Building Light Weight CORBA Based Middleware for the CAN Bus Systems

Seongsoo Hong

Abstract: The software components of embedded control systems get extremely complex as they are designed into distributed systems consisting of a large number of inexpensive microcontrollers interconnected by low-bandwidth real-time networks such as the controller area network (CAN). While recently emerging middleware technologies such as CORBA and DCOM address the complexity of distributed programming, they cannot be directly applied to distributed control system design due to their excessive resource demand and inadequate communication models. In this paper, we propose a CORBA-based middleware design for CAN-based distributed embedded control systems. Our design goal is to minimize its resource need and make it support group communication without losing the IDL (interface definition language) level compliance to the OMG standards. To achieve this, we develop a transport protocol on the CAN and a group communication scheme based on the well-known publisher/subscriber model. The protocol effectively realizes subject-based addressing and supports anonymous publisher/subscriber communication. We also customize the method invocation and message passing protocol, referred to as the general inter-ORB protocol (GIOP), 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 on the mArx real-time operating system we have developed at Seoul National University. Our experiments clearly demonstrate that it is feasible to use CORBA in developing distributed embedded control systems possessing severe resource limitations. Our design clearly demonstrates that it is feasible to use a CORBA-based middleware in developing distributed embedded systems on real-time networks possessing severe resource limitations.

Keywords: CORBA, CAN, IDL, GIOP, real-time operating system, middleware, protocol

I. Introduction

There is a growing demand for distributed computer control in sophisticated embedded control systems such as high-end passenger vehicles, numerical control machines, and avionics fly-by-wire systems. These systems are often equipped with tens of microcontrollers which oversee diverse functional units connecting hundreds, sometimes thousands, of analog and digital sensors and actuators. There are significant merits in designing such complex embedded control systems in a distributed fashion. First, it is more cost effective to build an embedded control system with several customized, inexpensive microcontrollers than to do so with a single high performance microprocessor. For example, a passenger vehicle consists of various functional components including engine control, anti-lock brake, and cruise control units. Since each of these units requires specific functionalities such as digital signal processing or interrupt driven event processing, functional units with dedicated microcontrollers can reduce the overall hardware cost. Second, a distributed embedded control system is more reliable than a centralized one since it is possible to isolate the breakdown of one subsystem from others in a distributed control system.

Unfortunately, such benefits come with a serious cost-- increased software complexity. This makes software systems in a distributed embedded control system get extremely complicated to handle the added complexity as well as inherent one. Note that embedded control software must operate in a harsh environment, run on a wide variety of microcontrollers, and interface with heterogeneous I/O devices. Thus, it is very difficult, though not impossible, to design a distributed embedded control system without supports from real-time operating systems, well-defined network protocols, and component-based middleware systems.

Such software complexity can be partially addressed with recently emerging component-based middleware technologies such as CORBA [5], DCOM [3], and Java RMI. They can provide embedded system designers with platform independence and component reuse through interface inheritance and software bus abstraction mechanisms. 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 propose a CORBA-based middleware design for distributed embedded control systems on the CAN (controller area network) bus. The CAN [2] is a rapidly emerging standard for embedded real-time network substrates and widely used in the automotive industry worldwide. In designing the new CORBA-based middleware, we put our emphasis on meeting three key requirements inherent to the 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 eight bytes long. Last, the 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

*The work reported in this paper was supported in part by MOST under the National Research Laboratory grant and by Automatic Control Research Center (ACRC).

Seongsoo Hong: School of Electrical Engineering and Computer Science, Seoul National University, Seoul 151-742, Korea. (sshong@redwood.snu.ac.kr.)

(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 on the CAN to support group object communication. The proposed CORBA design is compliant to the OMG (object management group) standard at the IDL (interface definition language) level and strictly follows the guidelines on ESIOP as given by OMG.

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, 9, 8]. DeviceNet 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 [15, 16]. Particularly, in [12, 13], 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 [13] 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 communication, such as the publisher/subscriber model. This requirement makes it impossible to support anonymous communication such as the publisher subscriber model.

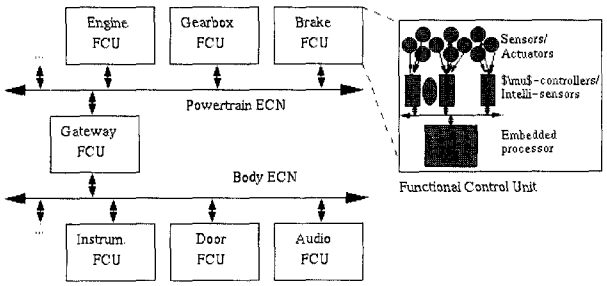


Fig. 1. Example distributed embedded control system: Passenger vehicle control system.

2. Advantages of CORBA-based Design

As mentioned earlier on, there are several advantages in designing distributed computer control systems with the CORBA-based middleware. First, the CORBA ORB separates the details and complications of communication hardware and protocols from application design, thus helps programmers focus only on application-speciffic issues. As a specific example, consider the marshaling process automatically performed by the CORBA-based middleware. The ORB translates data types in application code into network message types and further translates them into CAN messages. Consequently, application programmers need to pay little attention to CAN speciffic issues. Second, the CORBA-based middleware approach allows for the component-based design of distributed computer control systems, and this renders the resultant systems reconfigurable and extensible. Observe that programmers can easily extend an existing system by adding a new subscriber component to the system as long as they can access the specifications of provided services and locate the producers of these services. As will be clear in the remainder of this paper, the CORBA-based middleware maintains a service locator named a conjoiner and keeps the service specifications in the form of interface definitions. Third, the CORBA-based middleware improves interoperability between distributed computer control systems and supervisory computer systems through the well-defined message types of CORBA. As discussed in Subsection 1.1 the legacy CAN protocols alone cannot provide such advantages since they lack high-level protocol services above the data link layer. While commercial protocol suites such as DeviceNet were developed to provide high-level abstractions, they can only partially achieve interoperability and composability since they were designed to fit into only the CAN protocol and to rely on static service bind-

ing. As a result, users require a gateway to connect a distributed computer control system with a supervisory computer system.

II. Target System Hardware Model

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.

1. Functional Control Unit

Figure 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 Figure 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).

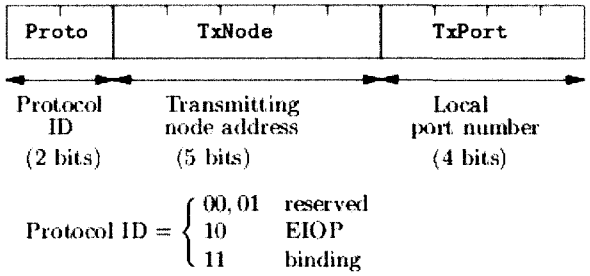


Fig. 2. Protocol header format using CAN identifier structure.

2. Embedded Control Network

As shown in Figure 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 [11] 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 [2]. 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 eight 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 straight- forward 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 [13, 1]. 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 [13, 1], 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.

III. Defining the Protocol Header

While CORBA relies on the point-to-point transport service provided by standard protocols such as TCP, distributed control systems require group communication capabilities. In this section, we design a publisher/subscriber protocol for our CAN-based CORBA. We first define the protocol header format using the CAN identifier structure. We then present the conjoiner-based announcement/subscription mechanism which allows for dynamic binding between publishers and a group of anonymous subscribers.

1. Defining the Protocol Header

In our design, we make use of the CAN identifier structure for the protocol header. The greatest challenge in defining the protocol header using the 11-bit CAN identifier structure is in its limited size. We put the greatest emphasis on making efficient use of the bits in the identifier. Also, we attempt to simplify the protocol design to warrant the small execution overhead and code size of the protocol stack as long as it can provide desired services for higher level CORBA layers. Figure 2 shows our protocol header format. We divide the CAN identifier structure into three sub-fields: a protocol ID, a transmitting node address, and a port number. They respectively occupy two, five and four bits amounting to 11 bits. 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 by `Proto`. In our current design, among four possible values of the `Proto` field only 10_2 and 11_2 are used for the EIOP and the channel binding protocol, respectively. The other two are reserved for potential user-defined protocols.

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 protocols. The `TxPort` field represents a port number which is local to a particular transmitting node. Since `TxNode`

serves as a domain name which 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 set of ports, using the message filtering mechanism of the CAN bus adaptor. In this way, anonymous publisher/subscriber communication is effectively supported.

The layout of the CAN data field is determined by `Proto` which designates the upper layer protocol. A CORBA object invocation message longer than eight bytes should be fragmented into multiple CAN messages. Since the CAN offers reliable and ordered message transmission based on physical error detection and recovery, message re-assembly at a receiving end is done in a straightforward manner.

2. Conjoiner-based Channel Binding Mechanism

Our publisher/subscriber model relies on an intermediary object we name a *conjoiner*. A conjoiner is a pseudo-CORBA object which establishes an invocation channel from publishers to a collection of anonymous subscribers. It must be started right after network initialization, and then operational during the entire system service period. It maintains a global binding database where each CORBA object in the system has a corresponding entry. One of important roles of the conjoiner object is to translate a CORBA object name string into a global port number consisting of `TxNode` and `TxPort`. This is done by looking up the global binding database. Figure 3 illustrates the conjoiner-based publisher/subscriber framework and the global binding database.

As shown in Figure 3, an entry in the global binding database is a quadruple consisting of a channel *tag*, an *OMG IDL interface identifier*, and `TxNode` and `TxPort`. The channel tag is a unique symbolic name associated with each invocation channel. An invocation channel is a virtual broadcast channel from publishers to a group of subscribers. Each publisher is attached to an invocation channel via its own port. A channel tag 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 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.

The conjoiner object oversees object registration, consumer subscription, and dynamic channel binding between publishers and subscribers. When a publisher wants to get attached to an invocation channel, it first obtains a communication port from its local free port pool, and then registers it to the conjoiner object. This procedure is illustrated in Figure 3 by an arrow labeled as (1) `announce()`. Such a registration process leads to the creation of an entry, or the modification of one if it exists, in the global binding database. Thus, 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.

A subscriber wishing to subscribe to an invocation channel also accesses the conjoiner object, as depicted in Figure 3 by an arrow labeled as (2) `subscribe()`. A subscriber's message contains a channel tag and an IDL interface identifier. If a matching entry is found in the global binding database, the conjoiner provides the subscriber with the binding information of the invocation channel. The subscriber ends its subscription process by updating the mask register of the CAN bus adaptor so that it can accept subscribed messages later on.

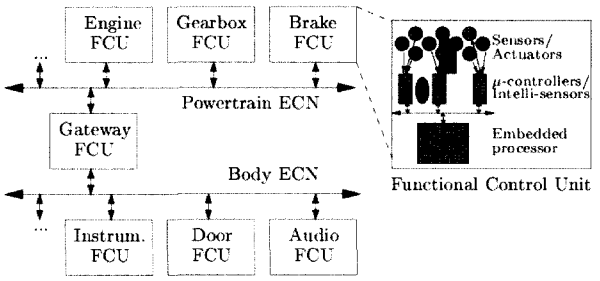


Fig. 3. Conjoiner-based object binding scheme.

```
// IDL

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

Fig 4. IDL program for subscriber interface.

After subscription, subscribers are asynchronously informed of changes in invocation channels. Note that a publisher may be dynamically attached to an invocation channel or detached from it. As shown in Figure 3, a binding agent in a subscriber reacts to such asynchronous updates.

A subscriber maintains its own local binding database, which contains the binding information of all the invocation channels it currently subscribes to. Unlike the group communication scheme in [7], subscribers in our scheme receive messages directly from publishers using the local binding databases. As an example, consider `update_temperature()` method in Figure 3. The publishers of temperature data invoke this method to send out messages via an invocation channel. The subscribers receive the temperature data when their `update_temperature()` methods are executed. Since a subscriber knows the port addresses of all of its publishers using its local binding database, it can conveniently pick up

subscribed messages from the broadcast bus. Note that every subscriber intending to receive the temperature data should possess the `update_temperature()` method. The common interface of such subscribers is defined in an IDL program.

3. Example Publisher/Subscriber Code

We present an example program which demonstrates the usage of our publisher/subscriber scheme. The program consists of publisher and subscriber objects which respectively perform temperature sampling and updating. The source program consists of an IDL interface definition and subscriber/publisher code. Figure 4 shows the IDL code which defines the interface of the subscriber objects. It specifies the signature of method update_temperature() which updates temperature values in the subscriber objects. This method is defined as a oneway operation which does not have output parameters.

Figure 5 shows the publisher/subscriber code in C++. In both source files, there exist 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.

```
// 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->get_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);

// Receive temperature values
while(1) {
   ...
}
```

```
// Publisher code in C++

// Define a channel tag
// for temperature monitoring.
#define TEMP_MONITOR_TAG 0x01

// Initialize the object request broker.
CORBA::ORB_ptr orb = CORBA::ORB_init(argc,argv);
// Get a reference to the conjoiner.
Conjoiner_ptr conjoiner = Conjoiner::_narrow(
    orb->get_initial_reference("Conjoiner"));
// Obtain a reference to the temperature
// monitor group TEMP_MONITOR_IFACE is an
// interface identifier generated
// by the IDL compiler.
HeatMonitor_ptr monitor =
    conjoiner->announce(TEMP_MONITOR_TAG,
                        TEMP_MONITOR_IFACE);

while(1) {
    ...
    // Invoke a method of subscribers.
    monitor->update_temperature(placeA, value,
                                currentTime);
}
```

Fig. 5. Publisher/subscriber code

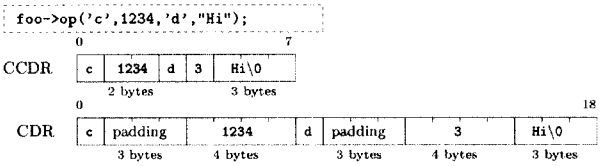


Fig. 6. Example encoding.

Table 1. Variable-length integer representations.

two MSB	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

IV. Embedded Inter-ORB Protocol

Remote method invocation in CORBA is handled through the general inter-ORB protocol which allows for interoperability among various CORBA implementations. The CORBA 2.2 GIOP defines a transfer syntax called common data representation (CDR) and eight messages types which cover all the ORB request/reply semantics. However, the GIOP is not suitable for our embedded CORBA since it triggers a large number of CAN message transfers upon every method invocation. In this section, we present a new inter-ORB protocol by defining a new transfer syntax and two message types. They are called the embedded inter-ORB protocol (EIOP) and the compact common data representation (CCDR).

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. It also addresses inter-platform issues such as byte ordering and memory alignments in such a way that it can support fast encoding and decoding of IDL data types. Specifically, CDR aligns integers on 32-bit boundaries and supports both little and big endian byte orderings rather than mandating a common network byte ordering. As a result, marshalling and unmarshalling of a GIOP message becomes very fast if it is performed on processors supporting the same ordering and alignment. Clearly, the generality and efficiency of CDR are achieved at the expense of increased network load.

In order to make CDR affordable on a slow network such as CAN, we propose compact common data representation (CCDR). We optimize CDR in two ways. First, we add packed

data encoding into CCDR in that integers need not be aligned on 32-bit boundaries. This saves padding bytes. Figure 6 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 which are needed to align two integers `1234` and `3` in CDR. (Integer `3` is internally used to specify the string length.) 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 minimized if the encoded message fits in a single CAN message, which is often the case in an embedded control system.

Second, we introduce a variable-length encoding scheme for integers. While an integer is stored in four 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. Obviously, these integer values are very small in most cases.

Table 2. CORBA 2.2 GIOP message types.

Message Type	Originator	EIOP support
Request	Client	yes
Reply	Server	no
CancelRequest	Client	yes
LocateRequest	Client	no
LocateReply	Server	no
CloseConnection	Server	no
MessageError	Client or Server	no
Fragment	Client or Server	no

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

Fig. 7. EIOP message format.

We thus devise a variable-length integer encoding scheme in that an integer occupies one to five bytes depending on the actual value it represents. As shown in Table 1, we use first two MSBs to denote the actual byte-length of an integer. We decide to support only the big endian byte ordering in CCDR to reduce the encoding/decoding overhead. Revert to the method invocation example in Figure 6. We observe that extra five bytes are saved through the variable-length encoding scheme and that these two schemes together yield total eleven-byte saving in this simple method invocation. As a result, the method invocation can fit in a single CAN message in CCDR while it needs three CAN messages in CDR.

2. EIOP Messages

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 message types. Table 2 shows eight message types supported in the CORBA 2.2 GIOP. We make two customizations on GIOP.

As the first customization, we reduce the number of supported message types into two in EIOP. To do so, we eliminate from GIOP LocateRequest, LocateReply, CloseConnectioin, and Fragment messages which are meaningful only in connection_oriented point_to_point communication. We also eliminate Reply and MessageError messages since our CORBA supports only asynchronous communication. As a result, EIOP supports only Request and CancelbRequest messages, as summarized in Table 2.

The second customizatin we make on GIOP is to reduce the length of the message header of the `Request` type. Note that messages of this type are most frequently seen in the system since they carry method invocation information. Since the header is included in every Request message, it is crucial to

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

Hardware
40MHz Intel 386 EX embedded processor (no cache) KVASER s PCcan CAN bus adaptor 2.0 [14] (Intel 82527 CAN controller [4])

Software
mArx real_time operating system [17, 18] our CORBA-based middleware (based on GNU ORBit 0.4.3) KVASER's PCcan device driver (ported onto mArx)

reduce its size. We first modify the common prefix by educing the 4-byte magic field into one-byte magic code and merging GIOP_verseion, flags, and message type fields into the one-byte flags. MessageHeader_1_0 in Figure 7 defines this new header format.

We then modify the type-specific header of the request message in two ways. First, we remove optional and reserved fields such as service context and requesting principal from the GIOP request header. They are used to store information required only when add-on services such as Security and Transaction are provided. Second, we encode name strings appearing in the GIOP request header into integer identifiers. RequestHeader_1_1 of GIOP 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 holds 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 identifiers for them. Finally, we remove request_id and response_ expected fields since the Reply mes-

sages are not supported in EIOP.

VI. Experiments and Performance Results

We have implemented our CORBA-based middleware using GNU ORBit version 0.4.3 [10]. The target hardware consisted of three PCs equipped with 40MHz i386 EX embedded processors and KVASER's PCcan interface boards with Intel 82527 CAN controllers. The data transfer rate of our CAN bus was 1Mbps. We wrote a CAN interface driver and incorporated it into the mArx real-time operating system [17, 18]. We replaced the GIOP and CDR of ORBit by our EIOP and CCDR libraries, and aggressively down-sized ORBit to make it conformant to the OMG minimum CORBA specification [6]. Table 3 summarizes the hardware and software platform for our implementation.

1. Performance Metrics

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

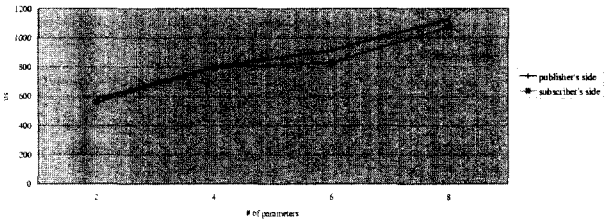


Fig. 8. Protocol processing latency.

2. EIOP Protocol Latencies

We measured the protocol processing latencies both on the publisher side and on the subscriber side. We summarize the measurement results in Figure 8. During our measurements, we used the TemperatureMonitor code shown in Figure 5. 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 Figure 8, the worst_case protocol processing latencies are less than ms when the number of parameters are reasonably small, specically, six. This is typical in most field bus applications of practical interest [19]. 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.

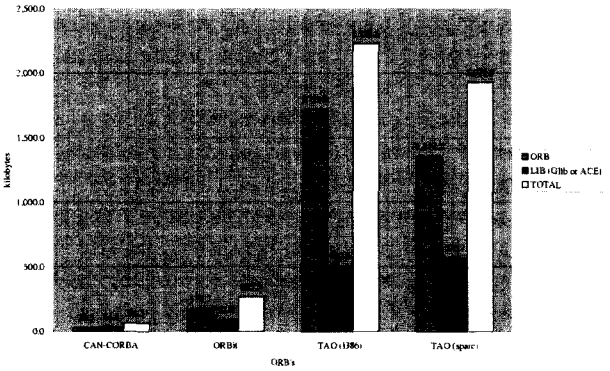


Fig. 9. Static memory requirements of three ORBs.

2. 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 Figure 9. We obtained these numbers by compiling the three ORB implementations with GNU C compiler V2.8.1 targeted to Intel 386 processor. We used –

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

Modules	Footprint (bytes)		Supported by minimum CORVA
	CAN-CORBA	ORBit	
IOP	7,493	19,350	yes
DII/skeleton	0	78,026	no
Dynamic any	0	32,932	no
POA	8260	10618	partially
Others	13770	33,776	yes
Total	29,523	174,657	

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 Figure 9 denotes the memory size of TAO hosted on the Sun Sparc workstation.

VI. 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 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 ($700\mu s$ on the average) on the sender side requiring only 64.3Kbytes 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] Bosch. CAN specification, version 2.0, 1991.

[3] N. Brown and C. Kindel. Distributed component object model protocol – DCOM/1.0, 1998.

[4] I. Corporation. 82527 serial communications controller architecture overview, Jan. 1996.

[5] O. M. Group. The Common Object Request Broker: Architecture and specification revision 2.2., Feb. 1998.

[6] O. M. Group. Minimum CORBA - joint revised submission, OMG document orbos/98-08-04 edition, Aug. 1998

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

[8] Honeywell. Micro switch specification: Application layer-protocol specification version 2.0, 1996.

[9] C. in Automation (CiA) Draft standard 301 version 3.0, CANopen, communication profile for industrial systems based on CAL.

[10] R. H. Inc. ORBit. http://www.labs.redhat.com/orbit, 1999.

[11] ISO-IS 11898. Road vehicles - interchange of digital information - controller area network (CAN) for high speed communication, 1993.

[12] J. Kaiser and M. A. Livani. Invocation of real-time objects in a CAN bus-system. In *IEEE International Symposium on Object-oriented Real-time distributed Computing*, May 1998.

[13] J. Kaiser and 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*, May 1999.

[14] S. KVASER, AB, Kinnahult. PCcan 2.0, Sept. 1998.

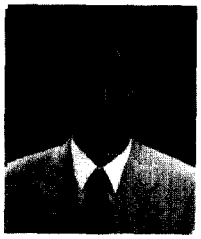
[15] G. Pardo-Castellote and S. Schneider. The network data delivery service: Real-time data connectivity for distributed control applications. In *IEEE International Conference on Robotics and Automation*. May 1994.

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

[17] S. N. U. RTOS Lab. mArx: micro-Arx, http://arx.snu.ac.kr, 1998.

[18] Y. Seo., J. Park, and 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.

[19] G. Ulloa. Fieldbus application layer and real-time distributed systems. In *IEEE International Conference on Industrial Electronics, Control and Instrumentation, IECON*. Oct. 1991.



Seongsoo Hong

Seongsoo Hong is currently an assistant professor of School of Electrical Engineering and Computer Science at Seoul National University. He received the B.S. and M.S. degrees in computer engineering from Seoul National University, Korea in 1986 and 1988, respectively. Dr. Hong received the Ph.D. in computer science from the University of Maryland, College Park in 1994. From April to August of 1995 he worked at Silicon Graphics Inc. as a Member of Technical Staff. From December 1994 to April 1995, he was with the University of Maryland as a Faculty Research Associate.

His current research interests include embedded real-time systems, operating systems, multimedia systems, distributed computer control systems, embedded real-time middleware, and software tools and environment for embedded real-time systems. He is now undertaking research projects to develop the Velso real-time operating system and systematic methodologies for distributed embedded real-time systems.

Dr. Hong has served as a program co-chair of 2001 ACM LCTES and a program committee co-chair of 2002 IEEE ISORC. He has served on numerous program committees.