Portable Interceptors

IBM Corporation

OMG Document orbos/99-04-04

April 26, 1999

Contents

1

Preface

1.1 Submitting Companies

The IBM Corporation is pleased to submit this specification in response to the CORBA Portable Interceptor RFP, or Request for Proposal (OMG Document orbos/98-09-11).

1.2 Submission Contact Points

Russell Butek
11400 Burnet Rd.
Internal Zip 9640
Austin, Texas 78758
USA
phone: 1-512-823-8268
email: butek@us.ibm.com

Alex McLeod
11400 Burnet Rd.
Internal Zip 9640
Austin, Texas 78758
USA
phone: 1-512-838-8182
email: amcleod@us.ibm.com

1.3 Status of this document

This is the initial proposal, document orbos/99-04-04, prepared in response to Portable Interceptors RFP, Request For Proposal, OMG document orbos/98-09-11.

1.4 Guide to this Submission

Chapter 1 provides contact information and a guide to this submission, statement of proof of concept, resolution of RFP mandatory and optional requirements, changes or extensions to adopted OMB specifications, and a discussion of “Issues to be Discussed”.

Chapter 2 provides the proposed specification, an architectural model of interceptors in the Object Request Broker, or ORB

Chapter 3 provides the proposed compliance points, and complete IDL definitions.

1.5 Proof of Concept

The IBM Corporation has used the model of System Request Interceptors defined in Chapter 2 of this document in the implementation of both a C++ ORB and a Java ORB used within IBM’s Component Broker product. Differences between this specification and the two implementations are primarily confined to changes made to operation names defined on the interceptors, and the technique defined for interceptor registration.

1.6 Changes or Extensions to Adopted OMG Specifications

The changes required of the existing OMG Specifications involve enhancements to the *resolve_initial_references* operation of the ORB interface.

Given an input string of “InterceptorRegistry”, *resolve_initial_references* will return an InterceptorRegistry object. The InterceptorRegistry interface is a new interface which provides the means by which RequestInterceptors can be registered with the ORB. See Registering Request Interceptors with the ORB on page 18.

1.7 Mandatory Requirements

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

   System and User Interceptors are discussed in this document.

   - See Request Interceptors - System Service Interceptors on page 11
   - See Request Interceptors - User Interceptors on page 26

   This document also defines a MessageInterceptor, which is used by Security Services

   - See The MessageInterceptor Interface on page 31

2. 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.

   - The System level Interceptor implementation defined in this document can be tailored to include a range of intercept points based on the interfaces that are sub-classed. See Client Side Request Interceptors on page 20, Server Side Request Interceptors on page 23, as well as The Number of System Level Request Interceptor Execution Points on page 44.
   - For discussion of how the ORB handles errors in interceptor processing, see SystemException Client Side Processing on page 22, UserException Client Side Processing on page 22, System Exception Server Side Processing on page 25, and UserException Server Side Processing on page 25.

3. 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 interceptors of such operations to system interceptors.

   - The mechanism through which the System Interceptor can query and set pertinent attributes associated with the Request is discussed in the section titled the RequestHolder Interface on page 13.

4. 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.

   - User Interceptors are discussed in Request Interceptors - User Interceptors on page 26

5. 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).
   - See Overview of Request Interceptor Execution Points on page 17
   - See Client Side Interceptor Execution points on page 21, and Server Side Interceptor Execution Points on page 24.
6. A submissions 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 interceptors is required. The analysis will not be normative for these 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.
   - See Generic Flow Showing System Interceptor Processing on page 33
   - Security Service Use of Interceptors on page 35
   - Transactions Service Use of Interceptors on page 39
7. 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 issuing a request using the negotiated security.
   - The Request Interceptor architecture defined here does not make special provisions for recursive calls. The Security and Transactions services implementations built upon this architecture maintain state data within the interceptor implementation to control this aspect. See Security Service Use of Interceptors on page 35, and Transactions Service Use of Interceptors on page 39.
8. A submission shall specify administrative interfaces including, at a minimum, registering and un-registering interceptors, both system and user.
   - Registering Request Interceptors with the ORB on page 18
   - Request Interceptors - User Interceptors on page 26

1.8 Optional Items

None.

1.9 Issues to be discussed

1. Submissions shall discuss the security of the interceptor architecture, that is, who is allowed to add system interceptors to the ORB and under what conditions. For system interceptors, this might state that adding interceptors is controlled administratively. For user level interceptors, the submission shall discuss what considerations are needed to protect the ORB and the service contexts of a request.

   - Adding System level interceptors is controlled administratively. This first draft does not address any additional security aspects to control who can create and register a System level interceptor. This will be discussed in a future draft of this document.
   - The User level Request Interceptors have the ability to query aspects of the request, but not update the request. See Request Interceptors - User Interceptors on page 26

2. By the time the first submissions are due, it is likely that the submissions for the ORB related Firewall and Messaging RFPs will have reached final approval, as will have several modifications to GIOP. These changes may add new CORBA Core and GIOP capabilities not present when CORBA 2.2 Interceptors were first specified. If so, submissions should take the new specifications into account. For example,
   - The Messaging RFP introduces a new programming invocation model, asynchronous invocation with the reply being a call-back invocation directed to another process or machine. How does this affect the presumed symmetry of interceptors in CORBA 2.2?
   - The Firewall RFP and the Interop RTFs may allow redirection of a request to an address different from that bound to originally. Does this make a difference?

   This initial proposal does not address recent changes involving the Messaging and Firewall RFPs mentioned above. These will be discussed in the revised proposal.

Portable Interceptors 2

2.1 Introduction

This document presents an architectural model of interceptors in the Object Request Broker, or ORB.

Request Interceptors allow various services associated with request processing to monitor the requests as they flow through the ORB, as well as exceptions when they occur. A Message Interceptor is another specialized type of interceptor required by Security Services.

This document will discuss architectural models for both Request Interceptors and Message Interceptors.

2.2 Request Interceptors - System Service Interceptors

2.2.1 Overview

System services associated with the processing of a request must be able to intercept a request at various points as the request flows through the ORB. A request related interceptor is implemented by a service provider and is invoked by the ORB during ORB processing to provide that service.

Examples of the types of services which would implement Request Interceptors to register for use in conjunction with the ORB, include the following:

- Security Services
- Transactions Services
- Reliability and Serviceability (RAS), Trace Facilities
- Session Services
- Code Set Management, etc.

Each system level Request Interceptor must be instantiated and registered with the ORB. Enhancements to the *resolve_initial_references* operation of the CORBA::ORB interface have been proposed to provide access to an instance of an InterceptorRegistry object through which the system level Request Interceptor registration and subsequent un-registration process is performed.

The ORB make calls out to the defined operations on the registered Request Interceptors at various stages or monitoring points during the processing of a request by the ORB. The monitoring points at which time the Request Interceptor operations are invoked by the ORB are well defined.

The stages in the ORB request processing during which the Request Interceptors are given control are based upon the state of the request data, namely either marshalled data (wire format) or un-marshalled data (non-wire format):

- Client Side Request Interceptor processing

  The client side Request Interceptors can get access to the request both before and after the client side ORB marshals the request in preparation for sending the request to the server.

  The client side Request Interceptors can get access to the response to the request both in its marshalled state upon arrival from the server and after it has been de-marshalled by the client side ORB.

  The client side Request Interceptors can also get access to the request/response in the event either a CORBA::SystemException or CORBA::UserException has been thrown.

- Server Side Request Interceptor processing

  The server side Request Interceptors can get access to the request both in its marshalled state upon arrival at the server, and after the request is de-marshalled by the server side ORB to its un-marshalled state.

  The server side Request Interceptors can get access to the response to the request both before and after the server side ORB marshals the response in preparation for sending it back to the client.

  The server side Request Interceptors can also get access to the request/response in the event either a CORBA::SystemException or CORBA::UserException has been thrown.

Each system level Request Interceptor must be unregistered with the ORB prior to the removal of the Request Interceptor instance.

2.2.2 RequestHolder Interface

Each RequestInterceptor method that is called must have a means to query and/or modify pertinent attributes associated with the request. The RequestHolder interface provides the means by which request related information may be accessed or modified. An instance of a RequestHolder object is passed as the input parameter to each System RequestInterceptor method.

The *RequestHolde*r is a locally constrained interface that is basically a wrapper for the Request that provides query and update methods. Two interfaces named *RequestHolder_ge*t and *RequestHolder_set* are defined to provide the "get/set" operations on the RequestHolder. Following is the IDL for RequestHolder_get, RequestHolder_set, and RequestHolder.

Interface Definition for RequestHolder_get

```
interface RequestHolder_get {
    typedef  sequence <octet> request_message;
    typedef  ReferenceData     ObjectKey;

    request_message            get_requestMessage ();
    string                     get_operation ();
    Context                    get_context ();
    NVList                     get_arguments ();
    NamedValue                 get_result ();
    ExceptionList              get_exceptions ();
    ContextList                get_contextlist ();
    boolean                    get_oneWay ();
    unsigned long              get_requestId ();
    ObjectKey                  get_objectKey ();
    Principal                  get_principal ();
    IOP::IOR                   get_ior ();
    IOP::ServiceContextList    get_serviceContextList ();
    Object                     get_proxy ();
    Object                     get_target ();
    boolean                    get_forceRetry ();
};
```

Interface Definition for RequestHolder_set

```
interface RequestHolder_set {
      typedef  sequence <octet> request_message;
      typedef  ReferenceData      ObjectKey;

      void                          set_requestId (in unsigned long rid);
      void                          set_objectKey (in ObjectKey objkey);
      void                          set_principal (in Principal p);
      void                          set_ior (in IOP::IOR ior);
      void                          set_serviceContextList
                                          (in IOP::ServiceContextList scl);
      void                          set_forceRetry
                                          (in Boolean forceRetryValue fr);
};
```

The *RequestHolder_get* and *RequestHolder_se*t operations defined above provide the *RequestHolder* with the ability to get/set qualities normally associated with a Request

The *RequestHolder* interface defines 3 operations in addition to those provided via *RequestHolder_get* and *RequestHolder_set* that allow a System Service the ability to set/get information unique to the Service (see the following IDL, and the description that follows).

Interface Definition for RequestHolder

```
interface RequestHolder : RequestHolder_get, RequestHolder_set {
        unsigned long       get_ServiceDataKey ();
        void                set_ServiceData
                              (in unsigned long service_data_key,
                               in any          service_data);
        any                 get_ServiceData
                              (in unsigned long service_data_key);
};
```

get_ServiceDataKey

This operation returns to the System Service a “ServiceDataKey”, which can then be used to “set” or associate the ServiceData with the RequestHolder.

Parameters

none

set_ServiceData

This operation sets or associates Service specific service data with the RequestHolder, associating the data with a key previously obtained from *get_ServiceDataKey.*

Parameters

service_data_key	This parameter is the service_data_key with which the service_data will be associated once this operation is completed.
service_data	This parameter is an Any data type which contains the Service specific data.

get_ServiceData

This operation retrieves the Service specific service data from the RequestHolder using the service_data_key as the search key. An Any data type is returned which contains the service data.

Parameters

service_data_key	This parameter is the service_data_key used to locate the service data.

2.2.3 The RequestInterceptor Interface

The RequestInterceptor is the base interface from which the client Request Interceptor and server Request Interceptor interfaces subclass. The RequestInterceptor is a locally constrained interface, the interface definition for which follows:

```
interface RequestInterceptor {
 };
```

The concept of symmetry across client and server request interceptor methods is defined with respect to which type of data the interceptor method is dealing with, namely whether the data is in a marshalled (wire format) or non-marshalled state (non-wire format).

There are six interfaces which subclass from the RequestInterceptor interface to provide adequate granularity of RequestInterceptor functionality. Three of these interfaces are described in the section titled Client Side Request Interceptors on page 20:

1. **clientNonMarshalledDataRI** interface
   - *client_nonmarshalled_request* operation
   - *client_nonmarshalled_response* operation
2. **clientMarshalledDataRI** interface
   - *client_marshalled_request* operation
   - *client_marshalled_response* operation
3. **clientExceptionRI** interface
   - *client_system_exception* operation
   - *client_user_exception* operation

while the remaining three are defined in the section titled Server Side Request Interceptors on page 23:

4. **serverMarshalledDataRI** interface
   - *server_marshalled_request* operation
   - *server_marshalled_response*operation
5. **serverNonMarshalledDataRI** interface
   - *server_nonmarshalled_request* operation
   - *server_nonmarshalled_response* operation
6. **serverExceptionRI** interface
   - *server_system_exception* operation
   - *server_user_exception* operation

Overview of Request Interceptor Execution Points

The following flow shows the order in which the various execution points of the System RequestInterceptors (Sys RI) are given control by the ORB:

```
Client Side                                Server Side

(1) client_nonmarshalled_request (Sys RI)
      . . . request is marshalled . . .
(2) client_marshalled_request (Sys RI)
      . . . request is sent to server . . .
-------------------------------------------------------------------->
                                           (3) server_marshalled_request (Sys Int)
                                                 . . . request is demarshalled . . .
                                           (4) server_nonmarshalled_request (Sys Int)
                                                 . . . request processing . . .
                                           (5) server_nonmarshalled_response (Sys Int)
                                                 . . . response is marshalled . . .
                                           (6) server_marshalled_response (Sys Int)

                                           (following are invoked when exception type occurs)
                                           server_system_exception (Sys Int)
                                           server_user_exception (Sys Int)
                                                 . . . response is sent to client . . .
<--------------------------------------------------------------------
7) client_marshalled_response (Sys RI)
      . . . response is demarshalled . . .
8) client_nonmarshalled_response (Sys RI)

(following are invoked when the exception type occurs)
client_system_exception (Sys RI)
client_user_exception (Sys RI)
```

2.2.4 Registering Request Interceptors with the ORB

A new interface has been introduced called “InterceptorRegistry” with which to register Request Interceptors.

The *resolve_initial_references* operation of the ORB interface is enhanced to return an instance of an InterceptorRegistry object (given “InterceptorRegistry” as the input string). Once this InterceptorRegistry object is obtained, Request Interceptors can be registered and un-registered with the ORB.

The IDL snapshot for InterceptorRegistry follows:

```
interface InterceptorRegistry {
    void register_RequestInterceptor
            (in RequestInterceptor          requestInterceptor,
             in boolean                     makeMeFirst);

    void unregister_RequestInterceptor
            (in RequestInterceptor          requestInterceptor);
};
```

register_RequestInterceptor

This operation registers the RequestInterceptor with the ORB. The target RequestInterceptor will then have access to the ORB monitoring points based upon the sub-classes of RequestInterceptor chosen in the implementation:

- clientNonMarshalledDataRI
- clientMarshalledDataRI
- clientExceptionRI
- serverNonMarshalledDataRI
- serverMarshalledDataRI
- serverExceptionRI

Parameters

requestInterceptor	This parameter is the RequestInterceptor which is to be registered with the ORB.
makeMeFirst	This parameter is a Boolean that when set to 1 indicates the requirement that this RequestInterceptor must be the first in the ordered list of registered RequestInterceptors. Once a RequestInterceptor has been successfully registered with this parameter set to 1, any subsequent registration attempts requesting this feature will generate a BAD_PARAM SystemException. Registration attempts with this parameter set to 0 will place the RequestInterceptor in the list in the order in which the registration request is received.

Note – The makeMeFirst option above is a requirement imposed by Security Services.

unregister_RequestInterceptor

This operation un-registers a RequestInterceptor with the ORB such that it no longer has access to any ORB RequestInterceptor related execution points.

Parameters

requestInterceptor	This parameter is the RequestInterceptor which is to be un-registered with the ORB. A BAD_PARAM SysemException results if the input RequestInterceptor is not in a registered state.

When the RequestInterceptor is registered with the ORB, the RequestInterceptor instance is placed on one or more ordered lists within the ORB. The ORB will then invoke the RequestInterceptor related methods at the appropriate times during the request/reply processing. This is explained in more detail for the client and server sides in Client Side Request Interceptors on page 20, and Server Side Request Interceptors on page 23, respectively.

When the service no longer wishes to intercept ORB requests, it will call the unregister_RequestInterceptor operation on the InterceptorRegistry instance

The ORB must maintain lists of the Request Interceptors that have been registered for a particular execution point. The ORB implementation must assure that the registration and un-registration of Request Interceptors does not affect the successful operation of Request Interceptors that have already been registered with the ORB.

2.2.5 Client Side Request Interceptors

The concept of symmetry across client request interceptor methods is defined with respect to which type of data the interceptor method is dealing with, namely whether the data is in a marshalled (wire format) or non-marshalled state (non-wire format).

The **clientNonMarhshalledDataRI** interface is intended for request interceptor implementations that are interested in processing the request in its non-marshalled state. It provides the client side service access during both the request and response level processing.

The **clientMarhshalledDataRI** interface is intended for request interceptor implementations that are interested in processing the request in its marshalled state. It provides the client side service access during both the request and response level processing.

The **clientExceptionRI** interface is intended for request interceptor implementations that are interested in attaining access to a request/reply in the event either a SystemException or a UserException takes place.

Six operations are distributed across these three locally constrained interfaces, the interface definitions for which follow:

Interface Definition for clientNonMarshalledDataRI

```
interface clientNonMarshalledDataRi : RequestInterceptor {
    void client_nonmarshalled_request        (in RequestHolder rh);
    void client_nonmarshalled_response       (in RequestHolder rh);
};
```

Interface Definition for clientMarshalledDataRI

```
interface clientMarshalledDataRi : RequestInterceptor {
    void client_marshalled_request        (in RequestHolder rh);
    void client_marshalled_response       (in RequestHolder rh);
};
```

Interface Definition for clientExceptionRI

```
interface clientExceptionRi : RequestInterceptor {
    void client_system_exception        (in RequestHolder rh,
                                         in ClientFlow cf);
    void client_user_exception          (in RequestHolder rh,
                                         in ClientFlow cf);
};
```

Client Side Interceptor Execution points

The implementor of a client side request interceptor can subclass from the appropriate interfaces depending upon the desired execution points, and override the associated operations. The request interceptor instance is then registered with the ORB (see "Registering Request Interceptors with the ORB"). The ORB logically maintains three ordered lists of client side RequestInterceptor instances, one list for each of the three client RequestIntercepor interfaces. Only those interceptors which implement a particular interception point will be called for that point.

The client side ordered list of RequestInterceptor instances is processed by the ORB in the following manner:

- client side requests

  During client requests, the ORB will invoke the corresponding methods on all RequestInterceptor instances in the reverse of the registration list order, unless an exception is thrown.
- client side responses

  For client responses, the ORB will invoke the corresponding methods on all registered RequestInterceptors in the order in which they appear in the registration list, unless an exception is thrown.
- Exceptions

  In the event a SystemException or UserException occurs, the list is also processed in the registration list order.

The six client side RequestInterceptor operations are invoked by the ORB in the following sequence of execution points:

1. **client_nonmarshalled_request**

   Access is given to an Outgoing Request, before it has been marshalled.
2. **client_marshalled_request**

   Access is given to an Outgoing Request, after it has been marshalled.
3. **client_marshalled_response**

   Access is given to an Incoming Response, before it is de-marshalled.
4. **client_nonmarshalled_response**

   Access is given to an Incoming Response, after it is de-marshalled.
5. **client_system_exception**

   Access is given when a SystemException has been thrown
6. **client_user_exception**

   Access is given when a UserException has been thrown.

SystemException Client Side Processing

- If a SystemException occurs anywhere on the client during a request, or a SystemException is returned from the server, the flow is interrupted and control is immediately transferred to the list of RequestInterceptors that are registered for **client_system_exception** processing.
- If the SystemException is thrown while processing a list of non-SystemException RequestInterceptors, no other RequestInterceptors on that list are invoked and control is immediately transferred to the **client_system_exception** list.
- The entire **client_system_exception** list is invoked when any SystemException occurs
- No **client_system_exception** operation should itself throw a SystemException.
- The **client_system_exception** operations are given access to an additional parameter, an instance of a **ClientFlow** interface. The ClientFlow object provides information as to how far the processing of the request proceeded prior to the SystemException being thrown. The interface definition for the locally constrained ClientFlow interface is as follows:

Interface Definition for ClientFlow

```
interface ClientFlow {
    enum client_completion_status {YES, NO, FAILED};

    short client_non_marshalled_requestCalled ();
    short client_marshalled_requestCalled ();
    short client_marshalled_responseCalled ();
    short client_non_marshalled_responseCalled ();
};
```

UserException Client Side Processing

- When a UserException is generated via the interface implementation at the server, the UserException is passed back to the client and thrown at the client side. Prior to the UserException being thrown, the **client_user_exception** operation is called for each RequestInterceptor that is registered to handle Exceptions.
- No **client_user_exception** operation should itself throw a SystemException.

2.2.6 Server Side Request Interceptors

The concept of symmetry across server request interceptor methods is defined with respect to which type of data the interceptor method is dealing with, namely whether the data is in a marshalled (wire format) or non-marshalled state (non-wire format).

The **serverMarhshalledDataRI** interface is intended for request interceptor implementations that are interested in processing the request in its marshalled state. It provides the server side service access during both the request and response level processing.

The **serverNonMarhshalledDataRI** interface is intended for request interceptor implementations that are interested in processing the request in its non-marshalled state. It provides the server side service access during both the request and response level processing.

The **serverExceptionRI** interface is intended for request interceptor implementations that are interested in attaining access to a request/reply in the event either a SystemException or a UserException takes place.

Six operations are distributed across these three locally constrained interfaces, the interface definitions for which follow:

Interface Definition for serverMarshalledDataRI

```
interface serverMarshalledDataRi : RequestInterceptor {
    void server_marshalled_request        (in RequestHolder rh);
    void server_marshalled_response       (in RequestHolder rh);
};
```

Interface Definition for serverNonMarshalledDataRI

```
interface serverNonMarshalledDataRI : RequestInterceptor {
    void server_nonmarshalled_request        (in RequestHolder rh);
    void server_nonmarshalled_response       (in RequestHolder rh);
    };
```

Interface Definition for serverExceptionRI

```
interface serverExceptionRi : RequestInterceptor {
    void server_system_exception      (in RequestHolder rh,
                                       in ServerFlow sf);
    void server_user_exception        (in RequestHolder rh,
                                       in ServerFlow sf);
};
```

Server Side Interceptor Execution Points

The implementor of a server side request interceptor can subclass from the appropriate interfaces depending upon the desired execution points, and override the associated operations. The request interceptor instance is then registered with the ORB (see "Registering Request Interceptors with the ORB"). The ORB logically maintains three ordered lists of server side RequestInterceptor instances, one list for each of the three server RequestIntercepor interfaces. Only those interceptors which implement a particular interception point will be called for that point.

The server side ordered list of RequestInterceptor instances is processed by the ORB in the following manner:

- server side requests

  When a request is received at the server, the ORB will invoke the corresponding methods on all RequestInterceptor instances in the registration list order, unless an exception is thrown.
- server side responses

  For server responses back to the client, the ORB will invoke the corresponding methods on all registered RequestInterceptors in the reverse order in which they appear in the registration list, unless an exception is thrown.
- Exceptions

  In the event a SystemException or UserException occurs, the list is also processed in the reverse of the registration list order.

The six server side RequestInterceptor operations are invoked by the ORB in the following sequence of execution points:

1. **server_marshalled_request**

   Access is given to an Incoming Request, before it is de-marshalled.

2. **server_nonmarshalled_request**

   Access is given to an Incoming Request, after it is de-marshalled.

3. **server_nonmarshalled_response**

   Access is given to an Outgoing Response, before it is marshalled.

4. **server_marshalled_response**

   Access is given to an Outgoing Response, after it has been marshalled.

5. **server_system_exception**

   Access is given when a SystemException has been thrown.

6. **server_user_exception**

   Access is given when a UserException has been thrown

System Exception Server Side Processing

- If a SystemException occurs anywhere on the server during a request, the flow is interrupted and control is immediately transferred to the list of RequestInterceptors that are registered for **server_system_exception** processing.
- If the SystemException is thrown while processing a list of non-SystemException RequestInterceptors, no other RequestInterceptors on that list are invoked and control is immediately transferred to the **server_system_exception** list.
- The entire **server_system_exception** list is invoked when any SystemException occurs
- No **server_system_exception** operation should itself throw a SystemException.
- The **server_system_exception** operations are given access to an additional parameter, an instance of a **ServerFlow** interface. The ServerFlow object provides information as to how far the processing of the request proceeded prior to the SystemException being thrown. The interface definition for the locally constrained ServerFlow interface is as follows:

Interface Definition for ServerFlow

```
interface ServerFlow {
    enum server_completion_status  {YES, NO, FAILED};

    short server_marshalled_requestCalled ();
    short server_non_marshalled_requestCalled ();
    short server_non_marshalled_responseCalled ();
    short server_marshalled_responseCalled ();
};
```

UserException Server Side Processing

- When a UserException is generated via the interface implementation at the server, the **server_user_exception** operation is called for each RequestInterceptor that is registered to handle server side Exceptions. The UserException is then passed back to the client in response to the request.
- No **server_user_exception** operation should itself throw a SystemException.

2.3 Request Interceptors - User Interceptors

2.3.1 Overview

The User Level Request Interceptor allows the client and server access to the request at specific execution points in the request processing. The primary differences between the User level Interceptor and the System Service Interceptor discussed previously involve the execution points and the powers assigned at those execution points.

Once the User RequestInterceptor has been registered with the ORB, the User RequestInterceptor will have the ability to access the request or response:

- before the first System RequestInterceptor has processed the request
- after the last System RequestInterceptor has processed the request

Through operations on the RequestHolder interface, the System RequestInterceptor has the ability to both query and change qualities associated with the request. However, the input to the User RequestInterceptor is a modified form of the RequestHolder interface called the UserRequestHolder interface.

The UserRequestHolder interface allows the User RequestInterceptor access to all of the “get” methods defined in the RequestHolder used by the System RequestInterceptors, but non of the “set” methods. This allows the User RequestInterceptor to evaluate and track the request, without permitting it to alter the request/response directly.

2.3.2 UserRequestHolder Interface

Each UserRequestInterceptor method that is called may require the means to query pertinent attributes associated with the request. The UserRequestHolder interface provides the means by which request related information may be accessed. An instance of a UserRequestHolder object is passed as the input parameter to each UserRequestInterceptor method.

The UserRequestHolder is a locally constrained interface that is basically a wrapper for the Request that provides query methods. The following section provides the IDL specification for the UserRequestHolder interface (see the Interface Definition for RequestHolder_get on page 13).

```
interface UserRequestHolder:RequestHolder_get {
};
```

2.3.3 Registering User Request Interceptors with the ORB

The User Request Interceptor is registered with the ORB in the same fashion as a System Request Interceptor. See the discussion in Registering Request Interceptors with the ORB on page 18.

2.3.4 Client Side and Server Side User Request Interceptors

The **clientSideRequestURI** interface is intended for user request interceptor implementations that are interested in processing before and after System Request Interceptors have processed an outgoing request.

The **clientSideResponseURI** interface is intended for user request interceptor implementations that are interested in processing before and after System Request Interceptors have processed an incoming response.

The **serverSideRequestURI** interface is intended for user request interceptor implementations that are interested in processing before and after System Request Interceptors have processed an incoming request.

The **serverSideResponseURI** interface is intended for user request interceptor implementations that are interested in processing before and after System Request Interceptors have processed an outgoing response.

Following are the interface definitions for the User RequestInterceptors:

Interface Definition for clientSideRequestURI

```
interface clientSideRequestURI : RequestInterceptor {
    void client_pre_request                    (in RequestHolder ri);
    void client_post_request                   (in RequestHolder ri);
};
```

Interface Definition for clientSideResponseURI

```
interface clientSideResponseURI : RequestInterceptor {
    void client_pre_response                (in RequestHolder ri);
    void client_post_response               (in RequestHolder ri);
};
```

Interface Definition for serverSideRequestURI

```
interface serverSideRequestURI : RequestInterceptor {
    void server_pre_request                 (in RequestHolder ri);
    void server_post_request                (in RequestHolder ri);
};
```

Interface Definition for serverSideResponseURI

```
interface serverSideResponseURI : RequestInterceptor {
    void server_pre_response            (in RequestHolder ri);
    void server_post_response           (in RequestHolder ri);
};
```

2.3.5 ORB Intercept Points for User Request Interceptors

The following flow illustrates the User Request Interceptor (User RI) execution points in relationship to those already define for System Request Interceptors.

<u>**Client Side**</u> <u>**Server Side**</u>

client_pre_request (User RI)

(1) client_nonmarshalled_request (Sys RI)

. . . request is marshalled . . .

(2) client_marshalled_request (Sys RI)

client_post_request (User RI)

. . . request is sent to server . . .

--->

server_pre_request (User RI)

(3) server_marshalled_request (Sys Int)

. . . request is demarshalled . . .

(4) server_nonmarshalled_request (Sys Int)

server_post_request (User RI)

. . . request processing . . .

server_pre_reply (User RI)

(5) *server_nonmarshalled_response (Sys Int)*

. . . response is marshalled . . .

(6) *server_marshalled_response (Sys Int)*

server_post_reply (User RI)

server_system_exception (Sys Int)

server_user_exception (Sys Int)

. . . response is sent to client . . .

<---

client_pre_reply (User RI)

7*) client_marshalled_response (Sys RI)*

. . . response is demarshalled . . .

8) client_nonmarshalled_response (Sys RI)

client_post_reply (User RI)

client_system_exception (Sys RI)

client_user_exception (Sys RI)

2.4 Message Interceptors

2.4.1 Overview

Like Request Interceptors, Message Interceptors are called for each request that is processed by the ORB. However, a Message Interceptor is distinctly different in concept from a Request Interceptor, and the Message Interceptor operations are called from different execution points during the ORB request processing. Only one Message Interceptor can be registered with the ORB. The single Message Interceptor instance monitors the marshalled IIOP request just before it is sent (after all RequestInterceptor processing has completed). The Message Interceptor also monitors all incoming replies as they are received, before any other processing takes place (including RequestInterceptor intervention).

Currently, the only service which has an implementation of a Message Interceptor is Security Services.

2.4.2 The MessageHolder Interface

Each MessageInterceptor method that is called must have a means to query pertinent attributes associated with the request. The RequestHolder interface used by the RequestInterceptor interface provides part of the input to MessageInterceptor operations. MessageInterceptor operations also require an interface known as the MessageHolder, which is a wrapper for a byte array. This interface provides the Security MessageInterceptor operations the means to pass back to the caller a new byte array.

Following is the IDL specification for the locally constrained MessageHolder interface.

Interface Definition for MessageHolder

```
interface MessageHolder {
    typedef sequence <octet>       message_buf;
    attribute message_buf          message;
};
```

2.4.3 The MessageInterceptor Interface

The MessageInterceptor is an interface that defines the Message Interceptor support, and must be sub-classed by the implementor. The interface definition for the locally constrained MessageInterceptor interface is as follows:

```
interface MessageInterceptor {
    Void    send_request_message
             (inout MessageHolder     message,
            in RequestHolder           request);

    Void    receive_response_message
            (inout MessageHolder      message);

    Void    receive_request_message
            (inout MessageHolder      message);

    Void    send_response_message
            (inout MessageHolder      message,
            in RequestHolder           request);
};
```

These four methods provide the following functionality:

1. **send_request_message**

   This operation intercepts IIOP requests as they are sent by the client. This allows the Security Service MessageInterceptor at the client to encrypt the buffer prior to sending it to the server.

2. **receive_response_message**

   This operation intercepts IIOP responses as they arrive at the client. This allows the Security Service MessageInterceptor at the client to decrypt the response buffer as it is received from the server.

3. **receive_request_messag**e

   This operation intercepts IIOP requests as they arrive at the server. This allows the Security Service MessageInterceptor at the server to decrypt the request buffer as it is received from the client

4. **send_response_message**

   This operation intercepts IIOP responses as they leave the server. This allows the Security Service MessageInterceptor at the server to encrypt the response buffer as it is sent back to the client.

2.4.4 Registering Message Interceptors with the ORB

A new interface has been introduced called "InterceptorRegistry" with which to register Message Interceptors (as well as Request Interceptors).

The *resolve_initial_references* operation of the ORB interface is enhanced to return an instance of an InterceptorRegistry object (given "InterceptorRegistry" as the input string). Once this InterceptorRegistry object is obtained, Message Interceptors can be registered and un-registered with the ORB.

The IDL snapshot for the InterceptorRegistry interface follows:

```
interface InterceptorRegistry {
    void register_MessageInterceptor
            (in MessageInterceptor          messageInterceptor);

    void unregister_MessageInterceptor
            (in MessageInterceptor          messageInterceptor);
};
```

register_MessageInterceptor

Parameters

messageInterceptor	This parameter is the MessageInterceptor instance which is to be registered with the ORB. Only one MessageInterceptor instance is allowed to register with the ORB. A BAD_PARAM SystemException will be thrown for any MessageInterceptor registration attempt subsequent to the initial successful registration.

unregister_MessageInterceptor

messageInterceptor	This parameter is the MessageInterceptor instance which is to be un-registered with the ORB. A BAD_PARAM SystemException will be thrown if the input MessageInterceptor is not already registered with the ORB.

2.5 Discussion of Flows Involving Request/Message Interceptors

The design points of System and User RequestInterceptors and the Message Interceptor have been discussed above. Following are two sections which will help illustrate how these Interceptors are used in conjunction with the ORB.

2.5.1 Generic Flow Showing System Interceptor Processing

The following flow shows the order in which the various execution points of the System RequestInterceptors (Sys RI) and Message Interceptors (Msg Int) are given control by the ORB:

```
Client Side                               Server Side


(1) client_nonmarshalled_request (Sys RI)
    . . . request is marshalled . . .
(2) client_marshalled_request (Sys RI)

(3) send_request_message (Msg Int)
-------------------------------------------------------------------->

                                          4) receive_request_message (Msg Int)

                                          (5) server_marshalled_request (Sys Int)
                                              . . . request is demarshalled . . .
                                          (6) server_nonmarshalled_request (Sys Int)
                                              . . . request processing . . .
                                          (7) server_nonmarshalled_response (Sys Int)
                                              . . . response is marshalled . . .
                                          (8) server_marshalled_response (Sys Int)

                                          server_system_exception (Sys Int)
                                          server_user_exception (Sys Int)

                                          (9) send_response_message (Msg Int)
<--------------------------------------------------------------------
```

Client Side	**Server Side**
10) receive_response_message (Msg Int)	
11) client_marshalled_response (Sys RI)	
. . . response is demarshalled . . .	
12) client_nonmarshalled_response (Sys RI)	
client_system_exception (Sys RI)	
client_user_exception (Sys RI)	

2.5.2 Security Service Use of Interceptors

A possible implementation of the Security Service can be mapped into the ORB's RequestInterceptor/MessageInterceptor design in the following manner:

1. **Security Service RequestInterceptors**

   The Security Service implements both a Client Side System RequestInterceptor and a Server Side RequestInterceptor.

2. **Client Side RequestInterceptor Implementation**

   At the Client Side, the Security Service wishes to intercept the Request in its non-marshalled state (before the request is marshalled into its wire format to be sent to the server). When the response to the request is received from the server, Security wishes to intercept the response after it has been de-marshalled (into its non-marshalled state). The Client Side RequestInterceptor implementation therefore inherits from the:

   clientNonMarshalledDataRI

   interface which provides the ***client_nonmarshalled_request*** and ***client_nonmarshalled_response*** request interceptor operations.

3. **Server Side Request Interceptor Implementation**

   At the Server Side, the Security Service wishes to intercept the request at one point, upon receipt of the request from the client after it has been de-marshalled into its non-wire, or non-marshalled format. The Server Side RequestInterceptor implementation therefore inherits from the:

   serverNonMarshalledDataRI

interface, such that a Security Service implementation of the ***server_nonmarshalled_request*** operation can be created.

4. **Registering the Security Service RequestInterceptors**

   Security Services registers both the client side RequestInterceptor and the server side RequestInterceptor implementations via the ***register_RequestInterceptor*** operation (see Registering Request Interceptors with the ORB on page 18). Security Services utilizes the ***makeMeFirst*** boolean parameter to assure that each interceptor is registered first in the respective RequestInterceptor lists, a requirement of Security Services.

5. **Implementation and Registration of the Security Service Message Interceptor**

   Security Services provides an implementation of a MessageInterceptor, which handles the encryption/de-cryption of data between the client and server. The MessageInterceptor implementation is registered with the ORB via the ***register_MessageInterceptor*** operation (see Registering Message Interceptors with the ORB on page 32).

The Security Service has now provided client and server side RequestInterceptor implementations, as well as a MessageInterceptor implementation, and registered these with the ORB. The following diagram illustrates the Security Service flow for the round trip of the request/reply through the ORB (the bold entries in the flow are the Interceptor execution points at which Security gets control, the others are not implemented by Security Services):

Client Side **Server Side**

(1) client_nonmarshalled_request *(Sys RI)*

. . . request is marshalled . . .

(2) client_marshalled_request (Sys RI)

(3) send_request_message *(Msg Int)*

--->

4) receive_request_message *(Msg Int)*

(5) server_marshalled_request (System RI)

. . . request is demarshalled . . .

(6) server_nonmarshalled_request *(Sys Int)*

. . . request processing . . .

(7) server_nonmarshalled_response (Sys RI)

. . . response is marshalled

(8) server_marshalled_response (Sys RI)

server_system_exception (Sys RI)

server_user_exception (Sys RI)

(9) send_response_message *(Msg Int)*

<---

Client Side	Server Side
10) receive_response_message *(Msg Int)*	
11) client_marshalled_response (Sys RI)	
. . . response is demarshalled . . .	
12) client_nonmarshalled_response*(Sys RI)*	
client_system_exception (Sys RI)	
client_user_exception (Sys RI)	

The following is a synopsis of the Security Service processing performed at the bold execution points in the previous diagram:

- at the ***1) client_nonmarshalled_request*** RequestInterceptor execution point:

  The Security Service here uses the TaggedComponent to obtain the server's secure name. It in turn generates a _NON_EXISTENT method call to the target server. Security Services maintains state information within its RequestInterceptor implementation with which to handle the recursive call. Security Session ID information is added to the ServiceContextList associated with the request.

- at the ***3) send_request_message*** MessageInterceptor execution point:

  The client side Security Service MessageInterceptor accesses information in the ServiceContextList to determine encryption requirements, and the message is sent to the server.

- at the ***4) receive_request_message*** MessageInterceptor execution point

  The server side Security Service MessageInterceptor accesses the ServiceContextList to determine decryption requirements as the message is received.

- at the ***6) server_nonmarshalled_request*** RequestInterceptor execution point:

  Security Service uses the ServiceContextList to obtain the SessionId sent by the client. If the session id is in the table maintained by Security Service, the processing is complete. If the session id is not, interaction is made with DCE to complete the session information prior to inserting the new entry in the session table.

- at the ***9) send_response_message*** MessageInterceptor execution point

  The server side Security Service MessageInterceptor accesses the ServiceContextList to determine encryption requirements as the response message is sent back to the client.

- at the ***10) receive_response_message*** MessageInterceptor execution point

  The client side Security Service MessageInterceptor accesses the ServiceContextList to determine decryption requirements as the response message is received at the client.

- at the ***12) client_nonmarshalled_response*** RequestInterceptor execution point

  The TaggedComponents are accessed to determine the server's secure name. The ServiceContextList is accessed to obtain the completed security session id, such that the client side session table entry can be created.

2.5.3 Transactions Service Use of Interceptors

A possible implementation of the Transactions Service can be mapped into the ORB's RequestInterceptor/MessageInterceptor design in the following manner:

1. **Transactions Service RequestInterceptors**

   The Transaction Service implements both a Client Side System RequestInterceptor and a Server Side System RequestInterceptor.

2. **Client Side RequestInterceptor Implementation**
   - At the Client Side, the Transactions Service wishes to intercept the Request in its non-marshalled state (before the request is marshalled into its wire format to be sent to the server).
   - When the response to the request is received from the server, Transactions wishes to intercept the response after it has been de-marshalled (into its non-marshalled state).
   - The Client Side RequestInterceptor implementation therefore inherits from the **clientNonMarshalledDataRI** interface which provides the ***client_nonmarshalled_request*** and ***client_nonmarshalled_response*** request interceptor operations. The Transactions Services client side RequestInterceptor also inherits from the **clientExceptionRI** interface which provides the ***client_system_exception*** request interceptor operation.

3. **Server Side Request Interceptor Implementation**
   - Upon receipt of the request from the client before it is de-marshalled from its wire, or marshalled format, Transactions Services intercepts the request. The Server Side RequestInterceptor implementation therefore inherits from the **serverMarshalledDataRI** interface, such that a Transactions implementation of the ***server_marshalled_request*** operation can be created.
   - The request is also intercepted by Transactions Services prior to marshalling the response to send back to the client. The Server Side RequestInterceptor implementation therefor inherits from the **serverNonMarshalledDataRI** interface, such that a Transactions Service implementation of the ***server_nonmarshalled_response*** operation can be created.
   - The Server Side Transactions Service RequestInterceptor also inherits from the **serverExceptionRI** interface, which provides the ***server_system_exception*** operation.

4. **Registering the Security Service RequestInterceptors**

   Transactions Services registers both the client side RequestInterceptor and the server side RequestInterceptor implementations via the ***register_RequestInterceptor*** operation.

The Transactions Service has now provided client and server side RequestInterceptor implementations. The following diagram illustrates the Transactions Services flow for the round trip of the request/reply through the ORB (the bold entries in the flow are the Interceptor execution points at which Transactions Services gets control, the others are not implemented by Transactions Services):

```
Client Side                               Server Side

(1) client_nonmarshalled_request (Sys RI)
      . . . request is marshalled . . .
(2) client_marshalled_request   (Sys RI)

(3) send_request_message   (Msg Int)
---------------------------------------------------------------------->

                                          4) receive_request_message (Msg Int)

                                          (5) server_marshalled_request (System RI)
                                                . . . request is demarshalled . . .
                                          (6) server_nonmarshalled_request (Sys Int)
                                                . . . request processing . . .
                                          (7) server_nonmarshalled_response (Sys RI)
                                                . . . response is marshalled . . .
                                          (8) server_marshalled_response (Sys RI)

                                          server_system_exception (Sys RI)
                                          server_user_exception (Sys RI)

                                          (9) send_response_message (Msg Int)
<----------------------------------------------------------------------
```

Client Side	**Server Side**

10) receive_response_message (Msg Int)

11) client_marshalled_response (Sys RI)

. . . response is demarshalled . . .

12) client_nonmarshalled_response*(Sys RI)*

client_system_exception

client_user_exception

The following is a synopsis of the Transaction Services processing performed at the bold execution points in the previous diagram:

- at the ***1) client_nonmarshalled_request*** RequestInterceptor execution point:

  This is called in a client process prior to marshalling an outbound request. The job the Transactions Service has to do here is to determine whether or not there is a transaction associated with the current thread and, if so, build a transaction service context to represent it and add it to the ServiceContextList associated with the request.

  As a performance optimization, Transactions Services may flow a transaction service context for a 'deferred' transaction, the desire being to avoid creating a transaction on a server different from that on which the business object is deployed. Part of the processing that occurs here involves determining, in the case where the local transaction is 'deferred', whether the remote system's Transaction Service supports the 'deferred transaction' semantic. This is done by examining the TaggedComponents. If the 'deferred transaction' semantic is not supported, the interceptor creates (i.e. un-defers) the transaction at that point., builds the service context for that transaction and adds it to the ServiceContextList of the request in the normal way.

  In order to support its 'deferred transaction' behavior, the Transaction Service client interceptors have to recognize whether or not they are nested. The client interceptor must not flow a transaction service context on a request that is nested (either as a result of a nested request being generated during processing of the *client_nonmarshalled_request* or resultant *client_nonmarshalled_response*). The Transactions Service request interceptor therefore maintains its own nesting count.

- at the **5*) server_marshalled_request*** RequestInterceptor execution point:

  This is called in server processes on all inbound requests prior to de-marshalling the request. The job the Transactions Service has to do is to examine the ServiceContextList associated with the request to determine whether a transaction service context is included. It first checks to see whether the target object is transactional by invoking the '_is_a' method on the target object to determine if it implements the CosTransactions::TransactionObject interface, thus introducing a nested call in the interceptor. If the target object is transactional, the interceptor looks to see whether a transactions service context is included in the ServiceContextList. If a transactions service context is not found, the interceptor has no more work. When the interceptor determines that there is a transactions service context, it imports the transaction represented by the service context and associates the transaction with the current thread.

- at the ***7) server_nonmarshalled_response*** RequestInterceptor execution point

  This is called in server processes on all outbound responses prior to marshalling. The job the transaction service has to do here is to determine whether or not there is a transaction associated with the current thread and, if so, build a transaction service context to represent it and add it to the ServiceContextList associated with the response. Having sent the response, the transaction associated with the current thread is suspended.

- at the ***12) client_nonmarshalled_response*** RequestInterceptor execution point

  This is called in a client process after de-marshalling an inbound response. The job the transaction service has to do here is to examine the ServiceContextList associated with the inbound response to determine whether a transaction service context is included. If there is a transactions context, then the interceptor builds a CosTransactions::PropagationContext from the service context and checks that this represents the same transaction that was on the thread prior to sending the request. In the case where an outbound service context represented a ‘deferred transaction’, the inbound response context replaces the ‘deferred transaction’ context. As part of the processing of the transaction service context, a number of remote _is_a methods may be called to validate the contents of the inbound context.

- at the ***client_system_exception*** RequestInterceptor execution point

  This is called in a client process when any CORBA::SystemException is received by the client. The Transactions Service interceptor determines whether it has already been driven for this response and if not calls *client_nonmarshalled_response*. Transactions services must ensure that *client_nonmarshalled_response* is driven once and once only (per *client_nonmarshalled_request*) in order to keep the nesting count accurate that is maintained by the interceptor.

- at the ***server_system_exception*** RequestInterceptor execution point

  This is called in the server process when any CORBA::SystemException is to be thrown back to the caller. The Transactions Service interceptor checks to see whether a transaction service context is already included in the response’s ServiceContextList, and if not calls *server_nonmarshalled_response*.

A

A.1 Appendix A - Design Considerations

A.1.1 Management of the Request Interceptor Lists by the ORB

The ORB must maintain lists of the Request Interceptors that have been registered for a particular execution point. The ORB implementation must assure that the registration and un-registration of Request Interceptors does not affect the successful operation of Request Interceptors that have already been registered with the ORB.

A.1.2 The Number of System Level Request Interceptor Execution Points

There are six operations defined for each of the client and server side RequestInterceptors. For each of the client and server:

1. Two operations involve processing of the request and response while in its non-marshalled state.
2. Two operations involve processing of the request and response while in its marshalled state.
3. Two operations involve processing of the request and response when a SystemException or UserException occur.

It is very possible that the number of monitoring points could be reduced. Any of the Request related information available in 1) above is also available in 2) above. Regardless of the decision on this matter, the architecture provided in this document eliminates any performance penalty if more execution points are architected than needed. This is due to the fact that only those RequestInterceptors which have a real implementation for a given execution point are registered to get control at that point.

A.1.3 The Processing Order for a Request Interceptor List

The System Level RequestInterceptor architecture developed in this document has provisions for a given RequestInterceptor registration to demand that the ORB make it the first in the processing list across the various execution points for which it is registered. This consideration was made to accommodate current requirements imposed by Security Services.

While the “make Security first” capability is sufficient at this time, there may be a need in the future to provide more capability to control the order at registration time of RequestInterceptor processing. A RequestInterceptor may find the need to be before or after another RequestInterceptor in the list. To achieve this, it is likely that RequestInterceptor Ids would need to be assigned for each of the services such that other RequestInterceptors could be referenced during the registration process.

3

3.1 Proposed Compliance Points

Two compliance points are suggested:

- The System Interceptor architecture in conjunction with the Message Interceptor architecture defined in chapter 2.
- The User Interceptor architecture defined in chapter 2.

3.2 Complete IDL Definitions

```
//---------------------------
// RequestHolder_get interface
//---------------------------
interface RequestHolder_get {
   typedef sequence <octet> request_message;
   typedef ReferenceData    ObjectKey;

   request_message          get_requestMessage ();
   string                   get_operation ();
   _Context                 get_context ();
   NVList                   get_arguments ();
   NamedValue               get_result ();
   ExceptionList            get_exceptions ();
   ContextList              get_contextlist ();
   boolean                  get_oneWay ();
   unsigned long            get_requestId ();
   ObjectKey                get_objectKey ();
   Principal                get_principal ();
   IOP::IOR                 get_ior ();
   IOP::ServiceContextList get_serviceContextList ();
   Object                   get_proxy  ();
   Object                   get_target ();
   boolean                  get_forceRetry ();
};


//---------------------------
// RequestHolder_set interface
//---------------------------
interface RequestHolder_set {
   typedef sequence <octet> request_message;
   typedef ReferenceData    ObjectKey;

   void              set_requestId (in unsigned long ri);
   void              set_objectKey (in ObjectKey objkey);
   void              set_principal (in Principal p);
   void              set_ior (in IOP::IOR ior);
   void              set_serviceContextList
                     (in IOP::ServiceContextList scl);
   void              set_forceRetry
                     (in boolean forceRetryValue);
};


//-----------------------
// RequestHolder interface
//-----------------------
interface RequestHolder : RequestHolder_get, RequestHolder_set {
   unsigned long     get_ServiceDataKey ();
   void              set_ServiceData
                           (in unsigned long service_data_key,
                           in any          service_data);
   any               get_ServiceData
                           (in unsigned long service_data_key);
};


//---------------------------
// UserRequestHolder interface
//---------------------------
interface UserRequestHolder : RequestHolder_get {

};


//----------------------------
// RequestInterceptor interface
//----------------------------
interface RequestInterceptor {

};


//-----------------------------------
// clientNonMarshalledDataRI interface
//-----------------------------------
interface clientNonMarshalledDataRi : RequestInterceptor {
   void client_nonmarshalled_request  (in RequestHolder rh);

   void client_nonmarshalled_response (in RequestHolder rh);
};


//--------------------------------
// clientMarshalledDataRI interface
//--------------------------------
interface clientMarshalledDataRi : RequestInterceptor {
   void client_marshalled_request  (in RequestHolder rh);

   void client_marshalled_response (in RequestHolder rh);
};


//--------------------
// ClientFlow interface
//--------------------
interface ClientFlow {
   enum client_completion_status {YES, NO, FAILED};
   short client_non_marshalled_requestCalled ();
   short client_marshalled_requestCalled ();
   short client_marshalled_responseCalled ();
   short client_non_marshalled_responseCalled ();
};


//---------------------------
// clientExceptionRI interface
//---------------------------
interface clientExceptionRi : RequestInterceptor {
   void client_system_exception (in RequestHolder rh , in ClientFlow cf);
   void client_user_exception   (in RequestHolder rh , in ClientFlow cf);
};


//--------------------------------
// serverMarshalledDataRI interface
//--------------------------------
interface serverMarshalledDataRi : RequestInterceptor {
   void server_marshalled_request  (in RequestHolder rh);
   void server_marshalled_response (in RequestHolder rh);
};


//-----------------------------------
// serverNonMarshalledDataRI interface
//-----------------------------------
interface serverNonMarshalledDataRI : RequestInterceptor {
   void server_nonmarshalled_request  (in RequestHolder rh);
   void server_nonmarshalled_response (in RequestHolder rh);
};


//--------------------
// ServerFlow interface
//--------------------
interface ServerFlow {
   enum server_completion_status  {YES, NO, FAILED};
   short server_marshalled_requestCalled ();
   short server_non_marshalled_requestCalled ();
   short server_non_marshalled_responseCalled ();
   short server_marshalled_responseCalled ();
};


//---------------------------
// serverExceptionRI interface
//---------------------------
interface serverExceptionRi : RequestInterceptor {
   void server_system_exception (in RequestHolder rh , in ServerFlow sf);
   void server_user_exception   (in RequestHolder rh , in ServerFlow sf);
```

```
};

//interface UserRequestHolder:RequestHolder_get {
//};

//------------------------------
// clientSideRequestURI interface
//------------------------------
interface clientSideRequestURI : RequestInterceptor {
   void client_pre_request  (in RequestHolder rh);
   void client_post_request (in RequestHolder rh);
};

//-------------------------------
// clientSideResponseURI interface
//-------------------------------
interface clientSideResponseURI : RequestInterceptor {
   void client_pre_response  (in RequestHolder rh);
   void client_post_response (in RequestHolder rh);
};

//------------------------------
// serverSideRequestURI interface
//------------------------------
interface serverSideRequestURI : RequestInterceptor {
   void server_pre_request  (in RequestHolder rh);
   void server_post_request (in RequestHolder rh);
};

//-------------------------------
// serverSideResponseURI interface
//-------------------------------
interface serverSideResponseURI : RequestInterceptor {
   void server_pre_response  (in RequestHolder rh);
   void server_post_response (in RequestHolder rh);
};

//-----------------------
// MessageHolder interface
//-----------------------
interface MessageHolder {
   typedef sequence <octet> message_buf;
   attribute message_buf    message;
};
```

```
//-----------------------------
// MessageInterceptor interface
//-----------------------------
interface MessageInterceptor {
   void send_request_message     (inout MessageHolder message,
                                  in RequestHolder    request);
   void receive_response_message (inout MessageHolder message);
   void receive_request_message  (inout MessageHolder message);
   void send_response_message    (inout MessageHolder message,
                                  in    RequestHolder request);
};


//------------------------------
// InterceptorRegistry interface
//------------------------------
interface InterceptorRegistry {
   void register_RequestInterceptor
          (in RequestInterceptor requestInterceptor,
           in boolean            makeMeFirst);

   void unregister_RequestInterceptor
          (in RequestInterceptor requestInterceptor);

   void register_MessageInterceptor
          (in MessageInterceptor messageInterceptor);

   void unregister_MessageInterceptor
          (in MessageInterceptor messageInterceptor);
};
```