Bionic Buffalo Tech Note #100

Identifying CORBA Objects

last revised Tuesday 2003.04.01

Overview

This paper discusses the various means available to applications for identification of, and reference to, CORBA objects. These include the use of object references, IORs, URLs, and keys.

Object Reference Creation and Destruction

The CORBA model is not very specific about the creation and destruction of objects. There is a Life Cycle Service specification, which describes some interfaces which might be inherited by some classes, and which might be used by clients to cause the creation and destruction of objects. However, the Life Cycle Service interfaces are only conventions and their implementation is not required in all CORBA implementations.

Instead of examining the abstract model, we will instead discuss the practical aspects of client-object interaction.

The object request broker (ORB) is the starting point for all client interaction with objects. In its communication with an ORB, an application uses *object references.* An object reference is a handle used to refer to an object. It is akin to a file or stream handle provided by an operating system when a file or stream is opened. All object references are created by ORBs. An object reference is, in general, not portable, and is not guaranteed to be meaningful to any ORB other than to the one which created it. An application may talk to several ORBs at once or consecutively, but (with some exceptions to be discussed below) must keep separate the object references created by one ORB from the object references created by any other ORB.

Although ORBs create object references, they do not create objects. Objects are implemented by *servants,* which are programs that communicate with ORBs, and which give ORBs access to objects. However, an object is a logical concept, and an object may exist prior to the creation of a servant, and may survive the destruction of a servant. For example, a file may be an object, or a database may be an object while

containing other objects, and these may persist indefinitely. On the other hand, an object may be ephemeral, and may exist only as long as is needed to perform some operation, and then be destroyed.

Servants communicate with ORBs by means of *adapters*. The common specification for the adapter-servant interface is described in the IDL of the `PortableServer` module. There are other models used for adapters, but the `PortableServer` model has become the most common, and its architecture will be assumed in the rest of this document.

An ORB may have more than one adapter. The adapters are organized into a tree, whose base is the *root adapter*. Adapters are created by other adapters, and all trace their ancestry back to the root adapter. All adapters, whether or not they are terminal nodes in the tree, communicate with servants. (In other words, an adapter may have descendent adapters, and also communicate with servants.) Each adapter has a name, and a given adapter can be referenced from the ORB by following a path name from the root, similar to the path name of a file in a traditional file system.

Adapters sit between the ORB and the servants, and provide services to both. Each adapter may associate with zero or more servants, and each servant with zero or more objects. (The association of zero objects with a servant isn't normally a permanant condition, since connecting with objects is the reason servants exist.)

In addition to providing a namespace for child adapters, an adapter also provides a namespace for objects. Each object managed by an adapter is associated with an opaque sequence of bytes, the *object id*. An object id uniquely identifies an object within an adapter. Typically, the servants think in terms of object ids, and the ORB thinks in terms of object references. The adapter provides the mapping from one to the other.

The servants may use any convenient data as object ids. For instance, a servant representing a traditional file system (files and directories) as objects might use the fully-qualified path name of the file as the corresponding object id.

Since an object reference refers uniquely to an object, this leads to the one-to-one correspondence between each object reference and an <*adapter, object id*> pair. Although the internal structure of an object reference is considered opaque to applications, some ORB implementations in fact encapsulate the adapter name and object id into the object reference, for a straightforward mapping when needed. Other ORB implementations use lookup tables or other means to convert among object references and <*adapter, object id*> pairs.

It is possible that a single object may be represented by more than one servant or adapter. For this reason (among others), it is not in general possible to determine always if two different object references map to the same object.

Obtaining Object References

The first object reference obtained by an application is that of the ORB, which is itself an object. This is done through the operation, `CORBA::ORB_init`. Each language mapping defines a syntax for `CORBA::ORB_init`. Once the application has the ORB's object reference, it may then invoke ORB operations to obtain references for initial services.

The two most important initial services for obtaining additional object references are the Naming Service and the Trading Service, although other services are also defined. The Naming Service provides a standard interface to a tree-structured directory of names, similar to that of the directory associated with a traditional file system. The Trader Service is similar to a database, where objects can be looked up based on service type, interface, and name-value property pairs. It is not required that an object be discoverable using any such service, and it is possible that an object may be accessible using more than one such service. Some ORBs do not provide either of these standard initial services.

Interoperable Object References

An object reference is not, in general, portable among different ORBs. However, there is a need for ORBs to interoperate, so a special *interoperable object reference*, or *IOR*, is defined.

Rather than discussing interoperability in terms of disparate ORBs, the specifications introduce the concept of *domains*. The particular kind of domain of interest here is a *referencing domain*, within which a given object reference is valid. A referencing domain may encompass one or more ORBs. Among different referencing domains, IORs may be used. As with ordinary object references, IORs are opaque to applications. An application deals with object references, which the ORB maps to and from IORs as needed.

An IOR is created by an ORB. The IOR contains one or more profiles, each of which is associated with a communications protocol. (However, when more than one profile is present, some of these may contain information shared among multiple protocols.) The information in a profile allows another ORB to communicate with the originating ORB using the designated protocol. For example, the profiles used with the internet protocol include the internet address and port of the ORB which manages the object, as well as an opaque octet sequence (the *key*) used by the managing ORB to identify the specific object associated with the IOR. Although the ORB might use an object's <*adapter*, *object id*> pair as the key, this is not mandatory, and any other suitable mechanism might be used.

In addition to network address and object identification information, a profile may contain additional information related to ORB interoperability. Such information may include object type information, policy values, security parameters, code set

identification, and ORB descriptions.

Sometimes it is convenient to represent opaque IORs as strings, so they can be stored an manipulated by applications. For this purpose, two operations (`CORBA::ORB::object_to_string` and `CORBA::ORB::string_to_object`) are defined, to convert between an object reference and the string form of an object's IOR. These strings are called *stringified IORs*, and consist of the characters `"ior:"` or `"IOR:"`, followed by a series of hexadecimal digits encoding the IOR.

The `CORBA::ORB::string_to_object` operation, in addition to supporting stringified IORs, also supports two other formats, `corbaloc` and `corbaname`. These are discussed below.

References in `corbaloc` Format

References in `corbaloc` format are strings beginning with `"corbaloc:"`, and which conform to a specific format allowing resolution to an object. There are two variants of `corbaloc` reference: one is based on the `CORBA::ORB::resolve_initial_references` operation, and the other is based on protocol and key (as used in IORs).

The `resolve_initial_references` operation is used by an application to obtain initial object references, as described above. When used in a `corbaloc` reference, the format of the URL is `"corbaloc:rir:/key"`, where `key` is the name of the initial service. For example, the URL might be `"corbaloc:rir:/TradingService"` or `"corbaloc:rir:/NameService"`. Such references are, of course, limited to objects implementing the initial services available to an application.

The other variant of the corbaloc reference is similar to that of a simplified IOR. It includes one or more protocols and machine addresses or names, and an object key. The syntax is `corbaloc:<address>/<keystring>`, where `<address>` is `[<protocol>]:[<version>@]<host>[:<port>]`. Multiple `<address>` fields may appear, separated by commas. The default `<protocol>` is `"iiop"`. The default port is *2809*. The default host is `localhost`. Examples are:

> `"corbaloc::abc.xyz.com/filesys/usr/daffy"` (default protocol IIOP, host *abc.xyz.com*, default port *2809*, object key `"filesys/usr/daffy"`)
>
> `"corbaloc:iiop:192.168.1.99:49180/0xaa4362bc"` (protocol IIOP, host *192.168.1.99*, port *49180*, object key `"0xaa4362bc"`)

The syntax can be more complex than illustrated above, but these examples should

give a general idea what to expect in a `corbaloc` reference string.

Naming Service References

The Naming Service can be used in two ways to identify objects. One way uses context objects and name structures, the other way represents the same information in string form.

The Naming Service creates a tree-structured directory, similar to that of a traditional file system's directory. An object which approximates a directory or folder is a *naming context.* Each naming context can contain <*name component, object*> pairs, known as *bindings*. A name component consists of two strings: *id* and *kind*. The id is a simple name for the binding, while the type is meant to categorize the bound object. (The specification doesn't mandate how kind will be used. Some implementations use a simple, short string, analagous to the extension part of a file name, such as `"html"` in the name `"index.html"`. Others use a MIME type, and still others use the repository id of the object's interface definition.)

At each naming context, a path away from the root can be defined by a sequence of name components. This sequence is called a *name*, and it has a well-defined mapping to each programming language based in the mapping of IDL `sequence` and `struct` types. A <*naming context, name*> pair identifies a specific object. (Note, however, that a given object may have multiple names in any given context.)

The Naming Service also defines a procedure for combining these name components into a single string. This results in strings similar to the path names found in traditional file systems, strings such as `"abc/def/ghi.jkl"`. Such combined name components are known as *stringified names.*

Stringified names appear as the final component of a `corbaname` URL. The complete syntax is `corbaname:<corbaloc_object>[#<stringified_name>]`. The reference is similar to that of a `corbaloc` reference, except that the prefix is different (`"corbaname:"` instead of `"corbaloc:"`), and a stringified name optionally follows the base object reference. If there is a suffixed stringified name, then the base object must resolve to a naming context, and the stringified name is used to follow the path from there. Otherwise, it is interpreted the in the same way as a `corbaloc` URL.

Unlike basic object references and IORs, names and naming contexts aren't always available. They require that the CORBA or system implementor or application developer define a directory structure for any objects to be referenced in this fashion. If you have useful access to an object, you probably have the object's basic reference, from which you can create an IOR. On the other hand, objects don't always have names. However, names are useful when objects don't yet exist, because names can be presumed before the objects are instantiated. For example, a designer might by convention give a certain name to a commonly-needed service, allowing an

application to find that service without having to know the service's object reference or IOR in advance.

Indirect References

A final mechanism for identification of objects is the indirect reference. A URL of one of the forms `"file://..."`, `"http://..."`, or `"ftp://..."` indicates that the specified file is to be fetched, and its contents are to be interpreted as an object URL in one of the forms already discussed.

Availability and Use

Although the CORBA specifications describe all of the above mechanisms for identification of objects, not every implementation supports them all. Especially, the various forms of URL, except perhaps for `"ior:..."`, are not found in all CORBA ORBs.

This document has omitted discussion of the Trading Service, since that service is more akin to a search capability than it is to an identification mechanism. The Trading Service certainly can be used for object identification, however, and perhaps should be considered in circumstances where the other mechanisms don't satisfy the application requirements.

No form of reference will always describe what interfaces are supported by an object. For that reason, once an object identifier has been resolved, it may be appropriate to invoke `CORBA::Object::get_interface` (if supported) to determine the definition of the supported interface.

When multiple forms of representation are available, there is no simple rule telling which form is best in a situation. That must be determined by the application or designer. All other considerations being equal, when communicating among disparate referencing domains, IORs and their stringified equivalents tend to be most widely accepted, and they also contain other information which may be useful to communicating ORBs. Ultimately, of course, even when some other mechanism might be used to identify an object initially, ORBs will use IORs in their protocols among each other except when the parameters of an operation explicitly call for some other form of object identification.

To maximize portability, applications should avoid analysis or synthesis of opaque data, even when their internal formats are known or understood.