Interceptors 18

18.1 Introduction

This chapter defines ORB operations that allow services such as security to be inserted in the invocation path. Interceptors are not specific to security; they could be used to invoke any ORB service. These interceptors permit services internal to the ORB to be cleanly separated so that, for example, security functions can coexist with other ORB services such as transactions and replication.

Interceptors are an optional extension to the ORB to allow implementation of the Replaceable Security option defined in the Security Service specification (Chapter 15 of CORBA Services).

Contents

This chapter contains the following sections.

18.1.1 ORB Core and ORB Services

The ORB Core is defined in the CORBA architecture as “that part of the ORB which provides the basic representation of objects and the communication of requests.” ORB Services, such as the Security Services, are built on this core and extend the basic functions with additional qualities or transparencies, thereby presenting a higher-level ORB environment to the application.

The function of an ORB service is specified as a transformation of a given message (a request, reply, or derivation thereof). A client may generate an object request, which necessitates some transformation of that request by ORB services (for example, Security Services may protect the message in transit by encrypting it).

18.2 Interceptors

An interceptor is responsible for the execution of one or more ORB services. Logically, an interceptor is interposed in the invocation (and response) path(s) between a client and a target object. When several ORB services are required, several interceptors may be used.

Two types of interceptors are defined in this specification:

- Request-level interceptors, which execute the given request.
- Message-level interceptors, which send and receive messages (unstructured buffers) derived from the requests and replies.

Interceptors provide a highly flexible means of adding portable ORB Services to a CORBA compliant object system. The flexibility derives from the capacity of a binding between client and target object to be extended and specialized to reflect the mutual requirements of client and target. The portability derives from the definition of the interceptor interface in OMG IDL.

The kinds of interceptors available are known to the ORB. Interceptors are created by the ORB as necessary during binding, as described next.

18.2.1 Generic ORB Services and Interceptors

An Interceptor implements one or more ORB services. Logically, an interceptor is interposed in the invocation (and response) path(s) between a client and target object. There are two types of interceptors:

- **Request-level interceptor**, which perform transformations on a structured request.
- **Message-level interceptors**, which perform transformations on an unstructured buffer.

Figure 18-1 shows interceptors being called during the path of an invocation.

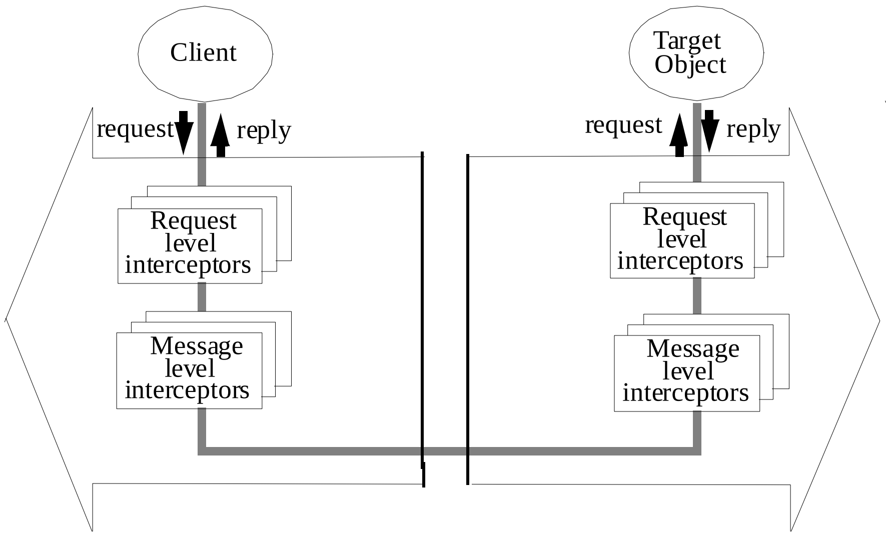
Client
Target Object
request
reply
request
reply
Request level interceptors
Message level interceptors
Request level interceptors
Message level interceptors

Figure 18-1 Interceptors Called During Invocation Path

18.2.2 Request-Level Interceptors

Request-level interceptors are used to implement services which may be required regardless of whether the client and target are collocated or remote. They resemble the CORBA bridge mechanism in that they receive the request as a parameter, and subsequently re-invoke it using the Dynamic Invocation Interface (DII). An example of a request-level interceptor is the Access Control interceptor, which uses information about the requesting principal and the operation in order to make an access control decision.

The ORB core invokes each request level interceptor via the **client_invoke** operation (at the client) or the **target_invoke** operation (at the target) defined in this section. The interceptor may then perform actions, including invoking other objects, before re-invoking the (transformed) request using **CORBA::Request::invoke.** When the latter invocation completes, the interceptor has the opportunity to perform other actions, including recovering from errors and retrying the invocation or auditing the result if necessary, before returning.

18.2.3 Message-Level Interceptors

When remote invocation is required, the ORB will transform the request into a message, which can be sent over the network. As functions such as encryption are performed on messages, a second kind on interceptor interface is required.

The ORB code invokes each message-level interceptor via the **send_message** operation (when sending a message, for example, the request at the client and the reply at the target) or the **receive_message** operation (when receiving a message). Both have a message as an argument. The interceptor generally transforms the message and then invokes **send.** Send operations return control to the caller without waiting for the operation to finish. Having completed the **send_message** operation, the interceptor can continue with its function or return.

18.2.4 Selecting Interceptors

An ORB that uses interceptors must know which interceptors may need to be called, and in what order they need to be called. An ORB that supports interceptors, when serving as a client, uses information in the target object reference, as well as local policy, to decide which interceptors must actually be called during the processing of a particular request sent to a particular target object.

When an interceptor is first invoked, a bind time function is used to set up interceptor binding information for future use.

18.3 Client-Target Binding

The selection of ORB Services is part of the process of establishing a binding between a client and a target object.

A binding provides the context for a client communicating with a target object via a particular object reference. The binding determines the mechanisms that will be involved in interactions such that compatible mechanisms are chosen and client and target policies are enforced. Some requirements, such as auditing or access control, may be satisfied by mechanisms in one environment, while others, such as authentication, require cooperation between client and target. Binding may also involve reserving resources in order to guarantee the particular qualities of service demanded.

Although resolution of mechanisms and policies involves negotiation between the two parties, this need not always involve interactions between the parties as information about the target can be encoded in the object reference, allowing resolution of the client and target requirements to take place in the client. The outcome of the negotiation can then be sent with the request, for example, in the GIOP service context. Where there is an issue of trust, however, the target must still check that this outcome is valid.

The binding between client and target at the application level can generally be decomposed into bindings between lower-level objects. For example, the agreement on transport protocol is an agreement between two communications endpoints, which will generally not have a one-to-one correspondence to application objects. The overall binding therefore includes a set of related sub-bindings which may be shared, and also potentially distributed among different entities at different locations.

18.3.1 Binding Model

No object representing the binding is made explicitly visible since the lifetime of such an object is not under the control of the application, an existing binding potentially being discarded, and a new one made without the application being aware of the fact.

Instead, operations that will affect how a client will interact with a target are provided on the **Object** interface and allow a client to determine how it will interact with the target denoted by that object reference. On the target side, the binding to the client may be accessed through the **Current** interface. This indirect arrangement permits a wide range of implementations that trade-off the communication and retention of binding information in different ways.

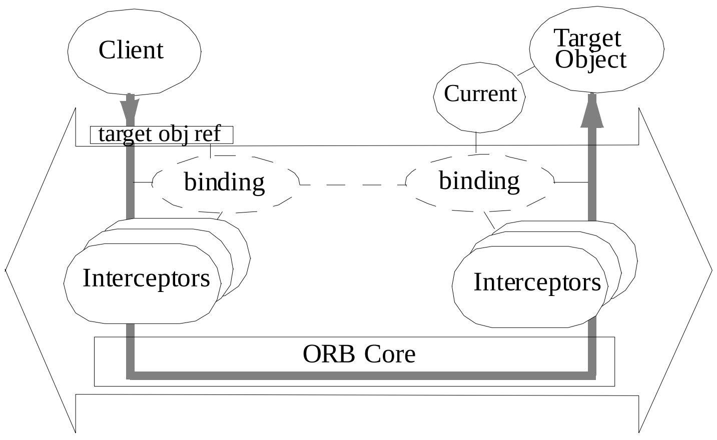

Figure 18-2 Binding Model

The action of establishing a binding is generally implicit, occurring no later than the first invocation between client and target. It may be necessary for a client to establish more than one binding to the same target object, each with different attributes (for example, different security features). In this case, the client can make a copy of the object reference using Object::duplicate and subsequently specify different attributes for that reference.

The scope of attributes associated with an object reference is that of the object reference instance (i.e., the attributes are *not* copied if the object reference is used as an argument to another operation or copied using Object::duplicate). If an object reference is an *inout* argument, the attributes will still be associated with the object reference after the call if the reference still denotes the same object, but not otherwise.

18.3.2 Establishing the Binding and Interceptors

An ORB maintains a list of interceptors, which it supports, and when these are called. Note that at the client, when handling the request, the request-level interceptors are always called before the message level ones, while at the target the message-level ones are called first.

When the ORB needs to bind an object reference, it refers to the characteristics of the target object and relates this to the types of interceptor it supports. From this it determines the appropriate type of interceptor to handle the request and creates it, passing the object reference in the call. (No separate interceptor initialization operation is used. The **client_invoke/target_invoke** or **send_message/receive_message** calls are used both for the first invocation and for subsequent ones.)

When an interceptor is created, it performs its bind time functions. These may involve getting the policies that apply to the client and to the target. This could involve communicating with the target, for example, a secure invocation interceptor setting up a security association. Note that the ORB Core itself is unaware of service-specific policies. In addition to performing its specific functions, the interceptor must continue the request by invoking object(s) derived from the given object reference.

The interceptors themselves maintain per-binding information relevant to the function they perform. This information will be derived from:

- The policies that apply to the client and target object because of the domains to which they belong, for example the access policies, default quality of protection.
- Other static properties of the client and target object such as the security mechanisms and protocols supported.
- Dynamic attributes, associated with a particular execution context or invocation (for example, whether a request must be protected for confidentiality).

If the relevant client or target environment changes, part or all of a binding may need to be reestablished. For example, the secure invocation interceptor may detect that the invocation credentials have changed and therefore needs to establish a new security association using the new credentials. If the binding cannot be reestablished, an exception is raised to the application, indicating the cause of the problem.

Similarly, at the target, the ORB will create an instance of each interceptor needed there. A single interceptor handles both requests and replies at the client (or target), as these share context information.

18.4 Using Interceptors

When a client performs an object request, the ORB Core uses the binding information to decide which interceptors provide the required ORB Services for this client and target as described in “Establishing the Binding and Interceptors” on page 18-5.

18.4.1 Request-Level Interceptors

Request-level interceptors could be used for services such as transaction management, access control, or replication. Services at this level process the request in some way. For example, they may transform the request into one or more lower-level invocations or make checks that the request is permitted. The request-level interceptors, after performing whatever action is needed at the client (or target), reinvoke the (transformed) request using the Dynamic Invocation Interface (DII)

CORBA::Request::invoke. The interceptor is then stacked until the invocation completes, when it has an opportunity to perform further actions, taking into account the response before returning.

Interceptors can find details of the request using the operations on the request as defined in the Dynamic Skeleton interface in CORBA 2. This allows the interceptor to find the target object[1], operation name, context, parameters, and (when complete) the result.

If the interceptor decides not to forward the request, for example, the access control interceptor determines that access is not permitted, it indicates the appropriate exception and returns.

When the interceptor resumes after an inner request is complete, it can find the result of the operation using the **result** operation on the Request object, and check for exceptions using the **exception** operation, etc. before returning.

18.4.2 Message-Level Interceptors

When remote invocation is required, the ORB will transform the request into a message that can be sent over the network. Message-level interceptors operate on messages in general without understanding how these messages relate to requests (for example, the message could be just a fragment of a request). Note that the message interceptors may achieve their purpose not by (just) transforming the given message, but by communicating using their own message (for example, to establish a secure association). Fragmentation and message protection are possible message-level interceptors.

send_message is always used when sending a message, so is used by the client to send a request (or part of a request), and by the target to send a reply.

When a client message-level interceptor is activated to perform a **send_message** operation, it transforms the message as required, and calls a **send** operation to pass the message on to the ORB and hence to its target. Unlike invoke operations, **send** operations may return to the caller without completing the operation. The interceptor can then perform other operations if required before exiting. The client interceptor may next be called either using **send_message** to process another outgoing message, or using **receive_message** to process an incoming message.

A target message-level interceptor also supports **send_message** and **receive_message** operations, though these are obviously called in a different order from the client side.

18.5 Interceptor Interfaces

Two interceptor interfaces are specified, both used only by the ORB:

1.It is assumed that the target object reference is available, as this is described in the C++ mapping for DSI, though not yet in the OMG IDL.

- **RequestInterceptor** for operations on request-level interceptors. Two operations are supported:
  - **client_invoke** for invoking a request-level interceptor at the client.
  - **target_invoke** for invoking a request-level interceptor at the target.
- **MessageInterceptor** for operations on message-level interceptors. Two operations are supported:
  - **send_message** for sending a message from the client to the target or the target to the client.
  - **receive_message** for receiving a message.

Request-level interceptors operate on a representation of the request itself as used in the CORBA Dynamic Invocation and Skeleton interfaces.

18.5.1 Client and Target Invoke

These invoke a request-level interceptor at the client or target. Both operations have identical parameters and return values.

```
module CORBA {
    interface RequestInterceptor: Interceptor {// PIDL
        void client_invoke (
            inout    CORBA::Request       request
        );
        void target_invoke (
            inout    CORBA::Request       request
        );
    };
};
```

Parameters

request	The request being invoked. This is defined in the Dynamic Invocation Interface. After invocation, output parameters and the associated result and exceptions may have been updated.

18.5.2 Send and Receive Message

These invoke a message-level interceptor to send and receive messages. Both operations have identical parameters and return values.

```
module CORBA {
    native Message;
    interface MessageInterceptor: Interceptor {// PIDL
        void send_message (
            in      Object              target,
            in      Message             msg
        );
        void receive_message (
            in      Object              target,
            in      Message             msg
        );
    };
};
```

Parameters

target	The target object reference. Note: The target here may not be the same as seen by the application. For example, a replication request-level interceptor may send the request to more than one underlying object.
msg	The message to be handled by this interceptor.

18.6 IDL for Interceptors

```
module CORBA {
    interface Interceptor {};                       // PIDL
    interface RequestInterceptor: Interceptor {// PIDL
        void client_invoke (
            inout   Request                     request
        );
        void target_invoke (
            inout   Request                     request
        );
    };
    interface MessageInterceptor: Interceptor {// PIDL
        void send_message (
            in      Object                      target,
            in       Message                    msg
        );
        void receive_message (
            in      Object                      target,
            in      Message                     msg
        );
    };
};
```