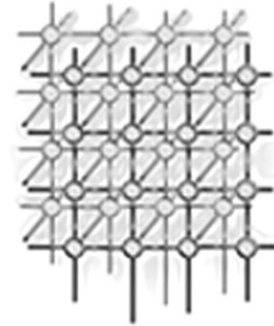


CORBA request portable interceptors: analysis and applications[‡]



R. Baldoni, C. Marchetti^{*,†} and L. Verde

*Dipartimento di Informatica e Sistemistica, Università di Roma 'La Sapienza',
Via Salaria 113, 00198, Roma, Italy*

SUMMARY

Interceptors are an emerging middleware technology enabling the addition of specific network-oriented capabilities to distributed applications. By exploiting interceptors, developers can register code within interception points, extending the basic middleware mechanisms with specific functionality, e.g. authentication, flow control, caching, etc. Notably, these extensions can be achieved *without modifying* either the application or the middleware code.

In this paper we report the results of our experiences with CORBA request portable interceptors. In particular, we point out (i) the basic mechanisms implementable by these interceptors, i.e. request redirection and piggybacking and (ii) we analyze their limitations. We then propose a proxy-based technique to overcome the interceptors' limitations. Successively, we present a performance analysis carried out on three Java-CORBA platforms currently implementing the portable interceptors specification. Finally, we conclude our work with a case study in which portable interceptors are used to implement the fault-tolerant CORBA client invocation semantic without impacting on the client application code and on the CORBA ORB. We also release fragments of Java code for implementing the described techniques. Copyright © 2003 John Wiley & Sons, Ltd.

KEY WORDS: interceptors; CORBA; fault-tolerant CORBA; middleware; performance analysis; distributed object computing

1. INTRODUCTION

In the last 20 years there has been a move away from software and hardware custom-made systems to 'commercial off-the-shelf' (COTS) systems. Generally speaking, a COTS system is something

*Correspondence to: C. Marchetti, Dipartimento di Informatica e Sistemistica, Università di Roma 'La Sapienza', Via Salaria 113, 00198, Roma, Italy.

[†]E-mail: marchet@dis.uniroma1.it

[‡]A preliminary version of this paper has been accepted for presentation at the Third International Symposium of Distributed Object Applications (DOA'01), September 2001, Rome, Italy.

Contract/grant sponsor: Alenia Marconi Systems
Contract/grant sponsor: MURST project 'DAQUINCIS'
Contract/grant sponsor: EU IST project MIDAS



which is 'catalog orderable' rather than custom made. In particular the early 1990s saw the take-off of software implemented on off-the shelf hardware. This solution was motivated by the economic reasons of needing to reduce the cost of system development. More recently, the notion of COTS has also been proposed for software. Using COTS software components reduces both the cost of components and the risk of integration activity while increasing the modularity of the system. COTS systems also gave a boost to the passage from two-tier distributed system architectures to three-tier architectures. In the past, distributed systems were built as two tiers: client and servers. Clients embedded the presentation part, servers embedded processing: the application logic was usually split between the two entities. In a three-tier architecture the client code has to occupy little space (possibly only the presentation code of an application should be installed in order to run as many devices as possible), the middle tier embeds the application logic possibly without containing any part of the application state which should be completely managed by *independent* servers accessed by means of standard interfaces.

In the classic three-tier design pattern, the tiers communicate by means of middleware. Middleware is a software layer located between the application and the operating system with the objective of simplifying the development of distributed applications by providing facilities such as location-transparent method invocation on remote objects despite distributed-system heterogeneity. Examples of middleware platforms include DCE [1] and DCOM [2]. Middleware undertakes the burden of dealing with low-level networking issues, data representations, marshalling and unmarshalling, naming and other similar problems, letting the programmer concentrate on the essential part of the development, without worrying about the incidental part. As a consequence, middleware platforms have had a deep impact on the design of current distributed applications.

Interceptors are hooks exported by a middleware platform in which users can register their code. By exploiting interceptors it is possible to extend the basic middleware functionality without modifying the existing code of the application and the middleware core infrastructure. Therefore, interceptors exhibit several useful advantages. First, by plugging in extensions to an existing middleware, one enjoys the benefit of an economy of scale. That is, core middleware infrastructure does not embed extensions that will be used only by a specific set of applications. Second, interceptors from various implementors and vendors can be plugged in the middleware. As described in [3], interceptors can help the application manage specific network-oriented tasks such as authentication, caching, load balancing and flow control to name just a few.

In this paper we focus on CORBA request portable interceptors [4]. CORBA [5] is a distributed object computing (DOC) middleware based on a standard architecture that allows programmers to create and access distributed objects. 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 marshalling, message transmission, message unmarshalling etc.) are undertaken by the object request broker (ORB) component. The basic idea of CORBA portable interceptors is thus to insert into the CORBA ORB some interception points where the developer can register some code. This code is automatically and transparently executed by the ORB upon the occurrence of relevant events such as the sending of a request or the receipt of a reply. The portable interceptors specification defines two types of interceptor, namely *request interceptors* and *interoperable object reference (IOR) interceptors*. Request interceptors can be used to modify the standard ORB behaviour upon the event of sending or receiving a request, a reply or an exception (e.g. to perform request redirection, piggybacking etc.). Similarly, IOR interceptors are



hooks into the ORB allowing modification of an IOR, the CORBA object identifier, at its creation time without impacting on the application code.

The aim of this paper is to help the reader to understand CORBA portable request interceptor capabilities, limitations, performances and applications and, in particular, the following characteristics.

- We identify the basic mechanisms implementable by portable request interceptors, namely request redirection and piggybacking. For each mechanism, we provide the Java interception code registered within the ORB.
- We point out the limitations of portable request interceptors, e.g. the impossibility of generating replies to client requests and, in Java ORB implementations, also of reading various request parameters.
- We analyze the implementation of a proxy-based technique that overcomes the limitations of portable interceptors.
- We evaluate the cost of the basic interception mechanisms and of the proxy-based technique while considering several design and deployment choices for the proxy component. This evaluation study proposes several benchmarks obtained by running experiments on three different Java ORB implementations, i.e. on the non-commercial JacORB [6], on Orbacus [7] ORB as well as on the commercial product Orbix 2000 for Java [8].
- We propose an application of CORBA request portable interceptors, i.e. the implementation of the fault-tolerant CORBA (FT-CORBA) [9] client invocation semantic by means of a client-side portable request interceptor. FT-CORBA mandates relevant client ORB modifications to allow client applications interacting with replicated objects to benefit from replication and failure transparency. In contrast, we propose an interceptor based solution that allows the enhancement of standard client applications with replication and failure transparency without requiring modifications to the ORB and to the client application code.

Let us finally remark that this study of interceptors is a part of the interoperable replication logic (IRL) project [10], currently being carried out in our department, whose mission is to investigate the impact of three-tier architectures on software replication [11]. One of the main IRL targets is to develop an interoperable and portable implementation of the FT-CORBA specification running over ORB implementations of various vendors [12,13]. As shown in the following, portable interceptors play a key role in the context of the IRL project in implementing client-side failover mechanisms and client replication transparency.

The remainder of this paper is structured as follows. Section 2 presents an overview of CORBA portable request interceptors by showing its limitations and benefits. Section 3 proposes the basic mechanisms implementable by portable request interceptors, i.e. redirection and piggybacking. The same section describes a proxy-based technique that can be used to overcome some of the limitations described in Section 2. Section 4 deals with the results of the experiments. Section 5 describes a case study in which we use request portable interceptors to implement the FT-CORBA client-side invocation semantic. Finally, Section 6 concludes the paper. The Java code registered within the ORB by interceptors and used in the experiments is available at [10], while in Appendix A we present Java code fragments illustrating how to use the interceptors.

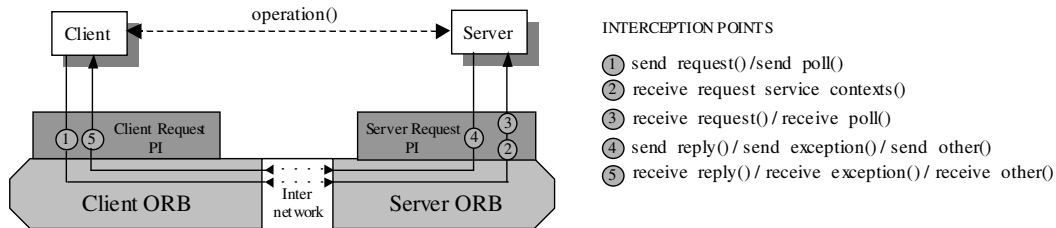


Figure 1. Client and server request portable interceptors.

2. OVERVIEW OF REQUEST PORTABLE INTERCEPTORS

Request portable interceptor (PI) are mechanisms which allow the ORB or application behaviour to be modified when sending or receiving a message (e.g. a request, a reply or an exception) without impacting either on the ORB or application code. Request PIs are logically set on top of the ORB layer (Figure 1) and can be registered within an ORB by invoking their interfaces (see Appendix A). Request interceptors are classified as either *client* or *server request interceptors*. The former are installed in client-side ORBs and can intercept outgoing requests and service contexts[§] as well as incoming replies and exceptions. Conversely, the latter are installed in server-side ORBs and can intercept incoming requests and service contexts as well as outgoing replies and exceptions. PIs can perform operations at different points during request processing. Figure 1 shows such *interception points*.

In particular, client request interceptors are activated either when a client issues a request (by implementing the `send_request()` or the `send_poll()` methods) or when a client receives a reply or an exception (by implementing the `receive_reply()`, the `receive_exception()` or the `receive_other()` methods).

Conversely, server request interceptors are activated either upon receiving a request (by implementing the `receive_request()`, `receive_poll()` or `receive_request_service_contexts()`) or upon the sending of a reply or of an exception (by implementing the `send_reply()`, `send_exception()` or `send_other()` methods).

By implementing these methods, request interceptors can be customized to implement the following actions.

- To access request or reply information: in each interception point an object that can be used by the interceptor developer to access request and reply information is provided. Client- and server-side interception points are concerned with different information, so there are two information objects:
 - the `ClientRequestInfo` is passed to the client-side interception points; or
 - the `ServerRequestInfo` is passed to the server-side interception points.

[§]Service contexts are slots containing service data transmitted transparently to the application. Its main use is to propagate information required by some ORB services.



Some information is common to both interfaces (operation, parameters, request id, etc.). Therefore both aforementioned interfaces inherit from a common interface, i.e. `RequestInfo`. Accessing the attributes of these interfaces is a strictly platform-dependent issue (see Section 2.1). Accessed information can be used to let the interceptor decide if it has to perform some action.

- To redirect a request to another target by throwing a CORBA `LOCATION_FORWARD` or `LOCATION_FORWARD_PERMANENT` exception.
- To throw other CORBA exceptions, e.g. to map an exception into another or a reply into an exception.
- To manipulate request service contexts, e.g. to piggyback additional information onto a message.
- To perform their own invocations.
- To delay a request or a reply.

By providing such features, PIs represent indeed a powerful development tool. Without impacting on the application code, they can be used, for instance, to piggyback authentication information into GIOP[¶] messages flowing between a client and a server, to uniquely identify client requests and to redirect them among different replicas of a fault-tolerant server [12,13], to share the load among different copies of a server, to implement caching mechanisms [14] or to implement flow control, as has also been shown in [3].

However, in some circumstances, PIs are not sufficient 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 CORBA `LOCATION_FORWARD` or `LOCATION_FORWARD_PERMANENT` exception;
- PIs cannot alter the parameters of a request or of a reply;
- PIs cannot modify request service contexts (but they can add their own contexts).

After its installation, a PI intercepts *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 request is intercepted, all the registered interceptor instances will be invoked by the ORB upon the arrival of a triggering message. The invocation order is ORB implementation dependent and cannot be either inspected or modified^{||}.

2.1. Java PI limitations

The main limitation of Java implementations of the PI specification consists of the impossibility of accessing some important attributes of the `RequestInfo` interface from an interceptor. In particular,

[¶]GIOP is the acronym of the General Inter-ORB Protocol, i.e. the specification of the abstract protocol allowing CORBA remote object interoperation. 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.

^{||}Actually, some ORB implementations allow interceptors to be chained in a customizable fashion, e.g. by defining their invocation order. However, assumptions about the order of invocation of multiple interceptor result in dependencies, i.e. in non-portable interceptors.



with Java portable bindings, an interceptor is not allowed to access the following `RequestInfo` attributes: `arguments`, `exceptions`, `contexts`, `operation_context`, `result`. In other words, a Java PI can only access the `operation` attribute among the attributes of the `RequestInfo` interface concerning the signature of an operation. If a PI tries to access one of the previously mentioned attributes, a `NO_RESOURCES` CORBA exception is thrown. This limitation implies that, with Java portable bindings, it is impossible for an interceptor to perform actions resulting from the attributes mentioned earlier (e.g. the operation arguments). This actually limits the scope of PIs. To overcome this limitation along with the impossibility of creating replies, the proxy design pattern [15] can be used. In Section 3.3 we show how to apply the proxy pattern in conjunction with portable interceptors to overcome their limitations. In [3], several common problems such as implementing caching, load-balancing, etc. are addressed using PIs and proxies. Note that the proxy pattern increases the flexibility of the implemented solutions by decoupling the request interception and redirection aspects from the application-dependent ones.

3. PI-BASED CLIENT-SIDE TECHNIQUES

In this section we focus on the basic building blocks of PI-based client-side enhancement. In particular, we deal with the request redirection techniques, the piggybacking technique and with the implementation of a proxy server that can access the request arguments, context etc. and can therefore perform actions resulting from such information.

3.1. Redirection techniques

Request redirection in CORBA is implemented by throwing a `LOCATION_FORWARD` or a `LOCATION_FORWARD_PERMANENT` exception. These exceptions contain an attribute, which is a reference to the object to which the request has to be redirected by the ORBs upon receiving them. If a `LOCATION_FORWARD` exception is received by a client ORB having issued a request, such a request is transparently redirected (i.e. issued again) to the object whose reference is contained in the exception. Following requests are *not* redirected. However, if a `LOCATION_FORWARD_PERMANENT` exception is received by a client ORB, not only the request being processed is redirected but also all the following requests issued towards the object for which a `LOCATION_FORWARD_PERMANENT` exception has been received (the client ORB permanently substitutes the reference it held with the new one contained in the `LOCATION_FORWARD_PERMANENT` exception).

However, these exceptions are commonly thrown by remote CORBA servers or by the CORBA Implementation Repository [16,17]. Without using PIs, it is not possible to redirect a request before the request leaves the client ORB. One of the main issues addressed by the PI specification is therefore client-side transparent request redirection. This technique allows a client request to be transparently redirected towards a target different from the one coded in the IOR held by the client.

As mentioned in Section 2, client-request PIs have access to a `ClientRequestInfo` object. The `ClientRequestInfo` object contains the `target` and `effective_target` attributes, which are usually equal and store the reference of the object to which the request is addressed.

Request redirection can be implemented by letting a client-request PI raise a `ForwardRequest()` exception in the `send_request()` or in the `send_poll()` interception points. This exception takes as input parameters a CORBA `Object` type attribute, namely `forward`, and a boolean



attribute, namely `permanent`. The `forward` object is used to set the new request target, i.e. the object to which the request has to be redirected. The value of the `permanent` flag determines whether the exception thrown to the ORB is `LOCATION_FORWARD` (`permanent=false`) or `LOCATION_FORWARD_PERMANENT` (`permanent=true`).

Independently from the `permanent` flag value, upon receiving a `LOCATION_FORWARD` or `LOCATION_FORWARD_PERMANENT` exception, the ORB

- reprograms the `effective_target` attribute of the request (and thus the `effective_target` attribute of `ClientRequestInfo` object) to the `forward` object value and then
- sends the request to the `forward` object.

It is important to note that the reissued request will also flow through the client-request PI. This means that a mechanism has to be implemented inside the PI in order to distinguish between first-time-caught and re-issued invocations (e.g. either by comparing the `effective_target` and the `target` `ClientRequestInfo` fields or by a flag**). Another way to avoid recursive scenarios is by exploiting the `request_id` attribute of the `RequestInfo` interface. A PI can store the ids of the requests for which it has thrown a forward exception and let requests pass through when reissued by comparing their ids.

The `permanent` flag indicates whether the `forward` object has to become the permanent target of all the following client requests or has to be used only on the request being forwarded, i.e. if the CORBA exception is `LOCATION_FORWARD` or `LOCATION_FORWARD_PERMANENT`. When using `LOCATION_FORWARD`, i.e. with the `permanent` flag set to `false`, the object reference held by the client ORB is not modified. This permits *per-request* redirection: each time the client sends a request to a given target, a `LOCATION_FORWARD` exception can be thrown to redirect the request each time to a distinct target. Figure 2 illustrates *per-request* redirection: each time a client performs an invocation to `Server2`, the client-request PI catches the request and creates a `ForwardRequest` exception object (see Appendix A). Then it sets the `forward` object value to the `Server1` object reference, the `permanent` flag to `false` and throws the (`LOCATION_FORWARD`) exception. Upon receiving such exception, the client ORB reissues the request having set `Server1` to be the request target. For each successive invocation, the client-request PI behaves exactly the same and it is also allowed to change the request target.

In contrast, when a client-request PI raises a `ForwardRequest()` exception with the `permanent` flag set to `true`, the object reference held by the client ORB is changed into the `forward` object value. This causes the ORB to redirect each successive client request to the new target automatically thus implementing a *per-client* redirection. As a consequence, both the `effective_target` and the `target` fields of the `ClientRequestInfo` object fields are set to the `forward` object value. Figure 3 illustrates this technique. The first time the client performs an invocation to `Server2`, the client-request PI throws a `LOCATION_FORWARD_PERMANENT` exception by setting to `true` the value of the `permanent` flag in a `ForwardRequest` exception. In Figure 3 the interceptor has set the `forward` object value to the `Server1` object reference value and

**The first method has to be preferred to the second which causes problems with concurrent CORBA clients. However, it applies only for non-PERMANENT forward exceptions.

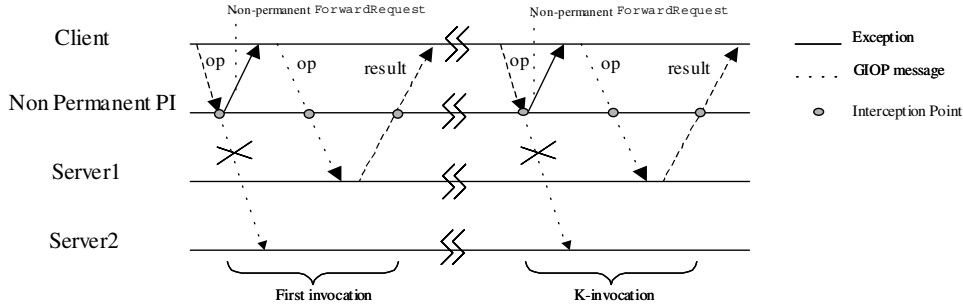


Figure 2. Per-request redirection.

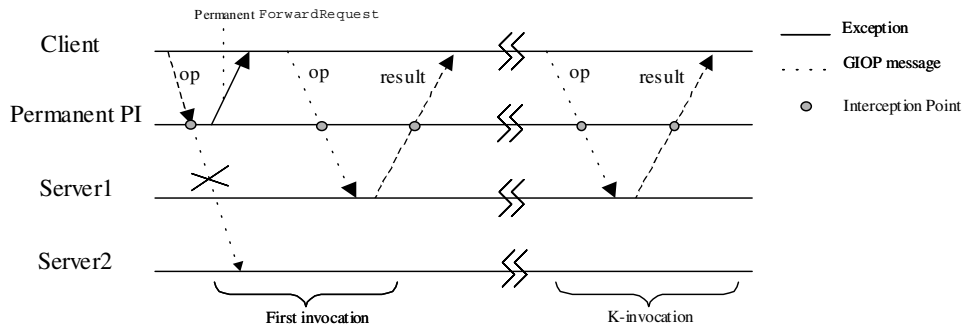


Figure 3. Per-client redirection.

the `permanent` flag to true. Upon receiving this exception, the client ORB permanently substitutes the `Server2` reference with `Server1`. Finally, the ORB reissues the request invoking `Server1`. All the following invocations will thus be addressed by the client ORB to `Server1`.

In this case a mechanism to distinguish between first-time-caught and reissued invocations is also needed in the PI but it cannot be implemented by comparing the `effective target` and the `target` fields of the `ClientRequestInfo` interface.

Note also that if a PI raises a `LOCATION_FORWARD` or a `LOCATION_FORWARD_PERMANENT` exception, no other installed interceptor is invoked at that interception point by the ORB. Furthermore, a client-request PI is allowed to throw at most one permanent forwarding exception per interception point.

3.2. Piggybacking

PIs can be used to piggyback information onto GIOP messages, e.g. for authentication purposes, without modifying client applications, server applications or ORBs, e.g. for authentication purposes.

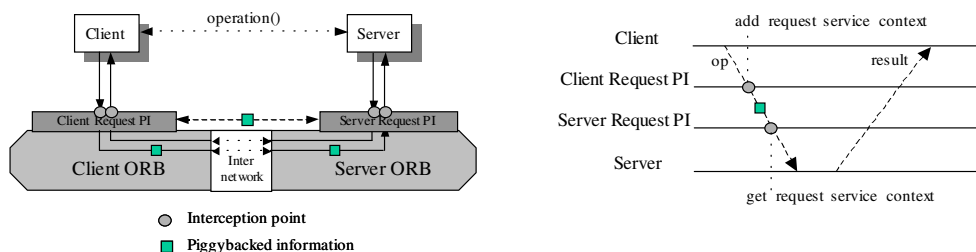


Figure 4. The piggybacking technique.

The flow of piggybacked information can be bidirectional, i.e. from a client to a server and *vice versa*. As an example, we show how to implement piggybacking from client to server. To achieve this, a client-request PI on the client ORB and a server-request PI on the server ORB have to be installed.

Figure 4 illustrates the piggybacking technique: the client-request PI intercepts outgoing requests (using the `send_request()` or the `send_poll()` method) and augments them with a GIOP service context (`IOP::ServiceContext`) by invoking a client-request PI method (`add_request_service_context()` of the `ClientRequestInfo` interface). This context contains a byte array in which the PI can insert the information to piggyback. When the request reaches the server ORB, the server-request PI intercepts it (by implementing `receive_request_service_context()` method) and then extracts the piggybacked information (by implementing the `get_request_service_context()` method) (see Appendix A for piggybacking interceptor code fragments).

3.3. Proxy-based techniques

In this section we describe how to overcome the general limitations to the PI described in Section 2 and the Java implementation ones described in Section 2.1. The main issues we address here is how to perform actions that strictly depend on the request content (e.g. parameters). As shown in [3], this problem arises in many contexts such as load balancing, caching, software fault tolerance etc. It can also arise in very simple scenarios concerning security, e.g. when a client has to transparently send a digest of the request to a server for authentication purposes.

Provided that a client PI cannot reply to caught requests on its own and, in the case of Java, cannot even read many of the `RequestInfo` attributes concerning the request under processing, a way of overcoming such limitations is to implement a local proxy server [15], i.e. a CORBA server object to which a client-request PI can redirect client requests. This local proxy is able to read the request contents and to perform actions that meet specific application requirements. However, depending on such requirements, several design choices concerning both the interceptor behaviour and the proxy deployment can be made.

In particular, we consider the following design choices.

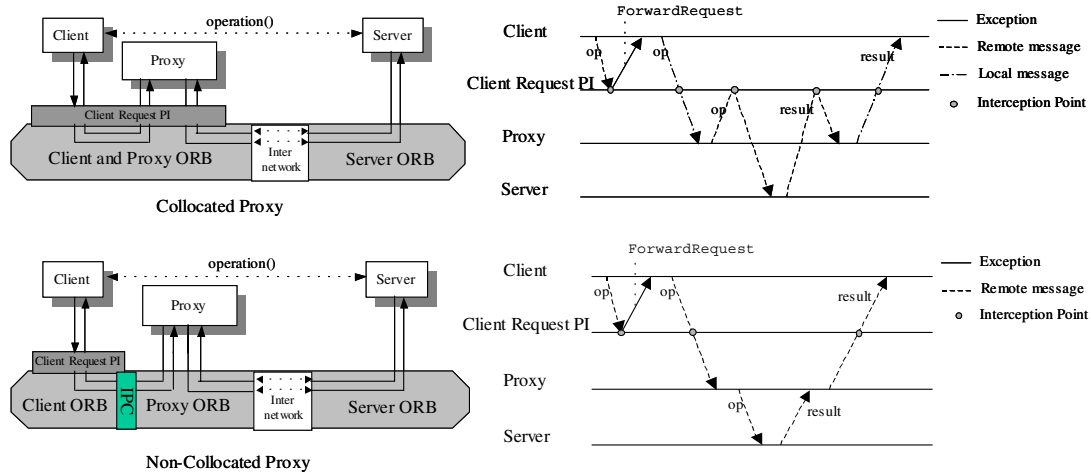


Figure 5. Collocated and non-collocated proxies.

- *Per-request and per-client redirection*: as pointed out in Section 3.1, client-request PIs can be programmed for performing either per-request redirection or per-client redirection to a proxy object. Per-request redirection to a proxy can be useful, for instance, when the proxy implements the functionality related to a given operation of an invoked object and the interceptor can decide by itself whether to invoke the proxy, or not, by reading the available attributes of the `RequestInfo` (e.g. the `operation` attribute). However, if the client interceptor is not allowed to access operation parameters to decide the request target or if the proxy entirely ‘wraps’ the server object functionality, then per-client redirection could be the appropriate choice.
- *Collocated and non-collocated proxies*: as shown in Figure 5, a proxy and a client are *collocated* [17] if compiled in the same addressable space. Otherwise they can be *non-collocated*, i.e. deployed in distinct processes^{††} running on a single host (this allows many clients to share the same proxy). We do not consider *remote* proxies, i.e. proxies residing on hosts different from the client one.

In the collocated client–proxy configuration, two distinct deployments can be devised:

- *collocated-one-ORB*: in this case, the client and proxy share the same ORB. As the PIs are registered on a per-ORB basis, the client and proxy also share the same set of PIs registered within the ORB. In particular, client and proxy share the client-request PI implementing the redirection of client requests to the proxy. Hence, outgoing requests from collocated proxies are also intercepted by the underlying client-request PI and in

^{††}In the case of Java, a non-collocated client and proxy run in different Java virtual machines.



this case a mechanism to avoid recursive scenarios has to be implemented. Unfortunately, there is no way of filtering PI actions on the basis of the references of the objects issuing requests, i.e. the mechanism to avoid recursive scenarios is application dependent (see Section 3.1).

- *collocated-two-ORBs*: in this case client and proxy, even if compiled in the same addressable space, run on distinct ORBs. This design is much more flexible than the previous one, allowing, for example, recursive scenarios to be easily avoided, and client-request PIs to be registered where it is more appropriate (e.g. the client-request PI performing the redirection has to be registered within the ORB on which the client runs) and to set ORB policies for the two ORBs independently.

Collocated proxies have to be compiled in bundle with the client application code. This reduces the application modification flexibility but can improve performance (see Section 4). However, non-collocated proxies can handle requests coming from more than one client application, while keeping both the client application and the proxy code modifiable without recompiling the other. However, non-collocated proxies can have performances lower than collocated ones due to ORBs performing marshalling and unmarshalling when interconnecting non-collocated objects.

- *Static and dynamic proxies*: CORBA allows a client to invoke a remote object in two ways, i.e. by using either static stub or a *Dynamic Invocation Interface* (DII). Analogously, a server object can accept incoming requests either by implementing a static skeleton or by using the *Dynamic Skeleton Interface* (DSI) [16,17]. We say that a proxy is *static* if implemented through a stub and a skeleton, i.e. exposing a static skeleton to the client(s) and exploiting a stub to invoke the remote server(s). A proxy is *dynamic* if it is implemented through DSI and DII, i.e. exploiting the DSI to read client requests and the DII to invoke remote objects dynamically. Static proxies are simpler to implement but less flexible than dynamic ones, which also strongly decouple client, proxy and server interfaces [18].

The correct choice among such dimensions is actually application dependent. For example, for high-performance applications, an appropriate choice could be implementing a static, collocated proxy with a PI performing per-client redirection. In contrast, for highly flexible applications, a dynamic, non-collocated proxy with per-request redirection could be the appropriate choice. The costs of these techniques are evaluated in the following section.

4. PERFORMANCE EVALUATION

In this section we first describe the testbed environment and how we carried out the experiments, then we introduce the basic measurements we made, that is the average latency and throughput of a simple client–server interaction for each of the ORBs we tested. Finally, we present benchmarks for each of the techniques described in the previous section comparing them with the basic measurements. The Java code used for the experiments is available at [10].

4.1. Testbed platform, experiments and basic measurements

Our testbed environment consists of two workstations interconnected by a 10 Mbit Ethernet LAN. Each workstation is equipped with a 300 Mhz Pentium II processor and 128 Mbyte of RAM. The

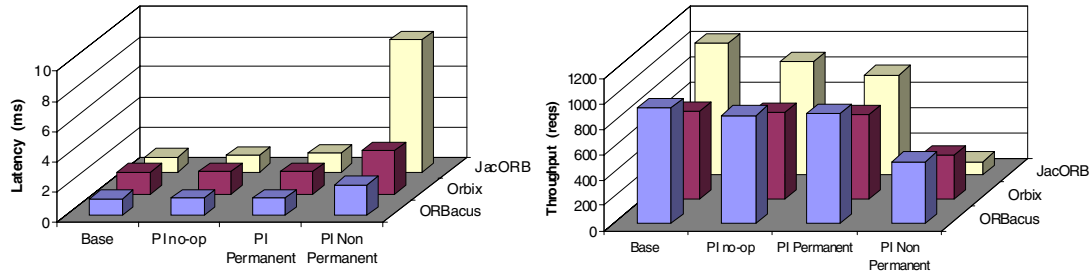


Figure 6. Latency and throughput of the basic measurements compared with the redirection benchmarks.

workstations run Microsoft Windows NT Workstation and are equipped with the Java Development Kit v.1.2.2. On each workstation, three Java ORBs implementing the PI specification have been installed and configured: JacORB v.1.3.21 [6], ORBacus v.4 [7] and Orbix 2000 for Java v.2.1 [8]. JacORB is a free Java implementation of the CORBA standard, while ORBacus and Orbix are commercial ORBs distributed by IONA.

For each ORB, we implemented a Java static CORBA client performing a synchronous invocation to a static CORBA server, deployed on the other host. The server replies as soon as it receives the request. In such a simple scenario, we tested and measured the cost of redirection, piggybacking and proxy-based configurations in terms of latency and throughput.

In particular, each of the following experiments measures the latency and throughput of the techniques described in the previous section. To measure latency, for each experiment, we launched a batch of 10 000 consecutive client requests 20 times and measured the time it took to complete the batch. To measure the throughput, for each experiment we launched a batch of 15s of duration 20 times, during which the client continuously issued requests, and we measured the number of requests completed within the time interval. Average latency and throughput were evaluated discarding the data concerning the first request, when the client and server ORBs interconnect. The network load was kept under control during the experiments. As a consequence, variance of both latency and throughput was always less than 5%, and is not reported further.

4.2. Basic measurements

The first set of measures is shown in the first column (labelled *Base*) in Figure 6, which reports the latency and throughput for a simple CORBA client-server interaction. In the test scenario, no interceptor is installed and a client invokes a server operation via a static stub. The server is also static (it implements a skeleton) and immediately returns.

Table I shows the average latency and throughput of the three CORBA ORBs. The second measurement is the latency increment (and the corresponding throughput reduction) when a no-op



Table I. Latency and throughput of a simple client–server interaction.

	JacORB	ORBacus	Orbix
Latency (ms)	1	1.1	1.44
Throughput (req s ⁻¹)	1040	917	694

Table II. Costs of potential flexibility (%).

	JacORB	ORBacus	Orbix
Δ Latency	+10	+1.39	+1.39
Δ Throughput	-15.94	-7.13	-1.46

client request PI is installed in the client ORB, i.e. the cost of potential flexibility due to interceptors^{‡‡} [18]. The results are shown graphically in the second column (labelled *PI no-op*) in Figure 6, while the percentage variations are reported in Table II.

4.3. Redirection performance

The third and fourth columns in Figure 6 (labelled *PI Permanent* and *PI Non Permanent*) show the latency and throughput of the permanent (i.e. per-client) and non-permanent (i.e. per-request) redirection techniques. Using per-client redirection results in performance which are similar to the PI no-op scenario. This is due to the fact that the `LOCATION_FORWARD_PERMANENT` exception is thrown only upon the first intercepted client request (see also Figure 3).

Concerning per-request redirection, a `LOCATION_FORWARD` exception is thrown for each request (see also Figure 2). Therefore performance sharply reduces, as reported in Table III, which shows the cost increases with respect to the Base scenario*.

^{‡‡}This test actually measures the cost of potential *client-side* flexibility, having not registered server-request PIs within the server ORBs.

*Note that the values obtained by JacORB in such a configuration (per-request redirection) are influenced by an ORB bug that we found during the experiments. JacORB developers rapidly delivered us a patched release that fixed this bug, allowing us to continue the experiments however without dealing with efficiency issues. This explains the high cost of per-request redirection in JacORB.



Table III. Costs of per-request redirection (%).

	JacORB	ORBacus	Orbix
Δ Latency	+780	+85.45	+97.92
Δ Throughput	-890.48	-87.91	-98.2

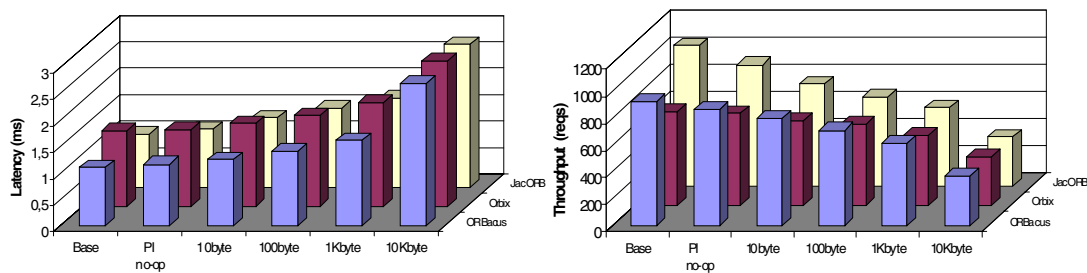


Figure 7. Piggybacking latency and throughput compared with the basic measurements.

4.4. Piggybacking performance

Figure 7 shows the throughput and latency of a client-server interaction exploiting the piggybacking technique described in Section 3.2 and transferring from the client-request PI to the server-request PI a variable length string cast in a byte array.

We evaluated piggybacking costs considering four distinct string sizes, i.e. 10, 100, 1000 and 10000 bytes. For completeness, we also draw in Figure 7, the latency and throughput of the basic measurements. As expected, the cost of piggybacking increases with the size of the piggybacked information on all of the platforms. Table IV shows the percentage variations of latency and throughput with respect to the performance measured in the Base configuration.

4.5. Proxy-based technique performance

We tested all the possible configuration of the proxy-based techniques described in Section 3.3. In each of these configurations, the client ORB has a client-request PI that redirects outgoing requests to a local proxy. Upon receiving a client request, the proxy issues a new request to the remote server, waits for the reply, and then generates a reply for the client request.

The results for *static* proxies are shown in Figure 8, in which we compare the basic measurements (i.e. the *Base* and *PI no-op* columns) with the throughput and the latency of collocated-one-ORB (*1ORBColloc*), collocated-two-ORBs (*2ORBColloc*) and non-collocated (*NonColloc*) proxies with an



Table IV. Costs of Piggybacking (%).

Size of piggybacking (bytes)	JacORB	ORBacus	Orbix
Latency variations			
10	+32	+13.64	+9.03
100	+50	+28.18	+19.44
1000	+69	+47.23	+36.83
10 000	+171	+145.45	+91.67
Throughput variations			
10	-37.54	-15.49	-9.81
100	-57.58	-29.89	-15.28
1000	-79.62	-50.32	-33.20
10 000	-185.7	-149.18	-92.24

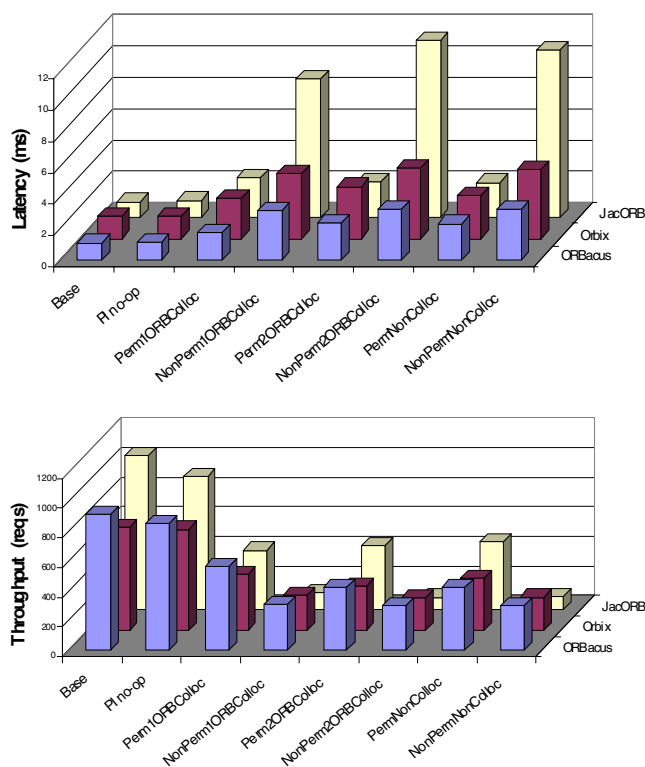


Figure 8. Latency and throughput of static proxy-based configurations compared with the basic measurements.

Table V. Costs of *static* proxy configurations (%).

Configuration	JacORB	ORBacus	Orbix
Latency variations			
Perm1ORBColloc	+150.00	+60.00	+83.33
NonPerm1ORBColloc	+780.00	+188.18	+192.36
Perm2ORBColloc	+128.00	+112.73	+131.25
NonPerm2ORBColloc	+1030.00	+199.09	+215.97
PermNonColloc	+120.00	+111.82	+94.44
NonPermNonColloc	+970.00	+200.00	+205.56
Average delay	+529.67	+145.30	+153.82
Throughput variations			
Perm1ORBColloc	-61.06	-37.95	-45.53
NonPerm1ORBColloc	-88.94	-66.19	-66.14
Perm2ORBColloc	-57.98	-53.65	-56.92
NonPerm2ORBColloc	-91.44	-66.96	-68.44
PermNonColloc	-56.25	-53.44	-48.70
NonPermNonColloc	-90.96	-67.28	-67.44
Average gap	-74.44	-57.58	-58.86

underlying client-request PI performing either per-client (*Perm*) or per-request (*NonPerm*) redirection. The analytical results are given in Table V.

Similarly, Figure 9 shows the result obtained with *dynamic* proxies compared with the basic measurements. The analytical results for dynamic proxy configuration are reported in Table VI.

As the experimental results illustrate, the redirection technique heavily influences the cost of adopting a proxy-based technique. As we expected, proxy-based configuration exploiting per-client redirection (*Perm* in the tables and figures) performs better than configurations exploiting per-request redirection (*NonPerm* in the tables and figures).

When dealing with proxy deployment issue, we expected a better performance using collocated proxies exploiting both per-request and per-client redirections. Surprisingly, non-collocated proxies perform similar to collocated ones. This issue can be traced back to ORBs not optimizing invocations between collocated objects, performing a complete request processing even if the interacting objects run within the same addressable space.

To address the problem of measuring the cost of flexibility due to exploiting dynamic invocation mechanisms (i.e. DII and DSI) in the proxy-based redirection contexts, we also compare the cost of implementing static proxies with that of implementing dynamic proxies. Figure 10 shows the increment in costs due to the use of dynamic interfaces for implementing the proxy, i.e. the percentage increment in the latency and the percentage decrement in the throughput due to the choice of implementing flexible, dynamic proxies. The analytical results are given in Table VII.

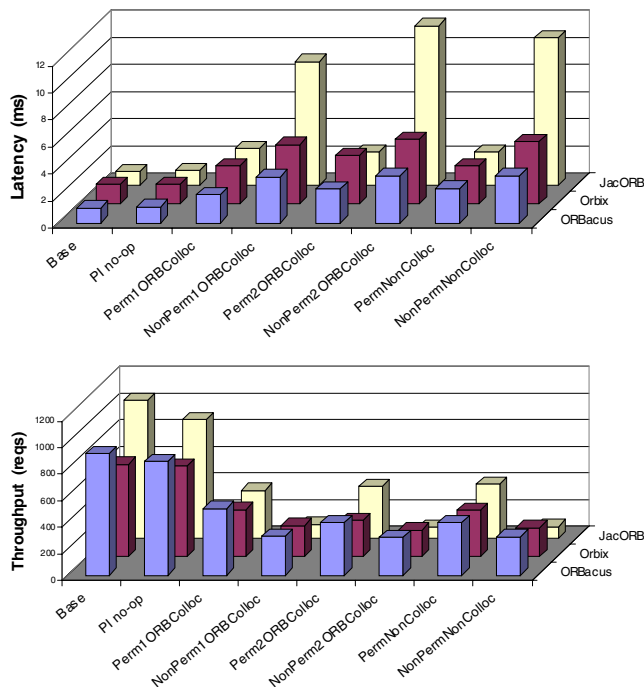


Figure 9. Latency and throughput of dynamic proxy-based configurations compared with the basic measurements.

Table VI. Costs of *dynamic proxy configurations* (%).

Configuration	JacORB	ORBacus	Orbix
Latency variations			
Perm1ORBColloc	+175.00	+83.64	+95.14
NonPerm1ORBColloc	+820.00	+205.45	+205.56
Perm2ORBColloc	+149.00	+126.36	+148.61
NonPerm2ORBColloc	+1080.00	+214.55	+232.64
PermNonColloc	+144.00	+130.00	+97.22
NonPermNonColloc	+990.00	+215.45	+220.14
Average delay	+559.67	+162.58	+166.55
Throughput variations			
Perm1ORBColloc	-64.90	-45.91	-48.70
NonPerm1ORBColloc	-89.52	-68.38	-66.71
Perm2ORBColloc	-62.21	-57.36	-60.37
NonPerm2ORBColloc	-91.92	-69.14	-71.04
PermNonColloc	-60.67	-56.92	-49.57
NonPermNonColloc	-91.15	-68.81	-68.88
Average gap	-76.73	-61.09	-60.88

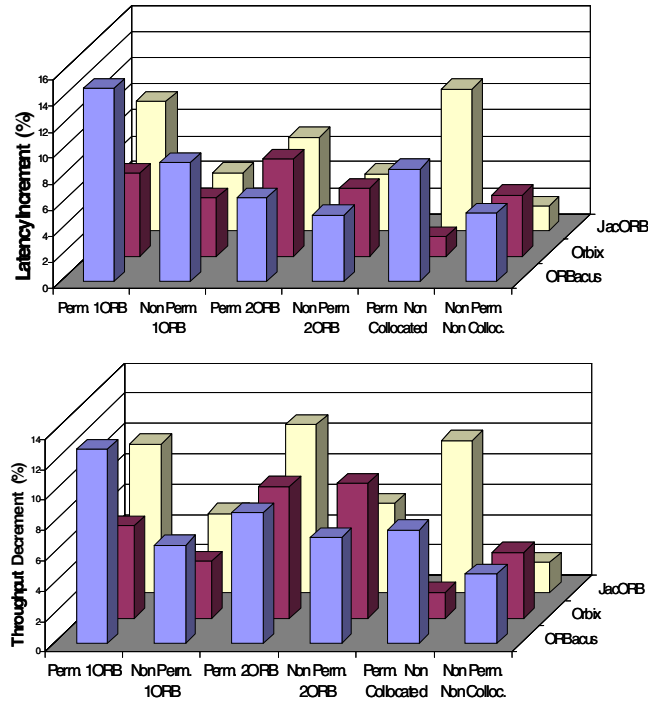


Figure 10. Comparison of dynamic and static proxy-based techniques.

Table VII. Costs of *dynamic* proxies compared with *static* proxies (%).

Configuration	JacORB	ORBacus	Orbix
Latency variations			
Perm1ORBColloc	+10.00	+14.80	+6.40
NonPerm1ORBColloc	+4.50	+9.10	+4.50
Perm2ORBColloc	+7.20	+6.40	+7.50
NonPerm2ORBColloc	+4.40	+5.10	+5.20
PermNonColloc	+10.90	+8.60	+1.50
NonPermNonColloc	+1.90	+5.20	+4.70
Average delay	+6.48	+8.20	+4.97
Throughput variations			
Perm1ORBColloc	-9.80	-12.80	-6.10
NonPerm1ORBColloc	-5.20	-6.50	-3.75
Perm2ORBColloc	-11.10	-8.60	-8.70
NonPerm2ORBColloc	-5.90	-7.00	-8.90
PermNonColloc	-10.00	-7.50	-1.70
NonPermNonColloc	-2.00	-4.60	-4.40
Average gap	-7.33	-7.83	-5.59



This cost ranges from about 1.5% of the Orbix platform (obtained in the *PermNonCollocated* case) up to about 15% measured on Orbacus (in the *PermIORBColloc* case). Note that different ORBs behave notably differently with respect to the use of dynamic invocation mechanisms.

Let us finally compare a proxy-based configuration that could be required by high-performance applications (i.e. static, collocated proxy, one or two ORBs with per-client redirection) and highly flexible (i.e. dynamic, non-collocated proxy with per-request redirection) applications (see Section 3.3). The cost of a proxy-based configuration, in term of latency, that adds maximum flexibility to the application is about three times the cost of the most rigid proxy-based configuration on ORBacus and Orbix, and increases by a factor of seven on the non-commercial JacORB ORB.

5. A PI-BASED FT-CORBA CLIENT SEMANTIC IMPLEMENTATION

In order to emphasize the potentials and limitations PIs, in this section we present a case study. In particular, we deal with the recently adopted Fault-Tolerant CORBA (FT-CORBA) specification [9,19]. In this specification, fault tolerance through entity redundancy, fault detection and recovery is achieved. The replicated entity is the CORBA object and replicated objects are managed through the abstraction of *object group* [20] to guarantee strong consistency among replicas. FT-CORBA mandates client ORB modifications to let CORBA clients benefit from failure and replication transparency. In other words, the client ORB has to mask CORBA client applications as much as possible from failures occurring to some member of a replicated object. FT-CORBA compliant client ORBs will achieve this task by implementing the so-called *transparent client reinvocation* and *redirection mechanisms*.

In the rest of the section, first we summarize the FT-CORBA client-side mechanisms to get client failure and replication transparency, then we show how it is possible to implement such mechanisms on ORBs implementing the PI specification without impacting on the ORB code.

5.1. FT-CORBA client-side mechanisms

In this summary of the client-side FT-CORBA specification, we omit some optional aspects (such as transport heartbeating) of the FT-CORBA specification to focus on the main mechanism implemented by FT-CORBA compliant client ORBs, i.e. Interoperable Object Group References (IOGR), transparent client reinvocation and redirection mechanisms.

5.1.1. Interoperable Object Group References

In order to deal with the group abstraction [20] in an object-oriented environment such as CORBA, FT-CORBA introduces the notion of Interoperable Object Group Reference (IOGR). IOGRs address groups of replicated CORBA objects, each group appearing to clients as a highly available singleton object. 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



	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 11. Failover conditions with and without transparent reinvocation.

identification within an object group which follows a primary-backup replication style. IOGRs are thus introduced in the CORBA specification to cope with such issues.

5.1.2. Transparent client reinvocation

The transparent client reinvocation mechanism provides clients with replication and failure transparency exploiting two submechanisms that we define *transparent client failover* and *transparent request identification*.

Transparent client failover. This 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 a NO_RESPONSE exception to the client) until (i) it has tried to invoke the replicated servers by exploiting *all* profiles contained in the IOGR while receiving only exceptions falling into the so-called *failover conditions*; (ii) it has received an exception that does not fall among such failover conditions; (iii) the request expiration time has elapsed (see below); or (iv) it has received the reply. Failover conditions model circumstances under which an ORB is allowed to reissue a request using a (possibly different) profile contained in an IOGR. Such circumstances are CORBA exceptions caused by failures.

Figure 11 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 and do not support IOGRs.

Transparent request identification. An FT-CORBA compliant client ORB is allowed to reissue requests even if COMPLETED_MAYBE is the request completion status returned with a failover condition's exception. In this case, a server could actually have executed the request and updated its state. Therefore, to avoid repeated executions of reissued requests, FT-CORBA defines how a client ORB must uniquely identify a request, to allow servers to automatically perform duplicate filtering.



Request identification is implemented by FT-CORBA compliant clients using the REQUEST service context. This service context is put in each request and contains fields that enable us to (i) uniquely identify the request, and (ii) to define the maximum request expiration time. This time is expressed as a duration, and it is the basic building block of a garbage collection mechanism. It provides servers with an upper bound on the time until which they must retain the request information and the corresponding reply (if any). Moreover, if an FT-CORBA compliant client ORB does not receive a reply or an exception to a request after the request expiration time has elapsed, it throws a NO_RESPONSE exception to the client.

5.1.3. Transparent client redirection

IOGRs held by clients may become obsolete: a membership change can occur asynchronously with respect to the client invocations. Similarly, slow servers could not receive membership change notifications. 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 invocation[†]. The solution to this problem adopted by FT-CORBA is based on an explicit exchange of membership version number between clients and servers. To let clients and servers hold the most updated IOGRs, they need to know which is the current version of the IOGR held by each other. For this reason, a FT_GROUP_VERSION service context is inserted into each client request. The FT_GROUP_VERSION service context contains a single field in which the client ORB copies the version of the IOGR identifying the group to be invoked. In this way, when a 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 reference with the more recent one held by the server. In the second case, the request is normally executed and a reply is returned to the client. If the reference sent by the client is more recent than the one held by the server, the server updates its reference and executes the operation.

5.2. The ORGW component

The Outgoing Request GateWay (ORGW) component allows a CORBA client running on a CORBA compliant ORB implementing PIs to interact with an object group by exploiting the transparent client redirection and reinvocation mechanisms described earlier[‡].

ORGW is intended to substitute a generic client ORB with an *extended* ORB. The extended ORB is a standard CORBA ORB within which a particular CORBA compliant request PI has been registered. To enhance client transparency, a CORBA client initializes the ORGW component as it initializes the

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

[‡]The ORGW component has been developed in the context of the IRL project. ORGW interacts with a tool, namely IOGRmanager, to retrieve, modify and browse IOGRs. Both components can be download at the IRL web site [10].

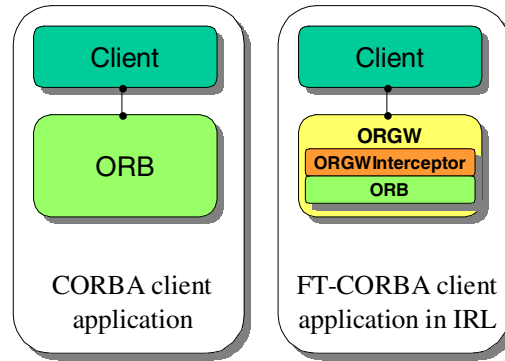


Figure 12. Common CORBA client compared with an FT-CORBA compliant client.

ORB. In particular, a Java CORBA client willing to exploit the ORGW functionality has to perform the following actions:

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 existing applications, a client has simply to substitute the `orb = org.omg.CORBA.ORB.init(args,prop)` instruction with `orb = orgw_init(args,prop, _FT_CORBA)`. 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 by using PIs and then its limitation concerning the request expiration time.

5.2.1. ORGW implementation

As already mentioned, the ORGW component logically extends an ORB with the functionality of a custom client-request PI, i.e. the `ORGWInterceptor`.

Figure 12 compares a standard CORBA client application with one using the ORGW component and shows the relationships among the client, ORGW, ORB and `ORGWInterceptor`.

`ORGWIntercetor` is the core of the ORGW component. `ORGWInterceptor` implements the following actions.



- *Request interception and filtering*: each time a client issues a request, the `ORGWInterceptor` checks whether the request is addressed to an object group or not. In the latter case, the `ORGWInterceptor` lets the request pass through it without interference, i.e. the request is normally handled by the ORB contained in the ORGW. If the request is addressed to an object group, the `ORGWInterceptor` executes the following steps.
- *Group request completion*: a request addressed to an object group is augmented with the `REQUEST` and the `FT_GROUP_VERSION` service contexts described in Section 5.1.2, used to implement the transparent request identification and redirection mechanisms, respectively.
- *Group request redirection and reinvocation*: after completing the request, the `ORGWInterceptor` extracts the first IIOP profile from the object group reference, 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 again pass through the `ORGWInterceptor`. Recursive scenarios are avoided in this case very easily: as the new request is addressed to an object and not to an object group, the interceptor lets the request pass through without interfering. After the ORB has reissued the request, one of the following events can happen:
 - (1) the invoked replica throws a `LOCATION_FORWARD_PERM` exception containing an updated object group reference;
 - (2) the invoked replica (or the client ORB) throws a CORBA exception falling into the *failover conditions* described in Section 5.1.2;
 - (3) the invoked replica (or the client ORB) throws a CORBA exception that does not fall into the *failover conditions* described in Section 5.1.2; or
 - (4) the invoked replica returns a reply or an user exception.

`ORGWInterceptor` listens for the occurrence of one of the previous four events by implementing the `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 whether it has exploited all 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 a limitation related to the request duration time. More specifically, in Section 5.1.2 we pointed out that an 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 timeouts to be set on request invocations, unless implementing the CORBA messaging specification [21] or the real-time CORBA specification



[22]. Moreover, such a timeout is not implementable inside ORGW as PIs are designed as pure stateless components. As a consequence our current ORGW implementation does not throw exceptions to the client after expiration of the request duration time, even though it inserts a client configurable request duration time into the REQUEST service contexts to enable server-side garbage collection of outdated requests (see Section 5.1.2)[§].

6. CONCLUSIONS

The interceptor technology is a promising tool allowing developers to extend the behavior of both applications running over a middleware and the middleware platform itself without impacting on their existing codes. For this reason, interceptors represent a powerful tool for adding specific network-oriented capabilities in a non-intrusive way. Exploiting interceptors also enables a sharp separation of application specific tasks from platform specific tasks, avoiding dangerous code mixtures in the application code.

In this paper we have reported the results of our experiences with a particular class of interceptors, namely the CORBA request PI, recently adopted as a part of the CORBA specification. In particular, we have pointed out (i) the main mechanisms implementable by PIs, e.g. request redirection and piggybacking, and (ii) the limitations imposed by the PIs programming model (such as no self-generation of replies and the impossibility of reading the arguments of a request at the PI level). We then discussed how to overcome these limitations by using a proxy. We have also presented a performance analysis summarizing the costs of adding PIs to a CORBA application while considering several scenarios and design choices. Finally, we have shown an application of CORBA PIs in the wider context of the FT-CORBA standard that mandates relevant modifications to client ORBs to implement a novel invocation semantic. The proposed interceptor-based solution allows us to implement this semantic without requiring modification to the ORB and to the client application code.

The code of the interceptors used in the experiments is available online at [10], while Appendix A presents commented code fragments to implement redirection and piggybacking.

APPENDIX A. JAVA CODE FRAGMENTS FOR IMPLEMENTING REQUEST PI-BASED TECHNIQUES

A.1. Initializing interceptors

```
// CLIENT INITIALIZER CLASS

import org.omg.PortableInterceptor.*;

/*
```

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



```
Class "ClientInitializer" is used here to register a client request PI
instance (named "client_interceptor") in the client ORB. By extending
org.omg.CORBA.LocalObject and by implementing the ORBInitializer
interface, this class is instantiated and is automatically invoked
by the ORB at creation time.
*/
public class ClientInitializer
    extends org.omg.CORBA.LocalObject
    implements ORBInitializer
{
    //empty constructor
    public ClientInitializer()
    {}

    //this method is invoked after ORB init phase, i.e. when all the
    //initial references are available
    public void post_init(ORBInitInfo info)
    {
        //...
        //creates a new client request PI instance
        Client_Request_Interceptor client_interceptor
            =new Client_Request_Interceptor();

        //method used to register the client request PI into
        //the client ORB
        info.add_client_request_interceptor
            (client_interceptor);

        //...
    }

    //method invoked before the ORB has resolved initial references.
    //Can be used to modify ORB init phase.
    public void pre_init(ORBInitInfo info) {
    }
}

//SERVER INITIALIZER CLASS

import org.omg.PortableInterceptor.*;

/*
Class "ServerInitializer" is used here to register a server request PI
instance (named "server_interceptor") in the server ORB. By extending
org.omg.CORBA.LocalObject and by implementing the ORBInitializer
interface, it is automatically instantiated and invoked by the ORB
at creation time.
*/
public class ServerInitializer
    extends org.omg.CORBA.LocalObject
    implements ORBInitializer
{
```



```

//empty constructor
public ServerInitializer()
{

//this method is invoked after ORB init phase, i.e. when all the
//initial references are available
public void post_init(ORBInitInfo info)
{
    //creates a new server request PI instance
    Server_Request_Interceptor server_interceptor
        =new Server_Request_Interceptor();
    //adds the server request PI to the ORB
    info.add_server_request_interceptor
        (server_interceptor);
}

//method invoked before the ORB has resolved initial references.
//It can be used to modify ORB init phase.
public void pre_init(ORBInitInfo info) {
}
}

```

A.2. Redirection techniques

```

//FORWARDREQUEST

import org.omg.PortableInterceptor.*;

/*
Class "Client_Request_Interceptor" is the client request PI registered
with the client ORB. By extending org.omg.CORBA.LocalObject and by
implementing the ClientRequestInterceptor interface, it is invoked
by the ORB at every client invocation.
*/
public class Client_Request_Interceptor
    extends org.omg.CORBA.LocalObject
    implements ClientRequestInterceptor
{

//empty constructor
public Client_Request_Interceptor()
{

//send_request is a ClientRequestInterceptor hook method.
//In this interceptor point it is possible to redirect
//the GIOP request message to another target object throwing
//the ForwardRequest exception.
public void send_request(ClientRequestInfo ri)
    throws ForwardRequest
    {

```



```
    //The Java ForwardRequest exception accepts two parameters: the first
    //is the object to which the GIOP request will be forwarded,
    //the second is a boolean flag indicating whether the forward object
    //has to become permanent or used only for the forwarded request.
        permanent = true //throws a permanent ForwardRequest exception
        throw new ForwardRequest(target, permanent);
    }
    //...
}
```

A.3. Piggybacking

```
//CLIENT REQUEST INTERCEPTOR PIGGYBACKING CODE

import org.omg.PortableInterceptor.*;
import org.omg.IOP.ServiceContext;

/*
Class Client_Request_Interceptor is the client request PI registered with
the client ORB. By extending org.omg.CORBA.LocalObject and by implementing
the ClientRequestInterceptor interface, it is invoked by the ORB at
every client invocation.
*/
public class Client_Request_Interceptor
    extends org.omg.CORBA.LocalObject
    implements ClientRequestInterceptor
{
    //The identifier of the IOP::ServiceContext object that is going to be
    //piggybacked. Useful for identifying contexts piggybacked by
    //different services
    private const org.omg.IOP.ServiceId SERVICE_CONTEXT_ID=1;

    //empty constructor
    public Client_Request_Interceptor()
    {}

    //send_request is a ClientRequestInterceptor hook method.
    //In this interception point, it is possible to insert piggybacked
    //information into the GIOP request.
    public void send_request(ClientRequestInfo ri)
        throws ForwardRequest
    {

        //The ClientRequestInterceptor add_request_service_context method
        //allows to insert in the GIOP request an IOP::ServiceContext
        //objects. The IOP::ServiceContext is a struct composed by an
        //byte array data field and by a ServiceId field (identifier)
        byte[] data="To fix or not to fix" //piggybacked info
        org.omg.IOP.ServiceContext sc =
            new org.omg.IOP.ServiceContext();
        sc.context_id =SERVICE_CONTEXT_ID;
    }
}
```



```

        sc.context_data = data;
        ri.add_request_service_context(sc, false);
    }

    //...
}

//SERVER REQUEST INTERCEPTOR PIGGYBACKING CODE

import org.omg.PortableInterceptor.*;
import org.omg.IOP.ServiceContext;

/*
Class Server_Request_Interceptor is the server request PI registered with
the server ORB. By extending org.omg.CORBA.LocalObject and by implementing
the ServerRequestInterceptor interface, it is invoked by the ORB
for each incoming request.
*/
public class Server_Request_Interceptor
    extends org.omg.CORBA.LocalObject
    implements ServerRequestInterceptor{

    //The identifier of the IOP::ServiceContext object that is going to be
    //piggybacked. Useful for identifying contexts piggybacked by
    //different services.
    private const org.omg.IOP.ServiceId SERVICE_CONTEXT_ID=1;

    //empty constructor
    public Server_Request_Interceptor()
    {}

    //receive_request_service_contexts is a ServerRequestInterceptor
    //hook method.
    //In this interception point it is possible to read the piggybacked
    //IOP::ServiceContext objects (if any), identified by a ServiceID
    public void receive_request_service_contexts(ServerRequestInfo ri)
        throws ForwardRequest
    {

        //The RequestInfo get_request_service_context method allows to read the
        //piggybacked IOP::ServiceContext objects. The service context is
        //identified by the ServiceId.
        org.omg.IOP.ServiceContext sc =
            ri.get_request_service_context(SERVICE_CONTEXT_ID);
        byte[] data =sc.context_data;
    }

    //...
}

```



ACKNOWLEDGEMENTS

The authors would like to thank the anonymous reviewers that helped us to clarify the presentation of the paper. The authors would also like to thank Roy Friedman for the interesting discussions on Portable Interceptors. A special thanks goes to Nicolas Noffke of the JacORB development team for his support during the experiments.

This work is partially supported by a grant from Alenia Marconi Systems, by a grant from MURST in the context of project 'DAQUINCIS', and by a grant from the EU in the context of the IST project MIDAS 'Middleware for Composable and Dynamically Adaptable Services'.

REFERENCES

1. DCE home page. <http://www.osf.org/dce>.
2. Eddon G, Eddon H. *Inside Distributed COM*. Microsoft Press: Redmond, WA, 1998.
3. Friedman R, Hadad E. 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; 179–185. Also appeared in *Journal of Computer System Science and Engineering* 2002; **17**(2):3–9.
4. Object Management Group (OMG). *Portable Interceptor Specification*, OMG Document orbos edn, OMG Final Adopted Specification. Object Management Group (OMG): Framingham, MA, December 1999.
5. Object Management Group (OMG). *The Common Object Request Broker Architecture and Specifications, Revision 2.4.2*, OMG Document formal edn, OMG Final Adopted Specification. Object Management Group (OMG): Framingham, MA, February 2001.
6. JacORB Web site. <http://www.jacorb.org>.
7. ORBacus Web site. <http://www.ooc.com/ob/>.
8. Orbix2000 Web site. http://www.iona.com/products/orbix2000_home.htm.
9. Object Management Group (OMG). *Fault Tolerant CORBA Specification, V1.0*, OMG Document ptc/2000-12-06 edn, OMG Final Adopted Specification. Object Management Group (OMG): Framingham, MA, April 2000.
10. IRL project Web site. <http://www.dis.uniroma1.it/~irl>.
11. Marchetti C. A three-tier architecture for active software replication. *PhD Thesis*, Dipartimento di Informatica e Sistemistica, Università degli Studi di Roma, 2002. <http://www.dis.uniroma1.it/~marchet>.
12. Marchetti C, Mecella M, Virgillito A, Baldoni R. An interoperable replication logic for CORBA systems. *Proceedings of the 2nd International Symposium on Distributed Objects and Applications*, Antwerpen, Belgium, September 2000; 7–16.
13. Marchetti C, Virgillito A, Baldoni R. Design of an interoperable FT-CORBA compliant infrastructure. *Proceedings of the 4th European Research Seminar on Advances in Distributed Systems (ERSADS01)*, Bertinoro, Bologna, Italy, May 2001; 155–160.
14. Chockler G, Dolev D, Friedman R, Vitenberg R. 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)*, April 2000. IFIP/ACM: New York, 2000; 1–23.
15. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley: Reading, MA, 1994.
16. Tair Z, Bukhres O. *Fundamentals of Distributed Object Systems—The CORBA Perspective*. Wiley: New York, 2001.
17. Henning M, Vinosky S. *Advanced CORBA Programming with C++*. Addison-Wesley: Reading, MA, 2000.
18. Wang N, Parameswaran K, Schmidt DC. The design and performance of meta-programming mechanisms for object request broker middleware. *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS01)*, San Antonio, TX, January/February 2001.
19. Object Management Group (OMG). *Fault Tolerant CORBA Finalization Task Force (FTF) Report*, OMG Document Formal edn, OMG Finalization Report. Object Management Group (OMG): Framingham, MA, December 2000.
20. Powell D (ed.). Special section on group communication. *Communications of the ACM* 1996; **39**(4):50–97.
21. Object Management Group (OMG). CORBA messaging. *The Common Object Request Broker Architecture and Specifications, Revision 2.4.2*, ch. 22, OMG Document formal edn, OMG Final Adopted Specification. Object Management Group (OMG): Framingham, MA, February 2001.
22. Object Management Group (OMG). Real time CORBA. *The Common Object Request Broker Architecture and Specifications, Revision 2.4.2*, ch. 24, OMG Document formal edn, OMG Final Adopted Specification. Object Management Group (OMG): Framingham, MA, February 2001.