Portable Interceptors

Eternal Systems, Inc.

Expersoft Corporation

Sun Microsystems, Inc.

April 26, 1999

Version 1.0 - Initial RFP Submission

OMG Document Number orbos/99-04-07

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.

Contacts

Louise Moser
Eternal Systems, Inc.
P.O. Box 13963
Santa Barbara, CA 93107
USA
phone +1 805 893 4897
fax +1 805 893 3262
E-mail: moser@ece.ucsb.edu

Shahzad Aslam-Mir
Expersoft Corporation
5825 Oberlin Drive
San Diego. CA 92121
USA
phone +1 619 824 4128
fax +1 619 824 4110
E-mail: sam@expersoft.com

Harold Carr
Sun Microsystems
901 San Antonio Road
MS CUP02-201
Palo Alto, CA 94303-4900
USA
phone +1 408 517 6783
fax +1 408 863 3195
E-mail: harold.carr@sun.com

Last Modified : 1999 Apr 26 (Mon) 11:24:28 by Harold Carr.

Contents

- Introduction
    - Typical Interceptor Usage
    - All objects locality constrained
- Interception Points
    - Interceptor Base Class
    - Invocation
        - Client Side
        - Server Side
        - Invocation Cookies
            - System Exception Interception Point
    - IOR Creation, marshaling and unmarshaling.
    - Connection management

    - Thread management
    - ORB Lifecycle
- Registering and chaining interceptors
- Converting between marshal streams and requests
- Abstract representations
    - IOR Representation
    - Service Context Representation
    - Transport Representations
- Interceptor Execution
    - Concurrency and Interceptors
- General open issues and notes.
- Examples.

<!----><!----><!---->

Introduction

Eternal Systems, Expersoft Corporation and Sun Microsystems, are pleased to provide this first submission in response to the OMG "Portable Interceptors" RFP, orbos/98-09-11.

This proposal focuses on identifying *interception points*, what information is given to each point, what effects each points may cause, and possible uses for each point.

A major goal of this proposal is to be independent of encoding and transport.

This proposal changes the focus from "request" and "message" level interceptors to "interception points." It enables Request objects to be constructed, but does not construct them in advance, since, in many cases, they are not necessary.

This proposal does not address "system" versus "user" level interceptors.

<!----><!---->

Typical Interceptor Usage

Interceptors provides hooks to programmers to execute their piece of code at certain points during ORB operation. Typical uses include:

- Transaction service integration
- Security
- Message compression and encryption
- Fault tolerance
- Tracing, profiling, debugging, logging

<!----><!---->

All Objects Locality Constrained

All types specified in this proposal are locality constrained. We do not specify how this is accomplished at this time. We anticipate either having the final specification say "locality constrained" (as does the POA), or perhaps use value types, or use some sort of "local" interface (as in orbos/99-03-06).

<!----><!----><!---->

Interception Points

We will refer to the ORB execution place at which a programmer interpose custom code as an "interception point."

An interceptor is an object that has several methods which are called at these different interception points by the ORB.

For a given class of interceptor, the application can create 0 or more interceptor objects.

<!----><!---->

Interceptor Base Class

There is a base interceptor class from which all interceptors derive:

```
module PortableInterceptors_1_0 {
       interface Interceptor {
              String getInterceptorName();
              int    getInterceptorType();

       };
};
```

This class is generalized by various interceptor classes: client invocation, server invocation, IOR creation, etc.

<!----><!---->

Invocation Interception Points

This section shows interception points on the path of an invocation request sent from a client to a server and, in the case of a synchronous invocation, a reply returned from a server to a client.

Each invocation interception point is given an:

- invocation cookie

which allows state to be passed between all client-side interceptors or all server-side interceptors in the context of a specific invocation. These cookies do not pass between the client and server side.

(The points are actually passed a "cookie jar" containing cookies. This enables multiple, independent developers to install interceptors without interfering with each others cookies.)

All invocation interceptors derive from:

```
module PortableInterceptors_1_0 {
       interface InvocationInterceptor : Interceptor {
              ...
       };
};
```

<!---->

Client Side Invocation Interception Points

```
module PortableInterceptors_1_0 {
     interface ClientInvocationInterceptor : InvocationInterceptor {
             void send_request_begin
                     (
                             inout InvocationInterceptorCookieJar cookies
                     );
             AIOR send_request_ior
                     (
                             in    Object                      obj
                             inout InvocationInterceptorCookieJar cookies
                     );
             void send_request_before_marhsal
                     (
                             in    unsigned long      request_id,
                             in    boolean            response_expected,
                             inout ServiceContextList service_contexts,
                             inout sequence<octet>    object_key,
                             inout string             operation,
                             inout InvocationInterceptorCookieJar cookies
                     );
             MarshalOutputStream send_request_after_marshal
                     (
                             in    MarshalOutputStream output_stream
                             inout InvocationInterceptorCookieJar cookies
                     );
             MarshalOutputStream send_request_transform
                     (
                             in    MarshalOutputStream output_stream
                             inout InvocationInterceptorCookieJar cookies
                     );
             void send_request_end
                     (
                             inout InvocationInterceptorCookieJar cookies
                     );
             void receive_reply_begin
                     (
                             inout InvocationInterceptorCookieJar cookies
                     );
             MarshalInputStream receive_reply_transform
                     (
                             in    ServiceContextList service_contexts,
                             in    unsigned long      request_id,
                             in    ReplyStatusType    reply_status,
                             in    MarshalInputStream input_stream
                             inout InvocationInterceptorCookieJar cookies
                     );
             MarshalInputStream receive_reply_before_unmarshal
                     (
                             in    MarshalInputStream input_stream
                             inout InvocationInterceptorCookieJar cookies
                     );
             void receive_reply_after_unmarshal
                     (
                             inout InvocationInterceptorCookieJar cookies
                     );
             void receive_reply_end
```

```
                     (
                             inout InvocationInterceptorCookieJar cookies
                     );
     };

     interface ClientLocateRequestInterceptor : InvocationInterceptor {
             void send_locate_request
                     (
                     ...
                             inout InvocationInterceptorCookieJar cookies
                     );
             void receive_locate_reply_object_here
                     (
                     ...
                             inout InvocationInterceptorCookieJar cookies
                     );
             void receive_locate_reply_object_forward
                     (
                     ...
                             inout InvocationInterceptorCookieJar cookies
                     );
             void receive_locate_reply_unknown_object
                     (
                     ...
                             inout InvocationInterceptorCookieJar cookies
                     );
     };

     interface ClientConnectionInterceptor : InvocationInterceptor {
             Connection send_request_get_connection
                     (
                             ...
                             inout InvocationInterceptorCookieJar cookies
                     );
     };
};
```

ClientInvocationInterceptor Methods

`send_request_begin`

- Point
  - When an invocation is made on a object reference this point is called before the ORB does *any* processing whatsoever.
  - Before `send_request_ior`.
- Context
  - Within the context of the client invoking thread
- Functionality
  - Create and install a cookie for this invocation.
- Exceptions
  - If this method throws an exception then that exception is given to the client code as the result of the invocation.
  - Further, any other `ClientInvocationInterceptors` chained after this one do not get invoked (i.e., any further client side `send_request_*` and `receive_reply_*` points).
- Uses
  - Set a timer.
- Issues

`send_request_ior`

- Point
    - After `send_request_begin`
    - Before the IIOP request header marshaled.
    - Before arguments are marshaled into request stream.
    - Before `send_request_before_marshal`.
- Context
    - Within the context of the client invoking thread
- Functionality
    - The interceptor is given the the object reference upon which this invocation is being made.
    - It may convert the reference to an IOR Representation (AIOR).
    - It may read/write the converted representation.
    - If it returns NULL then the given object reference is used as the target for the request (modulo other `send_request_ior` interceptors).
    - If it returns an AIOR then the contents of that AIOR is used as the target for the request (modulo other `send_request_ior` interceptors).
    - This is called on *every* invocation, giving the opportunity to redirect every request.
- Exceptions
    - See `send_request_begin`
- Uses
    - Load balancing
- Issues
    - Connection closing/opening on redirect of existing connections.
    - In the case of a client invokes with ref1 but receives ref2 from a LOCATE_FORWARD then which ref will appear in subsequent calls of this point on the initial ref? Most ORBs do not pass the forwarded refs to the client. Instead, the LOCATE_FORWARD causes the client ORB to reconnect to the new ref (transparently to the client).
- Notes
    - This is given an object reference rather than an AIOR to avoid creating an AIOR unnecessarily.

`send_request_before_marshal`

- Point
    - After `send_request_ior`.
    - After the ORB has created ORB-specific service contexts.
    - Before the IIOP request header marshaled.
    - Before arguments are marshaled into request stream.
- Context
    - Within the context of the client invoking thread
- Functionality
    - Read the request id.

    - Read one way status.
    - Read/write the service context list.
    - Read/write the object key.
    - Read/write the operation name.
- Exceptions
    - See `send_request_begin`
- Uses
    - Change/compress the object key.
    - Operation name-grained client-side ACL.
    - Insert transaction service context.
    - Insert code set service context.
    - Insert Java code base service context.
    - Insert security info in service context.
    - Insert realtime info in service context.
    - Monitoring the application, for debugging or performance reasons
- Issues
    - If service contexts added but ORB receives LocateForward then the ORB must add those service contexts to forward IOR (or use the LocateReply interception point).
- Notes
    - If the operation name is modified then the modified name is marshaled but the original operation's arguments are still marshaled.

`send_request_after_marshal`

- Point
    - After `send_request_before_marshal`
    - Request header has been marshaled.
    - All arguments have been marshaled.
    - Before `send_request_transform`
- Context
    - Within the context of the client invoking thread
- Functionality
    - Read/modify the marshaled data output stream.
        - Using the stream to request, converter, the point may read/write the Request object then convert that object back to a stream.
    - If it returns NULL then the given stream is passed to the next transform interceptor point or used for the invocation.
    - Otherwise the returned stream is passed to the next transform interceptor point or used for the invocation.
- Exceptions
    - See `send_request_begin`
- Uses

    - "Request" level interceptor via stream to request converters.
    - Add "out-of-band" data to end of marshaled data.
- Issues
    - It would be possible to have a "post request header, pre arguments marshaled" point to allow data to be inserted before the marshal output stream.
- Notes
    - Only the output stream is given. Any additional unmarshaled information (e.g., ServiceContextList) from earlier interceptors can be passed, when needed, in a cookie.

`send_request_transform`

- Point
    - After `send_request_after_marshal`
- Context
    - Within the context of the client invoking thread
- Functionality
    - Read/write the marshal data output stream
    - If it returns NULL then the given stream is passed to the next transform interceptor point or used for the invocation.
    - Otherwise the returned stream is passed to the next transform interceptor point or used for the invocation.
- Exceptions
    - See `send_request_begin`
- Uses
    - Transform (e.g., encryption or compression) the entire message.
- Issues
- Notes
    - This point is similar to but separate from `send_request_after_marshal`, which is used to *modify the contents* of the marshal stream.

      `send_request_transform` is to be used to *transform the entire stream* without changing its contents.
    - This point is an "ease-of-use protocol" API to enable independent teams to use interceptors for stream modification and transformation without interfering with each other.
    - Transform interceptors are applied only to the marshal data stream. They are not applied to the message header because it contains information that is required by the ORB for dispatching the message to the appropriate object representation and POA, etc.

`send_request_end`

- Point
    - The final point before the request is actually sent.
    - The ORB does nothing after this point except send the request and block.
- Context
    - Within the context of the client invoking thread

- Functionality
    - Any necessary info passed in cookie.
- Exceptions
    - See `send_request_begin`
- Uses
    - Set timers to cancel requests if reply does not arrive in a certain time.
    - Stop a timer set in `send_request_begin` to measure client-side processing time of outgoing request.
- Issues

`receive_reply_begin`

- Point
    - When client ORB receives a Reply message.
    - The first invocation point after a reply is received before the ORB does any processing.
    - Before anything is unmarshaled.
    - Before `receive_reply_transform`.
- Context
    - Within the context of the client invoking thread
- Functionality
    - Necessary data in a cookie.
- Exceptions
    - See `send_request_begin`
- Uses
    - Set timers.
    - Using time stamps in cookie, if the response took too long, a programmer could raise an exception to be given to client code (avoiding further reply processing).
- Issues

`receive_reply_transform`

- Point
    - After `receive_reply_begin`
    - After Reply header unmarshaled.
    - Before unmarshaling of reply input stream.
    - Before `receive_reply_before_unmarshal`
- Context
    - Within the context of the client invoking thread
- Functionality
    - Read service context, request id, reply status.
    - Read/write marshal data input stream (using converters).
    - If it returns NULL the given input stream is used.
    - If it returns a stream then that stream is used instead.

- Exceptions
    - See `send_request_begin`
- Uses
- Issues
- Notes
    - If further points need any of the input arguments (other than the stream) they should be passed in a cookie.

`receive_reply_before_unmarshal`

- Point
    - After `receive_reply_transform`
    - Before arguments unmarshaled.
    - Before `receive_reply_after_unmarshal`
- Context
    - Within the context of the client invoking thread
- Functionality
    - Read/write marshaled data input stream.
- Exceptions
    - See `send_request_begin`
- Uses
    - Remove out-of-band data.
- Issues

`receive_reply_after_unmarshal`

- Point
    - After `receive_reply_before_unmarshal`
    - After reply arguments have been unmarshaled.
    - Before `receive_reply_end`
- Context
    - Within the context of the client invoking thread
- Functionality
- Exceptions
    - See `send_request_begin`
- Uses
    - Returning additional or more specific exceptions for a reply
- Issues

`receive_reply_end`

- Point
    - After `receive_reply_after_unmarshal`
    - After this point the ORB does *nothing* except invoke the servant.
- Context
    - Within the context of the client invoking thread

- Functionality
- Exceptions
    - See `send_request_begin`
- Uses
    - Measure client-side processing time of incoming reply.
- Issues

`ClientLocateRequestInterceptor` Methods

The following interception points, modeled on the GIOP LocateRequest message cycle, may not be invoked, depending on the protocol in effect.

`send_locate_request`

- Point
    - Client ORB sends a LocateRequest IIOP message.
- Context
- Functionality
- Exceptions
- Uses
- Issues

`receive_locate_reply_object_here`

- Point
- Context
- Functionality
- Exceptions
- Uses
- Issues

`receive_locate_reply_object_forward`

- Point
    - When client ORB receives a LocateReply message with a new forward IOR.
- Context
- Functionality
    - Read/write the forwarded IOR.
- Exceptions
- Uses
    - If a interceptor adds service context info to a request, but that request gets an OBJECT_FORWARD LocateReply message, then the redirected request will needs its service context list updated accordingly.
- Issues
- Notes
    - This is the important part of the LocateRequest cycle to catch on the client side so it can update Service Context information added to the original request.

receive_locate_reply_unknown_object

- Point
    - When the client ORB receives a LocateReply UNKNOWN_OBJECT message.
- Context
- Functionality
- Exceptions
- Uses
- Issues

ClientConnectionInterceptor Methods

send_request_get_connection

- Point
    - A connection needs to be obtained for the invocation.
    - Called on *every* invocation.
- Context
    - Within the context of the client invoking thread
- Functionality
    - Given information necessary to make connection.
    - Should return NULL (meaning ORB does it default action) or a Connection to be used on this invocation.
- Exceptions
- Uses
    - Alternate mechanism for obtaining a connection (e.g., from a cache, use an object locate facility, etc.).
- Issues
    - Need to define where this point occurs in relation to other client side points.

<!---->

Server Side Invocation Interception Points

```
module PortableInterceptors_1_0 {
     interface ServerInvocationInterceptor : InvocationInterceptor {
             void receive_request_begin
                     (
                             ...
                             in    Connection connection,
                             inout InvocationInterceptorCookieJar cookies
                     )
             MarshalInputStream receive_request_transform
                     (
                             in    unsigned long      request_id,
                             in    boolean            response_expected,
                             inout ServiceContextList service_contexts,
                             inout sequence<octet>    object_key,
                             inout string             operation,
                             in    MarshalInputStream input_stream,
                             inout InvocationInterceptorCookieJar cookies
                     );
```

```
                MarshalInputStream receive_request_before_unmarshal
                        (
                                in    MarshalInputStream input_stream
                                inout InvocationInterceptorCookieJar cookies
                        );
                void receive_request_after_unmarshal
                        (
                                inout InvocationInterceptorCookieJar cookies
                        );
                void receive_request_end
                        (
                                inout InvocationInterceptorCookieJar cookies
                        );
                void send_reply_begin
                        (
                                inout InvocationInterceptorCookieJar cookies
                        );
                void send_reply_before_marshal
                        (
                                inout ServiceContextList  service_contexts,
                                in    ReplyStatusType     reply_status,
                                inout InvocationInterceptorCookieJar cookies
                        );
                MarshalOutputStream send_reply_after_marshal
                        (
                                in    MarshalOutputStream output_stream
                                inout InvocationInterceptorCookieJar cookies
                        );
                MarshalOutputStream send_reply_transform
                        (
                                in    MarshalOutputStream output_stream
                                inout InvocationInterceptorCookieJar cookies
                        );
                void send_reply_end
                        (
                                inout InvocationInterceptorCookieJar cookies
                        );
        };

        interface ServerLocateRequestInterceptor : InvocationInterceptor {
                ... receive_locate_request
                        (
                                ...
                                inout InvocationInterceptorCookieJar cookies
                        );
                ... send_locate_reply_object_here
                        (
                                ...
                                inout InvocationInterceptorCookieJar cookies
                        );
                ... send_locate_reply_object_forward
                        (
                                ...
                                inout InvocationInterceptorCookieJar cookies
                        );
                ... send_locate_reply_unknown_object
                        (
                                ...
                                inout InvocationInterceptorCookieJar cookies
                        );
        };

        interface ServerConnectionInterceptor : InvocationInterceptor {
                void connection_accepted
                        (
                                ...
                                inout InvocationInterceptorCookieJar cookies
                        );
                void connection_limit
                        (
                                ...
                                inout InvocationInterceptorCookieJar cookies
                        );
    };
};
```

ServerInvocationInterceptor Methods

Issue: the thread context in which the server invocation points execute should probably be unspecified.

`receive_request_begin`

- Point
  - This first point immediately after ORB receives a request.
  - Before `receive_request_transform`
- Context
- Functionality
  - Given the connection on which the request arrived.
- Exceptions
  - If this method throws an exception then that exception is passed back to the client side where the client code will receive it as the result of the invocation.
  - Further, any other `ServerInvocationInterceptors` chained after this one do not get invoked (i.e., any further server side `receive_request_*` and `send_reply_* points`).
- Uses
  - Set timers.
- Issues

`receive_request_transform`

- Point
  - After `receive_request_begin`
  - After request header unmarshaled.
  - Before `receive_request_before_unmarshal`
- Context
- Functionality
  - Read the request id.
  - Read one way status.
  - Read/write service contexts.
  - Read/write object key.
  - Read the marshaled data input stream.
  - If it return NULL then the given stream is passed on to unmarshaling.
  - Other the returned stream is used.
- Exceptions
  - See `receive_request_begin`
- Uses
  - Decompression.

    - Decryption.
- Issues

`receive_request_before_unmarshal`

- Point
    - After `receive_request_tranform`
    - Before arguments unmarshaled.
    - Before `receive_request_after_unmarshal`
- Context
- Functionality
    - Read/write marshaled data input stream
    - Return NULL means use given.
    - Otherwise use returned stream.
- Exceptions
    - See `receive_request_begin`
- Uses
    - Retrieve/remove out of band data
    - Transform to `Request` via stream to request mechanism (and back) to get "request level" interceptor.
    - Operation name-grained server-side ACL.
- Issues

`receive_request_after_unmarshal`

- Point
    - After `receive_request_before_unmarshal`
    - After arguments unmarshaled
    - Before dispatching to servant (POA provides flexibility for finding a servant).
    - Before `receive_request_end`
- Context
- Functionality
- Exceptions
    - See `receive_request_begin`
- Uses
- Issues

`receive_request_end`

- Point
    - After `receive_request_after_unmarshal`
    - The ORB does *nothing* after this point except invoke the servant.
- Context
- Functionality
- Exceptions

    - See `receive_request_begin`
- Uses
    - Measure server-side processing time of incoming request.
- Issues

`send_reply_begin`

- Point
    - After servant execution complete
    - This point is called before the ORB does anything else after servant code.
    - Before `send_reply_before_marshal`
- Context
- Functionality
- Exceptions
    - See `receive_request_begin`
- Uses
    - Set timer.
- Issues

`send_reply_before_marshal`

- Point
    - After `send_reply_begin`
    - After ORB-specific service contexts created.
    - Before Reply header marshaled.
    - Before return value/exception marshaled
    - Before `send_reply_after_marshal`
- Context
- Functionality
    - Read/write service contexts
- Exceptions
    - See `receive_request_begin`
- Uses
    - Put information on unknown exceptions in service context.
- Issues
- Notes
    - Servant result may be response or exception

`send_reply_after_marshal`

- Point
    - After `send_reply_before_marshal`
    - After Reply header marshaled.
    - After return value/exception marshaled
    - Before `send_reply_transform`
- Context

- Functionality
    - Read/write marshaled data output stream.
    - If NULL returned use given stream.
    - Otherwise use returned stream.
- Exceptions
    - See `receive_request_begin`
- Uses
    - Add out-of-band data to marshal stream.
- Issues

`send_reply_transform`

- Point
    - After `send_reply_after_marshal`
    - Before `send_reply_end`
- Context
- Functionality
    - Read/write marshaled data output stream.
- Exceptions
    - See `receive_request_begin`
- Uses
    - Compression
    - Encryption
- Issues

`send_reply_end`

- Point
    - After `send_reply_transform`
    - The ORB does *nothing* after this point except send the reply
- Context
- Functionality
- Exceptions
    - See `receive_request_begin`
- Uses
    - Measure server-side processing time of outgoing responses.
- Issues

`ServerLocateRequestInterceptor` **Methods**

`receive_locate_request`

- Point
    - Server ORB receives a LocateRequest IIOP message.
- Context
- Functionality
    - May return IOR which results in ORB issuing OBJECT_FORWARD reply

    status with that IOR.

- Exceptions
- Uses
    - Useful for apps that perform naming service type operations.
- Issues

`send_locate_reply_object_here`

- Point
    - When a LocateRequest results in ORB determining that the target object resides in this server.
    - Executed just prior to ORB issuing OBJECT_HERE reply.
- Context
- Functionality
- Exceptions
- Uses
- Issues

`send_locate_reply_object_forward`

- Point
    - When a LocateRequest results in ORB replying with a new IOR.
    - Called whenever the ORB or any interceptor (even itself) has caused a locate request to be forwarded to a new IOR.
- Context
- Functionality
    - Given the forwarded IOR.
- Exceptions
- Uses
- Issues

`send_locate_reply_unknown_object`

- Point
    - When a LocateRequest results in ORB determining that target object is either invalid or does not exist in this server.
    - Called just before ORB issues UNKNOWN_OBJECT.
- Context
- Functionality
- Exceptions
- Uses
- Issues

`ServerConnectionInterceptor` Methods

`connection_accepted`

- Point
    - Server ORB has accepted a new connection.

- Context
- Functionality
- Exceptions
- Uses

  To refuse a client connection

  - Change size of TCP/IP buffers
  - Gather list of open socket file descriptors
- Issues

`connection_limit`

- Point
  - When connection limit reached, or client exists or dies, or connection times out.
- Context
- Functionality
- Exceptions
- Uses
- Issues

<!---->

Invocation Cookies

The notion of interceptor cookie is useful for an interceptor to maintain state on a specific request. It is defined only for invocation interception points.

An invocation interception point can create an interceptor cookie object which is specific to the invocation. If created on the client side, any subsequent client side interception point may access/modify/replace the cookie passed to it. Similarly for the server side. Interceptor cookies do not pass between the client and server sides.

To define an interceptor cookie, the application must define a class that derives from:

```
module PortableInterceptors_1_0 {

     exception DuplicateName {};

     exception NameNotFound {};

     interface InvocationInterceptorCookie {

             attribute string name;

             void add_self_to_jar(inout InvocationInterceptorCookieJar jar)
                     raises (DuplicateName);

             void system_exception_raised_before_sending
                     (
                             in    SystemException exception,
                             inout InvocationInterceptorCookieJar cookies
                     );
             void system_exception_raised_after_receiving
                     (
                             in    SystemException exception,
                             inout InvocationInterceptorCookieJar cookies
                     );
     };
     interface InvocationInterceptorCookieJar {
             void remove_cookie(in string name)
                     raises (NameNotFound);
     };
};
```

A cookie be created and added to a cookie jar at any invocation interception point. Initial points are given NULL cookie jars.

It is the programmer's responsibility to delete/destroy cookies. The ORB deletes the cookie jar (not its contents) after the last point.

System Exception Interception Point

Cookies are also the system exception interception point as detailed below in the description of cookie methods.

`InvocationInterceptorCookie` Methods

`add_self_to_jar`

- Functionality
  - New cookies add themselves to a jar via this method.
  - If given a NULL jar it creates a new empty jar.
- Exceptions
  - If the jar already contains a cookie of the same name an exception is thrown.
- Uses
- Issues

`system_exception_raised_before_sending`

- Point
  - Client: If a system exception occurs on the client side before actually sending the request.
  - Server: If a system exception occurs on the server side after the servant code completes, before the reply is actually sent.
- Context
  - Client: Within the context of the client invoking thread
  - Server: ?
- Functionality
  - Given the exception.
  - If it returns NULL then the given exception is "returned" as the result of the request.
  - If it raises an exception then that exception becomes the "result".
- Exceptions
  - There may be more than one cookie per request.
  - They are all called.
  - If one of them throws an exception then that exception will be given to the "next" cookie.
- Uses

    - This is a "system exception" point.
    - To change the exception.
    - To clean up cookie state.
- Issues
    - Programmer must have control of order of cookie similar to the control they have of interceptors themselves.
- Notes
    - This "point" has per-request granularity (which may be different from the granularity of other interceptors).

`system_exception_raised_after_receiving`

- Point
    - Client: If a system exception occurs on the client side after receiving the request, before returning to client code.
    - Server: If a system exception occurs on the server side after receiving the request, before executing servant code.
- Context
    - Client: Within the context of the client invoking thread
    - Server: ??
- Functionality
- Exceptions
- Uses
- Issues

<!----><!---->

IOR Creation, marshaling and unmarshaling

```
module PortableInterceptors_1_0 {
       interface IORInterceptor : Interceptor {
              AIOR create_ior
                     (
                             ...
                             in  Object obj
                     );
              AIOR marshal_ior
                     (
                             ...
                             in    Object obj,
                             inout InvocationInterceptorCookieJar cookies
                     );
              AIOR unmarshal_ior
                     (
                             ...
                             in    Object obj,
                             inout InvocationInterceptorCookieJar cookies
                     );
};
```

IORInterceptor Methods

`create_ior`

- Point
    - After the ORB creates an object reference.
- Context
- Functionality
    - Convert the IOR to an AIOR.
    - Read/write all parts of AIOR.
    - If NULL is returned then the given object reference is used for the result of IOR creation.
    - Otherwise the returned AIOR is converted by the ORB into an object reference and used for the result of IOR creation.
- Exceptions
- Uses
    - Monitor the generation of IORs.
    - When additional information must be associated with a newly created object reference. (e.g., implementing security requires associating information about security policies with the object reference).
    - Monitoring object references as they are generated and as they move around the system may be useful for security.
    - May be used to include a security context as a tagged component within an object reference.
    - Replace an IOR of an object with an IOR of an entirely different object, like ORB daemons.
    - Augment the object reference with additional profiles that provide alternative Internet addresses for the object (e.g., if the object is reachable by multiple different TCP/IP paths).
    - Some systems may require published/exported object references that "escape" from the system to be the references to a firewall or gateway, rather than the real object.
- Issues
    - The IOR passed to the interceptor must be independent of encoding and transport. To accomplish this we propose an abstract IOR representation.
    - In cases where the IOR is propagated through a GIOP LocateReply message, for instance, additional GIOP-specific information may be necessary.

`marshal_ior`

- Point
    - Before marshaling an object reference.
- Context
- Functionality
    - Convert the IOR to an AIOR.
    - Read/write all parts of AIOR.
    - If NULL is returned then the given object reference is used for the result of IOR creation.
    - Otherwise the returned AIOR is converted by the ORB into an object reference and used for the result of IOR creation.

- Exceptions
- Uses
  - Firewall support
  - GIOP message compression schemes (e.g. put a dictionary in the service context).
- Issues

`unmarshal_ior`

- Point
  - After unmarshaling an object reference.
  - Before passing the resulting object reference to its "client."
- Context
- Functionality
  - Convert the IOR to an AIOR.
  - Read/write all parts of AIOR.
  - If NULL is returned then the given object reference is used for the result of IOR creation.
  - Otherwise the returned AIOR is converted by the ORB into an object reference and used for the result of IOR creation.
- Exceptions
- Uses
  - Useful for IOR decompression.
  - Needed for symmetry with marshaling.
- Issues
  - Is this point executed when an ORB sends a LocateReply containing an IOR?

<!----><!---->

Connection Management

Limited connection management is provided by:

`ClientConnectionInterceptor`

`ServerConnectionInterceptor`

A more extensive model is most likely necessary.

<!----><!---->

Thread Management

...

<!----><!---->

ORB Lifecycle

...

- `orb_startup`
    - Point
    - Context
    - Functionality
    - Exceptions
    - Uses
    - Issues
- ...

  `orb_shutdown`
    - Point
    - Context
    - Functionality
    - Exceptions
    - Uses
    - Issues

...

<!----><!----><!---->

Registering and chaining interceptors

Interceptors may be dynamically (un)registered at any time. The programmer is given complete control over order of invocation for interceptors of the same type.

<!----><!---->

(Un)registering Interceptors

A programmer registers interceptors on both the client and the server side. (Un)registration methods are provided on an object obtained from `ORB.list_initial_services("InterceptorRegistry")`

Interceptors can be (un)registered at any time. There is no protection against interceptor (un)registration while an invocation is in progress. The programmer must ensure that no invocation is in progress.

It is possible to (un)register interceptors concurrently by several threads. The programmer is responsible for synchronization issues.

At this time we do not have a recommendation on the granularity of interceptors. Possibilities are:

- per-process
- per-connection
- per-POA
- per-object

Regardless, once a granularity has been decided, (un)registration will be provided by an object obtained via `list_initial_services`. It may be necessary to define and register "interceptor factories," to create interceptors when certain events occur, such as connection acceptance.

<!----><!---->

Chaining Interceptors

When several interceptors are created, the interceptor methods at a specific interception point (i.e., same method name of the same kind of interceptor object) are called sequentially. Therefore, it is necessary to specify the order in which they will be invoked.

The programmer has control over the order of interceptor application when more than one interceptor of a given type is registered with the ORB. This is done via lists of registered interceptor objects.

The programmer gives each interceptor object a name (string). The scope of interceptor names is limited to each type interceptor. This enables a programmer to give the same name to a client interceptor, a server interceptor, etc. An exception is raised if the same name is used twice in the same type.

Interceptors names are used to find or unregister registered interceptors and to register new interceptors before or after existing interceptors by name.

Further, a programmer can obtain the list of registered interceptors of a given type and then iterate down this list, unregistering, or adding new interceptors before or after list members.

The list order of each type is the invocation order. Therefore programmers have complete control over order. It is the programmer's responsibility to deal with synchronization issues when (un)registering interceptors or traversing an interception list in a multithreaded environment.

<!----><!---->

Client and Server Agreement on Interceptors

Many uses of interceptors require that the client and server roles agree on matching interceptors. For example, if a client interceptor adds out-of-band data before the arguments are marshaled, then a server interceptor must be prepared to remove the data.

This proposal does not specify how client and servers agree on interceptors. However, we do provide the following discussion.

There are 3 cases:

- Client-only interceptor.
- Server-only interceptor.
- Client/Server interceptor.

The Client/Server case must consider:

- Interceptor implementations must have globally unique ids (solutions such as used in Java and in IDL type ids).

- Some mechanism is necessary for finding (and perhaps downloading) the locally required code (opens up security issues).
- The client and server roles must agree on the interceptors to be used:
    - Client tells the server.
        - An example here is a client determining what type of authentication to use.
    - Server tells client.
    - Client and Server negotiate.
    - Must handle dynamic interceptor (un)registration.
    - Would need to have finer-grained (un)registration for shared (i.e., client/server) interceptors and non-shared (e.g., client-side only) interceptors.

<!----><!----><!---->

Converting between marshal streams and requests

CORBA 2.2 defines request interceptors which represent requests as `CORBA::Request` objects. However:

- DII is too heavyweight
- Not possible in streams-based stubs and skeletons

Forcing the use of DII is not a good idea since some uses of interceptors call for modifying service context information while others call for simple "piggybacking" additional information on the request.

Rather than invent yet another `Request` type we specify an API for taking either a marshal input or output stream and converting it to a `Request` or taking a `Request` and converting it to a marshal input or output stream.

This way, those applications which need access to those parts of the request represented in `Request` can use the conversion routines while those that do not need this access do not have to pay the overhead of creating request-like objects.

...

Issues:

`CORBA::Request` versus `CORBA::ServerRequest`.

- Should frequently access Request slots be pass as arguments to interceptors to obviate the need to create Request objects to obtain this information?
- If information is duplicated should duplicate locations be kept in sync? By who?

<!----><!----><!---->

Abstract Representations

<!----><!----><!---->

IOR Representation

Manipulating the profile information in the IOR is essential for many ORB services such as security. However,

- The IDL for an IOR is too low level
- Non-GIOP encodings should be able to use interceptors

The original intent for IORs is to be an abstract information model for an object reference.

We propose an interface to the contents of an object reference that does not assume an encoding. This "abstract" IOR (AIOR) type allows modification, insertion and retrieval of specific parts of an IOR.

...

Service Contexts

GIOP provides a mechanism known as the Service Context that is used to associate extra information with an invocation. As in the IOR case, a more abstract representation is necessary to allow programmer's to easily access service contexts and to enable non-GIOP protocols.

...

<!----><!---->

Transport

`MarshalOutputStream` ...

`MarshalInputStream` ...

`Connection` ...

<!--><!----><!---->

Interceptor Execution

Although previous sections specify order of interception point execution, this section provides a focused detailed graphic representation of execution order.

The order of activation of all interception points on a successful synchronous invocation is: ...

One way: ...

LocateRequest: ...

Exception situations: ...

... etc.,

<!----><!---->

Concurrency and Interceptors

On multi-threaded platforms, interceptor implementations must be ready to be invoked by several threads at the same time.

...

<!----><!----><!---->

General Open Issues and Notes

- Do we want to provide interception points for all GIOP messages on both client and server sides?

  Uses:

  - If users want to encrypt their communication, they probably want this encryption for every message, not just for requests and replies.
  - It is necessary to support GIOP_Fragment, given that messages may be compressed or encrypted, with the compression or encryption algorithm applied to the whole message, rather than to individual fragments. In this case, all of the fragments that constitute a message must necessarily be intercepted and reassembled in order for the message to make sense.
- Specify collocated request semantics (i.e., invocations and responses that occur between collocated objects, that do not necessarily go out over the network through IIOP).
- Interceptor issues with replaceability
- How do we avoid continuously extending the number of interceptor points or can we add an extension framework?
- How does this relate to other OMG specifications?
  - security
  - transactions
  - messaging
    - has stream request custom marshaling or particular types
  - firewall
  - bi-directional GIOP
- How can we utilize Java unique strengths (downloadable code) while still supporting a language independent standard?
  - Perhaps this is just done using value types.
- Is the academic work in open implementations (PARC) or MOPs. useful here?
- This proposal views interceptors as a pipes and filters design pattern. It does not attempt to support:
  - Pipe redirection
    - It seems interceptors are best served by a two-way pipes/filters design pattern. Redirecting a two-way pipe requires knowledge of expected return

results.

Questions such as:

- Can after-marshal/before-unmarshal interceptors be used to route messages without ever reaching the ORB?
- Can they act in a forwarding or echoing role with regard to received messages?

are asked with respect to interceptors. At this time we think the general answer to both is no as explained next.

When making a request, the invocation thread in the ORB passes through various layers, finally transmitting the data output stream then blocking waiting for the reply. When the reply comes it packages the data input stream into some ORB specific data structure and passes that back up the thread as the reply.

To redirect the pipe (e.g., plug in a new protocol) the invocation sequence would need to define these outgoing and incoming structures so the new direction (e.g., alternate protocol) could package the reply appropriately.

It seems redirection (e.g., pluggable protocols) would best be served by a separate RFP.

Replaying has the same considerations.

- Restarts (e.g., Returning magic values or exceptions which cause the invocation to start again, while perhaps simultaneously continuing the current invocation - useful for "repeaters"). The COOL ORB had limited support for restarts.

- Is it possible to have an interceptor that generates additional messages, as the security interceptors do for their authentication handshakes ? Could these messages themselves be intercepted by other interceptors, such as a logging or a traffic analysis interceptor.
- Can interceptors be coupled, ie, can an inbound interceptor and an outbound interceptor "co-operate" to achieve a certain effect? Or are inbound and outbound interceptors essentially decoupled? An example of where this would be useful is a security interceptor performing an authentication handshake.
- Can a message-level interceptor be coupled with an IOR interceptor? If the IOR interceptor had replaced the object key earlier, but "remembered" the original object key that the object issued, then, the message-level interceptor needs to replace the replaced object key with the original object key.
- Will this support heartbeat messages exchanged between a client ORB and a server ORB (without the client's or the server's knowledge)? This can be used to ascertain if the connection is "alive". May be necessary for those applications that are dissatisfied with the variability of the TCP timeout when waiting for a server to complete a request, or those that want to make the TCP timeout context-dependent.
- Are "per-thread" interceptors useful?
- Are various threading models compatible with the interceptor model (e.g., can the interceptor model deal with the thread-per-request concurrency model, for instance, where a thread should only be able to access the interceptor related to its request alone).
- It may be problematic to allow `ServerConnectionInterceptor` methods to be invocable by multiple threads at the same time since a connection may be waiting to time out. Reentrant code may be necessary.

<!----><!----><!---->

Examples

...

<!----><!---->

Transactions

...

<!----><!---->

Security

<!---->

Authentication

Additional outbound messages might be generated by a security interceptor to authenticate the far end of a connection. At the far end, the security interceptor would be invoked to examine the incoming authentication message. The interceptor would suppress the incoming message so that it would not reach the application object. Instead, the security interceptor might generate an additional outgoing message as a security handshake response.

<!---->

Access Control

...

<!----><!---->

Logging

Log events at interception points.

The store-and-forward mechanism of the Messaging Service could be implemented by a logging mechanism that can record messages into the store (log), and that can replay them out of the store (log).

The Object Transaction Service currently uses, but does not describe, mechanisms for logging -- a serious drawback for commercial systems that may want to use OTS, in contrast to typical commercial transaction services. Without a log of invocations and responses, a recovering server must depend on the clients to reinvoke their transactions. With a log, the server can initiate the redos itself.

The forthcoming fault tolerance service for CORBA will need to log invocations and responses, and to replay such invocations and responses out of the log during recovery.

The most tricky part of the logging interceptor is the replaying, rather than the logging, of messages. The existing DII provides heavyweight mechanisms that could be exploited for the replaying of messages as requests and replies. However, precisely because the DII is heavyweight, and also because logging needs to occur much lower in the ORB, acceptance of the DII as a replay mechanism will not be popular.

When replaying an invocation or response from the log. The recovering object might itself generate invocations of other objects. Those invocations would be matched against the log and suppressed. The appropriate response, extracted from the log, would be generated as an additional inbound message, along with additional inbound messages representing invocations of this object.

<!----><!----><!----><!---->