

Building of a Fault-Tolerant CORBA Infrastructure within the Context of Embedded ORB and the CAN Bus

Tarek GUESMI
Laboratoire SYSCOM
Ecole Nationale d'ingénieurs de
Tunis
1002 Tunis, Tunisia
Tarek.guesmi@isecs.rnu.tn

Salem HASNAOUI
Laboratoire SYSCOM
Ecole Nationale d'ingénieurs de
Tunis
1002 Tunis, Tunisia
Salem.hasnaoui@enit.rnu.tn

Houria REZIG
Laboratoire SYSCOM
Ecole Nationale d'ingénieurs de
Tunis
1002 Tunis, Tunisia
Houria.rezig@enit.rnu.tn

Abstract

An increasing number of applications are being developed using distributed object computing middleware, such as CORBA. Many of these applications require the underlying middleware, operating systems, and networks to provide end-to-end quality of service (QoS) support to enhance their efficiency, predictability, scalability, and fault tolerance. The Object Management Group (OMG), which standardizes CORBA, has addressed many of these application requirements recently in the Real-time CORBA and Fault-tolerant CORBA specifications.

In this paper, we have extended an existing open source ORB, to support fault-tolerant communications over Controller Area Networks, and combine it with a collection of service objects following the FT-CORBA standard. Consequently we present a CAN-based Inter-ORB multicast protocol which supports timely delivery of messages and guarantees atomic order, under anticipated fault conditions.

Keywords: Fault-Tolerant CORBA, Controller Area Network (CAN), multicast protocol, CIOP, total order.

1. Introduction

Future data/telecom applications are built on top of object-oriented distributed system platforms that need high levels of dependability. Therefore these industries stress the importance of enforcement of system-level properties such as fault tolerance, timeliness, and security in such platforms. Nevertheless, the ambition is to keep the application writer efforts minimal when adding these features to a certain application. While there could be more elegant approaches for supporting fault tolerance in runtime systems of a specific language like Java, Ada or

Erlang, real systems are typically multi-language, multi-platform. Therefore, building fault-tolerance in a generic middleware is an interesting undertaking. CORBA is one of these types of middleware, and the one we used to support our Next Generation Switching Equipments we developed according to the MSF (Multi-Service Switching Forum) architecture.

CAN is an industrial real-time network which is widely used in the automotive industry. Since its maximum network bandwidth is 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 RT-CORBA, we developed CIOP protocol [1] which is an extension of Kim's embedded Inter-ORB protocol (EIOP) [5][6] for a CAN-based Real-Time CORBA. The main idea discussed within the context of this paper is how to enhance the mentioned architecture with fault tolerance mechanisms.

Following the FT-CORBA standard specification [2], we have built our infrastructure by combining:

- A service approach seen by the application writer.
- CORBA portable interceptors [3] for requests.
- Extension of the CIOP protocol to support timely delivery of messages and guarantees total order.

The remainder of this paper is organized as follows: section 2 describes the related work; section 3 briefly introduces the technical background and terminology. Section 4 deeply describes the details of our generic infrastructure. Finally, some general assessments of the lessons learned are provided and some conclusions are drawn in section 5.

2. Related Work

There have been attempts to build infrastructures to provide application writers with the possibility to construct fault-tolerant application in an easy way. The Java RMI technology is, for example, used to provide fault-tolerant distributed services in the Jgroup toolkit [4]. Clients interact with these services by an external group method invocation. Group members cooperate via internal group method invocations. This cooperation is achieved by using a group membership service, as well as a reliable communication service. The Horus toolkit [8] provides a platform independent framework for building fault-tolerant applications. It offers application writers a set of building blocks to choose from in order to fit system requirements.

Prior to the specification of the FT-CORBA extension (April 2000), few works have studied alternative augmentations of CORBA with a process (object) group module. Little et.al. present a way of integrating a group communication service with transactions [9]. They start with a system that supports transactions (CORBA), but no process groups. Then, they consider enhancing the use of transactions by introducing process groups. Felber et.al show some possible approaches for introducing object groups in CORBA [11].

Three different approaches are presented depending on the position of the group communication module relative to the ORB. These are: the interception approach, the integration approach, and the service approach. Narasimhan et al. implemented operating system level interceptors to provide fault tolerance to CORBA [10]. The result of their research efforts in this direction is the Eternal System. With this approach, an ORB's functionality can be enhanced for fault tolerance without changes in the ORB, or in the application. Chung et. al. present a fault-tolerant infrastructure (DOORS) built using the service approach, on top of CORBA [13]. In this setup, application objects register to DOORS in order to be made fault-tolerant. Fault tolerance services are realized with two components: ReplicaManager and WatchDog.

3. Technical Backgrounds

3.1. Overview of Fault Tolerant CORBA

To obtain fault-tolerant applications the FT-CORBA approach uses replication in space (replicated objects). Temporal replication is supported by request retry, or

transparent redirection of a request to another server. Replicated application objects are monitored in order to detect failures. The recovery in case of failures is done depending on the replication strategy used. Support is provided for use of active replication, primary/backup replication, or some variations of these. The non-active replication styles provided are warm passive and cold passive (primary/backup). The choice of policy is left to the application writer when initiating the server.

Figure 1 shows the architecture for fault tolerance support according to the standard.

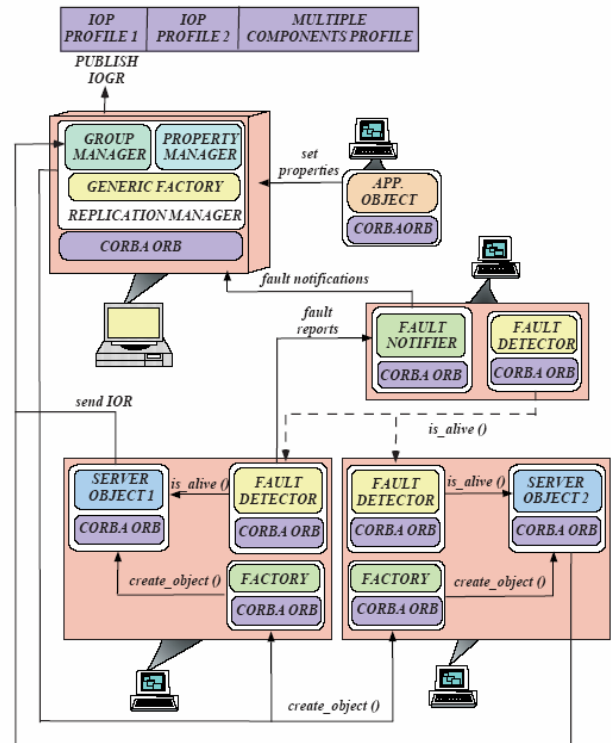


Figure1. The Architecture of Fault Tolerant CORBA

There is a Replication Manager that implements interfaces such as Property Manager, Object Group Manager, Generic Factory. The standard also informally refers to the notion of Fault-Tolerance Infrastructure. This is implicitly the collection of mechanisms added to the basic CORBA to achieve fault-tolerance. The Property Manager interface has methods used to set the abovementioned policy i.e. the replication style, the number of replicas, the consistency style (infrastructure-controlled or application-controlled), the group membership style (infrastructure-controlled or application-controlled). The Object Group Manager interface has methods that can be invoked to support the application-controlled membership style, at the price of losing transparency. The Generic Factory interface has the create_object and delete_object methods. The Replication Manager's create_object method is invoked when a new

object group has to be created. As a consequence, the Object Factories' create_object methods are called. Each object in a group has its own reference, but the published one is the interoperable object group reference.

Application replicas are monitored by Fault Monitors by means of calling the *is_alive* method on them. Thus, the FT-CORBA specification mostly focuses on the pull monitoring style. Push monitoring style is also mentioned but not specified in the standard. The fault monitors, as mentioned by the standard, are unreliable (they cannot decide whether an object crashed or it is just slow).

Monitoring is initiated by the Replication Manager. Fault Monitors are given indications about the fault monitoring granularity, such as member level, type identifier level or host level. The Replication Manager is involved in the recovery process of the object group. For this, it needs to register as a consumer at the Fault Notifier. The Fault Monitors announce faults to the Fault Notifier that further announces it towards its consumers. For checkpointing purposes, application replicas must implement the Checkpointable interface and provide two methods in the application class. These are named *get_state* and *set_state*. Checkpointing has to be used in cold/warm passive replication. The standard also requires logging of method calls and replies in those cases. When using the active replication style the specification strongly recommends the use of a gateway for accessing the group members. There is also an indication about an alternative, namely the usage of proprietary broadcast primitives at the client, and thus, direct access to the group. To be able to manage large applications, the notion of fault tolerance domain is introduced. Each fault tolerance domain contains several hosts and object groups. There is one Replication Manager associated with it, as well as one Fault Notifier. Object Factories and Fault Monitors are specified as separate entities (objects) running on every host within the FT domain.

This standard architecture (specified by the OMG) is particularly suitable for TCP/IP networks, the main challenge presented within the context of our work is to adapt this architecture to support and exploit the broadcasting features of the Controller Area Network (CAN) and the CIOP protocol.

In the section below, we describe some properties of CAN and the CIOP protocol.

3.2. Some Properties of CAN

The CAN bus [12] is a priority bus targeted to operate in a noisy environment with speeds of up to 1 Mbit/s, exchanging, small real-time control messages. The

priority-based arbitration mechanism of the CAN bus can be exploited to guarantee the timely transmission of hard real-time messages under anticipated local and fault conditions (Tindell and Burns, 1994). In order to provide best effort scheduling of soft, real-time communication in a CAN bus while guaranteeing hard deadlines, different approaches may be applied (Davis, 1994; Liva.ni *et al.*, 19981) which are based on multiple priority classes.

The CAN protocol provides efficient hardware-implemented error handling, which is based on various error detection mechanisms with a very high total coverage, and an approach to immediately signalling the error condition. These features of the CAN protocol ensure atomic broadcast delivery in most fault situations. However, if an error occurs within the transmission time of the two last bits of a data frame, *some* receivers might accept and other receivers reject the frame. Although the sender (or in case of immediate sender crash an alternative mechanism like Eager Diffusion [14] or Shadow Retransmitter [16] will retransmit the frame, the consistent ordering of the incoming messages in different sites is not trivial due to the possible transmission of other (high-priority) messages between the first and the second transmission of a frame. Another problem caused by such errors is the duplicate frame reception by some receivers. In such situations, commercially available application level protocols for CAN like CAL (CiA, 1993), SDS (Grovela, 1994), and DeviceNet, (Noonen *et al.*, 1994) manage to discard frame duplicates, but they fail to provide consistent message ordering among a group of receivers.

Due to these broadcasting capacities and fail recovery, it is very challenging to run Fault-Tolerant CORBA applications on a CAN-based distributed platform. In previous work, we proposed the definition of new inter-orb protocol (CIOP) [1] customized for the CAN bus and Real-Time Network.

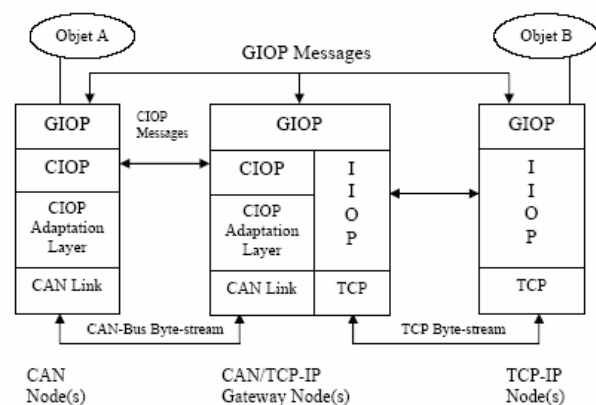


Figure2. CIOP and CIOP Adaptation Layer

As depicted in figure 2, the CIOP protocol is the transport layer developed to support the point to point communication over the CAN bus. One major part of the current work is to provide the CIOP protocol with group communication mechanism and a total ordering scheme and thus to support active replication.

4. Our Generic Architecture

We have chosen to use a Java based open source ORB, AdironOrb [17], as a basis for our enhanced infrastructure. To be able to build a fault tolerant application using our infrastructure, the application writer needs the following ingredients:

- the collection of service objects (Replication Manager, Object Factories, Fault Monitors, Fault Notifier, and Logging & Recovery Controllers)
- the enhanced CIOP protocol and the total order multicasting scheme
- the proper Portable Request Interceptors for different replication styles.

These constituents of our infrastructure are now explained in turn.

4.1. Service Objects

The Replication Manager, as specified in the standard, contributes to object group creation and recovery. It was implemented according to the specification. The Object Factory runs on every host connected to the CAN bus in the fault tolerance domain, and have the role to create a replica for an object group. The standard does not specify exactly how or where the created CORBA object replicas will run. In our implementation every replica runs in a different Java Virtual Machine (a new Unix process).

The Fault Monitor, as specified in the standard, runs on every host and monitors replicas. Besides that, also as a result of the specification, our implementation allows the monitoring at different granularities. Due to the usage of Java language, the most straightforward implementation of the monitor is as a CORBA object with a collection of internal threads. Also, a characteristic of our fault monitors is that they are forced to be reliable, by simply killing an object as soon as it is suspected to have failed. The Fault Notifier takes the fault reports generated by the Fault Detectors or the Fault Analyzers, filters them, and propagates them to entities that have registered for fault notifications, such as the Replication Manager, the Fault Analyzer or other application objects. In our

implementation we choose to map the Fault Notifier to the CosNotification Service because the Distributed Real-Time Notification Service developed within the context of RT-CORBA and the CAN bus [15] is more adapted for use with multicasting networks. The Fault Notifier creates a notification channel and registers itself both as a structured event supplier and a sequence of structured event supplier with the notification channel.

The FT-CORBA standard does not mention how or where the logging and checkpointing has to be done. In our implementation, the Logging & Recovery Controller is also a service (CORBA) object. It performs the checkpointing of objects on its host by using a separate internal thread, for every object. The state is recorded in a log located in memory (the location of the log could be easily changed to be on stable storage). Also, the method calls and replies are recorded in the same log, from where they are retrieved later at recovery. The idea is to have the logging and recovery objects on every host. The usage of these objects is dependent on the replication policy used. In case of cold passive replication the Logging & Recovery Controller at the location of a failed group member is contacted to obtain the latest state and set of method calls. In case of warm passive replication, the logged items are taken from the Logging & Recovery Controller on the new primary's host.

An ObjectGroup object is logically associated to each set of replicas (i.e., a group) of a given application server object: it receives all the requests directed to a particular group, executes them on each replica and is responsible, in the case of a stateful server, for maintaining consistency among the replicated data managed by replicas. Moreover, an ObjectGroup object implements a portion of the failure detection mechanism, it can trigger replica recovering and it can add/remove replicas to the group. To perform these tasks, each ObjectGroup object maintains the following information:

- the Interoperable Object References (IORs) of the object replicas and the information about their internal state and availability;
- the properties of a group, i.e., minimum number of replicas, the replication style, etc.

4.2. Extension of the ORB

We found that some of the basic ORB classes had to be extended. For example, in case of primary backup replication, the method for choosing the target address or the request had to be changed to be able to find the

primary's address (as opposed to the standard non-FTORB which had no notion of primary).

The standard recommends the usage of a retention identifier and a client identifier for a certain request, to uniquely identify it. To generate such unique identifiers, the ORB class itself had to be modified. Another extension to the ORB relates to the use of portable request interceptors that will be described below.

4.3. Portable Request Interceptors

The retention identifier and client identifier are sent in a request service context. A group reference version service context is also recommended by the standard. To add these service contexts at the client side, as depicted in figure 3, a portable *client* request interceptor is used. This applies both when the client is just a simple one, and when it is itself replicated. The interceptor is also used, in the latter case, for recording replies to outgoing requests. This avoids unnecessary resending requests if resends are attempted.

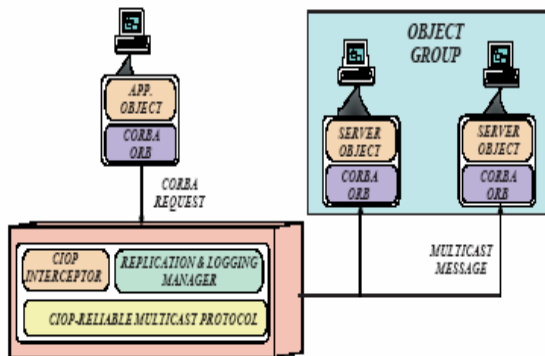


Figure 3. Portable Request Interceptors

Portable request interceptors have an important role at the server side as well. Here, we will talk about server request interceptors. Their main function is to contact the Logging & Recovery Controller for method call and reply logging purposes. For different server replication styles, different server interceptors are used. For example, in case of stateless replication style, there is no need for logging, at least not for logging of method calls. In case of warm passive replication style there is need for broadcasting of method calls to the rest of the Logging & Recovery Controllers for the group. Any changes to the group will cause a group reference version update. Thus, when serving a request one has to verify whether the client has the right version of the object group reference or not. This is also done in the server side interceptor. Sometimes, a request has to be stopped at the level of the server portable interceptor and the answer sent from that

level. For this purpose, a special exception had to be introduced as well as its handling. This was a further extension to the ORB. In addition, a mechanism for sending the reply from within the interceptor had to be devised.

In case of active replication, we choose the total ordered multicast protocol to access server object groups. In the next section we described the enhancements we introduced to our CIOP protocol to support group communications and totally ordered multicasting scheme.

4.4. CIOP: The Total Order Multicast

Transport Protocol for the CAN bus

CORBA fault-tolerant applications are structured as one or more group of objects (replicas) that cooperate by multicast messages to each other. The building of such applications is considerably simplified if the members of a group have a mutually consistent view of the order in which events (such as message delivery, process failures) have taken place.

In order to support real-time object groups with consistent information, the CIOP protocol must deliver real-time messages to all members of a group both *timely*, and in a *consistent order*. First, we define the *Message Group Name* field (13-bit) that is mapped to IOGR of the group objects of replicas and that should be encoded in the CIOP request header in order to support group addressing and multicast message filtering by the CIOP adaptation layer.

The second enhancement is to provide the CIOP protocol with mechanisms to be able to deliver all messages to replicas from the same group in a consistent and ordered way and thus to ensure the Strong Replica Consistency. Before to explain the total order multicasting scheme for the CIOP protocol, some definitions have to be introduced:

Atomic multicast delivery: Let G be a multicast group, i.e. a set of objects which must receive a multicast message. Let $del_i(m)$ denote the delivery of a message m to an object I and ' \rightarrow ' define the precedence relation. The two message m and m' are delivered atomically to the group G , if and only if the following holds:

$$\forall i, j : i, j \in G \text{ et } i, j \text{ non-faulty}$$

$$\Rightarrow (del_i(m) \rightarrow del_i(m'))$$

$$\Leftrightarrow (del_j(m) \rightarrow del_j(m'))$$

In other words, atomic delivery of multicast messages implies two properties:

- *Consistent Delivery*: if a multicast message is received by a non-faulty group member, then it is received by all non-faulty group members.
- *Consistent ordering*: messages received by different non-faulty group members, are received in the order.

Relying on the real-time scheduling policy implemented with the CIOP protocol [7] combined with a reliable broadcast delivery mechanisms, following assumptions can be made about the communication system in a CAN bus:

(A1) Hard real-time message are transmitted timely under anticipated fault scenarios, even in overload situations.

(A2) Soft real-time message are scheduled by dynamic priorities, but their deadline may be missed in overload situations.

(A3) If a non-faulty receiver receives a message, then eventually all non-faulty receivers receive the message.

Given these assumptions, atomic multicasting can be achieved by establishing a consistent global delivery order for multicast message. Relying on the assumptions **A1** through **A3** the CIOP implement an algorithm that achieves atomic multicast with minimum communication overhead, based on the knowledge about the message transmission deadline. Thus, to be able to order hard and soft real time message, the message deadline must be included in the CIOP message header (*CIOP_Request* and *CIOP_Reply*)

Let m and m' be two real-time messages with transmission deadlines d_m and $d_{m'}$. Then the deadline-based ordering algorithm implements the following rule:

$$d_m < d_{m'} \Rightarrow del_j(m) \rightarrow del_j(m')$$

This means that if the deadline of the message m is before the deadline of the message m' , then m must be delivered to destination objects before m' .

In order to minimize the communication overhead, the nodes must not exchange explicit information about the status of their queues. Thus they have to conclude this information from implicit knowledge. The proposed atomic multicast delivery algorithm is based on the following rules:

Order: Messages are ordered by their deadlines (priority).

R1: every received hard real-time message is delivered at its deadline. (This realizes the deadline-based order

implicitly)

R2: Every received soft real-time message must be delivered as soon as either another message with later deadline (less priority), or a bus idle time is observed after its transmission deadline.

Another enhancement implemented with the CIOP protocol is the *Most Recent Object Group Reference*. This mechanism allows the server to determine whether the client is using the most recent object group reference for the server object group when the client issues a request. This mechanism consists on encoding the CCDR encapsulation of **FTGroupVersionServiceContext** struct within **context_data** component, member of the **ServiceContext** which encoded with the CIOP message header. As depicted in figure 4, the structure of the CIOP message header is modified to support the new fields specified to enhance this protocol with group communication mechanisms.

```

Struct CIOPRequestHeader {
    Unsigned long requestID ;
    String      Message_group_name
    Octet      response_flag ;
    TargetAdress target ;
    String      operation ;
    ServiceContextList svc_ctxt ;
    Unsigned long deadline;

    };

Struct CIOPReplyHeader {
    Unsigned long requestID ;
    Unsigned long reply_status ;
    };

```

Figure 4. Structure of the CIOP Request and Reply

In the following, the atomic multicast delivery is presented. This algorithm assumes the availability of a real-time clock and allows the delivery of hard and soft real-time CIOP messages to the target objects in atomic order according to the rules mentioned above.

CIOP_orderd_multicast_delivery

initialization:

SRT_q ← empty-q ;

HRT_q ← empty-q ;

Next_HRT_delivery ← eternity ;

receive (m,t) : /* m is received at t */

if (m.dest_group ∈ my_groups)

and (m.category = HRT) **then**

```

HRT_q.insert (m);
if (m.dl < next_HRT_delivery then
    next_HRT_delivery ← m.dl ;
    wakeup (next_HRT_delivery);
if m.dest_group ∈ my_groups
    and m.category = SRT then
    SRT_q.insert (m);
    if (m.dl < t) then
        while (SRT_q.head.dl < m.dl) do
            mx ← SRT_q.gethead;
            SRT_deliver (mx, mx.dest_group);
    else
        while ( SRT_q.head.dl < t) do
            mx ← SRT_q.gethead;
            SRT_deliver (mx, mx.dest_group);
end receive;
wakeup():
    mx ← HRT_q.gethead ;
    HRT_deliver (mx, mx.dest_group) ;
    if (HRT_q ≠ empty_q) then
        next_HRT_delivery ← HRT_q.head.dl ;
        set_wakeup (next_HRT_delivery) ;
    else
        next_HRT_delivery ← eternity ;
end wakeup;
end CIOP_orderd_multicast_delivery

```

5. Conclusion

A growing number of CORBA applications with stringent performance requirements also require fault tolerance support. In this paper we have given an account in building a fault-tolerant middleware based on the FT-CORBA specifications adapted for the CAN bus features. We presented a strategy based on higher-level CORBA services, enhanced ORB Core and real-time ordering multicast protocol. To make FT-CORBA usable by performance-sensitive applications it must incur negligible overhead. To address this requirement, the CIOP protocol provides ordered multicast scheme based on implicit knowledge about messages deadline and don't require additional communications.

6. References

[1] T. Guesmi, S. Hasnaoui, Design and Implementation of a CAN-based Inter-ORB Protocol, Using RT-CORBA, Data Acquisition from Industrial Systems and the underlying Real-Time Control Area Network, In PDPTA'05 Int. Conf. Las Vegas, USA, June 2005

[2] Object Management Group, "Fault-Tolerant CORBA Specification V1.0" available as ftp.omg.org/pub/docs/ptc/00-04-04.pdf

[3] Priya Narasimhan, Louise E. Moser, and P. Michael Melliar-Smith, "Using Interceptors to Enhance CORBA", IEEE Computer, pp.62-68, July 1999

[4] Alberto Montresor, "Jgroup Tutorial and Programmer's Manual", Technical Report 2000-13 available at ftp.cs.unibo.it/pub/TR/UBLCS

[5] T.Kim, K.Kim, G.Jeon and S.Hong, "Integration subscription-based and connection-oriented communication into the embedded CORBA for the CAN Bus," *IEEE Real-Time Technology and Application Symposium, Washington D.C., USA*, May 2000.

[6] T.Kim, K.Kim, G.Jeon and S.Hong, "Resource-conscious customization of CORBA for CAN-based distributed embedded systems," *IEEE International Symposium on Object-Oriented Real-Time Computing, Newport Beach, CA, USA*, March 2000.

[7] T. Guesmi, S. Hasnaoui and H. Rezig, Using RT-CORBA Scheduling Service And Prioritized Network Traffic to Achieve End-to-End Predictability, In IEEE proceeding of ISCCSP' 06 Int. Conf Marrakech, Morocco, March 2006

[8] Robert van Renesse, Kenneth P. Birman, and Silvano Maffei, "Horus: A Flexible Group Communication System", Communication of the ACM, Vol. 39, Nr. 4, pp. 76-83, April 1996

[9] Mark C. Little and Santosh K. Shrivastava, "Integrating Group Communication with Transactions for Implementing Persistent Replicated Objects", Distributed Systems, Lecture Notes in Computer Science 1752, Springer Verlag, pp.238-253, 2000

[10] Priya Narasimhan, Louise E. Moser, and P. Michael Melliar-Smith, "Using Interceptors to Enhance CORBA", IEEE Computer, pp.62-68, July 1999

[11] Pascal Felber, Rachid Guerraoui, and Andre Schiper, "Replication of CORBA Objects", Distributed Systems, Lecture Notes in Computer Science 1752, Springer Verlag, pp.254-276, 2000

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

[13] P. Chung, Y. Huang, S. Yajnik, D. Liang, and J. Shih, "DOORS - Providing fault tolerance to CORBA objects" poster session at IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), The Lake District, England, 1998.

[14] J. Rufino, P. Verissimo, G. Arroz, C. Almeida and L. Rodrigues, Fault-tolerant broadcasts in CAN, In Int. Symposium on Fault Tolerant Computing, 1998

[15] T. Guesmi, S. Hasnaoui, H. Rezig, Design and Implementation of Real-Time Notification Service within the Context of Embedded ORB and the CAN Bus, In Proceeding of ACM Symposium on Applied Computing, Dijon, France 2006

[16] M. A. Livani, J. Kaiser and W.J. Jia, Scheduling hard and soft real-time communication in the controller area network (CAN). In proceeding IFAC/IFIP Workshop on Real Time Programming, 1998

[17] Adiron secure system design, webpage <http://www.adiron.com>