

# Implementing Software Replication Through CORBA Interceptors: Lessons Learned

Massimo MECELLA, Paolo PAPA, Antonino VIRGILLITO

Carlo MARCHETTI, Roberto BALDONI

Dipartimento di Informatica e Sistemistica

Università di Roma "La Sapienza"

Via Salaria 113, I-00198 Roma, Italy.

{mecella,papa,virgi,marchet,baldoni}@dis.uniroma1.it

## Abstract

*The Common Object Request Broker Architecture (CORBA) currently does not provide any specific support for software replication, a core aspect of reliable and dependable distributed applications. In this paper we discuss the use of CORBA interceptors, introduced by the Object Management Group in the CORBA 2.2 specification, as a basic building block for handling object replication. More specifically, we show some lessons learned while programming CORBA interceptors provided by several ORBs. From this experience we grasped several guidelines that could be followed by ORB designers for the implementation of CORBA interceptors and for the specification of the interface provided to programmers in order to get an easier development of reliable and dependable distributed applications.*

**Keywords:** CORBA, Interceptors, Distributed Systems, Replication, Fault-Tolerance.

## 1 Introduction

The Common Object Request Broker Architecture (CORBA) is an established standard for object-oriented distributed applications used in many contexts (e.g., object-oriented distributed databases, wrapping of legacy systems, cooperative-work applications, etc.) where heterogeneous technologies have to coexist. However, CORBA paid little attention on providing tools for building reliable distributed applications which could take an effective advantage from the distributed nature of the platform. As an example, reliable applications need to handle object replication ([7],[1]) which, in turn, needs one-to-many communication paradigm in order to disseminate status information across the replicas. CORBA communication is mainly based on an unreliable one-to-one basis.

This is why the first attempt to provide fault-tolerance and high-availability on CORBA was to *integrate* an ORB (e.g., Orbix [9]) with a group toolkit (e.g., Isis [2]). The latter provides the ORB with the abstraction of object group as it was a single object, and several crucial services for sharing a common state among the replicas, such as group membership, state transfer, atomic broadcast, etc. [10]. Drawbacks of this solution are: (i) the ORB needs to be modified in order to translate one-to-one CORBA communication in multicast system calls and (ii) the usage of a heavy-weight (and proprietary) group toolkit.

In order to avoid a tight integration between ORB and group toolkit, Narasimhan, Melliar-Smith and Moser proposed an architecture, namely Eternal [13], that *intercepts* all local Operating System (OS) calls, issued by the ORB, before they reach the OS kernel and redirects them to a group interface that provides group abstraction. This interface uses a Reliable Multicast system such as Totem [12]. While the idea to intercept (transparently to the ORB) ORB requests is worthwhile, it is not easy to develop an interception layer for any OS.

Shiper et al. [6] proposed an architecture, namely Object Group Service (OGS), compliant with the CORBA philosophy. OGS is a CORBA service composed by a set of sub-services such as consensus, reliable multicasting, messaging and monitoring. To get reliability, a client and a server must use methods provided by the OGS, and this introduces a lack of transparency when writing client/server code.

Recently, the Object Management Group (OMG) has introduced the notion of CORBA interceptors in the CORBA specification 2.2 [15]. The aim of CORBA interceptors is to add ORB services to a CORBA object in a transparent, flexible and portable way. The interception layer is logically interposed between a client and a server object. Operationally, the client and the server have their own customizable interceptors that cooperate to offer ad-hoc functionalities such as encryption, compression, access control, etc. This frees CORBA applications to handle such functionalities. As an example, if a pair of running CORBA objects will require in the future, for security reasons, to exchange encrypted and compressed data, one option is to use customized CORBA interceptors. This choice has a very low impact on the source code of both objects.

In this paper we present some practical lessons learned by programming CORBA interceptors for handling object replication, in order to understand which tasks can be executed by interceptors for enhancing reliability in CORBA applications<sup>1</sup>. The more tasks can be embedded in the interceptors, the less the required functionalities external to the ORB will be. This approach is CORBA compliant and reduces as much as possible the interactions between the ORB and a lightweight group toolkit.

We studied several common problems encountered when handling object replication: group abstraction and replication management, ordered and reliable client-server interaction, failure

---

<sup>1</sup>This work is part of a project for adding reliability to CORBA applications which is currently worked out in our department. One of the aims of this project is to maximize the work done in handling object replication by the software layers "over" the ORB with respect to the one done by those "below" the ORB.

detection and notification. For each of these problems we show limits and capabilities of CORBA interceptors. From lessons learned we grasped some guidelines on how CORBA interceptors should be implemented in order to make easier the solutions to previous problems.

The rest of the paper is structured into four sections. Section 2 describes CORBA interceptors, Section 3 proposes the practical lessons learned and Section 4 suggests guidelines for implementing interceptors for adding reliability to CORBA applications. Section 5 concludes the paper.

## 2 CORBA Interceptors

The Common Object Request Broker Architecture (CORBA) defines a framework for developing object-oriented, distributed applications. It supports the development and the integration of object-oriented software components in a heterogeneous, distributed computing environment. The two major components of CORBA are the Object Request Broker (ORB) and the Interface Definition Language (IDL). The former enables communications between clients and remote server objects, the latter is a declarative language that describes the interfaces to the server objects. CORBA communications are unreliable and (i) synchronous (i.e., the client object invokes a method and is blocked till it receives the invocation result), (ii) one-way and (iii) deferred synchronous (i.e., the client object invokes a method, continues its execution, and then later can poll for the reply)[16]. The synchronous interaction is, for its simplicity, the most widely used CORBA communication mechanism.

The last CORBA specifications (2.2 and 2.3) define further concepts, such as the one of interceptor, useful for enhancing applications with specific aspects such as security or fault-tolerance without dropping the fundamental principles of independence and transparency between client and server objects. The interceptor of an object encompasses the object capturing every incoming and outgoing message (request or reply). The inteceptor maintains its own variables and data structures which are useful when running complex protocols. There are two layered levels of interception (see Figure 1(a)):

- Request Level Interceptor (RLI). It operates on a structured request object.
- Message Level Interceptor (MLI). It operates on a buffer containing the message as it is sent on the network.

Every message can be processed sequentially through RLI and MLI. These two levels differ in the kind of manipulations they can perform. RLI handles request objects that can be then accessed and altered in various ways. For example, an RLI can be used to provide access control on objects or to dynamically redirect the request to different target objects. Therefore, typical applications of RLI are security, reliability and profiling. MLI allows more low-level operations on messages, such as data encryption, data compression and piggybacking of information.

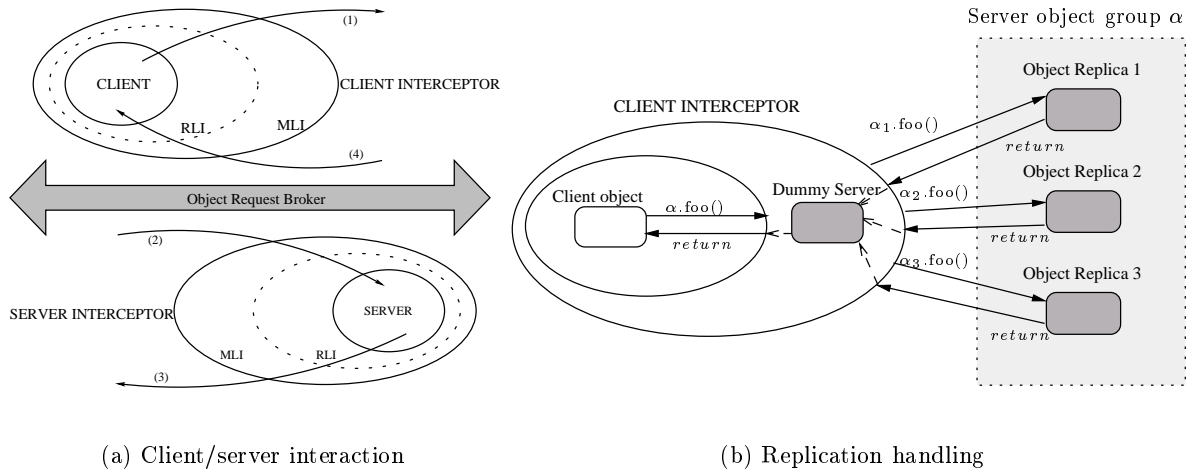


Figure 1: Object replication through interceptors

### 3 Lessons Learned

We studied several common problems encountered when handling object replication such as group abstraction and replication management, ordered and reliable client-server interaction, failure detection and notification . For each of these problems we show limits and capabilities of CORBA interceptors. All the proposed solutions have been implemented on two public domain CORBA systems, MICO [4] and JacORB [8].

In the following paragraphs, a logical model of the interception mechanism will be used in describing the proposed solutions. As shown in Figure 1(a), a client/server interaction is composed of four interceptor event handlers: (1) when the client issues the request, (2) when the server receives this request, (3) when the server issues the reply after executing the requested method and (4) when the client receives this reply. Each event handler adds a specific behaviour; the proposed protocols require to customize event handlers (1), (2) and (4).

**Lesson 1: Handling Object Replication.** Using CORBA interceptors it is possible to catch the client request and to forward it to a set of servers. This operation is hidden from the client. The protocol is described by the event handlers (1) and (4) shown in Figure 3. An example is shown in Figure 1(b): let  $\alpha$  be the reference of a "virtual" object. When the client invokes the method  $\alpha.foo()$ , this request is caught by the interceptor that, in turn, invokes the method  $\alpha_i.foo()$ , in a deferred synchronous way, on each replica forming the virtual object  $\alpha$ . As soon as the interceptor receives the first reply, the result is sent to the client object (line 4.3). Other replies from the other replicas are filtered out by the interceptor (line 4.1) and discarded (line 4.2). Note that the mapping among the "virtual" object and the replicas can be either coded in the program text of the interceptor (as we assume in this paper for simplicity) or, preferably,

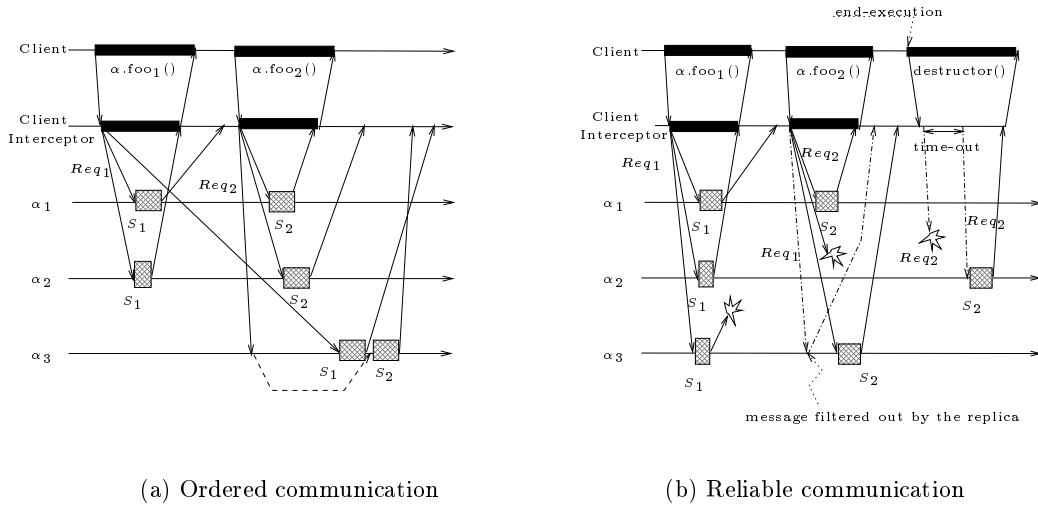


Figure 2: Ordered and reliable communication

retrieved from an "entity" that maintains a table of correspondence<sup>2</sup>.

When implementing an interceptor handling the previous scheme, request duplication does not pose any problems. Requests are duplicated and forwarded by using the CORBA Dynamic Invocation Interface (DII). Unfortunately, DII does not give any support for handling replies. In particular, for handling message filtering and for returning the result to the client. In the latter case, the interceptor must copy the result, contained in the first reply received from the replicas, in the `CORBA::Request` object where the client is waiting for its result. This operation is available only in the Dynamic Skeleton Interface (DSI), which, in turn, is available only on the server side of a CORBA system. As a consequence, it is necessary to encapsulate a "dummy server" in the client interceptor (see Figure 1(b)). Message filtering is not supported at all (see Section 4 for additional details).

**Lesson 2: FIFO Communication.** Using the previous replication protocol in CORBA interceptors, it makes possible that some requests, sent from the same client, arrive out-of-order to some replicas<sup>3</sup>. As an example, Figure 2(a) shows a communication pattern in which *Req2* arrives at replica  $\alpha_3$  before *Req1*. Out-of-order method execution may lead to a different state in object replicas. Hence, *Req2* must be delayed till *Req1* arrives at the  $\alpha_3$  server interceptor.

A simple protocol can be embedded in the interceptor of the client and in the one of the server to avoid this problem. This protocol is composed by the event handlers (1) and (2) depicted in Figure 3. A client maintains an integer variable *SN*, initialized to zero, that represents the sequence number of a request. This value is, first, piggybacked on the outgoing request (line

<sup>2</sup>In a recent response to the OMG RFP for Fault-Tolerant CORBA, Moser and Martin pointed out that such mapping can be done by a "Replication Service" running on top of the ORB [11].

<sup>3</sup>For simplicity, in this paragraph we assume reliable communication in order to focus on the ordering aspect. In the next paragraph, we drop this assumption.

1.1) and then it is increased by one (line 1.3). The server interceptor endows a variable, denoted  $SN_{client}$  (initialized to zero), indicating the expected request from  $client$ . Upon the arrival of a request at the server interceptor, it extracts the sequence number and waits until the sequence number of the incoming request is the one expected by the replica (line 2.1). Then the request is passed up to the object and  $SN_{client}$  is increased.

The programming of the event handler (2) requires many lines of sequential code (especially to implement step (2.1)), unless interceptors provide a multithreading mechanism that allows to handle each incoming request on a separate thread.

```

(1) when a CORBA::Request  $m$  with destination  $\alpha$  is caught by the client interceptor
    (1.1)  $m.sn := SN; m.sender := client;$ 
    (1.2)  $\forall i \in \{1, \dots, n\}$  send( $m$ ) in deferred synchronous way to  $\alpha_i$ ;           % using DII %
    (1.3)  $first_{SN} := true; SN := SN + 1;$ 

(2) when a CORBA::ServerRequest  $m$  is caught by the  $\alpha_i$  interceptor
    (2.1) wait ( $m.sn = SN_{m.sender}$ )
    (2.2) pass( $req$ ) up to object  $\alpha_i$ ;
    (2.3)  $SN_{m.sender} := SN_{m.sender} + 1;$ 

(4) when a CORBA::Request  $m$  from a replica is caught by the client interceptor
    (4.1) if  $\neg(first_{m.sn})$ 
    (4.2) then discard( $m$ );
    (4.3) else pass( $m$ ) up to client object;  $first_{m.sn} := false;$            % using DSI %

```

Figure 3: A simple FIFO protocol using interceptors.

**Lesson 3: FIFO and Reliable Communication.** Handling message losses requires interceptors to embed mechanisms for message retransmission (at client interceptor side) and filtering (at server interceptor side). As an example, Figure 2(b) shows a communication pattern in which the request  $Req_2$  is lost two times and therefore the client interceptor needs to send it again. It is shown also the situation in which the reply to  $Req_1$  is lost. The client interceptor sends again the request. The replica  $\alpha_3$ , that already executed the method, needs to filter out this request (in order not to execute the service again).

Figure 4 shows an extension of the protocol of Figure 3 to manage retransmission and filtering. For each replica  $\alpha_i$  the client interceptor maintains a variable  $ACK_i$  representing the sequence number of the last received reply<sup>4</sup>. In order to perform retransmission, a client interceptor maintains a log containing the issued requests not yet acknowledged by all the replicas. This log is accessed and updated by the following self-explicative functions: **log-retrieve**, **log-insert** and **log-remove**. Every time a new request is issued by the client, the intercep-

<sup>4</sup>A reply is used by the client interceptor as an acknowledgement of the method execution. Note that as every replica executes the methods in an ordered way, the arrival of a reply with sequence number  $x$  implies that all the requests with sequence number  $sn \leq x$  have been executed.

tor retransmits all the unacknowledged requests (line 1.1), then it executes the same protocol described in Figure 3 and finally inserts the new request in the log (line 1.5). When a server interceptor receives a request (event handler (2)), it is necessary to filter out already served requests (line 2.1), to send an acknowledgement and to discard them (line 2.2). If the request is not discarded, the same protocol of Figure 3 is executed. When the event handler (4) receives a reply  $m$  from a replica, it executes the same steps shown in the protocol of Figure 3, and then it updates the *ACK* variable related to the sender replica (line 4.4). Finally, it checks if the request  $m$  can be removed from the log by testing if all the replicas acknowledged the request with sequence number  $m.sn$  (line 4.5).

As the added code concerns mainly predicate evaluation and local information management, it does not pose any additional problems with respect to the ones pointed out in the previous lessons (about filtering and using DSI).

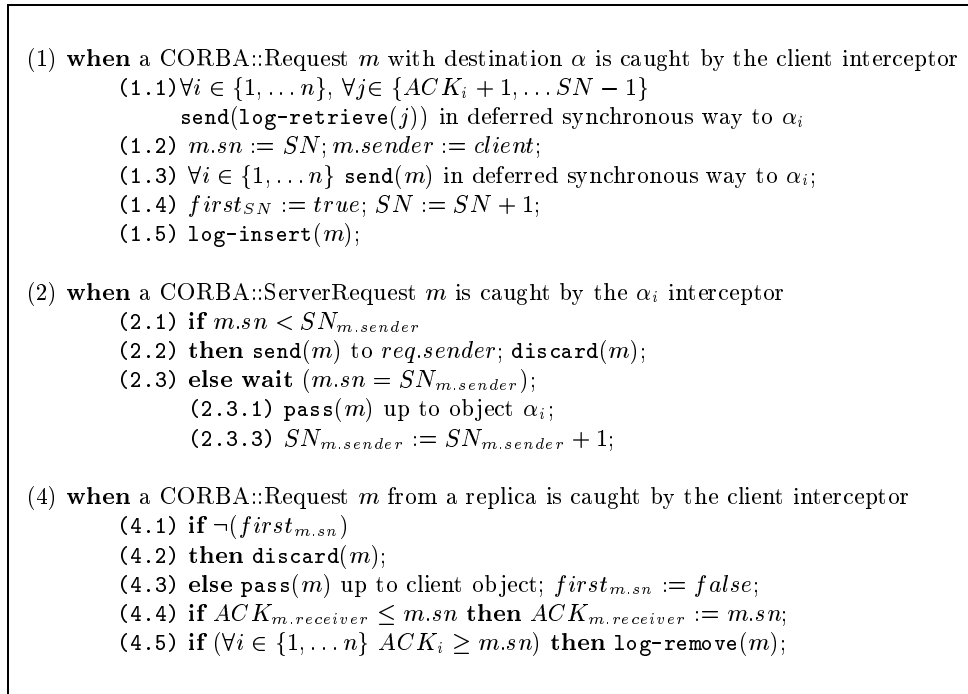


Figure 4: A FIFO and reliable protocol using interceptors.

**Lesson 4: Handling client-replicas consistency.** When a client ends its execution, it is necessary that each replica executes the same sequence of methods, issued by the client, in order to leave all the replicas in a consistent state with respect to that client<sup>5</sup>. Therefore the client interceptor, before ending its execution, must assure the condition ( $\forall i \in \{1, \dots, n\} ACK_i = SN - 1$ ). A solution to this problem is to customize the destructor of the client interceptor

<sup>5</sup>Note that this consistency criterion can be a basic block for getting more constrained methods ordering. For example, if clients issue their sequence of methods on the same set of replicas upon the receipt of a token, we obtain a computation in which method executions on the distinct replicas are totally ordered.

(see the right side of Figure 2(b)). The destructor (and hence the client) does not end until the previous condition is satisfied. At this aim, it embeds a time-out retransmission mechanism. Also in this case the implementation does not present particular problems, if the programming language allows to customize the interceptor destructor (as in C++).

**Lesson 5: Handling Failures.** In the previous part of the paper we have implicitly assumed that objects do not fail during the computation. The detection of such failures is clearly a key point when developing any kind of distributed applications. Unfortunately, current CORBA specification does not provide any support for failure detection and notification. Moreover this kind of features would require checkpointing and logging mechanisms for executing a safe rollback recovery [3]. Hence every effort in handling failures cannot do without new CORBA standard components specifying the previous mechanisms.

Therefore no one of the systems we used provides any (even rudimentary) failure detection mechanisms. Experimentally, we remarked that a failure of an object over a running ORB can be detected by using an exception handler associated with the CORBA `bind()` function.

## 4 Guidelines in Implementing CORBA Interceptors

During the implementation of the previous protocols we encountered several problems that derive either from incomplete specification or from specific ORB implementations ([4],[8]). Therefore the aim of this section is twofold: on the one hand we point out ORB implementation problems, on the other hand we underline some issues that should be addressed in the next CORBA specification.

1. In any interceptor implementations we tested, a client (resp. server) interceptor cannot access the server (resp. client) interfaces. This comes from the fact that an interceptor is seen by implementors as an extension of the object (client or server). When one has to cope with object replication, the interceptor of a client has to access the DSI in order to return the first result to the client (see Lesson 1). It would be preferable that every interceptor could access both interfaces (DII and DSI) independently of the object activating it. This corresponds to a logical vision of the interceptor as an ORB extension and a customizable bridge between objects and ORB (as it is already stated by CORBA specification).
2. We had many problems to filter out messages inside an interceptor. There is no method clearly responsible for doing this operation. Let us remark that the implementation of any replication policy requires message filtering. So the specification and the successive implementation of explicit methods carrying out this job is needed.
3. Concerning the implementation of the interceptors we remarked several differences. In MICO, the interceptor code does not run in a separate thread. If the interceptor is blocked,

the client is blocked too. This makes really difficult (and with low performances) the implementation of a protocol similar to the one proposed in Figure 3 (in particular the implementation of the wait statement). JacORB allows an "interceptor-per-invocation" semantic: it activates an interceptor thread every time a request is caught. Multiple threads can share common variables, therefore the implementation of previous protocols is easier and effective.

4. In the Lesson 4 we customized the destructor of the interceptor to get consistency among the replicas with respect to the invocation done by a client. Of course, the customization of the interceptor destructor can be useful in many other contexts where some conditions must be verified before an object ends its execution. However, we can use this technique only in programming languages such as C++. Other languages, like Java, do not offer this feature, then an explicit customizable method `interceptor_Deactivate()` should be invoked by an object before ending its execution. This makes less transparent to the object the existence of the interceptor, but, on the other hand, the object is already aware of the existence of the interceptor as a programmer must include in the its code the interceptor activation call.

## 5 Conclusions

The aim of this work was to use standard CORBA 2.2 interceptors in order to test if they can be used to set up fault-tolerant distributed object-oriented applications. We presented several typical problems encountered when handling replication of objects and we showed the protocols, to be embedded in the interceptors, that can solve them. These protocols can be used as building blocks for fault-tolerant programming.

Our opinion is that customizing interceptors can be a reasonable answer of previous problems without impacting on the client-server one-to-one interaction typical of CORBA compliant systems. One of the main drawbacks is the difficulty we encountered in customizing interceptors. However, OMG is working on the specification of portable interceptors that should allow to overcome many of the programming problems.

Moreover, one of the main issues we remarked is that programming interceptors cannot be the solution to the problem of fault-tolerant programming, there is the need of several additional mechanisms such as failure detectors, failure notifiers, replication manager (as an entity offering the group abstraction) as remarked by other studies [13] and by recent OMG investigations. On the other hand, this study shows that interceptors can be used to move part of the complexity of building fault-tolerant distributed application "above" the ORB. This simplifies the services we need "below" the ORB, making the development of fault-tolerant applications less dependent on the operating system platform.

## References

- [1] Baldoni R., Bonamoneta S., Marchetti C.: "Implementing Highly-Available WWW Servers Based on Passive Object Replication". *Proceedings of The Second IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'99)*, 1999.
- [2] Birman K., Van Renesse R., Eds.: *Reliable Distributed Computing with the ISIS toolkit*. IEEE Computer Society press, 1994.
- [3] Elnozahy E.N., Alvisi L., Johnson D.B. Wang Y.M.: "A Survey of Rollback-Recovery Protocols in Message-Passing Systems". *Technical Report No. CMU-CS-99-181, School of Computer Science, Carnegie Mellon University*, 1999.
- [4] Fachbereich Informatik - Johann Wolfgang Goethe Universität, Frankfurt am Main: *The MICO system*. <http://www.mico.org/>.
- [5] Felber P., Guerraoui R., Schiper A.: "Replicating Objects with CORBA Event Channels". *Proceedings of IEEE International Workshop on Future Trends in Distributed Computing Systems (FT-DCS'97)*, 1997.
- [6] Felber P., Guerraoui R., Schiper A.: "The Implementation of a CORBA Object Group Service". *Theory and Practice of Object Systems*, John Wiley & Sons, Vol.4, No.2, 1998.
- [7] Guerraoui R., Schiper A.: "Software-Based Replication for Fault Tolerance". *IEEE Computer*, April 1997: 68-74, 1997.
- [8] Institute of Computer Science - Freie Universität Berlin, Department of Mathematics and Computer Science: *The JacORB system*. <http://www.inf.fu-berlin.de/~brose/jacorb/>.
- [9] IONA Technologies, Inc.: *The Orbix system*. <http://www.iona-iportal.com/suite/orbix.htm>.
- [10] Landis S., Maffei S.: "Building Reliable Distributed Systems with CORBA". *Theory and Practice of Object Systems*, John Wiley & Sons, Vol.3, No.1, 1997.
- [11] Moser L.E., Martin R.J.: *Fault-Tolerance for CORBA*. Eternal System Inc., Sun Microsystem Inc., University of California - Santa Barbara, OMG Document orbos/98-10-08, 1998.
- [12] Moser L.E., Melliar-Smith P.M., Agarwal D.A., Budhia R.K., Lingley-Papadopoulos C.A.: "Totem: A Fault-Tolerant Multicast Group Communication System". *Communications of the ACM*, Vol.39, No.4: 54-63, 1996.
- [13] Moser L.E., Melliar-Smith P.M., Narasimhan P.: "Consistent Object Replication in the Eternal System". *Theory and Practice of Object Systems*, John Wiley & Sons, Vol.4, No.2, 1998.
- [14] Narasimhan P., Moser L.E., Melliar-Smith P.M.: "Using Interceptor to Enhance Corba". *IEEE Computer*, July 1999: 62-68, 1999.
- [15] Object Management Group: *The Common Object Request Broker: Architecture and Specifications*, Rev. 2.2. Framingham, Mass., February 1998.
- [16] Pope A.: *The CORBA Reference Guide: understanding the Common Object Request Broker Architecture*. Addison Wesley, 1997.