dispositivos constituyen la interfase entre las aplicaciones del usuario y el bus CAN. Estos perfiles acceden al bus a través de una capa intermedia, formada por un subconjunto de CAN Application Layer (CAL). Esta capa está compuesta por los siguientes cuatro servicios [Boterenbrood 00]:

- **NMT (Network management)**: es responsable de la inicialización, configuración y manejo de errores en el bus CAN. El servicio involucra el uso de objetos administrativos de Network Management. Específicamente, mensajes de boot-up, mensajes NMT propiamente dichos y el protocolo de heartbeat.

- **LMT (Layer management)**: permite leer y configurar ciertos parámetros de CAL y de CAN. El servicio involucra el uso de objetos administrativos de Layer Management.

  En CANopen, este servicio recibe el nombre de Layer Setting Services (LSS). Permite modificar el Node-ID (objeto con índice 0x100B), parámetros de bit timing de la capa física y la dirección LSS (objeto con índice 0x1018).

- **DBT (Identifier Distributor)**: su responsabilidad es asignar en forma dinámica identificadores (COB-IDs) a los COBs usados por el servicio CMS. Como los COB-IDs deben ser únicos, este servicio debe trabajar cooperando con los DBT de otros dispositivos conectados a la red CANopen. Se trabaja en una modalidad master / slave, definiéndose un DBT master.

  CAL define un esquema plano de alocación de identificadores, basado en la definición de 8 niveles distintos de prioridad. El campo ARBITRATION FIELD de un frame CAN no tiene una estructura definida. En el esquema, quedan disponibles 1760 identificadores para toda la red:

COB-ID	Utilización	Cantidad
0	NMT Start/Stop	1
1-220	Objetos CMS Prioridad 0	220
221-440	Objetos CMS Prioridad 1	220
441-660	Objetos CMS Prioridad 2	220
661-880	Objetos CMS Prioridad 3	220
881-1100	Objetos CMS Prioridad 4	220
1101-1320	Objetos CMS Prioridad 5	220
1321-1540	Objetos CMS Prioridad 6	220
1541-1760	Objetos CMS Prioridad 7	220
1761-2015	Monitoreo dispositivos NMT	255
2016-2031	Servicios de NMT, LMT y DBT	16

Tabla 4 - Esquema de alocación de COB-IDs en CAL

Para simplificar los dispositivos y el proceso de configuración de los mismos, CANopen especifica un conjunto predefinido de conexiones. Este conjunto sigue un esquema de alocación de identificadores orientado a los dispositivos, que permite direccionar hasta 128 en una misma red. El campo ARBITRATION FIELD de un frame CAN tendría la siguiente estructura:

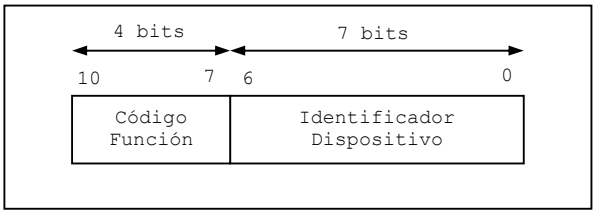


Ilustración 8 - Campo de arbitración de frame CAN bajo CANopen

Como la prioridad de un mensaje tiene que estar determinada por su funcionalidad, el código de función se coloca antes del identificador de dispositivo. De esta forma, se distinguen 16 funciones básicas para la recepción y transmisión de dos canales de datos de proceso, un canal de comunicaciones peer-to-peer, control y monitoreo de dispositivos así como notificación de emergencias y recepción de los mensajes de sincronización y time-stamp.

Se obtiene la siguiente distribución de COB-IDs:

COB-ID	Utilización	Cantidad
0	NMT Start/Stop (Código Func. 0x0)	1
128	Mensaje sincronización (Rx) (Código Func. 0x1)	1
129-255	Notificación emergencias (Código Func. 0x1)	127
256	Mensaje time-stamp (Código Func. 0x2)	1
385-511	Datos proceso PDO #1 (Tx) (Código Func. 0x3)	127
513-639	Datos proceso PDO #1 (Rx) (Código Func. 0x4)	127
641-767	Datos proceso PDO #2 (Tx) (Código Func. 0x5)	127
769-895	Datos proceso PDO #2 (Rx) (Código Func. 0x6)	127
1409-1535	Canal peer-to peer SDO (Tx) (Código Func. 0xB)	127
1537-1663	Canal peer-to peer SDO (Rx) (Código Func. 0xC)	127
1793-1919	Monitoreo dispositivos NMT (Código Func. 0xE)	127
2016-2031	Servicios de NMT, LMT y DBT (Código Func. 0xF)	16

Tabla 5 - Esquema de alocación de COB-IDs en CANopen

Por ejemplo, si el dispositivo con identificador 33 desea enviar un Service Data Object, debería solicitar al DBT un nuevo COB-ID, debiendo obtener el valor 1441 o 0x5A1.

- **CMS (CAN Message Specification)**: ofrece objetos de tipo variable, evento y dominio para diseñar y especificar cómo la funcionalidad de un dispositivo puede ser accedida a través de su interfase CAN. En fin, es un lenguaje que permite describir los objetos de comunicación y la forma en que estos deben ser transmitidos.

  CMS deriva de Manufacturing Message Specification (MMS), que es una capa de aplicación de ISO/OSI diseñada para el control y monitoreo remoto de dispositivos industriales.

Tomando como criterio la funcionalidad, se distinguen cuatro tipos de mensajes u objetos [CANopen 99]:

- **Service Data Objects (SDO)**: estos objetos o mensajes de servicio son usados para leer y escribir cualquiera de las entradas del diccionario de objetos de un dispositivo. Corresponden a mensajes CAN de baja prioridad.

- **Process Data Objects (PDO)**: estos objetos o mensajes de proceso son usados para el intercambio de datos de proceso o sea, datos de tiempo real. Por este motivo, típicamente corresponden a mensajes CAN de alta prioridad.

- **Mensajes predefinidos**: synchronization, time stamp y emergency messages. Permiten la sincronización de los dispositivos (objetos SYNC) y generar notificaciones de emergencia en forma opcional (objetos EMCY).

- **Objetos administrativos**: son mensajes administrativos que permiten la configuración de las distintas capas de la red así como la inicialización, configuración y supervisión de la misma. Se basa en los servicios NMT, LMS (LSS) y DBT de la capa CAL.

CANopen soporta los modelos de comunicación peer-to-peer, master / slave y productor / consumidor en sus variantes push y pull:

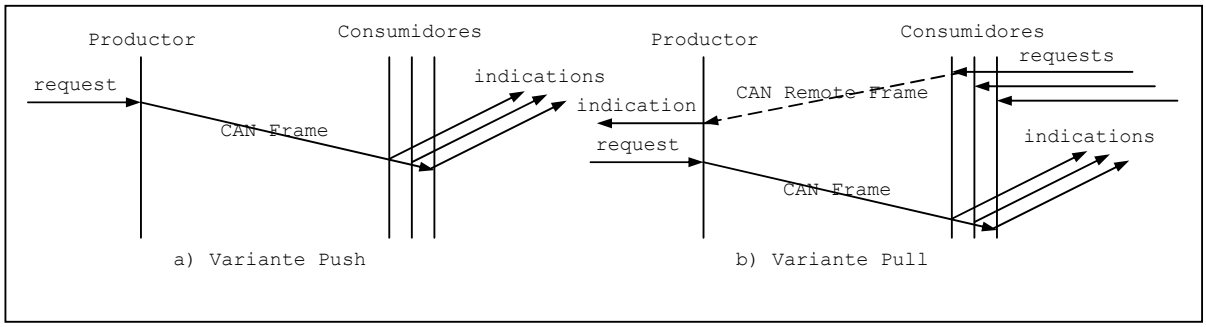


Ilustración 9 - Modelo de comunicación productor / consumidor en CANopen

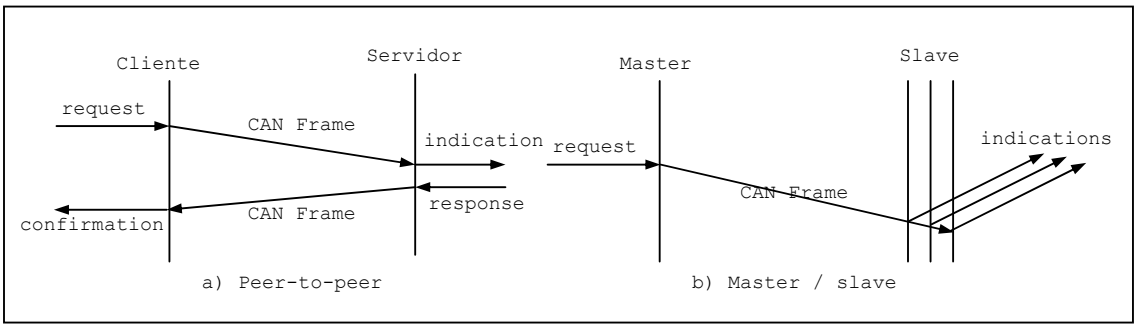


Ilustración 10 - Modelo de comunicación peer-to-peer y master / slave en CANopen

Los SDOs se utilizan para implementar comunicaciones peer-to-peer mientras que los PDOs, son utilizados para comunicaciones productor / consumidor. El modelo master / slave es implementado mediante objetos administrativos, usando el servicio NMT.

2.4.4 Capa de transporte: fragmentación y confirmación

En esta sección se describen ciertas características de CANopen relacionadas con funciones típicas de la capa de transporte en el modelo de referencia ISO/OSI. La capa de transporte provee transferencia confiable, eficiente y transparente de datos a las entidades de la capa de sesión. Provee servicios orientados y no orientados a conexión. Todos los protocolos involucrados en esta capa tienen significado end-to-end [ISO7498].

La funcionalidad de transporte ofrecida varía según el tipo de mensaje u objeto que se está enviando. Entonces:

- **Service Data Objects**: este tipo de objetos ofrecen un servicio sin conexión, con confirmación y por lo tanto, confiable.

  Se definen distintos tipos de transferencia para estos objetos. Cada tipo ofrece una funcionalidad de transporte distinta:

  - **Transferencia expédita**: usada para transmitir mensajes con longitud de datos menor o igual a 4 bytes. No se aplica fragmentación, se envía un único mensaje CAN y se recibe la confirmación o acknowledgement del servidor, implementado mediante un mecanismo stop-and-wait.

  - **Transferencia en segmentos**: usada para transmitir mensajes con longitud de datos mayor o igual a 5 bytes. Se aplica fragmentación en segmentos partiendo los datos en múltiples mensajes CAN. El cliente espera confirmación del servidor por cada segmento mediante un mecanismo stop-and-wait.

- **Process Data Objects**: este tipo de objetos permiten intercambiar datos del proceso controlado en tiempo real. Ofrecen un servicio de transferencia de datos sin conexión, sin confirmación y por lo tanto, no confiable. No se aplica un protocolo de fragmentación y reensamble de los objetos como en los SDOs.

  Los PDOs están pensados para tráfico de tiempo real de alta prioridad. Por este motivo, es conveniente evitar el overhead que produciría agregar un protocolo de fragmentación y confirmación como el que se usa en los SDOs [Boterenbrood 00].

2.4.5 Capa de sesión: protocolo de mensajes PDO en CANopen

En esta sección se describirán los objetos de proceso o PDOs que permiten implementar el modelo de comunicaciones productor / consumidor. Los datos del proceso controlado pueden ser transmitidos de un dispositivo, el productor, a uno o más dispositivos, los consumidores. Estos objetos ofrecen transferencia de datos sin conexión y sin confirmación.

Los PDOs proveen únicamente dos servicios, escribir datos (Write-PDO o T_PDO, por transmit) y leer datos (Read-PDO o R_PDO, por receive). El primero se implementa mediante un DATA FRAME mientras que para el segundo, se usa un REMOTE FRAME de CAN. En ambos casos, el DATA FIELD de 8 bytes se usa íntegramente para datos de la aplicación. CANopen no impone ningún formato sobre este campo. Vale aclarar que el segundo servicio, la lectura, es opcional y depende de la funcionalidad del dispositivo.

La cantidad de PDOs y la longitud de cada uno dependen de cada dispositivo y de la aplicación en sí. Por este motivo, esta información debe incluirse en el perfil del dispositivo correspondiente. Así, cada PDO corresponde a una entrada compleja, una estructura, en el diccionario de objetos. La estructura para el primer R_PDO tiene un índice de 0x1400 y el primer T_PDO, se ubica en la entrada 0x1800. Así, en una red CANopen se pueden definir hasta 512 R_PDOs y 512 T_PDOs.

Estas estructuras, que describen cada tipo de PDO usado por el dispositivo, contienen principalmente parámetros relacionados con la transmisión del objeto correspondiente. Reciben el nombre de PDO Communication Parameters. La definición de esta estructura se encuentra en la siguiente entrada del diccionario de objetos:

Índice	Subindice	Variable	Tipo de dato	Descripción
0x0020	**0x00**	Número de miembros	Unsigned8	Cantidad de miembros en la estructura. Siempre en el subíndice 0x00 se encuentra este dato para cualquier tipo de dato complejo.
	0x01	COB-ID	Unsigned32	Identificador del COB. Es el valor que va en el ARBITRATION FIELD del frame CAN.
	0x02	Transmission type	Unsigned8	Indica si los PDOs deben enviarse en forma sincrónica o asincrónica, cíclica o acíclica, etc.
	0x03	Inhibit time	Unsigned16	Para permitir que los objetos de comunicación (COB) con baja prioridad sean transmitidos de vez en cuando, se define un tiempo de inhibición para los PDOs. Define el tiempo que tiene que pasar entre dos invocaciones consecutivas a un servicio de un PDO.
	0x04	Reservado	Unsigned8	No se utiliza en la especificación actual de CANopen.
	0x05	Event timer	Unsigned16	Timer para disparar la transmisión del PDO correspondiente.

Tabla 6 - Definición de estructura PDO Communication Parameter en CANopen

CANopen define tres modos para disparar la transmisión de un mensaje de proceso:

- **Eventos o timer**: la transmisión de un mensaje es causada por la ocurrencia de un evento específico definido en el perfil del dispositivo. Adicionalmente, en los dispositivos que necesitan transmitir en forma periódica, un timer que expira puede causar la transmisión. Se trata de una transmisión asincrónica.

- **Solicitud remota**: la transmisión asincrónica de mensajes PDO puede comenzar al recibir una solicitud remota enviada por otro dispositivo.

- **Transmisión sincrónica**: la transmisión sincrónica de mensajes PDO es disparada por la expiración de un período de transmisión, sincronizado mediante la recepción de objetos SYNC. O sea, cada vez que llega un mensaje SYNC, se abre una ventana de transmisión sincrónica. Los PDOs sincrónicos deben ser enviados dentro de esa ventana. Se distinguen dos modos dentro de este tipo de transmisión:

    - **Modo cíclico**: son mensajes que se transmiten dentro de la ventana abierta por el objeto SYNC. No se transmiten en todas las ventanas sino con cierta periodicidad, especificada por el campo Transmission Type del Communication Parameter correspondiente. Los posibles valores para este campo son (los valores entre 1 y 240 indican el período):

Valor	Cíclico	Acíclico	Sincrónico	Asincrónico	CAN Remote Frame
0		X	X		
1-240	X		X		
241-251			Reservado		
252			X		X
253				X	X
254				X	
255				X	

Tabla 7 – Posibles valores del Transmission Type de PDO Communication Parameter

- **Modo acíclico**: son mensajes que se transmiten a partir de un evento de la aplicación. Se transmiten dentro de la ventana pero no en forma periódica.

Por último, para terminar de describir los mensajes de proceso, sería necesario describir como se mapean los objetos de aplicación a cada uno de los tipos de PDO definidos por cada dispositivo. Cada PDO puede mapear un máximo de 64 objetos de aplicación, no pudiéndose superar los 8 bytes de longitud como máximo. De esta forma, se define el contenido del campo DATA FIELD del frame CAN correspondiente. O sea, este mapeo consiste en definir el contenido de los mensajes.

Así como en la posición 0x0020 del diccionario de objetos se ubica la definición de la estructura con los parámetros de transmisión de un PDO, en la posición 0x0021 se define la estructura para el mapeo de objetos, llamada PDO Mapping Parameter, que tiene los siguientes miembros:

Índice	Subíndice	Variable	Tipo de dato	Descripción
0x0021	**0x00**	Número de mapeos	Unsigned8	Cantidad de mapeos en la estructura.
	0x01	Objeto #1	Unsigned32	Índice y subíndice del objeto mapeado en el diccionario del dispositivo.
	0x02	Objeto #2	Unsigned32	Índice y subíndice del objeto mapeado en el diccionario del dispositivo.
	...	...	...	...
	0x40	Objeto #64	Unsigned32	Índice y subíndice del objeto mapeado en el diccionario del dispositivo.

Tabla 8 - Definición de estructura PDO Mapping Parameter en CANopen

Para cada tipo de PDO, se debe definir una instancia de esta estructura con el mapeo correspondiente. Esto se hace en el diccionario de objetos a partir de la entrada 0x1600 para los R_PDO y 0x1A00 para los T_PDO, intercaladas con las entradas de los Communication Parameters que empiezan en 0x1400 y 0x1800.

2.4.6 Capa de sesión: protocolo de mensajes SDO en CANopen

Los objetos de servicio o SDOs permiten implementar los modelos de comunicación cliente / servidor o peer-to-peer, para acceder a los diccionarios de objetos de los dispositivos. Para un cierto SDO, el dispositivo cuyo diccionario está siendo accedido es el servidor mientras que el otro dispositivo, el que inicia la actividad, es el cliente.

Este tipo de objetos ofrece transferencia de datos sin conexión con confirmación o acknowledgement. Por este motivo, cada SDO involucra el intercambio de dos frames CAN, con valores distintos en el ARBITRATION FIELD.

Normalmente este tipo de objetos es usado para configuración de dispositivos y transferencia de grandes cantidades de datos no relevantes en forma directa para el control del proceso. En comparación con los PDOs, son mensajes de baja prioridad.

Un SDO es representado en CMS como un objeto de tipo Multiplexed Domain. Se definen una serie de protocolos request / response que se pueden aplicar a los SDOs para su transferencia. Estos protocolos son:

- Initiate Domain Download
- Initiate Domain Upload
- Download Domain Segment
- Upload Domain Segment
- Abort Domain Transfer

Al ser Multiplexed Domains desde el punto de vista de CMS, los SDOs pueden transferir datos de cualquier longitud [Boterenbrood 00]. Sin embargo, como se dijo anteriormente, CANopen define tres tipos de transferencia para los SDO basándose en el tamaño de los datos a transferir. Estas transferencias usan los protocolos de CMS antes enumerados. Entonces:

- **Transferencia expédita**

  Usada para transmitir mensajes con longitud de datos menor o igual a 4 bytes, para lo cual se usan los protocolos Initiate Domain Download o Initiate Domain Upload. Un SDO que se transmite del cliente al servidor, bajo este protocolo, tiene el siguiente formato:

  - **Command Specifier** (1 byte): es el byte inicial de los mensajes de los protocolos Initiate Domain Download e Initiate Domain Upload. Contiene básicamente información de control de flujo.

    Block Size Indicated (1 bit): flag que indica si se trata de una transferencia en bloques y si es así, se indica la cantidad de segmentos que componen el bloque. Tiene que estar siempre en 0 para este tipo de transferencia.

    Transfer Expedited (1 bit): indica si se trata de una transferencia expédita (sin fragmentación) o una transferencia en segmentos. Tiene que estar siempre en 1 para este tipo de transferencia.

Número de bytes (2 bits): cantidad de bytes en el campo de parámetros del cuerpo del mensaje.

Reservado (1 bit): bit reservado para uso futuro. El receptor lo ignora.

Client Command Specifier o CCS (3 bits): indicador de la operación que se desea realizar sobre la entrada del diccionario de objetos del servidor que se está accediendo (indica el protocolo que se está usando y si es un upload o un download).

- **Index** (2 bytes): índice de la entrada del diccionario de objetos del servidor que el cliente desea acceder mediante el SDO actual. Este campo y el siguiente forman el multiplexor del dominio.

- **Subindex** (1 byte): subíndice de la entrada del diccionario de objetos del servidor que el cliente desea acceder. Sólo tiene sentido si la entrada es de un tipo complejo. Si se trata de una entrada con tipo estático, debe ser 0.

- **Datos** (1 a 4 bytes): datos o parámetros del comando seleccionado por el cliente que se desean enviar al servidor (puede ser el valor de una entrada en el diccionario si se está escribiendo).

El servidor, al recibir el mensaje anterior y acceder exitosamente al diccionario de objetos contesta con una confirmación, también del protocolo Initiate Domain Download o Initiate Domain Upload. Entonces, un SDO que se transmite del servidor al cliente, bajo este protocolo, tiene el siguiente formato:

- **Command Specifier** (1 byte): es el byte inicial de los mensajes de confirmación de los protocolos Initiate Domain Download e Initiate Domain Upload. Contiene básicamente información de control de flujo.

  Reservado (5 bits): bits reservados para uso futuro. El receptor lo ignora.

  Server Command Specifier o SCS (3 bits): indicador de la operación que se desea realizar sobre la entrada del diccionario de objetos del servidor que se está accediendo (indica el protocolo que se está usando y si es un upload o un download).

- **Unused / Datos** (7 bytes): si se trata de una escritura o sea, un download, este campo se devuelve vacío al cliente y no es utilizado. En cambio, si es una lectura o un upload, contiene el valor leído del diccionario de objetos del servidor.

- **Transferencia en segmentos**

Usada para transmitir mensajes con longitud de datos mayor o igual a 5 bytes. Se aplica fragmentación en segmentos partiendo los datos en múltiples mensajes CAN. Para el primer mensaje, se usan los protocolos Initiate Domain Download o Initiate Domain Upload según corresponda.

Para los mensajes subsiguientes, se usan los protocolos Download Domain Segment o Upload Domain Segment según se quiera leer o escribir una entrada en el diccionario. Los mensajes CAN, bajo estos dos protocolos, responden al siguiente formato:

- **Command Specifier** (1 byte): es el byte inicial de los mensajes de los protocolos Initiate Domain Download e Initiate Domain Upload. Contiene básicamente información de control de flujo.

  Last Segment Indication (1 bit): flag que indica si se trata del último segmento del SDO que se está transmitiendo.

  Número de bytes (3 bits): cantidad de bytes en el campo de parámetros del cuerpo del mensaje.

  Toggle Bit (1 bit): es un bit que vale 0 o 1 en forma alternada en segmentos consecutivos. Permite hacer coincidir requerimientos con respuestas. Teniendo en cuenta que el mecanismo de intercambio sólo permite un mensaje pendiente de confirmación, un único bit es suficiente para esto. El servidor se limita a hacer un eco del valor recibido.

  Client Command Specifier o CCS (3 bits): indicador de la operación que se desea realizar sobre la entrada del diccionario de objetos del servidor que se está accediendo (indica el protocolo que se está usando y si es un upload o un download).

- **Datos** (1 a 7 bytes): datos o parámetros del comando seleccionado por el cliente que se desean enviar al servidor. Son los bytes que no caben en el mensaje CAN inicial.

El servidor, al recibir el mensaje anterior, contesta con una confirmación del segmento que responde al siguiente formato:

- **Command Specifier** (1 byte): es el byte inicial de los mensajes de confirmación de los protocolos Download Domain Segment y Upload Domain Segment. Contiene básicamente información de control de flujo.

  Reservado (4 bits): bits reservados para uso futuro. El receptor los ignora.

  Toggle Bit (1 bit): es un bit que vale 0 o 1 en forma alternada en segmentos consecutivos. El servidor se limita a hacer un eco del valor recibido en el mensaje original.

Server Command Specifier o SCS (3 bits): indicador de la operación que se desea realizar sobre la entrada del diccionario de objetos del servidor que se está accediendo (indica el protocolo que se está usando y si es un upload o un download).

- **Unused** (7 bytes): es una confirmación por lo cual, no se envían datos del dispositivo servidor al cliente.

Uno de los protocolos antes mencionados y que aún no fue explicado es el Abort Domain Transfer. Existe la posibilidad que al acceder a las entradas del diccionario de objetos del servidor, se produzca un error. Este protocolo es usado en esos casos para notificar tanto a clientes como a servidores. El formato de los mensajes de este protocolo es el siguiente:

- **Command Specifier** (1 byte): es el byte inicial de los mensajes de Abort Domain Transfer. Contiene los siguientes campos:

  Reservado (5 bits): bits reservados para uso futuro. El receptor los ignora.

  Command Specifier o CS (3 bits): indicador de la operación correspondiente al mensaje actual. Indica el protocolo que se está usando o sea, Abort Domain Transfer.

- **Index** (2 bytes): índice de la entrada del diccionario de objetos del servidor que causó el error que se está notificando. Este campo y el siguiente forman el multiplexor del dominio.

- **Subindex** (1 byte): subíndice de la entrada del diccionario de objetos del servidor que causó el error que se está notificando. Sólo tiene sentido si la entrada es de un tipo complejo. Si se trata de una entrada con tipo estático, debe ser 0.

- **Código de error** (4 bytes): código que identifica el error. El código de error es un entero de 32 bits formado por los siguientes elementos:

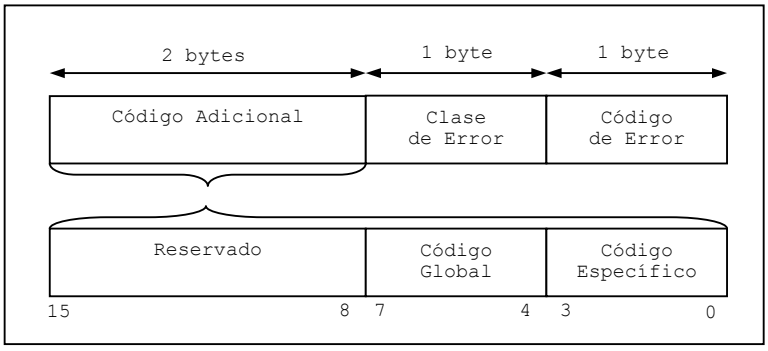


Ilustración 11 - Estructura de un código de error bajo CANopen

Los posibles valores para la clase de error, el código de error y el código adicional son [CANopen 99]:

Clase de Error	Código de Error	Descripción
0x05	**0x00**	Error de servicio – el toggle bit no está alternado en mensajes consecutivos.
0x05	**0x01**	Error de servicio – expiró el timeout.
0x06	**0x01**	Error de acceso – se intentó escribir una entrada read-only o leer una entrada write-only en el diccionario.
0x06	**0x02**	Error de acceso – el objeto no existe en el diccionario.
0x06	**0x06**	Error de acceso – el acceso al diccionario falló por un error del hardware.
0x06	**0x07**	Error de acceso – los tipos de dato no coinciden.
0x06	**0x09**	Error de acceso – el subíndice no existe.
0x08	**0x00**	Otro error – la transferencia fue abortada por el usuario.

Tabla 9 - Clase y códigos de error para SDOs en CANopen

Código Adicional	Descripción
0x00	No se conoce la causa del error.
0x10	Parámetro de servicio con valor inválido.
0x11	Parámetro de servicio con valor inválido (el subíndice no existe).
0x12	Parámetro de servicio con valor inválido (la longitud del parámetro es muy grande).
0x13	Parámetro de servicio con valor inválido (la longitud del parámetro es muy chica).
0x20	El servicio no puede ser ejecutado actualmente.
0x21	El servicio no puede ser ejecutado actualmente (por una falla en el control local).
0x22	El servicio no puede ser ejecutado actualmente (por el estado actual del dispositivo).
0x30	Se excedió el rango de valores para el parámetro.
0x31	Se excedió el rango de valores para el parámetro (el valor es demasiado grande).
0x32	Se excedió el rango de valores para el parámetro (el valor es demasiado chico).
0x36	Se excedió el rango de valores para el parámetro (el máximo es menor que el mínimo).

0x40	Incompatibilidad con otros valores.
0x41	Incompatibilidad con otros valores (no se pueden mapear los datos al PDO).
0x42	Incompatibilidad con otros valores (se superó la longitud máxima de un PDO).
0x43	Incompatibilidad con otros valores (incompatibilidad general de parámetros).
0x47	Incompatibilidad con otros valores (incompatibilidad interna de parámetros).

Tabla 10 - Códigos adicionales de error para SDOs en CANopen

Al igual que los PDOs, los SDOs requieren ser definidos en el diccionario de objetos mediante una estructura que contiene parámetros relacionados con la transmisión de los mismos. La estructura para el primer SDO para servidores tiene un índice de 0x1200 mientras que el primer SDO para clientes, se ubica en la entrada 0x1280. En total, en una red CANopen, se pueden definir hasta 128 SDOs para clientes y 128 SDOs para servidores.

Tanto para dispositivos cliente como para servidores se usa la misma estructura de parámetros, llamada Client SDO Parameter o Server SDO Parameter en uno u otro caso, aunque siempre con los mismos miembros. La definición de esta estructura se encuentra en la siguiente entrada del diccionario de objetos:

Índice	Subíndice	Variable	Tipo de dato	Descripción
0x0022	**0x00**	Número de miembros	Unsigned8	Cantidad de miembros en la estructura. Siempre en el subíndice 0x00 se encuentra este dato para cualquier tipo de dato complejo.
	0x01	COB-ID Cliente → Servidor	Unsigned32	Identificador del COB correspondiente al SDO. Es el valor que va en el ARBITRATION FIELD del frame CAN enviado por el cliente al servidor.
	0x02	COB-ID Servidor → Cliente	Unsigned32	Identificador del COB correspondiente al SDO. Es el valor que va en el ARBITRATION FIELD del frame CAN enviado por el servidor al cliente.
	0x03	Node ID	Unsigned8	Identificador del dispositivo. Es configurable mediante LMT (LSS).

Tabla 11 - Definición de estructura Client / Server SDO Parameter en CANopen

2.4.7 Capa de sesión: mensajes adicionales en CANopen

En esta sección, se presentan los mensajes u objetos predefinidos de CANopen que permiten la sincronización de los dispositivos y la generación de notificaciones de emergencia. Los objetos son:

- **Mensaje de emergencia (objeto EMCY)**

  La generación y transmisión de este tipo de mensajes está causada por la ocurrencia de un error interno en un dispositivo. Los objetos de este tipo se transmiten

del dispositivo en el que se presentó la falla a otros dispositivos como mensajes de alta prioridad. Por este motivo, sirven como interrupciones o notificaciones de alerta.

El campo DATA FIELD de los frames CAN usados para transmitir este tipo de mensajes, responde a la siguiente estructura:

- **Emergency Error Code o EEC** (2 bytes): código del error que causó la generación del mensaje de emergencia. Se trata de fallas internas de los dispositivos por lo cual, los errores se relacionan con fallas de tensión, de corriente, del software del dispositivo, del adaptador del bus CAN, etc.

- **Error Register o ER** (1 byte): cada bit de este registro indica una condición de error distinta cuando está en 1. El significado de cada bit es:

Bit	Significado
0	Error genérico.
1	Problema de corriente.
2	Problema de tensión.
3	Problema de temperatura.
4	Error de comunicaciones.
5	Específico del perfil de dispositivo.
6	Reservado.
7	Específico del fabricante del dispositivo.

Tabla 12 - Significado del Error Register en objetos EMCY de CANopen

- **Manufacturer-specific Error Field o MEF** (5 bytes): este campo puede usarse para información adicional sobre el error. Los datos incluidos y su formato son definidos por el fabricante del dispositivo.

- **Mensaje de sincronización (objeto SYNC)**

En una red CANopen, hay un dispositivo que es el productor de objetos SYNC y una serie de dispositivos consumidores de objetos SYNC. Cuando los consumidores reciben el mensaje del productor, abren su ventana de sincronismo y pueden ejecutar sus tareas sincrónicas.

Este mecanismo permite coherencia temporal y coordinación entre los dispositivos. Por ejemplo, un conjunto de sensores pueden leer las variables del proceso controlado en forma coordinada y obtener así una imagen consistente del mismo.

El COB-ID usado por este objeto de comunicación puede encontrarse en la entrada 0x1005 del diccionario. Para garantizar el acceso de estos objetos al bus, debería asignárseles un COB-ID bajo. El conjunto predefinido de conexiones de CANopen sugiere usar un valor de 128. El DATA FIELD del frame CAN de este objeto se envía vacío.

El comportamiento de estos mensajes es determinado por dos parámetros:

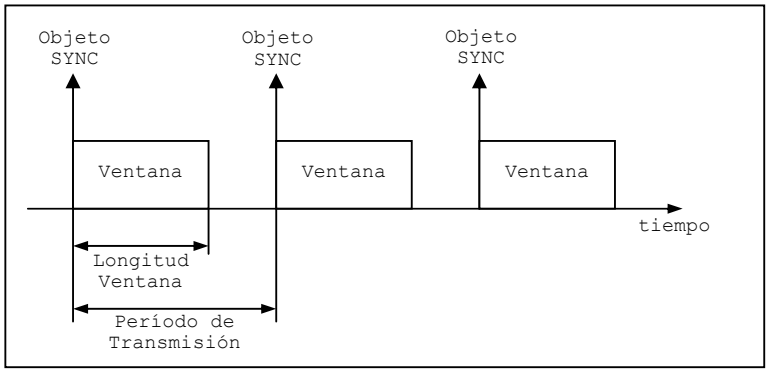


Ilustración 12 - Parámetros de un objeto SYNC en CANopen

La longitud de la ventana o Synchronous Window Length puede ubicarse en la entrada 0x1007 del diccionario. El período de transmisión o Communication Cycle Period se encuentra en la posición 0x1006.

- **Mensaje de time-stamp**

  Este tipo de objetos representan una cantidad absoluta de tiempo en milisegundos desde el 1^{ero} de Enero de 1984. La etiqueta temporal o time-stamp se implementa como una secuencia de 48 bits (6 bytes).

  Para aplicaciones grandes, que requieren un alto grado de sincronización, la resolución en milisegundos muchas veces no resulta suficiente. CANopen define un protocolo opcional que ofrece resolución en microsegundos. Se implementa mediante un contador de 32 bits que se reinicia cada 72 minutos. Puede ser accedido mapeando el objeto en 0x1013 a un PDO.

Aparte de los mensajes predefinidos, CANopen incluye una serie de mensajes para la administración y monitoreo de los dispositivos en la red. Estos están implementados en la capa CAL y reciben el nombre de servicios de Network Management (NMT). Se trabaja con un modelo de comunicaciones master / slave en el cual, un dispositivo cumple con el rol de NMT master y el resto son NMT slaves.

Un dispositivo esclavo NMT puede encontrarse en alguno de los siguientes estados:

- **Initialization**: luego de encender el dispositivo, se pasa por este estado para inmediatamente pasar a Pre-operational.

- **Pre-operational**: en este estado, el dispositivo puede ser configurado mediante envío y recepción de SDOs. Sin embargo, no tiene permitido enviar PDOs.

- **Operational**: el dispositivo ya ha sido configurado y funciona normalmente. Todos los objetos de comunicación están activos así que un dispositivo puede enviar y recibir tanto SDOs como PDOs.

El nivel de acceso de algunas entradas en el diccionario puede cambiar en este estado. Por ejemplo, la entrada que contiene el programa ejecutado por el dispositivo puede pasar de read-write en Pre-operational a read-only en Operational.

- **Stopped**: todos los objetos de comunicación dejan de estar activos. No se pueden enviar ni recibir PDOs ni SDOs, sólo mensajes de NMT.

El dispositivo NMT master envía un mensaje NMT que solicite un servicio determinado, el cual puede disparar transiciones en los esclavos y hacer que estos pasen, por ejemplo, al estado Operational. Así, podrían comenzar a funcionar, intercambiando libremente PDOs y SDOs. Un master también podría pasar a uno o más esclavos al estado Stopped.

Las posibles transiciones para un dispositivo esclavo NMT son:

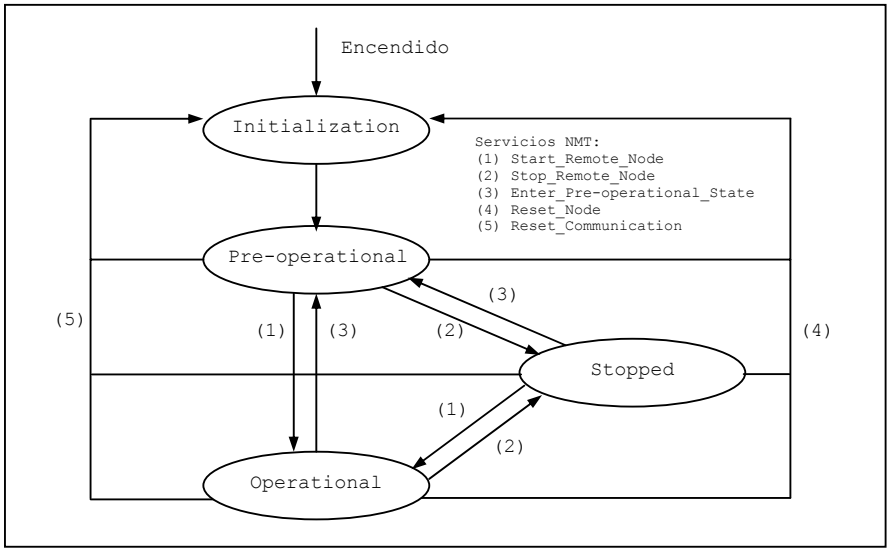


Ilustración 13 - Máquina de estados para dispositivo NMT slave en CANopen

2.4.8 Capa de presentación: CMS

Las capas más bajas de los sistemas se encargan de transportar datos del usuario entre unidades funcionales distribuidas. A ese nivel, los datos del usuario pueden ser vistos como streams no estructurados de bytes. Las capas superiores deben trabajar con estructuras más complejas. La función de la capa de presentación es proveer los medios necesarios para la representación común de los datos a ser transferidos entre agentes [ISO7498]. O sea, provee los medios para pasar, en una unidad funcional, de una estructura compleja a un stream de bytes. De igual forma, permite reconstruir y obtener la misma estructura compleja a partir de ese stream de bytes, en otra unidad funcional.

En CANopen, las funciones de la capa de presentación son realizadas en conjunto por el diccionario de objetos y CMS. El primero, constituye una sintaxis abstracta para describir los datos que serán intercambiados mediante SDOs y PDOs. El segundo, especifica un conjunto de reglas de codificación para los objetos del diccionario. Así, estos dos elementos constituyen una sintaxis de transferencia.

Los tipos de dato estáticos de CANopen, incluidos en el diccionario de objetos de todo dispositivo, son [CANopen 99]:

Índice	Tipo de Dato	Descripción
0x0001	Boolean	Puede valer TRUE o FALSE.
0x0002	Integer8	Valores desde -2^{n-1} a $2^{n-1}-1$
0x0003	Integer16	Valores desde -2^{n-1} a $2^{n-1}-1$ Codificados en little-endian
0x0004	Integer32	Valores desde -2^{n-1} a $2^{n-1}-1$ Codificados en little-endian
0x0005	Unsigned8	Valores desde 0 a $2^{n}-1$
0x0006	Unsigned16	Valores desde 0 a $2^{n}-1$ Codificados en little-endian
0x0007	Unsigned32	Valores desde 0 a $2^{n}-1$ Codificados en little-endian
0x0008	Real32	Van en notación de IEEE 754 y después en LSB
0x0009	Visible_String	Formada por un Unsigned8 más una cantidad variable de caracteres visibles de 1 byte (desde 0x20 a 0x7E)
0x000A	Octet_String	Formada por un Unsigned8 más una cantidad variable de caracteres de 1 byte (desde 0x00 a 0xFF)
0x000B	Unicode_String	Igual que los anteriores pero usando caracteres multibyte Unicode (codificados en little-endian)
0x000C	Time_Of_Day	Representa el tiempo en forma absoluta, desde el 1^{ero} de Enero de 1984 (resolución de milisegundos)
0x000D	Time_Difference	Representa una diferencia temporal en cantidad de días y milisegundos
0x000E	Bit_String	String de bits.

Tabla 13 - Tipos de dato estáticos de CANopen

En CANopen, todos los tipos de datos que ocupan más de un byte (Integer16, Integer32, Unsigned16, Unsigned32, etc.) se codifican con un orden little-endian. Esto quiere decir que primero se coloca el byte menos significativo (LSB) y luego, el más significativo (MSB). Es exactamente lo contrario del network byte order.

2.5 DeviceNet

2.5.1 Introducción

DeviceNet es un bus industrial que funciona sobre CAN. Fue desarrollado originalmente por Allen-Bradley. A partir de 1995, la especificación se abrió y desde ese momento, fue mantenida por un organismo independiente llamado Open DeviceNet Vendor Association (ODVA).

En su conjunto, DeviceNet puede ser visto como una red de bajo nivel que provee conexiones entre dispositivos industriales simples, como sensores y actuadores, y dispositivos de mayor nivel como por ejemplo, controladores, soportando modelos de comunicación peer-to-peer y master / slave.

2.5.2 Modelo de objetos: clases y objetos

La capa de aplicación de DeviceNet es orientada a objetos. Se hace uso de un modelo abstracto de objetos para describir el conjunto de servicios de comunicación disponible, el comportamiento visible en forma externa de un dispositivo DeviceNet (por ejemplo, los servicios que ofrece un sensor o un actuador) y como medio común para acceder e intercambiar información.

Este modelo se construye en base a dos entidades:

- **Clases**: una clase es un conjunto de objetos que representan el mismo tipo de componente con los mismos atributos, servicios y con un comportamiento similar. Una clase puede exponer por sí sola atributos y ofrecer servicios.

- **Objetos**: un objeto o instancia es la representación efectiva de un objeto dentro de una clase. Una instancia tiene atributos (cuyos valores determinan el estado del objeto), provee servicios (que son invocados para disparar alguna acción del objeto) e implementa un comportamiento (que indica como responder a eventos particulares).

Un dispositivo se modela como una colección abstracta de objetos que luego, se mapea al dispositivo real en una forma dependiente de la implementación usada. O sea, desde el punto de vista de DeviceNet, cada dispositivo está formado por una serie de objetos los cuales, al ser abstractos, no necesariamente se corresponden con objetos físicos de la implementación.

Para acceder a estos objetos abstractos, DeviceNet define un esquema de direccionamiento que se basa en la asignación de identificadores a cada elemento (clases y objetos). Esto resulta en un esquema de direccionamiento no plano con los siguientes niveles de indirección:

- **MAC ID**: entero único asignado a cada nodo en cada red. Es similar a una dirección de red, asignada manualmente por los usuarios. La intervención humana hace factible que se presenten errores y que haya duplicados. DeviceNet define un protocolo para identificar direcciones repetidas y evitar un mal comportamiento del bus. Los valores posibles son de 0 a 63, utilizando 6 bits.

- **Class ID**: entero único asignado a cada clase. Se pueden usar 1 o 2 bytes para guardar este valor. El tamaño es configurable. La especificación reserva una serie de identificadores de clase para su propia biblioteca de objetos.

- **Instance ID**: entero asignado a cada instancia, único dentro del espacio de direcciones MAC ID:Class ID. Se pueden usar 1 o 2 bytes para guardar este valor. El tamaño es configurable.

Entonces, para poder acceder a un objeto en particular es necesario conocer, en primer lugar, la MAC ID del dispositivo donde corre. Luego la clase o sea, su Class ID y por último, el identificador de instancia Instance ID. Al conjunto de identificadores también puede llamárselo path del objeto.

DeviceNet permite solicitar servicios o acceder a atributos de objetos y clases mediante un protocolo de mensajería explícita. Igualmente el acceso a un servicio o atributo se basa en el uso de un identificador único para el mismo. Así, se definen dos nuevos identificadores:

- **Attribute ID**: entero asignado a cada atributo de clase o instancia.

- **Service Code**: identificador de un método de instancia o clase. Los valores de códigos de servicio son de 0 a 127, insumiendo 7 bits.

Existen una serie de servicios comunes, definidos en el estándar, que deben ser implementados por los objetos de todas las clases. Para esto, se reservan los identificadores de 0x00 a 0x31 para los DeviceNet Common Services, asignados de la siguiente forma:

Código	Nombre del servicio	Descripción Funcional
0x00	**Reservado**	
0x01	**Get_Attributes_All**	Devuelve el contenido de todos los atributos de un objeto o de una clase.
0x02	**Set_Attributes_All Request**	Modifica el contenido de los atributos de un objeto o de una clase.
0x03-0x04	**Reservado**	
0x05	**Reset**	Depende del objeto o de la clase pero típicamente se va a pasar a un estado / modo inicial.
0x06	**Start**	Depende del objeto o de la clase pero típicamente se va a pasar a un estado / modo running.
0x07	**Stop**	Depende del objeto o de la clase pero típicamente se va a pasar a un estado / modo stopped/idle.
0x08	**Create**	Creación de una nueva instancia de la clase correspondiente.
0x09	**Delete**	Elimina una instancia de la clase especificada.
0x0A-0x0C	**Reservado**	
0x0D	**Apply_Attributes**	Hace que los valores de atributos cuyo uso está pendiente pasen a ser usados activamente.
0x0E	**Get_Attribute_Single**	Devuelve el contenido de un atributo específico.
0x0F	**Reservado**	
0x10	**Set_Attribute_Single**	Modifica el contenido de un atributo específico.

0x11	**Find_Next_Object_Instance**	Únicamente soportado a nivel de clase. Hace que la clase busque sus instancias y devuelva una lista de Instance IDs.
0x12-0x13	**Reservado**	
0x14	**Error Response**	Se usa para responder negativamente a un explicit request message recibido previamente.
0x15	**Restore**	Restaura los contenidos de los atributos de una clase / instancia de un almacenamiento accesible por el servicio Save (0x16).
0x16	**Save**	Copia los contenidos de los atributos de una clase / instancia a un almacenamiento accesible por el servicio Restore (0x15).
0x17	**No Operation (NOP)**	Hace que la clase / instancia destinataria simplemente conteste con un response.
0x18	**Get Member**	Devuelve información de un miembro de un atributo (que debe ser un array de un tipo básico).
0x19	**Set Member**	Guarda información de un miembro en un atributo (que debe ser un array de un tipo básico).
0x1A	**Insert Member**	Inserta un miembro en un atributo (que debe ser un array de un tipo básico).
0x1B	**Remove Member**	Elimina un miembro en un atributo (que debe ser un array de un tipo básico).
0x1C-0x31	**Reservado**	

Tabla 14 – Service Codes de los DeviceNet Common Services

Entonces, para poder solicitar un servicio a un objeto, operación similar a una invocación remota de método en otras tecnologías, es necesario conocer el Service Code del mismo una vez que fue ubicado el objeto.

Los posibles valores de cada uno de los identificadores antes definidos, principalmente Class ID y los Service Codes, se dividen en un rango abierto, llamado Open, otro específico del fabricante, llamado Vendor Specific, y por último, uno específico por clase, llamado Object Class Specific.

El primer rango, tiene significado definido por la ODVA y común a todos los participantes de dicho consorcio. O sea, todos están de acuerdo que el Service Code 0x0D corresponde al servicio Apply_Attributes. El segundo rango está reservado para el fabricante del dispositivo el cual, puede usar los identificadores dentro del mismo como considere más conveniente (normalmente para servicios específicos del dispositivo). Por último, el tercer rango es definido por cada clase y se aplica sólo para Service Codes.

2.5.3 Modelo de objetos: perfiles de dispositivo

Para mantener la interoperabilidad y promover la intercambiabilidad de dispositivos similares, tiene que existir cierta consistencia entre estos. O sea, dispositivos similares deberían tener el mismo comportamiento, producir y consumir el mismo conjunto básico de datos de I/O, tener los mismos atributos configurables, etc.

En DeviceNet, la definición de estas características comunes se llama perfil de dispositivo (o por su nombre en inglés, device profile). Existen una serie de perfiles estándar que pueden ser adoptados o extendidos para nuevos dispositivos. A continuación se incluye una lista de los perfiles predefinidos [DevNet2 99]:

Tipo de Dispositivo	Profile ID
AC Drives	0x02
Barcode Scanner	0x0D
Communications Adapter	0x0C
Contactor	0x15
Control Station	0x01
DC Drives	0x13
Encoder	0x08
General Purpose Analog I/O	0x0A
General Purpose Discrete I/O	0x07
Generic Device	0x00
Human-Machine Interface	0x18
Inductive Proximity Switch	0x05
Limit Switch	0x04
Mass Flow Controller	0x1A
Message Display	0x12
Motor Overload	0x03
Motor Starter	0x16
Photoelectric Sensor	0x06
Pneumatic Valve	0x19
Position Controller	0x10
Resolver	0x09
Servo Drives	0x14
Soft Start	0x17
Weigh Scale	0x11

Tabla 15 – Perfiles estándar de dispositivo definidos por DeviceNet

Existe un esquema de numeración para los perfiles de dispositivo. Puede haber perfiles públicos o propietarios no publicados. ODVA define los siguientes rangos:

Tipo	Rango	Cantidad de Perfiles
Público	0x0000-0x0063	100
Específico del fabricante	0x0064-0x00C7	100
Reservado	0x00C8-0x00FF	56
Público	0x0100-0x02FF	512
Específico del fabricante	0x0300-0x04FF	512
Reservado	0x0500-0xFFFF	64256

Tabla 16 – Rangos de numeración de perfiles de dispositivo

Un perfil de un dispositivo está formado por los siguientes elementos:

- **Modelo de Objetos**: cada producto DeviceNet contiene múltiples objetos que interactúan para proveer el comportamiento básico del dispositivo. Como el comportamiento de cada objeto individual es fijo, un mismo agrupamiento de objetos relacionado de igual forma, debería producir el mismo comportamiento. Ese agrupamiento es el modelo de objetos del dispositivo.

  Puede haber objetos requeridos y opcionales para los dispositivos de un perfil. Algunos objetos requeridos en cualquier perfil son los que modelan la interfase DeviceNet (como Connection, DeviceNet, Identity y Message Router). Los objetos opcionales pueden formar parte de la DeviceNet Object Library o bien, ser definidos por el diseñador del dispositivo. El modelo debe incluir:

  - Lista de todos los objetos presentes en el dispositivo.
  - Cantidad máxima de instancias para cada una de las clases.
  - Descripción de la forma en que los objetos afectan el comportamiento del dispositivo.
  - Interfases de todos los objetos.

- **Formato de Datos de I/O**: es una especificación de cómo el dispositivo se comunica en una red DeviceNet, incluyendo una especificación exacta del formato de datos a usar en las conexiones con formato libre (conexiones de I/O).

  Se debe incluir la definición de todos los I/O assemblies (agrupamientos de datos relacionados) usados por el dispositivo, el formato de los mismos y un mapeo entre los miembros de cada assembly y los atributos de los objetos. Por ejemplo, una posible especificación de formato sería:

Byte	7	6	5	4	3	2	1	0
0	Flow Value LSB							
1	Flow Value MSB							
2	Reservado					Diag #1	Diag #2	Diag #3

Ilustración 14 - Ejemplo de assembly de datos de I/O

Un posible mapeo entre este assembly y objetos sería:

Nombre del Componente	Clase		Nro. Instancia	Atributo		Tipo Dato
	Nombre	Número		Nombre	Número	
Flow Value	Low Pass Filter	0x63	1	Filter Output	0x03	UINT
Diag #1	Diagnostic	0x62	1	Wire Off Detect	0x07	BOOL
Diag #2	Diagnostic	0x62	1	Reverse Flow	0x08	BOOL
Diag #3	Diagnostic	0x62	1	Transducer Failure	0x09	BOOL

Tabla 17 – Mapeo entre assembly de datos de I/O y objeto

- **Configuración del Dispositivo**, definición de los parámetros configurables y la interfase pública de los mismos. Dispositivos similares deben comportarse igual por lo cual, deberían tener los mismos parámetros de configuración (al menos, los básicos ya que puede haber extensiones del fabricante). Se debe incluir la siguiente información:

    - Nombre del parámetro, path o dirección del atributo (Class ID, Instance ID, Attribute ID), tipo de dato, unidades y por último, valores mínimo, máximo y por defecto.

    - Efecto de cada parámetro en el comportamiento del objeto.

    - Grupos de parámetros si existen definidos algunos.

    - Interfase pública para configuración.

2.5.4 Modelo de comunicaciones

DeviceNet define sobre CAN un modelo de comunicaciones con conexión. Por el contrario, CAN ofrece un servicio sin conexión en la capa data-link, al ofrecer un servicio de LLC tipo 3 [Thomesse 99]. Las conexiones DeviceNet proveen, por lo tanto, un "camino de comunicación" entre múltiples end-points. Estos end-points son aplicaciones que necesitan compartir datos.

DeviceNet define dos tipos de conexiones en base a la funcionalidad que ofrecen y el uso que las aplicaciones van a hacer de las mismas:

- **Conexiones I/O (I/O Connections)**

    Son conexiones dedicadas de propósito específico. Se establecen entre dispositivos productores y uno o más consumidores. No se imponen restricciones sobre el formato de los mensajes cuyo significado es transparente para DeviceNet. O sea, sólo lo conocen los dispositivos que intercambian estos mensajes. Este tipo de conexiones permite implementar modelos de comunicación master / slave.

- **Conexiones de Mensajería Explícita (Explicit Messaging Connections)**

    Son conexiones genéricas de propósito múltiple. Se establecen entre dos dispositivos que intercambian mensajes cuyo formato está especificado por el estándar de DeviceNet. Los mensajes se intercambian con una modalidad request / response que permite implementar modelos de comunicación peer-to-peer. Este tipo de conexiones son las usadas para solicitar servicios de objetos o bien, para acceder a los atributos de los mismos.

Las conexiones también pueden clasificarse dependiendo del mecanismo que se usó para su creación. Existen dos posibilidades:

- **Conexiones Dinámicas**

    Se definen un conjunto de mensajes especiales que pueden ser intercambiados entre dispositivos sin necesidad de una conexión DeviceNet entre ellos. Estos mensajes son los Unconnected Explicit Messages y permiten establecer conexiones de mensajería explícita.

- **Conexiones Predefinidas**

  El estándar de DeviceNet define un conjunto de conexiones predefinidas (Predefined master / slave Connection Set) que facilitan las comunicaciones típicamente vistas en relaciones master / slave. Así, los dispositivos pueden evitar incluir el hardware necesario para manejo de conexiones dinámicas (registro mask-and-match en el adaptador CAN).

Cada conexión entre end-points, se identifica mediante un Connection ID o CID asignado en el momento del establecimiento de la misma. Si la conexión involucra un intercambio bidireccional, entonces se asignan dos identificadores.

Estos identificadores tienen que permitir mantener separados los mensajes de cada conexión. Además, para minimizar la carga de trabajo sobre el controlador de un dispositivo, su adaptador CAN debería interrumpirlo únicamente cuando llegan mensajes de conexiones existentes en el dispositivo en cuestión. Teniendo en cuenta el funcionamiento del bus CAN y los requerimientos anteriores, los CID deberían corresponderse con el valor del campo de arbitración de los frames CAN usados para intercambiar mensajes DeviceNet.

Así, el estándar DeviceNet particiona el espacio de posibles valores del campo de arbitración CAN, definiendo cuatro grupos de mensajes con características funcionales distintas y con distintos rangos de CIDs. Estos grupos son:

Bits de Identificación (10 9 8 7 6 5 4 3 2 1 0)	Rango	Utilización
0 | Group 1 MsgId | Source MAC ID	0x000-0x3FF	Grupo de mensajes 1
1 | 0 | MAC ID | Group 2 MsgId	0x400-0x5FF	Grupo de mensajes 2
1 | 1 | Group 3 MsgId | Source MAC ID	0x600-0x7BF	Grupo de mensajes 3
1 | 1 | 1 | 1 | 1 | Group 4 MsgId	0x7C0-0x7EF	Grupo de mensajes 4
1 | 1 | 1 | 1 | 1 | 1 | 1 | X | X | X | X	0x7F0-0x7FF	Identificadores CAN inválidos

Tabla 18 - Grupos de mensajes en DeviceNet

Según la tabla anterior, el campo de arbitración puede estar formado por los siguientes componentes:

- **Message ID**: identificador dentro del grupo de mensajes correspondiente. Este campo permite establecer más de una conexión dentro de un mismo end-point.

- **Source MAC ID**: MAC ID del nodo transmisor.

- **Destination MAC ID**: MAC ID del nodo receptor.

Tanto las conexiones de mensajería explícita como las de I/O pueden usar identificadores pertenecientes a los grupos 1, 2 o 3. Se debe tener en cuenta que estos grupos tienen distintas prioridades en el mecanismo de arbitración de CAN. Se debería considerar este aspecto al momento de diseñar las aplicaciones y de establecer conexiones en forma dinámica.

Este esquema de grupos de mensajes da la siguiente distribución de identificadores CAN para cada dispositivo [Etschberger 97]:

CAN Id.	Utilización	Cantidad
0-63	Grupo 1 – Message ID 0 – Disp. 0 a 63	64
64-127	Grupo 1 – Message ID 1 – Disp. 0 a 63	64
...	...	...
960-1023	Grupo 1 – Message ID 15 – Disp. 0 a 63	64
Total Grupo 1: 1024 mensajes, 16 por dispositivo		
1024-1031	Grupo 2 – Disp. 0 – Message ID 0 a 7	8
1032-1039	Grupo 2 – Disp. 1 – Message ID 0 a 7	8
...	...	...
1528-1535	Grupo 2 – Disp. 63 – Message ID 0 a 7	8
Total Grupo 2: 512 mensajes, 8 por dispositivo		
1536-1599	Grupo 3 – Message ID 0 – Disp. 0 a 63	64
1600-1663	Grupo 3 – Message ID 1 – Disp. 0 a 63	64
...	...	...
1920-1983	Grupo 3 – Message ID 7 – Disp. 0 a 63	64
Total Grupo 3: 512 mensajes, 8 por dispositivo		
1984	Grupo 4 – Message ID 0	1
...	...	...
2031	Grupo 4 – Message ID 63	1
Total Grupo 4: 64 mensajes		

Tabla 19 - Utilización de valores de arbitración en DeviceNet

La especificación de CAN no define como se puede pasar un nodo en el estado BUS OFF al estado ERROR ACTIVE. DeviceNet define esta transición mediante el uso de un conjunto de conexiones específico llamado Offline Connection Set. Los mensajes del grupo 4 están reservados para estas conexiones.

El siguiente esquema describe el funcionamiento del modelo antes descripto en un dispositivo DeviceNet. Será usado para describir el establecimiento de conexiones (dinámicas o predefinidas) y el protocolo de mensajería explícita.

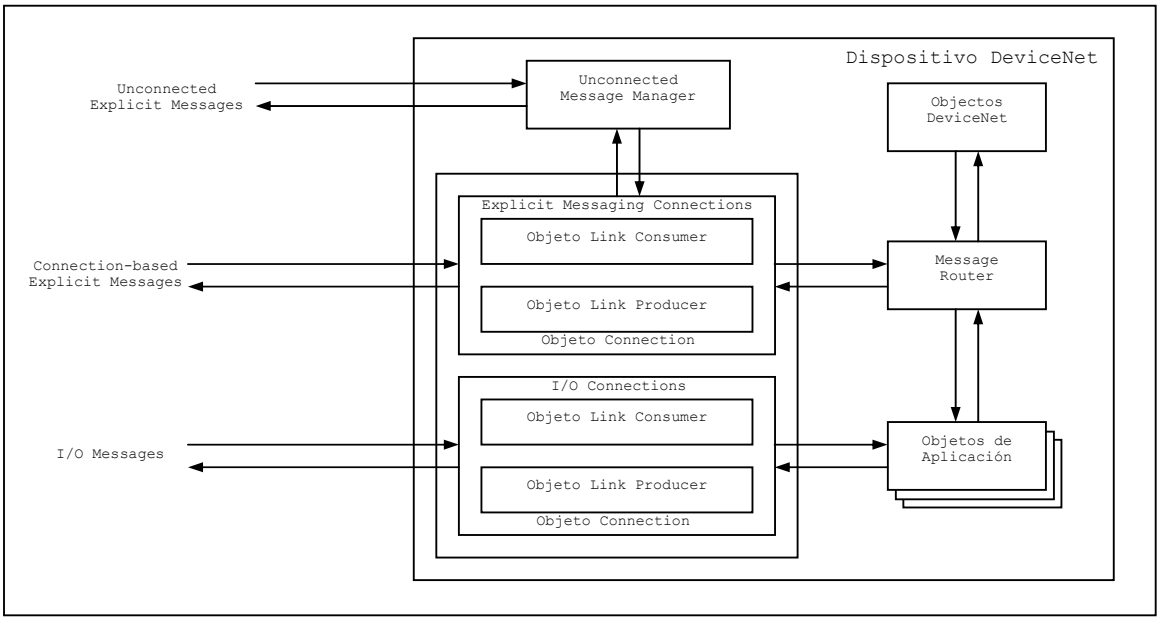


Ilustración 15 – Modelo de comunicaciones en un dispositivo DeviceNet

2.5.5 Capa de transporte: establecimiento dinámico de conexiones

DeviceNet implementa funciones típicas de la capa de transporte en el modelo de referencia ISO/OSI. En primer lugar, se describirá el mecanismo de establecimiento de conexiones dinámicas, el cual presenta diferencias dependiendo del tipo de conexión.

- **Conexiones de Mensajería Explícita**

Para solicitar el establecimiento de una de estas conexiones punto a punto se usan mensajes del grupo 3 con Message ID en 0x06, definidos en la Tabla 6. Las respuestas a estas solicitudes se transmiten con mensajes del grupo 3 con Message ID en 0x05.

En ambos casos, se trata de mensajes sin una conexión asociada o Unconnected Explicit Messages. Estos son procesados por el Unconnected Message Manager (UCMM), mostrado en la Ilustración 15. Para soportar este objeto DeviceNet, el adaptador del bus CAN del dispositivo debe permitir configurar su registro mask-and-match para que tome mensajes del grupo 3 con Message ID en 0x05 o 0x06.

Los únicos servicios que se pueden solicitar con mensajes del grupo 3 con Message ID en 0x06 son:

- Open Explicit Messaging Connection Request
- Close Connection Request

De igual forma, los únicos mensajes que se pueden transmitir con un valor de arbitración correspondiente al grupo 3 con Message ID en 0x05 son:

- Open Explicit Messaging Connection Response

- Close Connection Response
- Error Response
- Device Heartbeat Message
- Device Shutdown Message

- **Conexiones I/O**

Estas conexiones, que pueden ser punto a punto o multicast, se establecen en forma dinámica interactuando con el objeto Connection a través de una conexión de mensajería explícita previamente establecida.

Los pasos a seguir son, en primer lugar, abrir una conexión de mensajería explícita para luego enviar una solicitud de servicio Create Request a la clase Connection del dispositivo. Esto crea una nueva instancia de la clase I/O Connection la cual debe ser configurada. Finalmente, se envía una solicitud de aplicación de la configuración y se entrega la nueva instancia a la aplicación que va a consumir o producir datos (modelo master / slave o productor / consumidor).

- **Offline Connection Set**

Este tipo de conexión es usado por herramientas cliente para recuperar dispositivos en el estado BUS OFF y pasarlos a ERROR ACTIVE. Esta es la única transición que CAN no define en su maquina de estados para confinamiento de fallas. Usando mensajes del grupo 4 (ver Tabla 6), una herramienta de este tipo puede:

- Identificar visualmente el dispositivo (flashing LED).
- Enviar mensajes de recuperación al dispositivo fallado.
- Recuperar el dispositivo fallado sin tener que desconectarlo.

Si hay más de una herramienta cliente que desea recuperar dispositivos fallados, para evitar problemas de consistencia, DeviceNet define un protocolo de ownership del Offline Connection Set. De esta forma, se tiene seguridad que en un momento dado, sólo existe un nodo usando alguna de las conexiones del conjunto de conexiones Offline.

El protocolo de obtención del ownership es el siguiente. La herramienta produce un Offline Ownership Request Message usando una conexión del grupo 4 con Message ID en 0x2F. Luego, espera una respuesta de tipo Offline Ownership Response Message por la conexión del grupo 4 con Message ID en 0x2E durante 1 segundo. Si no se recibe respuesta, se genera otro mensaje y se vuelve a esperar por la respuesta.

Transcurrido el segundo período de espera, si no llegó ninguna respuesta, la herramienta gana el ownership del Offline Connection Set. Una vez que la gana, la herramienta debe responder a todos los mensajes Offline Ownership Request Message para que el protocolo antes descripto funcione correctamente.

Cuando un dispositivo pasa al estado BUS OFF, sólo tiene que filtrar mensajes del grupo 4 con Message ID en 0x2D. De igual forma, sólo va a producir mensajes del grupo 4 con Message ID en 0x2C. Una herramienta cliente, que ya haya ganado el ownership del Offline Connection Set, transmite mensajes por la conexión del grupo 4

con Message ID en 0x2D y recibe respuestas por la conexión del grupo 4 con Message ID en 0x2C.

2.5.6 Capa de transporte: utilización de conexiones predefinidas

El modelo general de comunicaciones de DeviceNet define dos tipos de conexiones: las de mensajería explícita para comunicaciones peer-to-peer, y las de I/O para comunicaciones maestro / esclavo o bien, productor / consumidor.

El modelo utiliza conexiones de mensajería explícita para crear y configurar manualmente objetos Connection dentro de cada end-point y así, obtener conexiones de I/O. Este enfoque impone mayores requerimientos de procesamiento sobre los dispositivos DeviceNet. Por ejemplo, supóngase un dispositivo DeviceNet que no trabaja con comunicaciones peer-to-peer y desea manejarse únicamente con un modelo maestro / esclavo. Bajo el modelo general, está obligado a soportar el UCMM y conexiones de mensajería explícita aunque estas sólo van a ser usadas para crear conexiones de I/O.

Para evitar estos requerimientos innecesarios y bajar los costos de los dispositivos DeviceNet, se define un conjunto de conexiones de I/O predefinidas que facilitan las comunicaciones típicas de relaciones master / slave. De esta forma, es posible construir dispositivos sin UCMM.

En primer lugar, es necesario definir qué criterio usa DeviceNet para decir que un dispositivo es master o slave. El master es el dispositivo que junta y distribuye datos de I/O al controlador del proceso. El slave es el dispositivo que envía datos de I/O a su master. Los masters inician las actividades mediante mensajes de solicitud y los slaves contestan con sus datos de I/O. Esto indica que los masters se comportan como clientes mientras que los slaves, como servidores.

Estos clientes y servidores pueden comunicarse mediante tres tipos de conexión, predefinidos dentro del conjunto de conexiones master / slave:

- **I/O Bit-Strobe Connection**

  Esta conexión es usada para mover rápidamente pequeñas cantidades de datos de I/O entre un master y los bit-strobed slaves. El master, en base a su lista de slaves, construye una palabra de 64 bits (8 bytes) en la cual, pone un bit para cada slave (de ahí, bit strobed). Así, el bit 0 corresponde a la MAC ID 0 y el 63, a la MAC ID 63. Esta solicitud se envía usando el Grupo 2 con Message ID en 0x00.

  En la respuesta, los slaves mandan directamente 8 bytes de datos de I/O dependiendo de cómo hayan interpretado su bit en la solicitud. Para esto, usan mensajes del grupo 1 con Message ID en 0x0E. Se trabaja con un único objeto Connection del lado del slave y múltiples del lado del master.

- **I/O Poll Connection**

  La conexión I/O Poll, a diferencia de la anterior, sirve para mover cantidades mayores de datos (sólo punto a punto, en modo unicast). El master tiene un objeto Connection para cada slave.

  El master envía la solicitud de poll a un slave con un mensaje del grupo 2 con Message ID en 0x05. El slave correspondiente contesta con un mensaje del grupo 1 con Message ID en 0x0F.

- **I/O Change Of State/Cyclic Connection**

  Es similar al tipo de conexión anterior siendo la única diferencia el uso de acknowledgement en ambos sentidos o sea, de master a slave y viceversa. La transmisión se dispara cuando se produce un cambio de estado en la aplicación o cuando expira un timeout. Este comportamiento es configurable al reservar la conexión.

  El master envía una solicitud usando una conexión polled o sea, envía mensajes del grupo 2 con Message ID en 0x05. El slave envía el acknowledgement correspondiente usando el grupo 1 con Message ID en 0x0F.

  Luego, el slave envía al master el dato solicitado usando mensajes del grupo 1 con Message ID en 0x0D y el master contesta con el acknowledgement del grupo 2 y Message ID en 0x02.

Como se puede ver, los servidores o slaves siempre reciben mensajes del grupo 2 por lo cual, es posible construirlos para que sólo soporten mensajes de dicho grupo. Esto permite usar adaptadores CAN más sencillos sin necesidad de registros mask-and-match para filtrar mensajes. Esta es una posibilidad contemplada por el estándar de DeviceNet.

Un dispositivo con soporte para UCMM y mensajería explícita, puede usar conexiones de este conjunto predefinido solicitando el servicio Allocate_Master/Slave_Connection_Set del objeto DeviceNet. Si en cambio, se trabaja con dispositivos que sólo soportan el grupo 2 y no incluyen el UCMM, se define un mecanismo especial y se reserva el Message ID 0x06 del grupo 2 para solicitar el uso del conjunto predefinido.

2.5.7 Capa de sesión: protocolo de mensajes DeviceNet

El propósito de la capa de sesión es permitir la organización del diálogo, su sincronización y la administración del intercambio de datos [ISO7498]. En esta sección se describen los protocolos usados por los dispositivos DeviceNet para organizar su diálogo y así, comunicarse entre ellos. O sea, es una descripción del contenido que los dispositivos DeviceNet colocan en los DATA FIELDS de los frames CAN que intercambian.

En primer lugar, se describirán los protocolos usados por las conexiones de mensajería explícita y de I/O, que reciben el nombre de Control and Information Protocol (CIP) [VanGompel 01]:

- **Conexiones de Mensajería Explícita**

  Un mensaje explícito o explicit message está formado por un encabezado de 1 byte y un cuerpo de 7 bytes. Si el cuerpo del mensaje no entra en estos 7 bytes, se aplica el protocolo de fragmentación, intercalando 1 byte entre el encabezado y el cuerpo (ahora de 6 bytes).

  Los campos del encabezado del mensaje son:

  - **Frag** (Fragment Bit, 1 bit): indica si el frame CAN contiene un fragmento (1, Fragmented) o el mensaje completo (0, Non-fragmented).

  - **XID** (Transaction ID, 1 bit): usado por las aplicaciones para hacer corresponder una solicitud con su correspondiente respuesta. Teniendo en cuenta que el mecanismo de control de flujo sólo permite un mensaje pendiente de confirmación, un único bit es suficiente para esto. El server se limita a hacer un echo del valor recibido.

- **MAC ID** (MAC ID, 6 bits): MAC ID o dirección del dispositivo de origen o de destino. Si en el campo de arbitración, al generar el Connection ID o CID, se utilizó la dirección de origen entonces, este campo contiene la dirección de destino y viceversa.

Los campos del cuerpo del mensaje son:

- **Service Field** (1 byte): identificación del servicio que se está solicitando o respondiendo. Este campo, a su vez, se divide en:

  R/R (1 bit): determina si se trata de un request (bit en 0) o de un response (bit en 1).

  Service Code (7 bits): identificador del servicio que se está solicitando o respondiendo. Podrían ser, por ejemplo, los códigos de los DeviceNet Common Services definidos en la tabla 2.

- **Service Specific Arguments** (1 o más bytes): parámetros específicos del servicio que se está solicitando. Por ejemplo, supóngase que se está llamando a un servicio particular de una instancia, este campo contendría los siguientes elementos:

  Class ID (1/2 bytes): define la clase a la cual se está enviando la solicitud. Se usan 1 o 2 bytes dependiendo de los parámetros usados al crear la conexión de mensajería explícita (Requested Message Body Format).

  Instance ID (1/2 bytes): define la instancia de la clase identificada por Class ID a la cual se dirige la solicitud. Se usan 1 o 2 bytes dependiendo de los parámetros usados al crear la conexión de mensajería explícita (Requested Message Body Format). El valor 0 está reservado para indicar que la solicitud va dirigida a una clase y no a una instancia.

  Service Data (0 o más bytes): datos específicos de la solicitud.

En el caso de una respuesta exitosa a la solicitud, sólo se recibiría en este campo datos específicos de la solicitud. Podrían ser, por ejemplo, valores de atributos o valores de retorno del servicio solicitado.

- **Conexiones de I/O**

DeviceNet no define ninguna estructura ni protocolo dentro del DATA FIELD de un mensaje de I/O excepto el protocolo de fragmentación para mensajes de I/O mayores a 8 bytes.

Las condiciones de aplicación del protocolo de fragmentación son distintas para este tipo de conexión que para el anterior.

En secciones anteriores, se describió parcialmente el mecanismo de establecimiento de conexiones de mensajería explícita las cuales son usadas, posteriormente, para crear conexiones de I/O. Se enumeraron los mensajes procesados por el UCMM los cuales, en cierta forma, son los servicios ofrecidos por este objeto DeviceNet.

Estos mensajes tienen el formato que se acaba de presentar, usado en las conexiones de mensajería explícita. A continuación se dan detalles, para cada tipo de unconnected explicit message, de los contenidos del campo Service Specific Arguments:

- **Open Explicit Messaging Connection Request**: solicita el establecimiento de una conexión lógica entre dos módulos a través de la cual se transmitirán mensajes explícitos. Este mensaje se envía con un valor de arbitración del grupo 3 con Message ID en 0x06, usando un Service Code igual a 0x4B. Los argumentos específicos son:

  - **Reserved Bits** (4 bits): reservados para uso futuro. El receptor los ignora y el transmisor debe poner todos en 0.

  - **Requested Message Body Format** (4 bits): el formato en que se deberán transmitir los mensajes subsecuentes por esta conexión. Los posibles valores son:

Body Format	Descripción
0x0	DeviceNet(8/8), 8 bits de Class ID, 8 bits de Instance ID
0x1	DeviceNet(8/16), 8 bits de Class IS, 16 bits de Instance ID
0x2	DeviceNet(16/16), 16 bits de Class ID, 16 bits de Instance ID
0x3	DeviceNet(16/8) , 16 bits de Class ID, 8 bits de Instance ID
0x4-0xF	Reservado por DeviceNet

Tabla 20 – Request Message Body Format en Open Connection Request

El receptor del mensaje (a partir de ahora, el servidor) es el que toma la decisión final sobre el formato que se va a usar, dependiendo si puede cumplir o no con lo que pide el emisor (a partir de ahora, el cliente). En la respuesta se manda el valor efectivo para la conexión.

  - **Group Select** (4 bits): grupo de mensajes a usar. Los valores son 0 para el grupo 1, 1 para el 2 y 3 para el 3. Si el servidor no soporta o temporalmente no puede usar algún grupo, debe responder con un mensaje de error. En este caso, no hay negociación como en el anterior.

  - **Source Message ID** (4 bits): depende del valor de Group Select. Si este último es 0 o 3, especifica el Message ID asignado por el cliente de su pool de IDs del grupo 1 o grupo 3. Si por el contrario, Group Select es 1 este campo es ignorado y se llena con 0.

- **Open Explicit Messaging Connection Response**: este servicio se utiliza para responder una solicitud exitosa de establecimiento de conexión. Este mensaje se envía con un valor de arbitración del grupo 3 con Message ID en 0x05, usando un Service Code igual a 0x4B y con el campo R/R en 1. Los argumentos específicos son:

  - **Reserved Bits** (4 bits): reservados para uso futuro. El receptor los ignora y el transmisor debe poner todos en 0.

  - **Actual Message Body Format** (4 bits): el formato en que se deberán transmitir los mensajes subsecuentes por esta conexión. Puede ser un valor definido por el servidor o bien, el que solicitó el cliente si éste puede cumplirlo.

  - **Destination Message ID** (4 bits): depende del valor de Group Select. Si este es 1, especifica el Message ID asignado por el servidor de su pool de IDs del grupo 2. Si por el contrario, Group Select es 0 o 3 este campo es ignorado y se llena con 0.

  - **Source Message ID** (4 bits): el Message ID que el servidor asignó de su pool de IDs para el grupo 1, 2 o 3. El servidor usa este valor junto con su MAC ID como identificador de la conexión o CID.

  - **Connection Instance ID** (1/2 bytes): identificador de la instancia de la clase Connection creada por el UCMM. Este valor puede ser usado posteriormente para cerrar la conexión o bien, para abrir una conexión I/O invocando servicios del objeto Connection.

- **Close Connection Request**: servicio usado para cerrar una conexión I/O o mensajería implícita, es indistinto. Al recibir este mensaje, el UCMM llama al servicio Delete de la clase Connection. Este mensaje se envía con un valor de arbitración del grupo 3 con Message ID en 0x06, usando un Service Code igual a 0x4C. Los argumentos específicos son:

  - **Connection Instance ID** (2 bytes): identificador de la instancia de la clase Connection a ser eliminada. Como estos mensajes se transmiten como unconnected messages, el receptor puede no conocer el message body format así que se usan siempre 2 bytes.

- **Close Connection Response**: este servicio es usado para responder en forma exitosa a un close connection request. Este mensaje se envía con un valor de arbitración del grupo 3 con Message ID en 0x05, usando un Service Code igual a 0x4C y con el campo R/R en 1. No tiene argumentos específicos.

- **Error Response**: este tipo de mensajes se envía cuando se produce un error tratando de dar servicio a una solicitud recibida en un explicit message. Este mensaje se envía con un valor de arbitración del grupo 3 con Message ID en 0x05, usando un Service Code igual a 0x14. Los argumentos específicos del mismo son:

  - **General Error Code** (1 byte): identificador general del error.

- **Additional Error Code** (1 byte): valor específico del objeto o servicio, que aporta información adicional sobre el error. Si no hay información adicional, se coloca en este campo 0xFF.

Estos son los errores que se pueden presentar frente a una solicitud Open Explicit Messaging Connection Request o Close Connection Request:

Condición de Error	Nombre del Error	Código de Error General	Código de Error Adicional
El Service Code no es Open ni Close.	Service not supported	0x08	0xFF
Error de recursos al seleccionar el grupo de mensajes.	Resource unavailable	0x02	0x01
El grupo de mensajes está fuera del rango válido.	Invalid Parameter	0x20	0x01
El servidor no dispone de más conexiones.	Resource unavailable	0x02	0x02
El servidor no dispone de más Message IDs para nuevas conexiones.	Resource unavailable	0x02	0x03
El Message ID de origen, enviado por el cliente, no es válido.	Invalid Parameter	0x20	0x02
El Message ID de origen, enviado por el cliente, está repetido.	Resource unavailable	0x02	0x04
El Instance ID de la conexión no es válido.	Object does not exists	0x16	0xFF

Tabla 21 – Posibles códigos de error en mensajes Error Response de DeviceNet

Por último, es necesario describir los servicios de fragmentación y de control de flujo end-to-end provistos por las conexiones DeviceNet. Si bien estos servicios normalmente se asocian con la capa de transporte, en el presente trabajo se incluyen junto con la capa de sesión para facilitar el orden de exposición de los temas.

La fragmentación en DeviceNet está planteada como un servicio opcional provisto por el objeto Connection. La lógica que decide si hay que aplicar fragmentación es distinta según el tipo de conexión:

- **Conexiones de Mensajería Explícita**: se examina la longitud de cada mensaje y si supera los 8 bytes, se aplica fragmentación acknowledged (con control de flujo).

- **Conexiones de I/O**: se examina el atributo produced_connection_size del objeto Connection y se decide en base a su valor, configurado al momento de crear el objeto. Si es mayor que 8, se aplica fragmentación unacknowledged (sin control de flujo).

El protocolo de fragmentación ocupa un byte entre el encabezado y el cuerpo del mensaje, si es necesario aplicarlo. Los campos definidos por este protocolo dentro de ese byte son:

- **Fragment Type** (2 bits): indica si se trata del primer, último o un fragmento intermedio. Los posibles valores son 0 para el primer fragmento, 1 para fragmentos intermedios, 2 para el último fragmento y finalmente, 3 para el acknowledge de un fragmento.

- **Fragment Count** (6 bits): índice o número de secuencia del fragmento que se está enviando. Se incrementa en 1 para cada nuevo fragmento de forma de poder efectuar un control de secuencia y determinar si se perdió algún fragmento.

En el caso de una conexión I/O, cuando se invoca el servicio Send_Message se examina el valor del atributo produced_connection_size. Si este valor es mayor a 8, indistintamente del tamaño del dato a transmitir, se coloca en el mensaje el byte del protocolo de fragmentación. Si el tamaño del dato a transmitir es mayor a produced_connection_size se produce un error.

Si el receptor detecta que falta un fragmento, al ver valores de Fragment Count no consecutivos, descarta todos los mensajes subsiguientes. La aplicación no se entera de que llegó un mensaje y la conexión se reinicia para esperar nuevamente por un fragmento inicial.

Por otro lado, en el caso de una conexión de mensajería explícita, cada vez que se transmite un fragmento, se espera el acknowledge del nodo receptor. La disciplina de línea aplicada corresponde a un algoritmo stop-and-wait. Los campos de un mensaje de acknowledgement son:

- **Fragment Type** (2 bits): indica que se trata de un fragment acknowledge. Tiene siempre valor 3.

- **Fragment Count** (6 bits): es el valor del índice del último frame recibido.

- **Ack Status** (1 byte): indica si se produjo (1) o no (0) un error en el receptor del mensaje. Los valores de 0x02 a 0xFF están reservados.

El nodo transmisor envía un fragmento y espera el acknowledgement del receptor. Se inicia un timer para controlar la expiración del tiempo de espera. Si se da un timeout, el objeto Connection reintenta. Si nuevamente se da un timeout, se le informa a la aplicación del error. Al recibir un acknowledge, se controlan los campos Fragment Count y Ack Status, aceptándose el fragmento si así corresponde. Para las capas superiores, la utilización o no de fragmentación es completamente transparente.

2.5.8 Capa de presentación: Compact Encoding

En DeviceNet, las funciones de la capa de presentación son realizadas por Compact Encoding. Se trata de una sintaxis de transferencia que permite pasar de una estructura compleja a un stream de bytes y viceversa.

En primer lugar, es necesario definir los tipos de dato de estas estructuras complejas a convertir. Para mantener la independencia entre capas, estos tipos se definen mediante una sintaxis abstracta. En el caso de Compact Encoding, se usa la Abstract Syntax Notation One (ASN.1) especificada en la normas ISO 8824.1-4 / ITU-T X.680-683 [ISO8824 02].

En segundo lugar, se debe aplicar un conjunto de reglas de codificación que permitan pasar de los datos abstractos a una secuencia de bytes que pueda ser transmitida. A diferencia de lo que sucede con la sintaxis abstracta, en el caso de las reglas de codificación, no se

adopta la solución OSI dada por las Basic Encoding Rules (BER) definidas en los estándares ISO 8825.1-4 / ITU-T X.690-693 [ISO8825 02].

DeviceNet define su propio conjunto de reglas de codificación, respetando las siguientes restricciones:

- Las entidades involucradas en una conexión (los end-points de la misma) tienen conocimiento previo del tipo de la variables por lo cual, no es necesario transmitirlo. Así se evita enviar el tipo de cada variable en cada mensaje.

- El tipo de la variable es de longitud fija y no tiene campos condicionales u opcionales.

- La codificación de una variable es representada con un número constante de bytes, derivados del tipo de la misma. Así se evita transmitir la longitud de cada variable.

- Si la variable es entera y ocupa más de un byte, se usa siempre ordenamiento Least Significant Byte (LSB) o little endian. Esto es exactamente lo contrario del ordenamiento Most Significant Byte (MSB) o big endian, usado en ISO/OSI.

A partir de estas restricciones, se obtiene las siguientes reglas de codificación para tipos básicos:

Tipo de Dato	Bytes	Regla de Codificación
BOOL	1	TRUE → 0x01 FALSE → 0x01
Codificación de SignedInteger (enteros con signo)		
SINT	1	OLSB
INT	2	0LSB 1LSB
DINT	4	0LSB 1LSB 2LSB 3LSB
LINT	8	0LSB 1LSB 2LSB 3LSB ...
Codificación de UnsignedInteger (enteros sin signo)		
USINT	1	OLSB
UINT	2	0LSB 1LSB
UDINT	4	0LSB 1LSB 2LSB 3LSB
ULINT	8	0LSB 1LSB 2LSB 3LSB ...
ENGUNIT	2	OLSB 1LSB
Codificación de Strings		
STRING	2 de longitud + 1 por caracter	
STRING2	2 de longitud + 2 por caracter	Los caracteres multibyte (más de un byte) van siempre en LSB.

STRINGN	2 de tamaño de carácter + 2 de longitud + N por caracter	Los caracteres multibyte (más de un byte) van siempre en LSB.
SHORT_STRING	1 de longitud + 1 por caracter	
Codificación de FixedLengthReal (punto flotante)		
REAL	4	Van en notación de IEEE 754 y después en LSB.
LREAL	8	Van en notación de IEEE 754 y después en LSB.
Codificación de Date y Time		
TIME	4	Se codifica como FixedLengthReal.
DATE	2	
TIME_OF_DAY	4	Se codifica como FixedLengthReal.
DATE_AND_TIME	6	Se codifica como FixedLengthReal (4 para la hora y 2 para la fecha)
FTIME	4	Se codifica como FixedLengthReal
LTIME	8	Se codifica como FixedLengthReal
Codificación de FixedLengthBitStrings (cadenas de bits)		
BYTE	1	Orden de bits: 7-0
WORD	2	Orden de bits: 7-0, 15-8
DWORD	4	Orden de bits: 7-0, 15-8, 23-16, 31-24
LWORD	8	Orden de bits: 7-0, 15-8, 23-16, 31-24, ...

Tabla 22 - Reglas de codificación de Compact Encoding

Los arreglos (arrays) se codifican como una simple concatenación repetitiva de los elementos que lo componen (usando las reglas anteriores para cada uno de estos). Un array de 1 dimensión, definido en ASN.1, sería:

```
ARRAY ::= SEQUENCE OF {
    array_dimension_low_bound,
    array_dimension_high_bound,
    DeviceNetData
}
```

Código 1 - Ejemplo de array DeviceNet en ASN.1

Para un array de 2 dimensiones:

```
ARRAY ::= SEQUENCE OF {
    array_dimension_low_bound,
    array_dimension_high_bound,
```

```
    SEQUENCE OF {
        array_dimension_low_bound,
        array_dimension_high_bound,
        DeviceNetData
    }
}
```

Código 2 - Ejemplo de array multidimensional DeviceNet en ASN.1

Las estructuras (structs) se codifican de forma similar. Para cada uno de los campos, se aplican las reglas anteriores y luego se concatenan los bytes resultantes.

3. Middleware

Resumen del capítulo

En este capítulo se describen las características del middleware y su función en la construcción de aplicaciones distribuidas. Se presentan las distintas variedades existentes, las ventajas que cada una trae al desarrollador de aplicaciones distribuidas y se analizan cuales pueden ser adaptadas para ser usadas como capa de aplicación del bus CAN en aplicaciones de control distribuido.

3.1 Características

El middleware se define como un software distribuido constituido por un conjunto de recursos y servicios que permiten a múltiples procesos, corriendo en una o más máquinas, interactuar a través de una red [Bray 97]. Es una colección no estructurada de recursos y servicios, ubicados entre la capa de transporte y la capa de aplicación, que pueden ser utilizados individualmente o conjuntamente [Aiken 00] por los procesos de la aplicación. Provee una interfase abstracta que ofrece al programador de aplicaciones una visión uniforme de elementos heterogéneos de bajo nivel, como sistemas operativos y redes subyacentes [Sun 04].

Un esquema de middleware típico es:

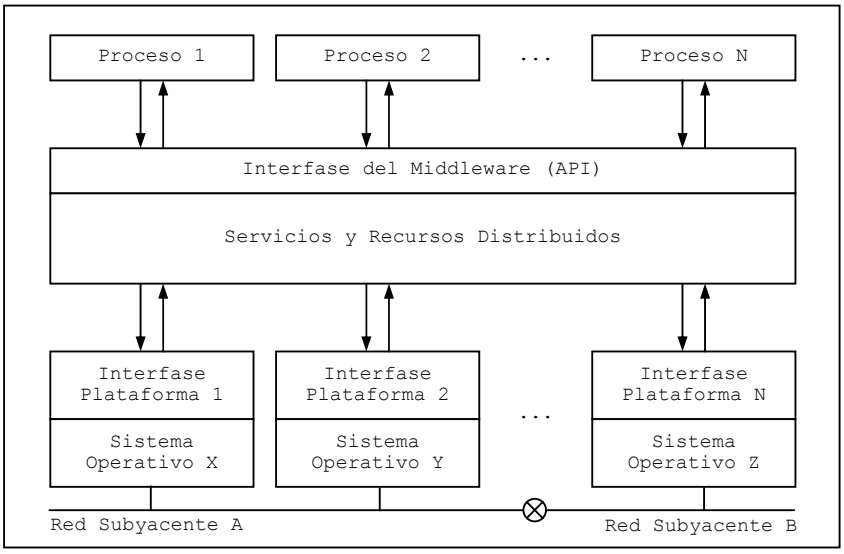


Ilustración 16 - Esquema de Middleware típico

El middleware debe presentar las siguientes características [Bray 97]:

- **Interoperabilidad**: es la habilidad de dos o más sistemas o componentes de intercambiar información y utilizarla [IEEE 90]. El middleware debe permitir el intercambio de información entre hardware y sistemas operativos distintos e inclusive, entre implementaciones de distintos fabricantes (siempre que adhieran a un mismo estándar).

- **Transparencia**: se deben ocultar las diferencias de las capas subyacentes de forma tal que el middleware sea visto en forma consistente por las aplicaciones y procesos que se construyen sobre él. Por ejemplo, debe ser posible ubicar servicios y recursos distribuidos sin recurrir a direcciones de red.

- **Confiabilidad y Disponibilidad**: un middleware es confiable si es capaz de cumplir con la funcionalidad que se le requiere bajo ciertas condiciones durante un período específico de tiempo [IEEE 90]. La disponibilidad es la capacidad de prestar un servicio correcto. Para lograr middlewares confiables y de alta disponibilidad, es conveniente la distribución de recursos y servicios evitando componentes centralizados. No puede suceder que por fallar una máquina o un proceso, todo el middleware quede inutilizado.

- **Escalabilidad**: es una medida de la facilidad con la que un sistema o componente puede ser modificado para acomodarse a la envergadura del problema a resolver [IEEE 90]. A medida que crece la cantidad de procesos de aplicación, el middleware no debe perder funcionalidad ni performance en forma excesiva.

- **Abstracción**: un middleware tiene que facilitar la construcción de aplicaciones distribuidas. Debe proveer un mayor nivel de abstracción que las capas inferiores de forma que el desarrollador de aplicaciones, en lugar de manejar entidades como conexiones, puertos o semáforos, trabaje con abstracciones de mayor nivel como pueden ser objetos, eventos y transacciones [Bray 97].

3.2 Interfase del middleware (API)

Todo middleware provee una interfase de programación de aplicaciones (API). Está formada por un conjunto de entidades (funciones, procedimientos, clases y métodos, etc.) que permiten a un desarrollador construir aplicaciones distribuidas sobre él. Al momento de definir el API de un middleware se presentan dos objetivos contrapuestos [Bray 97] [Aiken 00].

Por un lado, es deseable que el middleware sirva para construir la mayor cantidad posible de aplicaciones, inclusive de distinta naturaleza. Para conseguir esto, debe manejar múltiples servicios y recursos lo cual, haría crecer su API. O sea, estaría compuesta por un mayor numero de funciones o clases.

Por otro lado, los desarrolladores de aplicaciones desean trabajar con un ambiente simple y manejable. Por este motivo, les preocupa principalmente minimizar el tamaño del API que deben conocer para acceder a los servicios y recursos provistos por el middleware.

Las interfases ofrecidas por un middleware tienen que incluir mínimamente los siguientes elementos [Aiken 00]:

- **Interfase de descubrimiento de ambiente**

  Permite descubrir e identificar recursos de hardware, estado y capacidad de las redes subyacentes, aplicaciones y servicios remotos o información de los usuarios.

- **Interfase de ejecución remota**

  Permite solicitar a recursos o servicios administrados por el middleware la ejecución remota de cierta funcionalidad. Puede ser una llamada remota a un procedimiento o una invocación de un método.

- **Interfase de administración de datos**

  Permite manipular datos entre caches distribuidos, replicación de sistemas de archivos o bien, directamente almacenamiento de datos.

- **Interfase de administración de procesos**

  Permite combinar el movimiento o manipulación de datos con la ejecución remota o bien, encadenar múltiples pasos de procesamiento.

3.3 Tipos de middleware

De acuerdo a la funcionalidad ofrecida y al criterio de distribución de recursos y servicios, se pueden definir distintos tipos de middleware [Bray 97]:

3.3.1 Transaction processing (TP) monitors

Esta pensado para dar a ambientes cliente / servidor la capacidad de correr y administrar en forma eficiente y confiable aplicaciones transaccionales. El funcionamiento es sencillo: una gran cantidad de clientes envían solicitudes transaccionales a un monitor de procesamiento, el cual las multiplexa en un número controlado de rutinas de servicio. Estas rutinas son las que finalmente acceden a los almacenamientos de datos.

La multiplexación permite mejorar la performance, la escalabilidad y la flexibilidad del sistema en su conjunto. Es una tecnología probada, en uso desde hace unos 30 años, pero que presenta como gran desventaja la utilización de lenguajes como el COBOL para implementación de aplicaciones [Sadoski 97].

3.3.2 Remote procedure call sincrónico (RPC)

Es una infraestructura cliente / servidor que permite a una aplicación distribuida correr en múltiples maquinas heterogéneas. En este caso, la distribución consiste en mover rutinas del cliente y colocarlas en el programa servidor. En el cliente, las rutinas son reemplazadas por otras con igual nombre y parámetros, pero con código generado por RPC llamado “stub”. Este código se encarga de efectuar la llamada remota o sea, contactar al programa servidor, solicitar la ejecución del procedimiento y esperar la respuesta del mismo. En un esquema esto sería:

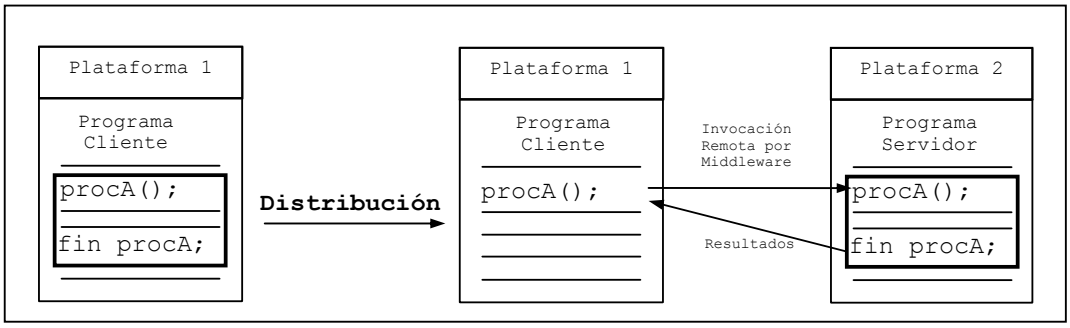


Ilustración 17 - Funcionamiento de RPC

La complejidad inherente a construir una aplicación distribuida se reduce a mantener igual la sintaxis de una llamada remota y de una llamada local, sin importar que cliente y servidor se ubiquen o no en la misma máquina. Esto se consigue mediante un protocolo sincrónico tipo request / response, que hace que los clientes se bloqueen hasta que el servidor complete la ejecución del procedimiento.

Entre las ventajas de este middleware se cuentan la interoperabilidad, portabilidad y flexibilidad. Sin embargo, no es el middleware indicado para aplicaciones que involucren objetos distribuidos o para desarrollar con lenguajes orientados a objetos ya que presenta todas las características de los lenguajes procedurales. Actualmente, junto con los modelos de comunicaciones de mensajería sincrónica o asincrónica, es el modelo mas utilizado para comunicaciones en los middleware orientados a mensajes para servicios Web sobre protocolos de Internet [XML-RPC].

Existen dos variantes de este tipo de middleware ampliamente usadas: Open Network Computing (ONC) RPC, desarrollado por Sun, y Distributed Computing Environment (DCE)

RPC, desarrollado por Open Software Foundation (OSF). Ambas han sido implementadas en gran variedad de sistemas operativos [Vondrak 97].

3.3.3 Message-oriented middleware (MOM)

Provee intercambio de datos de programa a programa, mediante envío y recepción de mensajes, permitiendo de esta forma la creación de aplicaciones distribuidas. Los mensajes no se direccionan a una máquina en particular sino a una cola por lo cual, al realizar un envío, no es necesario conocer la ubicación del programa receptor. Estas colas también pueden usarse como almacenamiento temporario en caso que el programa receptor no este disponible o se encuentre desconectado.

Normalmente, ofrecen un servicio peer-to-peer, asincrónico y no imponen ningún formato a los mensajes. Este debe ser acordado previamente por las aplicaciones que participan en el intercambio [Bray 97].

Una de las ventajas de este middleware es mayor flexibilidad que RPC para construir aplicaciones por el tipo de servicio ofrecido (peer-to-peer asincrónico). Esto último, sin embargo, puede traer problemas ya que no hay forma de frenar a un transmisor excesivamente rápido, que puede llenar la cola del receptor o bien, sobrecargar la red.

Otra de las desventajas es que este tipo de middleware es implementado normalmente como un producto propietario, incompatible con otras implementaciones. También es acotada la compatibilidad de estos productos con distintas plataformas de hardware y sistemas operativos. Esto compromete la interoperabilidad así como la portabilidad del MOM.

3.3.4 Object request brokers (ORBs)

Este tipo de middleware administra la comunicación y el intercambio de solicitudes y respuestas entre objetos, ocultando todos los detalles de implementación y dejando que el desarrollador de aplicaciones distribuidas sólo se preocupe por las interfases de los objetos [Wallnau 97]. Las ORBs sirven como un medio para entregar las solicitudes y respuestas entre los objetos de un sistema [Kleindienst 96] en forma transparente, sin importar que estos sean locales o remotos.

En cierta forma, una ORB provee un directorio de servicios provistos por objetos y ayuda a establecer conexiones entre los clientes y estos servicios. En un esquema esto sería:

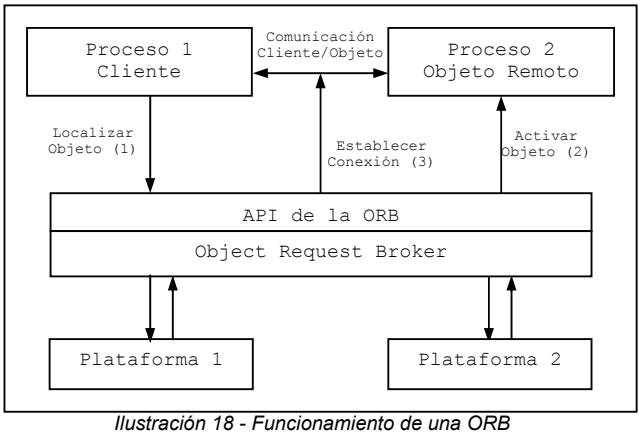


Ilustración 18 - Funcionamiento de una ORB

Un cliente desea invocar un servicio ofrecido por algún objeto. Entonces, solicita a la ORB que ubique el objeto que provee ese servicio *(1)*. Cuando la ORB lo encuentra, prepara la implementación *(2)* para dar servicio a la solicitud en cuestión. Finalmente, la ORB instruye al

cliente y al servidor para que se conecten y empiecen a dialogar *(3)*. Todo el proceso anterior se da internamente en la ORB y resulta completamente transparente para el desarrollador. Para él, simplemente se está llamando a un método de un objeto.

Vale aclarar que el cliente también puede ser un objeto y ofrecer sus propios servicios que a su vez, pueden ser solicitados por otros objetos. Esto está indicando que se soporta un modelo de comunicaciones peer-to-peer más que un cliente / servidor puro como en otros middlewares.

Las funciones más relevantes de una ORB son [Corba Core 02]:

- Definición de interfases para los objetos
- Ubicación de objetos para cada solicitud de servicio
- Activación de objetos para cada solicitud de servicio, si es necesario
- Comunicación entre clientes y objetos

La ORB permite a los clientes ver todos los objetos como locales inclusive cuando estos residen en otros procesos o bien, en otras máquinas. También permite que los objetos oculten sus detalles de implementación como ser el lenguaje de programación usado para su construcción, su ubicación física, y el sistema operativo y el hardware de la máquina sobre la cual corren. Estas dos características se traducen en interoperabilidad y transparencia [Wallnau 97].

Las implementaciones de este tipo de middleware más ampliamente usadas son: Common Object Request Broker Architecture (CORBA) del Object Management Group, Distributed Component Object Model (COM/DCOM) de Microsoft y Java Remote Method Invocation (RMI) de Sun.

Dentro de estas opciones, la más adecuada resulta CORBA, es un estándar, es multiplataforma y no es una solución propietaria, ofrece interoperabilidad entre implementaciones de fabricantes distintos y soporte para múltiples lenguajes de implementación de objetos. Hay disponibilidad de implementaciones de código abierto para plataformas diversas, por lo tanto, es posible adaptar sus protocolos para operar sobre un bus industrial y cumplir con los requisitos de estas redes.

DCOM requiere exclusivamente plataforma de sistemas operativos Microsoft, la cual no es habitualmente la plataforma de operación de muchos de los dispositivos que pueden requerir sistemas operativos en tiempo real y no fue pensado para operar sobre un bus industrial, ni provee una forma de adaptarse a los requerimientos de este tipo de redes.,

Por último, RMI es multiplataforma pero fue diseñado para operar sobre redes de datos con protocolos TCP/IP, que no cumplen con los requerimientos de un bus industrial. Aunque puede operar sobre CORBA usando IIOP (TCP/IP sobre CORBA), TCP no garantiza los requerimientos de tiempo de un sistema de control distribuido.

Se describe a continuación la implementación de CORBA.

3.4 CORBA

3.4.1 Introducción

La propuesta de middleware del Object Management Group (OMG) va más allá de definir sólo una ORB. Se plantea una arquitectura llamada Object Management Architecture (OMA) que engloba la visión del OMG de lo que debería ser un ambiente orientado a componentes conectables y listos para usar (componentes plug-and-play). La arquitectura está compuesta por los siguientes elementos [Soley 95]:

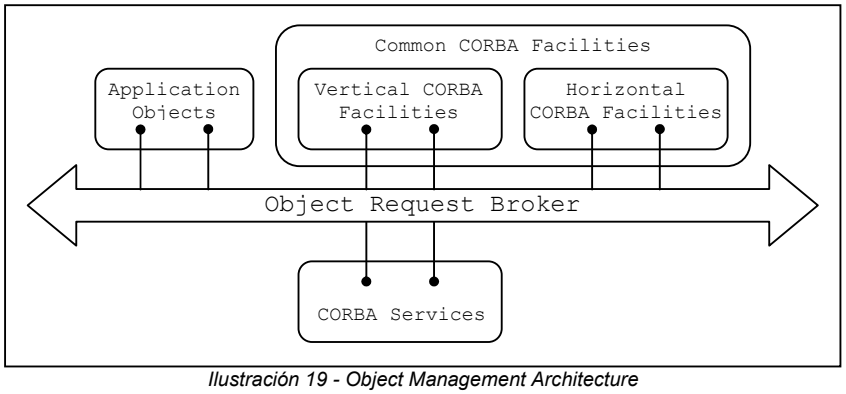


Ilustración 19 - Object Management Architecture

- **Object Request Broker**: es el elemento central de la arquitectura ya que se encarga de las comunicaciones entre objetos en un ambiente distribuido. Provee la infraestructura que permite a estos comunicarse, independientemente de la plataforma donde corren y de cómo fueron implementados (lenguaje de programación, arquitectura del hardware, sistema operativo, etc.). En fin, garantiza portabilidad e interoperabilidad a través de una red de maquinas heterogéneas. Common Object Request Broker Architecture (CORBA) es la arquitectura de ORB propuesta por el OMG.

- **CORBA Services**: este elemento estandariza la administración del ciclo de vida de los objetos. Incluye servicios para crear objetos, controlar el acceso a los mismos, llevar control de objetos relocalizados y para mantener consistentes las relaciones entre grupos de objetos. Esta estandarización lleva a la consistencia entre aplicaciones diferentes y a un incremento en la productividad de los desarrolladores de aplicaciones distribuidas.

- **Vertical CORBA Facilities**: componentes que proveen soluciones a problemas de negocios dentro de un mercado vertical específico como ser la actividad financiera, bancaria, hospitalaria, manufacturera, etc.

- **Horizontal CORBA Facilities**: componentes que proveen funcionalidad a través de un negocio o a través de una empresa. Aplicaciones de diferentes áreas podrían necesitar estos componentes comunes como por ejemplo, componentes para contabilidad, para envío de correo, pago de sueldos, etc.

- **Application Objects**: esta parte de la arquitectura representa aquellos objetos de aplicación que realizan tareas especificas de los usuarios y que invocan métodos en objetos remotos (o son invocados) a través de la ORB.

El presente trabajo se centrará en describir principalmente la ORB. Para esto se basa en la especificación central de CORBA [Corba Core 02] y de CORBA para ambientes con restricciones de recursos [Corba Min 02], ambas publicadas por el OMG.

3.4.2 Modelo de objetos CORBA

El modelo de objetos de CORBA permite describir los Application Objects de la OMA y la forma en que se construyen aplicaciones distribuidas a partir de estos. El modelo está compuesto por las siguientes entidades:

- **Cliente**: un sistema de objetos provee servicios a clientes. Así, un cliente de un servicio es cualquier entidad o agente capaz de solicitar el servicio en cuestión.

- **Objeto**: entidad abstracta, encapsulada e identificable, que provee uno o más servicios que pueden ser solicitados por un cliente.

- **Referencia**: es un valor que denota un objeto en forma confiable. Los clientes usan referencias para identificar los objetos a los cuales desean solicitar servicios. Confiabilidad, en este caso particular, quiere decir que una referencia va a identificar el mismo objeto cada vez que sea usada en una solicitud de servicio.

- **Implementación de un objeto**: es la entidad encargada de realizar las actividades computacionales necesarias para ejecutar el comportamiento asociado al servicio solicitado para cierto objeto.

- **Servidor**: es cualquier entidad o agente capaz de recibir solicitudes de servicio, procesarlas y enviar el resultado al cliente que inició la solicitud. Puede contener una o más implementaciones de objeto, usadas para atender las diferentes solicitudes.

- **Operación**: es una entidad identificable que denota una primitiva indivisible de provisión de servicio. El nombre de la operación es directamente el identificador de la misma. El acto de solicitar un servicio puede ser referido también como una invocación de operación.

- **Interfase**: una interfase es una descripción de las operaciones que un cliente puede solicitar a un objeto que adhiere a esta. Un objeto puede adherir a múltiples interfases. El tipo del objeto está dado por las interfases a las que adhiere.

- **Solicitud de servicio (request)**: toda la secuencia de eventos causalmente relacionados que se dan entre que un cliente inicia una solicitud de servicio y el último de los eventos causados por dicha iniciación. La información asociada a una solicitud consiste en una operación, un objeto destino, cero o más parámetros y un contexto de invocación opcional.

En las próximas secciones, usando estas definiciones como marco de referencia, se describirán todos los componentes de CORBA, las relaciones existentes entre estos y su funcionamiento.

3.4.3 Object request broker (ORB)

El modelo de objetos de CORBA plantea la existencia de agentes clientes y agentes servidores que pueden correr en máquinas distintas. Los clientes solicitan servicios provistos por objetos, ubicándolos mediante referencias a los mismos. Los objetos son implementados en estos servidores mediante entidades llamadas implementaciones de objetos.

La ORB interviene en la interacción entre clientes y servidores, haciéndose responsable de las siguientes tareas:

- Proveer todos los mecanismos para ubicar la implementación del objeto correspondiente a cada solicitud.

- Preparar la implementación del objeto para recibir y procesar la solicitud enviada por el cliente.

- Proveer los mecanismos y protocolos de comunicación entre las máquinas donde corren los clientes y los servidores (implementaciones de los objetos), ocultando todos los detalles de implementación como por ejemplo, protocolos de transporte o la arquitectura de las máquinas.

Una ORB CORBA está formada por los siguientes componentes:

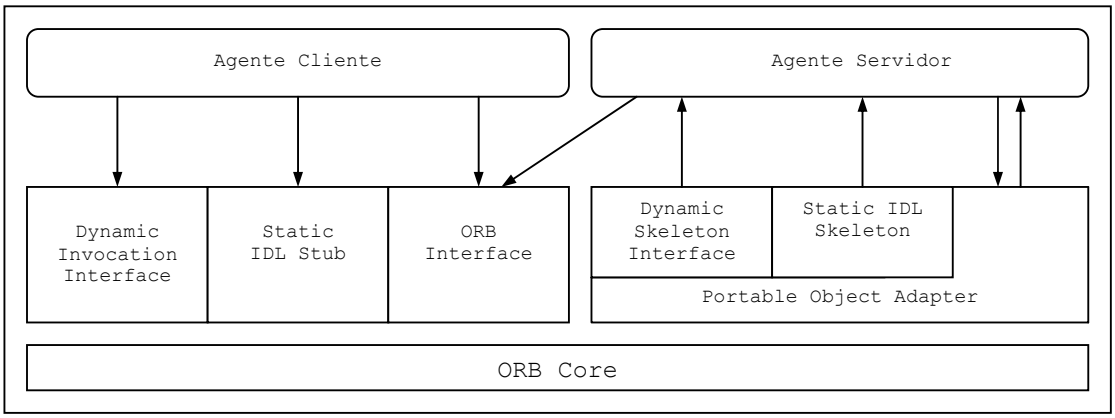


Ilustración 20 - Componentes de una ORB CORBA

- **Dynamic Invocation Interface (DII)**: forma parte del mecanismo de invocación dinámico. La interfase está formada por un conjunto de funciones que permiten a los clientes construir solicitudes de servicio en forma dinámica.

- **Static IDL Stub**: código generado a partir de la descripción de la interfase de un objeto. Se compila y se enlaza con el programa cliente, permitiendo así solicitar servicios. Existe un stub por cada tipo de objeto o sea, para cada interfase.

- **ORB Interface**: es una interfase que agrupa funcionalidad genérica de la ORB como configuración, inicialización, generación y publicación de referencias de objetos para su uso por los clientes, etc.

- **Dynamic Skeleton Interface (DSI)**: forma parte del mecanismo de invocación dinámico. Es la contraparte de DII del lado de las implementaciones de objetos o servidores.

- **Static IDL Skeleton**: código generado a partir de la descripción de la interfase de un objeto. Se compila y se enlaza con el programa servidor, permitiendo llamar a la implementación del objeto correspondiente cuando llega una solicitud. Existe un skeleton por cada tipo de objeto o sea, por cada interfase.

- **Portable Object Adapter (POA)**: permite a los programas servidores registrar en la ORB los objetos que implementan. De esta forma, cuando llega una solicitud, ubica el objeto destinatario de la misma y dispara la llamada a la operación que corresponda.

- **ORB Core**: es una abstracción que contiene el núcleo de la ORB o sea, su funcionalidad principal. Incluye los mecanismos y protocolos (de sesión y de presentación) usados para la comunicación entre las máquinas donde se distribuyen los clientes y servidores.

La ORB es normalmente descripta como una entidad abstracta, accedida tanto por clientes como por servidores. Dentro de esta abstracción, queda encapsulada la red que conecta las máquinas así como los mecanismos y protocolos usados para la comunicación entre ellas. En la ilustración anterior, todo esto quedaría adentro del núcleo de la ORB (ORB Core).

En la realidad, una ORB puede ser implementada de múltiples maneras. Por ejemplo, puede ser construida como un conjunto de librerías, enlazadas en forma estática o dinámica con los programas cliente y servidor. Estas librerías son conscientes de la red subyacente e implementan los mecanismos y protocolos de comunicación que sean necesarios.

A pesar de este tipo de implementación, desde los programas de aplicación, la ORB se sigue viendo en forma abstracta como una única entidad gracias a la cual, cualquier cliente puede acceder a cualquier objeto en forma transparente e independiente de su ubicación.

A partir de ahora, los elementos que implementan la ORB y que corren en cada máquina, van a ser llamados, en forma colectiva, instancia de la ORB.

3.4.4 Portable object adapter (POA)

El Portable Object Adapter (POA) es un componente de la ORB. Cumple principalmente con dos funciones. En primer lugar, debe actuar como adaptador entre la ORB y la implementación de los objetos. En segundo lugar, debe definir una interfase estándar entre ORB e implementaciones de objetos de forma de garantizar la portabilidad de estas últimas.

El siguiente esquema describe la forma en que el POA interactúa con la ORB y la implementación de los objetos de la aplicación del usuario:

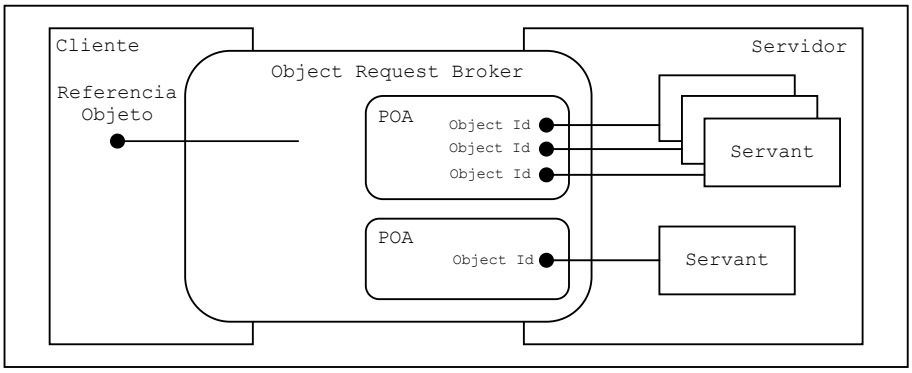


Ilustración 21 - Interacción del POA con la ORB y servants

Un servant es una entidad de un lenguaje de programación que implementa las rutinas para dar servicio a los requests o solicitudes sobre uno o más objetos. O sea, es una entidad (clase, módulo o función, dependiendo del lenguaje usado) que constituye la implementación real de uno o más objetos abstractos.

El POA es un componente relevante únicamente en los servidores. Permite crear objetos CORBA, referenciables por los clientes y residentes en los servidores, a partir de los servants antes descriptos. O sea, permite crear objetos CORBA a partir de los objetos abstractos implementados en cierto lenguaje de programación. Entonces, la diferencia entre un objeto CORBA y un objeto abstracto (o lo que es igual, su implementación llamada servant) es que los primeros son referenciables y accesibles en forma remota por los clientes.

Para poder crear un objeto CORBA a partir de un servant es necesario asignar a cada uno de estos un identificador único llamado Object Id. Este identificador se implementa como un string de caracteres sin ningún formato predefinido.

Los servants se registran con el POA bajo uno o más identificadores, asignados por la aplicación o en forma automática, por el mismo Object Adapter. Al registrarse, los servants pasan a estar activos y se crea un objeto CORBA, accesible en forma remota, para cada identificador. Una vez completado este proceso de activación, los servidores pueden publicar las referencias a los objetos CORBA resultantes para que sean usadas por los clientes

Estas referencias reciben el nombre de Interoperable Object Reference o IOR. La forma más sencilla de publicar una referencia es convertir un IOR a string y enviarla a los clientes de alguna forma (no definida por el estándar CORBA). El string resultante comienza con el prefijo “IOR:”, seguido de una secuencia de dígitos hexadecimales de longitud variable.

El cliente mantiene una referencia a cierto objeto CORBA, la cual va a usar para enviarle una solicitud de servicio. Usando alguno de los mecanismos de invocación (DII o Static IDL stub), el cliente accede al núcleo de la ORB y esta, se encarga de llevar la solicitud o request hasta el proceso servidor.

En el mensaje de request que llega al servidor se incluye el Object Id, contenido en forma transparente en la referencia usada por el cliente. El POA busca el servant registrado bajo ese identificador usando uno de varios métodos disponibles. Una vez que lo encuentra, usando también alguno de los mecanismos de invocación, DSI o Static IDL skeleton, accede al servant encontrado e invoca la entidad (proceso, función o procedimiento) que implementa la operación solicitada en el request.

Un servidor puede tener más de un POA, siendo cada instancia identificable en el contexto del servidor. Cada instancia provee un espacio de validez de nombres tanto para Object Ids como para otras instancias del POA, constituyéndose así un espacio de nombres

jerárquico. Esto hace que un mismo Object Id puede usarse para más de un objeto CORBA, siempre que estos sean creados con instancias distintas del POA.

El comportamiento de un POA está determinado por una serie de políticas o policies, las cuales sólo pueden ser especificadas al momento de crear una instancia del mismo. Las políticas de un POA existente no pueden ser modificadas ni heredadas del POA padre. Estas políticas son:

Política	Descripción	Valores
Thread Policy	Determina como deben administrarse y asignarse los threads a los requests que debe procesar la ORB.	• **ORB_CTRL_MODEL:** la ORB es responsable de asignar threads a los requests en nombre del POA. En ambiente multithreaded, se pueden atender solicitudes en forma concurrente. • **SINGLE_THREAD_MODEL:** las solicitudes se procesan en forma secuencial. Las llamadas up-call a código implementado por el usuario, servants y servant managers, se hacen en forma segura para código que probablemente no sea thread-safe. • **MAIN_THREAD_MODEL:** es igual al anterior excepto que cierto código puede ejecutarse en un thread especial llamado main thread.
Lifespan Policy	Esta política se refiere al período de vida de los objetos. O sea, si el POA va a trabajar con objetos persistentes o no.	• **TRANSIENT:** un objeto no puede sobrevivir a la instancia del POA en la cual fue creado originalmente. • **PERSISTENT:** un objeto puede sobrevivir a la instancia del POA en la cual fue creado originalmente. Cuando llegue un request para este objeto y el POA asociado al mismo ya haya sido desactivado, va a ser necesario un mecanismo de creación de POAs on-demand (no definido por el estándar).
Object Id Uniqueness Policy	Determina la cantidad de Object Ids que puede soportar el POA (indirectamente, se trata de la cantidad de objetos por servant).	• **UNIQUE_ID:** se soporta un único Object Id por servant. • **MULTIPLE_ID:** se soportan múltiples Object Ids por servant.
Id Assignment Policy	Determina quién asigna Object Ids a los objetos creados con el POA.	• **USER_ID:** los Object Ids para los objetos creados con el POA son asignados únicamente por la aplicación. • **SYSTEM_ID:** los Object Ids para los objetos creados con el POA son asignados únicamente por el mismo.
Servant Retention Policy	Se relaciona con la siguiente política. Determina que hacer con los servants que se obtienen para procesar los requests.	• **RETAIN:** se retienen los servants. O sea, el POA mantiene una tabla interna, llamada Active Object Map, en la cual asocia un Object Id con un servant activo. Al asociar el servant, el POA lo está reteniendo. • **NON_RETAIN:** los servants no son retenidos por el POA.

Request Processing Policy	Determinan el mecanismo que debe usar el POA para asociar un Object Id con un servant activo al momento de atender un request.	• **USE_ACTIVE_OBJECT_MAP_ONLY**: si la política RETAIN está activa, el POA asocia en el Active Object Map un Object Id con un servant activo. Luego, al llegar un request lo único que se debe hacer es buscar en la tabla. • **USE_DEFAULT_SERVANT**: se permite registrar con el POA un servant por default para todos los requests que lleguen a la ORB. • **USE_SERVANT_MANAGER**: si se usa esta política, un servant manager implementado por el usuario se registra con el POA. Así, cuando llega un request se invoca al servant manager para que indique que servant usar. Si RETAIN está activa, el servant devuelvo por el servant manager se guarda en el Active Object Map.
Implicit Activation Policy	Determina si el POA puede o no activar servants no activos si alguna operación lo requiere.	• **IMPLICIT_ACTIVATION**: se permite al POA activar los objetos. • **NO_IMPLICIT_ACTIVATION**: no se permite al POA activar los objetos.

Tabla 23 - Políticas del POA

Para que una aplicación pueda crear y activar servants, creando así nuevos objetos CORBA, primero necesita obtener una instancia del POA. La ORB, mediante su interfase de inicialización, ofrece una instancia especial llamada Root POA. El comportamiento de esta instancia está dado por las siguientes políticas:

Política	Descripción
Thread Policy	ORB_CTRL_MODEL
Lifespan Policy	TRANSIENT
Object Id Uniqueness Policy	UNIQUE_ID
Id Assignment Policy	SYSTEM_ID
Servant Retention Policy	RETAIN
Request Processing Policy	USE_ACTIVE_OBJECT_MAP_ONLY
Implicit Activation Policy	IMPLICIT_ACTIVATION

Tabla 24 - Políticas por defecto del Root POA

Si estas políticas no son las adecuadas para el objeto que se desea crear, el usuario puede crear un nuevo POA a partir del Root POA y asignarle las políticas que resulten más convenientes.

Por último, se incluye un ejemplo de activación de un servant, en una aplicación desarrollada en lenguaje C, usando la ORB ORBit versión 0.5.13 [ORBit 04] para Linux:

```
#include <stdlib.h>
#include <orb/can_orb.h>
#include "empty.h"

Empty empty_client=CORBA_OBJECT_NIL;

CORBA_Environment ev;
CORBA_ORB orb;

int main (int argc, char *argv[]) {
	PortableServer_ObjectId objid={0,sizeof("Empty_Prueba"),"Empty_Prueba"};
	POA_Empty poa_empty_servant;
	PortableServer_POA poa;

	CORBA_exception_init(&ev);
	orb=CORBA_ORB_init(&argc,argv,"can_orb-local-orb",&ev);

	/* Se obtiene una referencia al Root POA */
	poa=(PortableServer_POA)CORBA_ORB_resolve_initial_references(orb,"RootPOA",&ev);
	PortableServer_POAManager_activate(PortableServer_POA__get_the_POAManager(poa,&ev),&ev);

	/* Se crea el servant que implementa la interfase Empty */
	POA_Empty__init(&poa_empty_servant,&ev);

	/* Se activa el objeto con un Object Id definido por la aplicación "Empty_Prueba" */
	PortableServer_POA_activate_object_with_id(poa,&objid,&poa_empty_servant,&ev);

	/* Se obtiene un CORBA object, accesible por los clientes */
	empty_client=PortableServer_POA_servant_to_reference(poa,&poa_empty_servant,&ev);

	CORBA_ORB_run(orb,&ev);
}
```

Código 3 - Activación de un servant en C usando ORBit 0.5.13

3.4.5 Descripción de interfases

Una de las funciones que debe cumplir una ORB es permitir definir interfases para los objetos que componen las aplicaciones distribuidas. En CORBA, esto se hace con un lenguaje abstracto, independiente y común para todas las plataformas, llamado Interface Definition Language (OMG IDL).

Este lenguaje es puramente descriptivo y sólo sirve para definir las interfases que implementan los objetos, las operaciones que estas ofrecen y los parámetros que toma cada una. No sirve para implementar clientes ni objetos. Para esto, se puede usar cualquier lenguaje de programación para el cual se haya definido un mapeo entre los elementos que componen una definición IDL y estructuras propias del lenguaje. Existen mapeos estandarizados por la OMG para diversos lenguajes de programación orientados a objetos como C++, SmallTalk o Java o bien, no orientados a objetos como puede ser C.

El separar la definición de la implementación, permitiendo usar lenguajes de programación distintos para implementar los objetos de una misma aplicación distribuida, incrementa la portabilidad y la flexibilidad. También permite mejorar la interoperabilidad entre plataformas heterogéneas ya que los intercambios entre clientes y objetos se definen en forma abstracta.

A continuación se incluye una descripción de los elementos que componen este lenguaje:

- **Identificadores y comentarios**

  Un identificador es una secuencia arbitrariamente larga de caracteres ASCII alfabéticos, numéricos o bien, de guiones bajos. El primer carácter tiene que ser siempre un carácter ASCII alfabético.

El lenguaje no es case sensitive por lo cual identificadores que sólo difieren en mayúsculas o minúsculas producen errores de compilación. De cualquier forma, las ocurrencias de un identificador, posteriores a la declaración inicial del mismo, deben respetar la combinación de mayúsculas y minúsculas usada originalmente. De esta forma, se permite un mapeo natural a lenguajes case sensitive como C.

Los comentarios se manejan en forma similar a ANSI C++. Los caracteres "/*" abren un comentario mientras que "*/", lo cierran. También es posible usar "//", que indica el comienzo de un comentario, el cual termina al final de la misma línea que incluye la combinación anterior.

- **Alcance de identificadores**

El contenido completo de un archivo OMG IDL, junto con los contenidos de todos los archivos incluidos por directivas al preprocesador "#include", forman un scope de nombres. Cualquier definición que no aparezca dentro de este scope forma parte del scope global.

La mayoría de los elementos de una definición, como interfases, operaciones, y tipos, forman un scope de nombres. Sin embargo, existe un elemento en OMG IDL cuyo único propósito es definir scopes de nombres: el módulo. Se lo declara mediante la palabra clave "module" de la siguiente forma:

```
module M_1 {
    module M_1_1 {
        module M_1_1_1 {
            ...
        };
    };

    module M_1_2 {
        ...
    };
};
```

Código 4 – Ejemplo de módulos en OMG IDL

Para acceder a los elementos definidos en este scope de nombres desde otro scope es necesario usar el nombre calificado como por ejemplo, "M_1::M_1_1" o "M_1::M_1_2". Sin embargo, OMG IDL incluye la sentencia "import" para hacer visible un scope de nombres en el contexto de otra especificación:

```
import M1::M_1_1;
import M1::M_1_2;

module M_3 {
    ...
};
```

Código 5 – Ejemplo de import en OMG IDL

De esta forma, dentro del módulo M_3, elementos de M_1_1 o M_1_2 pueden ser referenciados en forma directa sin necesidad de usar sus nombres calificados.

- **Tipos de dato básicos**

Estos son los tipos abstractos básicos definidos por OMG IDL. Son mapeados a tipos nativos a través del mapeo de lenguaje seleccionado para implementar objetos o clientes.

Tipo	Descripción
Tipos enteros	
short	De -2^{15} a $2^{15}-1$, tamaño mínimo 16 bits.
long	De -2^{31} a $2^{31}-1$, tamaño mínimo 32 bits.
long long	De -2^{63} a $2^{63}-1$, tamaño mínimo 64 bits.
unsigned short	De 0 a $2^{16}-1$, tamaño mínimo 16 bits.
unsigned long	De 0 a $2^{32}-1$, tamaño mínimo 32 bits.
unsigned long long	De 0 a $2^{64}-1$, tamaño mínimo 64 bits.
Tipos de punto flotante	
float	IEEE 754, simple precisión (32 bits).
double	IEEE 754, doble precisión (64 bits).
long double	IEEE 754, doble extendido (15 y 64 bits).
Otros tipos	
char	Cantidad de 8 bits que codifica un carácter de 1 byte de cualquier juego de caracteres (code set) orientado a bytes.
wchar	Codifica un carácter multibyte de cualquier juego de caracteres (code set). Puede sufrir conversiones al ser transmitido.
boolean	Sólo puede tomar los valores TRUE o FALSE. Su tamaño no está definido.
octet	Cantidad de 8 bits. Se garantiza que no sufre ninguna conversión durante la transmisión.
any	Permite almacenar un valor de cualquier tipo expresado en OMG IDL. Internamente es un TypeCode (descripción del valor) y el valor en sí. Cada mapeo de lenguaje ofrece operaciones para acceder al TypeCode interno.

Tabla 25 - Tipos de dato básicos de OMG IDL

- **Definición de tipos y constantes**

OMG IDL permite al programador definir nuevos tipos de dato a partir de los tipos básicos. Para hacer esto, se debe usar la palabra clave “typedef” de la siguiente forma:

```
typedef float t_atmos_pressure;
typedef unsigned long t_device_id;
```

Código 6 - Ejemplo de OMG IDL typedef

En cuanto a la definición de constantes, se debe usar la palabra clave “const” de la siguiente forma:

```
const float NORMAL_ATMOS_PRESSURE = 1.0;
const unsigned long MAX_ARRAY_SIZE = 10000;
const long FILTER_MASK = 0xC000FFFF;
const char NEW_LINE = '\n';
const string DEVICE_NAME = "/dev/hda1";
```

Código 7 - Ejemplo de OMG IDL const

Las constantes sólo pueden ser de alguno de los tipos básicos antes presentados, de tipo string o bien, un decimal de punto fijo. No se pueden usar tipos definidos por el programador o el tipo "any". El valor asignado a la constante tiene que ser consistente con el tipo de la misma. Por ejemplo, si se intenta asignar un valor entero negativo a una constante de tipo unsigned short, se produce un error de compilación

- **Tipos de dato complejos**

Estos son tipos más complejos y que son definidos por el programador de OMG IDL. En primer lugar, se encuentran los tipos construidos:

Tipo	Descripción
struct	La palabra clave "struct" permite agrupar un conjunto de variables dentro de una estructura. Una vez definida, constituye un nuevo tipo de dato. A este tipo se le puede asociar un nombre mediante la palabra clave "typedef". Un ejemplo de estructura podría ser:

```
struct basic_device {
    unsigned long id;
    unsigned long firmware_rev_major;
    unsigned long firmware_rev_minor;
    boolean enabled;
};
typedef struct basic_device t_basic_device;
```

Código 8 - Ejemplo de OMG IDL struct

enum	La palabra clave "enum" permite definir una lista ordenada de identificadores. Una vez definida, constituye un nuevo tipo de dato.

```
typedef enum device_states {
    idle,
    bound_tx,
    bound_rx,
    waiting
} t_device_states;
```

Código 9 - Ejemplo de OMG IDL enum

Se pueden definir hasta 2^{32} identificadores dentro de una enumeración. El mapeo del lenguaje debe usar un tipo nativo con igual capacidad para representarlos. Si el tipo nativo usado permite comparaciones, se debería respetar el orden dado por el código OMG IDL.

union	La palabra clave "union" permite definir uniones discriminadas que vendrían a ser una mezcla entre una "union" y una sentencia "switch" del lenguaje C. Las uniones IDL requieren que se especifique un campo discriminador de la siguiente forma:

```
enum device_type { producer, consumer };
union device switch(device_type) {
    case producer:
        unsigned long evt_source_id;
    case consumer:
        unsigned long evt_dest_id;
};
```

Código 10 - Ejemplo de OMG IDL union

El acceso al discriminador y los miembros relacionados es dependiente del mapeo de lenguaje que se use.

Tabla 26 - Tipos de dato construidos de OMG IDL

Luego, se encuentran los tipos template que incluyen secuencias, cadenas de caracteres y decimales de punto fijo:

Tipo	Descripción
sequence	La palabra clave "sequence" permite definir un array unidimensional con tamaño máximo (fijado al compilar) y una longitud (calculada en tiempo de ejecución). Las secuencias con restricción de tamaño máximo se llaman bounded. Sin restricción, se llaman unbounded.

```
/* Secuencia bounded de structs basic_device */
struct basic_device {
    unsigned long id;
    ...
};
typedef sequence<basic_device,10> t_basic_device_list;

/* Secuencia unbounded de longs */
typedef sequence<long> t_long_seq;
```

Código 11 - Ejemplo de OMG IDL sequence

El acceso al tamaño máximo y a la longitud es dependiente del mapeo de lenguaje que se use.

string	Representa una secuencia de char excepto null. Hay bounded strings (con tamaño máximo especificado) y unbounded strings (sin tamaño máximo). Antes de la secuencia, hay que pasar la longitud en alguna forma dependiente del mapeo de lenguaje usado.

```
/* bounded string */
const unsigned short DEVNAME_LENGTH=256;
typedef string<DEVNAME_LENGTH> t_dev_name;

/* unbounded string */
typedef string t_command_buffer;
```

Código 12 - Ejemplo de OMG IDL string

wstring	Representa una secuencia de wchar excepto null. Para este tipo, aplican las mismas condiciones que para string.

fixed	Representa un número decimal en punto fijo con hasta 31 dígitos significativos. Este tipo deberá ser mapeado al tipo de punto fijo nativo provisto por el lenguaje. Si este no existiese, el mapeo de lenguaje deberá proveer uno.
	
	Si se usan distintos lenguajes para desarrollar clientes y objetos, se deberán tener en cuenta las posibles diferencias en cuanto a manejo de overflow, redondeo y precisión.
	
	La cantidad de dígitos significativos se define de la siguiente forma:
	
	`typedef fixed<2> t_price;`
	`typedef fixed<4> t_stock_value;`
	Código 13 - Ejemplo de OMG IDL fixed

Tabla 27 - Tipos de dato template de OMG IDL

Por último, OMG IDL también provee facilidades para definir vectores y matrices multidimensionales. Una definición de este tipo consta de un tipo base para los elementos que componen el vector o la matriz, un identificador y una especificación del tamaño para cada una de las dimensiones. Por ejemplo:

```
/* Vector de 10 posiciones */
typedef float t_vector[10];

/* Matriz de 10x10 */
typedef float t_matrix[10][10];

/* Lista de 256 posiciones de nombres de dispositivos */
const unsigned short DEVNAME_LENGTH=256;
typedef string<DEVNAME_LENGTH> t_dev_name;

const unsigned short DEVLIST_SIZE=256;
typedef t_dev_name t_dev_list[DEVLIST_SIZE];
```

Código 14 - Ejemplo de vectores y matrices en OMG IDL

La implementación de estos vectores y matrices depende del mapeo de lenguaje que se esté usando. Por este motivo, el manejo de índices depende también del lenguaje. No se recomienda pasar índices y contadores sobre vectores o matrices como parámetros de operaciones (distintos lenguajes trabajan con índices basados en 0 o en 1).

- **Declaración de interfases**

La declaración de una interfase consta de dos partes principales. La primera de ellas es el encabezado, formado por un modificador opcional que indica si se trata de una interfase abstracta, un identificador precedido por la palabra clave “interface” y por último, una especificación opcional de herencia. La segunda parte es el cuerpo, compuesto por múltiples declaraciones de constantes, tipos, excepciones, atributos y operaciones.

```
module Devices {
    const unsigned long DEVNAME_LENGTH = 256;
    typedef string<DEVNAME_LENGTH> t_dev_name;

    abstract interface Device {
        ... /* operaciones, atributos, etc */
    };

    interface Motor : Device {
        ... /* operaciones, atributos, etc */
    };

    interface ACMotor : Motor {
        ... /* operaciones, atributos, etc */
    };

    interface DCMotor : Motor {
        ... /* operaciones, atributos, etc */
    };
};
```

Código 15 - Ejemplo de interfases en OMG IDL

El identificador de la interfase puede ser usado como cualquier otro tipo de dato o sea, en parámetros de operaciones o en miembros de estructuras pero con semántica distinta. Un elemento, cuyo tipo es una interfase, representa una referencia a un objeto que soporta dicha interfase (salvo que sea abstracta).

Las interfases abstractas son aquellas para las cuales no va a existir ninguna instancia en forma directa pero pueden existir en forma indirecta. O sea, pueden existir instancias de objetos que implementen interfases que soportan la interfase abstracta (forma indirecta). Sirven para aplicar polimorfismo y también, en aquellos casos en los que se desconoce en tiempo de compilación si un objeto va a ser pasado por valor o por referencia.

En las secciones anteriores, se presentaron las declaraciones de constantes y tipos. Para poder completar el cuerpo de una interfase y así, la declaración de la misma, faltarían los siguientes elementos:

Declaración	Descripción
Operaciones	La declaración de operaciones o métodos en OMG IDL es similar a la declaración de funciones en C. Está formada por los siguientes elementos: • **Atributos:** indica la semántica que debería proveer la ORB cuando la operación sea invocada. Si se incluye la palabra clave "oneway", se trabaja con semántica de mejor esfuerzo sin garantía de entrega. • **Valor de retorno:** el tipo de dato que devuelve la operación. Si la operación no devuelve ningún valor, esto se indica mediante la palabra clave "void". • **Identificador:** nombre de la operación dentro del scope de nombres en el cual se la está definiendo. • **Lista de parámetros**: es una lista separada por ",". Cada parámetro incluye un atributo direccional, un tipo de dato y un nombre. El

atributo direccional indica a la ORB la dirección en la cual se debe transmitir el parámetro. Los posibles valores son:

- in, del cliente al objeto
- out, del objeto al cliente
- inout, en ambas direcciones

- **Excepciones:** es una lista separada por "," de las excepciones definidas por el programador que puede generar la operación al ser invocada. Aparte, hay una lista implícita de excepciones que pueden ser generadas directamente por la ORB.

- **Contexto:** especifica que elementos del contexto del cliente tienen que estar disponibles para la implementación del objeto en el momento de la invocación. La ORB se encarga de transmitirlos.

A continuación se incluyen algunas declaraciones de ejemplo:

```
module Devices {
    typedef enum method { method_1,  method_2} t_method;

    const unsigned long DEVNAME_LENGTH = 256;
    typedef string<DEVNAME_LENGTH> t_dev_name;

    abstract interface Device {
        long Initialize(in unsigned long id);
        void GetId(out unsigned long id);
        void Terminate();
    };

    interface Motor : Device {
        void Start() raises(MotorStartException);
        void Stop();
        float GetCurrentAcceleration(in t_method m);
        float GetCurrentSpeed(in t_method m);
    };
};
```

Código 16 - Ejemplo de operaciones en OMG IDL

Todas las operaciones de Device se encuentran disponibles en Motor debido a que esta última hereda de la primer interfase.

Atributos

Una interfase puede tener atributos aparte de operaciones. Una definición de este tipo de elementos es lógicamente equivalente a declarar un par de funciones proyectoras (accesors en terminología de objetos): una para leer el valor del atributo y otra para escribirlo.

```
module Devices {
    typedef enum method { method_1,  method_2} t_method;

    const unsigned long DEVNAME_LENGTH = 256;
    typedef string<DEVNAME_LENGTH> t_dev_name;

    abstract interface Device {
        attribute unsigned long firmware_rev_major;
        attribute unsigned long firmware_rev_minor;
        attribute boolean enabled;

        long Initialize(in unsigned long id);
        void GetId(out unsigned long id);
        void Terminate();
    };
```

```
    interface Motor : Device {
        attribute float max_acceleration;
        attribute float max_speed;

        void Start() raises(MotorStartException);
        void Stop();
        float GetCurrentAcceleration(in t_method m);
        float GetCurrentSpeed(in t_method m);
    };
};
```

Código 17 - Ejemplo de atributos en OMG IDL

Todos los atributos de Device se encuentran disponibles en Motor debido a que esta última hereda de la primer interfase.

Excepciones

Son estructuras de datos, similares a los structs, que pueden ser devueltas por un objeto para indicar que ha ocurrido una condición excepcional mientras se atendía el request. La declaración está formada por la palabra clave "exception", un identificador y una lista de miembros entre "{" y "}". Por ejemplo:

```
module Devices {

    ...

    const unsigned long EX_REASON_LENGTH = 1024;
    typedef string<EX_REASON_LENGTH> t_ex_reason;

    exception MotorStartException {
        unsigned long reason_code;
        t_ex_reason reason_desc;
    };

    ...
};
```

Código 18 - Ejemplo de excepciones en OMG IDL

Tabla 28 - Elementos de una interfase en OMG IDL

3.4.6 Mecanismo de invocación estático

La arquitectura general del middleware CORBA provee a los agentes clientes mecanismos estáticos y dinámicos de solicitud de servicios. O sea, se puede llamar a las operaciones provistas por los objetos CORBA en forma estática o bien, en forma dinámica.

Para usar el mecanismo estático, hay que compilar una definición en OMG IDL mediante el compilador incluido con la ORB. Esto genera el código del stub del cliente y del skeleton del objeto, en un lenguaje particular dependiendo del mapeo que se haya seleccionado. Luego, se generan el programa cliente y el programa servidor (el cual contiene la implementación del objeto) con el compilador y el enlazador del lenguaje en cuestión. El código OMG IDL no es necesario en tiempo de ejecución, sólo en tiempo de compilación.

El proceso completo de compilación involucra los siguientes pasos:

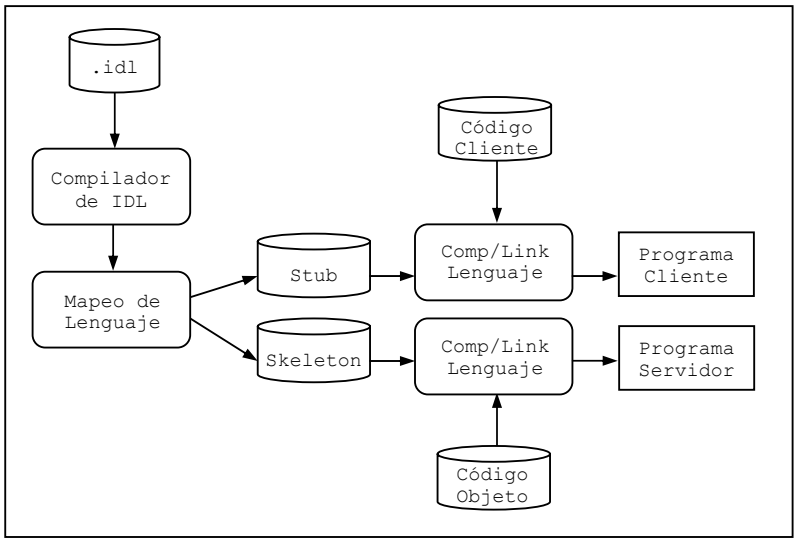


Ilustración 22 - Mecanismo de invocación estático (Generación)

El stub está formado por una serie de rutinas que representan las operaciones de una interfase, originalmente definidas en OMG IDL, en una forma consistente con el lenguaje y el mapeo que se esté usando para la implementación. Estas rutinas hacen llamadas a la ORB, mediante interfases privadas, que permiten construir y transmitir las solicitudes una vez ubicado el objeto CORBA destinatario de las mismas.

Por otro lado, es necesaria una interfase ente la ORB (específicamente, el POA) y la implementación de los métodos del objeto CORBA en el servant. El skeleton supone que la implementación de un cierto método en un lenguaje determinado, dada su definición en OMG IDL, va a tener una forma particular. Esta forma constituye una interfase up-call, que debe ser respetada por las rutinas de implementación. Así, la ORB, mediante su POA, puede llamar al skeleton y este, a su vez, puede ubicar e invocar dichas rutinas.

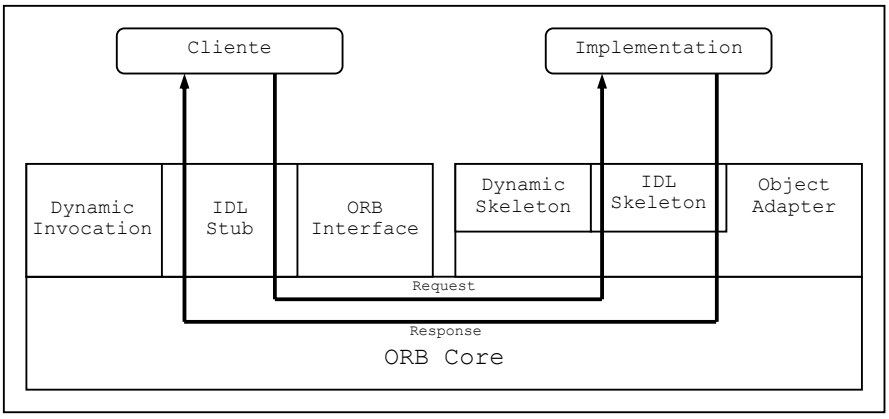


Ilustración 23 - Mecanismo de invocación completamente estático (Funcionamiento)

La utilización de un stub estático no necesariamente implica que en el servidor se vaya a usar un skeleton estático. La implementación del objeto puede usar el mecanismo dinámico para interactuar con la ORB. De igual forma, un cliente puede llamar a un objeto con skeleton estático usando el mecanismo dinámico para generar la solicitud, en lugar del stub. Cualquier combinación es válida y funciona en forma correcta.

A continuación se incluye un ejemplo de definición en OMG IDL y los prototipos de las rutinas del stub y del skeleton generados con el mapeo para el lenguaje C de ORBit versión 0.5.13 [ORBit 04], implementación CORBA de código abierto usada por el proyecto GNOME para su arquitectura de componentes bajo Linux.

```
module Devices {
	typedef enum method { method_1,  method_2} t_method;

	interface Motor {
		attribute float max_acceleration;
		attribute float max_speed;

		void Start() raises(MotorStartException);
		void Stop();
		float GetCurrentAcceleration(in t_method m);
		float GetCurrentSpeed(in t_method m);
	};
};
```

Código 19 - Stub y skeleton en C (definición en OMG IDL)

```
CORBA_float Devices_Motor__get_max_acceleration(
	Devices_Motor _obj,
	CORBA_Environment* ev);

void Devices_Motor__set_max_acceleration(
	Devices_Motor _obj,
	const CORBA_float value,
	CORBA_Environment* ev);

CORBA_float Devices_Motor__get_max_speed(
	Devices_Motor _obj,
	CORBA_Environment* ev);

void Devices_Motor__set_max_speed(
	Devices_Motor _obj,
	const CORBA_float value,
	CORBA_Environment* ev);

void Devices_Motor_Start(
	Devices_Motor _obj,
	CORBA_Environment* ev);

void Devices_Motor_Stop(
	Devices_Motor _obj,
	CORBA_Environment* ev);

CORBA_float Devices_Motor_GetCurrentAcceleration(
	Devices_Motor _obj,
	const t_method m,
	CORBA_Environment* ev);

CORBA_float Devices_Motor_GetCurrentSpeed(
	Devices_Motor _obj,
	const t_method m,
	CORBA_Environment* ev);
```

Código 20 - Stub generado con el compilador de ORBit versión 0.5.13, mapeo para ANSI C.

```
static CORBA_float impl_Devices_Motor__get_max_acceleration(
       impl_POA_Devices_Motor* servant,
       CORBA_Environment* ev);

static void impl_Devices_Motor__set_max_acceleration(
       impl_POA_Devices_Motor* servant,
       CORBA_float value,
       CORBA_Environment* ev);

static CORBA_float impl_Devices_Motor__get_max_speed(
       impl_POA_Devices_Motor* servant,
       CORBA_Environment* ev);

static void impl_Devices_Motor__set_max_speed(
       impl_POA_Devices_Motor* servant,
       CORBA_float value,
       CORBA_Environment* ev);

static void impl_Devices_Motor_Start(
       impl_POA_Devices_Motor* servant,
       CORBA_Environment* ev);

static void impl_Devices_Motor_Stop(
       impl_POA_Devices_Motor* servant,
       CORBA_Environment* ev);

static CORBA_float Devices_Motor_GetCurrentAcceleration(
       impl_POA_Devices_Motor* servant,
       t_method m,
       CORBA_Environment* ev);

static CORBA_float Devices_Motor_GetCurrentSpeed(
      impl_POA_Devices_Motor* servant,
      t_method m,
      CORBA_Environment* ev);
```

Código 21 - Interfase up-call de skeleton, generado con el compilador de ORBit versión 0.5.13, mapeo para ANSI C.

En los dos ejemplos anteriores, se puede observar como el compilador genera los proyectores necesarios para cada atributo (este tipo de funciones también se llaman accessors). Ahora, usando la misma definición OMG IDL pero con la ORB OpenORB versión 1.3.1 [OpenORB 04] y un mapeo de lenguaje para Java, se obtienen los siguientes resultados:

```
package Devices;

public interface MotorOperations {
       public float max_acceleration();
       public void max_acceleration(float value);

       public float max_speed();
       public void max_speed(float value);

       public void Start() throws Devices.MotorStartException;
       public void Stop();

       public float GetCurrentAcceleration(Devices.method m);
       public float GetCurrentSpeed(Devices.method m);
}

public interface Motor
       extends MotorOperations,
       org.omg.CORBA.Object,
       org.omg.CORBA.portable.IDLEntity {
       ...
}

public class _MotorStub
       extends org.omg.CORBA.portable.ObjectImpl
       implements Motor {

       ...
```

```
	public float max_acceleration() {...}
	public void max_acceleration(float value) {...}

	public float max_speed() {...}
	public void max_speed(float value) {...}

	public void Start() throws Devices.MotorStartException {...}
	public void Stop() {...}

	public float GetCurrentAcceleration(Devices.method m) {...}
	public float GetCurrentSpeed(Devices.method m) {...}
}
```

Código 22 - Stub generado con el compilador de OpenORB versión 1.3.1, mapeo para Java.

```
package Devices;

public interface MotorOperations {
	public float max_acceleration();
	public void max_acceleration(float value);

	public float max_speed();
	public void max_speed(float value);

	public void Start() throws Devices.MotorStartException;
	public void Stop();

	public float GetCurrentAcceleration(Devices.method m);
	public float GetCurrentSpeed(Devices.method m);
}

// Esta clase es la que efectivamente implementa el skeleton
public abstract class MotorPOA
	extends org.omg.PortableServer.Servant
	implements MotorOperations,
	org.omg.CORBA.portable.InvokeHandler {

	public final org.omg.CORBA.portable.OutputStream _invoke(
		final String opName,
		final org.omg.CORBA.portable.InputStream _is,
		final org.omg.CORBA.portable.ResponseHandler handler) {...}

	private org.omg.CORBA.portable.OutputStream _invoke__get_max_acceleration(
		final org.omg.CORBA.portable.InputStream _is,
		final org.omg.CORBA.portable.ResponseHandler handler) {...}

	private org.omg.CORBA.portable.OutputStream _invoke__set_max_acceleration(
		final org.omg.CORBA.portable.InputStream _is,
		final org.omg.CORBA.portable.ResponseHandler handler) {...}

	private org.omg.CORBA.portable.OutputStream _invoke__get_max_speed(
		final org.omg.CORBA.portable.InputStream _is,
		final org.omg.CORBA.portable.ResponseHandler handler) {...}

	private org.omg.CORBA.portable.OutputStream _invoke__set_max_speed(
		final org.omg.CORBA.portable.InputStream _is,
		final org.omg.CORBA.portable.ResponseHandler handler) {...}

	private org.omg.CORBA.portable.OutputStream _invoke_Start(
		final org.omg.CORBA.portable.InputStream _is,
		final org.omg.CORBA.portable.ResponseHandler handler) {...}

	private org.omg.CORBA.portable.OutputStream _invoke_Stop(
		final org.omg.CORBA.portable.InputStream _is,
		final org.omg.CORBA.portable.ResponseHandler handler) {...}

	private org.omg.CORBA.portable.OutputStream _invoke_GetCurrentAcceleration(
		final org.omg.CORBA.portable.InputStream _is,
		final org.omg.CORBA.portable.ResponseHandler handler) {...}

	private org.omg.CORBA.portable.OutputStream _invoke_GetCurrentSpeed(
		final org.omg.CORBA.portable.InputStream _is,
		final org.omg.CORBA.portable.ResponseHandler handler) {...}
```

```
}

// Esta clase implementa el objeto que soporta la interfase Motor
public class MotorImpl
      extends MotorPOA {

      public float max_acceleration() {...}
      public void max_acceleration(float value) {...}

      public float max_speed() {...}
      public void max_speed(float value) {...}

      public void Start() throws Devices.MotorStartException {...}
      public void Stop() {...}

      public float GetCurrentAcceleration(Devices.method m) {...}
      public float GetCurrentSpeed(Devices.method m) {...}
}
```

Código 23 – Skeleton generado con el compilador de OpenORB versión 1.3.1, mapeo para Java.

3.4.7 Mecanismo de invocación dinámico

El mecanismo de invocación dinámico puede dividirse en dos partes. Por un lado, las interfases usadas por los clientes (Dynamic Invocation Interface o DII) y por otro, las usadas por los servidores para llamar a las implementaciones de objetos (Dynamic Skeleton Interface o DSI).

La invocación dinámica principalmente sirve para resolver problemas de interoperabilidad. Por ejemplo, se desconoce en tiempo de compilación el tipo del objeto que se desea invocar por lo cual, no es posible generar un stub específico para ese tipo.

- **Dynamic Invocation Interface**

  Esta interfase permite a los clientes crear y enviar en forma dinámica requests a los objetos. Un cliente, usando esta interfase, obtiene exactamente la misma semántica que obtendría usando el stub estático generado por el compilador de OMG IDL.

  DII es soportada directamente por el núcleo de la ORB. Se incluye un método que permite crear un pseudo-objeto para encapsular todos los componentes de una solicitud. Estos son un identificador de operación, un objeto de destino para la solicitud, cero o más parámetros y por último, un contexto de invocación opcional. El método en cuestión tiene la siguiente declaración en OMG IDL:

```
void create_request(
      in Context ctx, in            // Contexto de invocación opcional
      in Identifier operation,      // Nombre de la operación
      in NVList arg_list,           // Lista de parámetros (pares nombre-valor)
      inout NamedValue result, // Resultado de la operación
      out Request request,          // Nueva instancia del pseudo-objeto Request
      in Flags req_flags            // Flags de invocación
);
```

Código 24 - Definición en OMG IDL del método create_request

El método create_request crea la solicitud, una instancia del pseudo-objeto Request, pero no la envía al servidor correspondiente. Para hacer esto, hay que usar alguno de los métodos de Request. Esta es la definición en OMG IDL de dicho objeto:

```
module CORBA {
    native OpaqueValue;

    interface Request {
        void add_arg(
            in Identifier name,
            in TypeCode arg_type,
            in OpaqueValue value,
            in long len,
            in Flags arg_flags
        );

        void invoke(
            in Flags invoke_flags
        );

        void delete();

        void send(
            in Flags invoke_flags
        );

        void get_response() raises (WrongTransaction);

        boolean poll_response();

        Object sendp();

        void prepare(
            in Object p
        );

        void sendc(
            in Object handler
        );
    };
};
```

Código 25 - Definición en OMG IDL del pseudo-objeto Request

El método invoke permite completar la invocación en forma sincrónica. O sea, se bloquea hasta tanto el cliente no recibe el mensaje de respuesta del servidor. Por el contrario, los métodos send, sendc y sendp permiten invocaciones en forma asincrónica ya que no se bloquean esperando la respuesta del servidor.

Para determinar cuando se completó una solicitud, en una invocación asincrónica, se pueden usar los métodos poll_response y get_response. El primero permite determinar si se completo o no la solicitud mientras que el segundo, obtiene los resultados de la solicitud y se bloquea si estos aún no están disponibles.

- **Dynamic Skeleton Interface**

Esta interfase permite el manejo dinámico de invocaciones a objetos. Es una forma de entregar los requests que llegan a la ORB a una implementación de un objeto cuyo tipo no es conocido en tiempo de compilación. Permite reemplazar al skeleton estático del objeto, generado por el compilador de OMG IDL, manteniendo la misma semántica en las invocaciones.

La idea principal detrás de esta interfase es implementar todos los requests a un objeto en particular, haciendo que la ORB llame, a través del POA, a una única función o rutina up-call, llamada Dynamic Implementation Routine (DIR). Esta rutina recibe de la ORB el objeto que se desea invocar, un identificador de la operación y todos los parámetros de la misma. Estos datos van encapsulados en el pseudo-objeto ServerRequest, análogo al Request, usado en DII:

```
module CORBA {
	...

	interface ServerRequest {
		readonly attribute Identifier operation;

		void arguments(
			inout NVList nv
		);

		Context ctx();

		void set_result(
			in Any val
		);

		void set_exception(
			in Any val
		);
	};
};
```

Código 26 - Definición en OMG IDL del pseudo-objeto ServerRequest

El POA despacha una invocación a una implementación de objeto basada en DSI, pasando una instancia del pseudo-objeto ServerRequest a la DIR correspondiente. El objeto sobre el que se desea invocar la operación es provisto directamente por el POA y su mapeo de lenguaje.

DSI es soportado enteramente a través del POA. Este último está diseñado para permitir conectar servants con skeletons estáticos, específicos para cierto tipo de objeto y generados con el compilador de OMG IDL o bien, con skeletons dinámicos. Este último tipo de servants se llama DSI servants.

El mapeo de estos DSI servants es especifico para cada lenguaje. Se requiere que cada mapeo defina un conjunto de interfases up-call, que serán llamadas únicamente por el POA. Estas interfases son dos funciones que deben implementar los DSI servants:

- Una función o rutina que tome una instancia de CORBA::ServerRequest del POA y ejecute el procesamiento necesario para completar el request. Vendría a ser la DIR del DSI servant (que puede encarnar múltiples implementaciones de objetos).

- Una función o rutina que devuelva el identificador (Interface Repository Id) de la interfase más derivada soportada por el objeto destino de la solicitud. Esto es necesario porque un servant DSI puede encarnar múltiples implementaciones de objetos CORBA y no necesariamente todos soporten la misma interfase IDL como la más derivada.

3.4.8 Capa de transporte: IIOP

El protocolo de sesión usado por CORBA para las comunicaciones entre las instancias de la ORB, es un protocolo abstracto, diseñado para correr sobre cualquier capa de transporte que cumpla con ciertas condiciones. Para implementarlo, es necesario mapearlo sobre el servicio de transporte que corresponda. El mapeo de este protocolo abstracto sobre TCP/IP recibe el nombre de Internet Inter-ORB Protocol (IIOP). Los mapeos específicos para otros ambientes reciben el nombre de Environment-Specific Inter-ORB Protocol (ESIOP).

En CORBA, las entidades capaces de aceptar solicitudes para objetos o proveer información sobre la ubicación de los mismos son llamados servidores. Estas entidades publican referencias a objetos en forma de Interoperable Object References (IORs) los cuales, contienen la dirección TCP/IP en la cual está esperando conexiones el servidor correspondiente. Así, se maneja en forma transparente la ubicación de los objetos.

Las direcciones TCP/IP están formadas normalmente por la dirección IP o el nombre de la máquina donde corre el servidor y el número de puerto TCP en el cual espera conexiones. Esta información, llamada perfil IIOP, se incluye dentro del IOR en una estructura llamada ProfileBody, formada por los siguientes campos:

```
module IIOP {
    ...
    struct Version {
        octet major;
        octet minor;
    };

    struct ProfileBody {
        Version iiop_version;
        string host;
        unsigned short port;
        sequence<octet> object_key;
        sequence<IOP::TaggedComponent> components;
    };
    ...
};
```

Código 27 - Estructura de un perfil IIOP en OMG IDL

Campo	Descripción
iiop_version	Versión del protocolo que el agente servidor, corriendo en la dirección especificada por los campos host y port, está preparado para recibir. El posible valor del campo major es 1 mientras que para minor, son válidos 0, 1 y 2. En esta sección se está describiendo la versión 1.1.
host	El nombre completo de la máquina (incluyendo el dominio) o la dirección IP en notación de números y puntos.
port	Número de puerto TCP en el cual el agente que publico el IOR está esperando conexiones de clientes.
object_key	Identificador del objeto para el cual se va a enviar el mensaje GIOP/IIOP. Cuando un agente servidor genera un nuevo identificador, debe ser capaz posteriormente de recuperar el objeto, en forma no ambigua, a partir del mismo.
components	Secuencia de estructuras TaggedComponent que contienen información adicional que puede ser usada al invocar el objeto descrito por el perfil IIOP actual.

Tabla 29 - Campos de un perfil IIOP

El perfil IIOP incluye el nombre de la máquina y el puerto donde corre el servidor que implementa el objeto identificado por el campo object_key. Cuando un cliente necesita alguno de los servicios ofrecidos por dicho objeto, la instancia de la ORB que corre en el cliente debe

recuperar el perfil IIOP del IOR emitido por el servidor. De esta forma, obtiene una dirección TCP/IP a la cual debe conectarse.

Una vez aceptada la conexión por el servidor, el cliente puede enviarle mensajes directamente escribiendo en el socket TCP/IP. De igual forma, el servidor puede enviarle un mensaje de respuesta al cliente simplemente escribiendo en el socket.

3.4.9 Capa de sesión: GIOP

En las primeras especificaciones de CORBA, el OMG había definido la funcionalidad de la ORB y todas sus interfases. Sin embargo, no se definió un protocolo estándar para la comunicación entre las instancias de la ORB que corren en cada máquina. Cada implementador desarrollaba sus propios protocolos. Así, la interoperabilidad entre ORBs de distintos fabricantes no era posible salvo que se construyesen bridges específicos para cada combinación.

En especificaciones siguientes y para garantizar la interoperabilidad, la OMG definió el protocolo General Inter-ORB Protocol (GIOP) que cumple funciones de la capa de sesión. El propósito de esta capa es proveer los medios necesarios para que las entidades de la capa superior puedan organizar su diálogo, sincronizarlo y administrar el intercambio de datos entre ellas [ISO7498]. Justamente, GIOP permite organizar el diálogo entre las instancias de la ORB que corren en cada máquina.

En esta sección se describe la versión 1.1 del protocolo que es la que implementa la ORB ORBit versión 0.5.13 [ORBit 04].

Los objetivos propuestos por el OMG para el diseño de GIOP son:

- **Amplia disponibilidad**: el diseño está basado en el mecanismo de transporte más ampliamente usado que es TCP/IP si bien, podría llegar a ser implementado sobre otras capas de transporte.

- **Simplicidad**: se trató de mantener el diseño lo más simple posible ya que la simplicidad es clave para tener implementaciones independientes compatibles.

- **Escalabilidad**: el diseño debería soportar un número arbitrariamente grande de ORBs y redes de ORBs con bridges.

- **Bajo costo**: agregar el protocolo a implementaciones de ORBs existentes no debería demandar demasiado trabajo. De igual forma, su funcionamiento no debería tener un impacto elevado en el rendimiento de las ORBs.

- **Generalidad**: los mensajes tienen que estar diseñados para funcionar sobre cualquier capa de transporte, que cumpla con ciertas condiciones.

- **Neutralidad**: el diseño no debe asumir prácticamente nada sobre la arquitectura de los agentes que lo van a soportar.

GIOP está pensado para ser implementado sobre una amplia variedad de protocolos de transporte. La definición de este protocolo de sesión requiere el cumplimiento de ciertas condiciones por parte del transporte subyacente:

- **Orientado a conexión**: la capa de transporte tiene que ser orientada a conexión. Cada conexión es usada por GIOP para definir el alcance de los identificadores de los requests.

  La unicidad de estos identificadores está garantizada para una conexión, durante el tiempo de vida de la misma. Esto permite a GIOP solapamiento de solicitudes e invocación asincrónica de operaciones.

- **Confiable**: la capa de transporte garantiza que los bytes son recibidos en el orden en que fueron enviados, sin repeticiones. También debe existir algún tipo de confirmación positiva de entrega (acknowledgement).

- **Stream de bytes**: la capa de transporte tiene que ser vista como un stream de bytes. O sea, no hay limitaciones arbitrarias al tamaño de los mensajes ni se requiere fragmentación ni alineamientos de los mismos.

- **Notificación de pérdida de conexión**: si el proceso remoto aborta, se da una falla en la máquina en la que corre o bien, se pierde conectividad de red, el proceso propietario de la conexión (el cliente) debería ser notificado.

- **Manejo de conexiones**: el modelo usado por la capa de transporte para establecer conexiones puede ser mapeado al modelo usado por TCP/IP.

GIOP utiliza las conexiones en forma asimétrica definiendo dos roles distintos para los agentes que participan en las mismas: cliente y servidor. Específicamente, un servidor publica una dirección de red en una referencia o Interoperable Object Reference (IOR). Luego, el cliente usa dicha dirección para establecer una conexión IIOP o ESIOP.

Los agentes servidores tienen que ser capaces de escuchar por conexiones y aceptarlas o rechazarlas. Los clientes son los únicos que pueden establecer conexiones. Luego, cualquiera de las dos partes puede cerrarla.

Uno de los objetivos del OMG al diseñar GIOP fue mantenerlo lo más simple posible. Por este motivo, sólo se definen 7 mensajes, relativamente simples, que soportan la totalidad de la funcionalidad ofrecida por CORBA. Por otro lado, la asimetría en el manejo de conexiones se mantiene para los mensajes. O sea, ciertos mensajes sólo pueden ser emitidos por los clientes mientras que otros, sólo pueden ser emitidos por los servidores.

Mensaje	Origen
Request	Cliente
Reply	Servidor
CancelRequest	Cliente
LocateRequest	Cliente
LocateReply	Servidor
CloseConnection	Servidor
MessageError	Ambos

Tabla 30 - Mensajes definidos por GIOP

En GIOP, todos los mensajes responden al mismo formato. En primer lugar, se incluye un encabezado general del protocolo. Luego, un encabezado particular para cada tipo de mensaje y por último, información adicional si es que el mensaje lo requiere. Este sería el encabezado general GIOP en OMG IDL:

```
module GIOP {
    ...
    enum MsgType {
        Request,
        Reply,
        CancelRequest,
        LocateRequest,
        LocateReply,
        CloseConnection,
        MessageError,
        Fragment
    };

    struct Version {
        octet major;
        octet minor;
    };

    struct MessageHeader {
        char magic [4];
        Version GIOP_version;
        octet flags;
        octet message_type;
        unsigned long message_size;
    };
    ...
};
```

Código 28 - Encabezado general de mensajes GIOP en OMG IDL

Campo	Descripción
magic	Identificador de los mensajes GIOP. El valor de este campo siempre tiene que ser los cuatro caracteres "GIOP" codificados en ISO Latin-1 8859.1.
GIOP_version	Contiene el número de versión del protocolo GIOP que se está usando en el mensaje. El posible valor del campo major es 1 mientras que para minor, son válidos 0, 1 y 2. Si un servidor dice que soporta GIOP 1.n y recibe un mensaje con versión 1.m con m > n debe devolver un error. En cualquier otro caso, debería ser capaz de procesarlo.
Flags	 *Ilustración 24 - Bits del campo flags del encabezado GIOP* • **BO (Byte Ordering):** el orden de bytes que se va a usar en el resto del mensaje. El bit debe valer 0 para big-endian y 1 para little-endian.

	• **MF (More Fragments):** indica si siguen o no más fragmentos del mensaje. Un valor de 0 indica que es el último fragmento y 1 indica lo contrario.
message_type	Tipo del mensaje GIOP. Los posibles valores están dados por el enum MsgType, comenzando con 0 para el tipo Request.
message_size	Número de bytes en el mensaje sin contar el encabezado general de mensajes GIOP. El valor se codifica con el byte order especificado por el bit BO del campo flags.

Tabla 31 – Campos del encabezado general de mensajes GIOP

El primer tipo de mensaje que se va a describir es el Request. Este permite codificar invocaciones, generadas por los clientes y enviadas a los servidores. Un mensaje de este tipo está formado por un encabezado general GIOP, un encabezado especifico del tipo de mensaje y un cuerpo, formado por los parámetros in e inout de la operación que se esté invocando (codificados de acuerdo a las reglas definidas por la capa de presentación de CORBA). Esta sería la definición del encabezado especifico de Request:

```
module GIOP {
    ...
    struct RequestHeader {
        IOP::ServiceContextList service_context;
        unsigned long request_id;
        boolean response_expected;
        octet reserved[3];
        sequence<octet> object_key;
        string operation;
        CORBA::OctetSeq requesting_principal;
    };
    ...
};
```

Código 29 - Encabezado de mensajes GIOP (Request) en OMG IDL

Campo	Descripción
service_context	Contiene el contexto de invocación del cliente que se envía al servidor. El contexto se puede pensar como una lista de pares nombre-valor, configurados por el cliente y leídos por el servidor. Vendrían a ser parámetros "ocultos" de la invocación a un método. OMG define algunos contextos estándar (para ciertos CORBA Services o para CORBA-RT).
request_id	Permite identificar los mensajes y asociar un Request con un Reply. El cliente es responsable de la generación de estos números y se debe asegurar que no haya valores ambiguos. Los valores pueden ser reutilizados pero de acuerdo a las reglas de manejo de conexiones que se dieron anteriormente.
response_expected	Puede ser TRUE o FALSE e indica al cliente si debe esperar un mensaje Reply del servidor una vez que envió el Request.
reserved	Tres bytes reservados para uso futuro. Deben estar siempre en 0.

object_key	Identificador del objeto destino de la invocación. Este valor se obtiene del IOR y sólo es relevante para el servidor. El cliente no lo interpreta ni lo modifica.
operation	Nombre del método que se está invocando. Es directamente el mismo nombre que se le dio en la definición OMG IDL, calificado en el scope de la interfase a la cual pertenece. Para el caso de los proyectores o accessors de atributos se antepone "_get_" o "_set_" al nombre del atributo.
requesting_principal	No se usa más en las implementaciones actuales de CORBA. En las nuevas versiones de GIOP (1.2 y 1.3) el campo ya ha sido eliminado del encabezado.

Tabla 32 - Campos del encabezado de mensajes GIOP (Request)

El servidor recibe del cliente un mensaje Request, ubica el objeto destino de la invocación mediante el POA y llama a la función que implementa la operación OMG IDL requerida. Luego, tiene que enviarle al cliente los resultados, mediante un mensaje Reply.

Este tipo de mensajes están formados por un encabezado general GIOP, un encabezado especifico del tipo de mensaje y un cuerpo, formado por los parámetros inout, parámetros out y el valor de retorno de la operación que se esté invocando. Esta sería la definición del encabezado especifico de Reply:

```
module GIOP {
    ...
    enum ReplyStatusType {
        NO_EXCEPTION,
        USER_EXCEPTION,
        SYSTEM_EXCEPTION,
        LOCATION_FORWARD
    };

    struct ReplyHeader {
        IOP::ServiceContextList service_context;
        unsigned long request_id;
        ReplyStatusType reply_status;
    };
    ...
};
```

Código 30 - Encabezado de mensajes GIOP (Reply) en OMG IDL

Campo	Descripción
service_context	Al igual que el campo service_context de Request, contiene el contexto de invocación pero esta vez, el servidor se lo pasa al cliente.
request_id	El servidor se limita a poner en este campo el valor que recibió del cliente en el mensaje Request correspondiente.
reply_status	Indica el resultado de la operación. Los posibles valores de este campo están definidos en ReplyStatusType: • **NO_EXCEPTION**: no se produjo ningún error y la solicitud se completó exitosamente. El

	cuerpo del mensaje contiene parámetros out e inout junto con el valor de retorno, si existe alguno definido para la operación. • **USER_EXCEPTION:** se produjo un error y la operación levantó una excepción definida por el usuario en OMG IDL. El cuerpo del mensaje contiene dicha excepción. • **SYSTEM_EXCEPTION:** se produjo una excepción del sistema y el cuerpo del mensaje responde al siguiente formato:

```
module GIOP {
    ...
    struct SystemExceptionReplyBody {
        string exception_id;
        unsigned long minor_code_value;
        unsigned long completion_status;
    };
    ...
};
```

Código 31 – Definición de SystemExceptionReplyBody

	Como se ve, el cuerpo contiene información sobre la excepción. • **LOCATION_FORWARD:** esto indica que el objeto se encuentra en otra máquina. En el cuerpo del mensaje se envía un IOR, con la nueva dirección de red donde ubicar el objeto. El cliente vuelve a enviar la solicitud al nuevo destino en forma transparente para el usuario.

Tabla 33 - Campos del encabezado de mensajes GIOP (Reply)

El mensaje CancelRequest puede ser enviado por un cliente a un servidor para indicarle que no está esperando más una respuesta para un mensaje Request o LocateRequest previo. En cierta forma, permite cancelar una solicitud asincrónica previa. El mensaje está formado únicamente por un encabezado general GIOP y un encabezado especifico del tipo de mensaje, que tiene el siguiente formato:

```
module GIOP {
    ...
    struct CancelRequestHeader {
        unsigned long request_id;
    };
    ...
};
```

Código 32 - Encabezado de mensajes GIOP (CancelRequest) en OMG IDL

Campo	Descripción
request_id	Identificador del mensaje Request o LocateRequest enviado previamente y para el cual se desea cancelar.

Tabla 34 - Campos del encabezado de mensajes GIOP (CancelRequest)

Otro elemento a tener en cuenta es que el mensaje CancelRequest tiene un carácter informativo únicamente. O sea, no se requiere que el servidor haga un acknowledge de la

cancelación al punto que, posteriormente, puede enviar el mensaje de respuesta correspondiente. El cliente no debería tener ninguna expectativa en lo referente a la cancelación.

El mensaje LocateRequest permite a un cliente determinar si una referencia a un objeto sigue teniendo validez. O sea, permite saber si el objeto efectivamente se encuentra en la máquina que la misma indica. Si esto no es así, permite al cliente obtener una nueva referencia que apunta a la nueva máquina donde se ubica el objeto buscado.

Este tipo de mensajes permite soportar migración dinámica de objetos. Está pensado como una optimización para los clientes. Si un objeto ya migró a otra máquina y se envía un Request a la ubicación vieja, el mensaje Reply contiene una nueva referencia con la nueva ubicación. Así, el cliente se actualiza en forma transparente pero es necesario enviar dos mensajes Request. LocateRequest reemplazaría uno de estos mensajes.

Un mensaje de este tipo está formado por un encabezado general GIOP y un encabezado especifico, que tiene el siguiente formato:

```
module GIOP {
    ...
    struct LocateRequestHeader {
        unsigned long request_id;
        sequence<octet> object_key;
    };
    ...
};
```

Código 33 - Encabezado de mensajes GIOP (LocateRequest) en OMG IDL

Campo	Descripción
request_id	Permite identificar los mensajes y asociar un LocateRequest con un LocateReply. El cliente es responsable de la generación de estos números y se debe asegurar que no haya valores ambiguos. Los valores pueden ser reutilizados pero de acuerdo a las reglas de manejo de conexiones que se dieron anteriormente.
object_key	Identificador del objeto para el cual se desea obtener información. Este valor se obtiene del IOR y sólo es relevante para el servidor. El cliente no lo interpreta ni lo modifica.

Tabla 35 - Campos del encabezado de mensajes GIOP (LocateRequest)

Un servidor contesta a un mensaje LocateRequest mediante un LocateReply, formado por un encabezado general GIOP, un encabezado especifico y un cuerpo, cuyo valor depende del resultado de la búsqueda del objeto. El formato del encabezado especifico es el siguiente:

```
module GIOP {
    ...
    enum LocateStatusType {
        UNKNOWN_OBJECT,
        OBJECT_HERE,
        OBJECT_FORWARD
    };

    struct LocateReplyHeader {
        unsigned long request_id;
        LocateStatusType locate_status;
    };
```

```
    ...
};
```

Código 34 - Encabezado de mensajes GIOP (LocateReply) en OMG IDL

Campo	Descripción
request_id	El servidor se limita a poner en este campo el valor que recibió del cliente en el mensaje LocateRequest correspondiente.
locate_status	Indica el resultado de la búsqueda del objeto. Los posibles valores de este campo están definidos en LocateStatusType: • **UNKNOWN_OBJECT:** el objeto, identificado por object_key de LocateRequest, es desconocido para el servidor. En este caso, el mensaje de respuesta no incluye un cuerpo. • **OBJECT_HERE:** el objeto, identificado por object_key de LocateRequest, reside directamente en el servidor que recibió el mensaje. En este caso, el mensaje de respuesta no incluye un cuerpo. • **OBJECT_FORWARD:** esto indica que el objeto se encuentra en otra máquina. En el cuerpo del mensaje se envía un IOR, con la nueva dirección de red donde ubicar el objeto. El cliente vuelve a enviar la solicitud al nuevo destino en forma transparente para el usuario.

Tabla 36 - Campos del encabezado de mensajes GIOP (LocateReply)

El mensaje CloseConnection puede ser enviado únicamente por un servidor a un cliente, para informarle que se dispone a cerrar la conexión de transporte subyacente y que por lo tanto, no debería esperar más respuestas a solicitudes pendientes. Los clientes saben que estas solicitudes nunca serán procesadas por lo cual, pueden reenviarlas por otras conexiones, sabiendo que se mantendrá la semántica de a lo sumo una vez.

El mensaje MessageError se usa como respuesta cada vez que se recibe un mensaje inválido desde el punto de vista de GIOP. Por ejemplo, un mensaje con formato inválido o con un número de versión no soportado en el encabezado general de GIOP generaría una respuesta de este tipo.

Por último, el mensaje Fragment se usa cuando fue necesario fragmentar un mensaje de request o response previo. O sea, se envió el primer fragmento con el tipo de mensaje correspondiente y con el bit More Fragments (MF) del campo flags del encabezado general de GIOP en 1. Los fragmentos subsiguientes son de tipo Fragment en lugar de Request, Reply, LocateRequest, etc.

3.4.10 Capa de presentación: CDR

La función de la capa de presentación es proveer los medios necesarios para la representación común de los datos a ser transferidos entre agentes [ISO7498].

En CORBA, esta función la cumple la Common Data Representation o CDR. Es la sintaxis de transferencia usada para mapear todos los tipos de datos definidos en OMG IDL a

una representación de bajo nivel (un stream de bytes) que pueda ser transferida entre agentes. Las principales características de esta sintaxis son:

- **Ordenamiento variable de bytes**

  Distintas máquinas pueden tener distinto byte order (big-endian o little-endian). En CDR, el transmisor del mensaje usa su propio byte order pero indica en el encabezado GIOP cuál está usando (campo flags, bit BO del encabezado genérico de mensajes GIOP). De esta forma, el receptor sabe si existe alguna diferencia con su propio byte order y determina fácilmente si debe aplicar o no intercambio de bytes.

- **Alineamiento de tipos primitivos**

  Los tipos primitivos de OMG IDL se alinean en sus límites naturales dentro de los mensajes GIOP. Esto quiere decir que cualquier tipo primitivo de n bytes debe comenzar en una posición dentro del stream que sea múltiplo de n. Así, se permite el manejo eficiente de streams CDR por aquellas arquitecturas que requieren alineamiento de datos en memoria (se asume que al hacer el demarshalling, directamente se copian los datos del buffer a las variables en memoria).

Tipo de Dato	Alineamiento
char	1
wchar	1
octet	1
short	2
unsigned short	2
long	4
unsigned long	4
long long	8
unsigned long long	8
float	4
double	8
long double	8
boolean	1
enum	4

Tabla 37 - Alineamiento requerido por tipos primitivos de OMG IDL

Respecto a los tipos construidos como structs, arrays y strings, no se impone ninguna restricción adicional más allá de las ya impuestas sobre sus componentes primitivos.

- **Reglas de codificación**

  CDR es una sintaxis de transferencia las cuales están formadas por dos componentes.

El primer componente consiste en una sintaxis abstracta para describir los datos que deben transmitirse. En el mundo OSI, esto se hace mediante la Abstract Syntax Notation One (ASN.1) especificada en la normas ISO 8824.1-4 / ITU-T X.680-683. En el caso de CDR en CORBA, esto se hace con el lenguaje de definición de interfases OMG IDL.

El segundo, es un conjunto de reglas de codificación que dice como pasar de los datos abstractos a un stream de bytes que pueda ser transmitido. En el mundo OSI, esto se hace mediante las Basic Encoding Rules (BER) y otras reglas más recientes como las XML Encoding Rules (XER), definidas en los estándares ISO 8825.1-4 / ITU-T X.690-693. Estas reglas trabajan con un formato Type-Length-Value (TLV) en el cual, se codifica primero un tag con el tipo de dato, luego la longitud del mismo y por último, el valor en sí [ISO8824 02] [ISO8825 02].

En CDR se trabaja con un enfoque distinto, similar al usado en External Data Representation (XDR), capa de presentación de ONC RPC [Vondrak 97]. Se asume que existe un acuerdo implícito entre el stub del cliente y el skeleton del objeto sobre el orden de la secuencia de datos que se van a intercambiar. Este acuerdo, viene dado por la definición de las interfases del objeto en OMG IDL.

Así, ambos tienen suficiente información para codificar (marshalling) y decodificar (demarshalling) los parámetros de cada operación que se puede llegar a solicitar al objeto en cuestión. Por este motivo, no se incluye información sobre el tipo de los datos y la longitud de los mismos, colocándose en el stream de bytes, directamente, el valor del dato en el byte order de la máquina que lo genera.

3.4.11 CORBA Services

Estos son los Common CORBA Services, definidos en la Object Management Architecture:

Servicio	Propósito
Collection Service	Provee una manera uniforma de crear y manipular los tipos más comunes de colecciones (pila, stack, lista enlazada, etc.) en forma genérica.
Concurrency Service	Media en el acceso a un objeto de forma tal que la consistencia del mismo no se vea comprometida si existen accesos concurrentes.
Event Service	Permite intercambiar eventos en un modelo productor / consumidor mediante requests CORBA estándar. Los objetos se transforman en productores de eventos o bien, consumidores.
Externalization Service	Define los protocolos y convenciones para grabar (externalizar) el estado del objeto en un stream de datos. De igual forma, provee lo necesario para recuperarlo (internalizar).
Licensing Service	Define las interfases necesarias para soportar la administración y manejo de licencias de software.
Life Cycle Service	Define servicios y convenciones para crear, eliminar, copiar y mover objetos.
Naming Service	Provee el mecanismo de nombres a través del cual la mayoría de los clientes ubican los objetos que pretenden invocar.

Notification Service	Extensión del Event Service. Se le agrega la transmisión de eventos como estructuras de datos, suscripción a eventos, descubrimiento, QoS y un repositorio de tipos de evento.
Persistent State Service	El servicio ofrece interfases que presentan información persistente como objetos de storage almacenados en storage homes. Estos storage homes, a su vez, son almacenados en datastores (entidades que manejan datos como una base de datos, un conjunto de archivos o un esquema en una base de datos relacional).
Property Service	Este servicio permite asociar valores con nombre con objetos (pares nombre-valor) por fuera del sistema estático normalmente usado con OMG IDL.
Query Service	Permite realizar consultas (queries) sobre colecciones de objetos.
Relationship Service	Permite representar en forma explícita entidades y las relaciones entre estas. Las entidades están representadas por objetos CORBA y el servicio permite definir relaciones y roles.
Security Service	Este servicio provee el framework de seguridad CORBA.
Time Service	Permite a un cliente obtener el tiempo actual junto a una estimación de error asociada.
Trading Object Service	Este servicio facilita la oferta y descubrimiento de instancias de objetos (servicios) de tipos particulares.
Transaction Service	Este servicio provee una interfase que combina el paradigma transaccional, esencial para desarrollar aplicaciones distribuidas confiables, y el paradigma de objetos, necesario para garantizar productividad y calidad en el proceso de desarrollo.

Tabla 38 - CORBA Services

4. Descripción del problema

4.1 Planteo del problema

Los sistemas distribuidos constituyen la arquitectura más apropiada para la construcción de sistemas de control distribuido. En cualquier arquitectura distribuida es necesario un sistema de comunicación entre las unidades de distribución. Para este tipo de sistemas computacionales, los buses de campo o buses industriales son los más adecuados [Kaiser 98]. En particular para este trabajo, se decidió usar el bus CAN por contar con una especificación estándar, por ser aplicable en un amplio rango de sistemas de control distribuido, por su mecanismo de arbitración y control de acceso al medio y por la libertad de elección de los protocolos a usar en capas superiores.

Los requerimientos generales de las aplicaciones sobre este tipo de redes o sistemas de comunicación [Thomesse 99], además de restricciones de periodicidad, jitter, demoras, despacho garantizado, etc, son:

- Seguridad, disponibilidad y dependabilidad
- Mantenibilidad y flexibilidad
- Modularidad y capacidad de evolución
- Interoperabilidad e intercambiabilidad
- Mejor rendimiento a menores costos

Cualquier capa de aplicación para el bus CAN debería estar, en la medida de lo posible, comprometida con el cumplimiento de los requerimientos anteriores.

Además de esta lista general, existen una serie de requerimientos o consideraciones para el bus CAN en particular. Básicamente, son los servicios que esta capa debería ofrecer [Etschberger 97]:

- Asignación de identificadores
- Método de comunicación peer-to-peer
- Método de intercambio de datos del proceso
- Principios de modelado de dispositivos y definición de perfiles
- Administración de la red

Como el bus CAN es la red que se va a utilizar para construir sistemas de control distribuido siguiendo una arquitectura distribuida, es necesario seleccionar alguna herramienta que facilite su construcción. Existen muchas opciones pero la más conveniente resulta ser el middleware por las siguientes características:

- Interoperabilidad
- Transparencia

- Confiabilidad y Disponibilidad
- Escalabilidad
- Abstracción

El middleware se va a usar para construir sistemas computacionales modelados a partir de procesos reales. De los múltiples tipos de middleware descriptos previamente, dado el tipo de sistemas a construir, la opción más conveniente resulta ser un Object Request Broker (ORB), dado su enfoque basado en objetos. En particular para el presente trabajo, se decidió usar CORBA.

El problema que resuelve el presente trabajo es el diseño y especificación de una nueva capa de aplicación para el bus CAN. Esta capa permite construir sobre ella sistemas de control distribuido. La nueva capa de aplicación está constituida por un middleware basado en CORBA, capaz de satisfacer todos los requerimientos que imponen las aplicaciones sobre el bus y de ofrecer todos los servicios necesarios.

4.2 Soluciones existentes

El problema planteado en este trabajo ya ha sido abordado previamente y se han publicado algunas posibles soluciones.

En primer lugar, el trabajo de Kaiser se concentra en la invocación de objetos en tiempo real sobre el bus CAN. Plantea un modelo de objetos y un mecanismo de invocación que le permite expresar los requerimientos temporales de cada solicitud de servicio. El trabajo discute aspectos teóricos y no llega a describir ninguna implementación. Sin embargo, ofrece algunos conceptos que resultan relevantes para el presente trabajo, por ejemplo, direccionamiento de objetos, y que fueron incorporados en la solución propuesta [Kaiser 98].

En segundo lugar, los trabajos de Hong y Kim plantean específicamente un diseño de middleware basado en CORBA para correr sobre el bus CAN. El trabajo tiene un enfoque más práctico que el propuesto por Kaiser pero incluye algunas ideas que son cuestionables en un sistema distribuido. Se tomaron algunos conceptos de ambos trabajos para la representación de datos y asignación del campo ARBITRATION FIELD de los frames CAN [Hong 00] [Kim 00].

Por último, en el trabajo de Lankes se plantea algo similar a los dos trabajos anteriores pero poniendo el acento en aplicaciones de tiempo real. Se describe una nueva capa de aplicación para el bus CAN constituida por un middleware basado en RT-CORBA. De este trabajo, se tomaron algunas ideas para la administración y asignación del campo ARBITRATION FIELD de los frames CAN [Lankes 03].

5. Solución propuesta

Resumen del capítulo

En este capítulo se introduce la solución propuesta al problema planteado en el capítulo anterior. Se describe el diseño de un middleware basado en CORBA, adaptado para funcionar sobre el bus CAN. También se enumeran características del prototipo construido a partir de este diseño.

El diseño se propone respetar, en la medida de lo posible, los principios y filosofía de CORBA. El objetivo detrás de esto es tratar de preservar las características que el middleware en general y CORBA en particular exhiben en otros ambientes, y transportarlas a un ambiente basado en el bus CAN.

Se pretende que esta preservación de características permita cumplir mejor que otras alternativas (como son CANopen y DeviceNet) con los requerimientos generales de un bus de campo y en particular, con los requerimientos para capas de aplicación para el bus CAN. En el capítulo 6 se presenta una evaluación del grado de cumplimiento que se alcanzó, comparando el diseño propuesto con CANopen y DeviceNet.

5.1 Descripción de la arquitectura

El modelo general de CORBA se adaptaría a un ambiente basado en el bus CAN de la siguiente forma:

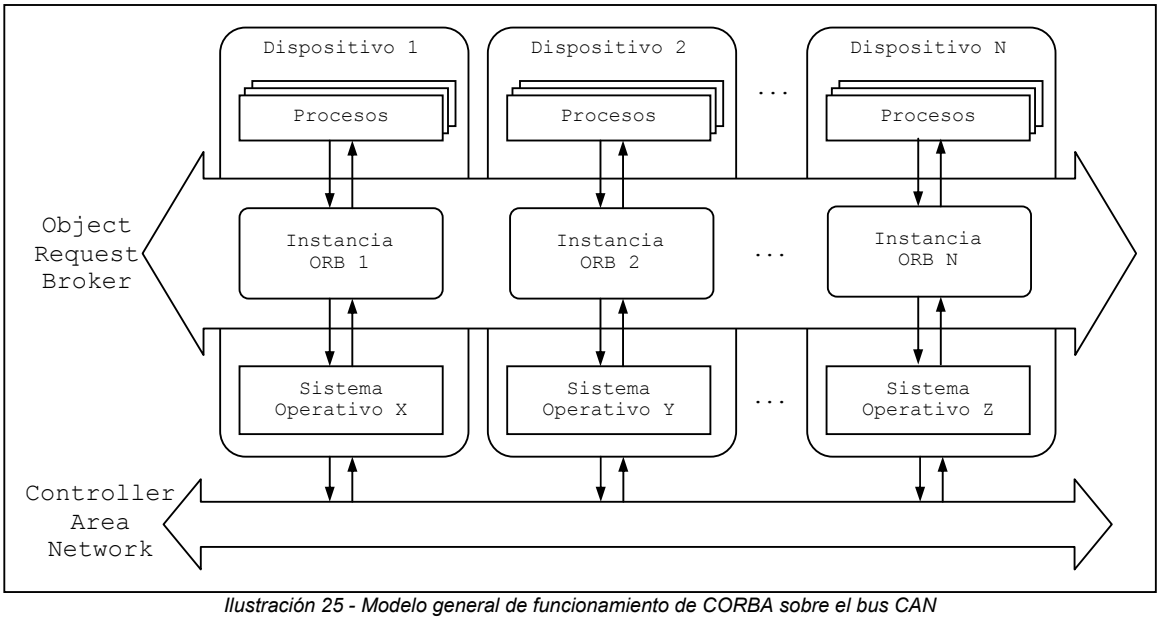


Ilustración 25 - Modelo general de funcionamiento de CORBA sobre el bus CAN

Como se puede ver en la ilustración anterior, cada dispositivo conectado al bus corre una instancia de la ORB, potencialmente sobre sistemas operativos distintos. Estas instancias se comunican entre si mediante una serie de protocolos de transporte, de sesión y de presentación (CANIOP, GIOP modificado y CCDR respectivamente) los cuales se describen en las próximas secciones del presente capítulo.

En el tipo de plataforma sobre el que correrían las instancias de la ORB los recursos computacionales son escasos. Sería complicado correr la totalidad de la funcionalidad CORBA ofrecida por una ORB estándar. Por este motivo, siguiendo las recomendaciones de CORBA para ambientes con recursos restringidos [Corba Min 02], se omitieron o modificaron algunos componentes presentes en una ORB estándar:

- **Dynamic Invocation Interface (DII)**: se removió completamente esta interfase de la ORB para CAN. Lo más probable es que los objetos se utilicen para encapsular los componentes de los dispositivos por lo cual, todas las interfases son conocidas de antemano.

- **Dynamic Skeleton Interface (DSI)**: se removió completamente esta interfase de la ORB para CAN. Al igual que en el caso anterior, lo más probable es que los objetos se utilicen para encapsular los componentes de los dispositivos por lo cual, todas las interfases son conocidas a priori.

- **Portable Object Adapter (POA)**: se restringe la capacidad de configurar el comportamiento del Object Adapter. El valor de Thread Policy, Implicit Activation Policy, Request Processing Policy y Servant Retention Policy se encuentran fijos a los valores por defecto del Root POA y no pueden ser modificados. Estos valores son

ORB_CTRL_MODEL, IMPLICIT_ACTIVATION, USE_ACTIVE_OBJECT_MAP_ONLY y RETAIN, respectivamente.

Teniendo en cuenta estos cambios, la ORB para el bus CAN quedaría conformada por los siguientes componentes:

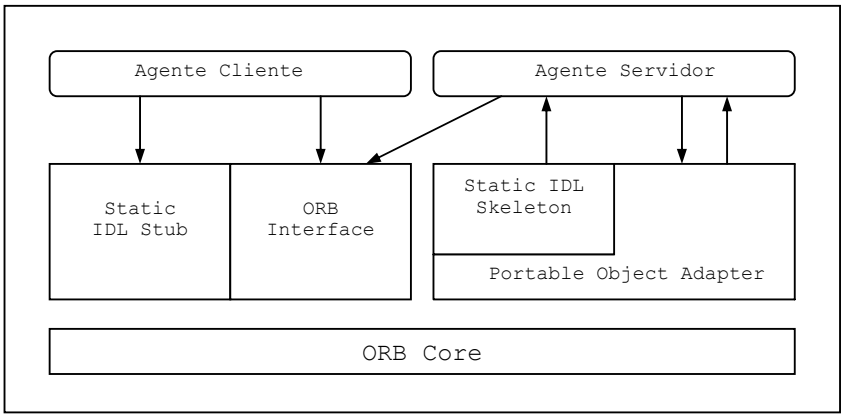


Ilustración 26 - Componentes de una ORB CORBA para CAN

- **Static IDL Stub**: corresponde al código generado a partir de la descripción de la interfase de un objeto. Se compila y se enlaza con el programa cliente, permitiendo así solicitar servicios. Es igual que en CORBA estándar.

- **ORB Interface**: es una interfase que agrupa funcionalidad genérica de la ORB como configuración, inicialización, generación y publicación de referencias para clientes, etc.

- **Static IDL Skeleton**: código generado a partir de la descripción de la interfase de un objeto. Se compila y se enlaza con el programa servidor, permitiendo llamar a la implementación del objeto correspondiente cuando llega una solicitud. Es igual que en CORBA estándar.

- **Multicast Agent**: permite la comunicación bajo un modelo master / slave o productor / consumidor basado en objetos. Este componente no existe en las implementaciones de CORBA estándar pero es necesario en un ambiente basado en el bus CAN.

- **Portable Object Adapter (POA)**: permite a los programas servidores registrar los objetos que implementan con la ORB. Cuando llega una solicitud, ubica el objeto destinatario y dispara la llamada a la operación que corresponda.

- **ORB Core**: es una abstracción que contiene el núcleo de la ORB o sea, su funcionalidad principal. Incluye los mecanismos y protocolos (de sesión y de presentación) usados para la comunicación entre las máquinas donde se distribuyen los clientes y servidores.

5.2 Capa de transporte: MCA

Previamente se enumeraron los requerimientos que debe cumplir una capa de aplicación para el bus CAN. Esta lista incluye la definición de un método de comunicación peer-to-peer, de un método productor / consumidor y por último, de un método master / slave.

Ni el modelo de comunicaciones productor / consumidor ni el master / slave son soportados por la versión original de CORBA [Corba Core 02]. El objetivo principal del modelo productor / consumidor, así como del master / slave, es hacer llegar datos o comandos en forma rápida de un nodo o dispositivo, el productor o master, a un conjunto de uno o más nodos o dispositivos, los consumidores o slaves.

CORBA constituye un ambiente orientado a objetos tradicional por lo tanto, el único mecanismo disponible de intercambio de datos es la invocación de métodos, que en forma abstracta, es una llamada sincrónica y directa, dirigida a un único objeto, que permite intercambiar datos o comandos entre este último y el cliente. Para poder aplicar en CORBA tanto el modelo productor / consumidor como el master / slave, habría que modificar este mecanismo para que permita dirigir un intercambio a múltiples destinatarios (consumidores o slaves). En fin, una invocación a un método debería poder dirigirse a múltiples objetos.

Para conseguir lo anterior, se propone incluir en la ORB para CAN el soporte para grupos de objetos. Los objetos que forman parte de un mismo grupo tienen, al menos, una interfase en común. El cliente puede llamar a algún método de alguna interfase común sobre todos los objetos del grupo en forma transparente, sin poder distinguir si se trata de una llamada normal o no. Las invocaciones a métodos sobre estos grupos se hacen mediante multicast asincrónicos.

El soporte para grupos de objetos en la ORB permite una distribución rápida y sencilla de la información, aplicando el modelo productor / consumidor o master / slave, así como un aprovechamiento de las ventajas del bus CAN, principalmente, del multicast confiable. Gracias a este último, un mensaje llega correctamente a todos los nodos o no llega a ninguno. Inclusive, permite aplicar replicación para garantizar tolerancia a fallas, aunque este punto queda fuera del alcance del presente trabajo.

A la hora de incluir esta funcionalidad en el prototipo, se contemplaron dos alternativas. La primera, se basa en la existencia de un agente centralizado que lleva control de los miembros de cada grupo propuesta por [Hong 00] [Kim 00]. La segunda propuesta, orientada a construir aplicaciones de tiempo real sobre CAN, se basa en la utilización de agentes distribuidos para el control de la membresía de los grupos [Kaiser 98].

La primer alternativa plantea la existencia de un agente centralizado, implementado como un pseudo-objeto CORBA llamado CONJOINER. Este último, se encarga de establecer un canal de invocación desde un publicador a una colección de suscriptores anónimos. Un canal de invocación podría pensarse como un canal virtual de multicast o broadcast de los publicadores a grupos de suscriptores.

El CONJOINER mantiene una base de datos de binding global en la cual, en cada entrada, se guarda un tag para identificar el canal de invocación, un identificador de la interfase OMG IDL y el valor para el ARBITRATION FIELD de los mensajes generados por el productor.

El funcionamiento de esta alternativa es el siguiente:

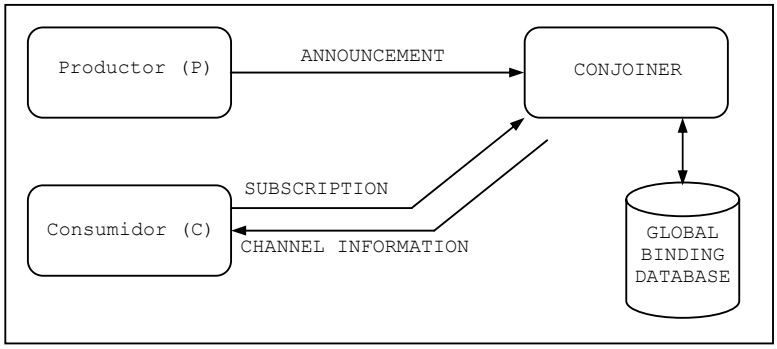


Ilustración 27 - Funcionamiento de alternativa centralizada para grupos de objetos en CORBA sobre CAN

Un nodo productor P desea registrarse como tal con el CONJOINER. Para esto, busca algún valor de ARBITRATION FIELD que esté localmente libre y lo envía al pseudo-objeto mediante un mensaje ANNOUNCEMENT, junto con el tag para el canal de invocación. Un nodo consumidor C debe suscribirse al canal de invocación para lo cual, envía un mensaje SUBSCRIPTION al CONJOINER con el tag del canal correspondiente. Recibe de vuelta un mensaje CHANNEL INFORMATION con el ARBITRATION FIELD que debería empezar a aceptar, actualizando el registro mask-and-match de su adaptador CAN, para recibir datos del productor P.

Esta alternativa se descartó por la existencia del CONJOINER, un componente centralizado en una arquitectura que se pretende distribuida. Al introducir un punto único de falla se afecta la confiabilidad, disponibilidad y dependabilidad del bus en su conjunto y de las aplicaciones que se construyen sobre él. En conclusión, una falla en el CONJOINER impediría cualquier intercambio de datos del proceso controlado, el uso que normalmente se les da a las comunicaciones productor / consumidor.

La alternativa distribuida propone repartir el control de la membresía de los grupos en múltiples agentes distribuidos. Cada instancia de la ORB debe tener un agente, los cuales reciben el nombre de Multicast Agents (MCA). La arquitectura que se decribe a continuación es la que finalmente se implementó en el middleware CORBA para el bus CAN de este trabajo.

Un agente sólo maneja información local. Esto quiere decir que sólo maneja información sobre los grupos para los cuales existe, al menos, un objeto miembro activo en la instancia de la ORB asociada. De igual forma, un objeto que pertenece a un grupo no tiene conocimiento del número y ubicación de los restantes miembros. Esto hace que el cliente que invoca a un grupo, enviando un mensaje multicast, desconozca si está enviando un mensaje local o remoto. Esta transparencia simplifica el diseño e implementación de los objetos así como el esfuerzo de configuración agregando o quitando objetos de un grupo.

Cada agente distribuido o MCA mantiene la siguiente información:

- **Private Group List o PGL**: es una lista de los grupos de objetos para los cuales, al menos, un miembro se encuentra activo en la instancia de la ORB asociada al agente.

- **Private Membership List o PML**: para cada grupo, se mantiene una lista de los objetos que le pertenecen y que se encuentran activos en la instancia de la ORB asociada.

Cada grupo de objetos se identifica mediante un número entre 1 y 31. Esta restricción tiene que ver con el uso del ARBITRATION FIELD de los frames CAN. La PGL le sirve a la instancia de la ORB para determinar los grupos para los cuales debe aceptar mensajes GIOP. Así, en base a los identificadores de los grupos, puede calcular el valor del registro mask-and-

match de su adaptador CAN. Por otro lado, la PML le permite al POA, al recibir la instancia de la ORB un mensaje GIOP, ubicar los objetos pertenecientes a cierto grupo sobre los cuales se debe invocar un método.

Entonces, la estructura de un agente multicast, con dos grupos y dos objetos activos en el primero de estos, es:

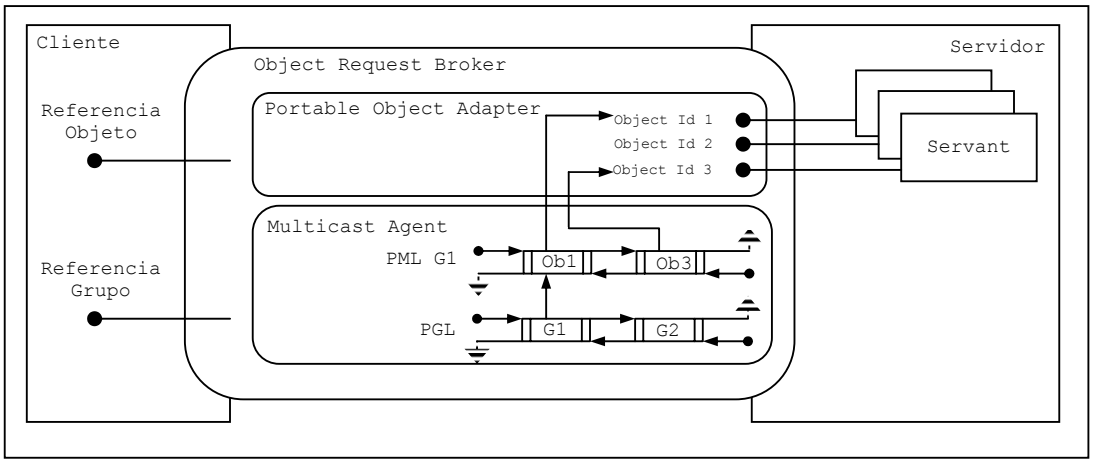


Ilustración 28 - Estructura de un agente MCA

A continuación se incluye la definición de las estructuras usadas para implementar el agente, la PGL y la PML en esta ORB CORBA para el bus CAN:

```
typedef struct {
	CORBA_ORB orb;

	/* Lista de grupos: PRIVATE GROUP LIST */
	t_list groups;
} t_can_mca;

typedef struct {
	char group_id;

	/* Lista de miembros de un grupo: PRIVATE
	MEMBERSHIP LIST, una por grupo */
	t_list members;

	/* Esto permite al MCA controlar las interfases implementadas
	por un objeto al ser agregado a algún grupo */
	int num_interfaces;
	char** interfaces;
} t_can_mca_group;

typedef struct {
	/* Identificador del objeto miembro. Lo usa el POA */
	CORBA_unsigned_short object_key;

	t_list groups;
} t_can_mca_member;
```

Código 35 - Definición de un agente MCA en C

En conclusión, el MCA funciona como un broker del servicio de membresía de los grupos, mediante los métodos Join, para ingresar, y Leave, para salir. Así, el MCA termina siendo una estructura o capa intermedia que permite separar el comportamiento de los objetos de aspectos de configuración no funcionales.

Las aplicaciones construidas sobre este middleware, ya sea que se trate de clientes o de servidores, pueden acceder a los servicio del agente mediante la interfase de su propia instancia de la ORB. A la interfase estándar para la ORB [Corba Core 02], se le agregaron las siguientes funciones:

```
/* Permite agregar el objeto obj al grupo identificado por group_id */
void CORBA_ORB_object_join(
        CORBA_ORB orb,
        CORBA_Object obj,
        char group_id,
        CORBA_Environment *ev);


/* Permite quitar el objeto obj del grupo identificado por group_id */
void CORBA_ORB_object_leave(
        CORBA_ORB orb,
        CORBA_Object obj,
        char group_id,
        CORBA_Environment *ev);


/* Devuelve TRUE si el objeto obj es miembro grupo identificado por group_id */
CORBA_boolean CORBA_ORB_object_is_member(
        CORBA_ORB orb,
        CORBA_Object obj,
        char group_id,
        CORBA_Environment *ev)
```

Código 36 - Interfase de acceso a los servicios del MCA de la ORB para CAN

La función CORBA_ORB_object_join controla que el nuevo miembro tenga al menos una interfase en común con los objetos que ya son miembros del grupo. Los clientes deben usar dicha interfase común para solicitar servicios al grupo en su conjunto. La ORB conoce todas las interfases que implementa un objeto dado porque la función de creación, perteneciente al stub generado por el compilador de OMG IDL, carga dicha información en la estructura CORBA_Object.

5.3 Capa de transporte: arbitración bajo CANIOP

De acuerdo a la especificación original de CORBA [Corba Core 02], GIOP es un protocolo de sesión abstracto, diseñado para correr sobre cualquier capa de transporte que cumpla con ciertas condiciones. A la hora de implementarlo, es necesario mapearlo sobre la capa de transporte que corresponda.

En esta sección se presentan algunos aspectos del mapeo de GIOP sobre CAN que fue finalmente implementado en la ORB. Al mapeo se le dió el nombre de CAN Inter-ORB Protocol (CANIOP). Sería un ESIOP para el bus CAN, equivalente a IIOP para TCP/IP.

CANIOP se identifica como funcionalidad perteneciente a la capa de transporte porque provee los mecanismos necesarios para la transferencia de datos end-to-end. No define un protocolo de transporte propiamente dicho ni ofrece funcionalidad típica de transporte, como control de flujo end-to-end o detección de errores. Simplemente explota las facilidades de CAN para ofrecer transferencia end-to-end confiable y transparente.

CAN no necesita usar direcciones de los dispositivos de destino cuando tiene que enviar un mensaje. A cada mensaje se le asigna un número, enviándolo luego a la red para que cada dispositivo determine, en base a ese número, si debe aceptar el mensaje o no [VanGompel 01]. Esta técnica se conoce como Message Filtering.

En los trabajos previos sobre este tema [Hong 00] [Kim 00] [Lankes 03], se plantean formatos para el ARBITRATION FIELD que respetan este comportamiento y no incluyen identificadores o direcciones del nodo de destino del mensaje. Sólo se incluyen identificadores del transmisor como por ejemplo, un identificador para el dispositivo y un número local de puerto.

Sin embargo, esto trae como consecuencia, la necesidad de ofrecer algún mecanismo para que los receptores conozcan el valor del ARBITRATION FIELD que van a usar los transmisores. En los trabajos de Hong y Kim esto se resuelve mediante el componente CONJOINER y un protocolo especial para suscripción a canales de invocación [Hong 00] [Kim 00]. En el trabajo de Lankes, al momento de establecer una conexión, transmisor y receptor intercambian los valores usados en el ARBITRATION FIELD en cada sentido de comunicación [Lankes 03].

Los formatos anteriores tienen como desventaja el overhead existente antes de poder enviar una solicitud de servicio, ya sea haciendo el bind de un canal de invocación o estableciendo una conexión entre dispositivos.

El formato que se plantee para el ARBITRATION FIELD de los frames CAN debe contemplar la necesidad de un modelo de comunicación peer-to-peer, de un modelo productor / consumidor y por último, de un modelo master / slave.

5.3.1 Modelos master / slave y productor / consumidor

Si se combina el concepto de grupos identificables de objetos, utilizado por el MCA, con la funcionalidad de Message Filtering del bus CAN, es posible definir un formato para el ARBITRATION FIELD que permita recibir sólo aquellos mensajes destinados a grupos para los cuales, la instancia de la ORB tiene, al menos, un miembro activo [Kaiser 98].

De esta forma, las instancias de la ORB que corren en los dispositivos CAN sólo procesan mensajes para grupos que les resultan relevantes. El resto son directamente descartados por el hardware del adaptador CAN. Así, se minimizan las interrupciones del controlador CAN al dispositivo y mejora la performance. Por otro lado, gracias al Message

Filtering y al broadcast de CAN, el transmisor puede estar seguro que la solicitud le llegó a todos los miembros del grupo.

Para conseguir lo anterior, es necesario incluir el identificador del grupo al cual está destinada la solicitud en el ARBITRATION FIELD. El formato propuesto por CANIOP es:

- **Protocol** (1 bit): este campo debe valer 0 para solicitudes destinadas a grupos de objetos (productor / consumidor o master / slave). Por el contrario, si la solicitud está destinada a un único objeto, este campo debe valer 1 (peer-to-peer).

- **RxGroupId** (5 bits): identificador del grupo al cual está destinada la solicitud contenida en el campo DATA FIELD del frame CAN. Cada dispositivo CAN que reciba el mensaje, debe aplicar Message Filtering y aceptar sólo aquellos mensajes destinados a grupos existentes en la instancia local de la ORB.

- **TxNodeId** (5 bits): es posible que más de un dispositivo quiera transmitir una solicitud al mismo grupo de objetos al mismo tiempo. Si no se incluyese un identificador del dispositivo transmisor, podría suceder que dos dispositivos compitan por el control del bus con el mismo valor de arbitración.

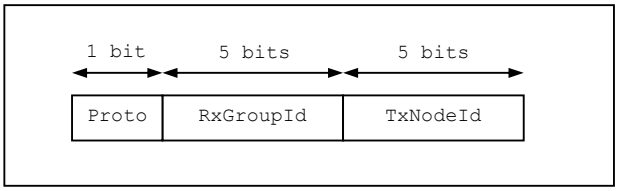


Ilustración 29 - Protocolo CANIOP para comunicaciones master / slave o productor / consumidor

Los clientes envían frames CAN que incluyen un identificador del grupo de objetos destinatario de la solicitud. Es necesario definir que valores deben aceptar los servidores. La Private Group List del MCA sirve para determinar los grupos para los cuales se deben aceptar mensajes GIOP. En base a los identificadores de estos grupos, se puede calcular el valor del registro mask-and-match de cada adaptador CAN.

Fue necesario definir un algoritmo que permitiese calcular la máscara (Receiver Message Identifier Mask) y el valor (Receiver Message Identifier Selector) de aceptación de frames CAN. Este algoritmo se describe a continuación.

Tomando todos los identificadores de los grupos presentes en la PGL, hay que ir comparando bit por bit. Cuando aparecen valores distintos, se debe dejar pasar tanto un 0 como un 1 en esa posición del ARBITRATION FIELD. Ese bit debe ser un “don’t care” lo cual se traduce en un 0 en la máscara. De igual forma, cuando un bit es 0 o 1 para todos los identificadores de la PGL, en la máscara debe figurar un 1. Entonces:

$$PGL = \{G_i\}, i \in <0, n-1>$$

$$\left.\begin{array}{cc|l} A & B & M \\ \hline 0 & 0 & 1 \rightarrow \overline{A} \cdot \overline{B} \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \rightarrow A \cdot B \end{array}\right\} \Rightarrow M = \overline{A} \cdot \overline{B} + A \cdot B \Rightarrow M = \prod_{i=0}^{n-1} \overline{G_i} + \prod_{i=0}^{n-1} G_i$$

$$\left.\begin{array}{cc|l} A & B & V \\ \hline 0 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \rightarrow A \cdot B \end{array}\right\} \Rightarrow V = A \cdot B \Rightarrow V = \prod_{i=0}^{n-1} G_i$$

Ecuación 2 - Algoritmo para cálculo de registro mask-and-match en CANIOP

La función M define el valor que debe tomar el Receiver Message Identifier Mask (RMIM) mientras que la función V define el valor de Receiver Message Identifier Selector (RMIS). En ambos casos, el cálculo se realiza a partir de los elementos presentes en la PGL del MCA y los valores resultantes, corresponden a los bits usados por el RxGroupId en el ARBITRATION FIELD de los frames CAN.

Supóngase que en un momento determinado, la PGL de una instancia de la ORB contiene los identificadores de grupo 1, 3 y 5. Estos son los posibles valores de RxGroupId que permitirían que el frame correspondiente sea aceptado.

$$PGL = \{G_1, G_3, G_5\}$$

$$M = \overline{G_1} \cdot \overline{G_3} \cdot \overline{G_5} + G_1 \cdot G_3 \cdot G_5$$

$$M = \overline{00001} \cdot \overline{00011} \cdot \overline{00101} + 00001 \cdot 00011 \cdot 00101$$

$$M = 11001$$

$$V = G_1 \cdot G_3 \cdot G_5 = 00001 \cdot 00011 \cdot 00101 = 00001$$

RMIM	1	11001	00000
RMIS	0	00001	00000

Ecuación 3 – Ejemplo de aplicación de algoritmo (registro mask-and-match en CANIOP)

Como se puede ver, los valores de RMIM y RMIS del adaptador CAN se configuran para aceptar mensajes con el primer bit en 0. O sea, el campo de protocolo indica una comunicación master / slave o productor / consumidor. Luego, los bits correspondientes al RxGroupId se configuran con los valores calculados de M y V, respectivamente. Por último, los 5 bits correspondientes a TxNodeId son “don’t care” (0 en la máscara) lo cual indica que cualquier dispositivo puede enviar una solicitud a los grupos presentes en la PGL del MCA.

Ahora, supóngase que llega un frame CAN al dispositivo correspondiente con RxGroupId en 5, un frame más con 7 y otro con 14. Al aplicar la máscara (RMIM) y comparar con el valor antes calculado (RMIS), se obtiene:

$M \cdot V ? M \cdot G_5$	$M \cdot V ? M \cdot G_7$	$M \cdot V ? M \cdot G_{14}$
$11001 \cdot 00001 ? 11001 \cdot 00101$	$11001 \cdot 00001 ? 11001 \cdot 00111$	$11001 \cdot 00001 ? 11001 \cdot 01110$
$00001 = 00001$	$00001 = 00001$	$00001 \neq 01000$
G_5 aceptado	G_7 aceptado	G_{14} rechazado

Ecuación 4 - Ejemplo de aplicación de algoritmo (registro mask-and-match en CANIOP

Como se puede ver en el ejemplo anterior, el frame con RxGroupId en 7 es aceptado en forma incorrecta porque no pertenece a la PGL del MCA. Sin embargo, el valor de RxGroupId cumple con lo requerido por RMIM y RMIS. Esto constituye una limitación del mecanismo mask-and-match del adaptador CAN a pesar de la cual, sigue siendo conveniente ya que la mayoría de los frames no deseados son rechazados por el hardware. El POA y el MCA del prototipo están preparados para manejar estos casos.

El mecanismo de comunicación en grupos funciona correctamente porque todos los dispositivos que en su PGL tengan el identificador de grupo 5, van a tener un valor en su RMIM y en su RMIS que deje pasar los frames con RxGroupId en 5. Así, la solicitud llega a todas las instancia de la ORB que tengan, al menos, un objeto activo para el grupo 5.

5.3.2 Modelo peer-to-peer

Hasta el momento, se presentó el protocolo usado por CANIOP para soportar los modelos de comunicación productor / consumidor y master / slave. Faltaría plantear el protocolo a usar para comunicaciones peer-to-peer en el prototipo de ORB para el bus CAN.

CAN no necesita usar direcciones de los dispositivos de destino cuando tiene que enviar un mensaje. Sin embargo, en el protocolo CANIOP para comunicaciones peer-to-peer, se incluye el identificador del dispositivo destinatario del mensaje en el ARBITRATION FIELD de los frames CAN.

El formato propuesto por CANIOP para el modelo de comunicaciones peer-to-peer sería:

- **Protocol** (1 bit): la solicitud corresponde al modelo de comunicaciones peer-to-peer. Está destinada a un único objeto, este campo debería valer 1.

- **RxNodeId** (5 bits): identificador del dispositivo al cual está destinada la solicitud contenida en el campo DATA FIELD del frame CAN. El mensaje debería ser aceptado únicamente por el dispositivo cuyo identificador coincide con el valor de este campo.

- **TxNodeId** (5 bits): es posible que más de un dispositivo quiera transmitir una solicitud al mismo destinatario al mismo tiempo. Si no se incluyese un identificador del dispositivo transmisor, podría suceder que dos dispositivos compitan por el control del bus con el mismo valor de arbitración.

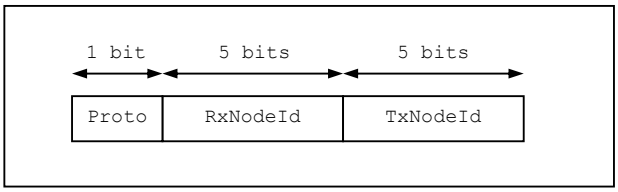


Ilustración 30 - Protocolo CANIOP para comunicaciones peer-to-peer

Todos los dispositivos deben configurar su adaptador CAN para recibir siempre mensajes con Protocol en 1 y RxNodeId con el identificador o dirección que tienen asignado. Por ejemplo, el dispositivo con identificador 23 debería fijar los bits correspondientes a RxNodeId del RMIM en 0x1F mientras que los del RMIS deberían ser 0x17 (23). Los bits correspondientes a TxNodeId son todos 0 tanto en el RMIM como en el RMIS (se reciben mensajes de cualquier transmisor).

Un dispositivo debe ser capaz de recibir mensajes con Protocol en 0 o en 1 y con los 5 bits siguientes fijados al valor del identificador del nodo o al valor calculado a partir de la PGL del MCA. Esto es posible porque el simulador del bus CAN, usado para construir y probar el prototipo de ORB para CAN, provee funcionalidad de Message Filtering tipo BasicCAN con registros mask-and-match dobles [Babiuch 00].

Una de las ventajas de no usar direcciones en los frames CAN es la flexibilidad del sistema, en tanto el software y el hardware pueden manejarse con identificadores de datos y no de dispositivos. Al incluir el campo RxNodeId en el ARBITRATION FIELD, aparentemente esta ventaja se perdería.

En IIOP, los clientes trabajan con referencias a objetos que contienen las direcciones TCP/IP de las instancias de la ORB donde estos se ubican. En forma similar, en CANIOP los clientes van a trabajar con referencias que contienen el identificador del dispositivo donde reside el objeto correspondiente. En ambos casos, las direcciones de las instancias servidor son manejadas en forma transparente.

Por el mecanismo de referencias interoperables de CORBA (IOR), la flexibilidad del sistema, generada por el uso de CAN y aparentemente perdida, se sigue manteniendo. El software no necesita conocer direcciones o identificadores de dispositivos. Sólo necesita las referencias las cuales manejan, en forma transparente, todas las direcciones necesarias.

5.4 Capa de transporte: perfiles CANIOP

Según la especificación original de CORBA, los servidores publican referencias a objetos en forma de Interoperable Object References (IORs). Para habilitar la comunicación entre instancias de la ORB, las referencias contienen la dirección TCP/IP donde puede ser ubicado el objeto referido. De esta forma, al encapsular la ubicación de los objetos mediante referencias, se consigue transparencia de ubicación.

En CANIOP esta transparencia se sigue manteniendo pero hay que tener en cuenta que ahora, las referencias pueden apuntar a objetos particulares o bien, a grupos de objetos. Al igual que IIOP, las referencias generadas por la ORB para CAN contienen una estructura llamada ProfileBody. Esta encapsula los elementos necesarios para ubicar los objetos o grupos de objetos. Los campos de esta estructura son:

```
module CANIOP {
    ...
    enum ReferenceType {
        ObjectProfile,
        GroupProfile
    };

    enum ProfileType {
        CanBusProfile
    };

    struct ProfileBody {
        octet profile_type;
        octet reference_type;

        union u switch(ReferenceType) {
            case ObjectProfile:
                octet rx_node_id;
                octet[2] object_key;

            case GroupProfile:
                octet group_id;
        }
    };
    ...
};
```

Código 37 - Estructura de un perfil CANIOP en OMG IDL

Campo	Descripción
reference_type	Indica si la referencia que contiene este perfil apunta a un objeto en particular (los campos rx_node_id y object_key son relevantes) o bien, a un grupo de objetos (sólo el campo group_id es relevante).
profile_type	Indica el tipo de perfil que contiene la IOR correspondiente. Para la ORB para CAN el único valor permitido es 0 (correspondiente a CanBusProfile en el enum ProfileType).
rx_node_id	Identificador (dirección) del nodo o dispositivo donde reside el objeto identificado por object_key. Este campo es relevante si la referencia que contiene el perfil apunta a un objeto en particular.

object_key	Identificador del objeto para el cual se va a enviar el mensaje de solicitud. Este campo es relevante si la referencia que contiene el perfil apunta a un objeto en particular.
group_id	Identificador del grupo de objetos al cual se está enviando la solicitud. Este campo es relevante si la referencia que contiene el perfil apunta a un grupo de objetos.

Tabla 39 - Campos de un perfil CANIOP

De acuerdo a la definición de CANIOP, cuando se utiliza el modelo peer-to-peer y un cliente desea acceder a un objeto particular, es necesario incluir en el ARBITRATION FIELD del frame CAN el identificador del nodo o dispositivo destinatario del mensaje (RxNodeId). El cliente puede recuperar este valor a partir del perfil CANIOP contenido en la referencia al objeto correspondiente.

Por el contrario, cuando se quiere acceder a un grupo de objetos y se utiliza el modelo master / slave o productor / consumidor, el ARBITRATION FIELD del frame CAN debe contener el identificador del grupo deseado (RxGroupId). Al igual que en el caso anterior, el cliente puede recuperar este valor a partir del perfil CANIOP.

El diseño de ORB propuesto en este capítulo incorpora una serie de funciones a la interfase de la ORB para permitir trabajar con este nuevo tipo de referencias. Estas funciones son:

```
/* Genera un string conteniendo un CAN_IOR para un objeto */
CORBA_char* CORBA_ORB_object_to_string(
      CORBA_ORB orb,
      CORBA_Object obj,
      CORBA_Environment *ev);

/* Genera un string conteniendo un CAN_IOR para un grupo de objetos */
CORBA_char* CORBA_ORB_group_to_string(
      CORBA_ORB orb,
      char group_id,
      CORBA_Environment *ev);

/* Genera una referencia a objeto o grupo a partir de un string que contenga un CAN_IOR */
CORBA_Object CORBA_ORB_string_to_object(
      CORBA_ORB orb,
      CORBA_char *str,
      CORBA_Environment *ev);
```

Código 38 - Interfase para manejo de referencias de la ORB para CAN

Las instancias que actúan como servidores tienen funciones diferentes para publicar referencias a objetos particulares (CORBA_ORB_object_to_string) o a grupos de objetos (CORBA_ORB_group_to_string). Sin embargo, los clientes tienen una única función para trabajar con ambos tipos de referencia (CORBA_ORB_string_to_object).

En consecuencia, el hecho que un IOR apunte a un objeto o a un grupo resulta completamente transparente para los clientes. Para estos, a la hora de construir una aplicación, no existe diferencia a nivel de código entre solicitar un servicio a un objeto o a un grupo de estos. Directamente llaman al stub pasando el valor que haya devuelto CORBA_ORB_string_to_object, sin saber si corresponde a un objeto o a un grupo.

La ORB para CAN también permite publicar referencias a objetos o grupos de objetos convirtiendo un IOR a string. Los strings correspondientes a referencias de CORBA estándar pueden diferenciarse de los generados por la ORB propuesta en el presente trabajo porque los primeros, comienzan con el prefijo “IOR:” mientras que los segundos, con “CAN_IOR:”.

5.5 Capa de sesión: GIOP modificado

La especificación original de CORBA [Corba Core 02] incluye un protocolo de sesión abstracto que permite organizar el diálogo entre las instancias de la ORB que corren en cada máquina, garantizando la interoperabilidad entre las mismas. Este protocolo, General Inter-ORB Protocol (GIOP), está pensado para correr sobre cualquier capa de transporte que cumpla con las condiciones fijadas por CORBA.

Al adaptar CORBA al bus CAN e implementar un prototipo de ORB, surgió la necesidad de modificar el protocolo de sesión estándar por dos motivos. En primer lugar, la capa de transporte subyacente, CANIOP, no cumple con los requerimientos de GIOP. En segundo lugar, CAN es una red de bajo ancho de banda con frames cuyo payload está limitado a 8 bytes. El protocolo de sesión estándar tiende a producir mensajes cuya longitud resulta excesiva para la capacidad de esta red.

En esta sección se explica la nueva capa de sesión cuyos objetivos principales son la minimización del tamaño de los mensajes GIOP y el funcionamiento sobre el protocolo CANIOP (incluyendo soporte para grupos de objetos). El nuevo protocolo de sesión se llama GIOP modificado.

Para cada uno de los requerimientos que GIOP impone a la capa de transporte subyacente [Corba Core 02], se analiza en que medida CANIOP los cumple o los deja de cumplir:

- **Orientado a conexión**: CANIOP ofrece un servicio de transporte end-to-end sin conexión. Por este motivo, el alcance de los identificadores de los requests pasa a ser una instancia de ORB en lugar de estar acotado por una conexión. La unicidad de los mismos debe estar garantizada en cada instancia de ORB.

- **Confiable**: CANIOP ofrece confirmación positiva de entrega (acknowledgement) aprovechándose directamente del mecanismo de confirmación del bus CAN. Los bytes son recibidos por la capa de sesión en el orden en que fueron enviados.

- **Stream de bytes**: CANIOP no impone limitaciones arbitrarias al tamaño de los mensajes ni se requiere fragmentación ni alineamiento de los mismos. CANIOP se maneja con streams de bytes.

- **Notificación de pérdida de conexión**: el transporte subyacente no se maneja con conexiones por lo cual, no es posible cumplir con este requerimiento.

- **Manejo de conexiones**: el transporte subyacente no maneja conexiones por lo cual, no existe la posibilidad de soportar un modelo de establecimiento de conexiones similar al de TCP/IP. Sin embargo, se siguen manteniendo los roles conceptuales de cliente y servidor para las instancias de la ORB que participan en un diálogo.

Dadas las características del ambiente en el cual va a correr GIOP modificado, muchos de los mensajes definidos en la especificación original pierden sentido. De los 7 mensajes definidos originalmente, en GIOP modificado quedan 3, suficientes para la funcionalidad ofrecida por el prototipo de ORB CORBA sobre el bus CAN.

Los mensajes que siguen formando parte de GIOP modificado son Request, Reply y MessageError. El resto de los mensajes fueron descartados por los siguientes motivos:

- **Mensaje CancelRequest**: no se incluye porque el prototipo no soporta invocación asincrónica de métodos. Los stubs estáticos, generados por el compilador de OMG IDL, sólo permiten invocación sincrónica.

  Para invocar un método en forma asincrónica es necesario usar la interfase DII, eliminada del prototipo de ORB para minimizar los requerimientos de recursos [Corba Min 02]. Uno de los objetivos de diseño del prototipo es permitir la utilización del mismo en ambientes con restricciones importantes de recursos.

- **Mensaje LocateRequest**: los objetos activos en cada instancia de la ORB son usados para modelar el dispositivo real en el cual están corriendo. O sea, cada dispositivo es visto como un conjunto de objetos que ofrecen servicios y que representan componentes reales del dispositivo.

  Por el uso que se les da a los objetos, es bastante poco probable que surja la necesidad de migrar objetos en forma dinámica. O sea, al representar componentes físicos que no cambian de dispositivo, los objetos tampoco van a necesitar migrar de dispositivo.

- **Mensaje LocateReply**: este mensaje fue excluido de GIOP modificado por el mismo motivo que el mensaje anterior. Los objetos usados en un sistema de control distribuido sobre un bus industrial como CAN, no tienen necesidad de migrar entre instancias de ORB o sea, entre dispositivos.

- **Mensaje CloseConnection**: el transporte subyacente, CANIOP, es sin conexión por lo cual, los servidores no tienen necesidad de indicar a los clientes que se disponen a cerrar la conexión de transporte entre ellos.

Al igual que en GIOP, en GIOP modificado todos los mensajes responden al mismo formato. En primer lugar, se incluye un encabezado general del protocolo. Luego, un encabezado particular para cada tipo de mensaje y por último, información adicional si es que el mensaje lo requiere. Este sería el encabezado general de GIOP modificado en OMG IDL:

```
module GIOPModif {
    ...
    enum MsgType {
        Request,
        Reply,
        MessageError
    };

    struct MessageHeader {
        octet type_and_flags;
        octet message_size[2];
    };
    ...
};
```

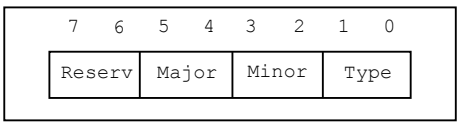
Código 39 - Encabezado general de mensajes de GIOP modificado en OMG IDL

Como se puede ver en el código anterior, se eliminaron varios campos del encabezado original. A continuación se detallan estos campos y se presentan los motivos de la eliminación:

Campo eliminado	Justificación
magic	Este campo contiene un identificador de los mensajes GIOP (cuatro caracteres "GIOP" codificados en ISO Latin-1 8859.1). Gracias a este campo, la capa de sesión puede distinguir si el transporte le está devolviendo o no un mensaje válido. En una red CAN, que corre una aplicación distribuida de control, todos los dispositivos van a correr el prototipo de ORB y hablar GIOP modificado. No tiene sentido mezclar capas de aplicación (por ejemplo, CORBA sobre CAN con DeviceNet). Por este motivo, no existe la posibilidad de recibir un mensaje de otro protocolo.
GIOP_version	Originalmente, contenía el número de versión del protocolo GIOP que se está usando en el mensaje. GIOP modificado también envía el número de versión del protocolo. Sin embargo, intenta usar una cantidad menor de bytes. Como se verá a continuación, el contenido de este campo aparece en el campo type_and_flags.
flags	Este campo contenía dos flags: BO y MF. Como se verá más adelante, la nueva capa de presentación CCDR no requiere que el transmisor envíe al receptor el byte order usado. Por este motivo, el flag BO desaparece. No se soporta la posibilidad de enviar mensajes fragmentados por lo cual, el flag MF tampoco es necesario.
message_type	Tipo de mensaje. Como GIOP modificado soporta más de un tipo de mensaje, sigue siendo necesario enviar un identificador en el encabezado general. Sin embargo, al existir sólo 3 tipos de mensaje es posible usar menos de un byte. Como se verá a continuación, el contenido de este campo aparece en el campo type_and_flags.

Tabla 40 - Campos eliminados del encabezado general de mensajes GIOP

Teniendo en cuenta la tabla anterior, el formato final del encabezado general de mensajes de GIOP modificado sería:

Campo	Descripción
type_and_flags	Este campo contiene una serie de flags y un indicador del tipo de mensaje de GIOP modificado que sigue al encabezado general. El formato del campo sería:  *Ilustración 31 – type_and_flags (encabezado GIOP modificado)* • **Type** (2 bits): es un indicador del tipo de mensaje. Como en GIOP modificado sólo se definen tres mensajes, 2 bits son suficientes para codificar este identificador. Los posibles valores van de 0 a 2.

	• **Minor** (2 bits): indica el minor de la versión del protocolo GIOP modificado que se está usando. Por el momento, el único valor posible es 0. • **Major** (2 bits): indica el major de la versión del protocolo GIOP modificado que se está usando. Por el momento, el único valor posible es 1. A la especificación de GIOP modificado que se describe en el presente trabajo, se le asigna el número de versión 1.0.
message_size	Número de bytes en el mensaje sin contar el encabezado general de GIOP modificado. A diferencia, de GIOP estándar, en el cual se usan 4 bytes para este campo, en GIOP modificado sólo se usan 2 bytes. El valor se codifica con el byte order especificado por la capa de presentación del prototipo, CCDR.

Tabla 41 - Campos del encabezado general de mensajes de GIOP modificado

El primer tipo de mensaje que se describe es el Request. Al igual que en GIOP, este permite codificar invocaciones, generadas por los clientes y enviadas a los servidores. Vale aclarar que una invocación puede estar destinada a un objeto en particular o bien, a un grupo de objetos, residentes en múltiples servidores.

Un mensaje de este tipo está formado por un encabezado general GIOP, un encabezado especifico del tipo de mensaje y un cuerpo, formado por los parámetros in e inout de la operación que se esté invocando (codificados de acuerdo a las reglas definidas por la capa de presentación del prototipo, llamada CCDR). Esta es la definición del encabezado específico de Request:

```
module GIOPModif {
    ...
    struct RequestHeader {
        octet request_id[2];
        octet object_key[2];
        char operation_id;
    };
    ...
};
```

Código 40 - Encabezado de mensajes de GIOP modificado (Request) en OMG IDL

Como se puede ver en el código anterior, se eliminaron varios campos del encabezado original de GIOP estándar. A continuación se presenta una lista de estos campos junto con los motivos de la eliminación de cada uno de ellos:

Campo	Descripción
service_context	En la especificación original de GIOP, contiene el contexto de invocación del cliente que se envía al servidor. Como ya se dijo, vendría a ser una lista de parámetros "ocultos" adicionales. En el prototipo de ORB sobre CAN no existe la posibilidad de enviar parámetros "ocultos". Todos los parámetros deben declararse en OMG IDL, al definir la interfase del objeto que se está invocando. En fin, no hay soporte para contextos de invocación.

response_expected	En GIOP estándar, indica al cliente si debe esperar un mensaje Reply del servidor una vez que envió el Request. En GIOP modificado, el cliente puede determinar si debe esperar o no una respuesta en base a los parámetros de la solicitud. Lo mismo puede hacer el servidor. Hay dos posibilidades: • **Invocación a un objeto particular**: todas las invocaciones bajo el modelo peer-to-peer (a un objeto en particular) son sincrónicas por lo cual, siempre hay que esperar una respuesta. • **Invocación a grupo de objetos**: todas las invocaciones bajo el modelo master / slave o productor / consumidor se realizan con semántica "oneway". Esto quiere decir que no hay respuesta de los múltiples servidores (slaves o consumidores) al cliente (master o productor).
Reserved	Se trata de 3 bytes reservados para uso futuro. Al no usarse en GIOP estándar, directamente son eliminados en GIOP modificado debido a las restricciones de ancho de banda y tamaño de mensajes del bus CAN.
requesting_principal	No se usa más en las implementaciones actuales de CORBA. En las nuevas versiones de GIOP estándar (versiones 1.2 y 1.3) el campo ya ha sido eliminado del encabezado. GIOP modificado hace lo mismo y elimina este campo.

Tabla 42 - Campos eliminados del encabezado de mensajes GIOP (Request)

Teniendo en cuenta los cambios propuestos por la tabla anterior, el encabezado de mensajes Request en GIOP modificado está formado por los siguientes campos:

Campo	Descripción
request_id	Permite identificar los mensajes y asociar un Request con un Reply. El cliente es responsable de la generación de estos números y se debe asegurar que no haya valores ambiguos. A diferencia de GIOP estándar, GIOP modificado usa sólo 2 bytes en lugar de 4 para este campo.
object_key	Identificador del objeto destino de la invocación. Este valor se obtiene del IOR y sólo es relevante para el servidor. El cliente no lo interpreta ni lo modifica. En GIOP estándar, este campo tenía una longitud variable al estar definido como una secuencia de octets. En GIOP modificado, un identificador de objeto siempre ocupa 2 bytes (es un unsigned short). En un sistema distribuido que use la propuesta de CORBA sobre CAN descripta en el presente trabajo, pueden existir un máximo de 65536 objetos repartidos en 32 dispositivos (o sea, un máximo de 2048 objetos por dispositivo).

	Esta limitación surge de la estructura del campo object_key. Cada object_key de 16 bits se divide en 5 bits, para identificar el dispositivo donde el objeto reside, y 11 más, para el identificador del objeto en sí. Es necesario el identificador de cada nodo para garantizar la unicidad, en toda la red, de estos object_keys. Cuando la solicitud está dirigida a un grupo, este campo debería ser 0.
operation	En GIOP estándar, consiste en el nombre del método que se está invocando. Es directamente el mismo nombre que se le dio en la definición OMG IDL. Enviar un string con el nombre del método hace crecer mucho el tamaño de los mensajes. De base, se tienen 4 bytes para la longitud del string más la cantidad de caracteres que sean necesarios. En GIOP modificado, se envía un identificador de operación de un único byte, cuyo valor puede estar entre 0 y 255. Este cambio es transparente para el usuario pero para poder implementarlo en el prototipo, fue necesario modificar el compilador de OMG IDL. En primer lugar, hubo que modificar el generador de stubs para que asigne a cada operación un identificador único por interfase. En segundo lugar, hubo que modificar el generador de skeletons para que seleccionase la implementación en base a un número y no al nombre de la operación.

Tabla 43 - Campos del encabezado de mensajes de GIOP modificado (Request)

Uno o más servidores reciben del cliente un mensaje Request. Estos deben ubicar el objeto o los objetos destinatarios de la invocación, mediante el POA y el MCA. Luego, se debe llamar a la función que implementa la operación OMG IDL requerida para cada objeto. Por último, hay que enviar al cliente los resultados de la invocación mediante un mensaje Reply, si la semántica de la solicitud lo permite. O sea, este último paso se completa únicamente si la invocación está dirigida a un objeto en particular y no a un grupo.

Los mensajes Reply están formados por un encabezado general GIOP, un encabezado especifico del tipo de mensaje y un cuerpo, formado por los parámetros inout, parámetros out y el valor de retorno de la operación que se esté invocando. Esta es la definición del encabezado específico de Reply en GIOP modificado:

```
module GIOPModif {
    ...
    enum ReplyStatusType {
        NO_EXCEPTION,
        USER_EXCEPTION,
        SYSTEM_EXCEPTION
    };

    struct ReplyHeader {
        octet request_id[2];
        octet reply_status;
    };
    ...
};
```

Código 41 - Encabezado de mensajes de GIOP modificado (Reply) en OMG IDL

En la definición anterior, se eliminó un campo del encabezado de GIOP estándar. Este es el motivo de la eliminación del mismo:

Campo	Descripción
service_context	En la especificación original de GIOP, contiene el contexto de invocación del servidor que se envía al cliente. Como ya se dijo, vendría a ser una lista de parámetros "ocultos" adicionales. En el prototipo de ORB sobre CAN no existe la posibilidad de enviar parámetros "ocultos". Todos los parámetros deben declararse en OMG IDL, al definir la interfase del objeto que se está invocando. En fin, no hay soporte para contextos de invocación en sentido cliente-servidor ni servidor-cliente, como es el caso en este punto.

Tabla 44 - Campos eliminados del encabezado de mensajes GIOP (Reply)

Teniendo en cuenta lo anterior, el encabezado de mensajes Reply en GIOP modificado queda compuesto únicamente por los siguientes campos:

Campo	Descripción
request_id	El servidor se limita a poner en este campo el valor que recibió del cliente en el mensaje Request correspondiente. A diferencia de GIOP estándar, GIOP modificado usa sólo 2 bytes en lugar de 4 para este campo.
reply_status	Indica el resultado de la operación. Los posibles valores de este campo están definidos en ReplyStatusType: • **NO_EXCEPTION:** no se produjo ningún error y la solicitud se completó exitosamente. El cuerpo del mensaje contiene parámetros out e inout junto con el valor de retorno, si existe alguno definido para la operación. • **USER_EXCEPTION:** se produjo un error y la operación levantó una excepción definida por el usuario en OMG IDL. El cuerpo del mensaje contiene dicha excepción. • **SYSTEM_EXCEPTION:** se produjo una excepción del sistema y el cuerpo del mensaje responde al formato de la estructura SystemExceptionReplyBody, codificada de acuerdo a la nueva capa de presentación CCDR. LOCATION_FORWARD no va más porque no se soporta migración de objetos. Por otro lado, en GIOP estándar se usan 4 bytes para este campo. En GIOP modificado, sólo se usa 1 byte.

Tabla 45 - Campos del encabezado de mensajes GIOP (Reply)

Los cambios propuestos en esta sección permiten reducir, en primer lugar, el tamaño del encabezado general de 12 bytes a sólo 3 bytes. Luego, para un mensaje de Request, se logra una reducción de un orden de magnitud, pasando de aproximadamente 60 bytes a 5 bytes para una invocación a un método sin parámetros. Algo similar sucede para el mensaje de Reply. En próximas secciones se harán comparaciones más exhaustivas del tamaño de los mensajes y de las reducciones logradas.

Al igual que en GIOP estándar, el mensaje MessageError se usa como respuesta cada vez que se recibe un mensaje inválido desde el punto de vista del protocolo de sesión. El mensaje está formado únicamente por un encabezado general de GIOP modificado.

5.6 Capa de presentación: CCDR

En la especificación original de CORBA [Corba Core 02], las funciones de la capa de presentación son llevadas a cabo por Common Data Representation (CDR). Sus principales características son ordenamiento variable de bytes, alineamiento de tipos primitivos y reglas de codificación.

Estas características indican que, al momento de especificar CDR, se tuvo como objetivo principal la optimización del procesamiento de mensajes. O sea, se pretendió definir un protocolo que permitiese una gran eficiencia en la codificación y decodificación de mensajes GIOP así como en el marshalling y unmarshalling de parámetros [Hong 00]. Por ejemplo, cuando se da un intercambio de mensajes entre un cliente y un servidor que comparten procesadores con el mismo byte order y la misma alineación para acceder a la memoria, la codificación y decodificación de mensajes CDR resulta prácticamente inmediata.

Si bien este enfoque permite minimizar el tiempo de procesamiento de mensajes en la ORB, termina generando mensajes de mayor longitud. Esto no es conveniente para una red de bajo ancho de banda como es CAN. Por este motivo, al adaptar CORBA al bus CAN, se propone una nueva capa de presentación que tenga como objetivo principal, al igual que la capa de sesión, la minimización del tamaño de los mensajes, tratando de acomodar una invocación en un único frame CAN.

La nueva sintaxis de transferencia, implementada en el nuevo middleware, recibe el nombre de Compact Common Data Representation (CCDR). Se basa en las propuestas realizadas en los trabajos de Hong y Kim [Hong 00] [Kim 00] y presenta las siguientes características:

- **Ordenamiento fijo de bytes**

  Dependiendo del procesador usado, distintas máquinas pueden tener distinto byte order. En CDR, se optó por dejar al transmisor (el stub del cliente) usar su propio byte order, incluyendo un indicador o flag en el mismo mensaje GIOP. De esta forma, sólo se intercambian los bytes de un dato si la capa CDR receptora (el skeleton del objeto) detecta diferentes ordenamientos entre transmisor y receptor. Por ejemplo, para enviar el dato n, un long en OMG IDL:

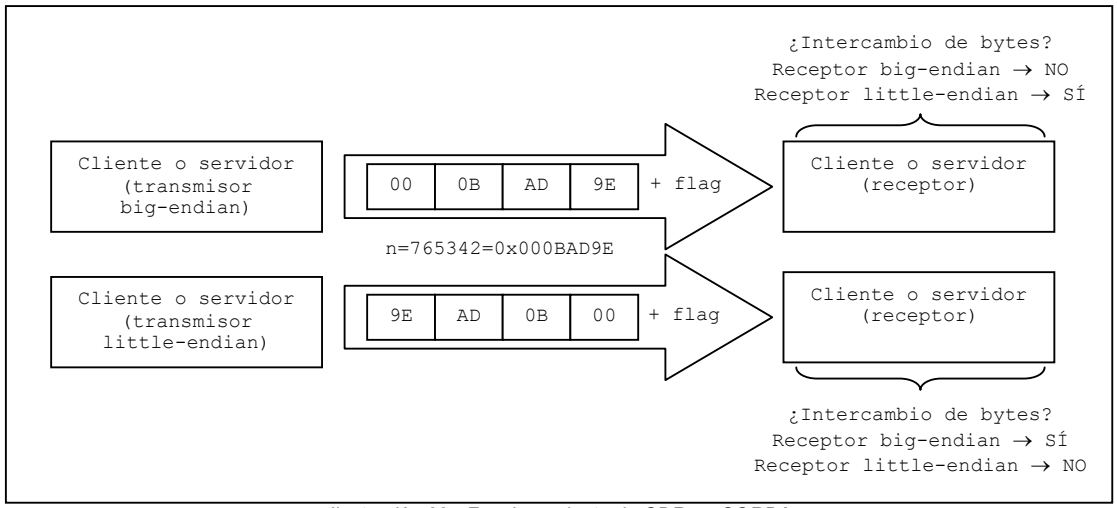


Ilustración 32 - Funcionamiento de CDR en CORBA

En CCDR se adoptó un enfoque distinto y se decidió trabajar con un byte order predeterminado. Todos los intercambios se codifican en network byte order o sea, se usa siempre un ordenamiento big-endian. Los skeletons de los objetos intercambian los bytes de los datos que reciben sólo si su ordenamiento no es big-endian.

En conclusión, el funcionamiento de CCDR en lo referente al ordenamiento de bytes es:

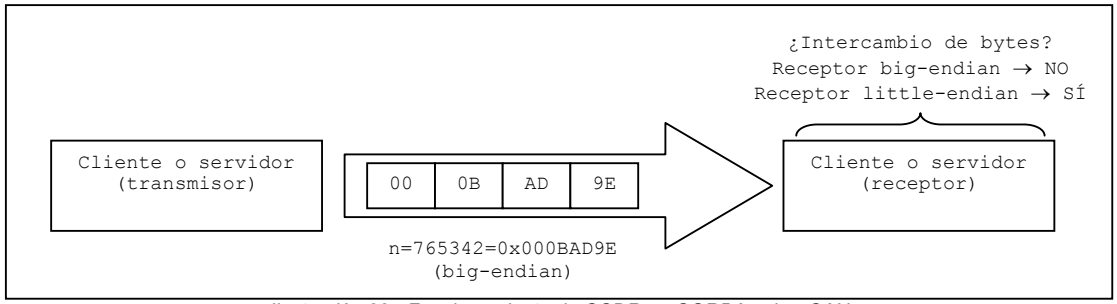


Ilustración 33 - Funcionamiento de CCDR en CORBA sobre CAN

Esta modificación trae dos ventajas. En primer lugar, permite evitar el envío del indicador del byte order usado por el transmisor. En segundo lugar, permite aplicar una regla de codificación de longitud variable para enteros con o sin signo la cual, reduce la cantidad de bytes necesarios para enviar datos numéricos en forma significativa.

- **Alineamiento de tipos primitivos**

La sintaxis de transferencia original de CORBA, CDR, requiere que cualquier tipo primitivo de n bytes comience en una posición dentro del stream que sea múltiplo de n. Este requerimiento pretende facilitar el manejo eficiente de streams CDR por arquitecturas con alineamiento de datos en memoria.

Este requerimiento llevaba a agregar muchos bytes de padding a los streams CDR, incrementándose en forma innecesaria la longitud de los mensajes intercambiados entre clientes y servidores.

En CCDR, para conseguir el objetivo de mensajes de menor longitud, se elimina este requerimiento. Así, con la nueva sintaxis de transferencia, en ningún caso es necesario agregar bytes de padding.

- **Reglas de codificación**

En CCDR, al igual que en CDR, se asume que existe un acuerdo implícito entre el stub del cliente y el skeleton del objeto sobre el orden de la secuencia de datos que van a intercambiar. Así, no es necesario incluir información de tipos en los streams CCDR. En fin, se trabaja en forma similar a External Data Representation (XDR) de ONC RPC [Vondrak 97], colocándose en el stream CCDR, directamente, el valor del dato usando un ordenamiento big-endian.

CCDR respeta las reglas de codificación definidas por CDR excepto para los tipos enteros con o sin signo, que ocupan más de un byte. Los tipos correspondientes de OMG IDL son:

Tipo de Dato	Tamaño	Rango
short	2	$-2^{15}...2^{15}-1$

unsigned short	2	$0...2^{16}-1$
long	4	$-2^{31}...2^{31}-1$
unsigned long	4	$0...2^{32}-1$

Tabla 46 – Tipos de dato con codificación especial en CCDR

Al definir una variable entera, es muy común que se termine asignándole a la misma valores pequeños comparados con el rango permitido. Por ejemplo, rara vez se asigna el valor 2^{32} a una variable de tipo long. De igual forma, en OMG IDL, es muy común el uso de enteros para la longitud de cadenas y secuencias, tratándose normalmente de valores muy pequeños.

Tendiendo en cuenta lo anterior y para disminuir aún más el tamaño de los mensajes GIOP, CCDR propone una codificación de longitud variable para los enteros con o sin signo. Se busca usar menos bytes para los valores más comunes o sea, para los valores más chicos.

Al representarse los enteros de múltiples bytes en big-endian, es posible usar los bits más significativos, no usados en valores pequeños, para indicar el rango del valor y la forma en que este debe ser interpretado.

Tanto en el trabajo de Hong como en el de Kim [Hong 00] [Kim 00], se propone una regla de codificación de longitud variable que define los siguientes rangos para enteros sin signo:

Bits más significativos	Tamaño (bytes)	Rango
00	1	$0...2^{6}-1$
01	2	$2^{6}...2^{14}-1$
10	3	$2^{14}...2^{22}-1$
11	5	$2^{22}-1...2^{32}-1$

Tabla 47 - Propuesta de regla de codificación de longitud variable

A continuación se incluyen algunos ejemplos de aplicación de la regla anterior:

Número	Bits más significativos	Valor Dato	Valor Final
43	00	101011	00101011=0x2B=43
118	01	01110110	0100000001110110=0x4076=118
234	01	11101010	0100000011101010=0x40EA=234

Tabla 48 - Ejemplos de aplicación de la regla propuesta

En el diseño de CCDR propuesto en el presente trabajo, a diferencia de los trabajos antes mencionados [Hong 00] [Kim 00], se plantea una regla similar pero que, aparte de una cantidad variable de bytes, usa una cantidad variable de bits para indicar el rango del número codificado. Se usan menos bits para los rangos aplicados con mayor frecuencia lo cual permite que una mayor cantidad de números sean representables mediante una menor cantidad de bytes.

Las reglas de codificación usadas en CCDR, tanto para enteros con signo como enteros sin signo, son:

Bits más significativos	Tamaño (bytes)	Rango
0	1	$0...2^{7}-1$
10	2	$2^{7}...2^{14}-1$
110	3	$2^{14}...2^{21}-1$
1110	4	$2^{21}...2^{28}-1$
11110	5	$2^{28}...2^{32}-1$

Tabla 49 - Regla de codificación de longitud variable usada en CCDR (enteros sin signo)

Bits más significativos	Tamaño (bytes)	Rango
0	1	-2^{6} a $2^{6}-1$
10	2	-2^{13} a $2^{13}-1$
110	3	-2^{20} a $2^{20}-1$
1110	4	-2^{27} a $2^{27}-1$
11110	5	-2^{31} a $2^{31}-1$

Tabla 50 - Regla de codificación de longitud variable usada en CCDR (enteros con signo)

Aplicando la nueva regla para enteros sin signo a los mismos valores usados anteriormente, se obtiene:

Número	Bits más significativos	Valor Dato	Valor Final
43	0	101011	00101011=0x2B=43
118	0	01110110	01110110=0x76=118
234	01	11101010	0100000011101010=0x40EA=234

Tabla 51 - Ejemplos de aplicación de la regla propuesta

Como se ve en el ejemplo anterior, se logró disminuir la cantidad de bytes para 118. Lo mismo va a suceder con valores grandes que en CCDR se representan mediante 4 bytes en lugar de 5, como sugiere las propuesta antes mencionadas [Hong 00] [Kim 00].

6. Comparación con CORBA estándar

Resumen del capítulo

El nuevo middleware es comparado con la versión estándar de CORBA, de forma de poder evaluar si la funcionalidad provista por este último sigue estando presente en la adaptación al bus CAN.

6.1 Modelo de objetos

El modelo de objetos definido originalmente por la especificación de CORBA prácticamente no sufrió modificaciones en el middleware propuesto para el bus CAN. Se mantienen las mismas entidades y las mismas relaciones entre estas que existían originalmente en el modelo de CORBA, aunque se prevee otro ambiente de ejecución.

En el diseño propuesto, cada dispositivo conectado al bus CAN corre una instancia de la ORB. En un momento dado, un dispositivo puede actuar como cliente y en otro, como servidor. Por ejemplo, cuando un controlador quiere acceder a un actuador, el primero actúa como cliente y el segundo como servidor. En otro momento, puede darse que un sensor quiera informar un valor leído del proceso real a un controlador. En este caso, el primero actúa como cliente y el segundo como servidor.

Los clientes siguen siendo aquellas entidades o agentes, que ahora corren en dispositivos conectados al bus CAN, capaces de solicitar un servicio. Un servidor sería cualquier entidad o agente que corre en un dispositivo y que contiene uno o más objetos CORBA activos. Debe ser capaz de recibir solicitudes de servicio, procesarlas y enviar el resultado al cliente que inició la solicitud.

Al igual que en la especificación original de CORBA, los dispositivos cliente envían a los dispositivos servidor solicitudes de servicio para invocar operaciones sobre objetos. Estas solicitudes están compuestas por un identificador de operación, un objeto destino y cero o más parámetros. Los dispositivos cliente siguen usando referencias, en el diseño para el bus CAN, para identificar los objetos a los cuales desean solicitarles servicios.

Un cambio importante en el modelo es que en el middleware propuesto, los clientes no sólo pueden solicitar servicios a objetos sino a todo un grupo de estos. Por ejemplo, un controlador podría solicitar a todo un grupo de sensores que lean un valor del proceso que ocurre en el mundo real en un momento dado. Esto se consigue agrupando todos los objetos CORBA que modelizan sensores en un único grupo. Vale aclarar que todos los objetos miembro de un grupo deben tener, al menos, una interfase en común y que los clientes, deben usar dicha interfase para solicitar servicios al grupo.

Los dispositivos servidor identifican en su instancia de ORB el objeto o el grupo al cual está destinada una solicitud utilizando el POA, que no sufrió modificaciones funcionales significativas excepto la restricción del soporte para políticas, y el MCA, un nuevo componente de la ORB que permite la administración distribuida de los grupos.

6.2 Capa de transporte

La especificación original está pensada para usar como protocolo de sesión GIOP. La capa genérica de sesión es mapeada sobre el transporte subyacente. En la especificación original de CORBA, este mapeo recibe el nombre de IIOP. En el middleware para el bus CAN propuesto en el presente trabajo, el mapeo recibe el nombre de CANIOP.

GIOP requiere que el transporte subyacente cumpla con ciertas condiciones para garantizar un correcto funcionamiento. Estas condiciones son ofrecer un servicio orientado a conexión, confiable, basado en un stream de bytes, que ofrezca notificación de pérdida de conexión y un modelo de manejo de conexiones similar al de TCP/IP, dado que TCP/IP es el transporte tenido en cuenta en la especificación original para definir el mapeo IIOP.

En el diseño propuesto, el mapeo de GIOP sobre el bus CAN recibe el nombre de CANIOP. Ofrece un servicio sin conexión con multicast y broadcast confiable, no existiendo ninguna limitación en cuanto a la cantidad de bytes a enviar. CANIOP le da al prototipo de ORB la capacidad de direccionar objetos particulares o bien, grupos de objetos mediante el

mecanismo de multicast / broadcast basado en message filtering, provisto por los adaptadores CAN. También define un mecanismo para asignar valores al campo ARBITRATION FIELD de los frames CAN.

La principal diferencia entre IIOP y CANIOP es que el primer mapeo cumple con todos los requerimientos planteados por GIOP en la especificación original de CORBA. El hecho que CANIOP no cumpla con todos los requerimientos genera la necesidad de introducir algunos cambios en GIOP para garantizar su funcionamiento.

6.3 Capa de sesión

En el diseño del prototipo se incluye un nuevo protocolo de sesión, referido como GIOP modificado. El middleware basado en CORBA sobre el bus CAN usaría este protocolo para comunicar cada una de las instancias de la ORB que corren sobre el mismo. El nuevo protocolo se basa en la especificación original, con la cual presenta algunas diferencias importantes.

La primer diferencia significativa surge del hecho que cierta funcionalidad ofrecida por CORBA y tenida en cuenta en la especificación original de GIOP, no resulta necesaria en un ambiente en los cuales se usa el bus CAN. Por este motivo se eliminan del protocolo los mensajes: LocateRequest, LocateReply, CloseConnection y CancelRequest. Los motivos por los cuales estos mensajes no resultan necesarios ya fueron expuestos en capítulos anteriores.

La segunda diferencia surge de cambios en el formato de los mensajes originales de GIOP que aparecen en GIOP modificado o sea, Request y Reply principalmente. Estos cambios fueron motivados por la necesidad de disminuir el tamaño de los mensajes de invocación y de esta forma, mejorar la eficiencia de uso del bus (el objetivo final es lograr una invocación por frame CAN).

A continuación, se incluye un ejemplo que permite comparar el tamaño de una invocación realizada mediante GIOP estándar y GIOP modificado para el bus CAN. Para esto, supóngase que se desea llamar al siguiente método:

```
module StdDevices {
    ...
    interface GenericDevice {
        void SetAttributeValueUL (
            in string attrName, in unsigned long idxVal, in unsigned long attrValue);
    };
    ...
};
```

Código 42 - Ejemplo de invocación de método en GIOP modificado

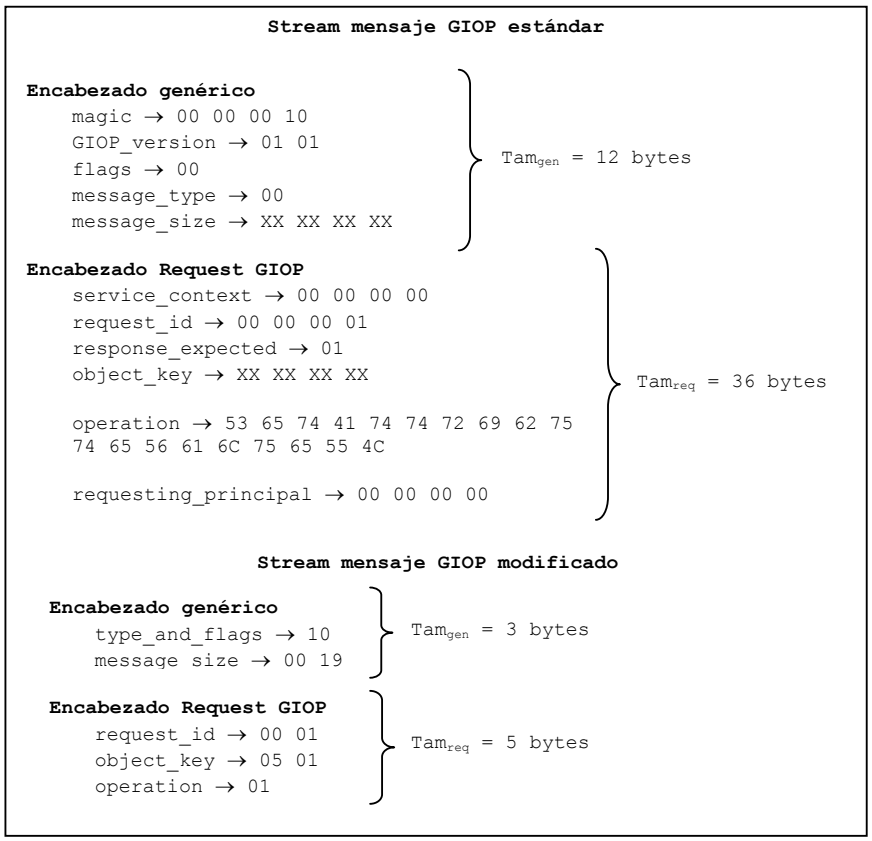


Ilustración 34 - Comparación entre mensajes de GIOP estándar y de GIOP modificado

Como se puede ver en la ilustración anterior, se consiguió disminuir el tamaño de la invocación de bytes a sólo 8 bytes. Para esto fue necesario introducir algunos cambios en la nueva versión de GIOP como usar un único byte para identificar las operaciones en lugar de todo un string con el nombre, eliminar los contextos de invocación, etc. En los dos ejemplos anteriores, no se tienen en cuenta los parámetros de la invocación.

La tercer diferencia que aparece se da en la semántica de invocación. En CORBA original se pueden invocar objetos particulares con una semántica oneway (mejor esfuerzo sin garantía de entrega) o bien, con semántica request / response. Por el contrario, en GIOP modificado los objetos particulares sólo pueden invocarse con semántica request / response mientras que los grupos de objetos, sólo con semántica oneway.

Por último, la cuarta diferencia se relaciona con limitaciones a la cantidad de objetos. Todas las modificaciones introducidas en GIOP modificado terminan limitando la cantidad de objetos que pueden ser manejados por una instancia de la ORB. Como ya se dijo, pueden existir un máximo de 65536 objetos repartidos en 32 dispositivos o lo que es igual, un máximo de 2048 objetos por dispositivo, agrupados en 32 grupos de objetos distintos.

Estas limitaciones no constituyen un problema. El diseño de ORB que se propuso en capítulos anteriores sería potencialmente usado en sistemas de control distribuido que no requieren gran cantidad de objetos. Por el contrario, la versión original de CORBA y su protocolo de sesión no presentan estos límites ya que están pensados para sistemas muy grandes con cantidades elevadas de objetos.

6.4 Capa de presentación

En el diseño presentado en el capítulo anterior, se definió una nueva capa de presentación para el prototipo de ORB para el bus CAN. Esta capa recibió el nombre de CCDR por Compact Custom Data Representation.

Uno de los principales objetivos perseguidos al definir CCDR fue disminuir la cantidad de bytes resultantes al codificar un mensaje GIOP. Para conseguir esto, en primer lugar, se omitió el requerimiento de alinear los datos a sus límites naturales en los mensajes. En segundo lugar, se modificaron parcialmente las reglas de codificación.

Las reglas de codificación de CDR siguen la filosofía usada en External Data Representation (XDR), capa de presentación de ONC RPC. Asumen que tanto el emisor como el receptor conocen el orden y el tipo de los datos a intercambiar por lo cual no es necesario enviar esta información en los mensajes. Esto es exactamente lo contrario a lo que se hace en OSI en donde las reglas trabajan con un formato Type-Length-Value (TLV).

En CCDR, se respeta la misma filosofía. Sin embargo, para disminuir el tamaño de los mensajes y aprovechando cierta forma de utilizar las variables enteras, se plantea un esquema de codificación de longitud variable para los datos numéricos de tipo entero.

Supóngase que se desea llamar al mismo método usado en la sección anterior para la comparación entre GIOP y GIOP modificado. Este es:

```
module StdDevices {
    ...
    interface GenericDevice {
        void SetAttributeValueUL (
            in string attrName, in unsigned long idxVal, in unsigned long attrValue);
    };
    ...
};
```

Código 43 - Ejemplo de método y codificación de parámetros en CCDR

Usando GIOP o GIOP modificado se codificaría la solicitud de invocación propiamente dicha. Luego, los parámetros se codificarían con CDR o con CCDR respectivamente. A continuación se incluyen los parámetros codificados para cada una de estas capas de presentación:

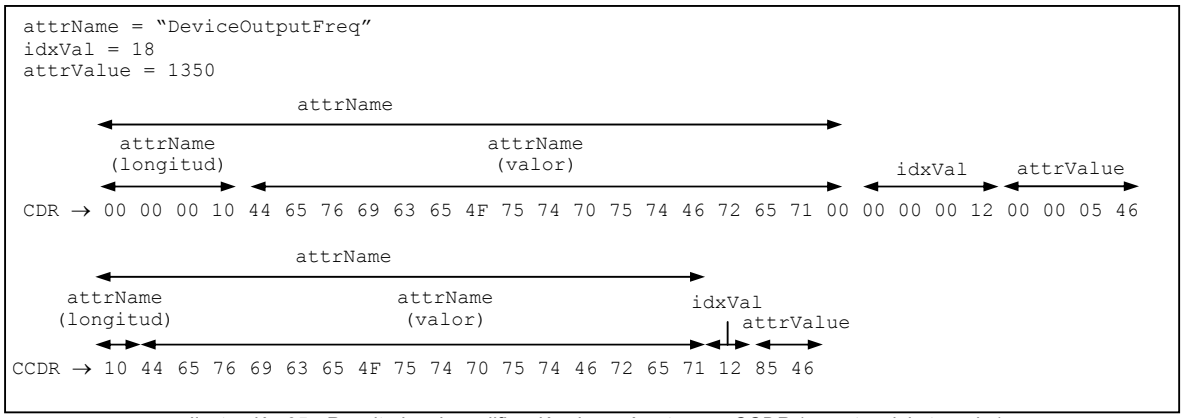


Ilustración 35 - Resultados de codificación de parámetros en CCDR (en network byte order)

Se pasó de usar 29 bytes en CDR a 20 en CCDR. Esto indicaría que los cambios introducidos en CDR son beneficiosos ya que permiten disminuir la cantidad de bytes

resultantes al codificar un mensaje GIOP. Cuantos más parámetros de tipo numérico entero se usan en una invocación, mayor es la reducción en el tamaño de los mensajes.

7. Comparación con capas de aplicación estándar

Resumen del capítulo

El nuevo middleware es comparado contra capas de aplicación para CAN, como son los estándares DeviceNet y CANopen. En la comparación, se toman en cuenta aspectos relacionados con el modelo de objetos y el modelo de comunicaciones usado por cada uno así como los protocolos de la capa de transporte, de sesión y de presentación.

Por último, tanto para DeviceNet como para CANopen, se evalúa en que forma cada uno cumple con los requerimientos para una capa de aplicación para el bus CAN [Etschberger 97].

7.1 Comparación con CANopen

7.1.1 Modelo de objetos

CANopen es una capa de aplicación orientada a objetos para el bus CAN bajo la cual, los dispositivos se describen mediante perfiles. Cada perfil define un dispositivo estándar, especificando la funcionalidad mínima que se le requiere a cualquier dispositivo que afirme adherir al perfil en cuestión.

La parte más importante de un perfil es el diccionario de objetos. Este consiste en un agrupamiento de objetos, único para cada perfil. El diccionario es accesible a través del bus CAN en una forma ordenada y predefinida, constituyendo así la interfase entre las aplicaciones y el bus. En conclusión, un dispositivo está formado por una serie de objetos agrupados en las entradas de un diccionario, las cuales pueden ser accedidas desde la red de comunicaciones.

Por otro lado, el modelo de objetos del middleware propuesto respeta prácticamente sin modificaciones el modelo definido en la especificación original de CORBA. Los objetos proveen uno o más servicios y residen en entidades llamadas servidores. Los clientes solicitan servicios a dichos objetos a través de las interfases que estos implementan, mediante el intercambio de solicitudes o requests. En el bus CAN, tanto clientes como servidores corren en dispositivos conectados al bus. Los dispositivos, bajo este modelo, se describen como una serie de objetos y de interfases que ofrecen servicios a otros dispositivos.

La primer diferencia que surge en forma evidente, entre el middleware propuesto y CANopen, se relaciona con el modelo de objetos en sí. En los dos modelos se manejan conceptos muy distintos respecto a la forma de construir aplicaciones.

El diseño propuesto plantea un modelo en el cual los dispositivos están formados por objetos, indistintamente activos o pasivos, que ofrecen servicios a sus pares. De esta forma, las aplicaciones se construyen mediante solicitudes recíprocas de servicios entre todos los objetos distribuidos en los dispositivos conectados al bus CAN.

Por el contrario, CANopen plantea un modelo en el cual los dispositivos están formados por objetos pasivos que no ofrecen servicios. Estos objetos se agrupan y ordenan en un diccionario. Las aplicaciones se construyen únicamente leyendo y escribiendo entradas en los diccionarios de objetos de cada uno de los dispositivos conectados al bus. Estas lecturas y escrituras pueden disparar cambios internos en los objetos o en el comportamiento del dispositivo.

Otra diferencia importante surge al comparar la forma de direccionar objetos. En el nuevo middleware, los objetos se direccionan mediante referencias, valores que denotan objetos en forma confiable llamados Interoperable Object References o IORs. Los clientes usan estos valores para indicar a que objetos desean solicitar cierto servicio. Estas referencias permiten que la ubicación de un objeto sea transparente para los clientes (no tiene que conocer la dirección de red del dispositivo donde corre el servidor).

En CANopen, para poder acceder a un objeto es necesario conocer, en primer lugar, la dirección de red del dispositivo donde reside. Luego, es necesario conocer el índice de la entrada en la cual se ubica en el diccionario de objetos del dispositivo correspondiente. El desarrollador de aplicaciones distribuidas tiene que conocer la dirección del dispositivo donde reside el objeto. Esto va en contra de la transparencia de distribución, la ubicación de los objetos no resulta transparente.

Por último, al comparar las herramientas para descripción de dispositivos, surge que la principal herramienta que ofrece el nuevo middleware es el lenguaje OMG IDL. Se podría usar para definir las interfases de todos los objetos que componen un dispositivo. En CANopen, el principal componente de un perfil de dispositivo es el diccionario de objetos. O sea, una lista de objetos, el tipo de los mismos y el índice de la entrada en la cual pueden ser ubicados.

7.1.2 Modelo de comunicaciones

CANopen está basado en un subconjunto de CAN Application Layer (CAL). Esta capa de aplicación para el bus CAN define un modelo de comunicaciones sin conexión, basado en el intercambio de mensajes. Se trata de un modelo de comunicaciones orientado completamente a mensajes.

Los mensajes son descriptos en términos de objetos llamados communication objects (COBs). Cada uno de estos objetos equivale a uno o más frames en el bus CAN. Los objetos son identificados mediante un COB-ID único, que se corresponde con el valor del ARBITRATION FIELD del frame CAN. Existen distintos tipos de COB entre los cuales se destacan los Service Data Objects (SDOs), los Process Data Objects (PDOs), los objetos predefinidos y los objetos administrativos.

CANopen plantea un modelo en el cual un dispositivo cliente puede acceder a otro dispositivo, llamado servidor, y a los objetos que lo componen, agrupados en el diccionario. Dichos objetos pueden ser leídos o modificados mediante la transmisión en el bus de objetos de comunicación como los SDOs, los PDOs, etc. Cada uno de estos tipos de objeto de comunicación ofrece una funcionalidad y un comportamiento distinto.

Los SDOs permiten leer y escribir cualquiera de las entradas del diccionario de objetos de un dispositivo. Corresponden a mensajes CAN de baja prioridad y permiten implementar aplicaciones bajo modelos peer-to-peer. Por su lado, los PDOs son usados para intercambiar datos del proceso que se está monitoreando y controlando. Corresponden a mensajes CAN de alta prioridad y permiten implementar aplicaciones bajo modelos productor / consumidor en modo push y pull. El formato de estos mensajes es definido por las aplicaciones usando el diccionario de objetos. Por último, los mensajes predefinidos cumplen funciones especiales como la sincronización de los dispositivos (objetos SYNC) y generar notificaciones de emergencia en forma opcional (objetos EMCY).

El middleware propuesto en este trabajo define también un modelo de comunicaciones sin conexión con un enfoque conceptual distinto. Las interacciones entre clientes y servidores se describen en términos de mensajes de solicitud de servicio y sus respectivas respuestas. Así, el modelo de comunicaciones permite a un cliente, mediante una serie de protocolos, expresar una invocación de servicio a un objeto a través de uno o más frames CAN. Estos frames llegan al servidor, anfitrión del objeto correspondiente, el cual atiende la solicitud de servicio y contesta mediante un mensaje de respuesta, expresado con los mismos protocolos que la invocación original.

Tanto CANopen como el middleware permiten utilizar un modelo de comunicaciones peer-to-peer para construir aplicaciones distribuidas. En el primer caso, se usan los SDOs para tal fin. Estos ofrecen un servicio sin conexión con confirmación o acknowledgement. En el segundo caso, se usa un protocolo request / response para invocar métodos de objetos remotos. Este protocolo también ofrece un servicio sin conexión con confirmación o acknowledgement. En ambos casos, los mensajes CAN usados para este tipo de comunicación tienen prioridad baja.

De igual forma, las dos capas proveen mecanismos para intercambiar en tiempo real datos del proceso monitoreado. En CANopen, esto se consigue construyendo modelos master / slave y productor / consumidor en sus variedades push y pull, usando PDOs y objetos administrativos. En el nuevo middleware, se soportan los modelos master / slave y productor / consumidor en su variante push. Ambos son construidos usando el mecanismo de invocación a grupos de objetos que, al igual que los PDOs de CANopen, ofrece un servicio sin conexión, sin confirmación y de alta prioridad.

Aparte de las diferencias conceptuales existentes entre ambos modelos, existen una serie de diferencias referidas a la funcionalidad provista por cada capa de aplicación. A continuación, se detallan dichas diferencias.

En primer lugar, en la nueva propuesta de CORBA sobre CAN no se soportan mensajes con formato libre para intercambio de datos de proceso. Todos los mensajes corresponden a invocaciones de operaciones de objetos o grupos de objetos, generados usando el protocolo de sesión GIOP modificado y codificados mediante CCDR. En CANopen, el contenido completo del campo DATA FIELD de un PDO es determinado por la aplicación a partir de los mapeos cargados en el diccionario de objetos. Como se verá más adelante, DeviceNet también ofrece esta posibilidad.

En segundo lugar, surgen diferencias respecto a la asignación de valores para el ARBITRATION FIELD. En el middleware, la asignación de estos valores está definida por el protocolo CANIOP. Depende del tipo de comunicación (peer-to-peer o master / slave), del grupo de objetos destinatario de la invocación y del dispositivo transmisor. Por el contrario, en CAL, base de CANopen, se asignan COB-IDs con un algoritmo distribuido implementado por entidades llamadas Identifier Distributors (DBTs) que garantiza la unicidad de las asignaciones. En CANopen se simplifica el mecanismo de CAL y el valor del ARBITRATION FIELD se determina en base al tipo de objeto que se desea transmitir y al identificador del dispositivo transmisor.

Por último, CANopen presenta objetos de comunicación con funcionalidades especiales como los objetos EMCY y SYNC. En el middleware propuesto no se define ningún mecanismo estándar que permita una sincronización entre objetos como la que ofrecen los objetos SYNC. Esta funcionalidad podría ser reemplaza por algún algoritmo de sincronización distribuido, implementado mediante invocaciones recíprocas entre objetos CORBA.

7.1.3 Capa de transporte

En CANopen, los servicios de transporte varían según el tipo de COB que se esté enviando. Para cada tipo de objeto se ofrecen distintos tipos de servicio (con confirmación o sin confirmación, con fragmentación o sin fragmentación, etc.).

Los SDOs ofrecen un servicio confiable sin conexión con confirmación. Hay distintos tipos de transferencia para este tipo de objetos: la transferencia expédita y la transferencia en segmentos. La primera se usa para mensajes cortos a los cuales no se aplica fragmentación. Implementa una disciplina de línea end-to-end mediante un mecanismo de stop-and-wait. La segunda se usa para transmitir mensajes mayores, aplicándose un protocolo de fragmentación y reensamble, recibiéndose una confirmación por cada segmento. Esto termina funcionando también como una disciplina de línea end-to-end de tipo stop-and-wait ya que no se envía un nuevo segmento hasta recibir la confirmación del anterior.

Por el contrario, los PDOs ofrecen un servicio de transferencia de datos sin conexión, sin confirmación y por lo tanto, no confiable. Esto es así porque están pensados para permitir intercambiar datos del proceso controlado en tiempo real.

En el prototipo, las funciones de la capa de transporte son llevadas a cabo por el protocolo CANIOP. Este no ofrece una disciplina de línea ni de fragmentación como el ofrecido por los SDOs en CANopen. Al no haber confirmación por fragmento, un cliente no sabe que un servidor recibió una solicitud de servicio hasta que recibe la respuesta correspondiente de vuelta. En forma similar, un servidor no puede determinar nunca si el cliente recibió o no la respuesta. En conclusión, tanto para modelos de comunicación tipo peer-to-peer como productor / consumidor, CANIOP ofrece un servicio sin conexión, sin confirmación y por lo tanto, no confiable. Se trata claramente de un servicio de mejor esfuerzo.

El protocolo CANIOP se identifica como la capa de transporte del prototipo porque es el responsable de la transferencia end-to-end de los mensajes. Define un mecanismo de asignación de valores para el ARBITRATION FIELD de los frames CAN que junto con el multicast y la técnica de Message Filtering, garantizan que un request llegué al objeto o al grupo de objetos al cual está destinado. Para conseguir esto, el ARBITRATION FIELD se divide en tres campos usados para indicar si el frame va a un objeto en particular (peer-to-peer) o a

un grupo de objetos (master / slave o productor / consumidor), y para identificar el dispositivo o el grupo de objetos destinatario del mensaje. Se reservan 5 bits para identificar a cada dispositivo por lo cual, en un bus CAN que use el middleware propuesto como capa de aplicación, pueden conectarse un máximo 32 dispositivos distintos.

En CANopen, se sigue un esquema de alocación de identificadores orientado a los dispositivos. El ARBITRATION FIELD de un frame CAN se divide en un campo usado para indicar el tipo de COB (SDO, PDO, EMCY, SYNC, NMT, etc.) y otro para identificar el dispositivo transmisor. Se reservan 7 bits para estos identificadores por lo cual se pueden direccionar un máximo de 128 dispositivos en una misma red. Las prioridades se reparten por tipo de COB, en forma uniforme entre los dispositivos.

Surge, en forma inmediata, que CANopen soporta mayor cantidad de dispositivos que el middleware. Por otro lado, en CANopen las prioridades se asignan dependiendo del COB que se desea transmitir. En el prototipo, las comunicaciones peer-to-peer tienen menor prioridad que las master / slave o productor / consumidor. Dentro de este último grupo, las prioridades se reparten entre los grupos de objetos. O sea, ciertos grupos tienen mayores prioridades que otros.

Por último, CANopen ofrece una serie de disparadores o triggers para la transmisión de PDOs. Esto permite mayor flexibilidad a la hora de publicar o producir datos sobre el proceso que se está monitoreando. Algunos posibles disparadores son un evento o un timer que expira, una solicitud remota, el comienzo de una ventana de sincronismo (disparada por la recepción de un objeto SYNC) o el cumplimiento de cierto período o ciclo de transmisión. DeviceNet ofrece una serie de conexiones de I/O predefinidas en las cuales, la transmisión es iniciada prácticamente por estos mismos disparadores.

En el nuevo middleware no se ofrece una funcionalidad similar en forma directa. Sin embargo, mediante un diseño adecuado de interfases de objetos y un aprovechamiento del servicio de comunicación en grupo, es posible construir esquemas de comunicación similares a los planteados por los triggers presentados anteriormente.

7.1.4 Capa de sesión

La capa de sesión permite organizar el diálogo entre las entidades de las capas superiores. En el middleware propuesto, la capa de sesión es implementada por un protocolo llamado GIOP modificado. Mediante un mecanismo de intercambio de mensajes de solicitud y sus respectivas respuestas, este protocolo permite a los clientes invocar las operaciones o métodos de los objetos residentes en los servidores.

En CANopen se sigue un modelo de objetos bajo el cual, los mensajes intercambiados entre dispositivos son descriptos en términos de objetos de comunicación o COBs. Los dos tipos principales de COBs son los Service Data Objects (SDOs) y los Process Data Objects (PDOs). Para ambos tipos de objetos se definen múltiples servicios (o protocolos) que permiten acceder, ya sea en forma directa o indirecta, a las entradas del diccionario de objetos de un dispositivo. Leer o escribir una entrada en uno de estos diccionarios, equivale a invocar un método de un objeto en el middleware aquí propuesto, usando el protocolo GIOP modificado.

Los SDOs son usados para el intercambio de datos no relevantes para el control y monitoreo del proceso en si. Se mapean a frames CAN de baja prioridad. Permiten implementar modelos de comunicación peer-to-peer. Al igual que en el prototipo, los frames CAN usados para este tipo de modelos presentan prioridades más bajas que los usados en modelos master / slave o productor / consumidor.

Se definen una serie de servicios de tipo solicitud y respuesta, como en GIOP modificado, que se pueden aplicar a los SDOs para su transferencia entre dispositivos. Dependiendo de la cantidad de datos a intercambiar, se puede usar una transferencia expédita, sin fragmentación, o una transferencia en segmentos o fragmentos. Para cada tipo de transferencia, se usan servicios o protocolos distintos, incluyéndose en el frame CAN, para el

segundo tipo de transferencia, información de control de flujo. A diferencia, GIOP modificado no incluye información de control de flujo ni confirmación por fragmentos, como ofrece CANopen a nivel de sesión.

Los SDOs permiten un acceso directo al diccionario de objetos. Cada objeto de este tipo, intercambiado usando cualquiera de los protocolos y de los tipos de transferencia antes descriptos, incluye el índice y el subíndice de la entrada del diccionario de objetos que pretende leer o modificar. En forma similar, cada mensaje GIOP de solicitud incluye el identificador del objeto que se está invocando.

A diferencia de los SDOs, los PDOs son usados para el intercambio de datos obtenidos del proceso y por lo tanto, relevantes para el control y el monitoreo del mismo. Constituyen tráfico de tiempo real por lo cual, se mapean a frames CAN de alta prioridad. Permiten implementar modelos de comunicación master / slave o productor / consumidor. En el middleware propuesto, los frames CAN usados para este tipo de modelos, correspondientes a invocaciones a grupos de objetos, también presentan prioridades altas en comparación con las invocaciones a objetos particulares.

Los PDOs proveen únicamente dos servicios, escribir datos y leer datos. El primero se implementa mediante un DATA FRAME mientras que para el segundo, se usa un REMOTE FRAME de CAN. Estos dos servicios permiten implementar el modelo de comunicaciones productor / consumidor en sus variantes push y pull. El nuevo middleware no hace uso de REMOTE FRAMES por lo cual, sólo la variante push está soportada.

Los PDOs permiten un acceso indirecto al diccionario de objetos. Únicamente el emisor y los receptores de un PDO conocen el formato de los datos incluidos en el DATA FIELD del frame CAN correspondiente. Cada receptor, en su diccionario de objetos, contiene una entrada en la cual se define el formato del PDO y un mapeo de los datos contenidos en el mismo a otras entradas del diccionario. Por este motivo, se habla de acceso indirecto.

Por último, CANopen presenta una serie de COBs adicionales que permiten la sincronización de los dispositivos y la generación de notificaciones de emergencia. Estos objetos son los mensajes de emergencia (EMCY), los mensajes de sincronización (objeto SYNC) y los mensajes de time-stamp. Ninguna de estas funcionalidades adicionales es provista en forma directa por el prototipo y debe ser implementada por el desarrollador de aplicaciones distribuidas, en caso de ser necesaria.

7.1.5 Capa de presentación

En CANopen las funciones de la capa de presentación son realizadas, en forma conjunta, por el diccionario de objetos y CMS (CAN Message Specification). El diccionario de objetos cumple la función de sintaxis abstracta, usada para describir los datos que serán intercambiados mediante SDOs y PDOs. Por su lado, CMS especifica las reglas de codificación para los objetos de dicho diccionario.

En el nuevo middleware basado en CORBA las funciones de la capa de presentación, llamada Compact Common Data Representation, son llevadas a cabo por el lenguaje OMG IDL y una serie de reglas de codificación. OMG IDL hace las veces de sintaxis abstracta para describir los datos intercambiados entre clientes y servidores.

Ambas capas se basan en un modelo que asume que las dos partes involucradas en una comunicación conocen de antemano el orden de transmisión de los datos y el tipo particular de cada uno. Así se evita la utilización de un modelo de codificación tipo TLV (Tipo, longitud y valor).

Al comparar las sintaxis de transferencia de ambas capas de presentación, surge, en primer lugar, que las sintaxis abstractas usadas por cada una son distintas. Mientras que en el middleware propuesto se usa el lenguaje OMG IDL en CANopen, el diccionario de objetos actúa directamente como descripción de los datos a intercambiar. En segundo lugar, se

observa que CMS y las reglas de codificación usadas en el prototipo son bastante parecidas excepto por algunas diferencias.

En primer lugar, en CANopen, de acuerdo a las reglas de codificación especificadas por CMS, las variables enteras de más de un byte se codifican siguiendo un ordenamiento Least Significant Byte (LSB) o little endian. En el diseño propuesto, al contrario de CANopen, este tipo de variables son codificadas siguiendo un ordenamiento Most Significant Byte (MSB) o big endian.

Por último, el prototipo usa una codificación de longitud variable para datos de tipo entero. Al usar una cantidad menor de bytes para aquellos valores enteros más pequeños, se pretende conseguir, en promedio, mensajes más cortos. Esto es así porque se asume que estos son los valores más comúnmente usados. A diferencia de CCDR, en CMS se usa una codificación de longitud fija. O sea, la cantidad de bytes a usar para codificar variables enteras, con o sin signo, está siempre determinada por el tipo de estas.

7.1.6 Cumplimiento de requerimientos

A continuación se analiza la forma en que CANopen cumple con los requerimientos para una capa de aplicación para el bus CAN [Etschberger 97]. Los requerimientos son:

- **Asignación de identificadores**: bajo CANopen, los valores del ARBITRATION FIELD quedan determinados por el tipo de objeto que se desea transmitir (SDO, PDO, EMCY, SYNC, objetos predefinidos, etc.) y por el identificador del dispositivo transmisor. Así, el espacio de posibles valores queda dividido en grupos de identificadores para cada tipo de COB, cada uno con distintas prioridades. Por ejemplo, los frames usados para PDOs tienen mayor prioridad que los usados para SDOs.

- **Método de comunicación peer-to-peer**: los Service Data Objects o SDOs permiten implementar aplicaciones bajo modelos de comunicación peer-to-peer. Este tipo de COB ofrece acceso a las distintas entradas del diccionario de objetos de cada dispositivo. Son usados para intercambiar cantidades grandes de información, no relevante en forma directa para el monitoreo y control del proceso.

- **Método de intercambio de datos del proceso**: los Process Data Objects o PDOs permiten implementar aplicaciones bajo modelos de comunicación master / slave y productor / consumidor en sus variantes push y pull. Estos objetos son usados para intercambiar datos, obtenidos del proceso monitoreado, en tiempo real. Para facilitar el cumplimiento de restricciones temporales, se usan frames CAN con prioridades altas.

- **Administración de la red**: como se vio en capítulos previos, CANopen ofrece los objetos de comunicación NMT. Estos facilitan la inicialización, configuración y manejo de errores en el bus CAN. También están disponibles objetos de tipo LMT, que permiten leer y configurar ciertos parámetros de CAL y de CAN.

- **Principios de modelado de dispositivos y definición de perfiles**: se basa en la definición de perfiles de dispositivos, formados principalmente por la descripción de un diccionario de objetos. Un diccionario consiste en un agrupamiento de objetos, único para cada perfil de dispositivo, accesible a través del bus CAN en una forma ordenada y predefinida.

7.2 Comparación con DeviceNet

7.2.1 Modelo de objetos

Tanto DeviceNet como el middleware propuesto en el presente trabajo constituyen capas de aplicación para el bus CAN orientadas a objetos. El principal uso que se le da al modelo de objetos es la descripción de los dispositivos conectados al bus, los servicios que estos ofrecen y su comportamiento.

El modelo de objetos del prototipo respeta casi sin modificaciones el modelo descripto en la especificación original de CORBA [Corba Core 02]. Los objetos son vistos como entidades abstractas e identificables que proveen uno o más servicios, residiendo en servidores. Por otro lado, estos mismos objetos implementan o adhieren a una o más interfases, las cuales son descripciones de los servicios que un cliente puede solicitar. En fin, los clientes solicitan servicios a los objetos que residen en los servidores a través de las interfases que estos últimos implementan, mediante el intercambio de solicitudes o requests.

Al llevar el modelo general al bus CAN, surge que tanto clientes como servidores (y los objetos correspondientes) corren en dispositivos conectados al bus. El intercambio de solicitudes o requests se lleva a cabo a través de la instancia de la ORB que funciona en cada uno de los nodos, mediante una serie de protocolos de transporte, de sesión y de presentación que en el nuevo middleware son CANIOP, GIOP modificado y CCDR respectivamente. Los dispositivos, bajo este modelo, se describen como una serie de objetos y de interfases que ofrecen servicios a otros dispositivos.

En DeviceNet, cada dispositivo está formado por una serie de objetos abstractos los cuales, no necesariamente se corresponden con objetos físicos existentes en la implementación. Estos objetos abstractos pertenecen a clases, definidas por DeviceNet como conjuntos de objetos que representan el mismo tipo de componente con los mismos atributos y servicios y con un comportamiento similar. Se definen también servicios comunes que deben ser implementados por los objetos de todas las clases.

Al comparar los modelos de objetos de ambas capas de aplicación surgen una serie de diferencias. La primera es que en DeviceNet, un objeto pertenece a una clase mientras que en el middleware propuesto, un objeto tiene únicamente tipo, el cual es determinado por las múltiples interfases a las que adhiere. Una clase es una generalización de los atributos, los servicios y el comportamiento que tienen en común un conjunto de objetos. Una interfase es simplemente una descripción de los servicios ofrecidos. En cierta forma, el concepto de clase es más abarcativo y determina con mayor precisión lo que hace un objeto.

Otra diferencia importante surge al comparar la forma de direccionar objetos en ambos modelos. En el prototipo al igual que en CORBA, los objetos se direccionan mediante referencias, valores que denotan objetos en forma confiable llamados Interoperable Object References o IORs. Los dispositivos que actúan como clientes usan estos valores para indicar a que objetos desean solicitar cierto servicio. Estas referencias permiten que la ubicación de un objeto sea transparente para el desarrollador de aplicaciones distribuidas (se obtiene transparencia de distribución [Tanenbaum 02]).

Por el contrario, en DeviceNet se plantea un direccionamiento de objetos basado en el uso de identificadores para cada elemento (dispositivo, clase, instancia, etc.) que resulta en un esquema no plano con múltiples niveles de indirección. Para poder acceder a un objeto en particular es necesario conocer el identificador del dispositivo donde reside, de la clase a la que pertenece y por último, de la instancia dentro de dicha clase. La necesidad de que el desarrollador de aplicaciones distribuidas conozca la dirección del dispositivo donde reside el objeto va en contra de la transparencia de distribución.

Respecto a la descripción de dispositivos, DeviceNet se basa en la definición de perfiles. Estos facilitan la interoperabilidad y la mantenibilidad, gracias a la intercambiabilidad

de dispositivos. Cada perfil está formado por un modelo de objetos, una especificación de formato de datos de I/O y el mecanismo de configuración del dispositivo. El primer elemento es una descripción de todos los objetos usados para modelar el dispositivo, el comportamiento de estos y los servicios que ofrecen. El segundo elemento es necesario por las conexiones de I/O ofrecidas por DeviceNet, que no imponen ninguna restricción sobre el formato de los datos enviados. Es necesario definir que datos se intercambian en cada tipo de conexión de I/O. Por ultimo, hay que definir el mecanismo de configuración del dispositivo.

El middleware propuesto no da muchos detalles respecto a la descripción de dispositivos. La principal herramienta que ofrece es el lenguaje OMG IDL, que permite definir las interfases de todos los objetos que componen un dispositivo. Comparando contra un perfil DeviceNet, surge que un modelo de objetos va a ser necesario también para especificar mínimamente los objetos CORBA que residen en un dispositivo. Respecto al formato de datos de I/O, en el prototipo esta información no es necesaria porque, para todos los intercambios de datos, se definen formatos (a través de CANIOP, GIOP modificado y CCDR). No hay un mecanismo de comunicación similar a las conexiones de I/O con formato libre de DeviceNet. Por último, la configuración debería realizarse también a través de interfases definidas con OMG IDL y no mediante un mecanismo especial.

7.2.2 Modelo de comunicaciones

DeviceNet ofrece un modelo de comunicaciones con conexión. Este modelo permite manejar dos tipos de conexión con distinta funcionalidad y distinto comportamiento. Ambos tipos de conexiones pueden establecerse en forma dinámica o bien, usarse alguna conexión predefinida.

En primer lugar, se encuentran las conexiones de I/O. Estas son conexiones dedicadas de propósito específico entre dispositivos productores y uno o más consumidores. Permiten trabajar en un modo master / slave o productor / consumidor e intercambiar datos del proceso que está siendo monitoreado por el sistema distribuido correspondiente. No se impone ninguna restricción sobre el formato de los datos enviados usando estas conexiones.

En segundo lugar, están las conexiones de mensajería explícita. Se trata de conexiones genéricas de propósito múltiple, establecidas entre dos dispositivos que intercambian mensajes en forma de solicitudes y respuestas. El formato de estos mensajes está especificado por el estándar de DeviceNet. Este tipo de conexiones permite implementar modelos de comunicación peer-to-peer.

Para el establecimiento dinámico de conexiones de mensajería explícita se define un protocolo especial que utiliza valores específicos en al ARBITRATION FIELD de los frames CAN. Al establecer una conexión, las partes involucradas negocian los valores a usar en el ARBITRATION FIELD de acuerdo al tipo de servicio que quieren. Dicho valor recibe el nombre de Connection ID.

Al contrario de DeviceNet, el prototipo se basa en un modelo de comunicación mucho más simple, sin conexión, que soporta modelos de comunicación peer-to-peer y también master / slave o productor / consumidor. En el primer modelo se trabaja con una mecánica de request / response mientras que en el segundo, se trabaja con una semántica de invocación oneway de mejor esfuerzo, explotando las características de multicast / broadcast del bus CAN.

No existen mensajes con formato libre como los usados en las conexiones de I/O de DeviceNet. Todos los mensajes corresponden a invocaciones de operaciones de objetos CORBA, generados usando el protocolo de sesión GIOP modificado y codificados mediante CCDR.

Al no haber establecimiento de conexiones, el valor del ARBITRATION FIELD a usar no puede negociarse entre las partes involucradas como en DeviceNet. La asignación de estos valores en el diseño propuesto está definida por el protocolo CANIOP. Depende del tipo de

comunicación (peer-to-peer o master / slave), del grupo de objetos destinatario de la invocación, del dispositivo transmisor, etc.

7.2.3 Capa de transporte

En DeviceNet, en las conexiones de mensajería explícita, los servicios de transporte se implementan usando el primer byte de cada frame CAN. Se incluye un bit para indicar si se va a usar el protocolo de fragmentación y un bit para el identificador de transacción. El protocolo de fragmentación permite la reconstrucción confiable de los mensajes fragmentados en el dispositivo de destino. El identificador de transacción permite implementar un mínimo mecanismo de disciplina de línea. Sólo se permite un mensaje pendiente de confirmación, un único bit es suficiente.

Las funciones de la capa de transporte son llevadas a cabo en el nuevo middleware por CANIOP. A diferencia de DeviceNet, CANIOP no ofrece una disciplina de línea ni un mecanismo de fragmentación. El identificador de transacción de un mensaje de DeviceNet permite hacer coincidir requests con respuestas. Esto se consigue en el prototipo directamente usando un campo de los mensajes de GIOP modificado.

CANIOP se asocia con la capa de transporte porque es el responsable de la transferencia end-to-end de los mensajes. Para conseguir esto se define un mecanismo de asignación de valores para el ARBITRATION FIELD de los frames CAN que junto con el multicast y la técnica de Message Filtering, garantizan que un request llegué al objeto o al grupo de objetos al cual está destinado.

CANIOP únicamente reserva 5 bits para identificar a cada dispositivo. Por el contrario, DeviceNet reserva 6 bits para el identificador de dispositivo o MAC ID. Esto hace que en un bus CAN con DeviceNet se puedan conectar hasta 64 dispositivos mientras que si se usa el middleware propuesto en el presente trabajo, 32 dispositivos es el máximo permitido.

El conjunto de conexiones predefinidas en DeviceNet incluye la I/O Bit-Strobe Connection, la I/O Poll Connection y la I/O Change Of State/Cyclic Connection. Cada tipo de conexión presenta una utilidad y un comportamiento distintos. A continuación, se verá como se puede obtener la misma funcionalidad usando el middleware propuesto en el presente trabajo.

El primer tipo de conexión permite mover rápidamente, entre un master y los bit-strobed slaves, pequeñas cantidades de datos del proceso controlado. En el prototipo, esto puede implementarse mediante la invocación de un método por parte del master a un grupo de objetos residentes en los slaves. Los slaves contestan invocando un método de un objeto recolector de datos residente en el master

El segundo tipo de conexión, I/O Poll Connection, permite mover cualquier cantidad de datos entre dos dispositivos. Se trata de una conexión punto a punto y no multipunto como la anterior. Esta funcionalidad puede implementarse invocando directamente un método de un objeto residente en otro dispositivo y pasándole todos los datos necesarios como parámetro del método. Los datos son serializados y codificados usando el protocolo CCDR.

Por último, en la I/O Change Of State/Cyclic Connection la transmisión se dispara cuando se produce un cambio de estado en la aplicación o cuando expira un timer. El trigger es configurable al momento de establecer la conexión. Este comportamiento es fácilmente reproducible utilizando el nuevo middlware ya que hay que invocar un método para transmitir datos al dispararse cierto trigger (timer, cambio de estado de algún objeto, etc.). Aparte, este tipo de conexión hace un acknowledgement de los datos enviados. En el prototipo, esta confirmación puede hacerse también mediante invocaciones en ambos sentidos.

7.2.4 Capa de sesión

En DeviceNet, el protocolo de sesión usado en las conexiones de mensajería explícita recibe el nombre de Control and Information Protocol o CIP. En el prototipo, el protocolo de sesión se llama GIOP modificado (General Inter-ORB Protocol).

Los mensajes de solicitud y respuesta en uno y otro caso son bastante similares. Incluyen un identificador del objeto al cual están destinados, un identificador del servicio o del método que se está invocando y los parámetros correspondientes. DeviceNet también incluye un identificador de clase porque la solicitud puede estar dirigida a la clase en sí y no a una de sus instancias. Por su lado, GIOP modificado incluye un identificador del request que el servidor debe enviar en la respuesta de forma de poder asociar cada request con su respectiva respuesta.

Un objeto o una clase en DeviceNet pueden ofrecer hasta 128 servicios o métodos distintos. Esto es así porque se reservan únicamente 7 bits en los mensajes CIP. Luego, dependiendo de si se usan 1 o 2 bytes para los identificadores, se pueden manejar hasta 65536 clases y 65536 objetos. Por su lado, un objeto CORBA, bajo el diseño de middleware propuesto, puede ofrecer hasta 256 servicios o métodos distintos. Pueden existir un máximo de 65536 objetos repartidos en 32 dispositivos o sea, un máximo de 2048 objetos por dispositivo. Los objetos pueden pertenecer a 32 grupos distintos.

7.2.5 Capa de presentación

La capa de presentación usada en DeviceNet, Compact Encoding, es bastante similar a la propuesta para el prototipo basado en CORBA. Esta última recibió el nombre de Compact Common Data Representation.

Ambas capas se basan en la utilización de un modelo que supone que las dos partes involucradas en una comunicación, emisor y receptor, conocen a priori los datos a intercambiar. Esto implica conocer el orden de transmisión de los mismos y el tipo particular de cada uno. De esta forma se evita tener que apelar a un modelo tipo TLV (Tipo, longitud y valor), que requiera transmitir esta información para cada dato intercambiado. Vale aclarar que tanto en DeviceNet como en el prototipo, para ciertos tipos de dato, es necesario transmitir la longitud. Por ejemplo, para strings o arrays.

Las sintaxis abstractas usadas por las capas comparadas, en sus respectivas sintaxis de transferencia, son distintas. Mientras que DeviceNet hace uso de ASN.1, de acuerdo al modelo OSI, en el middleware propuesto directamente se usa el lenguaje OMG IDL. Este último, al definir las interfases y las operaciones que estas ofrecen, termina definiendo el tipo y el orden de los datos a intercambiar (en este caso, entre cliente y servidor). Por otro lado, se puede decir, en líneas generales, que las reglas de codificación son bastante parecidas excepto por dos pequeñas diferencias.

En primer lugar, en DeviceNet, de acuerdo a lo definido por Compact Encoding, las variables de tipo entero que ocupan más de un byte son codificadas siguiendo un ordenamiento Least Significant Byte (LSB) o little endian. Por el contrario, en el prototipo propuesto, de acuerdo a lo especificado por CCDR, este tipo de variables son codificadas siguiendo un ordenamiento Most Significant Byte (MSB) o big endian.

En segundo y último lugar, el middleware propuesto usa una codificación de longitud variable para los datos de tipo entero con o sin signo. CCDR usa una cantidad menor de bytes para aquellos valores más pequeños, pretendiendo de esta manera ahorrar bytes al suponer que estos son los valores más comúnmente usados. Esto hace que la longitud de ciertos datos dependa no sólo de su tipo sino también, de su valor. A diferencia de CCDR, en Compact Encoding se usa una codificación de longitud fija en la cual, la cantidad de bytes para datos enteros está siempre dada por el tipo de los mismos y no por su valor.

7.2.6 Cumplimiento de requerimientos

En esta sección se analiza la forma en que DeviceNet cumple con los requerimientos para una capa de aplicación para el bus CAN [Etschberger 97]:

- **Asignación de identificadores**: el espacio de posibles valores se divide en cuatro grupos de mensajes. Existen grupos en los cuales los valores se asignan de forma de repartir equitativamente las prioridades de transmisión entre los dispositivos. En otros grupos, tiene mayor prioridad para transmitir el dispositivo con la menor MAC ID. Al momento de establecer conexiones, se negocian entre las partes involucradas el grupo que se va a usar y el valor dentro de dicho grupo.

- **Método de comunicación peer-to-peer**: las conexiones de mensajería explícita permiten implementar aplicaciones bajo modelos de comunicación peer-to-peer. El protocolo de sesión CIP usado sobre estas conexiones permite acceder a clases y objetos, invocando los servicios ofrecidos.

- **Método de intercambio de datos del proceso**: las conexiones de I/O permiten implementar aplicaciones bajo modelos de comunicación master / slave o productor / consumidor. La especificación de DeviceNet no define ningún formato para los datos intercambiados por este tipo de conexiones.

- **Administración de la red**: DeviceNet ofrece protocolos que permiten recuperar un nodo que se encuentra en el estado BUS OFF. Por otro lado, también define un protocolo que permite detectar dispositivos con direcciones o MAC IDs duplicadas.

- **Principios de modelado de dispositivos y definición de perfiles**: se basa en la definición de profiles, formados por un modelo de objetos, una especificación de formato de datos de I/O y el mecanismo de configuración del dispositivo.

8 Conclusión

Partiendo de los requerimientos para los sistemas computacionales futuros, el presente trabajo planteó la conveniencia del uso de cierta arquitectura de software para su construcción. Estos requerimientos apuntan principalmente a la confiabilidad de los sistemas y al cumplimiento de restricciones temporales.

En un sistema distribuido, al estar éste compuesto por múltiples máquinas, no existen puntos únicos de falla. Así, los sistemas construidos siguiendo una arquitectura distribuida presentan mayor confiabilidad. Por este motivo, una arquitectura distribuida es la mejor opción para la construcción de los sistemas computacionales futuros ya que facilitaría el cumplimiento del primer requerimiento.

Como se vio en secciones anteriores, hay muchas herramientas a la hora de construir sistemas distribuidos. El presente trabajo se concentró en analizar las características de la herramienta más conveniente, el middleware. Algunas de las ventajas que éste presenta, aparte de la confiabilidad inherente a los sistemas distribuidos, son la interoperabilidad, la transparencia, la transparencia de distribución, la escalabilidad y la abstracción.

El segundo de los requerimientos, de los cuales parte el presente trabajo, apunta al cumplimiento de restricciones temporales. Habiendo ya asumido que lo más conveniente es seguir una arquitectura distribuida, surge la necesidad de utilizar un sistema de comunicación entre las unidades de distribución. Dadas las restricciones temporales, los buses de campo o buses industriales resultan ser los sistemas de comunicación más adecuados. Estos deben proveer a las aplicaciones distribuidas seguridad, disponibilidad y dependabilidad; mantenibilidad y flexibilidad; modularidad y capacidad de evolución y por último, interoperabilidad e intercambiabilidad.

El presente trabajo pretende usar un bus industrial como sistema de comunicación subyacente a un sistema de control distribuido. Este sistema debería ser implementado, utilizando como herramienta de construcción, algún tipo de middleware. Se espera que de esta combinación, resulte un sistema que cumpla mejor con los requerimientos temporales y de confiabilidad antes planteados. De igual forma, se espera que se preserven en la combinación resultante las características ventajosas que presentan, en forma independiente, tanto los buses industriales como los sistemas distribuidos y el middleware.

Con mayor detalle, se puede decir que el problema que se pretendió resolver fue diseñar y construir un prototipo de una capa de aplicación para un bus industrial. Dentro de todos los posibles buses industriales se decidió trabajar con CAN por múltiples razones, entre las cuales se destacan: la existencia de una especificación estándar, una amplia adopción en diversos ambientes de aplicación así como una amplia disponibilidad de hardware, una técnica de arbitración del medio compartido que facilita la comunicación en grupo y por último, la libertad que se da al implementador de aplicaciones para seleccionar una capa de aplicación. Este último punto fue el que, principalmente, motivó y permitió el planteo del problema antes descrito.

El diseño de la capa de aplicación se basó en la herramienta más conveniente para la construcción de sistemas distribuidos, como es el middleware. Dentro de los múltiples tipos de middleware que existen, se optó por un Object Request Broker (ORB), dado su enfoque basado en objetos, conveniente para modelar procesos reales. Así, se tomó como punto de partida un diseño existente de ORB, adaptándolo y modificándolo para su correcto funcionamiento como capa de aplicación del bus CAN.

De todas las implementaciones existentes de ORB, se optó por CORBA como base del diseño de la capa de aplicación en cuestión. Esta decisión fue motivada por algunas características particulares de esta ORB como por ejemplo, la existencia de una especificación estandarizada, el no ser una solución propietaria, la interoperabilidad entre fabricantes, amplia

disponibilidad de implementaciones de código abierto para plataformas diversas, soporte para múltiples lenguajes de implementación, etc.

El presente trabajo analizó, a lo largo del capítulo 2, las características generales de los buses industriales y en particular, las características del bus CAN. Luego, en el siguiente capítulo, se describió el middleware como herramienta para la construcción de sistemas distribuidos y se presentó en detalle la arquitectura de CORBA. A lo largo del capítulo 5 se presenta la solución al problema planteado o sea, se describe el diseño del middleware basado en CORBA, propuesto como capa de aplicación para el bus CAN.

El otro objetivo del trabajo es la construcción de un prototipo de la capa de aplicación. Para cumplir con este punto, se tomó una implementación existente de código abierto de CORBA (ORBit versión 0.5.13 [ORBit 04]) y se la modificó para seguir el diseño propuesto en el capítulo 5. En el apéndice se describe la implementación del prototipo y del simulador del bus CAN así como la organización del código fuente, la secuencia de pasos a seguir para su compilación y los requerimientos de software que se deben cumplir para obtener un correcto funcionamiento del mismo.

Al analizar el diseño propuesto y el prototipo construido a partir de este, se ve se cumplen los requerimientos generales para un bus industrial, planteados por Thomesse y Babiuch [Thomesse 99] [Babiuch 00]:

- **Seguridad, disponibilidad y dependabilidad**: el requerimiento de seguridad apunta a la confiabilidad general de un sistema. Al trabajarse con sistemas distribuidos, se evitan los puntos únicos de falla lo cual, permite incrementar la confiabilidad. El prototipo contempla la posibilidad de errores de transmisión durante una invocación (explota el mecanismos de multicast confiable del bus CAN). Es capaz de recuperarse de estos potenciales errores mediante retransmisiones.

- **Mantenibilidad y flexibilidad**: el prototipo no prevee protocolos específicos para mantenimiento del bus, a diferencia de DeviceNet (protocolo de detección de MAC IDs duplicadas y Offline Connection Set) y CANopen (NMT y LMT). Sin embargo, un diseñador de aplicaciones podría definir alguna interfase de mantenimiento y requerir que al menos un objeto por dispositivo la implemente. Usando esta técnica junto con el modelo de comunicaciones master / slave, es relativamente sencillo implementar algún mecanismo de administración y mantenimiento de dispositivos.

  Respecto a la flexibilidad, el prototipo facilita la introducción de cambios en un sistema distribuido gracias al uso de objetos para su construcción e interfases para el acceso a estos últimos. La flexibilidad también se ve favorecida por la transparencia de ubicación de los objetos para los clientes que los invocan. Estos no necesitan conocer el dispositivo donde un objeto reside para invocarlo. Sólo necesitan una referencia interoperable o IOR.

- **Modularidad y capacidad de evolución**: el diseño propuesto permite construir los sistemas como conjuntos distribuidos de objetos. Los módulos se forman a partir de agrupamientos de objetos relacionados funcionalmente entre sí.

  De igual forma, el prototipo mejora la capacidad de evolución de una aplicación ya que los objetos que forman cada dispositivo pueden ser actualizados con relativa facilidad. Estos pueden ser reemplazados por otros objetos que implementen las mismas interfases e inclusive, incorporar nuevas interfases para funcionalidades adicionales, manteniendo las viejas por compatibilidad. Todo esto es posible por la separación entre definición e implementación de los objetos.

- **Interoperabilidad e intercambiabilidad**: dos aspectos del diseño propuesto facilitan la interoperabilidad e intercambiabilidad de los dispositivos. En primer lugar, el uso de protocolos estándar (como GIOP modificado y CANIOP) facilita la interoperabilidad. En segundo lugar, el uso de un lenguaje abstracto, como OMG IDL, para la descripción de los dispositivos facilita tanto la interoperabilidad como la intercambiabilidad de los mismos.

- **Mejor rendimiento a menores costos**: en principio, el uso del middleware basado en CORBA, propuesto en el presente trabajo, como capa de aplicación del bus CAN, no debería afectar en forma negativa la performance de los sistemas construidos sobre él. Partiendo del diseño original de CORBA, se realizaron múltiples modificaciones para obtener un middleware que funcionase adecuadamente sobre CAN. Así, se buscó minimizar el tamaño de los mensajes de forma de conseguir invocar métodos de objetos mediante un único frame CAN, asegurando una buena performance con un overhead mínimo, a pesar de la distribución de los objetos en múltiples dispositivos.

  Respecto a la reducción de costos, la posibilidad de distribución de la lógica de control y monitoreo del proceso en múltiples dispositivos contribuye en este sentido. Todos los dispositivos conectados al bus aportan y cooperan para cumplir con los requerimientos globales del sistema. Esta cooperación permite que cada dispositivo cumpla con su parte del procesamiento con una menor cantidad de recursos computacionales. De esta forma, se disminuyen los costos de los dispositivos y de la totalidad del sistema.

Aparte de esta lista general, existen una serie de requerimientos o consideraciones particulares para el bus CAN [Etschberger 97]. Estos requerimientos deberían ser tenidos en cuenta a la hora de definir una capa de aplicación. Más que nada, son los servicios que esta capa debería ofrecer a las aplicaciones distribuidas que se construyen sobre ella:

- **Asignación de identificadores**: a cada frame CAN transmitido por el prototipo, se le asigna un valor particular para el campo IDENTIFIER del ARBITRATION FIELD. Este valor depende del modelo de comunicaciones que se desee implementar (peer-to-peer o productor / consumidor). En el primer caso, el IDENTIFIER incluye el identificador o dirección del dispositivo de destino mientras que en el segundo, sólo se incluye un identificador del grupo de objetos destinatario de la invocación.

  El diseño propuesto en el presente trabajo plantea un esquema de comunicación en grupo (invocación de métodos sobre grupos de objetos remotos) basado en el multicast confiable provisto por el bus CAN. Todos los dispositivos pueden decidir si aceptan o no un mensaje mediante la técnica de Message Filtering, dependiendo de si tienen objetos activos para el grupo invocado. Por otro lado, está garantizado que un mensaje es aceptado por todos los dispositivos o por ninguno. De esta forma, se obtiene un esquema de comunicación en grupo confiable.

- **Método de comunicación peer-to-peer**: el modelo de comunicaciones implementado por el prototipo permite comunicaciones entre dispositivos en una modalidad peer-to-peer. Para ello, directamente en el campo IDENTIFIER del frame CAN, se indica que los datos contenidos representan una invocación (mensaje de request bajo GIOP modificado) peer-to-peer. En este caso, se indica la dirección del dispositivo de destino del mensaje.

  Este modelo de comunicaciones usa mensajes de menor prioridad que los mensajes enviados bajo un modelo master / slave o productor / consumidor. Esto es así porque, normalmente, las comunicaciones peer-to-peer no son usadas para intercambiar datos del proceso que se está monitoreando y controlando. Estos últimos son datos de tiempo real que por lo general, requieren transferencias de mayor prioridad.

- **Método de intercambio de datos del proceso**: este método de comunicación debe permitir intercambiar datos leídos del proceso que se está controlando y monitoreando. Este requerimiento apunta a que se soporten modelos de comunicación productor / consumidor o master / slave.

  En el prototipo, estos modelos se implementan mediante el esquema de comunicación en grupo antes descripto el cual, permite la invocación de grupos de objetos. Las ORBs que corren en cada dispositivo incluyen un agente distribuido o Multicast Agent (MCA). Este mantiene únicamente información local sobre los grupos y los objetos activos en la ORB asociada (PGL y PML). Con esta información, la ORB puede configurar el registro mask-and-match del adaptador CAN y sólo recibir solicitudes para aquellos grupos que tienen al menos un objeto activo localmente.

- **Principios de modelado de dispositivos y definición de perfiles**: la capa de aplicación tiene que definir alguna metodología para modelar dispositivos y definir los perfiles de los mismos de forma de garantizar la interoperabilidad y la intercambiabilidad de los mismos.

  La metodología de modelado provista por el prototipo se basa en la definición de interfases mediante el uso del lenguaje OMG IDL. O sea, a partir del dispositivo físico, se construye un modelo abstracto de objetos que lo representa. Así, cada dispositivo es visto como una colección de objetos relacionados entre sí. Estos, siguiendo la filosofía de CORBA, ofrecen servicios que pueden ser invocados por otros objetos cliente, que potencialmente pueden residir en otros dispositivos, mediante una referencia interoperable o IOR. Los servicios se describen mediante la definición de interfases en OMG IDL.

  En conclusión, la descripción o perfil de un dispositivo, bajo el diseño propuesto, está formada por un modelo de objetos abstracto y por la definición en OMG IDL de todas las interfases que los miembros de dicho modelo implementan.

- **Administración de la red**: el prototipo no prevee protocolos específicos para administración y mantenimiento de la red. DeviceNet y CANopen si lo hacen mediante el protocolo de detección de MAC IDs duplicadas y el Offline Connection Set en el primer caso, y los protocolos NMT y LMT en el segundo.

  Un posible enfoque para proveer esta funcionalidad en el prototipo, implica definir alguna interfase de administración y mantenimiento. Luego, se debe requerir que al menos un objeto por dispositivo la implemente. Usando esta interfase y un modelo de comunicaciones master / slave o productor / consumidor, es posible implementar algún mecanismo de administración y mantenimiento tanto de la red como de los dispositivos.

Por último, ciertas características generales del middleware [Bray 97], se presentan también, en mayor o menor medida, en el prototipo resultante de la implementación del diseño propuesto. Estas características son:

- **Interoperabilidad**: dispositivos de diversos fabricantes pueden conectarse a un bus CAN e invocar mutuamente sus objetos siempre que incluyan la ORB y respeten los prototipos definidos por el diseño (CANIOP, GIOP modificado y CCDR). Estos protocolos también se encargan de ocultar posibles diferencias entre dispositivos (por ejemplo, el byte-order es ocultado por CCDR). La interoperabilidad también se ve favorecida por el uso de OMG IDL para la descripción de los dispositivos.

- **Transparencia**: el prototipo favorece la transparencia en dos sentidos. En primer lugar, el uso de protocolos genéricos de comunicación permite ocultar y hacer transparentes posibles diferencias de implementación entre dispositivos (hardware, sistema operativo, etc.). El ejemplo de distinto byte-order también aplicaría en este caso. En segundo lugar, el uso de referencias interoperables o IORs para acceder a los objetos facilita la transparencia de distribución. En principio, un cliente no conoce ni debe conocer en forma directa la dirección del dispositivo donde reside el objeto que desea invocar. O sea, el prototipo provee transparencia de distribución [Tanenbaum 02].

- **Confiabilidad y Disponibilidad**: estas dos características se presentan también en la capa de aplicación propuesta. Para lograr middlewares confiables y de alta disponibilidad, es conveniente la distribución de recursos y servicios evitando componentes centralizados. En el prototipo no existen componentes centralizados y tanto recursos como servicios, se encuentran distribuidos convenientemente.

- **Escalabilidad**: el prototipo funcionaría en redes CAN con hasta 32 dispositivos conectados. Esta limitación surge del tamaño acotado del campo IDENTIFIER en frames CAN de la parte A. Si se hubiese seleccionado la parte B de la especificación de CAN para el diseño y construcción del prototipo, esta limitación no existiría ya que se dispondría de un campo IDENTIFIER de 29 bits. El prototipo sería capaz de escalar a sistemas aún más grandes.

- **Abstracción**: el prototipo permite al desarrollador de aplicaciones distribuidas trabajar con abstracciones de alto nivel como son los objetos CORBA. Se evita así trabajar directamente con frames CAN o primitivas de bajo nivel del sistema como podrían ser adaptadores, registros, conexiones, puertos o semáforos.

En conclusión, el middleware basado en CORBA propuesto en este trabajo constituye una opción válida a la hora de implementar un sistema de control distribuido sobre el bus CAN. Al igual que otras capas de aplicación disponibles hoy en día, como CANopen y DeviceNet, el diseño propuesto cumple con los servicios que una capa de este tipo debiera ofrecer a las aplicaciones [Etschberger 97]. Por otro lado, el bus industrial resultante de combinar CAN con un middleware basado en CORBA, tomado como conjunto, cumple también con los requerimientos planteados por Thomesse y Babiuch. Finalmente, y como un aporte adicional, se comprueba que ciertas características del middleware, planteadas por Bray**],** se presentan también en la solución propuesta.

La combinación de CAN con CORBA planteada aquí constituye una buena plataforma para la construcción de los sistemas computacionales futuros encargados, en su gran mayoría, del monitoreo y control de procesos en tiempo real, cumpliendo con restricciones temporales y a la vez, siendo confiables. El uso del bus CAN como sistema de comunicación, las facilidades para comunicación en grupo y el esquema de asignación de prioridades facilitan el cumplimiento de las primeras. De igual forma, el uso de una ORB basada en CORBA como capa de aplicación para dicho bus, facilita la distribución de las tareas en múltiples dispositivos. De esta distribución, surgen mejoras en aspectos tales como la confiabilidad general del sistema, la transparencia de distribución o la interoperabilidad entre dispositivos.

9 Referencias

[Aiken 00] *RFC 2768, Network Policy and Services: A Report of a Workshop on Middleware*, B. Aiken, J. Strassner, B. Carpenter, I. Foster, C. Lynch, J. Mambretti, R. Moore y B. Teitelbaum, The Internet Society, Reston, VA, 2000

[Babiuch 00] *Fieldbus Protocol Requirements,* Babiuch, Marek, Proceedings of XXIV. ASR 2000 Seminar on Instruments and Control, vol. 44, Ostrava, 2000

[Barbacci 95] *Quality Attributes (CMU/SEI-95-TR-021)*, Barbacci, Mario; Klein, Mark H.; Longstaff, Thomas H. & Weinstock, Charles B. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1995.

[Boterenbrood 00] *CANopen, high-level protocol for CAN-bus*, H. Boterenbrood, NIKHEF, Amsterdam, 2000

[Bray 97] *Software Technology Roadmap, Middleware*, Mike Bray, http://www.sei.cmu.edu/str/descriptions/middleware_body.html Software Engineering Institute, Carnagie Mellon University, Pittsburgh, PA, 1997

[CAL 96] *CiA DS201: CAN Application Layer for Industrial Applications - CAN in the OSI Reference Model*, Version 1.0, 1996

[CAN 91] *BOSCH, CAN Specification, Version 2.0*, Robert Bosch GmbH, 1991

[CANopen 99] *CiA DS 301: CANopen - Application Layer and Communication Profile*, Version 4.0, 1999

[CiA 04] *Sitio web de CAN in Automation (CiA)*, http://www.can-cia.de

[Corba Core 02] *Common Object Request Broker Architecture: Core Specification, Version 3.0, December 2002*, Object Management Group, Inc., 2002

[Corba Min 02] *Minimum CORBA Specification, Version 1.0, August 2002*, Object Management Group, Inc., 2002

[DevNet1 99] *Volume I: DeviceNet Communication Model and Protocol, Release 2.0*, Open DeviceNet Vendor Association, Inc., 1999

[DevNet2 99] *Volume II: DeviceNet Device Profiles and Object Library, Release 2.0*, Open DeviceNet Vendor Association, Inc., 1999

[Etschberger 97] *CAN-based Higher Layer Protocols and Profiles*, K. Etschberger, 4th International CAN Conference, Berlin, Octubre, 1997

[Hong 00] *Building Light Weight CORBA based Middleware for the CAN Bus Systems*, Seongsoo Hong, Transactions on Control, Automation, and Systems Engineering Vol. 2, No. 1, Enero, 2000

[ICELab] *Networked Control Systems,* ICE Lab (Intelligent Control Engineering Laboratory), University of Maryland at College Park, http://www.enme.umd.edu/ice_lab/ncs/ncs.html

[ISO7498] *Series X: Data Networks and Open System Communications, Open Systems Interconnection – Model and Notation, ITU-T X.200-X.216*, International Telecommunication Union, 1994

[ISO8824] *Series X: Data Networks and Open System Communications, OSI Networking and System Aspects, ITU-T X.680-X.693: Abstract Syntax Notation One (ASN.1)*, International Telecommunication Union, 2002

[ISO8825] *Series X: Data Networks and Open System Communications, OSI Networking and System Aspects, ITU-T X.690-X.693 :ASN.1 Encoding Rules*, International Telecommunication Union, 2002

[IXXAT 04] *CAN Driver for Windows*, http://www.ixxat.de/english/produkte/canprod/interf/candriver_windows.shtml

[Kaiser 98] *Invocation of Real-Time Objects in a CAN Bus-System*, Jörg Kaiser y Mohammad Ali Livani, Proceedings of the 1st International Symposium on Object-Oriented Real-Time Distributed Computing, Kyoto, 1998, p. 298-307

[Kalani 2002] *Industrial Process Control: Advances and Applications*, Ghodrat Kalani, Butterworth-Heinemann, Elsevier Science, Woburn, MA, 2002

[Kim 00] *Integrating Subscription-based and Connection-oriented Communications into the Embedded CORBA for the CAN Bus*, Kimoon Kim, Gwangil Jeon, Seongsoo Hong, Tae-Hyung Kim y Sunil Kim, 6th IEEE Real Time Technology and Applications Symposium (RTAS 2000), Washington, Junio, 2000

[Kleindienst 96] *CORBA and Object Services*, Jan Kleindienst, František Plášil y Petr Tůma, Invited paper, SOFSEM 96, Springer LNCS 1175, 1996

[Lankes 03] *Integration of a CAN-based Connection-oriented Communication Model into Real-Time CORBA*, Stefan Lankes, Andreas Jabs y Thomas Bemmerl, Proceedings of the 17th International Parallel and Distributed Processing Symposium, Niza, Abril, 2003

[Lian 01] *Performance Evaluation of Control Networks: Ethernet, ControlNet, and DeviceNet*, Feng-Li Lian, James R. Moine, and Dawn M. Tilbury, IEEE Control Systems Magazine, Vol. 1, Nro. 1, February 2001, p. 86-83.

[ORBit 04] *ORBit Project Documentation*, http://orbit-resource.sourceforge.net/

[Sadoski 97] *Software Technology Roadmap, Transaction Processing Monitor Technology*, Darleen Sadoski, http://www.sei.cmu.edu/str/descriptions/tpmt_body.html Software Engineering Institute, Carnagie Mellon University, Pittsburgh, PA, 1997

[Soley 95] *Object Management Architecture Guide*, Richard Mark Soley y Christopher M. Stone, John Wiley & Sons, Framingham, MA, 3era edición, 1995

[Sterling 03] *The Industrial Ethernet networking guide: understanding the infrastructure connecting business enterprises, factory automation and constrol systems*, Donald J. Sterling, Jr. y Steven P. Wissler, Thomson-Delmar Learning, Clifton Park, NY, 2003

[Sun 04] *Middleware: the key to next generation computing (Preface)*, Xian-He Sun y Alan R. Blatecky, Journal of Parallel and Distributed Computing nro. 64, 2004, p. 689-691

[Tanenbaum 96] *Sistemas Operativos Distribuidos*, Andrew S. Tanenbaum, Prentice Hall, 1era edición, 1996

[Tanenbaum 02] *Distributed Systems: Principles and Paradigms*, Andrew S. Tanenbaum y Maarten van Steen, Prentice Hall, 1era edición, 2002

[Thomesse 99] *Fieldbuses and interoperability*, Jean Pierre Thomesse, Control Engineering Practice nro. 7, 1999, p. 81-94

[Vondrak 97] *Software Technology Roadmap, Remote Procedure Call*, Cory Vondrak,
http://www.sei.cmu.edu/str/descriptions/rpc_body.html
Software Engineering Institute, Carnagie Mellon University, Pittsburgh, PA, 1997

[VanGompel 01] *CIP: Not Just Another Pretty Acronym, Bringing control services to any network, from DeviceNet to Ethernet*, Dave VanGompel, Rockwell Automation, July 2001

[Veríssimo 01] *Distributed Systems for System Architects*, Paulo Veríssimo and Luis Rodrigues, Kluwer Academia Publishers, 2001, p171-192.

[Wainer 97] *Sistemas de tiempo real: conceptos y aplicaciones*, Gabriel A. Wainer, Nueva Librería S.R.L., Buenos Aires, 1997

[Wallnau 97] *Software Technology Roadmap, Object Request Broker*, Kurt Wallnau y John Foreman,
http://www.sei.cmu.edu/str/descriptions/orb_body.html
Software Engineering Institute, Carnagie Mellon University, Pittsburgh, PA, 1997

[XML-RPC] *XML-RPC Specification*, http://www.xmlrpc.com/spec, 2003.

Apéndice A: Implementación del prototipo

1 Simulador del bus CAN

El simulador del bus CAN está compuesto por dos módulos. El primero de estos módulos consiste en un proceso simulador que representa el bus en sí y que se encarga, principalmente, de correr el algoritmo de arbitración. Este primer módulo recibe el nombre de sim_can. El segundo módulo, consiste en una librería llamada sim_can_lib. Esta presenta una interfase similar a la de un driver de un adaptador CAN real y es usada por los procesos que simulan dispositivos, para comunicarse con el proceso simulador. En cada uno de estos procesos, sim_can_lib actúa como un adaptador real, permitiendo a los dispositivos simulados competir por el control del bus y así, enviar y recibir frames CAN. En un esquema, esto sería:

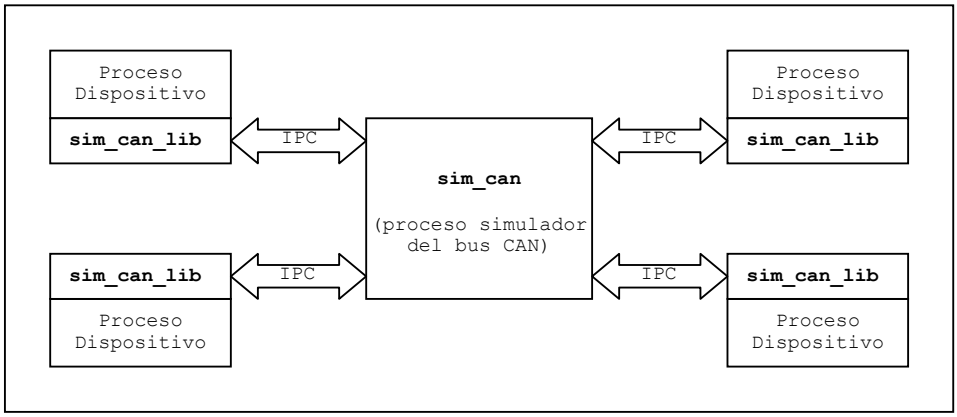


Ilustración 36 - Funcionamiento del simulador del bus CAN

Bajo el protocolo de comunicación definido entre sim_can_lib y sim_can, los procesos que simulan dispositivos cumplen el rol de clientes mientras que el proceso simulador o sim_can, cumple con el rol de servidor. Este protocolo se basa en la utilización del mecanismo de colas provisto normalmente por IPC (Inter-Process Communication) en los sistemas tipo UNIX.

En cierta forma, el protocolo antes mencionado es orientado a conexión. El servidor de simulación o sim_can crea una cola de mensajes y publica la clave que la identifica. Esta se usa para crear el recurso de IPC correspondiente mediante un llamado a la system call ftok. Recibe el nombre de srv_queue_key.

Un proceso cliente, para participar en la simulación, debe conectarse al proceso servidor sim_can. Para conseguir esto, debe abrir un handle a la cola del servidor, usando srv_queue_key, y enviar un mensaje de tipo SIM_CAN_MSG_TYPE_CONN por dicha cola. Previamente, tiene que haber creado una cola de recepción de mensajes identificada por la clave clnt_queue_key.

El servidor sim_can, al recibir el mensaje de establecimiento de conexión, reserva los recursos necesarios y, al igual que el cliente con la cola del servidor, abre un handle a la cola del cliente. Si pudo cumplir con estos pasos en forma exitosa, el servidor envía al cliente un mensaje SIM_CAN_MSG_TYPE_CONN_OK. De esta forma, la cola de mensajes IPC identificada por srv_queue_key es usada por el servidor para recibir mensajes de todos los clientes mientras que cada cliente, tiene su propia cola para recibir mensajes del servidor.

En conclusión, el mecanismo de establecimiento de conexiones entre los clientes y el servidor de simulación se vería de la siguiente forma en un esquema:

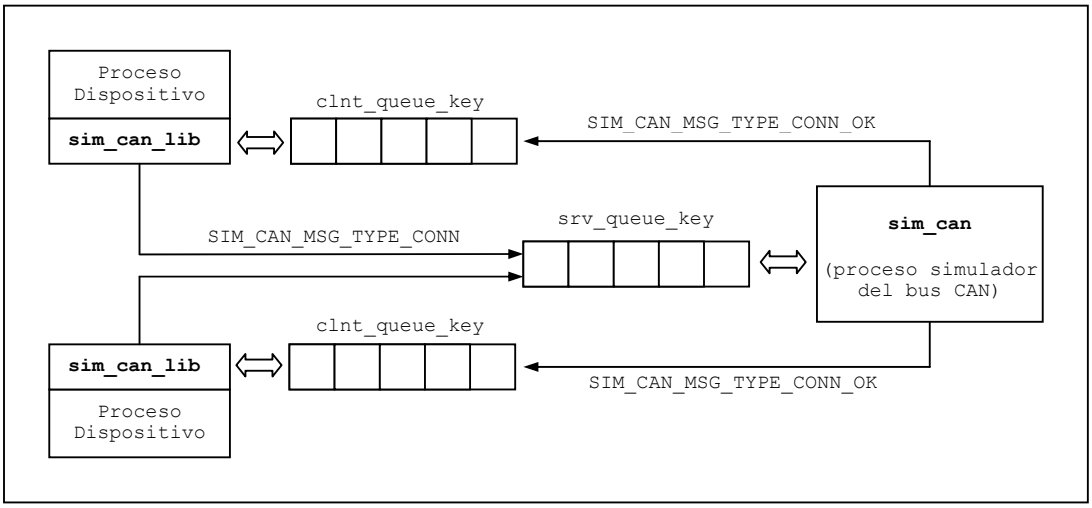


Ilustración 37 - Funcionamiento del simulador del bus CAN

Cuando un dispositivo simulado (un cliente) quiere enviar un frame CAN al bus, debe enviar un mensaje de tipo SIM_CAN_MSG_TYPE_FRAME al servidor de simulación sim_can. El cuerpo de este mensaje consiste prácticamente en un frame CAN de parte A sin modificaciones. El servidor sim_can simula el bus definiendo un período de simulación. Durante cada período, el servidor recibe mensajes de todos los dispositivos y los almacena en un buffer. Cada vez que expira uno de estos períodos, se corre en el servidor el algoritmo de arbitración sobre todos los frames recibidos durante el último intervalo.

El servidor notifica el resultado al cliente que ganó el bus mediante un mensaje SIM_CAN_ARBIT_RES_WON. Un único cliente de todos los conectados al servidor sim_can puede recibir un mensaje de este tipo ya que sólo uno gana el control del bus en cada período de simulación. Los clientes perdedores reciben como respuesta de su mensaje SIM_CAN_MSG_TYPE_FRAME un mensaje de tipo SIM_CAN_ARBIT_RES_LOST. Todo este proceso se da en forma sincrónica desde el punto de vista de los clientes o sea, desde la interfase de sim_can_lib.

A continuación se incluye un ejemplo del mecanismo de simulación del bus:

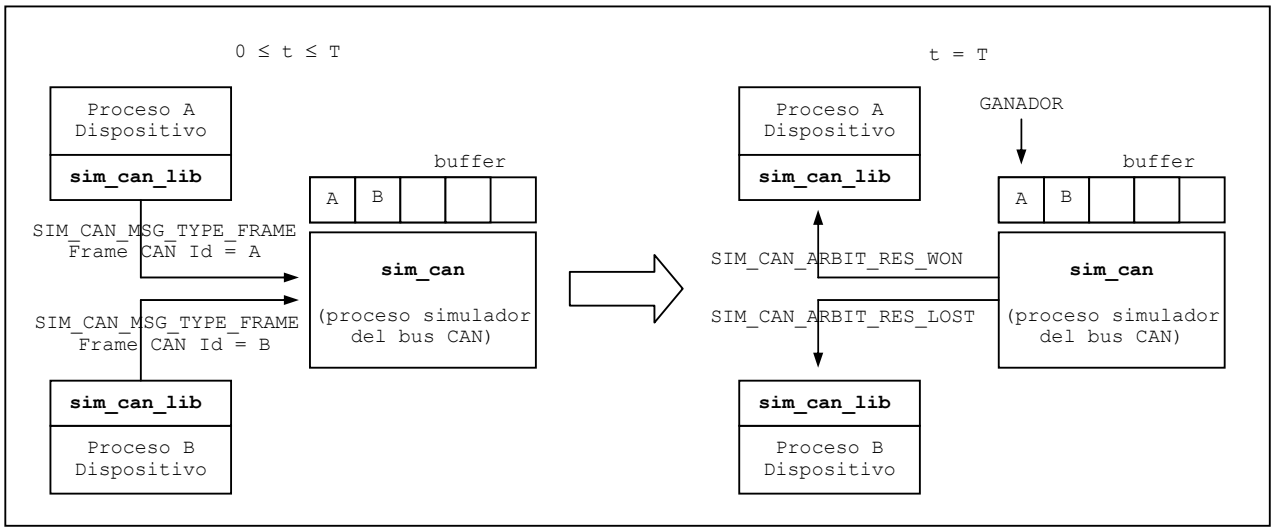


Ilustración 38 - Proceso de simulación del mecanismo de arbitración del bus CAN

Ambos módulos del simulador fueron construidos trabajando en C estándar bajo el sistema operativo Linux (particularmente, se usó la distribución Red Hat 9 con kernel versión 2.4.20-8). La interfase de la librería sim_can_lib, construida a partir de la API VCI (Virtual CAN Interface) de los adaptadores para PC provistos por IXXAT [IXXAT 04], está compuesta por las siguientes funciones:

- **can_lib_load_lib**: esta función permite iniciar la librería. Es necesario llamarla antes de llamar a cualquier otra función de la librería. Permite asignar a cada proceso cliente que use la librería un identificador para así, poder distinguirlos en el servidor de simulación. También permite definir un nivel de log y de generación de trazas.

- **can_lib_free_lib**: esta función se encarga de descargar la librería. Debería ser llamada por cualquier programa que se comporte como cliente de sim_can antes de salir. De lo contrario, quedan recursos reservados que pueden ser requeridos por otros programas (principalmente, recursos de comunicación IPC).

- **can_lib_init_adapter**: esta función inicializa el adaptador del bus CAN simulado mediante el servidor sim_can. En primer lugar, se crea la cola de mensajes del cliente y se abre un handle a la cola del servidor sim_can. Estas dos colas son usadas para intercambiar mensajes entre cliente y servidor. Una vez llamada esta función y habiendo obtenido un resultado exitoso, es posible llamar a can_lib_start_adapter (abre una conexión con el servidor sim_can).

- **can_lib_start_adapter**: esta función arranca el adaptador del bus CAN simulado mediante sim_can, usando los parámetros especificados en la llamada previa a can_lib_init_adapter. Antes de llamar a esta función es necesario inicializar el buffer de recepción mediante una llamada a can_lib_init_rx_queue y a can_lib_set_acc_mask, para inicializar el registro usado como filtro mask-and-match (para aceptar o descartar los mensajes).

- **can_lib_term_adapter**: esta función se encarga de cerrar el adaptador. Debería ser llamada por cualquier programa que se comporte como cliente de sim_can antes de salir y de descargar la librería (llamando a can_lib_free_lib). De lo contrario, quedan recursos reservados que pueden ser requeridos por otros programas (principalmente, recursos de comunicación IPC).

- **can_lib_init_rx_queue**: inicializa y aloca los recursos necesarios para el buffer de recepción de mensajes del adaptador CAN. Esta función tiene que ser llamada antes de arrancar el adaptador CAN llamando a can_lib_start_adapter. Se debe indicar la indicar la capacidad del buffer de recepción directamente pasando la cantidad de frames CAN que éste puede contener.

- **can_lib_set_acc_mask**: inicializa el registro mask-and-match del adaptador CAN mediante una mascara y un valor. Cada vez que llega un frame CAN al proceso cliente que simula un dispositivo, se toma el campo de arbitración (ARBITRATION FIELD) del mismo y se le aplica la mascara especificada. El resultado es comparado con el valor. Si estos dos son iguales, el mensaje es aceptado y se copia al buffer de recepción. De lo contrario, el mensaje es descartado. La librería simula un adaptador CAN de tipo BasicCAN con dos registros mask-and-match, siendo posible inicializar cualquiera de los dos.

- **can_lib_send**: esta función permite poner un mensaje en el bus. En realidad, lo que se termina haciendo es enviar un frame CAN desde sim_can_lib a sim_can mediante las colas de mensajes IPC antes descriptas. El proceso servidor se encarga de simular la arbitración de CAN y reenviar el mensaje a los dispositivos simulados que corresponda (a otros procesos cliente). La función devuelve un valor que indica el resultado de todo este proceso.

- **can_lib_recv**: esta función permite recibir un mensaje del bus. En realidad, lo que se termina haciendo es esperar un mensaje de sim_can si el buffer de recepción está vacío. Sino, directamente se devuelve un mensaje del buffer (el cual funciona como una cola FIFO). La función puede bloquearse si el buffer está vacío.

El código fuente de este componente de simulación puede encontrarse en el directorio */source/can/sim_can_lib*. Para los procesos que deseen utilizar la librería sim_can_lib y por medio de esta, el servidor de simulación, se debe incluir el archivo *sim_can_lib.h* al momento de compilación. Es necesario también linkear estáticamente contra el código objeto *sim_can_lib.a*. A continuación se incluye un ejemplo de makefile usado para compilar y linkear el programa de ejemplo can_lib_test_send:

```
SIM_CAN_LIB_OBJECT=sim_can_lib.a
SIM_CAN_LIB_INCLUDES=-I ../util -I ../util/ipclib -I ../sim_can_proto -I ./ -I ../sim_can
CC_OPTIONS=-g
MAKE=make
CC=gcc

all: can_lib_test_send.c sim_can_lib.h sim_can_lib.a
	${MAKE} can_lib_test_send

can_lib_test_send: can_lib_test_send.c sim_can_lib.h sim_can_lib.a
	${CC} ${CC_OPTIONS} ${SIM_CAN_LIB_INCLUDES} -o can_lib_test_send
	can_lib_test_send.c ${SIM_CAN_LIB_OBJECT}

clean:
	rm -f *.o
```

Código 44 – Makefile para el programa de ejemplo can_lib_test_send

2 Organización del código fuente del prototipo

En esta sección del documento se describe la organización del código fuente del prototipo de ORB así como de los módulos del simulador de bus CAN, sim_can y sim_can_lib. Todo este código fue desarrollado en C estándar bajo el sistema operativo Linux. En particular, se usó la distribución Red Hat 9 con kernel versión 2.4.20-8, el compilador / linker gcc versión 3.2.1 y la librería glibc versión 2.3.2.

La totalidad del código se incluye en el directorio *./source* del disco que acompaña el presente trabajo. A continuación se incluye una lista de los directorios y un detalle de los archivos contenidos por estos. Entonces:

Directorio ./can/sim_can

Archivos	Descripción
sim_can.c	Cuerpo principal del servidor de simulación sim_can. Inicializa el modulo sim_can_core de acuerdo a los parámetros recibidos del shell.
sim_can_core.h **sim_can_core.c**	Este módulo implementa prácticamente todo el servidor de simulación sim_can. Únicamente publica tres funciones: • can_core_init: inicializar el módulo. • can_core_term: permite cerrar el módulo. • can_core_run: función bloqueante que corre el servidor de simulación. El módulo se encarga de aceptar conexiones de clientes, recibir los mensajes enviados por los mismos, correr el proceso de arbitración entre los mensajes recibidos en el ultimo intervalo de simulación y hacer el broadcast del mensaje ganador.

Directorio ./can/sim_can_lib

Archivos	Descripción
sim_can_lib.h **sim_can_lib.c**	Este módulo implementa la interfase del simulador de bus CAN que deben usar los programas que quieran comportarse como nodos de dicho bus (dispositivos conectados al mismo). Presenta una interfase similar a la de un driver de adaptador CAN. Para que el módulo funcione correctamente es necesario que este corriendo el servidor de simulación sim_can.

Directorio ./can/sim_can_proto

Archivos	Descripción
sim_can_proto.h **sim_can_proto.c**	Este módulo implementa funciones necesarias para trabajar con los mensajes del protocolo usado entre los clientes (sim_can_lib) y el servidor de simulación (sim_can). También hay funciones que permiten comparar campos de arbitración de mensajes CAN y correr el proceso de arbitración entre múltiples mensajes.

Directorio ./can/util

Archivos	Descripción
logger.h **logger.c**	Este módulo permite enviar mensajes de log tanto a la terminal del proceso como a un archivo predeterminado. Se puede especificar un nivel de log para filtrar los mensajes a enviar a la consola o al disco, un tamaño máximo para cada archivo e inclusive, una cantidad máxima de archivos (vendría a ser un log circular). La interfase incluye las funciones logger_initialize, logger_write y logger_terminate.

t_list.h **t_list.c**	El módulo implementa una lista doblemente enlazada, con capacidad de crecer en forma dinámica. La interfase de la lista incluye funciones como list_add_head, list_add_tail, list_move_next, list_move_prev, list_get_current, etc.
timer.h **timer.c**	Implementa un timer que permite controlar que ciertas operaciones no demoren o se bloqueen más de lo deseado. Para esto, el módulo publica las funciones timer_init, timer_start y timer_term.

Directorio ./can/util/ipclib

Archivos	Descripción
t_msg.h **t_msg.c**	Conjunto de primitivas para la creación y manejo de colas de mensajes IPC y para el envío y recepción de los mismos. Algunas de las primitivas son msg_create, msg_open, msg_send, msg_recv y msg_close.
t_sem.h **t_sem.c**	Conjunto de primitivas para la creación y manejo de semáforos IPC. Algunas de las primitivas son sem_create, sem_open, sem_p, sem_v y sem_close.
t_shm.h **t_shm.c**	Conjunto de primitivas para la creación y manejo de áreas de memoria compartida mediante IPC. Algunas de las primitivas son shm_create, shm_open, shm_attach, shm_detach y shm_close.

Dentro del directorio *./source/can/can_orb/src* se puede encontrar el código fuente del prototipo de ORB que correría en cada uno de los dispositivos conectados al bus CAN. La versión incluida con el presente trabajo está preparada para funcionar usando el simulador de CAN, formado por sim_can y sim_can_lib. El código fuente que compone el prototipo se organiza en los siguientes directorios:

Directorios	Descripción
./can/can_orb/src/CANIOP	Este directorio incluye la implementación de las funciones necesarias para soportar el protocolo de transporte CANIOP y el protocolo de sesión usado por el prototipo, GIOP modificado. Se incluyen funciones para la codificación y decodificación de mensajes de ambos protocolos así como funciones para la administración de buffers de recepción y envío.
./can/can_orb/src/idl-compiler	El directorio incluye el código fuente del compilador de OMG IDL. Hay un conjunto de archivos que implementan un compilador genérico (parseo de archivos de entrada, análisis sintáctico, etc.). También se incluye el mapeo específico para el lenguaje C dentro del directorio *./backends/c*.
./can/can_orb/src/orb	Contiene el código de la ORB propiamente dicha. Esto comprende el código del POA, del MCA, las definiciones de servants y objetos CORBA, la interfase y el núcleo de la ORB.

./can/can_orb/src/util	Este directorio contiene definiciones de tipos de dato y funciones utilitarias usadas por el resto de la ORB. También incluye los archivos can_orb_logger.h y can_orb_logger.c que implementan el módulo de logueo del prototipo.

El prototipo de ORB puede ser compilado corriendo el script *./can/can_orb/build.sh*. Una vez completada en forma exitosa la compilación, en el directorio *./can/can_orb/bin* pueden ubicarse los programas ejecutables como el compilador de IDL. En *./can/can_orb/include*, se encuentran los archivos .h que deben ser incluidos por los procesos que deseen utilizar la ORB. Por último, en *./can/can_orb/lib* se ubican las librerías contra las cuales deben linkearse los procesos que simulan dispositivos conectados al bus CAN a través de la ORB.

Para poder instalar el prototipo, es necesario ejecutar el script ubicado en *./can/install.sh*. Una vez completada en forma exitosa la instalación, se puede correr un programa de demostración ejecutando el script *./can/can_orb_demo/run.sh*. Los requerimientos para poder correr el prototipo y el programa de demostración antes mencionado son:

- Linux con kernel versión 2.4.20-8 o superior.
- GNU Compiler Collection 3.2.2 o superior.
- GLib-1.2.10
- GTK+-1.2

3 Programas de demostración

El prototipo de ORB incluye una serie de programas de demostración los cuales pueden ser ejecutados mediante el script *./can/can_orb_demo/run.sh*. Este script va a ejecutar, en primer lugar, el simulador del bus CAN. En segundo lugar, va a ejecutar una serie de programas que simulan dispositivos conectados al bus CAN. Cada uno de estos dispositivos simulados corre una instancia de la ORB para el bus CAN.

Se desarrollaron tres tipos diferentes de dispositivos simulados: controlador, sensor y actuador. El dispositivo controlador se configura con la IOR de un grupo de sensores y con la IOR de un grupo de actuadores. A cada dispositivo sensor así como a cada actuador, se le debe asignar un identificador de grupo de objetos de forma de poder configurar el MCA de la instancia de ORB subyacente. Así, cada sensor y cada actuador es miembro de al menos un grupo de objetos.

Una vez inicializado el dispositivo controlador con las referencias interoperables de los grupos, este último hace un poll de los sensores en forma periódica, mediante una invocación de un método sobre el grupo correspondiente. Luego, los sensores informan al controlador, también mediante una llamada a un método, el valor de las variables del proceso monitoreado. Por último, el controlador, dependiendo de los valores informados por los sensores, decide si es necesario cambiar el estado de los miembros del grupo de actuadores. Este cambio de estado también se realiza mediante una llamada a un método a través de la ORB.

La interacción entre el controlador y el grupo de sensores o el grupo de actuadores implica la utilización de un modelo de comunicación master / slave. Por el contrario, cuando un sensor particular informa al controlador los valores leídos de las variables del proceso, se utiliza un modelo de comunicación peer-to-peer.

Cada uno de estos dispositivos se implementa como un objeto CORBA, que adhiere a alguna de las siguientes interfases:

```
interface Controller {
	enum ProcessVariables {
		PROC_VAR_1,
		PROC_VAR_2,
		PROC_VAR_3
	};

	void push_proc_var(in ProcessVariables var_id,in float var_value);
};

interface Sensor {
	void initialize(in string ior_cont);
	void poll_proc_vars();
};

interface Actuator {
	enum ActuatorStates {
		ACTUATOR_OFF,
		ACTUATOR_ON
	};

	void initialize(in string ior_cont);
	void set_state(in ActuatorStates state);
};
```

Código 45 – Interfases de objetos controlador, sensor y actuador

Cada tipo de dispositivo provee una interfaz gráfica, basada en GTK+, que facilita la configuración del dispositivo simulado y que permite observar en forma directa e inmediata el estado del mismo. La interfaz gráfica del dispositivo sensor es la siguiente:

Ilustración 39 - Interfaz del dispositivo sensor

La sección bajo el título *Proceso Controlado* permite ajustar el valor de tres variables leídas por el sensor desde el supuesto proceso real que se estaría monitoreando. El valor de dichas variables es informado al objeto controlador cuando este último llama al método *poll_proc_vars* del grupo de objetos sensores. El botón *Configurar ORB CAN* permite ajustar parámetros de la instancia de ORB asociada al dispositivo sensor, mediante el siguiente cuadro de diálogo:

Ilustración 40 – Diálogo para configuración de ORB CAN

Vale aclarar que para correr el programa de demostración no es necesario ajustar estos parámetros en ninguno de los dispositivos. El script *./can/can_orb_demo/run.sh* se encarga de especificar los parámetros necesarios para cada dispositivo. En total, se ejecutan tres actuadores junto con un sensor y un controlador.

El botón *Configurar MCA* permite definir y administrar la pertenencia del objeto sensor a cierto grupo de objetos. Al seleccionar dicha opción, se muestra el siguiente cuadro de diálogo:

Ilustración 41 – Diálogo para configuración de MCA

En la parte inferior derecha del cuadro de diálogo anterior se puede ver que se genera la IOR para el grupo al cual haya sido agregado el dispositivo en cuestión. Esta IOR es necesaria para configurar el controlador, ya que este último necesita invocar métodos sobre los grupos de sensores y sobre los grupos de actuadores (es conveniente copiar el valor utilizando el clipboard).

Por último, el botón *Correr ORB CAN* inicializa la instancia de ORB asociada al dispositivo sensor. Una vez hecho esto, el dispositivo y el objeto CORBA que lo modela están listos para recibir invocaciones de otros dispositivos u objetos como por ejemplo, el controlador. La interfaz gráfica de este último dispositivo es la siguiente:

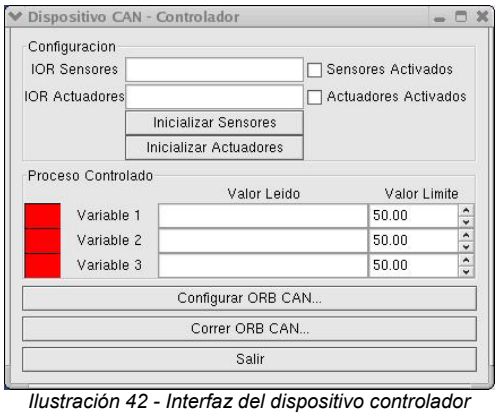


Ilustración 42 - Interfaz del dispositivo controlador

La sección *Configuración* permite especificar la IOR de los grupos definidos para los dispositivos sensores y para los dispositivos actuadores (esto se hace previamente en cada sensor y actuador, accediendo a la opción *Configurar MCA*). Una vez cargados estos dos valores, se pueden seleccionar los botones *Inicializar Sensores* e *Inicializar Actuadores*. Esto último dispara el timer que periódicamente va a pollear el grupo de sensores (mediante una llamada a *poll_proc_vars*) para leer el valor de las tres variables del proceso monitoreado.

Justamente, la sección inferior llamada *Proceso Controlado*, muestra el valor de las tres variables del proceso leídas de los sensores. También permite especificar un límite para cada una de estas variables. Cuando la variable está por encima del límite, el recuadro a la izquierda aparece en color rojo. Por el contrario, si el valor leído de los sensores está dentro del límite configurado, el recuadro aparece en color verde, como se muestra a continuación:

Ilustración 43 - Interfaz del dispositivo controlador

Por último, el dispositivo actuador ofrece una interfaz gráfica similar a la del sensor, excepto que el actuador muestra su propio estado en lugar de las variables del proceso en la parte superior de la ventana:

Ilustración 44 - Interfaz del dispositivo actuador

Cuando el controlador detecta que cualquiera de las variables está por encima del límite establecido, cambia el estado de los actuadores a *ACTUATOR_OFF* mediante una llamada al método *set_state*. En ese caso, el recuadro en la parte izquierda aparece en rojo. Por el contrario, si todas las variables están dentro de los límites, el estado de los actuadores es *ACTUATOR_ON* y el recuadro aparece en color verde:

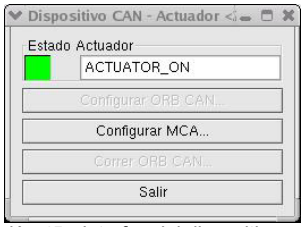


Ilustración 45 - Interfaz del dispositivo actuador

Glosario

ASN.1
Abstract Syntax Notation 1 (ASN.1) es la sintaxis abstracta, perteneciente a la capa de presentación, definida por el modelo de referencia ISO/OSI en el estándar ISO 8824. Esta sintaxis permite definir datos y estructuras de datos en forma abstracta.

Attribute ID
Es un entero asignado a cada atributo de clase o instancia en un dispositivo DeviceNet.

BER
Basic Encoding Rules (BER) son las reglas de codificación, pertenecientes a la capa de presentación definida por el modelo de referencia ISO/OSI en el estándar ISO 8824. Permiten transformar datos abstractos en un stream de bits.

CAN
Controller Area Network (CAN) es un protocolo de comunicaciones seriales o bus industrial, diseñado para soportar, en forma eficiente, aplicaciones de control distribuidas con un alto nivel de seguridad. Fue desarrollado originalmente por Bosch, la especificación da libertad para seleccionar distintas capas de aplicación.

CANopen
Capa de aplicación para el bus CAN, desarrollada originalmente como un proyecto Esprit de la Unión Europea, bajo el auspicio de Bosch. Permite modelar cada dispositivo conectado al bus CAN mediante un diccionario de objetos, accesible en forma remota.

CDR
Common Data Representation (CDR) es el protocolo de presentación utilizado por las ORB CORBA. Define una sintaxis de transferencia que permite obtener un stream de bytes a partir de una interfase definida mediante OMG IDL.

CiA
CAN in Automation (CiA) es una organización internacional sin fines de lucro de usuarios y fabricantes que desarrolla y soporta protocolos de aplicación para el bus CAN como por ejemplo, CANopen y DeviceNet.

Class ID
Es un entero único asignado a cada clase en un dispositivo DeviceNet. Se pueden usar 1 o 2 bytes para guardar este valor.

COB
Communication Object (COB) es un objeto CANopen asociado a una unidad de transporte en el bus CAN. Cada unidad de transporte equivale a alguno de los tipos de frame definidos por el protocolo CAN. Estos objetos se identifican mediante un COB-ID único que se corresponde con el valor del ARBITRATION FIELD del frame CAN.

CORBA
Common Object Request Broker Architecture (CORBA) es el middleware de tipo ORB propuesto por el Object Management Group (OMG).

CSMA
Carrier Sense Multiple Access (CSMA) es un protocolo usado para determinar como dos o más computadoras conectadas a una red deben comportarse cuando al menos dos de ellas intentan utilizar el medio compartido de la red en forma simultánea. Algunas variantes de este protocolo son CSMA/CD (Collision Detection) y CSMA/CA (Collision Avoidance).

DeviceNet
Capa de aplicación para el bus CAN, desarrollada originalmente por Allen-Bradley y aprobada por CiA. Está definida en términos de un modelo abstracto de objetos, que representa el comportamiento externo de un dispositivo DeviceNet. Hoy en día es mantenida por un organismo independiente llamado Open DeviceNet Vendor Association (ODVA), encargado de promover el uso de la misma.

DBT
Servicio de la capa CAN Application Layer (CAL) de CANopen. Su función es asignar en forma dinámica identificadores únicos a los COB. Debe trabajar cooperando con los DBT de otros dispositivos conectados a la red CANopen para garantizar la unicidad.

DII
Dynamic Invocation Interface (DII) es una interfase de la ORB CORBA que permite a los clientes construir solicitudes de servicio en forma dinámica. Gracias a DII, los clientes pueden trabajar sin necesidad de contar con el stub de los objetos.

DSI
Dynamic Skeleton Interface (DSI) es una interfase de la ORB CORBA que permite a las implementaciones de objetos recibir solicitudes de servicio en forma dinámica. Gracias a DSI, una implementación puede ser invocada sin necesidad de contar con su skeleton.

EMCY
Es un COB de CANopen utilizado para notificar un error interno en un dispositivo conectado al bus CAN. Se transmite del dispositivo en el que se presentó la falla a otros dispositivos como mensajes CAN de alta prioridad. Sirve como interrupción o notificación de alerta.

GIOP
General Inter-ORB Protocol (GIOP) es el protocolo de sesión utilizado por CORBA para organizar el diálogo entre las instancias de la ORB que corren en cada máquina. Se trata de un protocolo abstracto que garantiza la interoperabilidad entre instancias de la ORB y que debe ser mapeado sobre el protocolo de transporte de la red subyacente.

IIOP
Internet Inter-ORB Protocolo (IIOP) es el mapeo de GIOP, protocolo abstracto capaz de correr sobre cualquier capa de transporte, sobre TCP/IP. El mapeo incluye la definición de perfiles IIOP, utilizados posteriormente para generar referencias a objetos CORBA.

Instace ID
Es un entero único asignado a cada instancia de una clase en un dispositivo DeviceNet. Se pueden usar 1 o 2 bytes para guardar este valor. Es único dentro del espacio de direcciones MAC ID:Class ID.

ISO/OSI
Open System Interconnect (OSI) es un modelo de referencia para redes de computadoras, basado en capas, propuesto por la International Standards Organization en su estándar ISO 7498. El propósito de cada una de las capas (física, data-link, red, transporte, sesión, presentación y aplicación) es proveer un conjunto bien definido de funciones que permitan la interoperabilidad e interconexión entre distintas computadoras.

LLC
Logical Link Control (LLC) es una subcapa de la capa data-link en el modelo ISO/OSI y en el modelo IEEE 802 (ISO/IEC8802). Las funciones de esta capa consisten en sincronizar los frames transmitidos por la red, hacer cumplir la disciplina de línea y detectar los errores que puedan producirse.

LMT
Servicio de la capa CAN Application Layer (CAL) de CANopen. Permite leer y configurar ciertos parámetros de CAL y de CAN. El servicio involucra el uso de objetos administrativos de Layer Management.

MAC
Medium Access Control (MAC) es una subcapa de la capa data-link en el modelo ISO/OSI y en el modelo IEEE 802 (ISO/IEC8802).. Las funciones de esta capa consisten en determinar cuando una computadora puede acceder el medio compartido usado por la red y cuando puede transmitir por el mismo.

MAC ID
Es un entero único asignado a cada nodo en cada red DeviceNet. Equivale a una dirección de red, asignada manualmente por el administrador del bus. Los valores posibles son de 0 a 63, utilizando 6 bits.

MAU
Medium Attachment Unit (MAU) es una subcapa de la capa física en el modelo ISO/OSI. Está formada por el dispositivo que conecta la computadora al medio usado por la red.

NMT
Servicio de la capa CAN Application Layer (CAL) de CANopen. Su función es la inicialización, configuración y manejo de errores en el bus CAN. El servicio involucra el uso de objetos administrativos de Network Management.

OMG IDL
Object Management Group Interface Definition Language (OMG IDL) es un lenguaje abstracto definido por CORBA que permite describir las interfases de los objetos, en forma independiente al lenguaje usado para la implementación de los mismos.

ORB
Object Request Broker (ORB) es un tipo de middleware que administra la comunicación y el intercambio de solicitudes y respuestas entre objetos, ocultando todos los detalles de implementación y dejando que el desarrollador de aplicaciones distribuidas sólo se preocupe por las interfases de los objetos.

PDO
Process Data Object (SDO) es un COB de CANopen utilizado para el intercambio de datos de proceso o sea, datos de tiempo real. Corresponden a mensajes CAN de alta prioridad.

PLS
Physical Layer Signaling (PLS) es una subcapa de la capa física en el modelo ISO/OSI. Las funciones de esta capa consisten en determinar que señales deben transmitirse por el medio usado por la red, conectado mediante la MAU correspondiente.

POA
Portable Object Adapter (POA) es un componente de la ORB CORBA. Sus funciones son actuar como adaptador entre la ORB e implementaciones de objetos y proveer una interfase estándar entre la ORB y estas últimas.

RMIM
Receiver Message Identifier Mask (RMIM) forma parte del registro mask-and-match de un adaptador BasicCAN. Contiene la máscara que se debe aplicar al ARBITRATION FIELD de un frame CAN antes de compararlo con el RMIS.

RMIS
Receiver Message Identifier Selector (RMIS) forma parte del registro mask-and-match de un adaptador BasicCAN. Contiene el valor contra el que debe comparar el ARBITRATION FIELD de un frame CAN, luego de aplicar la máscara contenida por el RMIM.

SDO

Service Data Object (SDO) es un COB de CANopen utilizado para leer y escribir cualquiera de las entradas del diccionario de objetos de un dispositivo. Corresponden a mensajes CAN de baja prioridad.

Service Code
Es un identificador de un método de instancia o clase en un dispositivo DeviceNet. Los valores de códigos de servicio son de 0 a 127, insumiendo 7 bits.

SYNC
Es un COB de CANopen utilizado para sincronizar los dispositivos conectados al bus CAN, permitiendo coherencia temporal y coordinación entre estos últimos. Existe un dispositivo productor de objetos SYNC y dispositivos consumidores de objetos SYNC.

UCMM
Unconnected Message Manager (UCMM) es un objeto de los dispositivos DeviceNet, encargado de procesar los Unconnected Explicit Messages o sea, mensajes no asociados a una conexión. Permite establecer conexiones de mensajería explícita entre dos dispositivos utilizando este tipo de mensajes.

XER
XML Encoding Rules (XER) son las reglas de codificación basadas en XML, pertenecientes a la capa de presentación definida por el modelo de referencia ISO/OSI en el estándar ISO 8824. Permiten transformar datos abstractos en un stream de bits.