

Portable Interceptors

Joint Initial Submission

***Expersoft Corporation
GMD Fokus
Objective Interface Systems, Inc.
Object-Oriented Concepts, Inc.***

In collaboration with:

***Adiron LLC
Humboldt-Universität zu Berlin***

Supported by:

***Deutsche Telekom AG
KPN Research***

April 26, 1999
OMG Document orbos/99-04-10

Copyright 1999 by Adiron LLC
Copyright 1999 by Expersoft Corporation
Copyright 1999 by GMD Fokus
Copyright 1999 by Object-Oriented Concepts, Inc.
Copyright 1999 by Objective Interface Systems, Inc.
Copyright 1999 by Humboldt Universität zu Berlin
All rights reserved.

The companies listed above hereby grant to the Object Management Group, Inc. (OMG) and OMG members, permission to copy this document for the purpose of evaluating the technology contained herein during the technology selection process by the appropriate OMG task force. Distribution to anyone not a member of the Object Management Group or for any purpose other than technology evaluation is prohibited. The material in this document is submitted to the OMG for evaluation. Submission of this document does not represent a commitment to implement any portion of this specification in the products of the submitters.

WHILE THE INFORMATION IN THIS PUBLICATION IS BELIEVED TO BE ACCURATE, THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

The companies listed above shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material. The information contained in this document is subject to change without notice. This document contains information which is protected by copyright. All Rights Reserved.

Except as otherwise provided herein, no part of this work may be reproduced or used in any form or by any means—graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems— without the permission of one of the copyright owners. All copies of this document must include the copyright and other information contained on this page.

The copyright owners grant member companies of the OMG permission to make a limited number of copies of this document (up to fifty copies) for their internal use as part of the OMG evaluation process.

RESTRICTED RIGHTS LEGEND. Use, duplication, or disclosure by government is subject to restrictions as set forth in subdivision (c) (1) (ii) of the Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013.

CORBA and Object Request Broker are trademarks of Object Management Group.
OMG is a trademark of Object Management Group.

Table of Contents

Table of Contents	5
Introduction	7
Submitting Companies	7
Status of this Document	7
Guide to the Submission	7
Conventions	8
Submission Contact Points	8
Proof of Concept	9
Response to RFP Requirements	9
General Requirements	9
Mandatory Requirements	11
Optional Requirements	12
General Issues	13
The new module Dynamic	13
Locality Constrained Interfaces	13
The Module Dynamic	15
General Definitions	15
The Request Data Structure	15
The Dynamic Invocation Interface	16
The Dynamic Skeleton Interface	17
Usage Examples	18
Operation Invocation using the new DII	19
Operation Implementation using the new DSI	20
Request Level Interceptors.....	21
Interceptor Requests	21
Implementing Interceptors	23
Usage Examples	24

Server-Side Request Level Interceptor	24
Client-Side Request Level Interceptor	25
Message Level Interceptors.....	27
Representation of GIOP Messages	27
Exception and Type Definitions	28
Representation of Service Contexts	29
A Common GIOP Message Interface	30
Representation of a GIOP Request Message	31
Representation of a GIOP Reply Message	33
Representation of a GIOP CancelRequest Message	34
Representation of a GIOP LocateRequest Message	34
Representation of a GIOP LocateReply Message	35
Representation of a GIOP Fragment Message	36
The Message Level Interceptor	36
Definition of Message Level Interceptor	36
Example: A User-supplied Load Balancing Interceptor	39
Network Level Interceptors.....	43
IOR Management Interceptors	45
Management of Interceptors	47
Requirements for Management of Interceptors	47
Bootstrapping the Registry for Interceptors	48
The Binding Object	48
The Registry for Request Level Interceptors	50
Synchronous and Asynchronous Invocations	53
The Registry for Message Level Interceptors	53
Network Level Interceptors	55
The Registry for IOR Management Interceptors	55
Example for Registering an Request Level Interceptor	56
UML Diagram for the Interceptor Registry	57
Conformance Criteria.....	59
Conformance Points	59
Changes to CORBA	61
Changes to CORBA Core	61
Consolidated IDL	63
Module Dynamic	63
Request Level Interceptors	65
Message Level Interceptors	67
Interceptor Management	70

Introduction

1

1.1 Submitting Companies

The following companies are pleased to jointly submit this specification in response to the Portable Interceptors RFP, document number orbos/98-09-11:

- Expersoft Corporation
- GMD Fokus
- Objective Interface Systems, Inc.
- Object-Oriented Concepts, Inc.

1.2 Status of this Document

This document is an Initial Submission in response to the Portable Interceptors RFP for the May, 1999 Technical Committee Meeting in Tokyo, Japan.

1.3 Guide to the Submission

Chapter 1 provides contact information and a guide to this submission.

Chapter 2 introduces the module Dynamic.

Chapter 3 defines Request Level Interceptors and gives a usage outline.

Chapter 4 introduces interfaces for the representation of GIOP messages and defines Message Level Interceptors.

Chapter 5 introduces Network Level Interceptors.

Chapter 6 discusses interceptors for object reference management.

Chapter 7 provides a specification for the management of interceptors.

Chapter 8 provides the complete IDL specification.

1.4 Conventions

IDL appears using this font.

Concrete programming language (Java, C++, etc.) code appears using this font.

1.5 Submission Contact Points

Polar Humenn
Adiron, LLC
Syracuse, NY 13244-4100
USA
phone: (315) 443 3171
e-mail: polar@adiron.com

Shahzad Aslam-Mir
Expersoft Corporation
5825 Oberlin Dr
San Diego, CA 92121
USA
phone: (619) 824-4128
e-mail: sam@expersoft.com

Linda Strick
GMD Fokus
Kaiserin-Augusta-Allee 31
10589 Berlin
Germany
phone: +49-30-3463-7224
e-mail: strick@fokus.gmd.de

Marc Laukien
Object-Oriented Concepts, Inc.
44 Manning Rd.
Billerica, MA 01821
USA
phone: (978) 439 9285x245
e-mail: ml@ooc.com

Victor Giddings
Objective Interface Systems, Inc.
1892 Preston White Drive
Reston, Virginia, 20191-5448
USA
phone: (703) 295 6500
e-mail: victor.giddings@ois.com

Olaf Kath
Humboldt-Universitaet zu Berlin
Institut fuer Informatik
Rudower Chaussee 5
12489 Berlin

Germany
 phone: +49-30-2093-3116
 e-mail: kath@informatik.hu-berlin.de

1.6 Proof of Concept

This specification has completed the design phase and is in the process of being prototyped.

1.7 Response to RFP Requirements

1.7.1 General Requirements

- **Proposals shall express interfaces in OMG IDL. Proposals should follow accepted OMG IDL and CORBA programming style. The correctness of the IDL shall be verified using at least one IDL compiler (and preferably more than one). In addition to IDL quoted in the text of the submission, all the IDL associated with the proposal shall be supplied to OMG in compiler-readable form.**

All Interfaces defined in the submission are expressed in OMG IDL. The complete IDL specification is supplied in compiler-readable form.

- **Proposals shall specify operation behaviour, sequencing, and side-effects (if any).**

All operations and their semantics are explained in detail. Additionally, use cases illustrate the usage of operations and their sequencing.

- **Proposals shall be precise and functionally complete. There should be no implied or hidden interfaces, operations, or functions required to enable an implementation of the proposed specification.**

The submission claims to be functionally complete.

- **Proposals shall clearly distinguish mandatory interfaces and other specification elements that all implementations must support from those that may be optionally supported.**

All specification elements are denoted to be mandatory or optional.

- **Proposals shall reuse existing OMG specifications including CORBA, CORBAServices, and CORBAfacilities in preference to defining new interfaces to perform similar functions.**

The specification is based on CORBA 2.3. It includes a replacement for DII and DSI of CORBA 2.3.

- **Proposals shall justify and fully specify any changes or extensions required to existing OMG specifications. This includes changes and extensions to CORBA inter-ORB protocols necessary to support interoperability. In general, OMG favours upwards compatible proposals that minimize changes and extensions to existing OMG specifications.**

This specification replaces the Dynamic Invocation Interface and Dynamic Skeleton Interface specified in CORBA 2.3 with a unified facility. This is necessary to allow for daisy-chaining of interceptors.

All changes to CORBA and CORBA Services are identified in a separate chapter.

- **Proposals shall factor out functions that could be used in different contexts and specify their interfaces separately. Such minimality fosters re-use and avoids functional duplication.**

The module Dynamic is intended to be used not only by the Portable Interceptors, but also as a general replacement for the current DII and DSI of CORBA 2.3.

- **Proposals shall use or depend on other interface specifications only where it is actually necessary. While re-use of existing interfaces to avoid duplication will be encouraged, proposals should avoid gratuitous use.**

This specification replaces the specification of Dynamic Invocation Interface and Dynamic Skeleton Interface of CORBA 2.3, and makes use of that replacement. It does not use any other interface specifications.

- **Proposals shall specify interfaces that are compatible and can be used with existing OMG specifications. Separate functions doing separate jobs should be capable of being used together where it makes sense for them to do so.**

This specification specifies interfaces that are compatible with existing OMG specifications. Separate functions doing separate jobs are capable of being used together.

- **Proposals shall preserve maximum implementation flexibility. Implementation descriptions should not be included, however proposals may specify constraints on object behaviour that implementations need to take into account over and above those defined by the interface semantics.**

This specification does not restrict implementation flexibility. No implementation descriptions are included.

- **Proposals shall allow independent implementations that are substitutable and interoperable. An implementation should be replaceable by an alternative implementation without requiring changes to any client.**

As the specification claims to be complete and is based on CORBA, a specific implementation is replaceable by alternative implementations without changes to the client.

- **Proposals shall be compatible with the architecture for system distribution defined in ISO/IEC 10746, Reference Model of Open Distributed Processing (ODP). Where such compatibility is not achieved, the response to the RFP must include reasons why compatibility is not appropriate and an outline of any plans to achieve such compatibility in the future.**

The Proposal is fully compatible with ODP. In particular, it defines a mapping of the ODP term Interceptor (ISO/IEC 10746 part 3, 8.1.11) into CORBA.

- **In order to demonstrate that the service or facility proposed in response to this RFP, can be made secure in environments requiring security, answers to the following questions shall be provided:**

- **What, if any, are the security sensitive objects that are introduced by the proposal?**

- Which accesses to security-sensitive objects must be subject to security policy control?
- Does the proposed service or facility need to be security aware?
- What CORBAsecurity level and options are required to protect an implementation of the proposal? In answer to this question, a reasonably complete description of how the facilities provided by the level and options (e.g. authentication, audit, authorization, message protection etc.) are used to protect access to the sensitive objects introduced by the proposal shall be provided.
- What default policies should be applied to the security sensitive objects introduced by the proposal?
- Of what security considerations must the implementers of your proposal be aware?

All interceptor objects are security-sensitive and are therefore subject to control by security policies. Because interceptor objects are locality constrained, security is restricted to locality, too. The specification presented here allows the implementation of a security service.

1.7.2 Mandatory Requirements

- **A submission shall specify two kinds of interceptors: system and user.**

This specification does not distinguish between system and user interceptors. No characteristics justifying the introduction of these two different kinds of interceptors could be recognized. Instead, interceptors are distinguished by the information they process. The four kinds of interceptors are Request Level Interceptors, Message Level Interceptors, Network Level Interceptors and IOR Management Interceptors. The specifications for Network Level Interceptors and IOR Management Interceptors are still under development, and will be provided in a revised submission.

- **A submission shall provide a complete architectural model of interceptors in ORB and Object Adapter processing. This includes what flexibility an ORB has in implementing interceptors, how the ORB will handle errors in interceptor processing, and the legitimate actions of an interceptor (including what kind of invocations an interceptor can perform). A submission shall describe under what conditions and how the ORB deals with recursive calls, whether generated locally or remotely. Allowable flexibility, error handling, and actions may be different for different kinds of interceptors.**

This description will be provided in a revised submission.

- **System interceptors for ORB Services shall include Request-Related and IOR-management interceptors. For these system interceptors, a submission shall provide interfaces so that an interceptor can set and query GIOP service contexts (for request-related system interceptors) and IOR profile information (for IOR-management system interceptors). The submission shall specify methods of limiting the invocation of such operations to system interceptors.**

Because of the lack of system interceptors in this specification, there are no special limits for operation invocations.

- **A submission shall specify a mechanism for user interceptors at request-level and at additional points in ORB processing. The mechanism need not be the same as system interceptors at the request-level.**

There is no distinction between user interceptors and system interceptors.

- **A submission shall provide for multiple interceptors (system and user) to be called at each point they are applicable and shall specify how the multiple interceptors are called (for example, each called serially by the ORB or daisy-chained).**

This specification provides for multiple daisy-chained Request Level Interceptors and Message Level Interceptors.

- **A submission's system interceptor model shall be detailed enough that Security and Transactions services could be reasonably implemented using it. An analysis of how each of these services could use system interceptors is required. The analysis will not be normative for those services, but must be detailed enough to show the applicability of the model. The analysis shall include a specification of what information a Security or Transaction Service needs at various points and how the system interceptor model provides that information.**

This specification allows the implementation of a security service and a transaction service.

- **To meet the negotiation part of certain security mechanisms, a submission shall specify that at least some system, request-related interceptors must be capable of holding a request in abeyance while it performs communication of its own, perhaps at a lower protocol level. For example, a client-end, message-level, security interceptor might do handshaking with a target before issuing a request using the negotiated security.**

This requirement is achieved through blocking interceptors. In case of the example above, the interceptor performs its handshaking with the target and issues the request subsequently.

- **A submission shall specify administrative interfaces including, at a minimum, registering and unregistering interceptors, both system and user.**

The submission specifies several administrative interfaces for registering and unregistering interceptors. Each of them is explained in detail. Additionally, a static UML diagram is provided and use cases facilitate understanding.

1.7.3 Optional Requirements

None.

1.8 General Issues

1.8.1 The new module *Dynamic*

It is difficult to daisy-chain requests using the current DII and DSI. That is, it's difficult to forward a DSI **CORBA::ServerRequest** using the DII, since the DII uses a different interface, **CORBA::Request**. This requires an inefficient and cumbersome conversion between the two request forms.

The request level interceptors in this specification are based on DII/DSI mechanisms, and rely heavily on daisy-chaining. Therefore the current DII/DSI is not appropriate. Rather than modifying the current DII/DSI, the submitters propose a replacement in the form of the new module **Dynamic**. If adopted, the old DII/DSI should be deprecated.¹

Aside from the ability to easily daisy-chain requests, the proposed new DII/DSI offers many other advantages:

- *Separate IDL module*: The new DII/DSI is defined in a separate IDL module, **Dynamic**, which separates it clearly from other parts of CORBA.
- *No DII/DSI operations in **CORBA::ORB** or **CORBA::Object***: The new module **Dynamic** is self-contained, therefore no DII/DSI operations are needed in **CORBA::ORB** or **CORBA::Object**.
- *No PIDL*: The new DII/DSI uses standard IDL with locality-constrained interfaces.
- *No special language mapping rules*: Language mappings, like the IDL-to-C++ mapping, do not need to provide special mapping rules. Instead, the standard language mapping rules can be used.
- *No pseudo objects*: **CORBA::NamedValue** and **CORBA::NVList** are no longer required.
- *Efficient*: No unnecessary data copying is required.
- *Compact*: The IDL for the module **Dynamic** consists of just a few definitions.
- *Can be implemented as a wrapper*: The new DII/DSI can easily be implemented as a wrapper for the old DII/DSI (or vice versa).

1.8.2 Locality Constrained Interfaces

Unless otherwise noted, all interfaces discussed in this submission are locality-constrained.

1. Implementations of CORBA can certainly continue to support the DII and DSI as specified in CORBA 2; this specification does not invalidate the existing API. It is possible to implement the existing DII with the proposed interfaces; the only restriction is that an application relying on parameter names instead of parameter positions when constructing an NVList will need external run-time information about parameter names, e.g., from the IFR.

In contrast to other specifications, this proposal contains locality-constrained interfaces that are intended to be implemented by user code. This raises the issue of how such locality-constrained interfaces are implemented.

The submitters feel that it's not appropriate to implement locality-constrained interfaces like regular interfaces. Implementing locality-constrained interfaces like regular interfaces would require that such objects need to be registered with the POA, must support dynamic invocation, need TypeCode constants, **any** inserters and extractors, and everything else that's required for regular CORBA objects. This would incur an unreasonable overhead, both in terms of performance and code size.

The submitters therefore suggest simplified mapping rules for locality-constrained interfaces. The concrete mappings will have to be specified by the various language mappings, and are out of the scope of this submission.

For the C++ examples given in this submission, it is assumed that each locality-constrained IDL interface maps to a C++ class with the same name, using the same mapping rules as for regular interfaces. However, in contrast to regular interfaces, locality-constrained interfaces are implemented by directly deriving an implementation class from the class the interface maps to. Note that this implies that the lifetime of a reference to a locality-constrained interface is directly coupled to the lifetime of the locality-constrained object, because a reference is effectively implemented as a C++ pointer to the implementation object.

This chapter describes the module **Dynamic**. This module contains interfaces for the Dynamic Invocation Interface (DII) and Dynamic Skeleton Interface (DSI).

Unless otherwise noted, all IDL discussed in this chapter is located in the module **Dynamic**.

2.1 General Definitions

The module **Dynamic** contains the following general definitions:

```
struct Parameter
{
    any value;
    CORBA::ParameterMode mode;
};

typedef sequence<Parameter> ParameterList;
typedef sequence<CORBA::TypeCode> TypeCodeList;
```

Parameter holds the value and type of a parameter as an **any**, along with the parameter mode, which may be either **CORBA::PARAM_IN**, **CORBA::PARAM_OUT** or **CORBA::PARAM_INOUT**.

ParameterList represents all parameters of an operation. **TypeCodeList** is used to specify the types of all user-defined exceptions which an operation may raise.

2.2 The Request Data Structure

The structure **RequestData** contains all dynamic data for both DII and DSI requests. Since **RequestData** is the same for both the DII and DSI, requests can be daisy-chained easily.

```
struct RequestData
{
    ParameterList parameters;
    any result;
    TypeCodeList exceptions;
    any except;
    CORBA::Context ctx;
};
```

parameters contains all parameters for a particular operation invocation. **result** holds the operation result as an **any**. **exceptions** contains the types for all user-defined exceptions that may be raised by the operation. **ctx** represents the operation's context.

2.3 *The Dynamic Invocation Interface*

The DII allows dynamic creation and invocation of requests. A client using the DII to send a request to an object obtains the same semantics as a client using the operation stub generated from the type specification.

A request consists of an object reference, an operation, and a list of parameters. The ORB applies the implementation-hiding (encapsulation) principle to requests.

Parameters supplied to a request may be subject to run-time type checking upon request invocation. Parameters must be supplied in the same order as the parameters defined for the operation.

The DII consists of two interfaces: **ClientRequest**, which is used to send requests using the DII, and **ClientRequestFactory**, which serves as a factory for **ClientRequest** objects.

```
interface ClientRequest
{
    void invoke(inout RequestData data);
    void send(inout RequestData data);
    void send_oneway(inout RequestData data);
    void get_response(inout RequestData data)
        raises(CORBA::WrongTransaction);
    boolean poll_response(inout RequestData data)
        raises(CORBA::WrongTransaction);
    void destroy();
};
```

```
interface ClientRequestFactory
{
    ClientRequest create_request(in CORBA::Object target,
                               in CORBA::Identifier operation);
};
```

Regular twoway requests can be sent with **invoke**, or alternatively with either **send** and **get_response**, or **send** and **poll_response**. Oneway requests must be sent with **send_oneway**.

If twoway request are sent with **send**, and later the response is retrieved with **get_response** or **poll_response**, the same **RequestData** struct must be passed as argument to both operation calls, and this **RequestData** struct may not be changed until the request is completed.

Prior to an operation invocation, the types and values for all **in** and **inout** parameters must be set in the **RequestData** structure which is being passed as argument. The same applies to the operation's context, if there is any.

For **out** parameters and the return value, only the types must be set in the corresponding **any**. Furthermore, all user exception types the operation is allowed to raise must be added to the **exceptions** member.

destroy destroys the **ClientRequest** object.

New **ClientRequest** objects are created using **ClientRequestFactory**. The singleton **ClientRequestFactory** can be resolved by calling the ORB operation **resolve_initial_references**, with "ClientRequestFactory" as the argument.

To create a new **ClientRequest** object with **create_request**, the target object and the operation name must be specified. The operation name is the same operation identifier that is specified in the OMG IDL definition for this operation. In the case of attributes, it is the name as constructed following the rules specified in XXX.

The implicit object reference operations **non_existent**, **is_a** and **get_interface** may be invoked using the DII. No other implicit object reference operations may be invoked via the DII.

To create a request for any one of these allowed implicit object reference operations, **create_request** must be passed the name of the operation with a "_" prepended, in the parameter "operation". For example, to create a dynamic invocation request for **is_a**, the name passed to **create_request** must be "_is_a". If the name of an implicit operation that is not invocable through the DII is passed to **create_request** with a "_" prepended, **create_request** shall raise a **BAD_PARAM** exception. For example, passing "_is_equivalent" to **create_request** as the operation parameter will cause a **BAD_PARAM** exception to be raised.

2.4 The Dynamic Skeleton Interface

The new DSI consists of the interface **ServerRequest**, which provides information about the current request, and the interface **Server**, which is implemented by servants.

Note – There is currently an open issue regarding whether **Server** can follow the regular implementation rules for locality-constrained interfaces.

```
interface ServerRequest
{
    readonly attribute CORBA::Identifier operation;
};

interface Server
{
    void get_types(in ServerRequest request, inout RequestData data);
    void invoke(in ServerRequest request, inout RequestData data);
};
```

operation returns the current operation name.

The responsibility of **get_types** is to fill in type information for **in** and **inout** parameters into the **RequestData** argument. **get_types** is invoked before any request level interceptor is called.

Note – Must **get_types** also fill in the **exceptions** member?

invoke must implement all operations of all interfaces the servant implements. **invoke** is called after all interceptors have been called.

Upon return from **invoke**, either the return value and all **out** and **inout** parameters, or an exception, must have been placed in the appropriate members of the **RequestData** argument.

2.5 Usage Examples

The examples demonstrate how the operation **f** in the following IDL can be invoked using the new DII and implemented using the new DSI.

```
exception E { };

interface I
{
    long f(in double d, out char c) raises(E);
};
```

2.5.1 Operation Invocation using the new DII

```

// Get the ClientRequest factory
CORBA::Object_var obj = orb ->
    resolve_initial_references("ClientRequestFactory");
Dynamic::DII_var reqFac =
    Dynamic::ClientRequestFactory::_narrow(obj);

// Create the request data
Dynamic::RequestData_var data = new Dynamic::RequestData;
data -> parameters.length(2);
data -> parameters[0].mode = CORBA::PARAM_IN;
data -> parameters[0].value <= 3.14;
data -> parameters[1].mode = CORBA::PARAM_OUT;
data -> parameters[1].value <= CORBA::Any::from_char(0);
data -> result <= (CORBA::Long)0;
data -> exceptions.length(1);
data -> exceptions[0] = CORBA::TypeCode::_duplicate(_tc_E);

// Get an I object somehow
I_var i = ...

// Invoke the request
Dynamic::ClientRequest_var req =
    reqFac -> create_request(i, "f");
req -> invoke(data.inout());

// Get the result and out parameter
CORBA::Long result;
data -> result >= result;
CORBA::Char outArg;
data -> parameters[1].value >= CORBA::Any::to_char(outArg);

// Print the result and out parameter
cout << "Return value is " << result << endl;
cout << "The out argument is " << outArg << endl;

// Destroy the request
req -> destroy();

```

2.5.2 Operation Implementation using the new DSI

```

void
MyServantImpl::get_types(Dynamic::ServerRequest_ptr req,
                        Dynamic::RequestData*& data)
{
    // Get the operation name
    String_var op = req -> operation();

    // Check whether we know the operation
    if(strcmp(op, "f") == 0)
    {
        // Set the TypeCode for the in parameter
        data -> parameters.length(2);
        data -> parameters[0].value <= (CORBA::Double)0;

        data -> exceptions.length(1);
        data -> exceptions[0] =
            CORBA::TypeCode::_duplicate(_tc_E);
    }
}

void
MyServantImpl::invoke(Dynamic::ServerRequest_ptr req,
                    Dynamic::RequestData*& data)
{
    // Get the operation name
    String_var op = req -> operation();

    // Check whether we know the operation
    if(strcmp(op, "f") == 0)
    {
        // Set the return value and the out parameter
        data -> parameters[1].value <=
            CORBA::Any::from_char('A');
        data -> result <= (CORBA::Long)123;

        // That's it!
        return;
    }

    // Oops! Unknown operation!
    data -> except <= new CORBA::BAD_OPERATION();
}

```

Request Level Interceptors

3

This chapter describes request level interceptors. Request level interceptors are based on the new module **Dynamic**, and reuse the **Dynamic::RequestData** structure.

Unless otherwise noted, all IDL discussed in this chapter is located in the module **POI**.

3.1 Interceptor Requests

The interfaces **ServerInterceptorRequest** and **ClientInterceptorRequest** are defined for request level interceptors, and are analogous to the **ServerRequest** and **ClientRequest** interfaces in the module **Dynamic**.

```

exception OperationOverrideFailure { string reason; };
exception TargetOverrideFailure { string reason; };

interface ServerInterceptorRequest
{
    readonly attribute PortableServer::Servant orig_target;
    readonly attribute CORBA::Identifier orig_operation;
    readonly attribute PortableServer::Servant target;
    readonly attribute CORBA::Identifier operation;

    void next(inout Dynamic::RequestData data);
    void next_override(inout Dynamic::RequestData data,
                      in PortableServer::Servant target,
                      in CORBA::Identifier operation)
        raises(TargetOverrideFailure, OperationOverrideFailure);
};

interface ClientInterceptorRequest
{
    readonly attribute CORBA::Object orig_target;
    readonly attribute CORBA::Identifier orig_operation;
    readonly attribute CORBA::Object target;
    readonly attribute CORBA::Identifier operation;

    void next(inout Dynamic::RequestData data);
    void next_override(inout Dynamic::RequestData data,
                      in CORBA::Object target,
                      in CORBA::Identifier operation)
        raises(TargetOverrideFailure, OperationOverrideFailure);
};

```

The primary difference between these two interfaces is the “target,” which is a servant for **ServerInterceptorRequest** and an object reference for **ClientInterceptorRequest**.

The **target** and **operation** attributes return the target and the operation that the request is directed to. Since interceptors may override the servant and the operation name, there are also two attributes **orig_target** and **orig_operation**, which return the original target and operation name, respectively.

The **next** operations call the next interceptor, if one exists. Otherwise, **next** for **ServerInterceptorRequest** calls the operation implementation on the servant, and **next** for **ClientInterceptorRequest** invokes the request on the server.

The **next_override** operation works like the **next** operation, but it allows the interceptor to override the target and/or the operation name. The ORB may raise the **OperationOverrideFailure** or **TargetOverrideFailure**, such as when a policy explicitly set on the **orig_target** object disallows the rebinding of the request to a different target, e.g., a violation of the **CORBA::RebindPolicy** set on the **orig_target** object reference.

Note – The submitters are still in discussion about whether to add the operation **next_override_service_context**, which would allow the service context to be overridden.

On the client side, new target object references can be created using the **set_policy_overrides** operation on the current target object reference. In this case, the new target object reference still refers to the object referred to by the previous object reference. One such example would be the use of security policies, such as **Security::SecQOPPolicy**, which stipulate that the invocation will be made under the directed quality of protection from the ORB Security Service.

The use of the **next** and **next_override** operations is the basic mechanism in which control is transferred from one interceptor to another interceptor (i.e., “daisy-chaining”). The **next** and **next_override** operations partition an interceptor into a pre-invocation part and a post-invocation part. Everything that’s executed before the **next/next_override** statement is pre-invocation interceptor code. Everything after **next/next_override** is post-invocation code.

Furthermore, an interceptor implementor may choose to not call **next** or **next_override** at all. In that case, the interceptor must return the request result by filling in the **RequestData** struct directly.

3.2 Implementing Interceptors

Request level interceptors on the server-side must implement the locality-constrained interface **ServerInterceptor**. For the client-side, **ClientInterceptor** must be implemented.

interface ServerInterceptor

```
{
    void invoke(in ServerInterceptorRequest req,
               inout Dynamic::RequestData data);
};
```

interface ClientInterceptor

```
{
    void invoke(in ClientInterceptorRequest req,
               inout Dynamic::RequestData data);
    void send_oneway(in ClientInterceptorRequest req,
                   inout Dynamic::RequestData data);
    void send(in ClientInterceptorRequest req,
             inout Dynamic::RequestData data);
    void get_response(in ClientInterceptorRequest req,
                    inout Dynamic::RequestData data)
        raises(CORBA::WrongTransaction);
    boolean poll_response(in ClientInterceptorRequest req,
                       inout Dynamic::RequestData data)
        raises(CORBA::WrongTransaction);
};
```

The **invoke** operation must be implemented for both server-side and client-side request level interceptors. Furthermore, client-side request level interceptors must implement **send_oneway**, **send**, **get_response**, and **poll_response**.

3.3 Usage Examples

3.3.1 Server-Side Request Level Interceptor

Let's assume a server implements the following IDL:

```
interface I
{
    long f(in double d);
};
```

The following C++ code shows the **invoke** method of a server-side interceptor that multiplies the **in** argument by 3.14 prior to the execution of the servant's **f** method, and divides the return value by 2 after **f** returns.

```
void MyServerInterceptor::invoke(
    POI::ServerInterceptorRequest_ptr req,
    Dynamic::RequestData*& data)
{
    // Get the operation name
    String_var op = req -> operation();

    // Only modify if the operation name is "f"
    if(strcmp(op, "f") == 0)
    {
        // Modify the in parameter
        CORBA::Double d;
        data -> parameters[0].value >>= d;
        d *= 3.14;
        data -> parameters[0].value <<= d;
    }

    // Call the next interceptor
    req -> next() -> invoke(req, data);

    // Only modify if the operation name is "f"
    if(strcmp(op, "f") == 0)
    {
        // Modify the return value
        CORBA::Long l;
        data -> result >>= l;
        l /= 2;
        data -> result <<= l;
    }
}
```


3.3.2 *Client-Side Request Level Interceptor*

Consider a server implementing the following IDL for a German-to-English translator:

```
interface Translator  
{  
    void translate(in string german, out string english);  
};
```

translate takes a German word as an **in** argument and returns the English translation as an **out** argument.

The **invoke** operation of a caching client-side interceptor could be implemented in C++ as follows:

```

void MyClientInterceptor::invoke(
    POI::ClientInterceptorRequest_ptr req,
    Dynamic::RequestData*& data)
{
    // Get the operation name
    String_var op = req -> operation();

    // Check operation name
    if(strcmp(op, "translate") == 0)
    {
        // Get the in parameter
        char* german;
        data -> parameters[0].value >>= german;

        // Check cache
        char* english = TheCache -> get(german);

        // If lookup was successful, set out parameter
        // and return
        if(english)
        {
            data -> parameters[1].value <=& english;
            return;
        }
    }

    // Call the next interceptor
    req -> next(data);

    // Check operation name
    if(strcmp(op, "translate") == 0)
    {
        // Get the in and the out parameter
        char* german;
        data -> parameters[0].value >>= german;
        char* english;
        data -> parameters[1].value >>= english;

        // Put the out parameter in the cache
        TheCache -> put(english, german);
    }
}

```

TheCache is the German/English cache object, which has the operations **get** and **put**. The implementation of this cache object is not shown here.

Note the return statement in case the cache lookup was successful. No further interceptors are called in that case, and no operation is invoked on the server.

This chapter presents the specification and usage of Message Level Interceptors. Message level interceptors are invoked during GIOP message processing for incoming and outgoing messages. This submission first presents locality-constrained interfaces for accessing the contents of GIOP messages. Afterwards, the specification of message level interceptors and use cases for their application are presented.

4.1 Representation of GIOP Messages

GIOP messages are represented by locality-constrained interfaces which allow an interceptor implementation to access the contents of a particular GIOP message. Furthermore, access to certain attributes of a message can be restricted, for security or other reasons. Figure 1 on page 28 illustrates the relationships between the interfaces for GIOP message representation defined in this submission. The interface **Message** serves as base interface for all GIOP messages. It provides access to certain common attributes such as GIOP version, message type and message size. Furthermore, it provides access to the service context of a GIOP message, in the case that a given GIOP message type carries a service context. Service contexts are represented by the interface **ServiceContext**.

The interface **Message** also represents several specific GIOP message types, such as **MessageError** and **CloseConnection**. The interfaces **RequestMessage**, **ReplyMessage**, **LocateRequest**, **Fragment**, **CancelRequest** and **LocateReply**

are specializations of **Message**. Each of these represents a particular GIOP message type, and provides access to the type-specific message contents. These interfaces, their attributes and operations are defined in the following sections.

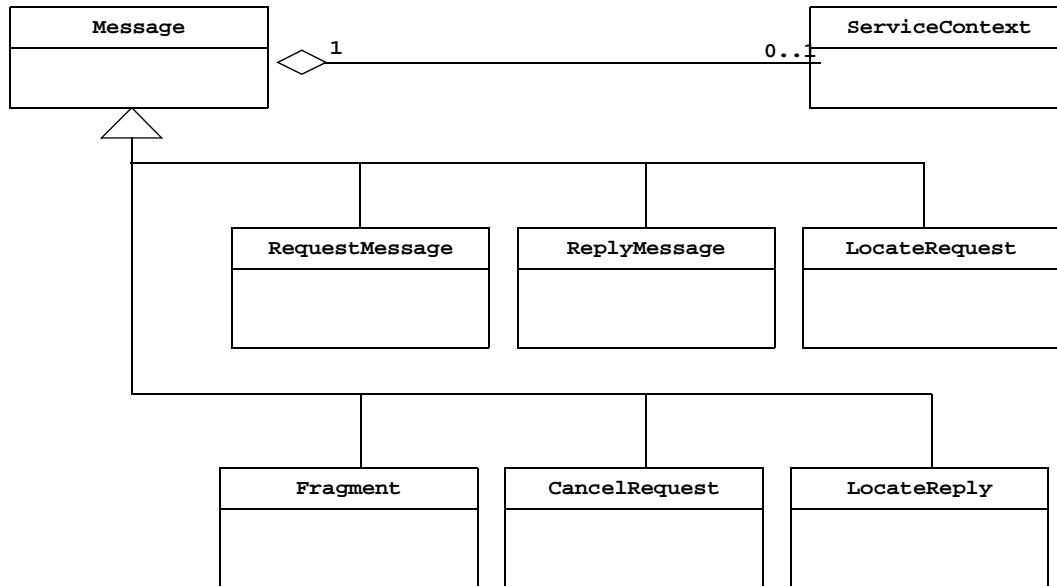


Figure 1 Inheritance Chart for Interfaces representing GIOP Messages

4.1.1 Exception and Type Definitions

The submission defines a number of exceptions, which can be raised by certain operations.

exception CannotModify {};

The **CannotModify** exception is raised when an attempt to modify the contents of a message is not allowed.

exception AlreadySet {};

The **AlreadySet** exception is raised when the current state of a message prevents an attempt to modify that message. For example, this situation could occur if one tries to add a service context with a service type that has already been set.

exception NotSupported {};

The **NotSupported** exception may be raised by operations that depend on a specific GIOP version or message type. As an example, the **NotSupported** exception is raised if a client attempts to retrieve the service context of a GIOP message that cannot carry a service context, such as the **CancelRequest** message.

exception NotFound {};

The **NotFound** exception is raised if the retrieval of a specific service context entry is unsuccessful.

exception InvalidServiceld {};

The **InvalidServiceld** exception is raised if an unknown service context ID is given as a parameter to the retrieval operation for service contexts. Valid service context ID values are those which are defined in [CORBA 2.3], chapter 13.6.7.

typedef sequence<octet> OctetSeq;

An unbounded byte stream, a “blob“, is defined as **OctetSeq**.

4.1.2 Representation of Service Contexts

The interface **ServiceContext** represents the list of service context entries of a given GIOP message. It provides operations to retrieve, add and remove service contexts.

```
interface ServiceContext
{
    IOP::ServiceContextList get_service_context_list();
    IOP::ServiceContext get_service_context( in IOP::Serviceld service_id )
        raises ( NotFound, InvalidServiceld );
    void remove_service_context ( in IOP::Serviceld service_id )
        raises ( NotFound, CannotModify, InvalidServiceld );
    void add_service_context ( in IOP::ServiceContext service_context )
        raises ( CannotModify, InvalidServiceld, AlreadySet );
};
```

The operation **get_service_context_list** returns the list of service contexts as **IOP::ServiceContextList**. If no service context is currently set for the message, the operation returns an empty list. The return type is specified in the CORBA 2.3 specification (ptc/98-12-04), chapter 13.6.7.

The operation **get_service_context** is used to retrieve a service context entry from the current list of service contexts. The service context to be returned is specified by the parameter **service_id**. If the given service context ID is invalid, the operation raises **InvalidServiceld**. Valid service context ID values are those specified in the CORBA 2.3 specification (ptc/98-12-04), chapter 13.6.7. If no service context is found with the given service context ID, the operation raises **NotFound**, otherwise the concrete service context is returned.

Note – An alternative way for the definition of **get_service_context** would be **valuetype ServiceContextOpt IOP::ServiceContext;**

```
interface ServiceContext {
    ServiceContextOpt get_service_context ( in IOP::Serviceld service_id )
```

```

        raises (InvalidServiceId);
};

```

This allows for return of NULL value and reduces the use of exceptions.

The operation **remove_service_context** removes a service context entry, identified by the parameter **service_id**, from the list of service contexts. The operation raises **NotFound** if no service context could be found with the given service context ID. If the given service context ID is invalid, the operation raises **InvalidServiceId**. If the represented GIOP message is of GIOP version 1.0 or 1.1, and the removal of the service context would influence the marshaling of an existing message body, the exception **CannotModify** is raised.

The operation **add_service_context** allows a client to add a service context, given by the parameter **service_context**, to the list of service contexts currently carried by the represented GIOP message. If a modification is not allowed or impossible due to alignment reasons, the **CannotModify** exception is raised. If the service context ID of the **service_context** parameter is invalid, the operation raises **InvalidServiceId**. If a service context with the given service context ID already exists in the message, the operation raises **AlreadySet**.

4.1.3 A Common GIOP Message Interface

The interface **Message** serves as a base representation for all GIOP messages. It provides a number of operations and attributes to access data common to all types of GIOP messages.

```

interface Message
{
    readonly attribute GIOP::Version GIOP_version;
    readonly attribute octet message_type;
    readonly attribute octet flags;
    readonly attribute unsigned long message_size;
    readonly attribute unsigned long transport_endpoint;
    ServiceContext get_service_context()
        raises ( NotSupported );
};

```

Note – These definitions must be aligned with CORBA Messaging.

The **GIOP_version** attribute holds a struct of type **GIOP::Version** as defined in the CORBA 2.3 specification (ptc/98-12-04), chapter 15.4.1.

The **message_type** attribute holds an **octet** value to define the GIOP message type. Valid values are those which are specified in the CORBA 2.3 specification (ptc/98-12-04), chapter 15.4.

The **flags** attribute holds information about byte ordering and fragmentation according to 15.4.1 of the CORBA 2.3 specification (ptc/98-12-04).

The **message_size** attribute contains the number of octets following the message header. Note that changes to the contents of a represented GIOP message will imply an appropriate change to the value of **message_size**.

The **transport_endpoint** attribute identifies a transport endpoint within the ORB which received this message or which will be used to send the message. The identification is unique within a client or server (i.e., unique within a process). It is the responsibility of the ORB to identify such transport endpoints. This identification is needed to allow a client of **Message** to relate separate messages to each other, e.g., a **RequestMessage** to a **ReplyMessage**.

Note – **transport_endpoint** identifies a connection. For support of OTS, it is absolutely necessary to relate a request to a reply message at the message level. The submitters believe that the **request_id** of a request should be the original one, and not changed to an unambiguous value within a server. The combination of **transport_endpoint** and **request_id** allows for unambiguous relation between request and reply.

It is not intended that a client of **Message** will change any of the attributes **GIOP_version**, **message_type**, **transport_endpoint**, **flags** and **message_size**. These attributes are therefore marked as read-only.

The **get_service_context** operation is used to access the **ServiceContext** interface of a GIOP message representation. If the represented GIOP message type supports a service context (i.e., messages of type GIOP Request and GIOP Reply), a reference to a **ServiceContext** interface (See “Representation of Service Contexts” on page 29.) is returned, otherwise **NotSupported** is raised.

The **Message** interface serves as the base interface for the representation of the GIOP messages **Request**, **Reply**, **LocateRequest**, **Fragment**, **CancelRequest** and **LocateReply**. Furthermore, the **Message** interface also serves as the representation of the GIOP messages **MessageError** and **CloseConnection**.

4.1.4 Representation of a GIOP Request Message

The interface **RequestMessage** is a specialization of **Message** providing access to the contents of a GIOP Request message.

```

interface RequestMessage : Message
{
    readonly attribute boolean response_expected;
    readonly attribute string operation;
    readonly attribute unsigned long request_id;
    GIOP::TargetAddress get_target_address ()
        raises ( NotSupported );
    void set_target_address ( in GIOP::TargetAddress new_target )
        raises ( NotSupported, CannotModify );
    OctetSeq get_message_body ()
        raises ( NotSupported );
    void set_message_body ( in OctetSeq new_body )
        raises ( NotSupported, CannotModify );
    ReplyMessage create_reply( in octet reply_status,
                               in OctetSeq message_body )
        raises ( NotSupported );
};

```

The **response_expected** attribute specifies whether the current request has twoway (**TRUE**) or oneway (**FALSE**) semantics.

The **operation** attribute holds the name of the requested operation.

The **request_id** attribute holds the request ID of this request.

It is not intended that a client of **RequestMessage** will change any of the attributes **response_expected**, **operation**, **flags** and **request_id**. These attributes are therefore marked as read-only.

The operation **get_target_address** allows access to the target address of a GIOP Request as it is specified for GIOP 1.2. If the represented GIOP RequestMessage is of GIOP version 1.2, a value of **union GIOP::TargetAddress** is returned, otherwise the **NotSupported** exception is raised.

The operation **set_target_address** can be used by a client to redirect a GIOP Request to another target address, given by the parameter **new_target**. If the represented GIOP RequestMessage is not of GIOP version 1.2, the **NotSupported** exception is raised. If request redirection is forbidden, a **CannotModify** exception is raised.

The operation **get_message_body** returns the marshaled message body of a request message as an octet sequence. The **NotSupported** exception is raised if access to the message body is impossible. If the request does not contain any message body, it returns a sequence of octets with length 0.

The operation **set_message_body** is used to replace the message body of a request message to a value given by the **new_body** parameter. The operation raises the **NotSupported** exception if access to the message body is impossible. If the **RequestMessage** object represents a GIOP Request of GIOP version 1.0 or 1.1, it raises a **CannotModify** exception if alignment rules are violated.

If a client of **RequestMessage**, e.g., a fault analyzer interceptor or a caching interceptor, wants to create a reply message for the request and send the reply back to the client, it can use the operation **create_reply** to create a **ReplyMessage** object. The operation expects

- a reply status parameter, i.e., one of the values **NO_EXCEPTION** (value 0), **USER_EXCEPTION** (value 1), **SYSTEM_EXCEPTION** (value 2), **LOCATION_FORWARD** (value 3), **LOCATION_FORWARD_PERM** (value 4), **NEEDS_ADDRESSING_MODE** (value 5), and
- a corresponding message body.

create_reply can raise the **NotSupported** exception if the ORB doesn't allow interceptor-initiated construction of reply messages.

This mechanism can be used for framework implementations, where a specific initialization operation must be called on an object reference before other operations can be called. In this use case, an interceptor could always create a reply message containing a well defined user exception.

4.1.5 Representation of a GIOP Reply Message

The interface **ReplyMessage** is a specialization of **Message** providing access to the contents of a GIOP Reply message.

```
interface ReplyMessage : Message
{
    readonly attribute octet reply_status;
    readonly attribute unsigned long request_id;
    OctetSeq get_message_body ()
        raises ( NotSupported );
    void set_message_body ( in OctetSeq new_body )
        raises ( NotSupported, CannotModify );
};
```

The attribute **reply_status** holds the reply status value as specified in chapter 15.4.2.1 of the CORBA 2.3 specification (ptc/98-12-04). Possible values are therefore **NO_EXCEPTION** (value 0), **USER_EXCEPTION** (value 1), **SYSTEM_EXCEPTION** (value 2), **LOCATION_FORWARD** (value 3), **LOCATION_FORWARD_PERM** (value 4), and **NEEDS_ADDRESSING_MODE** (value 5).

The **request_id** attribute holds the request ID of the corresponding GIOP **RequestMessage**. In combination with the **transport_endpoint** attribute of the base interface **Message**, a unique relationship to one former Request message is possible.

It is not intended that a client of **ReplyMessage** will change the values of the attributes **reply_status** or **request_id**. These attributes are therefore marked as read-only.

The operation **get_message_body** returns the marshaled message body of a reply message as an octet sequence. The **NotSupported** exception is raised if access to the message body is impossible. If the reply does not contain any message body, it returns a sequence of octets with length 0.

The operation **set_message_body** is used to replace the message body of a reply message to a value given by the **new_body** parameter. The operation raises the **NotSupported** exception if access to the message body is impossible. If the current **ReplyMessage** object represents a GIOP Request of GIOP version 1.0 or 1.1, it raises a **CannotModify** exception if alignment rules are violated.

4.1.6 Representation of a GIOP CancelRequest Message

The interface **CancelRequest** is a specialization of **Message** providing access to the contents of a GIOP CancelRequest Message.

```
interface CancelRequest : Message
{
    readonly attribute unsigned long request_id;
};
```

The read-only attribute **request_id** provides access to the request ID of a former GIOP Request message to be canceled.

4.1.7 Representation of a GIOP LocateRequest Message

The interface **LocateRequest** is a specialization of **Message** providing access to the contents of a GIOP LocateRequest message.

```
interface LocateRequest : Message
{
    readonly attribute unsigned long request_id;
    OctetSeq get_object_key()
        raises ( NotSupported );
    GIOP::TargetAddress get_target ()
        raises ( NotSupported );
    LocateReply create_locate_reply
        ( in octet locate_status, in OctetSeq new_target )
        raises ( NotSupported );
};
```

The read-only attribute **request_id** provides access to the request ID of this LocateRequest message.

If **LocateRequest** represents a LocateRequest message of GIOP version 1.0 or 1.2, the object key of the object to be located can be obtained using the operation **get_object_key**. If the represented GIOP message is of GIOP version 1.2, **get_object_key** raises a **NotSupported** exception. **get_object_key** returns the object key as an octet sequence.

If **LocateRequest** represents a LocateRequest message of GIOP version 1.2, the target address of the object to be located can be obtained using the **get_target** operation. **get_target** returns a union of type **GIOP::TargetAddress**. If the GIOP message is not of GIOP version 1.2, **get_target** raises a **NotSupported** exception.

If a client of **LocateRequest**, e.g., a bridging interceptor or load balancer, wants to create a reply message for the request and send this reply back to the client, it can use the operation **create_locate_reply** to create a **LocateReply** object. The operation expects

- a locate status parameter, i.e., one of the values **UNKNOWN_OBJECT** (value 0), **OBJECT_HERE** (value 1), **OBJECT_FORWARD** (value 2), **OBJECT_FORWARD_PERM** (value 3), **LOC_SYSTEM_EXCEPTION** (value 4), **LOC_NEEDS_ADDRESSING_MODE** (value 5), and
- a target address in its CDR encapsulation according to chapter 15.4.5.2 of the CORBA 2.3 specification (ptc/98-12-04).

create_locate_reply can raise the **NotSupported** exception if the ORB doesn't allow interceptor-initiated construction of locate reply messages.

4.1.8 Representation of a GIOP LocateReply Message

The interface **LocateReply** is a specialization of **Message** providing access to the contents of a GIOP LocateReply message.

```
interface LocateReply : Message
{
    readonly attribute unsigned long request_id;
    readonly attribute octet locate_status;
    OctetSeq get_locate_reply_body ()
        raises ( NotSupported );
    void set_locate_reply_body ( in OctetSeq new_target )
        raises ( NotSupported, CannotModify );
};
```

The read-only attribute **request_id** holds the request ID of the corresponding LocateRequest message. In combination with the **transport_endpoint** attribute of the base interface **Message**, a unique relationship to a former LocateRequest message is possible.

The read-only attribute **locate_status** holds the result of the locate request. Valid values are those which are specified in the CORBA 2.3 specification (ptc/98-12-04), chapter 15.4.5.1: **UNKNOWN_OBJECT** (value 0), **OBJECT_HERE** (value 1), **OBJECT_FORWARD** (value 2), **OBJECT_FORWARD_PERM** (value 3), **LOC_SYSTEM_EXCEPTION** (value 4), **LOC_NEEDS_ADDRESSING_MODE** (value 5).

The operation **get_locate_reply_body** returns the result of the locate request, according to chapter 15.4.5.2. **get_locate_reply_body** raises the **NotSupported** exception if access to the message body is impossible. If the locate reply does not contain any message body, it returns a sequence of octets with length 0.

The operation **set_locate_reply_body** is used to replace the message body of a locate reply message to a value given by the **new_target** parameter. The operation raises the **NotSupported** exception if access to the locate reply body is impossible. If the current **LocateReply** object represents a GIOP LocateReply message of GIOP version 1.0 or 1.1, the operation raises a **CannotModify** exception if alignment rules are violated. Furthermore, it raises **CannotModify** if target redirection is not allowed for security reasons.

4.1.9 Representation of a GIOP Fragment Message

The interface **Fragment** is a specialization of **Message** providing access to the contents of a GIOP Fragment message.

```
interface Fragment : Message
{
    readonly attribute unsigned long request_id;
    OctetSeq get_message_body ()
        raises ( NotSupported );
    void set_message_body ( in OctetSeq new_body )
        raises ( NotSupported, CannotModify );
};
```

The read-only attribute **request_id** holds the request ID of the Request message this fragment belongs to.

The operation **get_message_body** returns the marshaled message body of a fragment message as an octet sequence. It raises the **NotSupported** exception if access to the message body is impossible.

The operation **set_message_body** is used to replace the message body of a fragment message to a value given by the **new_body** parameter. The operation raises the **NotSupported** exception if access to the message body is impossible. If the current **Fragment** object represents a GIOP Request message of GIOP version 1.0 or 1.1, it raises a **CannotModify** exception if alignment rules are violated.

4.2 The Message Level Interceptor

4.2.1 Definition of Message Level Interceptor

A Message Level Interceptor is defined as the locality-constrained interface **MessageInterceptor** providing the operation **message**.

```

interface MessageInterceptor
{
    enum InterceptionState {
        PROCEED_REVERSE,
        PROCEED_REGULAR
    };
    InterceptionState message( in Message message_in,
                              out Message message_out );
};

```

If an object supporting the interface **MessageInterceptor** is registered with an ORB, each time the ORB receives a GIOP message or is preparing to send a GIOP message, it invokes the **message** operation. The ORB supplies the representation of that GIOP message as defined in 4.1.2 as the in parameter **message_in**.

During the interception of a GIOP message, the interceptor (i.e., the implementation of **message** in the **MessageInterceptor** instance) can

- change the contents of the given GIOP message using the operations explained in 4.1.2 and force the ORB to proceed with the changed message in the regular processing direction by returning **PROCEED_REGULAR**.
- throw any system or user exception.
 - If the given message was a **RequestMessage**, the ORB sends this exception as a reply back to the client.
 - If the given message was a **LocateRequest**, the ORB sends the exception **LOC_SYSTEM_EXCEPTION** back to the client.
 - If the message was not of type **RequestMessage** or **LocateRequest**, the ORB sends a GIOP MessageError back to the client.

- build a corresponding GIOP message, given to the ORB as the out parameter **message_out**, and force the ORB to proceed with that new message by returning **PROCEED_REVERSE**.

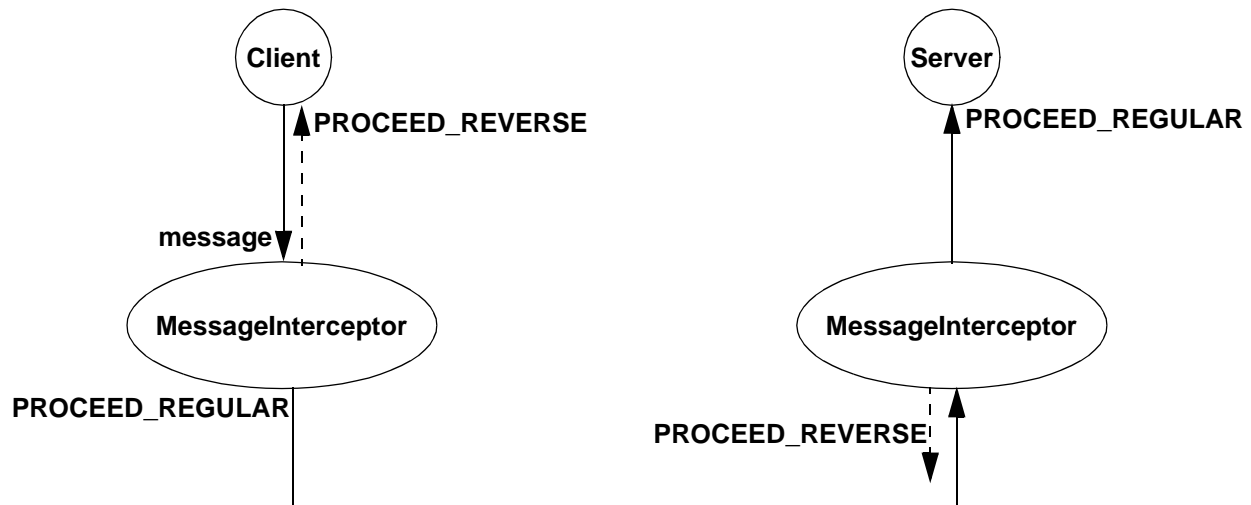


Figure 2 PROCEED_REVERSE and PROCEED_REGULAR in operation invocation phase

Figure 2 on page 38 depicts both directions **PROCEED_REVERSE** and **PROCEED_REGULAR** for the operation invocation phase.

The following rules apply:

- If the operation **message** on a message level interceptor at the client side is called with a **RequestMessage** parameter, then the interceptor can either
 - assign nil to the **message_out** parameter, change the contents of the **RequestMessage** object using the operations provided by **RequestMessage** and return **PROCEED_REGULAR**, or
 - create an instance of the corresponding **ReplyMessage** using **create_reply** on interface **RequestMessage**, assign that instance to **message_out** and return **PROCEED_REVERSE**.
- If the operation **message** on a message level interceptor at the client side is called with a **LocateRequest** parameter, then the interceptor can either
 - assign nil to the **message_out** parameter, change the contents of the **LocateRequest** object using the operations provided by **LocateRequest** and return **PROCEED_REGULAR**, or
 - create an instance of the corresponding **LocateReply** using **create_locate_reply** on interface **LocateRequest**, assign that instance to **message_out** and return **PROCEED_REVERSE**.
- If the operation **message** at a message level interceptor is called with any **Message** except **RequestMessage** or **LocateRequest**, it must assign nil to **message_out**, can change the contents of that message using the provided operations and must return **PROCEED_REGULAR**.

Note – We discussed an alternative specification of interface **MessageInterceptor** as follows:

```
interface MessageInterceptor {
    Message message ( in Message message );
};
```

If the return message is not nil, then the proceeding direction is **PROCEED_REVERSE**, or **PROCEED_REGULAR** otherwise. It is semantically equivalent, but probably less intuitive.

Message level interceptors are registered with an ORB using the operations defined in Section 7.5. There can be more than one interceptor registered with an ORB at any given time. The order of invoking the **message** operation at each **MessageInterceptor** instance is determined by the ORB according to its policies.

4.2.2 Example: A User-supplied Load Balancing Interceptor

In the first example, a message level interceptor for load balancing knows about a certain number of objects supporting the interface **LongTermCalulator**.

```
interface LongTermCalulator
{
    void takes_much_time();
};
```

The interceptor should redirect requests to objects of type **LongTermCalulator**. The interface definition for the example would be

```
interface LoadBalancer : LongTermCalulator, POI::MessageInterceptor
{};
```

Our **LoadBalancer** supports the interface **LongTermCalulator** and **MessageInterceptor**. Clients that want to use the operation **takes_much_time** always use an IOR referencing our **LoadBalancer** object. We implement only the **message** operation that is supported through inheritance from **MessageInterceptor**.

The **message** operation could be implemented as follows:

```

class MyLoadBalancer : public POI::MessageInterceptor
{
    POI::OctetSeq get_next_long_term_calculator()
    {
        // Get next available calculator and return
        // target address CDR encapsulated
    }

public:

    virtual POI::MessageInterceptor::InterceptionState
    message(POI::Message_ptr message_in,
            POI::Message_ptr& message_out)
    {
        if ( the_message->message_type() != 0 ) {
            message_out = POI::Message::_nil();
            return POI::MessageInterceptor::PROCEED_REGULAR;
        }

        POI::Message_var _request =
            POI::RequestMessage::_narrow ( message_in );

        if(strcmp(_request->operation,"takes_much_time") != 0){
            message_out = POI::Message::_nil();
            return POI::MessageInterceptor::PROCEED_REGULAR;
        }
        try {
            message_out = _request->create_reply
                (3, get_next_long_term_calculator ());
        }
        catch (const POI::NotSupported&) {
            throw CORBA::INTERNAL;
        }

        return POI::MessageInterceptor::PROCEED_REVERSE;
    }
};

```

Our implementation class `MyLoadBalancer` has a private method `get_next_long_term_calculator`, which resolves the next available object of type **LongTermCalulator** using some mechanism (e.g., the trading service) and returns the target address of that object in its CDR encapsulated representation. The **message** implementation first verifies that the message type of the message to be intercepted is Request (value 0). If this is not the case, it sets the `message_out` parameter to nil and returns **PROCEED_REGULAR**, which will result in regular invocation behaviour. If the requested operation is not "takes_much_time", our implementation again sets `message_out` to nil and forces regular invocation behaviour by returning **PROCEED_REGULAR**. Otherwise, it creates a location forward reply message using the **create_reply** operation of the given **RequestMessage** object, assigns this reply message to `message_out` and returns **PROCEED_REVERSE**. The return value forces the ORB to abandon the regular invocation mechanism and

return our GIOP reply message, containing a location forward and the resolved object reference of the next available **LongTermCalulator** object. The client in turn issues his request to that **LongTermCalulator** object.

Our interceptor is registered with the ORB using the appropriate operations described in Section 7.5.

The previous chapters described how CORBA object interactions are intercepted at the request and message level. Another point of interest for interceptors in the invocation processing of an ORB is the management of connections. This is especially important for monitoring and debugging tools. Such tools typically provide operations such as:

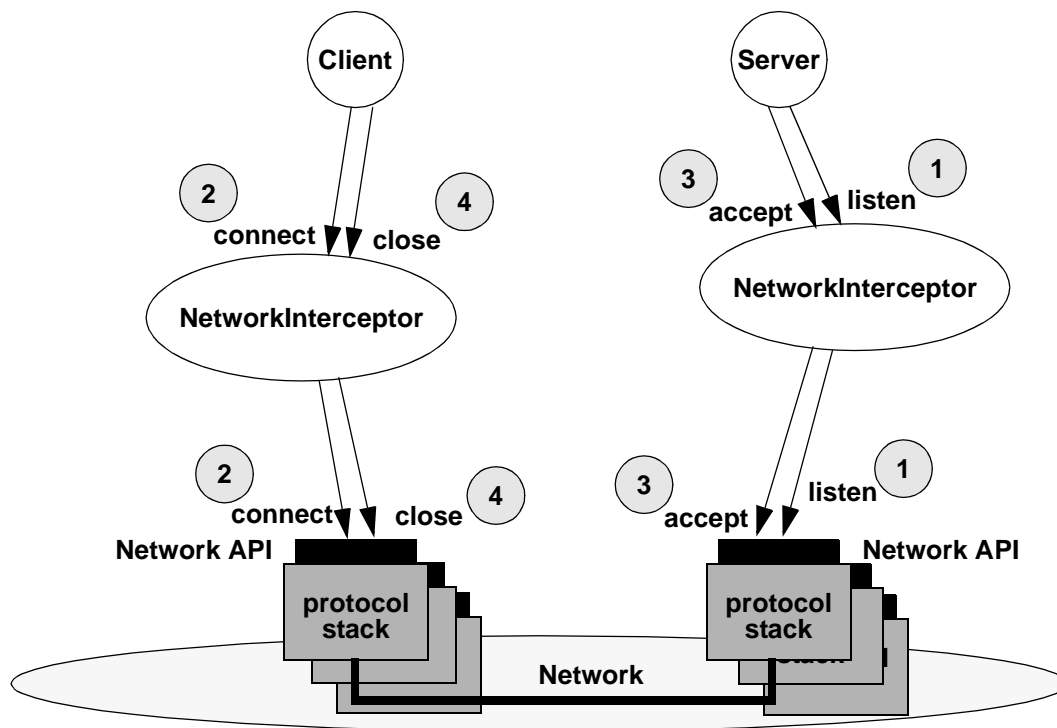


Figure 3 Interception Points in Connection Establishment

- Monitoring on which transport endpoints a server listens for incoming connection requests.
- Providing information about which transport endpoint will be used for a particular request from a client to a server.
- To intercept connection establishment and closure.
- Providing information about connection reuse for invocations on different target objects.
- Monitoring network connection re-establishment in case of failures.

Monitoring and debugging tools therefore need well-defined “hooks” within the ORB in order to be implemented in a standardized way. Such hooks are hereinafter referred to as *Network Level Interceptors*.

Note – This submission does not specify Network Level Interceptors, but the submitters intend to do so in a revised submission.

For CORBA services as well as for debugging and monitoring tools, the management of object references is an important point to be analysed and influenced. This analysis includes creation, changing and activation of object references. The submitters identified several processing points while an ORB processes object references.

Those processing points include:

- Creation of an object reference within the POA, as described in [CORBA2.3], chapter 11.2.4.
- Addition of profiles to such object references, including addition of standard components to a profile, as described in [CORBA2.3], chapter 13.6.3.
- Mapping of an object reference to an active object map in a POA, as described in [CORBA2.3], chapter 11.2.
- Reception of an object reference within a client domain.
- Transformation of an object reference into an ORB internal form (e.g., stub creation).
- Removal of such an internal form (e.g., stub release).
- First time usage of an object reference.

Note – This submission does not specify any of these interceptors, but the submitters intend to do so in a revised submission.

After the discussion of the different kinds of interceptors and how they relate to each other, a way to introduce them into an ORB both upon initialization and during runtime is required. This chapter discusses the registration of interceptors with an ORB and proposes a way to manage them.

7.1 Requirements for Management of Interceptors

Interceptors are hooked into an ORB and intercept invocation paths on both the client and server side. There are four different flavours of interceptors, for which different registries have to be provided to register interceptors of the desired type.

As discussed in previous chapters, it is necessary and desirable for interceptors to be invoked in a daisy-chained fashion. For this reason, a client or server should be able to explicitly enable or disable some of the interceptors in the chain. Complete disabling of any registered interceptors should also be possible as it should be possible to get the current activation status for each registered interceptor. The activation order of the registered interceptors should be the order of registration. A way to change the order of activation to satisfy requirements of a client or server should be addressed in a revised submission.

Interceptors have to be registered with the ORB. We distinguish static registration of interceptors and dynamic registration. Interceptors that are registered statically should be used by default in any invocation. The registration of these interceptors is implementation-dependent and could be done through configuration files, which are parsed before or during the initialization of the ORB. Dynamic interceptors are registered on behalf of the client or server, using a registry which is obtained from the ORB.

It must be possible to limit the activation of an interceptor to a particular set of objects, operations, interfaces, or a combination of these. Information about the registered interceptors, especially about the activation policies of a particular interceptor, must be obtainable from the ORB.

Furthermore, it must be possible to unregister interceptors, both on explicit requests from clients or servers, or because of garbage collection performed by the ORB. For this reason a way has to be found to avoid unregistering or at least limit the risk of accidentally unregistering interceptors registered by other entities.

Note – I don't understand the last sentence (ml)

7.2 *Bootstrapping the Registry for Interceptors*

When a client or a server deals with interceptors, it has to obtain the object references of the different registries. This is done by calling **resolve_initial_references** with an identifier like “**InterceptorRegistry**” to obtain the initial object reference of an object with the following interface:

```
module POI
{
    interface InterceptorRegistry
    {
        RequestLevelClientRegistry
        get_request_level_client_registry();

        RequestLevelServerRegistry
        get_request_level_server_registry();

        MessageLevelClientRegistry
        get_message_level_client_registry();

        MessageLevelServerRegistry
        get_message_level_server_registry();

        IORManagementRegistry
        get_ior_management_registry();
    };
};
```

Using this reference, an application can resolve the different registries and call the appropriate operations to register interceptors.

7.3 *The Binding Object*

Once an interceptor is registered with the ORB, a mechanism is needed to manipulate that registration for the purposes of management, activation and unregistration. This mechanism should be realized by an object representing one interceptor registration. The binding objects, which are defined to serve this purpose, allow for the activation/deactivation of a registered interceptor, the query of the interceptor

activation status and the unregistration of the interceptor. The lifetime of such a binding object is correlated with the registration of the interceptor, and destroying this object is accompanied by an unregistration of the corresponding interceptor.

The interfaces of the binding objects are defined as follows:

```

module POI
{
    enum BindingType { ALWAYS, OBJECT, OBJECT_OPERATION,
        OPERATION, INTERFACE };

    interface InterceptorBinding
    {
        exception CANNOT_DESTROY { string reason; };

        readonly attribute BindingType binding_type;

        void destroy()
            raises ( CANNOT_DESTROY );

        void enable();
        void disable();
        boolean is_enabled();
    };

    typedef sequence<InterceptorBinding> BoundInterceptors;
};

```

An object supporting the interface **InterceptorBinding** is created and its reference returned to the caller every time a request level, message level or IOR management interceptor is registered with the ORB. The caller should destroy this object when the corresponding interceptor is no longer needed. The ORB can implement a scheme for garbage collection to unregister interceptors that are no longer needed by any application, e.g., for interceptors which are registered to act on a particular object reference in case the lifetime of this object reference ends.

The interface **InterceptorBinding** provides operations for enabling and disabling the use of a corresponding interceptor. The operation **enable** activates the registered interceptor. Subsequent invocations in the ORB are now intercepted by this interceptor according to the registration policies. The counterpart to **enable** is **disable**, which deactivates this interceptor. Using the operation **is_enabled**, the activation status of the registered interceptor can be queried.

Note that the same interceptor can be registered in multiple interceptor bindings and disabling one binding doesn't affect other bindings of the same interceptor.

When the operation **destroy** is invoked on a binding object, the corresponding registration of the interceptor is removed from the ORB. Other registrations of the same interceptor are not affected by this call. Restrictions on this operation are explained later in this document.

7.4 The Registry for Request Level Interceptors

For request level interceptors all information about the called object is available, such as the interface of the object, and the operation and operation parameters. For this reason it is necessary and desirable to be able to register interceptors for different purposes according to the needs of the registering entity. Furthermore, a distinction between local calls and remote calls should be made on registering an interceptor.

For request level interceptors, there exist two registries: one is responsible for managing a list of interceptors on the client side and the other manages a list for interceptors on the server side.

This submission suggests different registries for the different kinds of interceptors. The interfaces of the registries for request level interceptors are shown below. The registries also provides operations to disable or enable the use of any interceptor.

```

module POI
{
    enum InterceptorActivationScope
    {
        LOCAL,
        REMOTE,
        ALWAYS
    };

    interface RequestLevelRegistryBase
    {
        BoundInterceptors get_interceptors();

        void enable_interceptors ();
        void disable_interceptors ();
    };

    interface RequestLevelClientRegistry : RequestLevelRegistryBase
    {
        InterceptorBinding
        bind_for_object (
            in ClientInterceptor req_inter,
            in Object obj,
            in InterceptorActivationScope act_scope );

        InterceptorBinding
        bind_for_object_operation (
            in ClientInterceptor req_inter,
            in Object obj,
            in CORBA::Identifier op_name,
            in InterceptorActivationScope act_scope );

        InterceptorBinding
        bind_for_interface (
            in ClientInterceptor req_inter,
            in CORBA::Identifier interface_repository_id,
            in InterceptorActivationScope act_scope );

        InterceptorBinding
        bind_for_operation (
            in ClientInterceptor req_inter,
            in CORBA::Identifier op_name,
            in InterceptorActivationScope act_scope );

        InterceptorBinding
        bind (
            in ClientInterceptor req_inter,
            in InterceptorActivationScope act_scope );
    };
}

```

```

interface RequestLevelServerRegistry : RequestLevelRegistryBase
{
    InterceptorBinding
    bind_for_object (
        in ServerInterceptor req_inter,
        in Object obj,
        in InterceptorActivationScope act_scope );

    InterceptorBinding
    bind_for_object_operation (
        in ServerInterceptor req_inter,
        in Object obj,
        in CORBA::Identifier op_name,
        in InterceptorActivationScope act_scope );

    InterceptorBinding
    bind_for_interface (
        in ServerInterceptor req_inter,
        in CORBA::Identifier interface_repository_id,
        in InterceptorActivationScope act_scope );

    InterceptorBinding
    bind_for_operation (
        in ServerInterceptor req_inter,
        in CORBA::Identifier op_name,
        in InterceptorActivationScope act_scope );

    InterceptorBinding
    bind (
        in ServerInterceptor req_inter,
        in InterceptorActivationScope act_scope );

};
};

```

The client side and server side request level interceptor registry provide similar operations for registering request level interceptors, with the difference that for registering a client side request level interceptor an object supporting the interface **ClientInterceptor** must be given as a parameter, and for the server side an object with the interface **ServerInterceptor** must be provided.

The operation **bind_for_object** registers a request level interceptor to intercept invocations on a particular object, whereas **bind_for_object_operation** registers an interceptor to only intercept invocations of a particular operation on an object. There are also operations to register interceptors to intercept invocations on objects of a particular interface, **bind_for_interface**, which takes an interface repository identifier as parameter, or to intercept invocations of a particular operation on any object, **bind_for_operation**. To register an interceptor to ultimately intercept every invocation, one can use the operation **bind**.

To retrieve the list of binding objects of all registered request level interceptors, the operation **get_interceptors** is used. This operation returns a list of **InterceptorBinding** objects in the registered order. This gives the opportunity to enable or disable interceptors in the chain.

To disable all request level interceptors, the operation **disable_interceptors** is called, whereas **enable_interceptors** turns on all request level interceptors. Request level interceptors that should be used on the client side are registered with the **RequestLevelClientRegistry** and for the server side with the **RequestLevelServerRegistry**.

7.4.1 Synchronous and Asynchronous Invocations

For the purposes of registering request level interceptors it is not necessary to distinguish between synchronous and asynchronous invocations, since all interceptors should be implemented to support both synchronous and asynchronous invocation styles. For this reason an ORB can simply decide to use either the **invoke** operation on the interceptor, or the asynchronous analogues **send** and **get_response** or **poll_response**, depending on the invocation mode.

7.5 The Registry for Message Level Interceptors

Message level interceptors operate over a marshaled blob of data and are able to access the information in the GIOP message header and the corresponding message body. Message level interceptors are also chainable and registries are provided for the client side and the server side invocation paths. The interfaces for the registry are listed below.

```

module POI
{
    interface MessageLevelRegistryBase
    {
        InterceptorBinding
        bind_for_object (
            in MessageInterceptor mess_inter,
            in Object obj,
            in InterceptorActivationScope act_scope );

        InterceptorBinding
        bind (
            in MessageInterceptor mess_inter,
            in InterceptorActivationScope act_scope );

        BoundInterceptors get_interceptors();

        void enable_interceptors ();
        void disable_interceptors ();
    };

    interface MessageLevelClientRegistry :
        MessageLevelRegistryBase {};

    interface MessageLevelServerRegistry :
        MessageLevelRegistryBase {};
};

```

The operation **bind_for_object** registers a message level interceptor for intercepting GIOP protocol message processing associated with a particular object. To intercept any GIOP protocol message processing, the operation **bind** is used. In both cases the parameter **act_scope** specifies whether to intercept only remote calls or only local calls or both.

As it was with the request level interceptors, the operation **get_interceptors** returns a list of **InterceptorBinding** objects in the order of registration of the message level interceptors. The single **InterceptorBinding** objects in the list can then subsequently be used to enable or disable an individual interceptor.

The operation **enable_interceptors** enables all message level interceptors in the ORB whereas **disable_interceptors** disables them.

Like request level interceptors, message level interceptors are registered in a registry for the client side, **MessageLevelClientRegistry**, or for the server side, **MessageLevelServerRegistry**.

7.6 Network Level Interceptors

It seems, that there are a number of applications for network level interceptors, which operate on a completely unstructured blob of data ready to send over a transport connection or received over a transport connection. Since the use of such interceptors in its full complexity isn't evaluated yet, this should be addressed in a revised submission.

7.7 The Registry for IOR Management Interceptors

IOR management interceptors are used to monitor and manage interoperable object references in the ORB. This includes collecting information about the lifetime of an IOR, and inserting, deleting or modifying profiles within an IOR.

There exist a number of use cases in which it is desirable to have IOR management interceptors to be invoked one after another, for instance when there are two interceptors and each of them is only able to handle a particular profile. For this reason, daisy-chaining is also supported for these interceptors, and enabling or disabling of an interceptor in the chain is done through a binding object supporting the interface **InterceptorBinding**.

For IOR management interceptors, only one registry is needed within an ORB since IOR management is only reasonable on the server side. The interface of the registry is listed below.

```
module POI
{
    interface IORManagementRegistry
    {
        InterceptorBinding
        bind ( in IORManagementInterceptor ior_inter );

        BoundInterceptors get_interceptors();

        void enable_interceptors();
        void disable_interceptors();
    };
};
```

The operation **bind** registers an IOR management interceptor with the ORB.

Using the operation **get_interceptors**, a list of all binding objects for IOR management interceptors known to the ORB is returned.

To enable all registered IOR management interceptors, the operation **enable_interceptors** is called, to disable all IOR management interceptors, the operation **disable_interceptors** is used.

7.8 *Example for Registering an Request Level Interceptor*

To demonstrate the introduction of a new interceptor to an ORB, the suggested way of registering a request level interceptor on the client side will be explained. The target language for the example is C++. The following code fragment is a rough sketch of what a client must do to make its own interceptor available to the ORB:


```

// Init ORB
...

// Resolve Registry
CORBA::Object_var initial =
    orb->resolve_initial_references ( "POI::InterceptorRegistry" );

POI::InterceptorRegistry_var registry =
    POI::InterceptorRegistry::_narrow ( initial );

POI::RequestLevelClientRegistry_var request_client_reg =
    registry -> get_request_client_registry ();

// New Interceptor from somewhere
MyDebugInterceptor_impl *my_debug_interceptor =
    new MyDebugInterceptor_impl ();

// Register my_debug_interceptor
POI::InterceptorBinding_var my_binding =
    request_client_reg -> bind ( my_debug_interceptor, POI::ALWAYS );

// Enable interceptor
my_binding -> enable();

// Get an object reference my_object from somewhere
...

// Invoke an operation on this object
my_object -> test_operation(); // gets intercepted by my_debug_interceptor

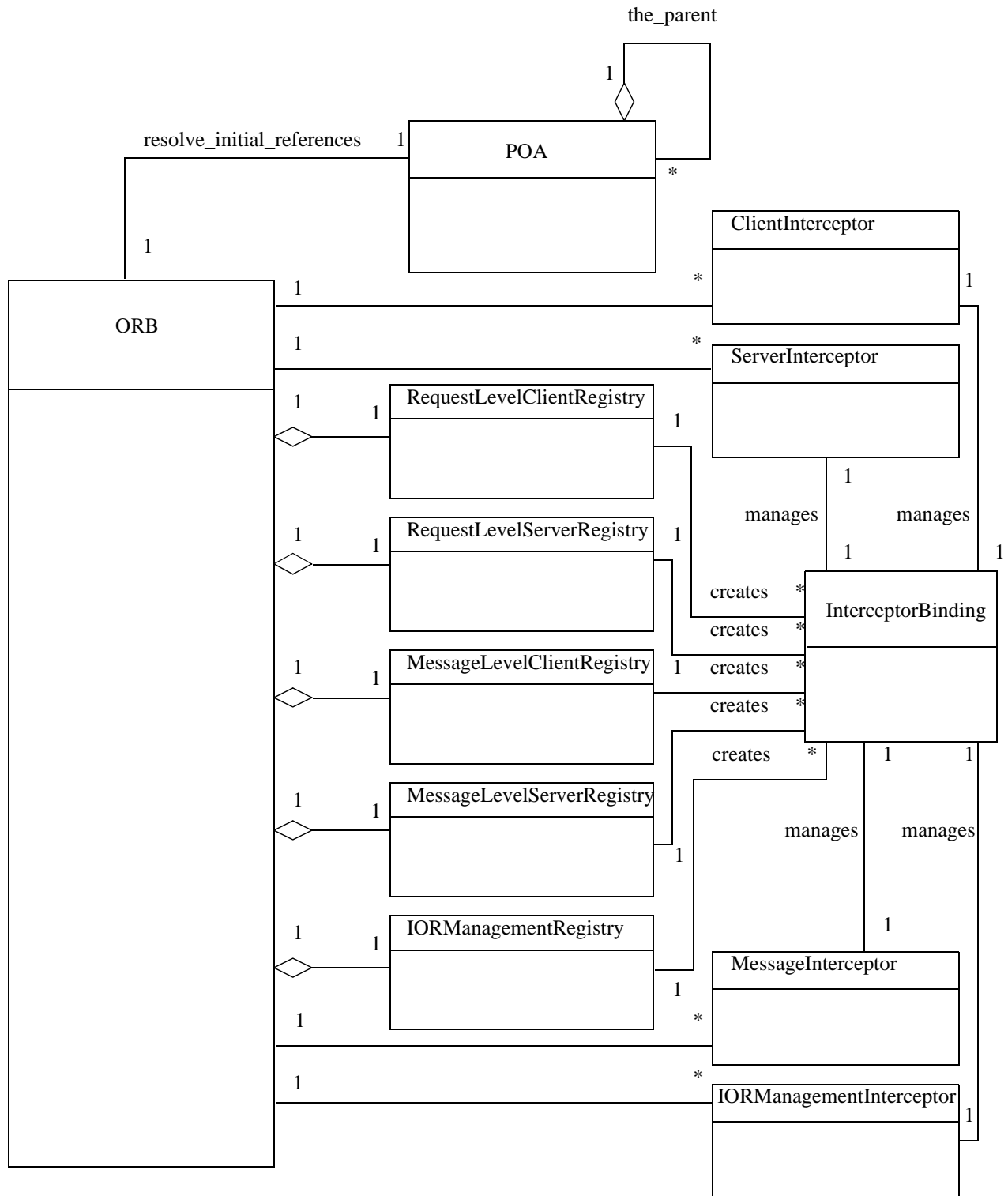
// Finally destroy the binding object
my_binding -> destroy(); // my_debug_interceptor gets unregistered

```

7.9 UML Diagram for the Interceptor Registry

The following UML diagram illustrates the relationships between the different interfaces comprising the registry for interceptors within an ORB. For simplicity the **InterceptorRegistry** object, which can be resolved by **resolve_initial_references**,

is omitted in this diagram.



This chapter identifies the conformance points required for compliant implementations of Portable Interceptors.

8.1 Conformance Points

This submission identifies two conformance points:

- **Changes to CORBA Core**
Chapter 9 identifies changes to CORBA Core. Compliant implementations shall implement these changes.
- **Portable Interceptors**
A product claiming compliance to the Portable Interceptors specification must support the following:
 - The Request Level Interceptors
 - The Message Level Interceptors
 - The support for management of interceptors

Note – If the network level and IOR management interceptors chapters are finalised, the conformance criteria must be updated accordingly

This chapter provides instructions for the OMG editors as to where the definitions of this submission will be placed in the existing OMG specifications.

9.1 Changes to CORBA Core

- New **resolve_initial_references** identifiers: The singleton **ClientRequestFactory** can be resolved by calling the ORB operation **resolve_initial_reference** with the identifier "**ClientRequestFactory**". The registry for interceptors of any type can be resolved by calling the ORB operation **resolve_initial_reference** with the identifier "**InterceptorRegistry**".
- The definition of the new module **Dynamic** replaces the current DII and DSI. The current DII and DSI should be deprecated and included "by reference" to an older specification.
- The interceptor definitions for Request Level Interceptors and Message Level Interceptors replace chapter 18 of the CORBA specification, revision 2.2.

10.1 Module Dynamic

```
module Dynamic
{
    struct Parameter
    {
        any value;
        CORBA::ParameterMode mode;
    };

    typedef sequence<Parameter> ParameterList;
    typedef sequence<CORBA::TypeCode> TypeCodeList;

    struct RequestData
    {
        ParameterList parameters;
        any result;
        TypeCodeList exceptions;
        any except;
        CORBA::Context ctx;
    };

    interface ClientRequest
    {
        void invoke(inout RequestData data);
        void send(inout RequestData data);
        void send_oneway(inout RequestData data);
        void get_response(inout RequestData data)
            raises(CORBA::WrongTransaction);
        boolean poll_response(inout RequestData data)
            raises(CORBA::WrongTransaction);
        void destroy();
    };
}
```

```
};

interface ClientRequestFactory
{
    ClientRequest create_request
        (in Object target, in CORBA::Identifier operation);
};

interface ServerRequest
{
    readonly attribute CORBA::Identifier operation;
};

interface Server
{
    void get_types(in ServerRequest request, inout RequestData data);
    void invoke(in ServerRequest request, inout RequestData data);
};
};
```


10.2 Request Level Interceptors

```

module POI
{
    exception OperationOverrideFailure { string reason; };
    exception TargetOverrideFailure { string reason; };
    interface ServerInterceptorRequest
    {
        readonly attribute PortableServer::Servant orig_target;
        readonly attribute CORBA::Identifier orig_operation;
        readonly attribute PortableServer::Servant target;
        readonly attribute CORBA::Identifier operation;

        void next(inout Dynamic::RequestData data);
        void next_override(inout Dynamic::RequestData data,
                           in PortableServer::Servant target,
                           in CORBA::Identifier operation)
            raises(TargetOverrideFailure, OperationOverrideFailure);
    };

    interface ClientInterceptorRequest
    {
        readonly attribute Object orig_target;
        readonly attribute CORBA::Identifier orig_operation;
        readonly attribute Object target;
        readonly attribute CORBA::Identifier operation;

        void next(inout Dynamic::RequestData data);
        void next_override(inout Dynamic::RequestData data,
                           in Object target,
                           in CORBA::Identifier operation)
            raises(TargetOverrideFailure, OperationOverrideFailure);
    };

    interface ServerInterceptor
    {
        void invoke(in ServerInterceptorRequest req,
                    inout Dynamic::RequestData data);
    };

    interface ClientInterceptor
    {
        void invoke(in ClientInterceptorRequest req,
                    inout Dynamic::RequestData data);
        void send_oneway(in ClientInterceptorRequest req,
                         inout Dynamic::RequestData data);
        void send(in ClientInterceptorRequest req,
                  inout Dynamic::RequestData data);
        void get_response(in ClientInterceptorRequest req,
                          inout Dynamic::RequestData data)
    };
}

```

```
        raises(CORBA::WrongTransaction);
    boolean poll_response(in ClientInterceptorRequest req,
                          inout Dynamic::RequestData data)
        raises(CORBA::WrongTransaction);
    };
};
```

10.3 Message Level Interceptors

```

module POI
{
    exception CannotModify {};
    exception AlreadySet {};
    exception NotSupported {};
    exception NotFound {};
    exception InvalidServiceId {};

    typedef sequence<octet> OctetSeq;

    interface ServiceContext
    {
        IOP::ServiceContextList get_service_context_list();
        IOP::ServiceContext get_service_context
            ( in IOP::ServiceId service_id )
            raises ( NotFound, InvalidServiceId );
        void remove_service_context ( in IOP::ServiceId service_id )
            raises ( NotFound, CannotModify, InvalidServiceId );
        void add_service_context ( in IOP::ServiceContext service_context )
            raises ( CannotModify, InvalidServiceId, AlreadySet );
    };

    interface Message
    {
        readonly attribute GIOP::Version GIOP_version;
        readonly attribute octet message_type;
        readonly attribute octet flags;
        readonly attribute unsigned long message_size;
        readonly attribute unsigned long transport_endpoint;
        ServiceContext get_service_context()
            raises ( NotSupported );
    };

    interface ReplyMessage;

    interface RequestMessage : Message
    {
        readonly attribute boolean response_expected;
        readonly attribute string operation;
        readonly attribute unsigned long request_id;
        GIOP::TargetAddress get_target_address ()
            raises ( NotSupported );
        void set_target_address ( in GIOP::TargetAddress new_target )
            raises ( NotSupported, CannotModify );
        OctetSeq get_message_body ()
            raises ( NotSupported );
        void set_message_body ( in OctetSeq new_body )
            raises ( NotSupported, CannotModify );
    };
}

```

```
ReplyMessage create_reply
( in octet reply_status, in OctetSeq message_body )
raises ( NotSupported );
};

interface ReplyMessage : Message
{
    readonly attribute octet reply_status;
    readonly attribute unsigned long request_id;
    OctetSeq get_message_body ()
        raises ( NotSupported );
    void set_message_body ( in OctetSeq new_body )
        raises ( NotSupported, CannotModify );
};

interface CancelRequest : Message
{
    readonly attribute unsigned long request_id;
};

interface LocateReply;

interface LocateRequest : Message
{
    readonly attribute unsigned long request_id;
    OctetSeq get_object_key()
        raises ( NotSupported );
    GIOP::TargetAddress get_target ()
        raises ( NotSupported );
    LocateReply create_locate_reply
        ( in octet locate_status, in OctetSeq new_target )
        raises ( NotSupported );
};

interface LocateReply : Message
{
    readonly attribute unsigned long request_id;
    readonly attribute octet locate_status;
    OctetSeq get_locate_reply_body ()
        raises ( NotSupported );
    void set_locate_reply_body ( in OctetSeq new_target )
        raises ( NotSupported, CannotModify );
};

interface Fragment : Message
{
    readonly attribute unsigned long request_id;
    OctetSeq get_message_body ()
        raises ( NotSupported );
    void set_message_body ( in OctetSeq new_body )
        raises ( NotSupported, CannotModify );
};
```

```
};

interface MessageInterceptor
{
    enum InterceptionState
    {
        PROCEED_REVERSE,
        PROCEED_REGULAR
    };

    InterceptionState message ( in Message message_in,
                                out Message message_out );
};
```

10.4 *Interceptor Management*

```

module POI
{
    interface IORManagementInterceptor {};

    interface RequestLevelClientRegistry;
    interface RequestLevelServerRegistry;
    interface MessageLevelClientRegistry;
    interface MessageLevelServerRegistry;
    interface IORManagementRegistry;

    interface InterceptorRegistry
    {
        RequestLevelClientRegistry get_request_level_client_registry();

        RequestLevelServerRegistry get_request_level_server_registry();

        MessageLevelClientRegistry get_message_level_client_registry();

        MessageLevelServerRegistry get_message_level_server_registry();

        IORManagementRegistry get_ior_management_registry();
    };

    enum BindingType
    {
        ALWAYS,
        OBJECT,
        OBJECT_OPERATION,
        OPERATION,
        INTERFACE
    };

    interface InterceptorBinding
    {
        exception CANNOT_DESTROY { string reason; };

        readonly attribute BindingType binding_type;

        void destroy()
            raises ( CANNOT_DESTROY );

        void enable();
        void disable();
        boolean is_enabled();
    };

    typedef sequence<InterceptorBinding> BoundInterceptors;

```

```

enum InterceptorActivationScope
{
    LOCAL,
    REMOTE,
    ALWAYS
};

interface RequestLevelRegistryBase
{
    BoundInterceptors get_interceptors();

    void enable_interceptors ();
    void disable_interceptors ();
};

interface RequestLevelClientRegistry : RequestLevelRegistryBase
{
    InterceptorBinding
    bind_for_object (
        in ClientInterceptor req_inter,
        in Object obj,
        in InterceptorActivationScope act_scope );

    InterceptorBinding
    bind_for_object_operation (
        in ClientInterceptor req_inter,
        in Object obj,
        in CORBA::Identifier op_name,
        in InterceptorActivationScope act_scope );

    InterceptorBinding
    bind_for_interface (
        in ClientInterceptor req_inter,
        in CORBA::Identifier interface_repository_id,
        in InterceptorActivationScope act_scope );

    InterceptorBinding
    bind_for_operation (
        in ClientInterceptor req_inter,
        in CORBA::Identifier op_name,
        in InterceptorActivationScope act_scope );

    InterceptorBinding
    bind (
        in ClientInterceptor req_inter,
        in InterceptorActivationScope act_scope );
};

interface RequestLevelServerRegistry : RequestLevelRegistryBase
{
    InterceptorBinding

```

```
bind_for_object (
    in ServerInterceptor req_inter,
    in Object obj,
    in InterceptorActivationScope act_scope );

InterceptorBinding
bind_for_object_operation (
    in ServerInterceptor req_inter,
    in Object obj,
    in CORBA::Identifier op_name,
    in InterceptorActivationScope act_scope );

InterceptorBinding
bind_for_interface (
    in ServerInterceptor req_inter,
    in CORBA::Identifier interface_repository_id,
    in InterceptorActivationScope act_scope );

InterceptorBinding
bind_for_operation (
    in ServerInterceptor req_inter,
    in CORBA::Identifier op_name,
    in InterceptorActivationScope act_scope );

InterceptorBinding
bind (
    in ServerInterceptor req_inter,
    in InterceptorActivationScope act_scope );

};

interface MessageLevelRegistryBase
{
    InterceptorBinding
    bind_for_object (
        in MessageInterceptor mess_inter,
        in Object obj,
        in InterceptorActivationScope act_scope );

    InterceptorBinding
    bind (
        in MessageInterceptor mess_inter,
        in InterceptorActivationScope act_scope );

    BoundInterceptors get_interceptors();

    void enable_interceptors ();
    void disable_interceptors ();
};

interface MessageLevelClientRegistry : MessageLevelRegistryBase
```

```
{
};

interface MessageLevelServerRegistry : MessageLevelRegistryBase
{
};

interface IORManagementRegistry
{
    InterceptorBinding
    bind ( in IORManagementInterceptor ior_inter );

    BoundInterceptors get_interceptors();

    void enable_interceptors();
    void disable_interceptors();
};
};
```