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

ORB Interface

7

The ORB interface is the interface to those ORB functions that do not depend on which object adapter is used. These operations are the same for all ORBs and all object implementations, and can be performed either by clients of the objects or implementations. Some of these operations appear to be on the ORB, others appear to be on the object reference. Because the operations in this section are implemented by the ORB itself, they are not in fact operations on objects, although they may be described that way and the language binding will, for consistency, make them appear that way.

The ORB interface also defines operations for creating lists and determining the default context used in the Dynamic Invocation Interface. Those operations are described in Chapter 4.

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::".

7.1 Converting Object References to Strings

Because an object reference is opaque and may differ from ORB to ORB, the object reference itself is not a convenient value for storing references to objects in persistent storage or communicating references by means other than invocation. Two problems must be solved: allowing an object reference to be turned into a value that a client can store in some other medium, and ensuring that the value can subsequently be turned into the appropriate object reference.

An object reference may be translated into a string by the operation **object_to_string**. The value may be stored or communicated in whatever ways strings may be manipulated. Subsequently, the **string_to_object** operation will accept a string produced by **object_to_string** and return the corresponding object reference.

```
module CORBA {

interface ORB { // PIDL

string object_to_string (in Object obj);

Object string_to_object (in string str);

Status create_list (
```

in long count,

out NVList new_list

);

Status create_operation_list (

in OperationDef oper,

out NVList new_list

);

Status get_default_context (out Context ctx);

};

};

To guarantee that an ORB will understand the string form of an object reference, that ORB's **object_to_string** operation must be used to produce the string. Since in general a client does not know or care which ORB is used for a particular object reference, the client can choose whatever ORB is convenient.

For a description of the **create_list** and **create_operation_list** operations, see "List Operations" on page 4-10. The **get_default_context** operation is described in the section "get_default_context" on page 4-14.
.

7.2 Object Reference Operations

There are some operations that can be done on any object. These are not operations in the normal sense, in that they are implemented directly by the ORB, not passed on to the object implementation. We will describe these as being operations on the object reference, although the interfaces actually depend on the language binding. As above, where we used interface Object to represent the object reference, we will define an interface for Object:

module CORBA {

interface Object { // PIDL

ImplementationDef get_implementation ();

InterfaceDef get_interface ();

boolean is_nil();

Object duplicate ();

```
void release ();

boolean is_a (in string logical_type_id);

boolean non_existent();

boolean is_equivalent (in Object other_object);

unsigned long hash(in unsigned long maximum);

Status create_request (

in Context ctx,

in Identifier operation,

in NVList arg_list,

inout NamedValue result,

out Request request,

in Flags req_flags

);

};

};
```

The **create_request** operation is part of the Object interface because it creates a pseudo-object (a Request) for an object. It is described with the other Request operations in the section "Request Operations" on page 4-4.

7.2.1 Determining the Object Implementation and Interface

An operation on the object reference, **get_interface**, returns an object in the Interface Repository, which provides type information that may be useful to a program. See Chapter 6 for a definition of operations on the Interface Repository. An operation on the Object called **get_implementation** will return an object in an implementation repository that describes the implementation of the object. See the Basic Object Adapter chapter for information about the Implementation Repository.

```
InterfaceDef get_interface (); // PIDL

ImplementationDef get_implementation ();
```

7.2.2 Duplicating and Releasing Copies of Object References

Because object references are opaque and ORB-dependent, it is not possible for clients or implementations to allocate storage for them. Therefore, there are operations defined to copy or release an object reference.

Object duplicate (); // PIDL

void release ();

If more than one copy of an object reference is needed, the client may create a **duplicate**. Note that the object implementation is not involved in creating the duplicate, and that the implementation cannot distinguish whether the original or a duplicate was used in a particular request.

When an object reference is no longer needed by a program, its storage may be reclaimed by use of the **release** operation. Note that the object implementation is not involved, and that neither the object itself nor any other references to it are affected by the **release** operation.

7.2.3 Nil Object References

An object reference whose value is OBJECT_NIL denotes no object. An object reference can be tested for this value by the **is_nil** operation. The object implementation is not involved in the nil test.

boolean is_nil (); // PIDL

7.2.4 Equivalence Checking Operation

An operation is defined to facilitate maintaining type-safety for object references over the scope of an ORB.

boolean is_a(in string logical_type_id); // PIDL

The **logical_type_id** is a string denoting a shared type identifier (RepositoryId). The operation returns true if the object is really an instance of that type, including if that type is an ancestor of the "most derived" type of that object.

This operation exposes to application programmers functionality that must already exist in ORBs which support "type safe narrow", and allows programmers working in environments that do not have compile time type checking to explicitly maintain type safety.

7.2.5 Probing for Object Non-Existence

boolean non_existent (); // PIDL

The **non_existent** operation may be used to test whether an object (e.g. a proxy object) has been destroyed. It does this without invoking any application level operation on the object, and so will never affect the object itself. It returns true (rather than raising **CORBA::OBJECT_NOT_EXIST**) if the ORB knows authoritatively that the object does not exist, and otherwise it returns false.

Services that maintain state that includes object references, such as bridges, event channels, and base relationship services, might use this operation in their "idle time" to sift through object tables for objects that no longer exist, deleting them as they go, as a form of garbage collection. In the case of proxies, this kind of activity can cascade, such that cleaning up one table allows others then to be cleaned up.

7.2.6 Object Reference Identity

In order to efficiently manage state that include large numbers of object references, services need to support a notion of object reference identity. Such services include not just bridges, but relationship services and other layered facilities.

unsigned long hash(in unsigned long maximum); // PIDL

boolean is_equivalent(in Object other_object);

Two identity-related operations are provided. One maps object references into disjoint groups of potentially equivalent references, and the other supports more expensive pairwise equivalence testing. Together, these operations support efficient maintenance and search of tables keyed by object references.

Hashing: Object Identifiers

Object references are associated with ORB-internal identifiers which may indirectly be accessed by applications using the **hash()** operation. The value of this identifier does not change during the lifetime of the object reference, and so neither will any hash function of that identifier.

The value of this operation is not guaranteed to be unique; that is, another object reference may return the same hash value. However, if two object references hash differently, applications can determine that the two object references are *not* identical.

The **maximum** parameter to the **hash** operation specifies an upper bound on the hash value returned by the ORB. The lower bound of that value is zero. Since a typical use of this feature is to construct and access a collision chained hash table of object references, the more randomly distributed the values are within that range, and the cheaper those values are to compute, the better.

For bridge construction, note that proxy objects are themselves objects, so there could be many proxy objects representing a given "real" object. Those proxies would not necessarily hash to the same value.

Equivalence Testing

The **is_equivalent()** operation is used to determine if two object references are equivalent, so far as the ORB can easily determine. It returns `TRUE` if the target object reference is known to be equivalent to the other object reference passed as its parameter, and `FALSE` otherwise.

If two object references are identical, they are equivalent. Two different object references which in fact refer to the same object are also equivalent.

ORBs are allowed, but not required, to attempt determination of whether two distinct object references refer to the same object. In general, the existence of reference translation and encapsulation, in the absence of an omniscient topology service, can make such determination impractically expensive. This means that a FALSE return from **is_equivalent()** should be viewed as only indicating that the object references are distinct, and not necessarily an indication that the references indicate distinct objects.

A typical application use of this operation is be to match object references in a hash table. Bridges could use it to shorten the lengths of chains of proxy object references. Externalization services could use it to "flatten" graphs that represent cyclical relationships between objects. Some might do this as they construct the table, others during idle time.

7.3 Overview: ORB and OA Initialization and Initial References

Before an application can enter the CORBA environment, it must first:

- Be initialized into the ORB and object adapter (BOA and OA) environments.
- Get references to ORB and OA (including BOA) pseudo objects-and sometimes to other objects-for use in future ORB and OA operations.

CORBA V2.0 provides operations, specified in PIDL, to initialize applications and obtain the appropriate object references. The following is provided:

- Operations providing access to the ORB. These operations reside in CORBA module, but not in the ORB interface and are described in Section 7.4, "ORB Initialization," on page 7-6.
- Operations providing access to the Basic Object Adapter (BOA) and other object adapters (OAs) These operations reside in the ORB interface and are described in Section 7.5, "OA and BOA Initialization," on page 7-8.
- Operations providing access to the Interface Repository, Naming Service, and other Object Services. These operations reside in the ORB interface and are described in Section 7.6, "Obtaining Initial Object References," on page 7-10.

In addition, this manual provides a mapping of the PIDL initialization and object reference operations to the C and C++ programming languages. For mapping information, refer to Section 14.26, "ORB and OA/BOA Initialization Operations," on page 14-30 and to Section 17.12, "ORB," on page 17-11.

7.4 ORB Initialization

When an application requires a CORBA environment it needs a mechanism to get ORB and OA pseudo-object references. This serves two purposes. First, it initializes an application into the ORB and OA environments. Second, it returns the ORB and OA pseudo object references to the application for use in future ORB and OA operations. The ORB and BOA initialization operations must be ordered with ORB occurring before OA: an application cannot call OA initialization routines until ORB initialization routines have been called for the given ORB.

The operation to initialize an application in the ORB and get its pseudo object reference is not performed on an object. This is because applications do not initially have an object on which to invoke operations. The ORB initialization operation is an application's bootstrap call into the CORBA world. The PIDL for the call (Figure 7-1) shows that the **ORB_init** call is part of the CORBA module but not part of the ORB interface.

Applications can be initialized in one or more ORBs. When an ORB initialization is complete, its pseudo reference is returned and can be used to obtain OA references for that ORB.

In order to obtain an ORB pseudo object reference, applications call the **ORB_init** operation. The parameters to the call comprise an identifier for the ORB for which the pseudo object reference is required, and an **arg_list,** which is used to allow environment-specific data to be passed into the call. PIDL for the ORB intialization is as follows:

```
// PIDL

module CORBA {

typedef string ORBid;

typedef sequence <string> arg_list;

ORB ORB_init (inout arg_list argv, in ORBid orb_identifier);

};
```

Figure 7-1

The identifier for the ORB will be a name of type ORBid (string). The allocation of ORBids is the responsibility of ORB administrators and is not intended to be managed by the OMG. Names are locally scoped and the ORB administrator is responsible for ensuring that the names are unambiguous. Examples of potential ORBids are "Internet ORB," "BNR_private," "BNR_interop_1_2." If a NULL ORBid is used then arg_list arguments can be used to determine which ORB should be returned. This is achieved by searching the arg_list parameters for one tagged ORBid, for example, -ORBid "ORBid_example." Other parameters of significance to the ORB can be identified, for example, "Hostname," "SpawnedServer," and so forth. To allow for other parameters to be specified without causing applications to be re-written, it is necessary to specify the format that ORB parameters may take. The format of those parameters will be

-ORB<suffix> <value>.

The **ORB_init** operation can be called any number of times and is expected to return the same pseudo object reference for the same parameters. Calling the **ORB_init** function multiples times for the same ORB may be required where an ORB is implemented as a shared library, or where several threads of a multi-threaded application require to use the same ORB and all wish to call the **ORB_init** operation.

7.5 OA and BOA Initialization

An ORB may have zero or more object adaptors associated with it. Servers must have a reference to an OA pseudo-object in order to access its functionality.

The only object adaptor defined in *CORBA* is the Basic Object Adaptor (BOA). However other adaptors such as the Library Object Adaptor (LOA) are also mentioned. Given an ORB reference, an application must be able to initialize itself in an OA environment and get the pseudo reference of the OA from the ORB.

Because OAs are pseudo-objects and therefore do not necessarily share a common interface, it is not possible to have a generic OA_init operation that returns an object type which is then explicitly narrowed or widened to the correct pseudo-object type. It is therefore necessary to provide an initialization function for each OA type separately. To achieve this a template is suggested for OA initialization, and the BOA initialization operation is generated from that template.

The operation to get the OA pseudo object reference is part of the ORB interface. The **<OA>_init** operation is therefore an operation on the ORB pseudo object. Figure 7-2 shows the PIDL for the for the **<OA>_init** (specifically BOA_init) operation.

```
// PIDL

module CORBA {

interface ORB

{

typedef sequence <string> arg_list;

typedef string OAid;

// Template for OA initialization operations

// <OA> <OA>_init (inout arg_list argv,

// in OAid oa_identifier);

BOA BOA_init (inout arg_list argv,

in OAid boa_identifier);

};

}
```

Figure 7-2

The identifier for the OA will be a name of the type **OAid** (string). The allocation of OAids is the responsibility of ORB administrators and is not intended to be managed by the OMG. Names are locally scoped and the ORB administrator is responsible for ensuring that the names are unambiguous. Examples of potential **OAid**s are "BOA," "BNR_BOA," "HP_LOA."

If a NULL OAid is used then arg_list arguments can be used to determine which OA should be returned. This is achieved by searching the **arg_list** parameters for one tagged OAid, e.g. -OAid "OAid_example".

In order to allow for other OA parameters to be specified in the future without causing applications to be re-written it is necessary to specify the format parameters may take. The format of OA specific parameters will be **- OA<suffix> <value>**.

The BOA_init function may be called any number of times and is expected to return the same pseudo object reference for the same parameters. Calling the **BOA_init** operation multiples times for the same BOA may be required where several threads of a multi-threaded application require to use the same BOA and therefore need to the **BOA_init** operation.

The **BOA_init** operation returns a BOA. Once the operation has returned the BOA is assumed to be initialised for the application object.

7.6 Obtaining Initial Object References

Applications require a portable means by which to obtain their initial object references. References are required for the Interface Repository and Object Services. (The Interface Repository is described in Chapter 6 of this manual; Object Services are described in *CORBAservices*.) The functionality required by the application is similar to that provided by the Naming Service. However, the OMG does not want to mandate that the Naming Service be made available to all applications in order that they may be portably initialized. Consequently, the operations shown in this section provide a simplified, local version of the Naming Service that applications can use to obtain a small, defined set of object references which are essential to its operation. Because only a small well defined set of objects are expected with this mechanism, the naming context can be flattened to be a single-level namespace. This simplification results in only two operations being defined to achieve the functionality required.

Initial references are not obtained via a new interface; instead two new operations are added to the ORB pseudo-object interface, providing facilities to list and resolve initial object references. Figure 7-3 on page 7-10 shows the PIDL for these operations.

```
// PIDL interface for getting initial object references

module CORBA {

interface ORB {

typedef string ObjectId;

typedef sequence <ObjectId> ObjectIdList;

exception InvalidName {};

ObjectIdList list_initial_services ();

Object resolve_initial_references (in ObjectId identifier)

raises (InvalidName);

}
```

}

Figure 7-3

The **resolve_initial_references o**peration is an operation on the ORB rather than the Naming Service's **NamingContext**. The interface differs from the Naming Service's resolve in that **ObjectId** (a string) replaces the more complex Naming Service construct (a sequence of structures containing string pairs for the components of the name). This simplification reduces the name space to one context.

ObjectIds are strings that identify the object whose reference is required. To maintain the simplicity of the interface for obtaining initial references, only a limited set of objects are expected to have their references found via this route. Unlike the ORB and BOA identifiers, the **ObjectId** name space requires careful management. To achieve this. the OMG may, in the future, define which services are required by applications through this interface and specify names for those services.

Currently, reserved **ObjectIds** are **InterfaceRepository** and **NameService.**

To allow an application to determine which objects have references available via the initial references mechanism, the **list_initial_services** operation (also a call on the ORB) is provided. It returns an **ObjectIdList,** which is a sequence of **ObjectIds**. **ObjectIds** are typed as strings. Each object, which may need to be made available at initialization time, is allocated a string value to represent it. In addition to defining the id, the type of object being returned must be defined, i.e. "InterfaceRepository" returns a object of type Repository, and "NameService" returns a **CosNamingContext** object.

The application is responsible for narrowing the object reference returned from **resolve_initial_references** to the type which was requested in the ObjectId. E.g. for InterfaceRepository the object returned would be narrowed to **Repository** type.

In the future, specifications for Object Services (in *CORBAservices*) will state whether it is expected that a service's initial reference be made available via the **resolve_initial_references** operation or not, i.e. whether the service is necessary or desirable for bootstrap purposes.

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