[Top] [Prev] [Next] [Bottom]

The Basic Object Adapter

8

An object adapter is the primary interface that an implementation uses to access ORB functions. The *Basic Object Adapter* (BOA) is an interface intended to be widely available and to support a wide variety of common object implementations. It includes convenient interfaces for generating object references, registering implementations that consist of one or more programs, activating implementations, and authenticating requests. It also provides a limited amount of persistent storage for objects that can be used for connecting to a larger or more general storage facility, for storing access control information, or other purposes.

Most of the *Basic Object Adapter* interface can be expressed in OMG IDL, since the interface is to the operations on the object adapter. Some of the operations to bind the implementation to the object adapter depend on the language mapping. Such dependencies are noted in this chapter, but OMG IDL will be used to describe the interface.

All types defined in this chapter are part of the CORBA module. When referenced in OMG IDL, the type names must be prefixed by "CORBA::".

8.1 Role of the Basic Object Adapter

One object adapter, called the Basic Object Adapter, should be available in every ORB implementation; although the BOA will generally have an ORB-dependent implementation, object implementations that use it should be able to run on any ORB that supports the required language mapping, assuming they have been installed appropriately.

Other Object Adapters are likely to be created. Ordinarily, it is not necessary for a client of an object to be concerned about which Object Adapter is used by the implementation.

The following functions are provided through the Basic Object Adapter:

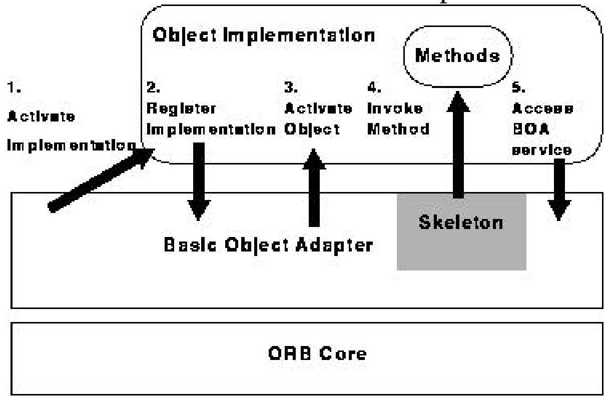
- Generation and interpretation of object references
- Authentication of the principal making the call
- Activation and deactivation of the implementation
- Activation and deactivation of individual objects
- Method invocation through skeletons

The Basic Object Adapter supports object implementations that are constructed from one or more programs[1]. The BOA activates and communicates with these programs using operating system facilities that are not part of the ORB. Therefore the BOA requires some information that is inherently non-portable. Although not defining this information, the BOA does define the concept of an Implementation Repository which can hold this information, allowing each system to install and start implementations in the way that is appropriate for that system.

The mechanism for binding the program to the BOA and ORB is also not specified because it is inherently system and language-dependent. We assume that the BOA can connect the methods to the skeleton by some means, whether at the time the implementation is compiled, installed, or activated, etc. Subsequent to activation, the BOA can make calls on routines in the implementation and the implementation can make calls on the BOA.

Figure 14 on page 8-3 shows the structure of the Basic Object Adapter, and some of the interactions between the BOA and an Object Implementation. The Basic Object Adapter will start a program to provide the Object Implementation, in this example, a per-class server (1). The Object Implementation notifies the BOA that it has finished initializing and is prepared to handle requests (2). When the first request for a particular object arrives, the implementation is notified to activate the object (3). On subsequent requests, the BOA calls the appropriate method using the per-interface skeleton (4). At various times, the implementation may access BOA services such as object creation, deactivation, and so forth. (5).

FIGURE 14. The Structure and Operation of the Basic Object Adapter



The BOA exports operations that are accessed by the Object Implementation. The BOA also calls the Object Implementation under certain circumstances. The interface between a particular version of the BOA and the ORB Core it runs on is private, as is the interface between the BOA and the skeletons. Thus, the BOA can exploit features or overcome limitations of a specific ORB Core, and can cooperate with the ORB Core and skeletons to provide a set of portable interfaces for the object implementation.

8.2 Basic Object Adapter Interface

The BOA interface is specified in OMG IDL, so that the way it is accessed in any programming language is specified by the client side language mapping for that language. Some data structures used by the BOA are specific to a given language mapping, so most IDL compilers will not be able to accept this definition literally.

In practice, the BOA is most likely to be implemented partially as a separate component and partially as a library in the Object Implementation. The separate component is required to do activation when the implementation is not present. The library portion is needed to establish the linkage between the methods and the skeleton. The exact partitioning of functionality between these parts is implementation dependent. Generally, there will appear to be a BOA object in the object implementation. When it is invoked, some operations are satisfied in the library, some in an external server, and some in the ORB Core.

The following is the approximate interface definition for the BOA object. More details will be provided as the operations are discussed.

```
module CORBA {

interface InterfaceDef; // from Interface Repository // PIDL

interface ImplementationDef; // from Implementation Repository

interface Object; // an object reference

interface Principal; // for the authentication service

typedef sequence <octet, 1024> ReferenceData;

interface BOA {

Object create (

in ReferenceData id,

in InterfaceDef intf,

in ImplementationDef impl

);

void dispose (in Object obj);

ReferenceData get_id (in Object obj);

void change_implementation (

in Object obj,

in ImplementationDef impl

);

Principal get_principal (

in Object obj,

in Environment ev
```

);

void set_exception (

in exception_type major, // NO, USER,

//or SYSTEM_EXCEPTION

in string userid, // exception type id

in void *param // pointer to associated data

);

void impl_is_ready (in ImplementationDef impl);

void deactivate_impl (in ImplementationDef impl);

void obj_is_ready (in Object obj, in ImplementationDef impl);

void deactivate_obj (in Object obj);

};

};

Requests by an implementation on the BOA are of the following kinds:

- Operations to create or destroy object references, or query or update the information the BOA maintains for an object reference.
- Operations associated with a particular request.
- Operations to maintain a registry of active objects and implementations.

Requests by the BOA to an implementation are made with skeletons or using an implementation's runtime language mapping information, and are of these kinds:

- Activating an implementation.
- Activating an object.
- Performing an operation (through a skeleton method).

Each of the BOA operations is described in detail later in this section; the requests of the BOA to an implementation are described in the language mapping section.

8.2.1 Registration of Implementations

The Basic Object Adapter expects information describing the implementations to be stored in an *Implementation Repository*. The Implementation Repository ordinarily is updated at program installation time, but may be set up incrementally or otherwise. There are objects with an OMG IDL interface called **ImplementationDef**, which capture this information. The Implementation Repository may contain additional information for debugging, administration, etc. Note that the Implementation Repository is logically distinct from the Interface Repository, although they may in fact be implemented together.

The *Interface Repository* contains information about interfaces. There are objects with an OMG IDL interface called **InterfaceDef**, which capture this information. The Interface Repository may contain additional information for debugging, administration, browsing, etc. The ORB Core may or may not make use of the Interface Repository or the Implementation Repository, but the ORB and BOA use these objects to associate object references with their interfaces and implementations.

8.2.2 Activation and Deactivation of Implementations

There are two kinds of activation that a BOA needs to perform as part of operation invocation. The first, discussed in this section, is *implementation activation*, which occurs when no implementation for an object is currently available to handle the request. The second, discussed later, is *object activation*, which occurs when no instance of the object is available to handle the request.

Implementation activation requires coordination between the BOA and the program(s) containing the implementation. This manual uses the term *server* as the separately executable entity that the BOA can start on a particular system. In a POSIX environment, a server would be a process. In most systems, a server corresponds to the notion of a program, but it can correspond to whatever the appropriate system facility is in a particular environment.

The BOA initiates activity by the implementation by starting the appropriate server, probably in an operating system-dependent way. The implementation initializes itself, then notifies the BOA that it is prepared to handle requests by calling **impl_is_ready** or **obj_is_ready**[2].

Between the time that the program is started and it indicates it is ready, the BOA will prevent any other requests from being delivered to the server. After that point, the BOA, through the skeletons, will make calls on the methods of the implementation.

void impl_is_ready (in ImplementationDef impl); // PIDL

void obj_is_ready (

in Object obj,

in ImplementationDef impl

);

An *activation policy* describes the rules that a given implementation follows when there are multiple objects or implementations active. There are four policies that all BOA implementations support for implementation activation:

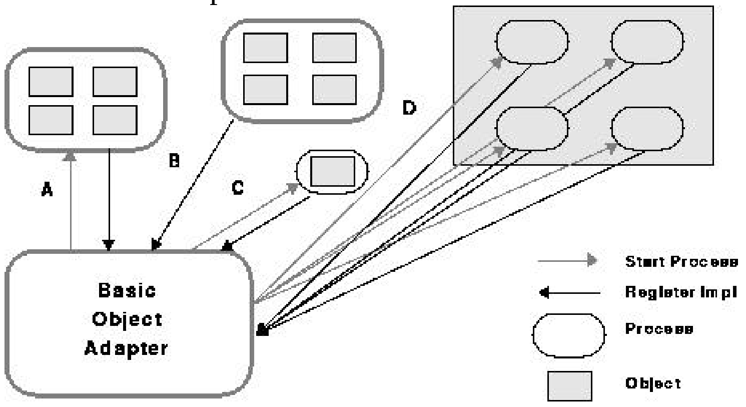
- A *shared server* policy, where multiple active objects of a given implementation share the same server.
- An *unshared server* policy, where only one object of a given implementation at a time can be active in one server.
- A *server-per-method* policy, where each invocation of a method is implemented by a separate server

being started, with the server terminating when the method completes.

- *Persistent server* policy, where the server is activated by something outside the BOA. The server nonetheless must register with the BOA to receive invocations. A persistent server is assumed to be shared by multiple active objects.

These kinds of implementation activation are illustrated in Figure 15 on page 8-7. Case A is a shared server, where the BOA starts a process which then registers itself with the BOA. Case B is the case of a persistent server, which is very similar but just registers itself with the BOA, without the BOA having had to start a process. An unshared server is illustrated in case C, where the process started by the BOA can only hold one object; the server-per-method policy in case D causes each method invocation to be done by starting a process.

FIGURE 15. Implementation Activation Policies



Shared Server Activation Policy

In a shared server, multiple objects may be implemented by the same program. This is likely to be the most common kind of server. The server is activated the first time a request is performed on any object implemented by that server. When the server has initialized itself, it notifies the BOA that it is ready by calling **impl_is_ready**. Subsequently, the BOA will deliver requests or object activations for any objects implemented by that server. The server remains active and will receive requests until it calls **deactivate_impl**. The BOA will not activate another server for that implementation if one is active.

Before the first request is delivered for a particular object, the object activate routine of the server is called. An object remains active as long as its server is active, unless the server calls **deactivate_obj** for that object.

Unshared Server Activation Policy

In an unshared server, each object is implemented in a different server. This kind of server is convenient if a object is intended to encapsulate an application or if the server requires exclusive access to a resource such as a printer. A new server is activated the first time a request is performed on the object. When the server has initialized itself, it notifies the BOA that it is ready by calling **obj_is_ready**. Subsequently, the BOA will deliver requests for that object. The server remains active and will receive requests until it calls **deactivate_obj**.

A new server is started whenever a request is made for an object that is not yet active, even if a server for another object with the same implementation is active.

Server-per-Method Activation Policy

Under the server-per-method policy, a new server is always started each time a request is made. The server runs only for the duration of the particular method. Several servers for the same object or even the same method of the same object may be active simultaneously. Because a new server is started for each request, it is not necessary for the implementation to notify the BOA when an object is ready or deactivated.

The BOA activates an implementation for each request, whether or not another request for that operation, object, or implementation is active at the same time.

Persistent Server Activation Policy

Persistent servers are those servers which are activated by means outside the BOA. Such implementations notify the BOA that they are available using the **impl_is_ready** operation. Once the BOA knows about a persistent server, it treats the server as a shared server, sending it activations for individual objects and method calls. If no implementation is ready when a request arrives, an error is returned for that request.

8.2.3 Generation and Interpretation of Object References

Object references are generated by the BOA using the ORB Core when requested by an implementation. The BOA and the ORB Core work together to associate some information with a particular object reference. This information is later provided to the implementation upon the activation of an object. Note that this is the only information an implementation may use portably to distinguish different object references. The BOA operation used to create a new object reference is:

Object create (// PIDL

in ReferenceData id,

in InterfaceDef intf,

in ImplementationDef impl

);

The **id** is immutable identification information, chosen by the implementation at object creation time, and never changed during the lifetime of the object. The **intf** is the Interface Repository object that specifies the complete set of interfaces implemented by the object. The **impl** is the Implementation Repository object that specifies the implementation to be used for the object.

A typical implementation will use the **id** value to distinguish different objects, but it is free to use it in any way it chooses or to assign the same value to different object references. Two object references created with the same parameters are *not* the same object reference as far as the ORB is concerned, although the implementation may or may not treat them as references to the same object. Note that the object reference itself is opaque and may be different for different ORBs, but the **id** value is available portably in all ORBs. Only the implementation can normally interpret the **id** value. The operation to get the **id** is a

BOA operation:

ReferenceData get_id (in Object obj); // PIDL

It is possible for the implementation associated with an object reference to be changed. This will cause subsequent requests to be handled according to the information in the new implementation. The operation to set the implementation is a BOA operation:

void change_implementation (// PIDL

in Object obj,

in ImplementationDef impl

);

Note ¯ Care must be taken in order to change the implementation after the object has been created. There are issues of synchronization with activation, security, and whether or not the new implementation is prepared to handle requests for that object. The **change_implementation** operation affects all copies of that particular object reference.

If an object reference is copied, all copies have the same **id**, **intf**, and **impl.**

An implementation is allowed to dispose of an object it has created by asking the BOA to invalidate the object reference. The implementation is responsible for deallocating all other information about the object. After a **dispose** is done, the ORB Core and BOA act as if the object had never been created, and attempts to issue requests on any existing object references for that object will fail.

void dispose (in Object obj); // PIDL

Note that all of the operations on object references in this section may be done whether or not the object is active.

8.2.4 Authentication and Access Control

The BOA does not enforce any specific style of security management. It guarantees that for every method invocation (or object activation) it will identify the principal on whose behalf the request is performed. The object implementation can obtain this principal by the operation:

Principal get_principal (// PIDL

in Object obj,

in Environment ev

);

The **obj** parameter is the object reference passed to the method. If another object is used the result is undefined. The **ev** parameter is the language-mapping-specific request environment passed to the method.

The meaning of the principal depends on the security environment that the implementation is running in. The decision of whether or not to permit a particular operation is left up to the implementation. Typically, an implementation will associate access rights with particular objects and principals, and will examine those access rights to determine if the principal making the request has the privileges required by the particular method. An implementation could store a reference to the access control information for an object in the **id** for the object.

8.2.5 Persistent Storage

Objects (or, more precisely, object references) are made persistent by the BOA and the ORB Core, in that a client that has an object reference can use it at any time without warning, even if the implementation has been deactivated or the system has been restarted. Although the ORB Core and BOA maintain the persistence of object references, the implementation must participate in keeping any data outside the ORB Core and BOA persistent.

Toward this end, the BOA provides a small amount of storage for an object in the **id** value. In most cases, this storage is insufficient and inconvenient for the complete state of the object. Instead, the implementation provides and manages that storage, using the **id** value to locate the actual storage. For example, the **id** value might contain the name of a file, or a key for a database system that holds the persistent state.

[Top] [Prev] [Next] [Bottom]

---

[1] The term "program" is meant to include a wide range of possible constructs, including scripts, loadable modules, etc., in addition to the traditional notions of an application or server.

[2] The latter is for per-object servers.