CORBA Request Portable Interceptors: A Performance Analysis

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

Abstract

Interceptors are a mean to add specific network-oriented capabilities (such as authentication, flow control, caching etc.) to a distributed application which runs over a middleware without changing either the application code or the middleware's one. However, interceptors could be non-intuitive and this could in turn limit their use on a large scale. In this paper we present results of an investigation on CORBA portable interceptors in Java on various CORBA platforms. This study includes the identification of the basic mechanisms provided by an interceptor, of their limitations, a proxy-based technique to overcome some of these limitations and a performance analysis. We also release fragments of Java code used for experiments on Interceptor.

1 Introduction

A middleware is a software layer that sits between the application and the operating system in a distributed system, facilitating the development and the deployment of distributed applications. Examples of middleware platforms include DCE [1], DCOM [5]. Most middlewares achieve this goal by providing location transparent method invocation on remote objects, and by taking care of all issues that arise from the heterogeneity of distributed systems. Middleware undertakes the burden of dealing with low level networking problems, data representations, marshaling and unmarshaling, location transparency and other similar problems, letting the programmer concentrate on the essential part of the development, without worrying about the incidental part [3]. For this reason, middlewares are becoming extremely important in designing distributed applications.

Interceptors are non-intrusive hooks allowing to extend the middleware functionality without using intrusive techniques such as either modifying the application existing code, or the middleware core infrastructure. Interceptors are preferable to intrusive solution for several reason: first, by plugging in extensions to an existing middleware, one enjoys the benefit of economy of scale. That is, core middleware infrastructure has not to take into account extensions that will be used only by a specific set of applications. Second, interceptors can be plugged in middleware from various vendors. As described in [6], interceptors can help the application to manage some specific network-oriented tasks such as: authentication, caching, load balancing, flow control just to name a few.

In this paper we focus on Request Portable Interceptors [13] for CORBA [12]. CORBA 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 marshaling, message transmission, message unmarshaling 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, automatically executed by the ORB, i.e. transparently to client and server applications. The Portable Interceptors specification defines two types of interceptors, namely *Request Interceptors* and *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 the modification of an Interoperable Object Reference (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 under-

stand request interceptor capabilities, limitations and performance. In particular, we focus on:

- identifying the base mechanisms implementable by portable request interceptors, namely, request redirection and piggybacking. For each mechanism, we provide the Java interception code registered with the ORBs (the code is available online at [9]);
- identifying limitations of portable interceptors (e.g. the impossibility of generating replies and, in Java ORB implementations, of reading various request parameters);
- analyzing the implementation of a proxy-based technique (proposed in [6]), which exploits the mechanisms implementable by interceptors and overcomes their limitations;
- evaluating the cost of the basic interception mechanisms and of the proxy-based technique when considering several design choices. This evaluation study proposes several benchmarks obtained by running experiments on three different Java ORB implementations (JaCorb [8], Orbacus [15] and Orbix 2000 for Java [16]).

Even though quite promising, the technology based on interceptor is far from being a mature technology: during our experiments we found several bugs in the ORBs under test. Some of these bugs were fixed by the developers of the ORBs during the experiments, other bugs, at the time of this writing, are still present in some ORB and made impossible the execution of a few experiments (see Section 4).

Let us finally remark that this study on interceptors is a part of a project that we are carrying out in our department to develop a Fault Tolerant CORBA Compliant [14] Interoperable Replication Logic (IRL) running over ORBs implementations of various vendors [10, 11]. In the IRL project, portable interceptors play a key role for implementing client-side failover and replication transparency.

The remainder of this paper is structured as follows: Section 2 presents and overview of request portable interceptor by showing their limitations and assets. Section 3 proposes the basic mechanisms implemented by portable interceptors: redirection and piggybacking. The same section shows a proxy-based technique that overcomes some of the limitations described in Section 2. Section 4 deals with the results of the experiments. Section 5 concludes the paper. Fragments of the code registered in the interception points and used in the experiments are available at [9].

2 Overview of Request Portable Interceptors

Portable Request Interceptors (PIs) are a mechanism allowing to modify the ORB or 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 on top of the ORB layer (Figure 1) 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 (see [9]). 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).

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 the methods listed above, request interceptors can be configured to:

- access request or reply information (this is a strictly platform dependent issue as pointed out in Section 2.1) to perform some action;
- redirect a request to another target by throwing a `ForwardRequest` exception;
- 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 information into GIOP[1] messages flowing between a client

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

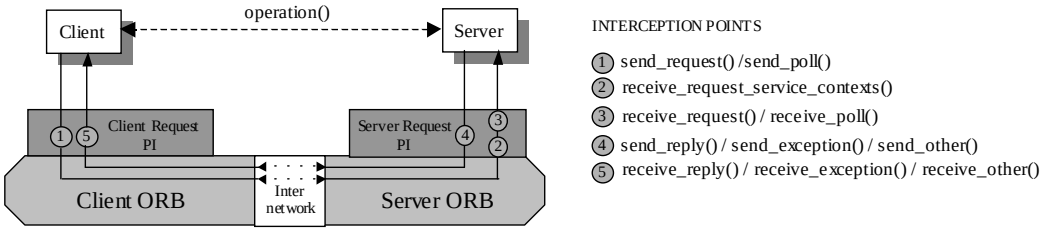

Figure 1. Client and server request Portable Interceptors

and a server, to uniquely identify client requests and redirect them among different replicas of a fault-tolerant server [10, 11], to share the load among different copies of a server, to implement caching mechanisms [4] or to implement flow control, as also shown in [6].

However, in some circumstances, PIs do not suffice to meet the application requirement. 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 `ForwardRequest` 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);

After their installation, PIs intercept *all* the requests or replies exchanged between client and server ORB. Different interceptor instances can be registered with 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 event. The invocation order is ORB implementation dependent and cannot be either inspected or modified[2].

2.1 Java Portable Interceptors Limitations

The main limitation of Java implementations of the PIs specification consists in the impossibility of accessing some important fields of a request or of a reply from a request interceptor. In particular, with Java portable bindings, 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. This limitation implies that, with Java portable bindings, it is impossible for an interceptor to perform actions depending from the attributes mentioned above (e.g. the operation arguments). This actually limits the scope of PIs. To overcome this limitations along with the impossibility of creating replies, the proxy design pattern [7] can be used. In [6], 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 solution, by decoupling the request redirection aspects from the application dependent ones. We thus suggest the application of this pattern also with languages allowing to access all of the `RequestInfo` interface attributes, e.g. with `C++`.

3 PI-based Client-side Techniques

In this section we focus on the PI-based client-side enhancement basic building blocks. 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 thus perform actions dependent from such information.

3.1 Redirection Techniques

One of the main issues addressed by the PI specification is transparent request redirection. This technique allows to transparently redirect a client request towards a target different from the one coded in the IOR held by the client.

In order to access request information, client request PIs have access to a `ClientRequestInfo` object. This object inherits the `RequestInfo` interface (and thus suffers

[2] Actually, 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.

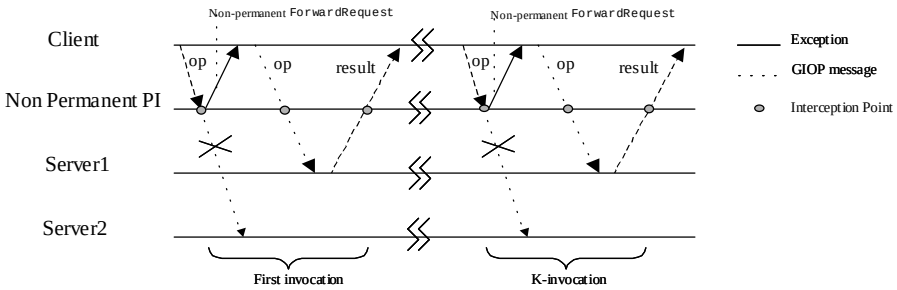

Figure 2. Per-request redirection

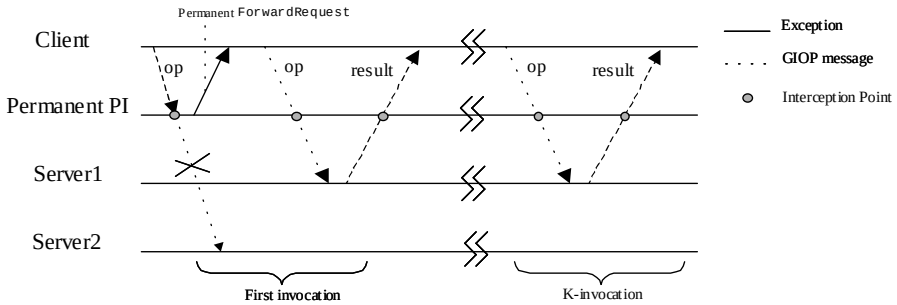

Figure 3. Per-client redirection

of all the restriction of such object) and stores information about client side request parameters. In particular, the `ClientRequestInfo` object contains the `target` and `effective_target` fields, which are usually equal and store the request destination IOR.

Request redirection can be implemented by letting a client request PI throw a `ForwardRequest` exception each time it captures an outgoing request, i.e. by throwing such exception in the `send_request()` or in the `send_poll()` method implementations. This exception has a CORBA `Object` type attribute, named `forward`, and a boolean attribute, named `permanent`. The `forward` object is used to set the new request target. Independently from the `permanent` flag value, upon receiving the `ForwardRequest` exception, the ORB:

- reprograms the `effective_target` field of the `ClientRequestInfo` object to the `forward` object value and then
- sends the request to the `forward` object.

It is important to note that also the re-issued request will 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).

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. When using a non-permanent `ForwardRequest` exception, i.e. with the `permanent` flag set to false, the object reference held by the client is not modified. This allows *per-request* redirection: each time the client sends a request to a given target, a non-permanent `ForwardRequest` exception can be thrown to redirect the request 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. Then it sets the `forward` object value to the Server1 object reference, the `permanent` flag to false and then throws the exception. Upon receiving such exception, the client ORB re-issues the request having set Server1 as 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.

On the contrary, when a client request PI throws a permanent `ForwardRequest` exception, i.e. with the `permanent` flag set to true, the object reference held by the client is changed into the `forward` object value. This causes the ORB to automatically redirect each following

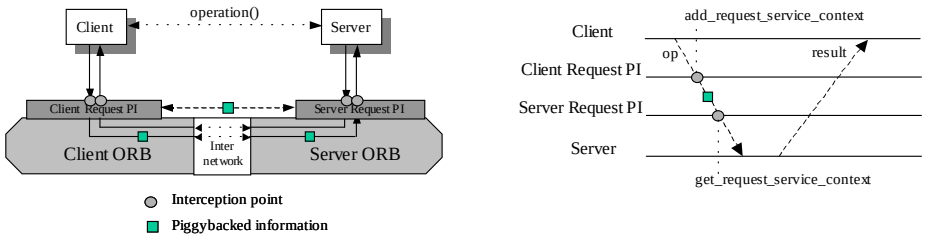

Figure 4. The piggybacking technique

client request to the new target, i.e. *per-client* redirection. As a consequence, both the `effective_target` and the `target` fields of the `ClientRequestInfo` object fields will be set to the `forward` object value. Figure 3 illustrates such technique. The first time the client performs an invocation to Server2, the client request PI throws a permanent `ForwardRequest` exception (having set the `permanent` object value to Server1 object reference and the `permanent` flag to true). Upon receiving such exception, the client ORB permanently substitutes the Server2 reference with the Server1 reference. Finally, the ORB reissues the request invoking Server1. All the following invocations will thus be addressed by the client ORB to Server1, and the client request PI will not be able to throw any other `ForwardRequest` exception.

Also in this case a mechanism to distinguish between first-time-caught and re-issued invocations is needed in the PI, but cannot be implemented by comparing the `effective target` and the `target` fields of the `ClientRequestInfo` interface[3].

Note that if a PI raises a `ForwardRequest` 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 `ForwardRequest` exception.

3.2 Piggybacking

PIs can be used to piggyback information onto GIOP messages (e.g. a request or a reply) without modifying clients, servers and ORBs (e.g. for authentication purposes). In such contexts a client request PI and a server request PI can be used. 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 shows the piggybacking technique implementation using PIs: the client request PI intercepts outgoing requests (e.g. 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()`). Such context contains a byte array to which piggybacked information has to be converted by the PI. 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).

3.3 Proxy-based Techniques

In this section we describe how to overcome the PI general limitations 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 depends on the request content. As shown in [6], this problem arises in many contexts such as load balancing, caching, software fault tolerance etc.[4].

Provided that a client PI cannot reply to caught requests by its own and, in the case of Java, cannot even read many of `RequestInfo` attributes, a natural way of overcoming such limitations is implementing a local proxy server [7]. In such a scenario a client request PI redirects each intercepted client request to a local proxy, which is able to read the request content and perform operations to meet the application requirements. However, depending from such requirements, different design choices concerning both the interceptor behaviour and the proxy deployment can be made.

In particular we consider the following design choices:

- **per-request and per-client redirection**: as pointed out is Section 3.1, client request PIs can be pro-

[3]This issue actually is application dependent. Recursive scenarios can be avoided by using a `thrown` boolean flag inside the interceptor, or by letting the interceptor know which are the request on which a `ForwardRequest` exception has or has not to be thrown.

[4]This problem can arise also 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.

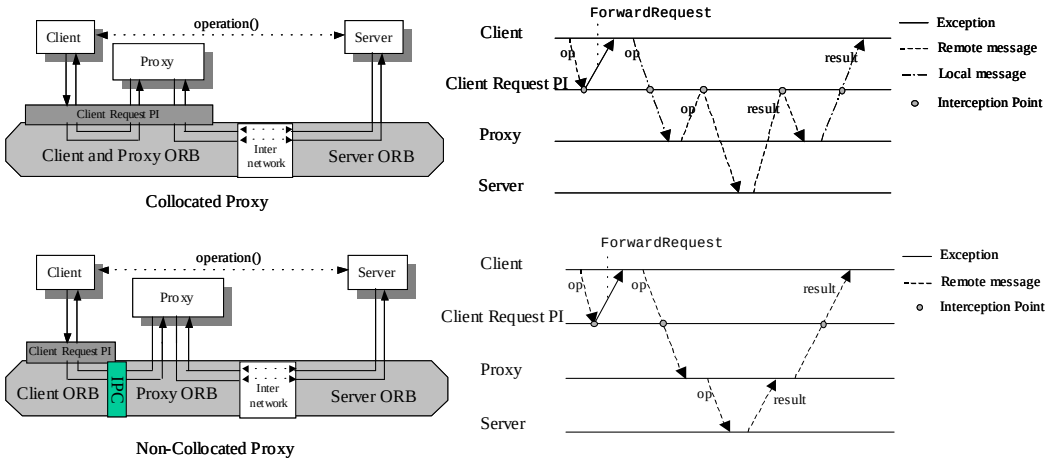

Figure 5. Collocated and non-collocated proxies

grammed for performing either per-request redirection or per-client redirection to a proxy object. Per-request redirection can be useful, for instance, when a proxy implements functionality related to a given operation of an object and the interceptor can decide by itself whether invoke or not the proxy by reading the `operation` attribute of the `RequestInfo` interface (always available). On the other hand, if a proxy entirely "wraps" a server object, then per-client redirection could be the appropriate choice.

- **collocated and non-collocated proxies**: as shown in Figure 5, a proxy is *collocated* if compiled in the same client addressable space. Otherwise it can be *non-collocated*, i.e. deployed in a distinct process running on the client host (thus allowing many clients to use the same proxy). We do not consider *remote* proxies, i.e. proxies residing on hosts different from the client one.

  Note that requests outgoing from collocated proxies are intercepted by the underlying client request PI and also in this case a mechanism to avoid recursive scenarios has to be implemented. Collocated proxies have to be compiled with the client, reducing modification flexibility but improving performance. On the other hand, non-collocated proxies can handle requests coming from more than one client and are modifiable without recompiling the clients.

- **static and dynamic proxies**: a proxy is *static* if implemented through a stub and a skeleton, i.e. exposing a static skeleton towards the client(s) and exploiting a stub to invoke the remote server(s). A proxy is *dynamic* if implemented through DSI and DII, i.e. exploiting DSI to read client requests and DII to dynamically invoke remote objects. Static proxies are simpler to implement but less flexible than dynamics, which also strongly decouple client, proxy and server interfaces [17].

The right choice among such dimensions actually is application dependent. For example, for high performance application, an appropriate choice could be implementing a static, collocated proxy with per-client redirection. On the contrary, for highly flexible applications, a dynamic, non-collocated proxy with per-request redirection could be the appropriate choice. Costs of such design choice will be 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 landmarks of our benchmarks, i.e. the latency and the throughput of simple stream-based client server interactions, for each of the ORBs we tested. Finally, we present benchmarks for each of the techniques described in the previous section.

4.1 Testbed Platform, Experiments and Landmarks

Our testbed environment consists in two workstations equipped with a 300Mhz Pentium II processor, 128Mbyte of RAM and interconnected by a 10Mbit Ethernet LAN. The workstations run Microsoft Windows NT Workstation as operative system 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

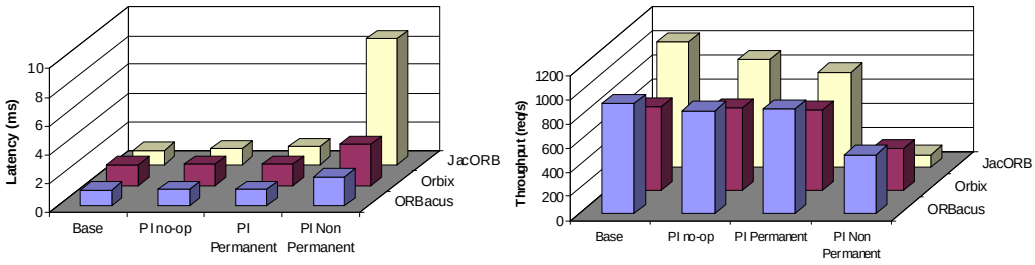

Figure 6. Latency and throughput of the landmarks compared with the redirection costs.

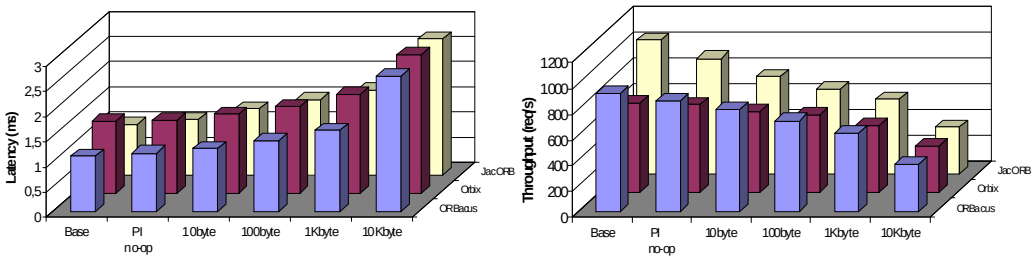

Figure 7. Piggybacking latency and throughput.

configured: JacORB v.1_3_21 [8], ORBacus v.4 [15] and Orbix 2000 for Java v.2.1 [16].

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

In particular, each of the following experiments measures latency and throughput of the techniques described in the previous section. To measure latency, for each experiment, we launched 20 times a batch of 10000 requests. To measure throughput, for each experiment we launched 20 times a batch of 15 seconds of duration. Average latency and throughput have been evaluated always discarding the data concerning the first request, during which the ORBs interconnect.

The first landmark for comparing the following results is shown in the first column (named *Base*) of Figure 6, that reports latency and throughput for a simple stream-based CORBA client server interaction. In the test scenario, no interceptor is installed and a client invokes a server operation via stub. The server is static (it implements a skeleton) and immediately returns. On average, to complete such a client server interaction ORBacus takes about 1,1 msec (with a throughput of 917 req/sec), Orbix about 1,44 msec. (with a throughput of 694 req/sec) and JacORB about 1 msec. (with a throughput of 1040 req/sec).

The second landmark is the latency increase (and the corresponding throughput decrease) when a no-op client request PI is installed in the client ORB, i.e. the cost of potential flexibility due to interceptors[5] ([17]). The results are shown in the second column (named *PI no-op*) of Figure 6. Installing client request PIs increases latency of about 6.36% in ORBacus, 1.39% in Orbix and 10% in JacORB, causing a throughput decrease of, respectively, 7.13%, 1.46% and 15.94%.

4.2 Redirection Performance

The third and the fourth columns of Figure 6 (named *PI Permanent* and *PI Non Permanent*) show latency and throughput for permanent (i.e. per-client) and non-

[5]This test actually measures the cost of potential *client-side* flexibility, having not registered server request PIs with the server ORB.

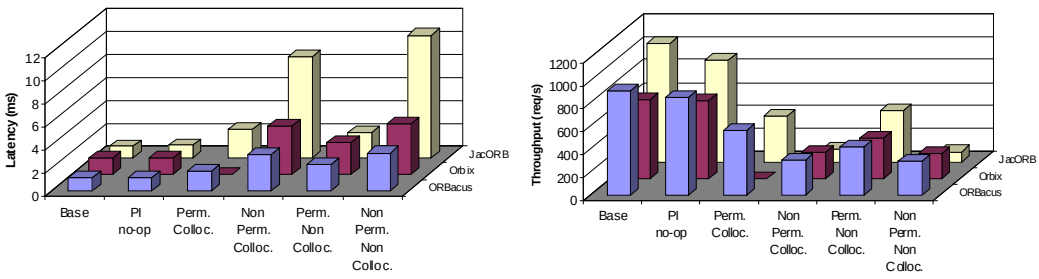

Figure 8. Latency and throughput of static proxy-based configurations.

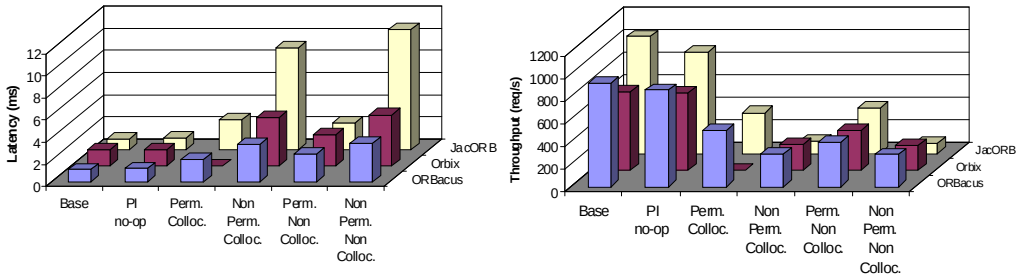

Figure 9. Latency and throughput of dynamic proxy-based configurations.

permanent (i.e. per-request) redirection technique. Using per-client redirection results in performances actually similar to the PI no-op scenario. This is due to the fact that the `ForwardRequest` exception is thrown only upon the first intercepted client request (see also Figure 3).

When using per-request redirection, however, the `ForwardRequest` exception is thrown for each request (see also Figure 2) and thus performance sharply reduces: with respect to the base scenario, latency increases of 85.45% for ORBacus, 97.92% for Orbix and of 780.00% for JacORB[6]. As a consequence, throughput has a reduction of about 87.91% for ORBacus, 98.29% for Orbix and of 890.48% for JacORB.

4.3 Piggybacking Performance

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

[6]Note that values obtained by JacORB in such 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 that bug, without dealing with efficiency issues. This explains the high cost of per-request redirection in JacORB.

We evaluated piggybacking cost for three distinct string sizes, i.e. 10,100, 1K and 10K bytes. For completeness, we also draw in Figure 7, latency and throughput of the two landmarks. As expected, the cost of piggybacking increases with the size of the piggybacked information for all the platforms. The following table shows the percentage variations of latency and throughput that we measured.

Latency Increment

	Comparison with landmark "Base"				Comparison with landmark "PI no-op"			
ORBs	10byte	100byte	1Kbyte	10Kbyte	10byte	100byte	1Kbyte	10Kbyte
ORBacus	13,64%	28,18%	47,23%	145,45%	6,84%	20,51%	38,46%	130,77%
Orbix	9,03%	19,44%	36,83%	91,67%	7,53%	17,81%	34,93%	89,04%
JacORB	32%	50%	69%	171%	20%	36,36%	78,18%	146,36%

Throughput Decrement

	Comparison with landmark "Base"				Comparison with landmark "PI no-op"			
ORBs	10byte	100byte	1Kbyte	10Kbyte	10byte	100byte	1Kbyte	10Kbyte
ORBacus	15,49%	29,89%	50,32%	149,18%	7,81%	21,25%	40,32%	132,61%
Orbix	9,81%	15,28%	33,20%	92,24%	8,23%	13,62%	31,28%	89,47%
JacORB	37,54%	57,58%	79,62%	185,71%	18,65%	35,91%	54,92%	146,43%

4.4 Proxy-based Technique Performance

We tested all off the possible configuration of the proxy-based techniques described in Section 3.3. In each of these configurations, upon receiving the client request redirected by the client request PI, the local proxy object issues a re-

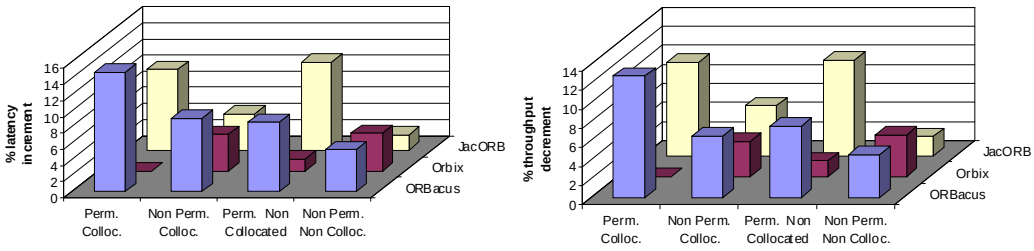

Figure 10. Comparison of dynamic and static proxy-based techniques.

quest to the remote server, waits for the reply and then generates a reply for the client (i.e. it essentially behaves relays to the server each client request). The results for *static* proxies, are shown in Figure 8, in which we compare with the landmarks (i.e. the *Base* and *PI no-op* columns) the throughput and the latency of collocated (*Colloc.*) and non-collocated (*Non Colloc.*) proxies with an underlying interceptor performing either per-client (*Perm.*) or per-request (*Non Perm.*) redirection[7].

Similarly, Figure 9 shows the result obtained with *dynamic* proxies compared with the landmarks.

The cost of implementing a proxy based technique strictly depends from the redirection technique exploited. As we expected, proxy-based configuration exploiting per-client redirection performs better than configurations exploiting per-request redirection.

When dealing with proxy deployment issue, we expected better performance using collocated proxies exploiting both per-request and per-client redirections. However, as the graphics points out, in JacORB this does not hold: using this ORB, in fact, we noticed that in the configurations exploiting per-client redirection, collocated proxies performs worse than non-collocated. This is actually impossible to verify for Orbix at the moment.

To address the problem of measuring the cost of flexibility due to exploiting dynamic interfaces in the proxy-based redirection context, we compared the cost of implementing static proxies with the ones of implementing dynamic proxies: Figure 10 shows the increment of costs due to the use of dynamic interfaces for implementing the proxy, i.e. the percentage increment of the latency and the percentage decrement of the throughput due to the choice of implementing flexible, dynamic proxies. Such cost ranges from about 3% of the Orbix platform (obtained in the *Perm. Non Collocated* case) , up to about 15% measured on the Orbacus one (obtained in the *Perm. Colloc.* case).

Let us finally compare the proxy-based configurations required by high performance (static collocated proxy with permanent redirection) and highly flexible (dynamic non-collocated proxy with non-permanent 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 twice the cost of the most rigid proxy-based configuration[8].

5 Conclusions

The interceptor technology is one of the most promising tools to add specific network-oriented capabilities to a distributed application which runs over a distributed system composed by one or several Middlewares. In this paper we have investigated limitations and assets of a particular type of interceptors, namely the Request Portable Interceptors, recently adopted by OMG as a part of the CORBA specification. In particular, we have pointed out the main mechanisms implementable by portable interceptors, such as redirection and piggybacking and discussed how to overcome some limitations (such as no generation of requests and impossibility of reading the context of a request at the portable interceptor level) by proposing a proxy-based technique. We did a performance analysis of the cost of adding interceptors in a simple client/server scenario when considering several designing choices. Fragments of the request interceptor code used in the experiments are available online at [9].

6 Acknowledgements

The author would like to thank Roy Friedman for interesting discussions on the Portable Interceptors and Antonino Virgillito for some interesting observations during the experiments. A special thank goes to Nicolas Noffke of

[7] In such figures, there are some columns missing. We got a bug when running the collocated-permanent scenario over Orbix ORB. The system enters an infinite loop. We are in touch with IONA tech. to receive a new ORBIX release that fixes this problem.

[8] This cost has been evaluated on Orbacus platform.

JacORB development team for his support during the experiments.

References

[1] DCE Home Page. http://www.osf.org/dce.

[2] R. Baldoni, C. Marchetti, A. Virgillito, F. Zito. A Failure Management Systems for FT-CORBA compliant Applications, *In Proc. of the 6th International Workshop on Object-Oriented Dependable Systems*, Rome, 2001.

[3] F. P. Brooks. *The Mythical Man-Month.* Addison Wesley, October 1995. 20th Anniversary Edition.

[4] G. Chockler, D. Dolev, R. Friedman, and R. Vitenberg. CASCADE: CAching Service for CorbA Distributed objEcts. In *ProcṀiddleware 2000: IFIP/ACM International Conference on Distributed Systems Platforms*, pages 1–23, April 2000. Best Paper Award.

[5] G. Eddon and H. Eddon. *Inside Distributed COM*. Microsoft Press, 1998.

[6] R. Friedman and E. Hadad. Client-side Enhancements using Portable Interceptors. In *Sixth IEEE International Workshop on Object-oriented Real-time Dependable Systems*, January 2001.

[7] E. Gamma, R. Helm, R. Johnson and J. Vlissides Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA *Addison Wesley*, 1995.

[8] JacORB web site: http://www.jacorb.org

[9] IRL Project web site: http://www.dis.uniroma1.it/˜ irl

[10] C. Marchetti, M. Mecella, A. Virgillito, R. Baldoni. An Interoperable Replication Logic for CORBA Systems, *In Proc. of the 2nd International Symposium of Distributed Object Applications*, 2000.

[11] C. Marchetti, M. Mecella, A. Virgillito, R. Baldoni. Integrating Autonomous Enterprise Systems through Dependable CORBA objects, *In Proc. of the 5th International Symposium on Autonomous Decentralized Systems* , Dallas, 2001.

[12] OMG. CORBA/IIOP Specification 2.4.

[13] OMG. CORBA/Portable Interceptor Specification. ptc/2000-04-05.

[14] OMG. Fault Tolerant CORBA Specification V1.0. ptc/2000-04-04.

[15] ORBacus web site: http://www.ooc.com/ob/

[16] Orbix2000 web site: http://www.iona.com/products/orbix2000_home.htm

[17] N. Wang, K. Parameswaran, and D. C. Schmidt, 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 (COOTS), San Antonio, Jan/Feb, 2001.*