Available online at www.sciencedirect.com
SCIENCE DIRECT®
Information and Software Technology xx (2004) 1–15
INFORMATION AND SOFTWARE TECHNOLOGY
www.elsevier.com/locate/infsof

# Resource conscious development of middleware for control environments: a case of CORBA-based middleware for the CAN bus systems ☆

Seongsoo Hong[a], Tae-Hyung Kim[b,*]

[a]School of Electrical Engineering and Computer Science, Seoul National University, Seoul 151-742, South Korea
[b]Department of Computer Science and Engineering, Hanyang University, 1271 Sa 1-Dong, Ansan, Kyunggi-Do 426-791, South Korea

Received 8 March 2003

**Abstract**

While it is imperative to exploit middleware technologies in developing software for distributed embedded control systems, it is also necessary to tailor them to meet the stringent resource constraints and performance requirements of embedded control systems. In this paper, we propose a CORBA-based middleware for Controller Area Network (CAN) bus systems. Our design goals are to reduce the memory footprint and remote method invocation overhead of the middleware and make it support group communication that is often needed in embedded control systems. To achieve these, we develop a transport protocol on the CAN and a group communication scheme based on the publisher/subscriber model by realizing subject-based addressing that utilizes the message filtering mechanism of the CAN. We also customize the method invocation and message passing protocol of CORBA so that CORBA method invocations are efficiently serviced on a low-bandwidth network such as the CAN. This customization includes packed data encoding and variable-length integer encoding for compact representation of IDL data types.

We have implemented our CORBA-based middleware using GNU ORBit. We report on the memory footprint and method invocation latency of our implementation.


*Keywords:* Distributed and embedded control systems; Customized middleware; Controller area network; Embedded inter ORB protocol; Compact common data representation

## 1. Introduction

It is an extremely tedious and error-prone task to develop software for distributed computer controllers in sophisticated embedded systems such as networked home service robots, high-end passenger vehicles, and numerical control machines [23]. These systems usually consist of a large number of hardware components and heterogeneous microcontrollers and digital signal processors that are interconnected via diverse buses with different network bandwidths. In addition to such architectural complexity, they are often required to host a wide range of embedded application programs. For instance, a mobile home service robot runs a localization and navigation task based on computational vision processing, a user interaction task based on voice recognition, and a motion control task based on feedback servo motor control. Since these tasks may incoporate different algorithms for different application environments, it should be easy to install them onto a given hardware platform and tear down them from it. To this end, these tasks must be able to seamlessly interface with heterogeneous I/O devices as well as network adapters.

These requirements make it very difficult, though not impossible, to design such a distributed embedded control system without support from real-time operating systems, well-defined network protocols, and middleware systems. Among these, middleware is a class of software technologies that specifically address the complexity and heterogeneity inherent in distributed systems. Recently emerging

☆ The work reported in this paper was supported in part by MOST under the National Research Laboratory (NRL) grant 2000-N-NL-01-C-136 and by MCIE grant No. 10006826.
* Corresponding author. Tel.: +82 31 400 5668; fax: +82 31 501 7502.
*E-mail addresses:* sshong@redwood.snu.ac.kr (S. Hong), tkim@cse.hanyang.ac.kr (T.-H. Kim).

0950-5849/$ - see front matter © 2004 Published by Elsevier B.V.
doi:10.1016/j.infsof.2004.09.012

component-based middleware technologies such as CORBA [17], DCOM [4], and Java RMI are the typical examples. However, these technologies cannot be directly applied to embedded control system design without careful customization and tuning since they were originally conceived and developed for use in a general purpose distributed computing environment.

In this paper, we develop a CORBA-based middleware for distributed embedded control systems on the controller area network (CAN) bus. CAN [3] is a well established standard for embedded real-time network substrates that is actively used in the automotive industry worldwide. In designing the new CORBA-based middleware for CAN, we put our emphasis on meeting three key requirements inherent to CAN-based embedded systems. First, the ORB implementation on each processing node should have a small memory footprint not exceeding a few hundred kilobytes. Second, the message traffic per service invocation should be kept low. Note that on the CAN the highest network bandwidth is only 1 Mbps and the payload of each message is only 8 bytes long. Last, the newly designed ORB should support group communication to facilitate easy dissemination of sensory data. The standard CORBA lacks group communication capabilities.

To meet these requirements, we redesign the general inter-ORB protocol (GIOP) into an environment specific IOP (ESIOP) for the CAN bus and define a compact common data representation (CCDR) format. We name the protocol the embedded inter-ORB protocol, or EIOP. We also develop a new transport protocol for CAN to support group object communication. The proposed CORBA-based design is compliant to the object management group (OMG) standard at the interface definition language (IDL) level and strictly follows the guidelines on ESIOP as given by OMG.

### *1.1. Related work*

Three areas in CAN-based systems and middleware come close to our work: (1) high-level protocol designs for CAN, (2) object-oriented modeling schemes on CAN, and (3) group communication supports for the standard CORBA.

Since the CAN standard specifies protocols only up to the data link layer, it lacks high-level protocol services such as distribution of media access identifiers and establishment of communication transports. Thus, it is laborious to build a distributed application, even with modest size and complexity, on the CAN. To address this problem, several commercial, high-level protocol suites have been developed and widely used in industry [1,5,7]. DeviceNet [1] by Allen-Bradley is one of such protocols for the CAN. One of noticeable features of DeviceNet is a high-level abstraction called device profiles. A CAN node in DeviceNet is assigned one of the standard device profiles, e.g. a photoelectric sensor profile, which specifies the type and behavior of a software component in the node. Together with many other features of DeviceNet, device profiles provide a desired level of abstraction for CAN programmers. These profiles are systematically defined by the Open DeviceNet's Vendor Association (ODVA) and distributed to end users by the vendors.

In a distributed real-time control system, it is typical that sensor data are periodically produced without specific requests from its consumers and then disseminated among different controllers. In such an operating environment, subscription-based group communication is more important than connection-oriented point-to-point communication. In the literature, group communication for real-time systems has been well studied on various network media [18,19]. Particularly, in [10,11], Kaiser et al. propose a real-time object invocation scheme and a publisher/subscriber scheme on the CAN 2.0B bus. These are one of seminal attempts to develop systematic paradigms for real-time object models on the CAN. Their approach in [11] uses an abstraction called an event channel, which establishes a virtual connection between publishers and subscribers. Each event channel is identified with a global event tag which takes up 14 bits in the 29-bit CAN 2.0B identifier. The remaining 15 bits are used for a message priority and a node identifier. A drawback of this approach is that it cannot be effectively applied to the CAN 2.0A bus which has only 11-bit identifiers: it would be able to offer at most 64 event channels in CAN 2.0A even if only five bits were used for a message priority and a node identifier. This poses an important practical problem. Note that the CAN 2.0A bus is preferred to the 2.0B bus since the extended 2.0B identifiers increase bus arbitration overhead [1]. Though our approach uses a similar abstraction called an invocation channel, it differs from the event channel since publishers access an invocation channel via their own port. Under a given upper layer protocol, our group communication scheme can support up to 512 ports in CAN 2.0A.

DeviceNet also supports group communication. However, it requires that an explicit bidirectional connection should be established between producers and consumers. Such a bidirectional connection is created by combining two one-way connections in reverse directions. This requirement makes it impossible to support anonymous communication, such as the publisher/subscriber model.

There are several research results which address the group communication problem for CORBA. In [6], Harrison et al. present the implementation of the CORBA event channel [15] for real-time systems. The CORBA event channel is an intermediary object which accepts event data from a supplier and retransmits it to related consumers. The event channel has the responsibility of grouping consumers and multicasting messages to them. It may well increase the communication overhead since every message is transmitted at least twice due to message indirection. Though our approach also relies on an intermediary object for managing subscriber groups, it does not incur message redirections

since each subscriber receives messages directly from a publisher.

In [14], Maffeis presents the Electra ORB (object request broker) which supports reliable multicasts in CORBA. In Electra, required multicast services are provided by sophisticated lower-level toolkits such as Horus and Isis [2,24]. In order to incorporate group communication semantics into Electra ORB, the approach extends the definition of CORBA object references so that object groups are addressed in a uniform manner. While Electra provides valuable features such as group objects, reliable multicast communication, and object replication, it is not appropriate for embedded systems built on a broadcast network with low bandwidth such as CAN.

## 2. Target system hardware model and software abstraction

A typical distributed embedded control system consists of a large number of function control units interconnected by embedded control networks. In this section, we present our distributed embedded control system model with its hardware components, communication abstraction, and software configuration.

### 2.1. Functional control unit

Fig. 1 demonstrates a typical distributed embedded control system which makes the electronic control system of a passenger vehicle. It consists of several functional control units (FCU) which are interconnected by embedded control networks. Each FCU conducts a dedicated control mission by interfacing sensors and actuators and executing prescribed control algorithms. As shown in Fig. 1, it has one or more microprocessors and microcontrollers attached to an on-board system bus. It is also equipped with a bus adaptor which enables the FCU to participate in communication via embedded control networks (ECN).

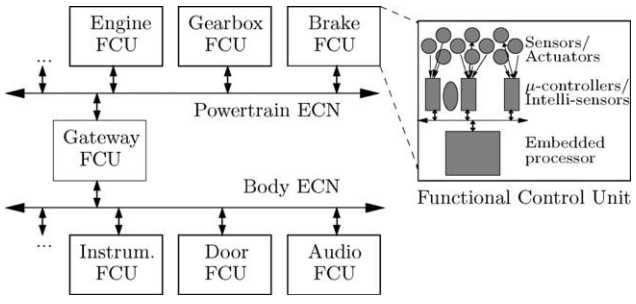


Fig. 1. Example distributed embedded control system: passenger vehicle control system.

### 2.2. Embedded control network

As shown in Fig. 1, embedded control networks (ECN) connect FCUs through inexpensive bus adaptors. Also, they are often required to provide real-time message delivery services, and subject to very stringent operational and functional constraints. In this paper, we have chosen the CAN [9] as our embedded control network substrate since it is an internationally accepted industrial standard satisfying such constraints.

The CAN standard specifies physical and data link layer protocols in the OSI reference model [3]. It is well suited for real-time communication since it is capable of bounding message transfer latencies via predictable, priority-based bus arbitration. A CAN message is composed of identifier, data, error, acknowledgment, and CRC fields. The identifier field consists of 11 bits in CAN 2.0A or 29 bits in 2.0B and the data field can grow up to 8 bytes. When a CAN network adaptor transmits a message, it first transmits the identifier followed by the data. The identifier of a message serves as a priority, and a higher priority message always beats a lower priority one.

The CAN possesses two important characteristics. First, it offers a consistent broadcast mechanism in a straightforward manner via a serial broadcast medium and non-destructive priority-based bus arbitration. Second, it supports the anonymous producer/consumer model of data transmission which is often referred to as the publisher/subscriber communication model [1,11]. In the CAN protocol, a producer of a message is totally unaware of its consumers and simply broadcasts messages over the bus without specifying their destinations. A CAN bus adaptor can be programmed to accept only a specific subset of messages that carry predefined identifier patterns with them. This filtering mechanism, which is made possible via a mask register on a CAN interface chip, allows consumer nodes on the CAN to select desired messages among all the broadcast messages. This addressing method, also known as subject-based addressing [1,11], renders the CAN suited to the publish/subscriber communication model.

In this paper, we intentionally consider only the CAN 2.0A standard. While some CAN controllers support both 2.0A and 2.0B, the 29-bit identifier format gains little support from most of commercial high-level protocol products such as DeviceNet. This is because CAN 2.0B networks incur a compatibility problem with already installed 2.0A networks. More importantly, the extra 18 bits of 2.0B messages increase the bus arbitration overhead and reduce determinism by increasing potential jitter during message transmission.

### 2.3. Two abstract communication channels

To delve into the details of communication protocols, we introduce two abstract communication channels: an *invocation channel* and a *connection*

• **Invocation channels**: Our CORBA-based middleware supports subscription-based, anonymous group communication that is often referred to as a publisher/subscriber scheme [11,19]. In this scheme, a communication session starts when a data producer announces a predefined invocation channel. An invocation channel is a virtual broadcast channel from publishers to a group of subscribers. Data consumers can subscribe to an announced invocation channel. In this announcement/subscription process, neither a publisher nor a subscriber has to know each other. This anonymity allows for easy reconfiguration of control systems. In our middleware, an invocation channel is uniquely identified with a CAN identifier, as will be described later.

• **Connections**: Our middleware also supports a bidirectional connection-oriented point-to-point communication. A connection is a virtual channel which must be established between a client and a server before any message is transmitted between them. Since CAN messages cannot carry destination addresses with them, we design the transport protocol such that a source address is stored and transmitted in the CAN identifier instead. The receiver is informed of this sender address during the connection establishment process, and later registers it into a CAN filter register to accept those messages sent from its connection peer. As a result, CAN messages are passed via a logical unidirectional pipe from a sender to a receiver. A bidirectional connection is easily realized by combining a pair of unidirectional pipes in opposite directions. Thus, a connection establishment process involves exchanging two source addresses between connection peers.

### 2.4. CORBA-based middleware configuration

The proposed CORBA-based middleware stems from the standard CORBA and possesses most of essential components of it. Fig. 2 illustrates layer-to-layer comparison between the standard CORBA and the proposed middleware. Specifically, Fig. 2(b) shows our middleware design. Both architectures consist of four layered components.

At the top of both hierarchies lies an application layer. While the standard CORBA provides the client/server model for application programmers, the proposed one offers the publisher/subscriber model as well. The object abstraction layer just below the application layer encapsulates computational processes into CORBA objects. A CORBA object is a building block, or a component, of a distributed application. While the implementation of a CORBA object is hidden from clients, its services are publically announced through an OMG IDL interface. A client process invokes a CORBA object using an object reference, which serves as a handle used to identify and locate a CORBA object.

At the inter-ORB protocol layer, a remote method invocation is transformed into a networked message representation according to a syntax called common data representation (CDR). The general IOP (GIOP) defines the contents of CORBA messages. For example, a CORBA service request message contains a message header, method parameters, and optional contextual information. The transport layer at the bottom actually delivers these messages over the network.

CORBA is based on the connection-oriented transport model and an object reference denotes only a single CORBA object. In order to extend CORBA to accommodate group communication, we extend the object reference and develop a new publisher/subscriber protocol on the CAN bus. In order to make CORBA affordable on low-bandwidth embedded networks, we customize the GIOP and CDR of the standard CORBA. We summarize the noticeable features of our new CORBA as below.

- **Group object reference**: An object reference in CORBA refers to a single object. It is internally translated into an interoperable object reference (IOR) denoting a communication end-point the object resides on. In our design, an object reference may refer to a group of receiver objects. An intermediary object named

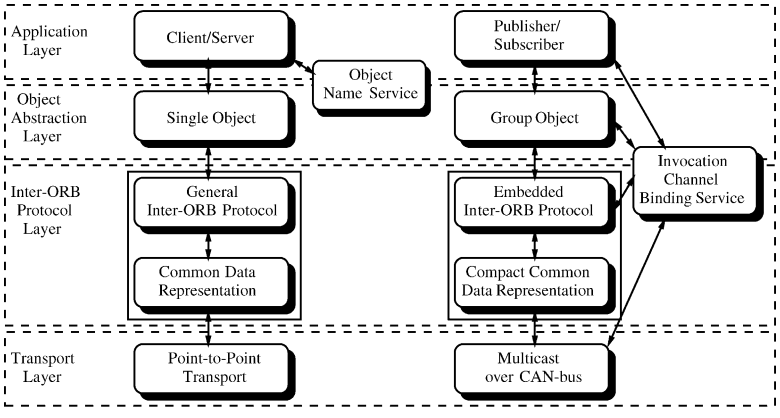


Fig. 2. Comparison with the standard CORBA configuration.

a conjoiner is responsible for managing object groups and implementing the internal representation of their references.

- **CAN-based transport protocol**: A new CAN-based transport protocol is designed to support group communication in our middleware. This protocol makes use of the CAN identifier structure to realize a subject-based addressing scheme, which supports the anonymous publisher/subscriber communication model. In this protocol, a sender is totally unaware of its receivers and simply sends out messages via its own communication port.
- **Publisher/subscriber scheme**: A new publisher/subscriber communication scheme is also designed on top of the transport protocol. This scheme relies on an invocation channel. Since each port is owned by a publisher, this scheme supports the one-way, many-to-many communication model. In this scheme, a conjoiner object takes care of group management, dynamic channel binding, and address translation. An invocation channel is uniquely identified as a channel tag in an IDL program.
- **Compact common data representation (CCDR)**: Common data representation is a syntax which specifies how IDL data types are represented in CORBA messages. In CDR, method invocations often take up tens of bytes in messages. Since a CAN message has only an 8-byte payload, a method invocation may well trigger a large number of CAN message transfers. To deal with this problem, we define the compact CDR. It exploits packed data encoding which avoids byte padding for data alignment, and introduce new data types for variable length integers to encode 4-byte integers in a dense form.
- **Embedded inter-ORB protocol (EIOP)**: In addition to CCDR, we customize GIOP by simplifying messages types and reducing the size of the IOP headers of messages.

## 3. CAN-based transport protocols for middleware

While CORBA relies only on the point-to-point transport service provided by standard protocols such as TCP, distributed computer control systems require both group and point-to-point communication capabilities. In this section, we design a connection establishment protocol for connection-oriented communication and integrate it with the channel binding protocol of the subscription-based communication protocol of our CORBA-based middleware. We first present the transport layer protocol header format which is capable of supporting multiple upper layer protocols. We then explain the conjoiner-based announcement/subscription protocol for subscription-based communication. Finally, we describe the connection establishment protocol.

### 3.1. Defining the protocol header

As described in [12], we use the CAN identifier structure to define the transport layer protocol header Clearly, this is a challenging task since the CAN imposes the following intrinsic constraints.

(1) The CAN identifier is only 11 bits long. Each bit should be used very frugally.
(2) Distinct CAN nodes should not attempt to simultaneously send different messages with the same identifier.
(3) CAN identifiers serve as message priorities during bus arbitration.

In addition to these constraints, the transport protocol should efficiently support multiple upper layer protocols. Thus, we put the greatest emphasis on making efficient use of the bits in the identifier and simplifying the protocol design to warrant the small execution overhead and code size of the protocol stacks as long as it can provide desired services for higher-level CORBA layers.

Fig. 3 shows our protocol header format. We divide the CAN identifier structure into three sub-fields: a protocol ID (`Proto`), a transmitting node address (`TxNode`), and a port number (`TxPort`). They respectively occupy 2, 5 and 4 bits amounting to 11. The `Proto` field denotes an upper layer protocol identifier. The data field following the identifier in a CAN message is formatted according to the upper layer protocol identifier denoted by `Proto`.

In our current design, among four possible values of the `Proto` field, $01_2$ and $10_2$, respectively denote the publisher/subscriber protocol and the client/server protocol of the EIOP. $11_2$ is used for the network management protocol which takes care of both invocation channel binding and connection establishment. In the CAN, a message identifier with a smaller value gets a higher priority during bus arbitration. Consequently, in our middleware, publisher/subscriber messages get precedence over client/server and network management messages. The reason behind this `Proto` assignment is because publisher/subscriber messages usually carry time sensitive sensory data and because starting a new session should not interfere with already admitted real-time message traffic. The remaining $00_2$

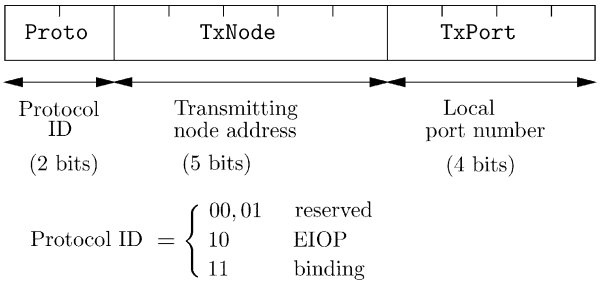


Fig. 3. Protocol header format using CAN identifier structure.

representing the top priority protocol is reserved for a potential user-defined protocol for a high urgency channel. Messages of this protocol can entirely bypass the CORBA object abstraction layer and EIOP protocol stacks, thus incurring very short message transfer latencies.

The `TxNode` field is the address of the transmitting node. In our design, one can simultaneously connect up to 32 distinguishable nodes with the CAN bus under a given upper layer protocol. The `TxPort` field represents a port number that is local to a particular transmitting node. Since `TxNode` serves as a domain name that is globally identifiable all across the network, `TxNode` and `TxPort` collectively make a global port identifier. This allows ports in distinct nodes to have the same port number and helps increase modularity in software design and maintenance. As the `TxPort` field supports the maximum of 16 local ports on each node, up to 512 global ports coexist in the network under a specific upper layer protocol.

Note that the header does not include any form of destination addresses and that receiving CAN nodes can select and accept messages sent from a specific subset of ports, using the message filtering mechanism of the CAN bus adaptor. In this way, subject-based addressing is effectively supported.

### 3.2. Channel binding protocol for subscription-based communication

The key components of the subscription-based communication in our middleware are an invocation channel and a conjoiner. An invocation channel is a virtual broadcast channel from publishers and a group of subscribers. A publisher announces an invocation channel to subscribers after plugging one of its ports into it. As described above, a port is a transport layer entity uniquely addressable with `TxNode` and `TxPort` pair. An invocation channel can have more than one plugged port. A publisher can send a message through an invocation channel via a plugged port and a subscriber can receive the message via the invocation channel.

Our channel binding protocol heavily relies on an intermediary object we name a *conjoiner*. It resides on a CAN node whose node identifier is well-known to every publisher and every subscriber in the system. It must be started right after network initialization and operational during the entire system service period.

#### 3.2.1. Global binding database

The conjoiner maintains a global binding database where each invocation channel has a corresponding entry which is announced and registered by a publisher Fig. 4 illustrates the conjoiner-based publisher/subscriber framework and the global binding database. As shown in the figure, an entry in the global binding database is a quadruple consisting of a *channel tag*, an *OMG IDL identifier*, `TxNode`, and `TxPort`. The channel tag is a unique symbolic name associated with each invocation channel. It is statically defined by programmers when they write the application code. Both publishers and subscribers use it as a search key in the global binding database later on. The OMG IDL interface identifier is a unique identifier associated with each IDL interface in the system. The OMG IDL compiler generates IDL interface identifiers. The CORBA run-time system uses these identifiers to perform type checking upon every method invocation. This ensures strong type safety as required by the CORBA standard. The channel tag and the interface ID together work as a unique name for each invocation channel. It is programmers' responsibility to define a system-wide unique name for an invocation channel.

#### 3.2.2. Channel announcement and subscription

The conjoiner exchanges messages with a publisher for channel establishment and with a subscriber for channel subscription. When a publisher wants to get attached to an invocation channel, it sends a registration message to the conjoiner. The conjoiner responds to it by passing an acknowledge message back to it after updating the global binding database. Similarly, a subscriber sends a message to the conjoiner, requesting subscription to an invocation channel. If the conjoiner finds a matching entry for the requested invocation channel in the global binding database, it provides the subscriber with the corresponding binding information. Note that subscribers may be asynchronously

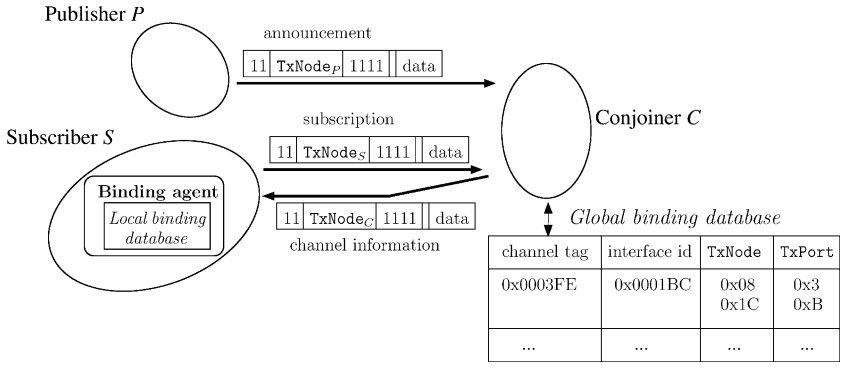


Fig. 4. Conjoiner-based channel binding protocol.

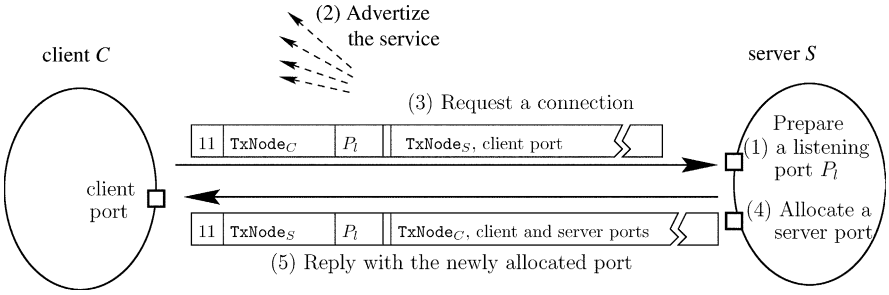


Fig. 5. Connection establishment process.

informed of changes in its subscribed invocation channel as a publisher is attached to, or detached from the channel. A local binding agent denoted by an oval inside `Subscriber` *S* in Fig. 4 takes care of such updates.

Note the conjoiner should be able to accept messages from any CAN nodes in the system. Thus, we reserve the local port number (`TxPort`) $1111_2$ for this purpose under the network management protocol. As shown in Fig. 4, all messages sent to the conjoiner use this local port. Consequently, the conjoiner unconditionally accepts all messages with this port number when Proto is $11_2$. On the other hand, other CAN nodes can accept messages from the conjoiner in a straightforward way since the `TxNode` and `TxPort` of the conjoiner is known a priori.

The data field of a binding message carries either full binding information or an actual query. Specifically, a publisher's registration message contains all necessary information to construct a database entry such as a channel tag, an OMG IDL interface ID, and a global port number. Using this information, the conjoiner either creates an entry or modifies one if it exists. A subscriber's request message contains a channel tag and an IDL interface identifier for an invocation channel. On successful retrieval of one or more entries from the binding database, the conjoiner sends a reply message containing information on these entries. These entries are stored into the local binding database of the subscriber.

### *3.3. Connection establishment protocol for point-to-point communication*

As stated earlier, a bidirectional connection in our approach is made up of a pair of unidirectional pipes. The connection makes use of two local ports, each of which belongs to the source node of each pipe, respectively. In order for a destination node to be able to accept a message from its source, it should know the `TxNode` and `TxPort` pair of the source. This requires that connection establishment be done between two communication end points. The connection establishment process thus involves combining a pair of pipes in opposite directions by exchanging the port address of each end point. The notion of a well-known listening port is used in our scheme. A listening port is a local port belonging to a server. It is advertised to clients wishing to access the server. Just like the conjoiner in the channel binding protocol, a server should be able to accept connection request messages from any CAN nodes. A server programs its CAN filter registers such that it can receive those connection request messages containing its listening port number in their CAN identifiers. Note that a connection request message has $11_2$ in its `Proto` field.

Fig. 5 demonstrates five-step process for connection establishment. It is described below.

(1) A server prepares a listening port.
(2) It advertises this port number along with its node identifier to clients in the system.
(3) A client sends the server a connection request message whose CAN identifier contains the listening port number with `TxNode` being the node identifier of the client. This message contains the server's node identifier and its available local port number in the data field.
(4) Accepting a connection request message, the server allocates a port from its local free port pool.
(5) It replies to the client by passing the newly allocated port number. Note that this port and the client's port will be used during normal communications between the client and the server. These ports will be used with the protocol identifier $10_2$, while the listening port is used with the protocol identifier $11_2$.

## 4. Programming models of two communication schemes

The proposed transport protocols surely affect the programming style of application programs. In this section, we give a detailed account of the programming models of the two communication schemes. We first present the anonymous publisher/subscriber model for subscription-based communication and then the client/server model for connection-oriented communication. We also show two pieces of source code as examples of the two communication models.

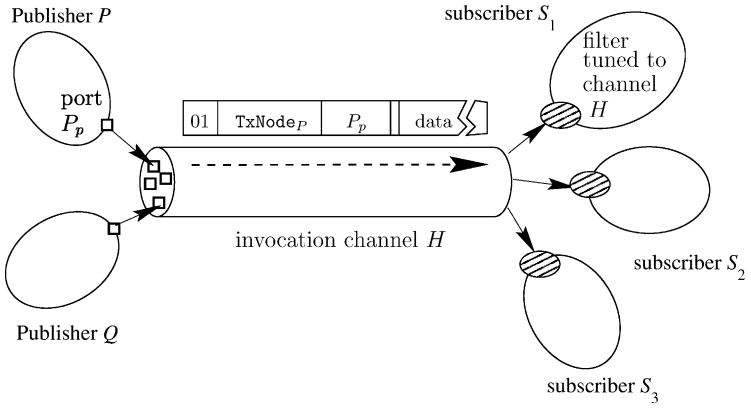


Fig. 6. Subscription-based communication via an invocation channel.

### 4.1. Anonymous publisher/subscriber communication

Fig. 6 pictorially illustrates an abstract invocation channel and the plugged ports of the publishers. A message transmitted over an invocation channel carries $01_2$ as its protocol identifier and its source node and local port numbers in the message identifier. The publisher/subscriber model in our scheme possesses the following properties.

- An invocation channel is defined as an interface in OMG IDL code by a programmer who writes the publisher's code. This interface specifies the signature of the exported methods that subscribers must implement. Unlike in the client/server model, the interface is exported to subscribers. Publishers call an exported method later on to send a message to subscribers.
- It offers anonymous communications from publishers to subscribers. Subscribers need to know only the channel tag and the IDL interface identifier of an invocation channel they want to subscribe to. They need not know the exact identity of the publisher.
- Unlike the event services of the standard CORBA, it does not require an intermediate object to forward messages from publishers to subscribers. In our middleware, subscribers keep all binding information of subscribed invocation channels in their local binding databases. This allows for efficient message transfers.

As an example, we show a publisher/subscriber program which performs temperature sampling and updating. It consists of an IDL interface definition given in Fig. 7 and subscriber/publisher code in Fig. 8. The IDL code defines the interface of an invocation channel. It specifies the signature of method `update_temperature( )`. The method is invoked by a publisher and then executed in a subscriber to update temperature values in the subscriber object. It is declared as a `oneway` operation which does not produce output values.

Fig. 8 shows two source code files that correspond to a publisher and a subscriber, respectively. Each of the source files contains a unique channel tag `TEMP_MONITOR_TAG` and an IDL interface identifier `TEMP_MONITOR_IFACE`. Note that `TEMP_MONITOR_TAG` is defined by programmers, while `TEMP_MONITOR_IFACE` is generated by our OMG IDL compiler.

In Fig. 8, a publisher wishing to distribute its data via an invocation channel accesses the conjoiner object to obtain an object reference to a group of subscribers. This is performed through a call to `Conjoiner::announce( )` method of the conjoiner. The conjoiner is a pseudo CORBA object that performs house-keeping tasks for the sake of the publisher. It allocates a local port from the publisher's free port pool and send an announcement message to the conjoiner. During this call, it provides the conjoiner with `TEMP_MONITOR_TAG` and `TEMP_MONITOR_IFACE`. Finally, the publisher invokes the `update_temperature` method to send out a message.

Similarly, a subscriber wishing to subscribe to an invocation channel accesses the conjoiner object via `Conjoiner::subscribe( )` method. During this call, the subscriber provides the conjoiner with `TEMP_MONITOR_TAG` and a servant. A servant is a collection of language-specific data and procedures that implement the actual object body. It is written by an application programmer and registered into the CORBA object system via a portable object adaptor (POA). Note that the `TEMP_MONITOR_IFACE` is not explicitly provided during a call to `Conjoiner::subscribe( )` method since the `monitor_servant` is a typed object whose interface information can be easily extracted.

```
// IDL

interface TemperatureMonitor {
    // Update temperature value for a location.
    oneway void update_temperature(
                  in char locationID, in long temperature);
}
```

Fig. 7. IDL definition for subscriber interface.

```
// Publisher code in C++
// Define a channel tag for temperature monitoring.
#define TEMP_MONITOR_TAG 0x01

// Initialize the object request broker (ORB).
CORBA::ORB_ptr orb = CORBA::ORB_init(argc,argv);

// Get a reference to the conjoiner.
Conjoiner_ptr conjoiner = Conjoiner::_narrow(
    orb->resolve_initial_reference("Conjoiner"));

// Obtain a reference to the temperature monitor group TEMP_MONITOR_IFACE
// E is an interface identifier generated by the IDL compiler.
TemperatureMonitor_ptr monitor =
    conjoiner->announce(TEMP_MONITOR_TAG, TEMP_MONITOR_IFACE);
while(1) { // Invoke a method of subscribers.
    ...
    monitor->update_temperature('A', value);
}

// Subscriber code in C++
// Define a channel tag for temperature monitoring.
#define TEMP_MONITOR_TAG 0x01

// Initialize the object request broker (ORB).
CORBA::ORB_ptr orb = CORBA::ORB_init(argc,argv);

// Get a reference to the conjoiner.
Conjoiner_ptr conjoiner = Conjoiner::_narrow(
    orb->resolve_initial_reference("Conjoiner"));

// Create a servant implementing a temperature monitor object.
TemperatureMonitor_impl monitor_servant;

// Assign a local CORBA object name to the monitor object.
PortableServer::ObjectId_ptr oid =
     PortableServer::string_to_ObjectId("Monitor1");

// Register the object name and servant to a portable object adaptor (POA).
poa->activate_object_with_id(oid, &monitor_servant);

// Bind the monitor object to the TEMP_MONITOR_TAG.
conjoiner->subscribe(TEMP_MONITOR_TAG, &monitor_servant);

// Enter the main loop where the monitor can receive temperature values.
orb->run();
```

Fig. 8. Publisher/subscriber example: temperature monitor.

`Conjoiner::subscribe()` method sends a subscription request message to the conjoiner to get the binding information of an invocation channel. Finally, the subscriber enters into a blocking loop where it waits for an invocation of the `update_temperature()` method from the publisher.

### 4.2. Client/server communication

CORBA puts distributed object-based computing in a familiar client/server perspective. Though subscription-based communication is natural and suitable for control applications, our middleware needs to support

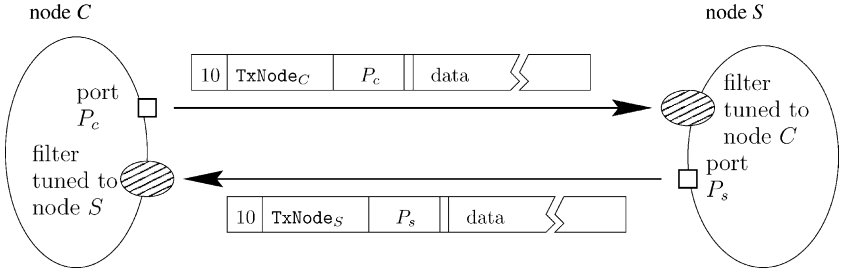


Fig. 9. Point-to-point communication via two unidirectional pipes.

```
// IDL

interface Door {
    // Update temperature value for a location.
    boolean is_open();
}
```

Fig. 10. IDL definition for door interface.

connection-oriented communication to improve interoperability with the standard CORBA. Fig. 9 pictorially depicts the connection-oriented communication via two unidirectional pipes. The messages transmitted over the connection have $10_2$ in `Proto` field to denote the connection-oriented communication.

As an example of a client/server application, we present an application program consisting of an IDL interface definition and client/server code. This program is fairly compatible with the standard CORBA since the client/server model is supported in CORBA. The server of the program works as a vehicle door manager. The client may ask if a door is open to react properly to external conditions, for example, by alarming passengers if the vehicle starts to move with the door open. Fig. 10 gives an IDL specification containing the signature of method `is_open()` that returns the door state managed by the server. Fig. 11 shows two pieces of source code that correspond to a client and a server, respectively. The server code is similar to subscriber code except that it does not subscribe to the conjoiner. It creates a servant by constructing an object reference. The object reference contains a listening port and a CAN node identifier of the server. The server encodes in into an inter-operable object reference (IOR) string. Later, the client may obtain it using the standard CORBA name service.

## 5. Extending embedded inter-ORB protocol

Remote method invocation in CORBA is handled through the general inter-ORB protocol (GIOP).

```
// Client code in C++

// Define an inter-operable object reference (IOR) string for
// the doorA object.
static char* doorA_IOR = "IOR:001F3E ... ";

// Initialize the object request broker (ORB).
CORBA::ORB_ptr orb = CORBA::ORB_init(argc,argv);

// Obtain a reference to the doorA object
Door_ptr doorA = orb->string_to_object(doorA_IOR);

while(1) {
    ...
    // Invoke a method on doorA.
    if (doorA->is_open()) {
       ...
    }
}

// Server code in C++

// Initialize the object request broker (ORB).
CORBA::ORB_ptr orb = CORBA::ORB_init(argc,argv);

// Create a servant implementing a door object.
Door_impl doorA_servant;

// Assign a local CORBA object name to the door object.
PortableServer::ObjectId_ptr oid =
    PortableServer::string_to_ObjectId("DoorA");

// Register the object name and servant to a portable object adaptor (POA).
poa->activate_object_with_id(oid, &doorA_servant);

// Extract an object reference from the doorA servant
Door_ptr doorA = poa->servant_to_reference(&doorA_servant);

// Encode the object reference into an inter-operable object
// string (IOR)
char* doorA_IOR = orb->object_to_string(doorA);

// Handle requests for object doorA
orb->run();
```

Fig. 11. Client/server example: vehicle door manager.

Table 1
Variable length integer encoding

| Two MSBs | Size (bytes) | Max. value (unsigned) |
|---|---|---|
| 00 | 1 | $2^6$-1 |
| 01 | 2 | $2^{14}$-1 |
| 10 | 3 | $2^{22}$-1 |
| 11 | 5 | $2^{32}$-1 |

The CORBA 2.2 GIOP defines a transfer syntax called common data representation (CDR) and 8 message types that cover all request/reply semantics of CORBA. However, the GIOP is not suitable for our CORBA-based middleware since it triggers too many CAN message transfers every method invocation. In [12], we defined a new transfer syntax we named the compact common data representation (CCDR), and made our inter-ORB protocol selectively support a subset of the message types of GIOP. We called this new inter-ORB protocol the embedded inter-ORB protocol (EIOP). Although we could effectively reduce message traffic through the use of the EIOP, we seriously sacrificed the interoperability of our EIOP. In this section, we briefly review the CCDR of our middleware and present the design of EIOP messages to support both subscription-based and connection-oriented communications and improve its interoperability.

### 5.1. Compact common data representation

CDR is a transfer syntax which maps data types defined in OMG IDL into a networked message representation so that GIOP sends IDL data types over the network. The key optimizations of our CCDR over CDR are packed data encoding and variable-length integer encoding. Unlike CDR, the packed data encoding scheme of CCDR does not require integer instances be aligned on 32-bit integer boundaries. This greatly saves padding bytes. In the variable-length integer encoding of CCDR, an integer occupies 1–5 bytes depending on the actual value it represents, as summarized in Table 1. While an integer is stored in 4 bytes in CDR, most of integer instances in IDL programs are smaller than $2^{32}$-1. For example, in CDR, integers are very frequently used to represent the sizes of `string` and sequence data types of IDL that are very small in size.

Fig. 12 illustrates the saving when method invocation `foo->op('c',1234,'d',"Hi")` is encoded in CCDR. In this example, we can save six padding bytes needed to align two integers `1234` and `3` in CDR. (Integer `3` is internally used to specify the string length.) Extra 5 bytes are saved through the variable-length encoding scheme. As a result of the two schemes, one can save 11 bytes in total in this simple method invocation, and the method invocation can fit in a single CAN message.

This packed encoding scheme may increase the processing overhead of message encoding and decoding and require extra buffer space on nodes. This drawback can be compensated if the encoded message fits in a single CAN message, which is often the case in an embedded control system.

### 5.2. EIOP messages

In CORBA, method invocations are converted into messages that are transmitted over the network. Table 2 shows 8 message types supported by CORBA 2.2 GIOP and Publish message type supported only in EIOP. A `Publish` message carries data asynchronously sent by a publisher in publisher/subscriber communication. In our original design, `Request` messages were used for publishing data [12]. Thus, from Table 2, we see that the original design that offered only the publisher/subscriber model supported only the `Request` message type. In contrast, the new middleware design supports five message types. Unsupported GIOP message types such as `LocateRequest` and `LocateReply` are meaningful only for migration-enabled CORBA objects. We choose not to support them since embedded control systems seldom use dynamic object migration.

In order to further reduce CAN message traffic, it is essential to cut down the length of common and type-specific message headers. Thus, we reduce the length of the `Publish` and `Request` message headers.

### 5.3. Optimizing EIOP message headers

In CORBA, every message transmitted over the network starts with a GIOP header A GIOP header is subdivided into a 12-byte common prefix and a type-specific header which varies in size depending on a message type. In order to minimize CAN message traffic, it is thus essential to reduce

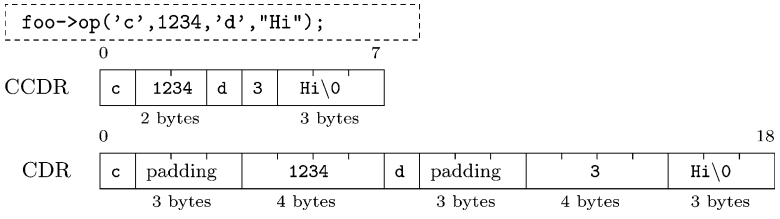


Fig. 12. Example CCDR encoding.

Table 2
Supported EIOP message types

| Message type | EIOP publisher/subscriber protocol | EIOP point-to-point protocol | Standard GIOP | Originator |
|---|---|---|---|---|
| Publish | yes | no | no | Client |
| Request | no | yes | yes | Client |
| Reply | no | yes | yes | Server |
| Cancel Request | no | yes | yes | Client |
| Close Connection | no | yes | yes | Server |
| Message Error | no | yes | yes | Client or Server |
| Locate Request | no | no | yes | Client |
| Locate Reply | no | no | yes | Server |
| Fragment | no | no | yes | Client or Server |

the length of common and type-specific message headers. Thus, we cut down the length of the `Publish` and `Request` message headers. Note that messages of these types are most frequently used in systems since they carry method invocation messages. Fig. 13 shows both the common prefix and the type-specific header of the `Request` message in GIOP. Since the header is included in every `Request` message, it is crucial to reduce its size. We first modify the common prefix by reducing the 4-byte magic field into 1-byte magic code and merging `GIOP_version`, `flags`, and `message_type` fields into the 1-byte `flags`. In Fig. 14, `MessageHeader_1_0` defines the new header format.

We also modify the type-specific headers of `Publish` and `Request` messages as follows. First, we remove optional and reserved fields such as `service_context` and `requesting_principal`. They are used to store information required only when add-on services such as `Security` and `Transaction` are provided. Second, we encode name strings into integer identifiers. As shown in Fig. 13, `RequestHeader_1_1` includes string fields such as `object_key` and operation. The `object_key` field contains an interface name, an object name, an object adaptor name, etc. and the operation field does the method name. Since programmers tend to use long and self-explanatory strings for these names to enhance the readability of programs, string fields in a request message header may well occupy excessively large space. We use integer-encoded `interface_id` and `operation_id` fields in EIOP. EIOP relies on the IDL compiler to obtain proper identifers for them. Finally, we remove the `request_id` field from the `Publish` header since the publisher/subscriber protocol does not support the `Reply` messages. We also remove the `response_expected` field from the `Request` header. This field is meaningful only for the dynamic invocation interface (DII) but we do not support it in our middleware.

As a result, our EIOP header is significantly smaller in size than the GIOP header. As shown in Fig. 14, an EIOP header takes up 5–17 bytes (1–3 CAN messages).

```
module GIOP {
  ...
  struct MessageHeader_1_1 {
    char                    magic[4]; // The string "GIOP"
    Version                 GIOP_version;
    octet                   flags;
    octet                   message_type;
    unsigned long           message_size;
  };
  struct RequestHeader_1_1 {
    IOP::ServiceContextList service_context;
    unsigned long           request_id;
    boolean                 response_expected;
    octet                   reserved[3];
    sequence<octet>         object_key;
    string                  operation;
    Principal               requesting_principal;
  };
}
```

Fig. 13. GIOP message format.

## 6. Experiments and performance results

We have implemented our CORBA-based middleware using GNU ORBit version 0.4.3 [20] on top of the mArx real-time operating system [21] that we have developed at Seoul National University. The mArx real-time operating system is a very lightweight kernel designed for MMU-less embedded microprocessors. It has a small memory footprint not exceeding 50 kB and which increases to 350 kB when TCP/IP protocol stacks are incorporated. It provides a bitmap based preemptive priority scheduler that is ideal for hard real-time multitasking. We replaced the GIOP and CDR of ORBit with our EIOP and CCDR libraries, and aggressively down-sized ORBit to make it conformant to the OMG minimum CORBA specification [16].

The target hardware consisted of three PCs equipped with 40 MHz i386 EX embedded processors and KVASER's PCcan interface boards with Intel 82527 CAN controllers. Such a low performance hardware configuration is typical of embedded control systems. We wrote a CAN interface driver and incorporated it into the mArx real-time operating system. The raw data transfer rate of our CAN bus was 1 Mbps. Since we had full control over the mArx source code as well as device drivers, we could insert instrumentation code to measure the performance of the proposed middleware. Table 3 summarizes the hardware and software platform for our implementation.

### 6.1. Performance metrics

Our CORBA-based middleware had two important design goals: (1) reducing the amount of message traffic

```
module EIOP {
  ...
  struct MessageHeader_1_0 {
    octet           magic; // 0xE0
    octet           flags; // Includes bit fields for
                           // version number and message type.
    unsigned short   message_size;
  };
  struct PublishHeader_1_0 {
    unsigned long   interface_id;
    unsigned long   operation_id;
  };
  struct RequestHeader_1_0 {
    unsigned long   interface_id;
    unsigned long   operation_id;
    unsigned long   request_id;
  };
  struct ReplyHeader_1_0 {
    unsigned long   request_id;
    ReplyStatusType reply_status;
  };
}
```

Fig. 14. EIOP message format.

required for each CORBA method invocation, and (2) minimizing the memory requirement of the ORB. While the simplified message headers of the EIOP contribute to reducing method invocation latencies, the CCDR incurs an extra processing overhead for unpacking and realigning integers. This might pose a critical problem in embedded control systems built with slow microcontrollers such as i386 EX embedded processors. We thus used the following performance metrics for the analysis of our CORBA-based middleware implementation.

- **Protocol processing latency**: In our CORBA-based middleware, the saving in message traffic is partially converted into the increased protocol processing overhead including marshaling and unmarshaling of the EIOP messages. The protocol processing latency on the sender side is defined as the execution time of the invocation stub, the CAN device driver, and the 82527 CAN controller. The protocol processing latency on the receiver side is defined as a time interval from when the first CAN message of a CORBA method invocation is received to when the skeleton code is dispatched.
- **Static memory footprint**: We measured the static memory requirement of our CORBA-based middleware. It is defined as the sum of the sizes of code and data sections of the ORB core and its accompanying library. The GNU glib V1.2.1 is such a library for our CORBA-based middleware and ORBit, and the ACE library is for the TAO ORB.

### 6.2. EIOP protocol processing latencies

We measured the protocol processing latencies both on the publisher side and on the subscriber side. We summarize the measurement results in Fig. 15. During our measurements, we used the `TemperatureMonitor` code shown in Fig. 8. Recall that the `update_temperature()` method in the code has two parameters of types `char` and `long`. Note that the protocol processing latency of a method invocation increases as the number of parameters increases. Thus, we measured different protocol processing latencies varying the number of parameters of the method. Even for a fixed number of parameters, the latency may vary depending on parameter values since CCDR uses the variable-length integer encoding scheme. During our measurements, we used the largest possible values for the parameters to obtain the worst-case latencies.

As shown in Fig. 15, the worst-case protocol processing latencies are less than 1 ms when the number of parameters are reasonably small, specifically, six. This is typical in most field bus applications of practical interest [22]. More importantly, the pure EIOP processing latency takes up only 34.5% of the entire sender side protocol processing latency, whereas the CAN device driver and the bus adaptor take up 24.6 and 40.9%, respectively. Our measurements also show that the EIOP yielded 37.5% reduction in the GIOP message traffic on the average.

Table 3
Hardware and software platforms for our CORBA-based middleware implementation

| |
|---|
| *Hardware* |
| 40 MHz Intel 386 EX embedded processor (no cache) KVASER's PCcan CAN bus adaptor 2.0 [13] (Intel 82527 CAN controller [8]) |
| *Software* |
| mArx real-time operating system our CORBA-based middleware (based on GNU ORBit 0.4.3) KVASER's PCcan device driver (ported onto mArx) |

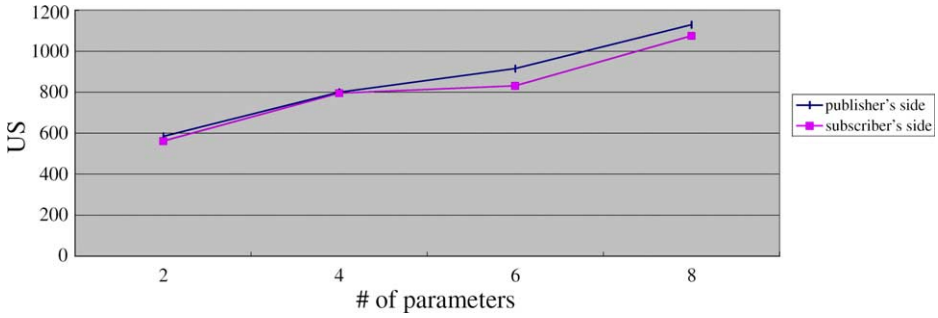


Fig. 15. Protocol processing latency.

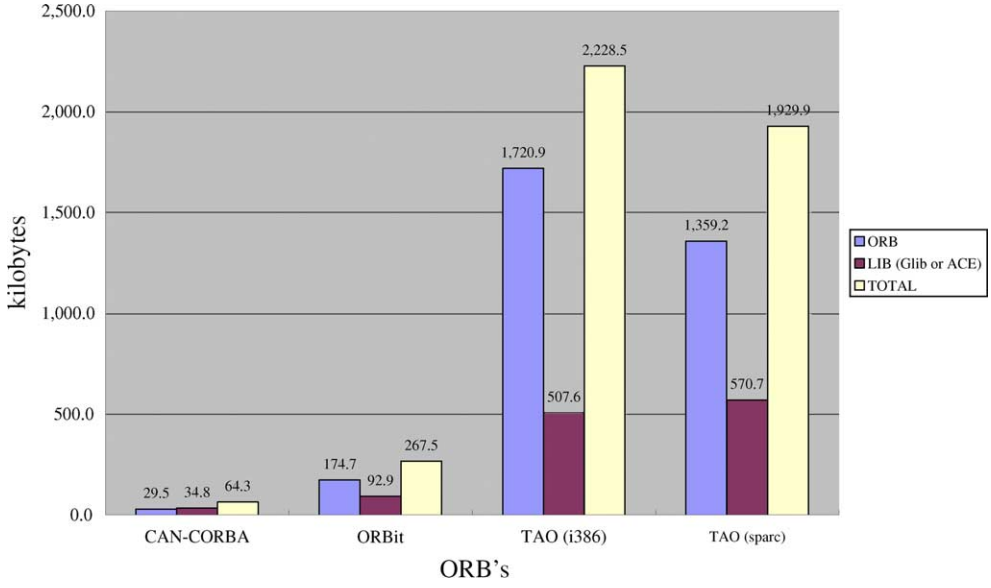


Fig. 16. Static memory requirements of three ORBs.

### 6.3. Static memory requirement

We measured the static memory requirements of our CORBA-based middleware, GNU ORBit V0.4.3, and minimum TAO V1.0 by running the GNU `size` utility. We give the measured memory sizes in Fig. 16. Table 4 illustrates the reason for the big reduction in the figure. We obtained these numbers by compiling the three ORB implementations with GNU C compiler V2.8.1 targeted to Intel 386 processor. We used `-O3 -m386 -frepo` as compiler options. The ORBit and the minimum TAO for i386 were built on Redhat Linux 5.1, while our CORBA-based middleware was built on mArx. Note that `TAO (sparc)` in Fig. 16 denotes the memory size of TAO hosted on the Sun Sparc workstation (Table 4).

## 7. Conclusion

We have presented the design and implementation of a CORBA-based middleware for distributed embedded systems built on the CAN bus. The design goal we had in our mind during the development of our middleware was to minimize its resource demand and to make it support anonymous publisher/subscriber communication without losing the IDL level compliance to the OMG standards. To achieve these goals, we have developed a transport protocol on the CAN and a group communication scheme based on the well-known publisher/subscriber model. This transport protocol makes efficient use of the CAN identifier

Table 4
Memory requirements of our CORBA-based middleware and ORBit modules

| Modules | Footprint (bytes) | | Supported by Minimum CORBA |
|---|---|---|---|
| | CAN-CORBA | ORBit | |
| IOP | 7493 | 19,305 | yes |
| DII/skeleton | 0 | 78,026 | no |
| Dynamic any | 0 | 32,932 | no |
| POA | 8260 | 10,618 | partially |
| Others | 13,770 | 33,776 | yes |
| Total | 29,523 | 174,657 | |

structure to realize a subject-based addressing scheme, which supports the anonymous publisher/subscriber communication model. In the proposed communication scheme, publishers are completely unaware of its subscribers and simply send out messages via their own communication ports. This scheme uses an invocation channel to establish a virtual broadcast channel which connects publishers and a group of subscribers.

We have also customized GIOP and CDR so as to reduce message traffic generated for each method invocation. Specifically, we have defined the compact CDR which exploits the packed data encoding scheme and the variable-length integer representation. In addition to the CCDR, we have simplified messages types and reduced the size of the header of GIOP messages. We have shown that the proposed EIOP along with CCDR contributes to significantly reducing the size of request messages. In spite of these vast modifications, the new CORBA is still compliant to CORBA at the application program and IDL level.

However, reduction in message traffic may lead to increase in the protocol processing overhead due to the integer unpacking and re-alignment overhead of the CCDR. This might pose a serious performance problem to a distributed control system built with low-end microcontrollers such as i386 EX embedded processors, unless it were not controlled. To validate our CORBA-based middleware design from the performance point of view, we implemented it on a CAN-based distributed platform and conducted several experiments and measurements. The experimental results showed that it incurred small method invocation latencies (700 μs on the average) on the sender side requiring only 64.3 Kbytes of memory. Our experiments clearly demonstrated that it would be feasible to use CORBA in developing distributed embedded control systems possessing severe resource limitations.

We are currently looking to extend our CORBA-based middleware so that it can provide timeliness guarantees for real-time messages and fault-tolerance for the centralized conjoiner object. These are challenging tasks due to the size limitation of the CAN 2.0A identifier structure.

## References

[1] Allen-Bradley. DeviceNet Specifications, Release 2.0, vol. I: Communication Model and Protocol, vol. II: Device Profiles and Object Library, 1997.

[2] K. Birman, R. Van Renesse, Reliable Distributed Computing with the Isis Toolkit, IEEE Computer Society Press, 1994.

[3] Bosch, CAN Specification. Version 20, 1991.

[4] N. Brown, C. Kindel. Distributed Component Object Model Protocol-DCOM/1.0, 1998.

[5] CAN in Automation (CiA). Draft Standard 301 Version 3.0, CANopen, Communication Profile for Industrial Systems based on CAL.

[6] T. Harrison, D. Levine, D. Schmidt. The design and performance of a real-time CORBA event service. In Conference on Object-oriented Programming, Systems, Languages and Applications, 1997.

[7] Honeywell Inc. Micro Switch Specification: Application Layer Protocol Specification, Version 2.0, 1996.

[8] Intel Corporation. 82527 Serial Communications Controller Architecture Overview, January 1996.

[9] ISO-IS 11898, Road Vehicles—Interchange of Digital Information—Controller Area Network (CAN) for High Speed Communication, 1993.

[10] J. Kaiser, M. Livani. Invocation of real-time objects in a CAN bus-system, in: IEEE International Symposium on Object-oriented Real-time distributed Computing, pp. 298–307, May 1998.

[11] J. Kaiser, M. Mock. Implementing the real-time publisher/subscriber model on the controller area network (CAN), in: IEEE International Symposium on Object-oriented Real-time distributed Computing, pp. 172–181, May 1999.

[12] K. Kim, G. Jeon, S. Hong, S. Kim, T.-H. Kim. Resource-conscious customization of CORBA for CAN-based distributed embedded systems, in: IEEE International Symposium on Object-Oriented Real-Time Computing, pp. 34–41, March 2000.

[13] Kvaser AB. PCcan 2.0, September 1998.

[14] S. Maffeis. Adding group communication and fault-tolerance to CORBA, in: USENIX Conference on Object Oriented Technologies, 1995.

[15] Object Management Group, Event Service Specification, Version 1.0, formal/01-03-01, March 2001.

[16] Object Management Group, Minimum CORBA Specification, Version 1.0, formal/02-08-01, August 2002.

[17] Object Management Group, The Common Object Request Broker Architecture: Core Specification, Version 3.0.3, March 2004.

[18] G. Pardo-Castellote, S. Schneider. The network data delivery service: Real-time data connectivity for distributed control applications, in: IEEE International Conference on Robotics and Automation, May 1994.

[19] R. Rajkumar, M. Gagliardi, L. Sha. The real-time publisher/subscriber inter-process communication model for distributed real-time systems: design and implementation, in: IEEE Realtime Technology and Application Symposium, June 1995.

[20] Red Hat Inc. ORBit, http://www.labs.redhat.com/orbit, 1999.

[21] Y. Seo, J. Park, S. Hong. Efficient user-level I/O in the ARX real-time operating system, in: ACM Workshop on Languages, Compilers, and Tools for Embedded Systems, June 1998.

[22] G. Ulloa. Fieldbus application layer and real-time distributed systems, in: IEEE International Conference on Industrial Electronics, Control and Instrumentation, IECON, October 1991.

[23] H. Utz, S. Sablatnog, S. Enderle, G. Kraetzschmar, Miro-Middleware for mobile robot applications, IEEE Transactions on Robotics and Automation 18 (4) (2002) 493–497.

[24] R. Van Renesse, K.P. Birman. Protocol composition in horus, in: ACM Symposium on Principlesof Distributed Computing, August 1995.