# Implementing Simple Replication Protocols using CORBA Portable Interceptors and Java Serialization

Taha Bennani, Laurent Blain, Ludovic Courtes, Jean-Charles Fabre,
Marc-Olivier Killijian, Eric Marsden, François Taïani
*LAAS-CNRS, 7, avenue du Colonel Roche, 31077 Toulouse Cedex 4, France*

**Copyright Notice**

© 2002 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.
This material is presented to ensure timely dissemination of scholarly and technical work. Copyright and all rights therein are retained by authors or by other copyright holders. All persons copying this information are expected to adhere to the terms and constraints invoked by each author's copyright. In most cases, these works may not be reposted without the explicit permission of the copyright holder.

This article will be presented at *the International Conference on Dependable Systems and Networks* (DSN-2004) to be held in Florence, Italy on June 28th - July 1, 2004. It will be published in the proceedings of the aforementioned conference.

# Implementing Simple Replication Protocols using CORBA Portable Interceptors and Java Serialization

Taha Bennani, Laurent Blain, Ludovic Courtes, Jean-Charles Fabre,
Marc-Olivier Killijian, Eric Marsden, François Taïani
LAAS-CNRS, 7, Avenue du Colonel Roche, 31077 Toulouse cedex 4 – France

## Abstract

The goal of this paper is to assess the value of simple features that are widely available in off-the-shelf CORBA and Java platforms for the implementation of fault-tolerance mechanisms in industry-grade systems. This work builds on knowledge gained at LAAS from previous work on the prototyping of reflective fault tolerant frameworks. We describe how we used the interception and state capture mechanisms that are available in CORBA and Java to implement a simple replication strategy on a small middleware-based system built upon GNU/Linux and JOrbacus. We discuss the benefits and the limits of the resulting system from a practical point of view.

## 1. Introduction

The objective of this practical experience report is to illustrate the extent to which basic mechanisms like CORBA *Portable Interceptors* and *Java Serialization* can be used to implement simple replication protocols. The portable interceptor mechanism introduced by recent versions of the CORBA standard [1] offers a powerful means to intercept interactions in a distributed system, while Java's serialization mechanisms [2] provide a transparent and portable state capture facility. From a practitioner's point of view, however, the interest of these features for the implementation of fault tolerance mechanisms remains to be assessed. We should emphasize that the objective of this paper is not to present a full-fledged framework for distributed fault-tolerance. Many works with industrial relevance already provide such frameworks, in particular the Fault-Tolerant CORBA standard [3], and corresponding implementations such as IRL [4] and Eternal [5]. Other existing approaches use portable interceptors, such as the FTS [6] project., but require the use of a non-standard object adapter to implement the group communication routines, resulting in heavyweight modifications of the underlying ORB. Our primary goal is to investigate lightweight implementation approaches based on standard off-the-shelf platforms and to evaluate how far this line of attack can be pushed. In this paper, we investigate this question by reporting on experiments performed on a prototype platform called DAISY (*Dependable Adaptive Interceptors and Serialization-based sYstem*) that uses these technologies. DAISY is based on our yearlong experience in the design and validation of fault tolerant reflective platforms and takes into account the lessons learned from their implementation [7].

The paper is organized as follows: In Section 2 we present our motivations in this work and the basic elements of our middleware-based platform. In Section 3 we describe how portable interceptors were used to develop a simple replication strategy. In Section 4 we discuss the benefits and the limits of the current version of these basic off-the-shelf mechanisms.

## 2. Motivations and platform

A computer system is said to be *reflective* if it can observe and modify itself as part of its own computation [8]. The use of reflection introduces a clean separation in a system's structure between what is called the *base level*, where normal computation occurs, and the *meta-level*, where computation about the system takes place. The interactions between levels are categorized as *reification* (when the meta-level is notified of a change occurring at the base level), *introspection* (when the meta-level observes the base level), and *intercession* (when the meta-level modifies the structure or the behavior of the base level).

These mechanisms are useful for separation of concerns and have made reflective architectures particularly attractive to implement non-functional mechanisms such as fault tolerance and security in more transparent and more adaptable ways. In the last decade, several projects have investigated this issue (Maud [9], Garf [10], Friends [7]). The Friends system for instance is a CORBA-compliant reflective fault-tolerant platform, based on a Meta Object Protocol that is built using open compilation facilities [11]. Today, limited reflection capabilities have made their way into some of the most popular development platforms, such as CORBA and Java. This evolution highlights the increasing attractiveness of reflection for industry, to dynamically adapt non-functional requirements. To investigate the benefits and limitations of these standard reflective facilities, we describe here how passive replication of CORBA objects

can be implemented. This platform is currently being used to develop other replication strategies and more importantly to implement reflection at various levels, both at the middleware and the operating system levels, using the concept of multi-layer reflection we presented at DSN 2003 [12].

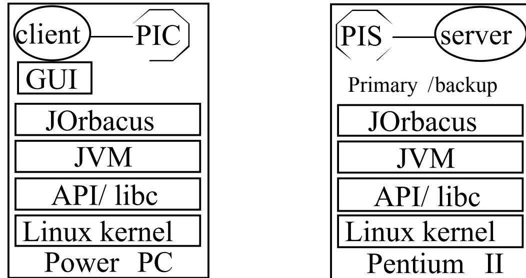



**Figure 1. Middleware-based platform**

Our prototype platform (see Figure 1) is organized in a layered architecture that integrates the middleware layers ORBacus (JORBacus v4.1.2), and Sun's Java v1.4.1, on top of the GNU/Linux operating system (Linux 2.4 for x86).

## 3. Implementation

Our implementation of the passive replication mechanism combines *Portable Interceptors*, to synchronize remote clients and the different server replicas, and the *Java Serialization* mechanisms to obtain the internal state of application objects.

We use two kinds of Portable Interceptors, according whether they handle the client or the server side of the interaction (Figure 2). On the client side, the **PIClient** intercepts the outgoing requests issued by clients and forwards them to the current primary replica. On the server side, the **PIServer** has two running modes: primary and backup. The PIServer handles the processing of requests, the transfer of request, reply and state information from the primary to the backup during fault-free phases, and ensures smooth reconfiguration when one of the replicas crashes. The PIServer of the primary delivers requests to the backup replica. We do not assume the availability of atomic multicast services.

The PIClient and PIServer work together to detect the crash of the Primary server and switch from the primary to the backup server. As crash detection is not the primary focus of this paper, we choose to use a simple client-side detection scheme that relies on the error detection mechanisms of the underlying ORB. Clients first discover crashes when the ORB raises an exception concerning the connection with the primary. After retrying the request a given number of times, a client issues its following requests to the backup replica. When the backup receives a request from a client, its PIServer checks that the primary is down by sending a ping request to the primary, then switches to primary mode. Duplicated requests on the server side are discarded using a system-wide unique ID.

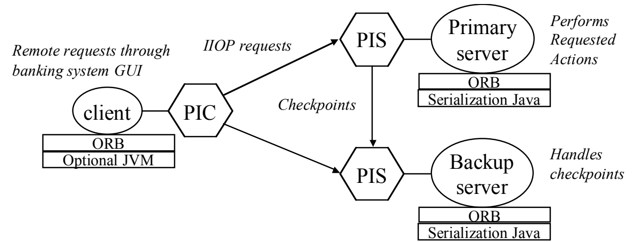



**Figure 2. Architecture with PIClient & PIServer for primary-backup replication**

Using these Portable Interceptors we can *observe* the ongoing requests between clients and the server, and trigger the different stages of the replication protocol accordingly. However, except for a number of exception-raising or -resignaling techniques described below, CORBA's portable interceptors do not allow requests or replies to be *modified*. In addition, a portable interceptor inherits its concurrency model from the server that it is attached to. Depending on this concurrency model, requests can be served either sequentially or in parallel; this is completely beyond the control of the PIs.

In this context, the only possibilities for controlling request handling from within a portable interceptor are either to suspend the current thread, thus blocking the request, or to throw an exception to prevent the request from being processed. A portable interceptor can intercept the sending and the receiving of a request, the reception of a reply, and the reception of CORBA exceptions. This last feature can be used to "transform" a CORBA exception, by propagating a different exception from the intercepted one (we use this technique, as we shall see).

### 3.1. Overview of the implementation

The behavior of the client's interceptor (PIClient) is shown by the pseudo-code below. The PIClient comprises three entry points: the `send_request` method is called by the ORB before an outgoing request is sent to the transport layer; the `receive_reply` method is called before an incoming request is handed to the application; and the `receive_exception` method is called by the ORB before a CORBA exception is delivered to the application level.

```
Constructor:
   obtain the primary and backup
   references through the name server,
   determine the client unique ID
send_request:
   add the request unique ID for the
   client to the request (= client
   unique ID + request number)
receive_reply:
   increment request counter
receive_exception:
   issue ForwardRequest exceptions to
   the primary (twice), then the backup
```

The core of the client-side interceptor's role takes place in the `receive_exception` method when the ORB signals that a transient communications error has affected a message sent to the server. A `ForwardRequest` is a special kind of exception that a CORBA server can raise to redirect a client to a different server. When it detects a communication error, the client interceptor raises a `ForwardRequest` exception to tell the client to reissue its request (this is a case of exception transformation, as mentioned earlier). After two consecutive errors from the primary, the client interceptor assumes that the primary has failed and redirects further requests to the backup, thus triggering a primary-backup switch.

On the server side, the primary and backup replicas each contain a PIServer working in two different modes, respectively primary and backup mode:

1. In primary mode, the PIServer relays requests to the primary server for processing, fetches the server's internal state after processing (using Java serialization), and sends a checkpoint to the backup replica. The checkpoint includes the ID of the corresponding request, the state information and a copy of the reply message, if any.
2. In backup mode, the PIServer receives checkpoints and uses them to update the state of the server replica using Java serialization. When the backup receives a primary failure notification, it switches to "primary mode" and starts processing requests. The unique request identifier included in both the requests and the checkpoints is used to avoid processing the same request twice when the backup switches to primary mode.

For clarity, we first show the pseudo-code for a scenario with a *single* client in the system. The interceptor's `receive_request` method is invoked by the ORB before a request is transferred to the server object, and the `send_reply` method is called before a reply is passed to the transport layer. The modifications necessary in the multi-client case are discussed later.

```
Constructor:
if (primaryRole)
   obtain the reference of the backup
Determine server unique ID
```

```
receive_request:
if (primaryRole)
   if first activation of PIServer
    obtain the reference of
    the Primary Server
    /** will be used to issue direct
    requests to the primary, like
    getState for instance **/
else   /** backup role **/
   if first activation of PIServer
    obtain the references of the Primary
    and the Backup Servers
   if current request is 'set_state'
    /** fault-free behavior **/
    Backup.set_state(bufferedState)
    bufferedState     := primaryState
    bufferedRequestID := requestID
   else  /** primary has crashed **/
    primaryRole := true  ;
    if (requestID==bufferedRequestID)
       /** re-execution to produce reply
   **/
       bufferedState     := nil ;
       bufferedRequestID := nil ;
    else   /** the received state info
   can be applied **/
       Backup.set_state(bufferedState)
    endIf
   endIf
endIf
```

```
send_reply:
if(primaryRole)
   state=Primary.get_state()
   backup.set_state(state,<requestID,cli
   entID>)
```

The start of `receive_request` contains some reference handling due to the fact that a portable interceptor is initialized *before* the CORBA server it is attached to. Thus, the reference of the primary and the backup cannot be obtained during the PI's initialization and must be obtained afterwards, once the system is completely initialized.

In the `receive_request` code, `set_state` requests received from the primary are not applied immediately by the backup. They are buffered until a new `set_state` request arrives (in which case they are applied), or until the backup becomes primary. The `set_state` requests must be buffered because an interceptor is not able to modify a request, and in particular can't force a request to return a predefined result (this limitation will be discussed in Section 4). When the primary crashes, the client and the remaining backup must reach a common view on *when* the crash occurred: (i) while the primary was idle, or (ii) while a request was being handled, or (iii) just after a `set_state` request was sent and *before* the corresponding reply goes out. Cases (i) and (ii) are easy to handle: the backup applies the last `set_state` request it received, and takes over the primary's role. Case (iii) is more difficult to tackle. In this case, the client re-issues its request to the backup, although the primary has already generated a `set_state` message for this request.

Because we can’t impose a result on a request from within a portable interceptor, the only possibility for the backup to answer the client is to **re-execute its request.** For this reason we must discard the last `set_state` request that was received by the backup, otherwise, we would execute the same request twice, leading to a possible system inconsistency. This is the rationale for the buffering of requests.

This buffering technique works correctly in a single-client case, because in case (iii) the first request received by the backup necessarily corresponds to the last received `set_state` request (we assume that our clients are single-threaded). The multi-client case is more subtle, and is discussed in the next section.

### 3.2. Multi-client implementation

Some difficulties here are inherent to distributed programming in an asynchronous context, and others arise from limitations of CORBA’s portable interceptors.

As in the single-client case, in case of failure, the backup and the remaining clients must agree on *when* the primary has crashed to avoid inconsistencies. However, unlike the single-client case, the backup can't determine with certainty whether the system is in case (i), (ii) or (iii) of the above classification. First, the backup can never know with certainty that it has received *all* `set_state` requests issued before the crash (in the mono client case this is done by numbering the client's requests and `set_state` info). Secondly, when the backup is asked to become the primary by some client C1, if the last `set_state` info it received corresponds to a request issued by *another* client C2, the backup can never be sure whether C2 has obtained its reply [case (i)] or whether C2 is going to contact it again by re-issuing its request [case(iii)]. In case (i), the backup *must* apply the `set_state` info, or C2 would have seen a future that would have been lost by the crash, resulting in an inconsistency. In case (ii), the backup *can't apply* the `set_state` info, otherwise it won't be able to answer C2 without re-executing its request twice (as in the single-client case), thus also resulting in an inconsistency.

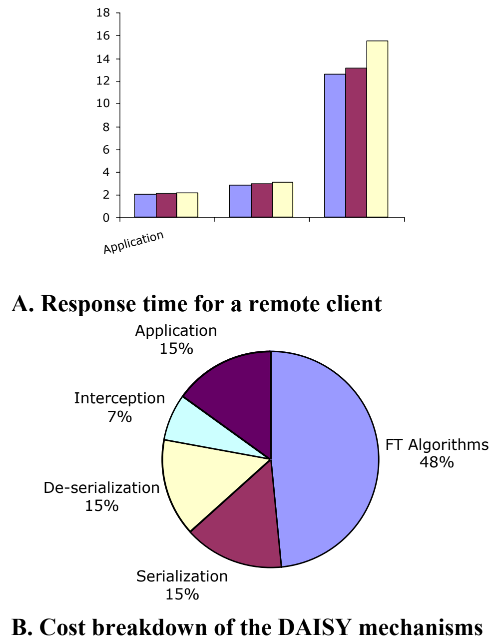
Because we're using TCP/IP connections, which are asynchronous, these problems are demonstrated to have no solution that works in all configurations. To solve the problem, we rely on an implicit synchrony assumption, by assuming that with a very high probability, messages don't take longer than a predefined duration to reach their destination. This means that after a given time window, the reply issued by the primary has been received by the client, or that the client has re-sent the request to the backup. Of course, the use of an atomic broadcast framework would resolve the problems caused by the system’s asynchrony by imposing a total order on all requests. Unfortunately, CORBA portable interceptors don't support this option, because the communication primitives used by the ORB cannot be adapted using the PI mechanisms.

### 3.3. Application

We have used a simple banking application to assess the performance of these fault tolerance mechanisms. The server is a CORBA service that implements the basic operations needed to operate bank accounts: account creation, balance of an account, deposit or withdrawal on an account, transfer between two accounts, etc. The client accesses the bank service through CORBA requests. We have compared the latency of client requests in three different system configurations:

A. The standalone banking application;
B. The banking application together with “transparent” portable interceptors, in order to measure interception overhead;
C. A fault-free execution of the banking application with interceptors implementing the full primary-backup replication mechanism.

Our experimental testbed comprises hosts with 1GHz i686 processors running Linux 2.4, interconnected over 100Mb/s Ethernet. The client runs on a different host from the server, and in the fault tolerant configuration the primary and backup run on the same host. We performed 1000 experiments, each experiment including 1000 invocations (account creation, withdrawal or deposit). We measured the duration of an experiment and obtained the results shown in Figure 3.




**A. Response time for a remote client**



**B. Cost breakdown of the DAISY mechanisms**

**Figure 3. Preliminary performance measurements**

Figure 3.A shows response latency (mean, minimum and maximum values) for configurations A, B and C. The results show that while the use of “empty” interceptors only increases response latency by 15%, the addition of

fault tolerance mechanisms in the interceptors significantly increases overhead. It is worth noting that our application is very simple, and that the relative cost of these mechanisms would be considerably lower for a more realistic application. Figure 3.B shows the breakdown of this overhead per aspect of the mechanism. Except for the interception overhead, which is independent of the nature of the application, most of these costs depend on the size of the checkpointed state. These variable costs include serialization of the primary's state, the transfer of information for replica synchronization, and the deserialization of the state on the backup.

## 4. Discussion

The previous section described the use of Java serialization and CORBA portable interceptors to implement a simple replication mechanism. In this section, we consider the advantages of these basic reflective mechanisms over more conventional implementation approaches, and discuss a number of limitations that they impose.

The JVM Serialization interface provides access to the internal state of Java objects in a portable format. This relieves application developers from the task of implementing this critical aspect of the replication protocol. We use Java serialization to checkpoint application objects on the primary and to apply checkpoints on the backup. In more traditional approaches, the application programmer would have to implement (correct!) methods for saving and restoring the state of the application. However, the state of an application includes several facets, not only the internal state of the application objects, but also some external state that is embedded within the underlying executive layers [13]. Java serialization deals only with the internal state, i.e. the attributes of the object. Handling the other facets calls for more advanced reflective features and requires a clear understanding of the mapping relations between application objects, middleware and operating system data structures.

CORBA portable interceptors enable requests to be intercepted and processed by the fault-tolerance mechanisms. The main benefit is that client-server interactions can be controlled transparently, without requiring any modifications to the application-level code. Portable interceptors can be inserted easily without disturbing the development of application objects and thus can be changed (at load time only) depending on system configuration and environmental conditions. Moreover, as it resides at the ORB level, the interceptor is independent from the operating system and transport layer. Interceptors are also independent of the server and client application, and are therefore portable and reusable.

However, CORBA's Portable Interceptors have a number of important drawbacks.

1. **PIs cannot modify a request's input parameters.**

Implementing certain non-functional mechanisms, such as ciphering client-server communications, would require the ability to modify the parameters of a request, by replacing them with their ciphered counterpart. However, PIs are able to piggyback some information along with the request and are thus able to sign requests.

2. **PIs cannot modify a request's output parameters.**

In our implementation, the backup cannot directly apply the checkpoints it receives to its object. It buffers them and waits in order to know whether the invoker of the request corresponding to the checkpoint has received the reply. When a client has not received its reply, the backup receives the corresponding request but cannot forge a reply; it has to re-process the request in order to produce the reply. The ability to modify output parameters would simplify this mechanism considerably: the backup could cache the reply received from the primary and apply the checkpoint. When receiving the corresponding request from the client, it would send the reply back without having to process the request twice.

3. **PIs must invoke every request.**

The only way to prevent a PI from invoking a request is to throw an exception that will probably be interpreted as an error signal at the client side. Some fault-tolerant mechanisms, such as the leader-follower replication algorithm, cannot be easily implemented with this restriction. In a leader-follower configuration, several replicas receive the requests but only one replies to them. This would necessitate cheating if implemented with PIs.

4. **PIs are not CORBA objects.**

PIs cannot communicate with one another directly. In our implementation, the PI of the primary sends *set_state* requests to the Backup server in order to communicate with the PI of the backup. The backup PI intercepts the invocation, checks what type of request it is (for the server or the PI). If it is for the PI, it will throw an exception in order not to process the request. This trick requires the server to implement an interface that includes an "empty" *set_state* method that is used only by the interceptors. Likewise, the server must implement an empty *ping* method that is used for error detection. These methods have nothing to do with application-level concerns, but limitations of interceptor mechanism means that they become visible to application programmers. This intrusiveness could be eliminated if interceptors were CORBA objects: each interceptor would have an object reference, would be able to implement an interface, and would be able to communicate with other interceptors. The fault tolerance mechanisms would be more transparent for the user and the interceptors would be easier to develop.

5. **PIs don't have their own thread.**

As we mentioned above, each interceptor inherits a concurrency model from its associated CORBA object. Whether the object is single- or multi-threaded, the

interceptor is only activated by the arrival of an incoming request; it cannot implement its own event loop. This means that certain mechanisms, such as periodic "I'm alive" messages, cannot be easily implemented.

**6. PIs cannot reorder requests.**

As discussed in section 3.2, handling multiple clients is very difficult. When the backup PI is switching to the primary mode, it has to wait for the last request processed by the former primary in order to decide whether or not to apply the last checkpoint it received. It might have received a certain number of requests from other clients in the meantime, and has to "freeze" them while waiting for this particular request. This is very difficult with some ORB threading strategies. A solution would be for requests to be first class objects, which means being able to manipulate them as regular objects: passivate, serialize and forward them [12]. This change would simplify the development of complex fault-tolerance strategies, but would require more work at the middleware level when providing information to the interceptor.

## 5. Conclusion

This experimental work shows that simple replication mechanisms can be implemented from basic and standardized interception and serialization mechanisms, namely *CORBA Portable Interceptors* and *Java Serialization*. A set of customized *Portable Interceptors* could be developed to offer a wide range of fault-tolerance strategies that could be selected according to the needs of the application and the infrastructure configuration. It is worth noting that these mechanisms can be provided in a non-intrusive manner, transparently to the application level. However, our work has highlighted a number of limitations in CORBA's interceptors. Some of the issues identified in section 4 could easily be solved by a new generation of portable interceptors, which would greatly facilitate the implementation of replication-based fault tolerance mechanisms. These improvements would be industrially relevant as these standard executive supports are now being used in several application domains where lightweight and non-intrusive implementation of replicated processing for availability has some merits.

We are currently using the DAISY platform to investigate the implementation of extended fault tolerance features, with a focus on wrapping techniques for improved error detection at the communication level, at the middleware level, and at the operating system level. Regarding behavioral and state control at all levels of the system architecture, we are currently investigating multilevel reflection, i.e. reflective principles applied to all system layers, previously presented at DSN'2003 [12].

## 6. References


[1] OMG, "Common Object Request Broker Architecture (CORBA/IIOP) 3.0.2," 2002-12-02, 2002.

[2] Sun, "Java Object Serialization Specification," Sun Microsystems, Technical Report November 1998.

[3] CORBA, "Common Object Request Broker Architecture: Core Specification," ch.23 Fault Tolerant CORBA, OMG formal/02-12-06, December 2002.

[4] R. Baldoni, C. Marchetti and A. Termini, "Active Software Replication throught Three-tier Approach," presented at 21st IEEE Symposium on Reliable Distributed Systems (SRDS2002), Osaka, Japan, 2002.

[5] L. E. Moser, P. M. Melliar-Smith, and P. Narasimhan, "A Fault Tolerant Framework for CORBA," presented at 29th International Symposium on Fault-Tolerant Computing, Madison, Wisconsin, USA, 1999.

[6] R. Friedman and E. Hadad, "A Group Adaptor-Based to CORBA Fault-Tolerance," IEEE distributed systems online, middleware 2001.

[7] J.-C. Ruiz-García, M.-O. Killijian, J.-C. Fabre, and P. Thévenod-Fosse, "Reflective Fault-Tolerant Systems: From Experience to Challenges," IEEE Transactions on Computers, Special Issue on Reliable Distributed Systems, vol. 52, pp. 237-254, 2003.

[8] Pattie Maes, "Concepts and Experiments in Computational Reflection," ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87), pp.147-155, Orlando, Florida, October 1987.

[9] G. Agha, S. Frolund, R. Panwar, and D. Sturman, "A Linguistic Framework for Dynamic Composition of Dependability Protocols," presented at DCCA-3, 1993.

[10] B. Garbinato, R.Guerraoui and K.Mazouni, "Implementation of the GARF Replicated Object Platform," Distributed System Engineering Journal, vol2, pp 14-27, 1995.

[11] S. Chiba, "A Metaobject Protocol for C++," presented at ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA'95), Austin, Texas, USA, 1995.

[12] F. Taiani, J.-C. Fabre, and M.-O. Killijian, "Towards Implementing Multi-Layer Reflection for Fault-Tolerance," presented at DSN'2003, The International Conference on Dependable Systems and Networks, San Francisco, CA, USA, 2003.

[13] M.-O. Killijian, J.-C. Ruiz-Garcia, and J.-C. Fabre, "Portable serialization of CORBA objects: a reflective approach," presented at 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2002, Seattle, Washington, USA, 2002.