

Handling FT-CORBA Compliant Interoperable Object Group References

R. Baldoni, C. Marchetti, R. Panella, and L. Verde
Dipartimento di Informatica e Sistemistica
Università di Roma “La Sapienza”
Via Salaria 113, 00198, Roma, Italy
E.mail: {baldoni, marchet, verde}@dis.uniroma1.it

Abstract

The handling of Interoperable Object Group References (IOGRs) is one of the basic building block of any FT-CORBA compliant platform. In this paper we introduce two basic components: IOGRManager, a tool to create, update, browse IOGRs, and an Object-Request-GateWay (ORGW), a CORBA compliant client request interceptor, which allows clients running on non-FT-Corba compliant ORBs to transparently interact with replicated objects residing on FT-CORBA-compliant platforms. Both components have been developed in the context of Interoperable Replication Logic (IRL) project which investigates the impact of three tiers architectures on software replication.

1 Introduction

Commonly market-available middlewares basically offer only the possibility of transparent remote method invocations of objects deployed over a distributed systems. This is usually done without guaranteeing anything about the quality of service (QoS). Such middlewares relay QoS management tasks to the underlying platforms or to the application itself, i.e. *out of the middleware QoS*. An emerging trend in the middleware area is thus the possibility of increasing the functionality provided by a middleware with QoS related middleware extensions, in order to allow programmers to exploit the same QoS-APIs on different platforms.

CORBA ([14]) is a distributed object computing (DOC) message passing middleware based on a standard architecture that allows programmers to create and access objects deployed over a distributed systems. CORBA achieves full interoperability in heterogeneous environments by providing location, platform and language transparency. In CORBA, the common middleware tasks (e.g. object location, request marshaling, message transmission, message

unmarshaling etc.) are undertaken by the Object Request Broker (ORB) component.

In this context, the Object Management Group has recently released Fault Tolerant CORBA (FT-CORBA, [13, 12]) specification, with the aim of simplifying the development of highly-available, fault-tolerant, distributed applications. One of the main limitations of release 1.0 of FT-CORBA is the so-called *vendor-dependency* limitation. In particular, in order to simplify the development of FT-CORBA compliant infrastructures, it is stated in the specification that such infrastructures should be non-interoperable, i.e. all the replicas belonging to a given fault tolerance domain should run over ORBs provided by the same vendor. This neutralizes one of the main advantages of CORBA technology, i.e. the possibility of run-time interactions among objects deployed on different platforms running over ORBs provided by different vendors.

A lot of work is going on in order to implement the FT-CORBA specification (or a portion of it). At the best of our knowledge, two existing systems are FT-CORBA compliant: *Eternal* [9] and *DOORS* [10]. While the *Eternal* system has been used as proof of concept during the standardization effort – and is non-interoperable – the *DOORS* system is still under development, and only implements a subset of the specification.

In this paper we present some experiences done in the context of Interoperable Replication Logic (IRL) project [18]. The mission of IRL is to study the impact of three tiers architectures on software replication [1]. One of the main IRL's targets is to develop an interoperable and portable implementation of the FT-CORBA specification [2, 6, 8]. In such a context, IRL components acts as a midtier between the distributed application and ORBs from distinct vendors running on different platforms¹.

¹Actually, the entire IRL development is being carried out on three different CORBA compliant platforms, i.e. JacORB [19] and IONA's [17] Orbacus and Orbix 2000.

More specifically, in this paper we present two IRL building blocks:

- IOGRManager object which allows to create and update FT-CORBA compliant interoperable object group references (IOGRs);
- ORGW (Object-Request-GateWay) which allows clients running on *non-FT-Corba compliant* ORBs (implementing Portable Interceptors) to transparently interact with replicated objects residing on IRL platform.

The latter component actually emulates the behaviour of a FT-CORBA compliant client ORB without impacting on the ORB code. By using such components, an administrative client is allowed to create object groups and to publish their references, while other clients can then interact with the object group as if it was a singleton highly available object.

The remainder of the paper is organized as follows: Section 2 describes IOGRs structure and how they must be managed by FT-CORBA client ORBs, Section 3 deals with the CORBA object we deploy to create and modify IOGRs, Section 4 introduces Portable Interceptors and the way we use them to implement the Outgoing Request Gateway. Finally, Section 5 concludes the paper.

2 Fault Tolerant CORBA Basic Mechanisms

In the FT-CORBA specification, fault tolerance is achieved through entity redundancy, fault detection and recovery. The replicated entity is the CORBA object, and replicated objects are managed through the abstraction of object group in order to guarantee strong consistency. To identify an object group, FT-CORBA introduces *interoperable object group references* (IOGRs). An IOGR is a particular type of an Interoperable Object Reference (IOR, [14]). IOGRs are used by FT-CORBA compliant client ORBs in order to provide clients with replication and failure transparency by the *transparent client reinvocation* and *redirection* mechanisms.

2.1 Interoperable Object Group References

An IOGR is a particular kind of multiprofile IOR with the additional capability of handling the notion of object group. A multiprofile IOR is an IOR composed by more than one profile. Each *profile* identifies a connection endpoint containing the information needed by the ORB to reach a CORBA object implementation (transport protocol, host address, etc.). However, a multiprofile IOR does not handle, for example, membership changes within an object group, object group versioning and primary identification within an object group which follows a primary-backup

replication style. IOGR are thus introduced in CORBA specification to cope with such issues.

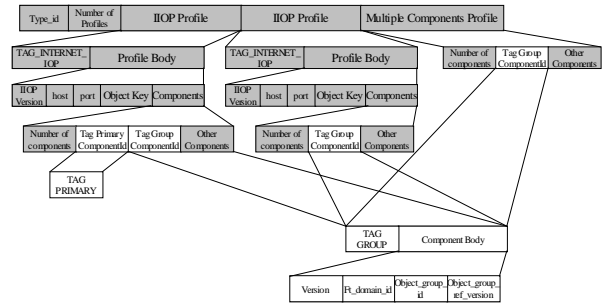


Figure 1. IOGR Structure

A generic IOGR is represented in Figure 1 where the components added by the FT-CORBA specification are within white rectangles. An IOGR contains multiple TAG_INTERNET_IOP profiles. Any of these profiles can be used by client ORBs to reach the server object group. The reference also contains a TAG_MULTIPLE_COMPONENTS profile. A TAG_MULTIPLE_COMPONENTS profile must be present in an IOGR when it identifies an empty group, i.e. with no replicas currently belonging to the group. Each profile must contain the TAG_FT_GROUP component, while the TAG_FT_PRIMARY component can be contained in at most one TAG_INTERNET_IOP profile. The TAG_FT_GROUP component is a struct composed by four fields, namely: `version`, `ft_domain_id`, `object_group_id` and `object_group_ref_version`. These fields allow to uniquely identify an object group in the context of a fault tolerance domain and to associate version numbers to membership changes. The TAG_FT_PRIMARY component of a IOP profile, if present, informs to the client ORB that the containing IOP profile probably is the primary member of a passively replicated object group. In the following section we describe how requests has to be performed by clients once they have obtained a non-empty IOGR, e.g. by invoking the CORBA Naming or Trading services.

2.2 Transparent Client Reinvocation and Redirection

A FT-CORBA client accessing an object group, i.e. a CORBA client running on a FT-CORBA compliant client ORB and invoking an object group deployed in a FT-infrastructure, behaves differently from a classical CORBA client. This is mainly due to the fact that (i) a client of a replicated object has to be masked as much as possible from failures of object group members or of the network

and (ii) it has to uniquely identify its requests in order to allow duplicate filtering upon request reinvocation. To deal with such issues, in the FT-CORBA standard two mechanisms are identified, i.e. *transparent client reinvocation* and *transparent client redirection*.

Transparent Client Reinvocation. The transparent client reinvocation mechanism provides clients with replication and failure transparency exploiting two sub-mechanisms that we define *transparent client failover* and *transparent request identification*.

The *transparent client failover* mechanism mandates a FT-CORBA compliant client ORB, invoking an object group identified by an IOGR, to try exploiting all of the profiles contained in the object reference before returning an exception to the client. More precisely, a FT-CORBA compliant client ORB must not abandon an invocation (throwing an exception to the client) until (i) it has tried to invoke the replicated servers by exploiting *all* of the TAG_INTERNET_IOP profiles contained in the IOGR while receiving only exceptions falling into the so called *failover conditions*, or (ii) it has received an exception that does not fall among such failover conditions, or (iii) the request expiration time has elapsed (see below), or (iv) it has received the reply. Failover conditions are the circumstances under which an ORB is allowed to reissue a request using a (possibly different) profile contained in an IOGR.

	Completion Status	CORBA System Exception
Without Transparent Reinvocation	COMPLETED_NO	COMM_FAILURE TRANSIENT NO_RESPONSE OBJ_ADAPTER
With Transparent Reinvocation	COMPLETED_NO COMPLETED_MAYBE	COMM_FAILURE TRANSIENT NO_RESPONSE OBJ_ADAPTER

Figure 2. Failover Conditions with and without Transparent Reinvocation

Figure 2 illustrates the CORBA exceptions signaling failover conditions for both: ORBs implementing transparent reinvocation and ORBs that do not (i.e. ORBs that support multiprofile IORs but do not support IOGRs). Such conditions define when a (FT-CORBA compliant) client ORB is allowed to reissue a request. Note that a FT-CORBA compliant client ORB is allowed to reissue requests even if the returned request completion status is COMPLETED_MAYBE. In this case a server could actually have executed the request. Therefore, to avoid repeated execution of reissued requests, FT-CORBA defines

how a client ORB must uniquely identify a request to allow servers to automatically perform duplicate filtering, and this is what we have defined as *transparent request identification*. Request identification is done in FT-CORBA by using the REQUEST service context. This service context is put in each request and contains three fields, namely: `client_id`, `retention_id` and `expiration_time`. The pair composed by `client_id` and `retention_id` is a unique request identifier that allows the server ORB to identify repetitions. If the request is a repetition, the server does not reexecute the request but rather returns the reply that was generated by the prior execution. The `expiration_time` is the basic building block of a garbage collection mechanism. It provides to servers a lower bound on the time until which they must retain the request information and the corresponding reply (if any). The REQUEST context's `expiration_time` field is determined by the client ORB by adding the local clock value to the value of the `request_duration_policy_value`, that is the mean by which a client communicates to the ORB its estimate about the duration of an invocation. When a client ORB starts issuing a request, it sets a timeout of duration equal to the `request_duration_policy_value`. When this timeout expires and the reply has not been received, the client ORB throws a NO_RESPONSE CORBA exception to the client.

Transparent Client Redirection. IOGRs held by clients may become obsolete: a membership change can occur asynchronously with respect to the client invocations. For this reason, FT-CORBA defines how to propagate membership changes without having the client resolve the object group reference each time it has to perform an invocation². The solution to this problem adopted by FT-CORBA is based on the assumptions that object group members rapidly track membership changes and always have updated references. Thus, to provide clients with the most updated IOGRs, a server needs to know which is the current version of the IOGR held by the client. For this reason, a FT_GROUP_VERSION service context is defined. It contains a single field `object_group_ref_version` in which the client ORB copies the `object_group_ref_version` field value of the IOGR's TAG_FT_GROUP component. In this way, when a group member server ORB receives a request, it can determine whether the client reference is obsolete or not. In the first case, it throws a LOCATION_FORWARD_PERM exception to the client ORB that, when received by the client ORB, has the effect of permanently updating its

²Note that also name resolution is asynchronous with respect to membership changes. Therefore this is not a solution to the problem of providing clients with most updated references. It is only a workaround to try providing clients with *fresh* references.

reference. In the second case, the request is normally executed and a reply is returned to the client.

It is clear that even when considering the "basic" FT-CORBA mechanisms discussed above, a set of core problems have to be addressed. In particular, in the next sections, we address the following issues:

- IOGR creation, updating and modification;
- Implementing transparent client reinvocation and redirection upon a non FT-CORBA compliant ORB;

While the first problem solution represents a basic building block of a fault tolerance infrastructure, the second problem has to be considered in the wider context of building an interoperable FT-CORBA compliant infrastructure, that is the aim of the IRL project.

3 Creating and Managing IOGRs: the IOGRManager CORBA Object

In this section we describe the object implemented in IRL to allow CORBA clients to build, update and manage IOGRs. Figure 3 shows the IDL interface of the IOGRManager CORBA object.

Note that in the IOGRManager interface we use the ObjectGroup type for passing parameters as well as to return results, i.e. object group references. This type is defined as the Object CORBA type. In this way, an IOGRManager client can handle object group references exactly as it handles generic object references. We preferred to define the ObjectGroup type to improve the IDL interface readability. The semantic of the operations provided by the IOGRManager object is described below:

create_IOGR: allows to create IOGRs. It takes as input parameter the ids of the object group and of the fault tolerance domain, the object group version and an ObjectList object, which contains a (possibly empty) list of references. If the list is empty, a simple IOGR containing a multiple components profile is returned (see Section 2.1). If the list is not empty, the returned reference will include all the IIOP profiles of the references belonging to the object_list, modified as explained in Section 2.1. In any case, the returned object group reference has no profile containing a TAG_FT_PRIMARY component. The primary member of an object group working in primary backup style can be set later using the set_Primary_Member operation, described below.

extract_object_references: allows to extract from an object group reference, passed as input parameter, a (possibly empty) list of object references, one for each member of the group.

```
typedef Object ObjectGroup;
typedef sequence<Object> ObjectList;

interface IOGRManager {

    ObjectGroup create_IOGR(
        in FT::FTDomainId ft_domain_id,
        in FT::ObjectGroupId object_group_id,
        in FT::ObjectGroupRefVersion
            object_group_ref_version,
        in ObjectList object_list)
        raises (IOGRManagerException);

    ObjectList extract_object_references(
        in ObjectGroup object_group)
        raises (IOGRManagerException);

    ObjectGroup add_Member(
        in ObjectGroup object_group,
        in Object object)
        raises (IOGRManagerException);

    ObjectGroup remove_Member(
        in ObjectGroup object_group,
        in Object object)
        raises (IOGRManagerException);

    Object get_Primary_Member(
        in ObjectGroup object_group)
        raises (IOGRManagerException);

    ObjectGroup set_Primary_Member(
        in ObjectGroup object_group,
        in Object primary_object)
        raises (IOGRManagerException);

    ObjectGroupRefVersion get_Group_Version(
        in ObjectGroup object_group)
        raises (IOGRManagerException);

    void set_Group_Version(
        in ObjectGroup object_group,
        in FT::ObjectGroupRefVersion version)
        raises (IOGRManagerException);

    boolean is_ObjectGroup (in Object object);

    boolean is_Equivalent(
        Object object_a,
        Object object_b)
        raises (IOGRManagerException);
};
```

Figure 3. The IOGRManager IDL Module

add_Member /remove_Member: allows to insert/remove a member into/from an object group reference. It extracts the IIOP profile from the object input parameter, and inserts/remove it into/from the object_group reference, that is finally returned as result.

get_Primary_Member/set_Primary_Member: accepts as input parameter an object group reference and returns/sets the reference of the IIOP profile containing the TAG_FT_PRIMARY component (if any, see Section 2.1).

get_Group_Version/set_Group_Version: returns/sets the version of the object group reference passed as input parameter. i.e. the value of the Object_Group_Ref_Version field of the TAG_FT_GROUP component of a profile of the object group reference.

is_ObjectGroup: can be used to check if an object reference is an object group reference. i.e. if the object reference contains in each profile the TAG_FT_GROUP component.

is_Equivalent: given two object references, this operation returns true if the two object references are object groups and the two pairs (ft_domain_id, object_group_id) of the two object group references are identical.

Each of the operations described above throws exceptions to notify clients about the occurrence of some anomalous condition. This is the case when, for example, an IOGRManager client tries to get the primary object of an object group when this primary has not been set, or when it tries to remove a member not belonging to a group or from an empty group.

Concerning the implementation of the IOGRManager object, we point out that it is mainly based on the syntactical manipulation of serialized references. The operation described above are actually based on the following pattern: (i) the references passed as input parameters are converted into serialized object references (by the orb.object_to_string() method, (ii) some syntactical manipulation/inspection takes place, (iii) if a serialized IOR is the result of the operation, it is first converted into an ObjectGroup type reference (by the orb.string_to_object() method) and then returned to the caller. In this way, clients are not involved in any IOR manipulation and can handle object group references exactly as they handle object reference. Note also that IOGRManager has been designed as a completely stateless component, in order to simplify its replication and to get high-availability.

4 Using IOGRs: the ORGW Component

The next logical step after having created IOGRs is letting them be used by clients of replicated objects addressed by IOGRs. In particular in this section we describe how a non FT-CORBA client ORB implementing Portable Interceptors [11] can be extended to get transparent client redirection and redirection mechanisms. Under a practical point of view this means that a CORBA client can interact with replicated objects benefiting of replication and failure transparency even though it runs over a non FT-CORBA compliant ORB. This could leverage the use of “legacy” client ORBs, without requiring to CORBA clients to update their ORB version to FT-CORBA compliant ones, provided that such “legacy” ORBs implement portable interceptors.

Before of describing such an extension we briefly describe Portable Interceptors. Interested readers can refer to [4, 7, 11] for additional details.

4.1 Portable interceptors

CORBA Portable Request Interceptors (PIs, [11]) are a mechanism allowing to modify the ORB and the application behaviour upon the event of sending or receiving a message (e.g. a request, a reply or an exception) without impacting either on the ORB code or on the application one. Request PIs are logically set between the ORB and the application layers (Figure 4) and can be installed in an ORB by invoking their interfaces. Request Interceptors are classified in *client request interceptors* and *server request interceptors*. The former are installed in client-side ORBs and can intercept outgoing requests and contexts as well as incoming replies and exceptions. Conversely, the latter are installed in server-side ORBs and can intercept incoming requests and contexts as well as outgoing replies and exceptions. PIs can perform operations at different points during request processing. Figure 4 shows such *interception points*.

After their installation, PIs intercept *all* the requests or replies exchanged between a client and a server ORB. Different interceptor instances can be registered within a single ORB and, once a message is intercepted, all the registered interceptor instances will be invoked by the ORB upon the arrival of a triggering event. The invocation order is ORB implementation dependent and cannot be either inspected or modified³.

By implementing the methods reported in Figure 4, request interceptors can be configured to:

- access request or reply information (this is a strictly platform dependent issue) to perform some action;
- redirect a request to another target by throwing a LOCATION_FORWARD exception to the client ORB⁴;
- throw other CORBA exceptions;
- manipulate the request service context, e.g. to piggyback additional information onto a message;
- perform their own invocations;
- delay a request or a reply.

Providing such features, PIs actually become a powerful development tool. Without impacting on application code, they can be used, for instance, to piggyback authentication

³Actually, some ORB implementations allow to chain interceptors in a customizable fashion, e.g. by defining their invocation order. However, assumptions on the order of invocation of multiple interceptor result in non-portable interception code.

⁴After a LOCATION_FORWARD exception has been thrown by an interceptor, no other registered interceptor is invoked by the ORB.

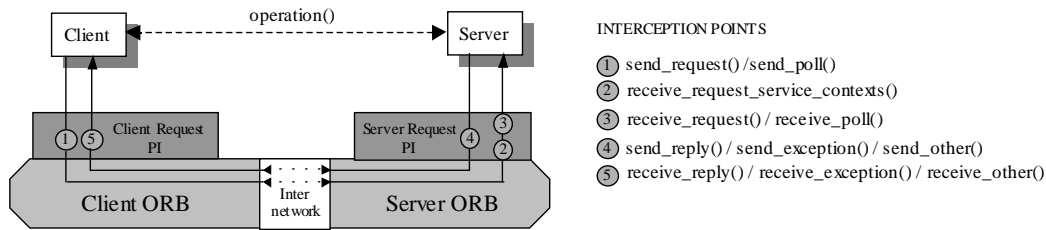


Figure 4. Client and server Portable Request Interceptors

information into GIOP⁵ messages flowing between a client and a server, to uniquely identify client requests and redirect them among different replicas of a fault-tolerant server [6, 8], to share the load among different copies of a server, to implement caching mechanisms [3] or to implement flow control, as also shown in [4].

However, in some circumstances, PIs do not suffice to meet the application requirements. In particular, PI limitations can be summarized as follows:

- client request PIs cannot generate their own replies to intercepted requests;
- PIs can definitively block a request or a reply only by raising an exception;
- PIs can redirect a request only by throwing a `LOCATION_FORWARD` exception;
- PIs cannot alter none of the parameters of a request or of a reply;
- PIs cannot modify the request service contexts (but they can add their own contexts);

Moreover, in Java implementations, because of the Java Portable Binding, an interceptor cannot access the `RequestInfo` interface to read the following attributes: `arguments`, `exceptions`, `contexts`, `operation.context`, `result`. Roughly speaking, a Java PI cannot access more than the `operation` name of the `RequestInfo` interface fields concerning the signature of an operation. If a PI tries to do so, a `NO_RESOURCES` system exception is thrown. To overcome this limitations along with the impossibility of creating replies, a design pattern using proxy [4, 5] can be adopted⁶.

⁵GIOP is the acronym of General Inter-ORB Protocol, i.e. the specification of the abstract protocol allowing CORBA remote object interoperability. This general specification is instantiated over a specific transport layer, e.g. the Internet Inter-ORB Protocol is the GIOP instance over the TCP/IP transport layer.

⁶Note that the proxy pattern also increases the flexibility of the solution by decoupling the request redirection aspects from the ones dependent from the application.

4.2 The ORGW Component

The Outgoing Request GateWay (ORGW) component allows a CORBA client running on a CORBA 2.4 compliant ORB implementing portable interceptors to interact with an object group by exploiting the transparent client redirection and reinvocation mechanisms described in Section 2.2.

The component is intended to entirely substitute a generic client ORB with an extended ORB. The extended ORB is standard CORBA ORB within which a particular CORBA compliant portable request interceptor has been registered. To enhance client transparency, a CORBA client initializes the ORGW component as it initializes the ORB. In particular, a Java CORBA client wanting to exploit the ORGW functionality has to:

1. import the package `irl.orgw`;
2. initialize the ORGW component by invoking the `org.omg.CORBA.ORB orgw_init()` method. This method takes as input parameters the same parameters of the `org.omg.CORBA.ORB.init()`, and a `int invocationSemanticType` parameter introduced to implement additional client invocation semantics. Currently, the only admitted value for the `invocationSemanticType` parameter is `_FT_CORBA`. As for the `org.omg.CORBA.ORB.init()` method, the `org.omg.CORBA.ORB orgw_init()` method returns a reference to an ORB.

In legacy applications a client initializing the ORB has simply to substitute the `orb = org.omg.CORBA.ORB.init(args,prop)` instruction with the `orb = orgw_init(args,prop, _FT_CORBA)` one. In this way, it will be able to interact with a replicated object as the FT-CORBA specification mandates.

In the following, we first describe how the ORGW component accomplishes the task of implementing transparent client reinvocation and redirection and then its limitation concerning the request expiration time timeout.

ORGW Implementation As mentioned before, the ORGW component logically extends an ORB with the functionality of a custom client request portable interceptor, i.e. the `ORGWInterceptor`.

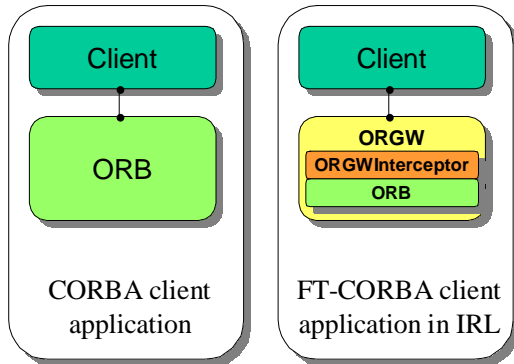


Figure 5. Common CORBA Client Compared with a FT-CORBA Compliant Client

Figure 5 compares a standard CORBA client application with one using the ORGW component, and shows the relationships among the client, the ORGW, the ORB and the `ORGWInterceptor` component.

The `ORGWInterceptor` is the core of the ORGW. `ORGWInterceptor` implements the following actions:

- **request interception and filtering:** each time a client issues a request, the `ORGWInterceptor` checks if the request is destined to an object group or not. In the latter case, the `ORGWInterceptor` let the request pass through it without interfere, i.e. the request is normally handled by the ORB contained in the ORGW. If the request is destined to an object group, the `ORGWInterceptor` executes the following steps.
- **request completion:** a request destined to object groups is augmented with service contexts. These are the `REQUEST` and the `FT_GROUP_VERSION` service contexts described in Section 2.2, used to implement, respectively, the transparent request identification and redirection mechanisms.
- **request redirection and reinvocation:** after request's completion, the `ORGWInterceptor` extracts the first IIOP profile from the object group reference (if any), converts it into an object reference and uses this reference to throw a `LOCATION_FORWARD_PERM` exception to the client ORB. The ORB then transparently reissues the request to the object referred to by the reference contained in the exception message. The request reissued by the ORB will pass again through the `ORGWInterceptor`. Recursive scenarios are

avoided in this case very easily: being the new request issued towards an object and not towards an object group, the interceptor let it pass through without interfering. After the ORB has reissued the request, one of the following events can happen:

1. either the invoked replica throws a `LOCATION_FORWARD_PERM` exception containing an updated object group reference or,
2. the invoked replica (or the client ORB) throws a CORBA exception falling into the *failover conditions* described in Section 2.2 or,
3. the invoked replica (or the client ORB) throws a CORBA exception that does not fall into the *failover conditions* described in Section 2.2 or,
4. the invoked replica returns a reply or a user exception.

`ORGWInterceptor` listens for the occurrence of one of the previous four events by implementing the PI's interface `receive_other()` and `receive_reply()` methods and reacts as follows:

- case (1).** the interceptor updates its internal state and throws a `LOCATION_FORWARD_PERM` to the server ORB, provoking a reinvocation that uses the new object group reference obtained by the previously invoked server;
- case (2).** the interceptor checks if it has already exploited all of the profiles contained in the reference. In the affirmative, it definitively stops the invocation by throwing a `NO_RESPONSE` CORBA exception to the client. Otherwise it incrementally selects another profile and uses it to throw a `LOCATION_FORWARD_PERM` exception to the client ORB, so that the request is issued again towards a different replica;
- cases (3) and (4).** the reply or the exception is returned unmodified to the client.

In its current implementation status, `ORGWInterceptor` suffers of a limitation due to the request timeout expiration. More specifically, in Section 2.2 we pointed out that a FT-CORBA compliant client ORB must throw a CORBA exception if a request is not served after its `expiration_time` has elapsed. A standard CORBA ORB, however, does not allow to set timeouts on request invocations, unless implementing the CORBA Messaging Specification [15] or the Real Time CORBA Specification [16]. Moreover, such a timeout is not implementable inside our ORGW being portable interceptors designed as pure stateless components. As a consequence our current ORGW implementation does not throw exceptions to the client after request

duration time expiration, even though it inserts a client configurable request duration times into the REQUEST service contexts to enable server-side garbage collection of outdated requests (see Section 2.2)⁷.

5 Conclusions

The race to develop software components for FT-CORBA compliant platforms is just started. In this paper we presented two components to handle and to use IOGRs: the IOGRManager and the ORGW. IOGRManager is actually a tool for building and managing FT-CORBA compliant IOGRs. This tool could be useful for anyone that is going to implement a FT-CORBA compliant Infrastructure. ORGW is a CORBA compliant client request interceptor that implements the FT-CORBA client invocation semantic without requiring ORB modifications. It allows a standard CORBA client running over a CORBA ORB implementing portable interceptors to benefit of the transparent client redirection and reinvocation mechanisms.

The IOGRManager CORBA object is available for download at the IRL Project web site [18]. The ORGW will be also available soon at the same address.

References

- [1] R. Baldoni and C. Marchetti, *Software Replication in Three tiers architectures: is it a real challenge?*, Proceedings of the IEEE International Workshop on Future Trends on Distributed Computing Systems (FTDSC2001) (Bologna, Italy), October 2001, to appear.
- [2] R. Baldoni, C. Marchetti, A. Virgillito, and F. Zito, *An Interoperable Replication Logic for CORBA Systems*, Proceedings of the 6th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS01) (Roma, Italy), January 2001, pp. 198–205.
- [3] G. Chockler, D. Dolev, R. Friedman, and R. Vitenberg, *CASCADE: Caching Service for Corba Distributed objEcts*, Proceedings of the 2nd IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware 2000) (New York, USA), April 2000, pp. 1–23.
- [4] R. Friedman and E. Hadad, *Client-side Enhancements using Portable Interceptors*, Proceedings of the 6th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS01) (Rome, Italy), January 2001, pp. 191–197.
- [5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: Elements of reusable object-oriented software*, Addison Wesley, 1994.
- [6] C. Marchetti, M. Mecella, A. Virgillito, and R. Baldoni, *An Interoperable Replication Logic for CORBA Systems*, Proceedings of the 2nd International Symposium on Distributed Objects and Applications (Antwerpen, Belgium), September 2000, pp. 7–16.
- [7] C. Marchetti, L. Verde, and R. Baldoni, *CORBA Request Portable Interceptors: a Performance Analysis*, Proceedings of the 3rd International Symposium on Distributed Objects and Applications (Rome, Italy), September 2001, pp. to appear, available on-line at: www.dis.uniroma1.it/~irl.
- [8] C. Marchetti, A. Virgillito, M. Mecella, and R. Baldoni, *Integrating Autonomous Enterprise Systems through Dependable CORBA Objects*, Proceedings of the 5th International Symposium on Autonomous Decentralized Systems (ISADS01) (Dallas, Texas, USA), March 2001, pp. 204–211.
- [9] L.E. Moser, P.M. Melliar-Smith, P. Narasimhan, L.A. Tewksbury, and V. Kalogeraki, *The Eternal System: an Architecture for Enterprise Applications*, Proceedings of the 3rd International Enterprise Distributed Object Computing Conference (EDOC'99) (Mannheim, Germany), July 1999, pp. 214–222.
- [10] B. Natarajan, A. Gokhale, S. Yajnik, and D.C. Schmidt, *DOORS: Towards High-Performance Fault Tolerant CORBA*, Proceedings of the 2nd International Symposium on Distributed Objects and Applications (Antwerpen, Belgium), September 2000, pp. 39–48.
- [11] Object Management Group (OMG), Framingham, MA, USA, *Portable Interceptor Specification*, OMG Document orbos ed., December 1999, OMG Final Adopted Specification.
- [12] Object Management Group (OMG), Framingham, MA, USA, *Fault Tolerant CORBA Finalization Task Force (FTF) Report*, OMG Document formal ed., December 2000, OMG Finalization Report.
- [13] Object Management Group (OMG), Framingham, MA, USA, *Fault Tolerant CORBA Specification, V1.0*, OMG Document ptc/2000-12-06 ed., April 2000, OMG Final Adopted Specification.
- [14] Object Management Group (OMG), Framingham, MA, USA, *The Common Object Request Broker Architecture and Specifications. Revision 2.4.2*, OMG Document formal ed., February 2001, OMG Final Adopted Specification.
- [15] Object Management Group (OMG), *CORBA Messaging, in the Common Object Request Broker Architecture and Specifications. Revision 2.4.2*, OMG Document formal ed., ch. 22, Framingham, MA, USA, February 2001, OMG Final Adopted Specification.
- [16] ———, *Real Time CORBA, in the Common Object Request Broker Architecture and Specifications. Revision 2.4.2*, OMG Document formal ed., ch. 24, Framingham, MA, USA, February 2001, OMG Final Adopted Specification.
- [17] IONA Web Site, <http://www.iona.com>.
- [18] IRL Project Web Site, <http://www.dis.uniroma1.it/~irl>.
- [19] JacORB Web Site, <http://www.jacorb.org>.

⁷Note that Orbacus and Orbix 2000 include proprietary policies to define maximum request duration time. By exploiting such features, ORGWInterceptor can be notified by the ORB after a request expiration timeout has elapsed.