Client-side Enhancements using Portable Interceptors*

Roy Friedman Erez Hadad

Department of Computer Science

Technion - Israel Institute of Technology

Haifa 32000, Israel

Abstract

Interceptors are a useful technique for extending the basic functionality provided by an Object Request Broker without changing its implementation, or affecting clients' code. The recently proposed CORBA standard includes a new definition of portable interceptors. *This definition is very powerful, yet not trivial to use. In this paper we review this definition, and discuss how it can be applied to provide client side enhancement for caching, load-balancing, flow control (quality of service), and fault-tolerant soft real-time. A recommendation to the OMG for a very minor change in the standard that could greatly simplify its usability is also provided.*

*This research is partially supported by the Israeli Ministry of Science, Basic Infrastructure Fund, Project #9762.

1 Introduction

CORBA is a standard architecture for accessing remote objects in an interoperable, location transparent manner [9], which unlike traditional RPC, supports object oriented programming. The emphasis on interoperability in CORBA plays a major role in the wide acceptance that CORBA is enjoying. At the heart of the CORBA architecture lies the Object Request Broker (ORB), which is responsible for passing client requests to the server and passing the replies back to the client. (We give a more precise description of how this is actually being done later in this paper.) An interesting aspect of CORBA is that it provides standard means for enhancing the basic functionality provided by a given ORB, which contributes to the flexibility and extensibility of the architecture, and of software designed on top of CORBA [8]. In particular, the newly proposed standard definition of *portable interceptors* is a powerful way of enhancing the functionality at both the client side and the servant side in a portable manner, i.e., in a way that is independent of specific ORB implementation [11].

The basic idea behind interceptors, as their name suggests, is to provide means for registering code that is automatically invoked in certain phases of the execution of a remote method. More specifically, a remote method invocation typically involves sending a request from a client to the servant, accepting the request at the servant, generating a reply from the servant to the client, and delivering the reply to the client. An interceptor can be invoked on any of the above events, thus enhancing the basic functionality provided by the ORB without changing the ORB's code, and transparently to the client's code.

CORBA has another standard way of extending the basic functionality of ORBs, namely using *smart stubs*. In order to provide the basic transparent remote invocation property, an interface must be defined for each CORBA object using the OMG Interface Definition Language. This interface is translated into an *IDL stub* on the side of the client, and a *skeleton* on the side of the servant. When the client invokes a method, it is actually invoking a corresponding proxy method in the stub, which then translates the invocation into a message that is sent by the ORB to the servant. Similarly, when the reply is received at the client side, it is passed to the stub that can then return control to the calling client code as if everything have happened in a normal local invocation. Thus, by overwriting the automatically generated stub, it is possible to add functionality in a transparent manner.

However, the difference between a smart stub and an interceptor is that a separate smart stub has to be written for each interface. On the other hand, the same interceptor can manipulate various types of objects, and is thus a more general solution. This of course comes at a price, since it is harder to implement application specific optimizations with interceptors.

In order to make interceptors' code portable, the new CORBA standard does not allow an interceptor to modify the request itself, or the contents of the reply. Moreover, an interceptor may not block the flow of a request or a reply other than by raising an exception, nor is it allowed to directly reroute a request other than by raising a `ForwardRequest` exception. At first sight, these limitations seem to dramatically limit the power of interceptors. However, as we show in this paper, many usual tasks that one would like to implement using interceptors can still be done with portable interceptors. At the same time, we have noticed that a very minor change in the standard that would not affect its portability, as describe below, could greatly simplify many of these tasks.

2 Portable Interceptors in CORBA

This section provides a very brief overview of CORBA. It is here for completeness. Readers familiar with CORBA may wish to skip this section.

2.1 A Brief Overview of CORBA

Under CORBA, each object implements an interface that is declared in the OMG IDL. This interface has to include all methods exported by the object, and the typed parameters they accept. As mentioned before, the IDL interface is then compiled using an IDL compiler into a client side stub, and a server side skeleton. The stub is linked with the client's code, and has the exact same interface as the object. When the client code invokes a remote method, it is in fact invoking the stub as a local call. The stub translates the call to a message according to the CORBA standard, and passes it to the local ORB, which sends the request to the ORB in which the object implementation runs.[1] The receiving ORB passes

[1] In the general case, the client and object implementation might be in different hosts, in the same host but in different processes, or even in the same process, in which case communication is handled either locally or remotely.

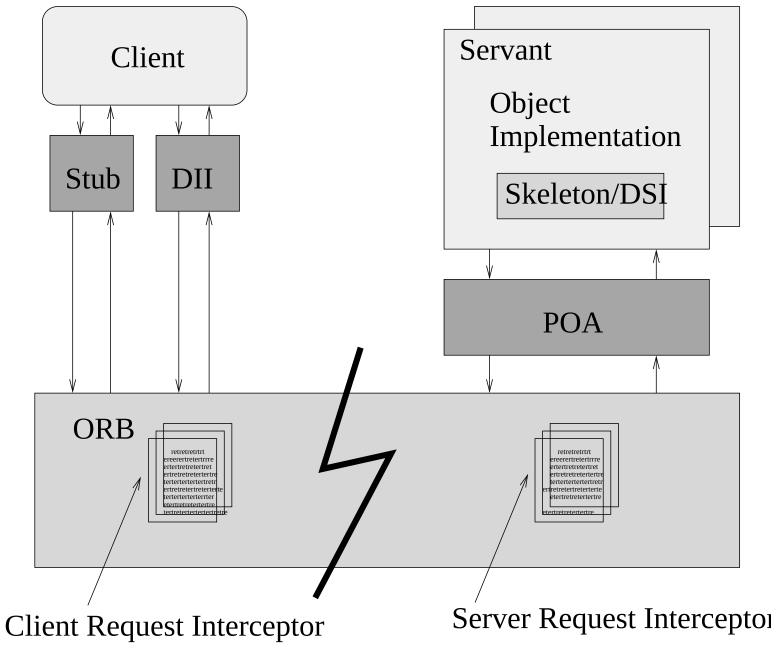

Figure 1. Main CORBA Components

the message to the *Portable Object Adapter (POA)*, or another object adapter. The object adapter locates the object and passes the message to the corresponding skeleton which runs within a given *servant*. The skeleton then invokes the object locally, and passes the inout and out parameters as well as the return value as a reply to the originating ORB. When the stub gets the reply, it returns to the client.

In order to access an object implementation, the client needs to obtain a reference to the object. CORBA defines a standard reference structure, called *Interoperable Object Reference*. The object implementation typically registers an IOR for itself in the *naming service*. The client can then lookup the IOR in the naming service. Alternatively, the client might obtain the IOR from another client, or read it from a file, etc.

In order to be useful, the IOR must include an object ID, a POA ID and an identifier for the host ORB on which the object implementation can be found. These are the stored in *profiles*, and the standard allows for the existence of multiple profiles in the same IOR (see Figure 2). Having multiple profiles is useful in order to support multiple paths to the same object implementation, possibly using different protocols, or paths to alternate implementations of the object. So, for example, in load balancing and fault-tolerance this can be used to code the list of potential hosts to which an invocation can be issued.

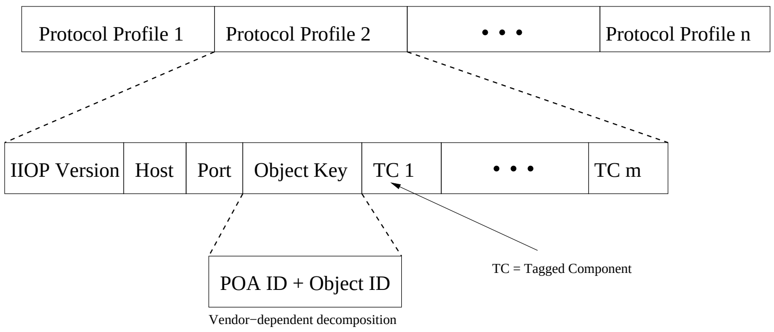

Figure 2. Interoperable Object Reference

In addition to the standard stub and skeleton, CORBA also defines a *Dynamic Invocation Interface (DII)* and a *Dynamic Skeleton Interface (DSI)*. The DII allows clients to invoke methods on objects whose interface is not known in advance. The client can obtain the interface specification from an *interface repository*. Similarly, DSI allows to incorporate generic objects that have no compile time interfaces. In the solutions we propose below, the interceptor reroutes requests to a generic local object that, depending on the specific case, may invoke the corresponding request on the original client. The DII is, thus, very instrumental in our solution, as it allows us to avoid interfering with the code generation process. Also, since we want our local object to be generic and transparent, it is registered by the interceptor under the target object's interface and invoked using DSI.

Note that naive usage of DII/DSI adds a non-negligible overhead. However, this overhead can be significantly reduced as follows: It is possible to preprocess the interface of the target object, e.g., by a preliminary scan of the interface repository, and dynamically construct "containers" for processing each type of request separately and efficiently [4].

Another useful feature of CORBA is the ability to define *policies*. Policies can be used to specify special behaviors. For example, policies are used in our implementation to indicate whether caching is needed or not, what type of load balancing protocol should be used (if any), maximum and expected completion time of operations, flow control parameters, etc. When an IOR is created, an IOR interceptor is called. The IOR interceptor can code the various policies as *tagged components* in the IOR.

2.2 Portable Interceptors

Portable interceptors are hooks into the ORB through which enhancements may be added to the ORB's core functionality [11]. Examples of such enhancements include monitoring, debugging, logging, authentication and encryption, fault-tolerance, load balancing, etc. Portable interceptors are allowed to delay a request or reply, and even generate their own requests, but they are not allowed to generate their own reply to an intercepted request, and the only way they can block a request or reply is by raising an exception. An interceptor can also be invoked due to an exception, and is allowed to alter the type of the exception, but not to block it. Portable interceptors can redirect a request, but only through the `ForwardRequest` exception. Also, portable interceptors are not allowed to alter the in and inout parameters of a request, nor the inout parameters, out parameters, or return value of a reply. In the case of Java, the interceptor cannot even read these values. Furthermore, a portable interceptor cannot modify the service context. This is quite different from the old interceptor standard, which essentially allows an interceptor to have full control over a request or reply it has intercepted.

There is a distinction in the proposed standard between client side interceptors and server side interceptors. In this work we are only interested in client side enhancements. However, some of our solutions rely on proxies, that appear as a server to the client, event though they only serve as a powerful middleman. Hence, we in fact utilize both client and server side interceptors, but logically, the enhancement only rely on interceptors that run in the client's ORB, which is why we refer to them as client-side enhancements.

The standard defines several interception points. Client side interception points include: *send_request*, *send_poll*, *receive_reply*, *receive_exception*, and *receive_other*; *send_request* is called when a request is sent; *send_poll* is called if the client polls to check the completion status of a pending request; *receive_reply* is called when a reply reaches the client's ORB; *receive_exception* is called upon an exception other than `ForwardRequest`, and *receive_other* in other events related to an invocation, and in particular on `ForwardRequest` exceptions and as a result of oneway invocations. Our solutions mainly acts on the *send_request* and *receive_reply* interception points. Server side corresponding interception points include: *receive_request_service_contexts*, *receive_request*, *send_reply*, *send_exception*, and *send_other*; interception point *receive_request_service_contexts* is invoked before the request is processed, and allows the interceptor to examine the request service contexts, *receive_request* is invoked just before the request is passed to the object implementation, *send_reply* is invoked right after the object has generated a reply, *send_exception* is invoked if the object or one of the previous interceptors raised an exception other than `ForwardRequest`, and *send_other* is invoked on `ForwardRequest` exceptions. We mostly use the *receive_request_service_contexts* interception point.

The standard allows for more than one interceptor to be registered with the ORB. If indeed several interceptors are registered, the ORB may invoke them in any arbitrary order. In particular, no order of invocation may be assumed between interceptors. However, a stack-like behavior is required, so that for every request/reply sequence, if interceptor A's interception point is invoked during client send before interceptor B's, then B's interception point is called before A's during client receive. The same rule applies for server interceptors.

Note that the standard also provides communications channels between client's code and its interceptors, between a client-side interceptor and a server interceptor, and between an implementation and its server-side interceptors, as depicted in Figure 3. Clients can write objects into *PICurrent*, which is an associative array. These fields are then visible to client-side interceptors when they are invoked. However, this is a unidirectional communication, in the sense that an interceptor cannot pass any information to clients. A client side interceptor can exchange information with a server side interceptor by using the *RequestServiceContext*, which is also an associative array. This, however, can be used for communication in both ways. Finally, a server side interceptor and an implementation can exchange objects using the PICurrent. However, unlike in the client side, on the server side this associative array can be used for bi-directional communication.

2.3 Client Redirection

In most of the solutions proposed in Section 3, client requests are rerouted using `ForwardRequest` exceptions to a local proxy object, and the actual functionality is provided by that object. This is based on the distinction in CORBA between the *target* vs. the *effective target* of a request. That is, an ORB that obtains an IOR for an object through some lookup mechanism, e.g., the naming service, initiates both the target and effective target to hold the IOR. If a *non-permanent* `ForwardRequest` exception is issued, this re-

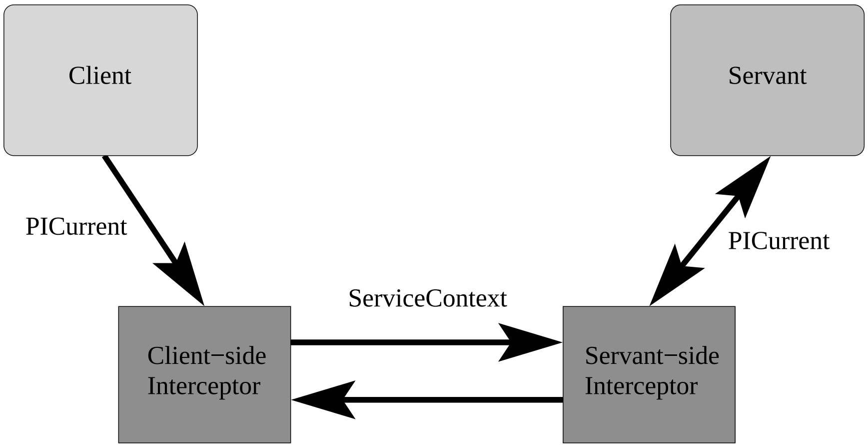

Figure 3. PICurrent and Service Context as control channels

programs the effective target to point to the newly obtained IOR, but the target still points to the initial IOR, while a *permanent* `ForwardRequest` exception changes both to hold the new IOR[2]. Requests are always issued to the effective IOR. In our case, this helps the interceptor to distinguish between requests that were already redirected and those that need to be redirected. Also, with the help of a server interceptor at the proxy and the use of PICurrent and ServiceRequestContext, the client side interceptor can pass the real target to the local proxy. The latter can use this information if it has to access the actual object implementation.

When a client passes an IOR to another client, only the real target IOR is passed. As a result, requests invoked on the IOR in the latter client have their effective target reset to the real target. This avoids creating a dependency between the latter client and the former client's proxy. Thus, only clients that have a proxy installed locally are affected by it. Also, fault tolerant implementations using a client proxy often require that a client depends only on a local proxy so that the proxy would not fail separately of the client, thereby interrupting the service to the still operating client.

[2]In CORBA 2.4 there is only one kind of `ForwardRequest` exception, which is equivalent to the non-permanent `ForwardRequest` in CORBA 2.3.

3 Client Side Enhancements Using Interceptors

In this section we outline several client side enhancements using interceptors. Due to space limitations, we only provide the details that are relevant to the use of interceptors here, while the full details will appear in the full version of this paper.

3.1 Caching

Caching is a useful way of reducing both the response time and network traffic, and is being used quite heavily in the Web and other mostly-read environments. In CORBA, we can maintain a cache of recently received replies, and use the replies stored in the cache whenever applicable. Of course, we also need a mechanism to issue the request to the real implementation of the object in case the reply is not found in the cache. In particular, we are interested in a solution that is general, and obeys the portable interceptors restrictions.

Since interceptors are not allowed to generate the reply to a request themselves, the architecture we propose includes a local caching object and an interceptor. Also, recall that the ORB maintains both the target and the effective target fields. Thus, when the interceptor is invoked with a request in which the effective target is different from the local caching object, it issues a non-permanent `ForwardRequest` exception. This exception results in reprogramming the effective target field to the IOR of the local caching object. Consequently, the client's ORB reissues the request, which is then routed to the local caching object.

The local caching object does a lookup to see if it can find a reply to the request. If a reply is found in the cache, a reply is generated. Otherwise, the caching object issues the request using the DII to the real object. Note that for this, the caching object must know the IOR of the real object. To obtain this IOR, note that the client-side interceptor has access to both the target and effective target IORs, and the target IOR does not change after a `ForwardRequest` exception. Thus, the client-side interceptor passes the IOR of the actual target in the service context to the server-side interceptor of the proxy, which then puts that information in PICurrent, where it is available to the proxy. When the reply arrives, it is placed in the cache, and a corresponding reply is returned to the client. Note that in order to be generic, the proxy itself must be invoked using the DSI mechanism. This however requires the proxy to use the interface repository to be able to learn the parameters of the requests and if necessary pass them correctly to the real object. From our experience, interface repository lookup is a slow remote operation. Therefore, we recommend pre-loading the invocation information for all the methods into a local structure upon startup and accessing only that structure during runtime.

There are still a few additional complications, related to the fact that interceptors are called for each invocation, including reissued invocations. Thus, the client-side interceptor must know not to issue a `ForwardRequest` exception to requests that are already routed to the proxy, to avoid an infinite loop. However, in theses requests the effective target is already pointing to the proxy, which indicates to the interceptor that no further redirection is needed. This is not enough though. The problem here is that when the proxy issues a request to the real object, e.g., due to a cache miss, the client-side interceptor is invoked again, and the effective target now includes the IOR of the real object. To prevent the interceptor from redirecting such requests, the proxy writes a special value in the PICurrent field. When the interceptor sees this field in PICurrent, it allows the request to go through as is. This problem can be entirely avoided by creating a separate unintercepted ORB in which the proxy is registered. This ORB can reside in the same process as the client's ORB. This way, the proxy's invocation does not pass through the client interceptor at all, saving this extra coordination. Yet, the effectiveness of such a solution depends on the ORB's implementation. A naive ORB implementation might make two ORBs that reside in the same process still communicate through low-level TCP. Although a local TCP connection is much faster than cross-net TCP, it may still prove considerably slower relative to communicating through a single ORB, since the former is at the OS level and the latter is at the application level.

Note that when cached objects are read only, it is sufficient to use the target and request ID to do the lookup for a matching reply, and hence it is possible to provide a generic solution. If cached objects are not read-only, we believe that an application dependent code must be used to do this lookup.

The last issue to worry about is co-existence with non-cacheable objects. Objects that can be cached must be registered with a Caching policy. When an IOR is created for such an object, the IOR interceptor turns the policy into an appropriate IOR tagged component. Caching is therefore enabled only to objects whose IOR includes the corresponding tagged component. The solution is also depicted in Figure 4.

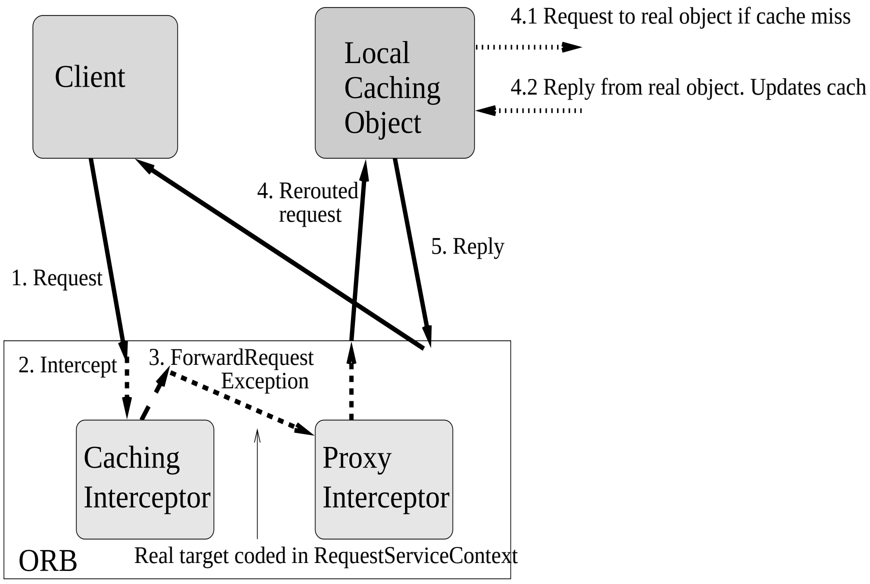

Figure 4. Caching implementation. Note that only the first request to an object is rerouted. All subsequent requests already have the local caching object as their effective target.

3.2 Load Balancing

Here, we are interested in a mechanism that redirects client's requests among a group of servers, providing an identical service, with the goal of keeping the load on each of the servers almost the same. This is especially useful when each request is computationally expensive, or when an extremely large number of short requests may be invoked concurrently by multiple clients.

The simplest approach here is to have the interceptor monitor the load on servers, and then redirect the requests to the least loaded server using the `ForwardRequest` exception mechanism. This naive approach is mainly viable when the rate by which the load on servers changes is slow compared to the speed of reinvocation. Note however that the interceptor is invoked also for rerouted requests when they are reissued. Thus, the interceptor needs a way to code the fact that a request has already been redirected once, to prevent a situation in which the same request is being rerouted over and over again if the load changes between consecutive reinvocations. Unfortunately, the current standard does not provide any means for a client interceptor to write anything that will be available to it when a request is reissued by the *same* client, and even the request ID of reissued requests may be different. The PI specification defines that the request scope PICurrent is the same for a request and its matching retries. However, the client interceptor has just read-only access to the PICurrent contents, forbidding it from differentiating between an original request and retries issued by the client. Our recommendation to OMG is to introduce a new type of request scope current field that can be written by a client interceptor and will be available if the request is reissued. This context should be inaccessible to client code to prevent the interceptor from affecting the client in methods other than those defined in the PI specification.

Note that FT-CORBA has addressed this problem by introducing the `FT_REQUEST` service context, which adds a unique identifier to a request and its retries [10]. Therefore, when using an FT-CORBA enabled ORB, the interceptor can use this field to remember requests it has already rerouted, thereby avoiding such infinite loop. On the other hand, this would force the interceptor to maintain possibly large tables of previous requests. Our recommendation would allow the interceptor to simply tag each request as "routed", thereby eliminating this bookkeeping.

Another drawback of the above method is that it requires the interceptor to generate a `ForwardRequest` exception for almost every new request. To solve both problems, we use a local load balancing proxy object. When the interceptor sees a request whose effective target is different than the local load balancing proxy, it generates a non-permanent `ForwardRequest` exception. This in turn programs the effective target to the IOR of the local load balancing proxy, but keeps the target field as the true object. Following this, the ORB reissues the request, but this time it goes directly to the local proxy. That is, the interceptor is still invoked for the reissued request, but ignores it in a similar fashion to what is being done for caching. The proxy then uses the DII to reissue the request to the server that seems to be the least loaded. When the reply returns to the proxy, the proxy generates a reply to the client. The general scheme is depicted in Figure 5. Compared with the naive method, as described above, this method has the advantage that an exception is only raised for the first request to a given object, and is feasible with the current standard. This comes at the expense of having to go through a proxy object.

There are many published papers about various techniques for achieving load balancing, e.g., [13, 15, 14]. Since in this paper we focus on the use of interceptors, we see no point in repeating all of them here. One such work that is also relevant to the next subsection

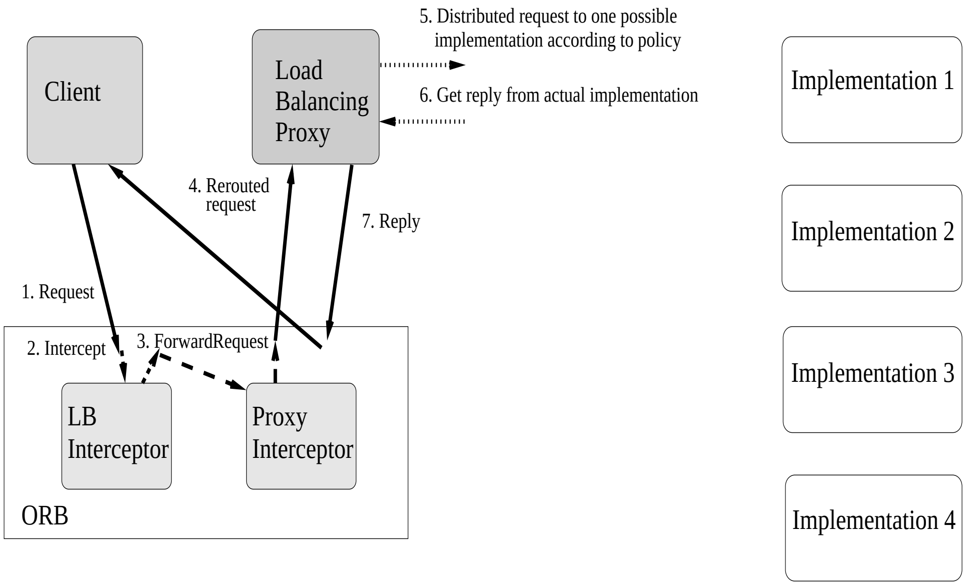

Figure 5. Load balancing implementation. Note that only the first request to an object is rerouted. All subsequent requests already have the local load balancing proxy as their effective target.

is the work of Friedman and Mosse [3]. In their proposal, the proxy remembers how many unanswered requests it had sent to each server, and sends the request to the server with the smallest number of unanswered requests. The nice thing about this approach is that it requires no communication, and yields near optimal results.

3.3 Reliable Soft Real-Time

Redundancy is the main tool for obtaining reliability. There are two principal forms of redundancy. In *resource redundancy*, a task is executed concurrently on several independent nodes. Thus, if one of these nodes fail, a timely reply will still be generated. In *time redundancy*, the assumption is that the typical execution time of a task is much shorter than its deadline. Thus, the task is started on one server. If there is no answer within the expected termination time, the task is reissued on another node. The latter approach is more efficient, and is more common in soft real-time systems [6].

Client side interceptors can be used to obtain time redundancy.[3] Note, however, that

[3]Obviously, some server side mechanism might be needed to filter duplicate requests. Also, there are

since an interceptor cannot block the flow of replies, the interceptor itself cannot be the one reissuing the request if a reply was not received after the expected service time. The reason is that the original reply might be simply late, e.g., due to an overload on the server, and since the interceptor cannot block any of the replies, the client will receive more than one reply. A violation of the required at most once semantics.

Thus, the solution here is to use a local RT proxy. The interceptor redirects requests to the local proxy using the `ForwardRequest` exception. When the ORB reissues the request, it is routed to the proxy. The local proxy then uses the DII to issue the request to one of the servers using some policy, e.g., primary-backup, round-robin, etc. If a reply is not received within the expected execution time, the proxy issues another request to another server, e.g., the backup or the next available server. The first reply that is received is returned to the client. Note that the proxy must remember the request until the deadline expires, so that if a redundant reply is received at the proxy, it can filter it out.

The messaging standard of CORBA [9] defines a policy for defining the maximal round-trip time of an invocation. It is used by the local RT proxy to determine if a reply came in too late, or that there is no point in reissuing a request. There are two options for getting this information into the local RT proxy: Given an IOR, the RT proxy can ask about the associated policies for this IOR using the standard interface. Alternatively, the interceptor can code the information obtained from the IOR tagged components in the current field, which is also accessible by the RT proxy. In a similar fashion, we introduce a new policy, ExpectedRoundTrip, which specifies the expected round-trip time for an invocation. This indicates to the local RT proxy when it needs to reissue a request. Also, the interceptor is designed such that only requests in which the IOR includes the corresponding tagged component are rerouted to the local RT proxy.

3.4 Flow Control

In flow control, the goal is to regulate the flow of traffic from the client to the server. This is typically done in accordance with either rate based methods, e.g., leaky bucket, or credit

many more issues that need to be dealt with in order to obtain true (or hard) real-time, as mentioned, e.g., in [12]. In this work we only look at what can be done in a client side interceptor, in order to obtain soft real-time, e.g., for systems like the ones explored in [1].

based methods, e.g., token bucket [5]. In rate based flow control, the idea is to enforce a maximum on the rate of requests/messages a client may generate. If the client attempts to generate requests/messages faster than the agreed upon rate, then his requests/messages are being held up by the system and only sent in the correct spacing. In credit based schemes, the idea is to limit the maximum number of requests/messages that a client can generate in a burst, without getting a reply. If the client attempts to generate more requests/messages than it is allowed, the system buffers them until enough replies (credit) were received.

In this case the entire work can be done by the interceptor, since interceptors are allowed to delay requests. If the client continues to send too many messages, the interceptor can proactively invoke a `TooFast` exception to the client, alerting the client to slow down.

4 Discussion

Interceptors are a powerful tool for enhancing the functionality of ORBs without modifying their code. The newly proposed standard of portable interceptor provides rules that are designated to keep interceptor code portable and interoperable, but in return imposes some non trivial restrictions on their programming model. In this work we have explored ways to provide several desired client side enhancements using portable interceptors, namely caching, load balancing, reliable soft real-time, and flow control. In all of these examples, we have demonstrated that by using a proxy object, to which the interceptor reroutes requests, we can provide the desired functionality. Moreover, our solution is completely standard, and relies on the ability to add policies and tagging to IORs, and utilize the PICurrent channel and RequestServiceContext that are associated with requests and replies. Furthermore, we have exploited the dynamic invocation capabilities of CORBA to make this proxy object portable and general.

To complete this study, it would be interesting to look at how to implement similar functionality to the ones discussed here using server side enhancements, as well as to look at other possible ORB enhancements, e.g., in [2, 7]. Perhaps an even more appealing result would be to provide a scheme that can formally verify that interceptors provide the functionality they intend. Specifically, a potential problem in using interceptors is that one interceptor might cancel or alter the functionality of other interceptors. A framework that could alert the user when such conflicts occur would be highly desired.

Finally, we would like to reiterate our recommendation that a new form of context will be defined for the use of client-side interceptors. The standard should allow a client-side interceptor to write objects in this context, and have them available if a request is reissued due to an exception it raised. As discussed, it would greatly simplify solutions to problems like load-balancing, caching, and fault-tolerance. Yet, given that this new context will only be available to the interceptor, it will not affect the portability of the standard.

Acknowledgements: We wish to thank the anonymous referees for the insightful comments that greatly improved our paper.

References

[1] R. Friedman and K. Birman. Using Group Communication Technology to Develop a Reliable and Scalable Distributed IN Coprocessor. In *Proc. of the TINA 96 Conference*, pages 25–41, September 1996.

[2] R. Friedman and E. Hadad. FTS: A High-Performance CORBA Fault-Tolerance Service. Submitted for publication, 2001.

[3] R. Friedman and D. Mosse. Load-Balancing Schemes for High-Throughput Distributed Fault-Tolerant Servers. In *Proc. of the 16th Symposium on Reliable Distributed Systems*, October 1997.

[4] E. Hadad. FTS: A High-Performance CORBA Fault-Tolerance Service. Technical report, Department of Computer Science, Technion, 2001.

[5] S. Keshav. *An Engineering Approach to Computer Networking*. Addison Wesley, April 1997.

[6] H. Kopetz and P. Verissimo. Real-Time and Dependability Concepts. In S. Mullender, editor, *Distributed Systems*, chapter 16. Addison Wesley, 1993.

[7] C. Marchetti, A. Virgillito, and R. Baldoni. Design of an Interoperable FT-CORBA Compliant Infrastructure. In *Proc. European Research Seminar on Advances in Distributed Systems (ERSADS)*, May 2001.

[8] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Using Interceptors to Enhance CORBA. *IEEE Computer*, 32(7):62–68, July 1999.

[9] OMG. CORBA/IIOP Specification 2.4.2. formal/2001-02-33.

[10] OMG. Fault Tolerant CORBA specification, v1.0. ptc/00-04-04.

[11] OMG. Portable Interceptors. orbos/01-03-04.

[12] D. C. Schmidt, D. L. Levine, and S. Mungee. The Design of the TAO Real-Time Object Request Broker. *Computer Communications*, 21(4):294–324, April 1998. Special Issue on Building Quality of Service into Distributed System.

[13] N. G. Shivaratri, P. Krueger, and M. Singhal. Load Distributing for Locally Distributed Systems. *IEEE Computer*, 25(12):33–44, December 1992.

[14] Y.T. Wang and R. J. Morris. Load Sharing in Distributed Systems. *IEEE Transactions on Computers*, C-34(3):204–217, March 1985.

[15] M. H. Willebeek-LeMair and A. P. Reeves. Strategies for Dynamic Load Balancing on Highly Parallel Computers. *IEEE Transactions on Parallel and Distributed systems*, 4(9):979–993, September 1993.