Integration of a CAN-based Connection-oriented Communication Model into Real-Time CORBA

Stefan Lankes, Andreas Jabs, Thomas Bemmerl
Lehrstuhl für Betriebssysteme,
RWTH Aachen, Kopernikusstr. 16,
52056 Aachen, Germany
E-mail: {stefan, jabs, thomas}@lfbs.rwth-aachen.de

Abstract

The Real-Time CORBA and minimumCORBA specifications are important steps towards defining standard-based middleware which can satisfy real-time requirements in an embedded system. These requirements can only be fulfilled if the middleware utilizes the features of a real-time network. The Controller Area Network (CAN) is one of the most important networks in the field of real-time embedded systems. Consequently, this paper presents a CAN-based connection-oriented point-to-point communication model and its integration into Real-Time CORBA. In order to make efficient use of the advantages of CAN, we present an inter-ORB protocol, which uses smaller message headers for CAN and maps the CAN priorities to a band of CORBA priorities. We also present design and implementation details with some preliminary performance results.

Keywords: *Real-Time CORBA, Controller Area Network (CAN), distributed real-time embedded (DRE) applications, real-time communication systems*

1. Introduction

Previously generation real-time applications were running on single processor environments, since the problems to be solved were relatively simple. Nowadays, distributed applications like avionics, telecommunication and process control need real-time properties in a distributed environment. In these systems, intelligent sensors, actuators and distributed control structures replace the centralized computer. This leads to a modular system architecture in which smart autonomous objects cooperate to control a physical process.

Middlewares like *CORBA* [14] and *DCOM*[1] help to improve the flexibility, extensibility, maintainability and reusability of distributed applications. Because these middleware architectures were not designed for real-time applications, the *Object Management Group*[2] (OMG) has specified a real-time extension for CORBA [16] and called it *Real-Time CORBA*. However, Real-Time CORBA is not immediately applicable to embedded real-time control systems for several reasons.

- Real-Time CORBA implementations have excessive resource demands. A first step to solve this problem is the *minimumCORBA* [15] specification, which is a cut-down version of CORBA specified by the OMG. In [8] Schmidt presents an alternative solution using a highly configurable real-time object request broker.
- Often Real-Time CORBA implementations are built on the top of unpredictable off-the-shelf soft- and hardware and do not support typical real-time networks for embedded systems like the *Controller Area Network* (CAN) [19].

To solve these problems we have developed a new Real-Time CORBA implementation, which is especially designed for embedded systems and supports the *Controller Area Network*. Therefore we called our implementation *Real-Time CORBA for embedded Systems*[3] (ROFES) [10][11].

An application example for ROFES is a passenger vehicle in an intelligent transportation system. Such a vehicle is equipped with numerous microprocessors and uses the CAN bus as network architecture. It is connected to the internet via a wireless network which makes it possible for driver to receive road traffic conditions. ROFES can be used to develop distributed real-time applications inside the vehicle, which use the CAN bus, and these applications can also

[1] `http://www.microsoft.com/com/tech/dcom.asp`

[2] `http://www.omg.org`

[3] `http://www.lfbs.rwth-aachen.de/users/stefan/rofes`

communicate via the wireless network with other CORBA applications outside the vehicle. Real-Time CORBA is the easiest way to integrate traditional distributed real-time applications into *normal* distributed applications. Therefore these new possibilities open new areas of applications to automotive industry.

CAN is an industrial real-time network which is widely used in the automotive industry. Since its maximum network bandwidth is only 1 Mbps and the maximum payload per message is only 8 bytes, it is very challenging to run Real-Time CORBA applications on a CAN-based distributed platform. To exploit the advantages of CAN for ROFES, we have designed a new inter-ORB protocol with smaller message headers. This inter-ORB protocol is an extension of Kim's *embedded inter-ORB protocol* (EIOP) [7] [5] [6] for a CAN-based CORBA. EIOP was however not developed for Real-Time CORBA and therefore it provides no translation between the priority handling of CAN and Real-Time CORBA. This paper presents a mapping between CAN priorities and a band of Real-Time CORBA priorities and evaluates its performance.

This article is organized as follows: Section 2 summarizes the technical backgrounds of this work and describes basic principles of Real-Time CORBA and the Controller Area Network. Before the special inter-ORB protocol for the CAN bus is described in section 4, the article presents the realization of a point-to-point protocol for the CAN bus and the mapping between CAN priorities and bands of Real-Time CORBA priorities in section 3. Section 5 evaluates and discusses the performance of our implementation. Finally, some general assessments of the lessons learned are provided and some conclusions are drawn in section 6.

2. Technical Backgrounds

2.1. Basic CAN Features

The Controller Area Network (CAN) [19] is an ISO defined serial communication bus. It was originally developed during the 80's by the Robert Bosch GmbH[4] for the automotive industry. The CAN bus works according to the Producer-Consumer-Principle: messages are not sent to a specific destination address, but rather as a broadcast (aimed at all receivers) or a multicast (aimed at a group of receivers). A CAN message has a unique identifier, which is used by devices connected to the CAN bus to decide whether to process or ignore the incoming message.

Two variants of the CAN protocol exist. The main difference between the first (CAN 2.0A) and second variant (CAN 2.0B) is that the former uses 11 bits to uniquely identify each message, while the latter uses 29 bit identifiers. For correct operation of the CAN bus, the identifiers of two messages sent at the same time must never be the same, consequently CAN 2.0B offers a greater variety and scope for concurrent message Id's.

The CAN bus is based on the arbitration scheme *Carrier Sense Multiple Access/Collision Avoidance* (CSMA/CA) [9]. During arbitration process, any node willing to send a CAN message starts sending bit by bit the 11 or (in case of CAN 2.0B) 29 identifier bits. Each time a bit is applied to the bus, the sending node checks whether the bus really is at the corresponding voltage level — high for an applied logical 1 and low for an applied logical 0.

If any of the attached nodes apply a logical 0, the whole voltage level of the bus is drawn to *low*. The bus thus offers the behavior of a "Wired-And". The CAN Specification therefore calls the logical 0 – corresponding to the low voltage level – the *dominant* bit and the logical 1 the *recessive* bit.

In case the sending node detects a difference between the bit it sent and the voltage level on the bus, it backs off, losing this arbitration cycle. As soon as the bus is free again, the node retries to send its message. If it was possible to have two messages with the same identifier at the same time, this arbitration process would fail and the system state would be undefined. As a consequence of this arbitration scheme, messages with a low identifier (i.e. starting with many leading zeros) have a high priority and are sent before any messages with a lower priority.

The addressing scheme of the CAN-Bus is particularly suited for a publisher/subscriber communication model. Since the publisher/subscriber model is extremely attractive for the structuring of object-oriented control applications, the OMG specified a CORBA event service [13] according to this model. For the real-time domain, several implementations [2][18] of the event service exist, which are often based on TCP/IP. Kaiser in [4] and Kim in [7] present possibilities to integrate the CAN-Bus into the event service. To use the CAN-Bus in Real-Time CORBA, a CAN-based connection-oriented communication model has also to be developed. Kim presents in [7] the *embedded inter-ORB protocol* (EIOP) for a CAN-based CORBA. But EIOP was not especially developed for Real-Time CORBA. Therefore, EIOP currently offers no translation between the priority handling of CAN and Real-Time CORBA. We extend this protocol to map CAN priorities to a band of Real-Time CORBA priorities.

2.2. Basic Real-Time CORBA Features

To understand the integration of our CAN protocol into Real-Time CORBA, this section explains the necessary features of the Real-Time CORBA specification. A more detailed description of the Real-Time CORBA specification is

[4] `http://www.can.bosch.com`

given in [16] and [20].

The original CORBA specification supports only implicit bindings, establishing resources on demand. For real-time applications with deterministic Quality of Service (QoS) requirements these implicit bindings are inadequate. Real-Time CORBA defines explicit binding mechanisms which improve the performance and predictability of invocations. One of the explicit binding mechanisms is the *priority banded connection* model. This facility allows a client to communicate with the server via multiple transport connections — each dedicated for carrying invocations with a different CORBA priority or a range of priorities, as shown in Figure 1. In such a model the server specifies the range of priorities that a connection will be used for. This allows the server to allocate all necessary resources. The selection of the appropriate connection for each invocation is transparent to the application and is done by the ORB based on the priority model. For instance, if the target object supports the *client propagated model* as priority model, the client selects the connection by its current priority and transmits this priority with the invocation. The server processes the incoming request at the priority of the client thread, which originally invoked the operation. Another model is the *server declared* model, which allows a server to dictate the priority at which an invocation made on a particular object will be executed. In such a model, the priority is specified *a priori* by the server. The server encodes the priority of the object in its reference, which is then published to the clients.

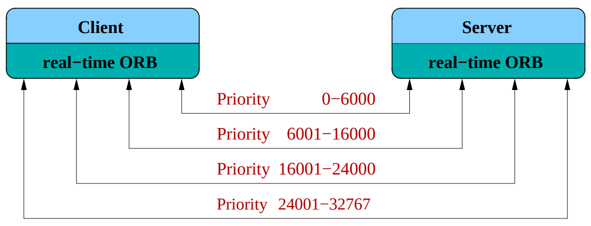


Figure 1. Example of a Priority Banded Connection

Client and server can run on different operating systems and use different native thread priority schemes. For the client-propagated and the server-declared priority model, Real-Time CORBA needs a priority model independent of the operating system. The OMG defines a logical priority which has a system-wide uniform representation called *CORBA Priority*. A real-time ORB provides a priority mapping between CORBA priorities and native priorities for each supported platform.

3. Connection-Oriented Protocol for ROFES

There are different application-layer protocols available for the CAN bus[1][12]. In these protocols it is difficult to implement a mapping between the priority of a CAN message and the importance of a Real-Time CORBA method invocation. For this reason we decided to design a new protocol.

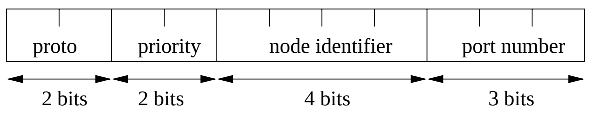
3.1. Selection of the CAN Identifier

Because the CAN identifier specifies the priority of the messages, the identifier is the main component of the controller area network and must be selected carefully. Our protocol is based on CAN 2.0A and thus only uses 11-bit CAN identifiers, because the arbitration process of CAN 2.0A is simpler and has better real-time characteristics than the arbitration process of CAN 2.0B. Thus, our Real-Time CORBA implementation must make efficient use of the bits of the identifier and take care of minimum execution overhead.

Figure 2 presents the structure of the CAN identifier. This CAN identifier is divided in four sub fields:

- The *proto* field denotes an upper layer protocol identifier and possesses four possible values:
  - 00_2 is representing the top priority protocol and is reserved for a potential user-defined protocol.
  - 01_2 is reserved for a publisher/subscriber protocol, which we are currently developing for ROFES.
  - 10_2 is representing the connection oriented-protocol, which is discussed in this paper.
  - 11_2 is used for the network management protocol. A connection between client and server or publisher and subscriber will be established over this protocol. As a result they both get a valid CAN identifier with the *proto* field 01_2 or 10_2 which they can use to exchange messages.
- The next field *priority* represents the priority of the message.
- The field *node identifier* specifies the transmitting node. Therefore, the identifier serves as a domain name which is globally identifiable all across the network.
- The last field represents a port number that is local to a particular transmitting node.

The small priority field reduces the granularity of priority supported by CAN and the small node identifier supports



$$\text{proto} = \begin{cases} 00 & \text{reserved} \\ 01 & \text{publisher/subscriber communication} \\ 10 & \text{point-to-point communication} \\ 11 & \text{binding} \end{cases}$$

Figure 2. Structure of the CAN Identifier

only 16 distinguishable nodes. If more nodes or priorities need to be differentiated, then the protocol can be extended to CAN 2.0B. This supports a larger CAN identifier, but the complexity of the arbitration process in CAN 2.0B is much higher than in CAN 2.0A and leads to a decrease in the network performance.

The node identifier, port number and priority form a global connection identifier. This allows to use the same port in distinct nodes with four different priorities. The header does not include any form of destination addresses. The receiving CAN nodes can select and accept messages sent from a specific subset of ports, using the message filtering mechanism of the CAN bus adapter.

3.2. Structure of the Bidirectional Connections

To realize bidirectional connections within Real-Time CORBA, our solution establishes pairs of unidirectional pipes. The connections use two local ports and each port belongs to the source node of each pipe. Four priority ranks exist for each port, which are specified by the priority field of the identifier. A pair of unidirectional pipes for one priority rank is shown in figure 3.

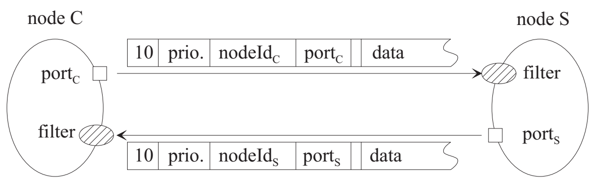


Figure 3. Point-to-point Communication via two Unidirectional Pipes

To accept a message, the destination node must know the node identifier and port number of the source. Therefore, the connection requires two communication endpoints and the nodes must exchange information about the port address of each endpoint. To exchange between the endpoints, a special listening port exists on the server-side. The server publishes its node identifier and port number of the listening port via an IOR profile [3]. In this profile, the node identifier replaces the host name. Therefore, the creation of an IOR profile for our CAN protocol is similar to IIOP[5] and the changes to the code of ROFES are minimal. With use of this profile, the client can evaluate the listening port number and the node identifier. To establish a connection, the client sends its local port number and node identifier to the listening port of the server. This information is sent by the client via the valid CAN identifier, which is made up of the listening port number, the node identifier of the server and the protocol type 11_2 of the network management protocol. The server answers with the port number on which it awaits incoming requests. During connection establishment, the priority field of the CAN-identifier is ignored and can take any value.

3.3. Integration into Real-Time CORBA

In order to achieve optimal real-time characteristics, the priority of a CORBA message, which is either inherited from the calling thread (client propagated model) or dictated by the server (server declared model), must be mapped to a priority of the CAN message, which is described by the priority field in the CAN identifier and shown in figure 2. A direct mapping between the priority of the CORBA message and the CAN message is not advisable, because the priority field is only two bits long and CORBA knows 32768 different priority ranks. If a distributed real-time application uses Real-Time CORBA's priority banded connections, our protocol determines the importance of a connection by its band of priorities. Therefore, our protocol can map four different bands of priorities into the priority of the CAN message, which is specified by the priority field in the CAN identifier.

The CAN bus allows a maximum of 8 bytes data in a single CAN message. In order to be able to send large CORBA messages, our protocol has to split them into a series of CAN messages with the same CAN identifier on the side of the sender, which is shown in figure 4, and rejoin these messages on the receiving side. This splitting is necessary because the basic header of a GIOP message[6] is 12 bytes long.

4. Size Reduction of the Inter-ORB Protocol

The GIOP protocol is designed and optimized for heterogenous systems and creates large messages. This is not suitable for the CAN bus, because a CAN message contains only 8 bytes of data and the bandwidth is very small

[5]Internet Inter-ORB Protocol

[6]The OMG defines General Inter-ORB Protocol (GIOP) as basic communication protocol between a client and a server.

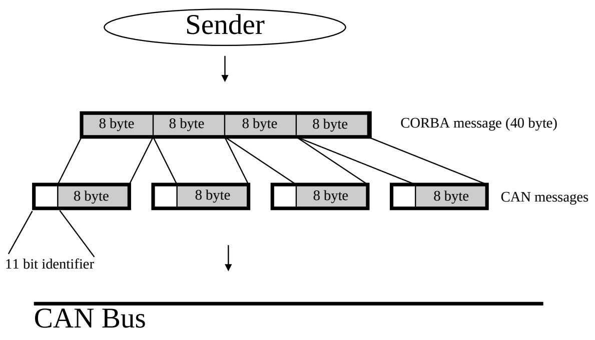


Figure 4. Fragmentation of a Real-Time CORBA Message

(1 Mbps). Therefore, we designed a smaller inter-ORB protocol for the CAN bus, which we called *CANIOP*, and used Kim's *Compact Common Data Representation* (CCDR) [7] to encode integers.

4.1. Smaller Data Representation of Integers

The *Compact Common Data Representation* (CCDR) is a integer codec with variable length. Unlike the CDR[7], which the OMG defined as data representation, the packed data encoding scheme of CCDR does not require integer instances to be aligned on 32-bit boundaries. Accordingly, CCDR creates no paddings between the various data types. In the variable length integer encoding of CCDR, an integer occupies one to five bytes depending on the actual value it represents, as summarized in table 1. While an integer is always stored in four bytes in CDR, most integer instances are smaller than $2^{32} - 1$. For instance, in CDR, integers are very frequently used to represent the sizes of `string` and `sequence` data types and are small values in the majority of cases. This encoding scheme increases the processing overhead of message encoding and decoding, but the advantages of smaller messages outweigh this factor for the CAN bus.

two MSBs	size (in bytes)	max. value (unsigned)
00	1	$2^6 - 1$
01	2	$2^{14} - 1$
10	3	$2^{22} - 1$
11	5	$2^{32} - 1$

Table 1. Variable Length of the Integer Codec

[7]*Common Data Representation*

```
struct GIOPMessageHeader_1_2 {
   char          magic[4];
   octet         major;
   octet         minor;
   octet         flags;
   octet         message_type;
   unsigned long message_size;
};

struct CANIOPMessageHeader {
   octet         type_and_flags;
   octet         message_size[2];
};
```

Figure 5. Structure of the GIOP- and CANIOP Message Header

4.2. Design of the CAN-based Inter-ORB Protocol

In this section we present a smaller protocol called *CANIOP* between server and client, which based on Kim's EIOP[7]. Kim's protocol is a smaller protocol than CANIOP. For instance, Kim's protocol does not support the service context list of GIOP. But to implement the client propagated protocol [16] of Real-Time CORBA this list is needed. Therefore, we analyzed all necessary components and integrated this into our protocol.

To explain our protocol the paper compares this with the standard protocol GIOP 1.2. A complete description of all message types is too long for this paper. Therefore, it concentrates on *request-* and *reply-messages* as they are the most important message types. Every GIOP message begins with the same 12 byte long header, which is shown in figure 5. To clearly identify a message as GIOP message the header begins with an array of octets, which contains the value `GIOP`. This array is not necessary for CANIOP, because the protocol is clearly identified by the first two bits of the CAN message identifier (see figure 2). Additionally, the GIOP header contains the version number of the protocol, a flag to identify the sender's processor architecture, the message type and the message length. In CANIOP the version number of the protocol, the flag to identify the sender's processor architecture and the message type are encoded in one byte, which is called `type_and_flags` in figure 5. This is possible because CORBA uses only eight different message types and thus the type can be encoded in 3 bits of the octet `type_and_flags`. Additionally, the message length is only encoded in two bytes because the CAN bus is designed for small messages. This steps reduce the message size from 12 bytes in GIOP to 3 bytes in CANIOP. Consequently, the first CAN message contains the complete message header and indicates the length of the incoming message to the receiver.

If the client invokes a server method, the client sends a message header followed by the request message, which is shown in figure 6, and the parameters of the method to the server. The main difference between the GIOP and CANIOP request message is the representation of the method name. In GIOP the complete name is sent to the server as a string. In CORBA, a string is encoded as an array of characters and its length as 32-bit integer. In CANIOP, the method name is represented by a hash number. This hash number is calculated by a collision-free hash function, which is created by the idl compiler for each interface.

```
struct GIOPRequestHeader_1_2 {
   unsigned long request_id;
   octet         response_flag;
   octet         reserved[3];
   TargetAddress target;
   string        operation;
   ServiceContextList svc_ctxt;
};

struct CANIOPRequestHeader {
   unsigned long request_id;
   octet         response_flag;
   TargetAddress target;
   unsigned long operation;
   ServiceContextList svc_ctxt;
};
```

Figure 6. Structure of the Request Message in GIOP and CANIOP

In conjunction with a request, the client sends additional information to the server in the service context list. For instance, if the server supports the client propagated model, the clients send their current priority in the service context list to the server. With the reply message, which is shown in figure 7, the server answers the request of the clients and sends the priority which the server internally used to serve the request back in the service context list. With this information, the client compares this priority with its current priority and verifies if the server has used the correct priority internally. Over the CAN bus, ROFES only sends the priority as additional information and the clients renounce the priority checking. Therefore the reply message of ROFES, which is shown in figure 7, lacks the service context list.

All integers in this messages are encoded in CCDR format. Therefore the size of the request and reply message depends on their contents. In the best case, a reply message can be encoded in 5 bytes and can consequently be sent with its header in a single CAN message.

```
struct GIOPReplyHeader_1_2 {
   unsigned long request_id;
   unsigned long reply_status;
   ServiceContextList svc_ctxt;
};

struct CANIOPReplyHeader {
   unsigned long request_id;
   unsigned long reply_status;
};
```

Figure 7. Structure of the Reply Message in GIOP and CANIOP

5. Performance Evaluation

The benchmark in this section is based on a benchmark which is described in [17]. In this benchmark the client has three rate-based threads running at different priority levels. The high-priority thread runs at 50 Hertz, the medium priority thread at 25 Hertz and the low priority thread at 12.5 Hertz. All these threads invoke the following method on the server:

```
void method(in unsigned long work)
```

The work parameter specifies the amount of CPU intensive work the server will perform to service this invocation. In our case, the method checks if the number 4591 is a prime number and the parameter `work` specifies how often the method checks this.

The client has also one best-effort thread making continuous invocations. The best-effort thread has a lower priority than the rated-based threads and invokes the same method with `work = 0` as parameter.

The test platform consists of two 400 MHz Pentium II systems with 128 MBytes RAM. *LynxOS 3.0.1*[8] is used as operating system for these computers. The systems are connected with the PCI-CAN interface from *ESD GmbH*[9], which based on Philips CAN controller SJA1000. The test platform is clearly not an embedded system. However, having very low memory footprints, the results should be comparable to the results on an embedded system.

All experiments in this section use the client propagated model as priority model. The first experiment uses a classic CORBA configuration with no priority banded connections and only one threadpool with three threads to process

[8]`http://www.lynx.com`
[9]`http://www.esd-electronics.de`

the incoming requests. Therefore this benchmark can not benefit from the additional priority information in the CAN messages. Figure 8 shows the throughput for each of three client threads as the workload increases. The combined capacity desired by the three client threads is 87.5 invocations per second (50 + 25 + 12.5). The maximum workload to achieve all desired frequencies can be calculated as follows:

$$(workload * t_{prime} + t_{invocation}) * 87.5\frac{1}{sec} \leq 1$$

In this equation t_{prime} represents the average time to evaluate that 4591 is a prime number, and $t_{invocation}$ the average time to invoke the method, which directly returns to the calling process. On the test system, their values were found to be $t_{prime} = 232\mu sec$ and $t_{invocation} = 945\mu sec$. Therefore, the maximum workload to achieve all desired frequencies is 45. Figure 8 shows that each of the three client threads achieves its desired frequency for lower workloads. After the workload is increased beyond 45, deadlines start being missed. The expected behavior of distributed real-time applications is to drop requests from client threads with lower priority before dropping requests from those with higher priority. Unfortunately, this solution can not realize this expected behavior.

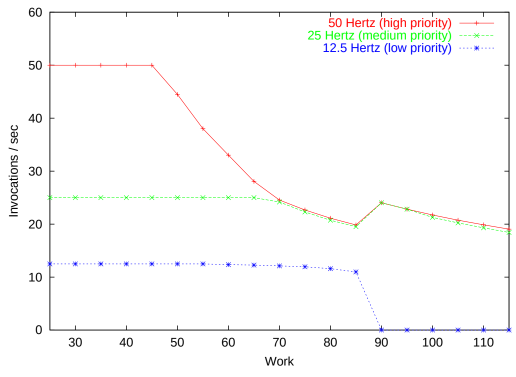


Figure 8. Throughput using CAN and no Priority Banded Connections

The second experiment creates an own priority banded connection for each thread. For each connection, the server possesses a threadpool with lanes for high, medium and low priorities to handle incoming requests and an additional threadpool for the continuous thread. Therefore this benchmark can benefit from the additional priority information in the CAN messages.

Figure 9 shows the throughput achieved for each of the three threads as the workload increases. Each of the three client threads achieves its desired frequency for workloads lower or equal 45. After the workload increases beyond 45, deadlines start being missed. Unlike the first experiment, the requests of the low priority thread are dropped and the medium and high priority thread can achieve their desired frequency. If the low priority thread dropped all requests, the medium priority thread starts to drop its requests and the high priority thread can achieve its desired frequency. Until the execution time of the method is larger than the period of the high priority thread, the high priority thread achieves its desired frequency. The theoretical maximum workload at which the high-priority thread can achieve its desired frequency is in our implementation 82.

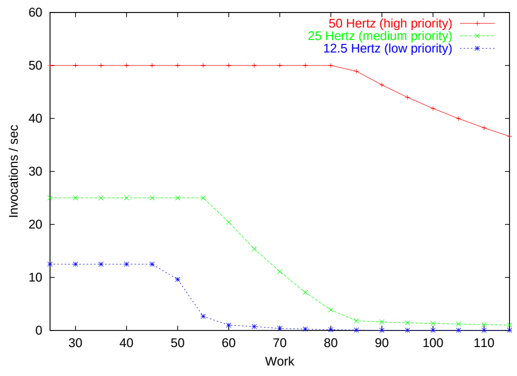


Figure 9. Throughput using Priority Banded Connections over CAN

The last experiment, whose results are shown in figure 10, uses the same configuration as the second experiment, but instead of CAN this experiment uses fast ethernet as network interface. In contrast to the CAN experiment requests from the lower **and** medium threads are dropped at the same time to achieve the desired frequency of the high priority thread. Only the CAN-based solution can fulfill the expected behavior and drops at first the request from the low priority thread and then the requests from the medium thread to achieve the desired frequency of the high priority thread. Therefore, the CAN-based solution is the better one for distributed real-time applications.

The main advantage of ethernet is its high bandwidth. Therefore the average time to invocate a method, which directly returns to the calling process, is lower at $t_{invocation} = 416\mu sec$ and the maximum workload to achieve the desired frequency of the high priority thread is with 85 higher than CAN-based solution.

6. Conclusions and Future Work

With the work described herein, the CAN bus has been rendered more usable in the field of distributed real-time systems. To realize this the importance of a priority banded connection has been mapped to the priority of CAN message specified by its identifier. The overhead in the GIOP

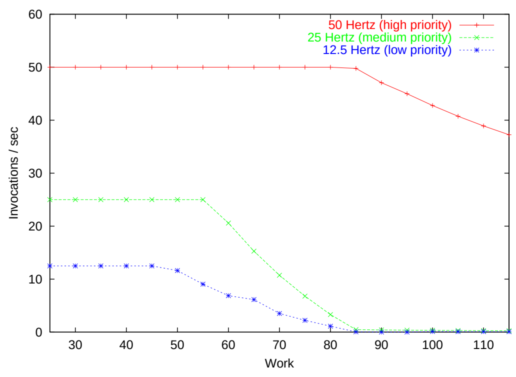


Figure 10. Throughput by using Priority Banded Connections over Ethernet

protocol has been reduced by utilizing the features of the CAN-Network, especially the compact message header, the using of hash numbers as operation names and the omission of mostly unneeded features, considering the fact that this implementation is targeted at embedded systems. A significant reduction of occupied bandwidth and fragmentation has been achieved by porting the CCDR encoding scheme to ROFES. Our benchmarks prove that Real-Time CORBA over CAN represents a good solution to implement distributed real-time applications. Also, the availability and preemptiveness of the network has been enhanced as a result of the shorter message length.

Further research could be targeted at enlarging the number of possible nodes on a network, e.g. through the usage of CAN2.0B message headers. The achieved decrease in message length should help alleviate eventual problems of increased network latency for greater numbers of nodes on a multicast/broadcast network. The arbitration scheme of the Controller Area Network is an ideal fundament for an event service. For that reason we are currently developing such a CAN-based event service for Real-Time CORBA.

References

[1] CAN in Automation (CiA) Draft standard 301 version 3.0. *CANopen – Communication Profile for Industrial Systems based on CAL*, 1996.

[2] T. H. Harrison, D. L. Levine, and D. C. Schmidt. The Design and Performance of a Real-Time CORBA Event Service. In *Proceedings of OOPSLA '97*. ACM, 1997.

[3] M. Henning and S. Vinoski. *Advanced CORBA Programming with C++*. Addison-Wesley, 1999.

[4] J. Kaiser and M. Mock. Implementing the Real-Time Publisher/Subscriber Model on the Controller Area Network (CAN). In *2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 1999)*, Saint-Malo. France, May 1999.

[5] T. Kim, G. Jeon, and S. Hong. Seamless Integration of Real-Time Communications into CAN-CORBA with Extended IDL and Fast-Track Messages. In *IFAC Workshop on Distributed Computer Control Systems (DCCS)*, Sydney, Australia, December 2000.

[6] T. Kim, K. Kim, G. Jeon, and S. Hong. Resource-Conscious Customization of CORBA for CAN-Based Distributed Embedded Systems. In *IEEE International Symposium on Object-Oriented Real-Time Computing*, Newport Beach, CA, USA, March 2000.

[7] T. Kim, K. Kim, G. Jeon, S. Hong, and S. Kim. Integrating Subscription-based and Connection-oriented Communications into the Embedded CORBA for the CAN Bus. In *IEEE Real-time Technology and Application Symposium*, Washington D.C., USA, May 2000.

[8] R. Klefstad, D. C. Schmidt, and C. O'Ryan. Towards Highly Configurable Real-Time Object Request Broker. In *5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2002)*, Washington DC, USA, April 2002.

[9] H. Kopetz. *Real-Time Systems – Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.

[10] S. Lankes, M. Pfeiffer, and T. Bemmerl. Design and Implementation of a SCI-based Real-Time CORBA. In *4th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2001)*, Magdeburg, Germany, May 2001.

[11] S. Lankes, M. Reke, and A. Jabs. A Time-Triggered Ethernet Protocol for Real-Time CORBA. In *5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2002)*, Washington DC, USA, April 2002.

[12] D. Noonen, S. Siegel, and P. Malony. DeviceNet Application Protocol. In *1st International CAN Conference*, Erlangen, Germany, 1994.

[13] OMG Technical Document formal/01-03-01. *Event Service Specification*, 1.1 edition, 2001.

[14] OMG Technical Document formal/98-07-01. *The Common Object Request Broker – Architecture and Specification*, 2.2 edition, 1998.

[15] OMG Technical Document orbos/98-08-04. *minimumCORBA – Joint Submission*, 1998.

[16] OMG Technical Document orbos/98-10-05. *Realtime CORBA – Joint Submission*, 1998.

[17] I. Pyarali, D. C. Schmidt, and R. K. Cytron. Techniques for Enhancing Real-time CORBA Quality of Service. *Submitted to the IEEE Proceedings*, 2002.

[18] 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 *Proceedings of the IEEE Real-time Technology and Applications Symposium*, June 1995.

[19] ROBERT BOSCH GmbH. *CAN Specification Version 2.0*, 1991.

[20] D. C. Schmidt and F. Kuhns. An Overview of the Real-Time CORBA Specification. *IEEE Computer*, 33(6):56–63, 2000.