Bionic Buffalo Tech Note #25

Quick Start Guide to Writing CORBA Server Applications

last revised Saturday 2005.02.26

Introduction

This is a quick introduction to writing CORBA server applications. In this context, a CORBA server application is defined as some application implementing on one or more CORBA objects. There is no conflict if the same application is both a client and a server. In fact, most servers also tend to be clients at the same time. Because of this, the reader should first become familiar with the separate companion document, *Tech Note #20: Quick Start Guide to Writing CORBA Client Applications*. The material from *Tech Note* #20 will be assumed as a prerequisite to the material in this *Tech Note*.

This *Tech Note* summarizes the steps, and gives some examples, but does not provide extensive details. However, the reader is referred to specific additional documents more information.

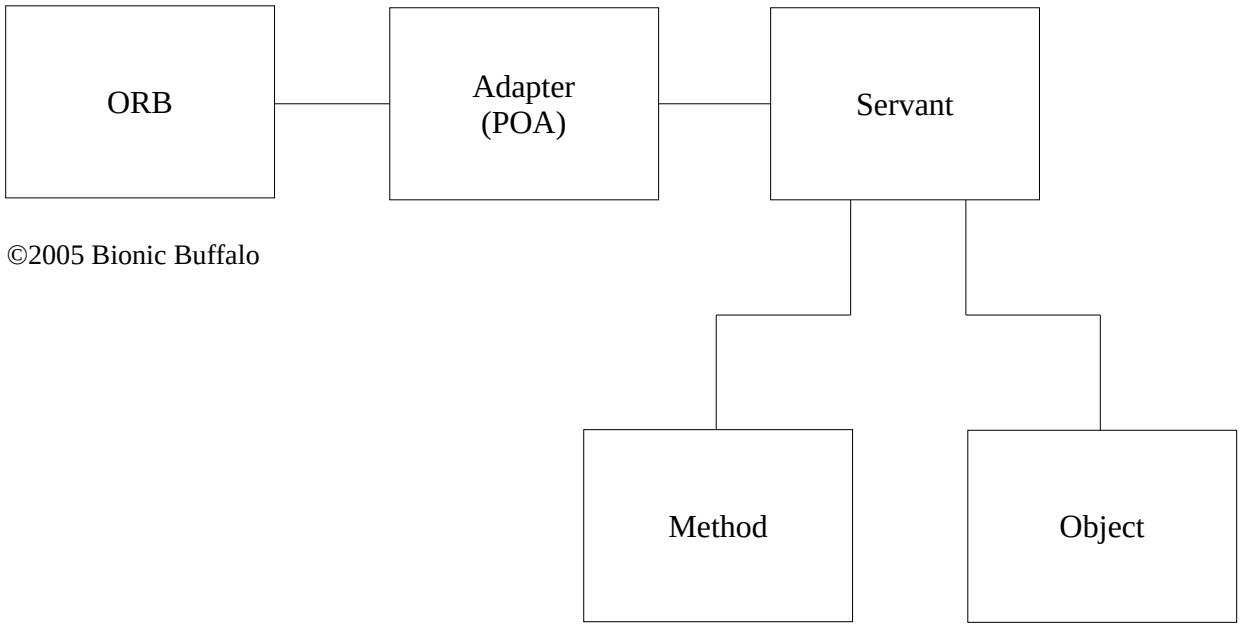
This *Tech Note* illustrates the use of Bionic Buffalo tools. The procedures using other tools are generally the same, but some non-Bionic Buffalo compilers will not generate all of the programs, data structures and other information (such as static repository definitions, allocation routines for aliases, and so on) that are generated by Bionic Buffalo's `france` IDL compiler. Workarounds must be made in such cases by the developer; this *Tech Note* attempts to convey the background and the basic process so that the necessary procedures for such workarounds will be reasonably obvious.

Although this *Tech Note* discusses programming languages in general, it uses examples in the C language. However, the same principles are applicable to applications written in other programming languages.

Fundamental Principles and Terminology

(In what follows, the architecture is simplified for two reasons: the number of possibilities make the full architecture too complex to deal with quickly, and for most purposes the simplification is all that is needed. Possible complications will be mentioned later.)

The implementation of CORBA server applications requires the creation of the following basic structure:



An adapter is also called a *POA* (for "*Portable Object Adapter*"), since it has the interface `PortableServer::POA`.

In this model, there may be more than one `POA` for each `ORB`, more than one servant for each `POA`. The methods are considered part of the servant, and the servant, using the methods, implements the requests made upon the objects. There may be more than one method for a servant (depending on the number of operations and attributes to be implemented), and a servant may implement requests on more than one object.

Together, these components act as follows:

- The ORB mediates requests made by clients upon the objects. It selects the correct `POA` for the target object, and passes the request to the POA. The details of the interaction and interfaces between the ORB and POA are not covered in the specification, but the application programmer need not be concerned with that, anyway.

- The POA accepts requests from the ORB, and passes the request to the appropriate servant.

- The servant directs the request to the methods. There is a method for each operation. A method is basically a subroutine, function, or procedure which implements an operation.

The `POA` is normally provided by the `ORB` developer. In the beginning (when the *ORB* initializes), there is one `POA`, the *root POA*. Every `POA`, except the root `POA`, is created from some other `POA`, giving rise to a tree of `POA`s anchored in the root `POA`. Each `POA` has a character string name, so a kind of "path name" from the root `POA` to any given `POA` is possible. Each `POA` has one of four states: *active*, *inactive*, *holding*, or *discarding*. These states determine how requests are processed.

When an POA is created from another POA (using the POA::create operation), the only arguments (that is, configuration options) are the name, the manager, and a set of policies. These are the only things that distinguish one POA from another. The only purpose of the manager is to select the state of the POA, and there are only seven policies. (Each policy may have more than one value, but not all combinations are permissible.) POAs are not so complicated as they might appear at first glance.

The Seven Adapter Policies: What They Do, and How To Use Them

IdAssignmentPolicy

Every request from a client to the ORB specifies an object reference, which must be unique to, and within, the ORB. The ORB must maintain a mapping of object references to POAs, so it will know to which POA an object reference belongs. Object references are opaque to the POAs, however. When the ORB communicates with a POA, it uses an opaque sequence of octets, the object id (PortableServer::ObjectId). Each object id must be unique within the POA. Who assigns the object id is defined by the one of the seven POA policies, the IdAssignmentPolicy. If the POA itself assigns the object id, then the IdAssignmentPolicy is SYSTEM_ID. If the servant assigns the object id, then the IdAssignmentPolicy is USER_ID. An object id can be any sequence of octets, so it can be very useful to assign those octets systematically.

When you are designing a server application, if you want to keep track of your servant's objects in some systematic fashion, consider the USER_ID policy. If you don't want to be bothered (maybe you only have one object per servant, or maybe you just have a simple lookup table), then consider the SYSTEM_ID policy.

For example, with the USER_ID IdAssignmentPolicy, you can use as the object id a memory address, file name, or database key to allow easy lookup of an object's state variable or value.

RequestProcessingPolicy and ServantRetentionPolicy

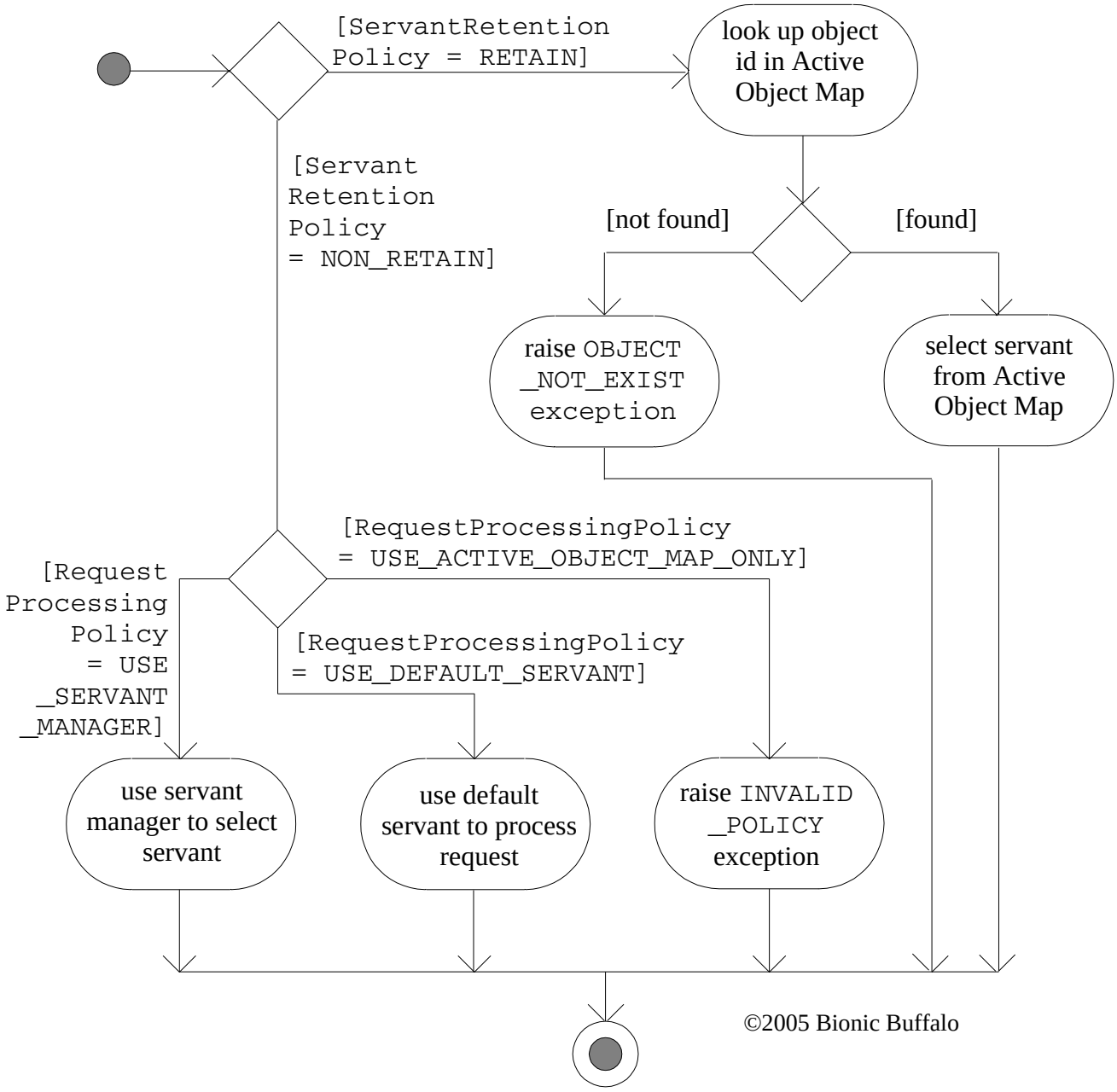
When a request is given to the POA by the ORB, the POA must decide which of its servants to use to handle the request. There are three different tools which may be used by the POA to make its decision:

- The POA may keep a table of object ids, associating each object id with a specific servant. This table is known as the *Active Object Map*. Whether or not the POA keeps an Active Object Map is determined by the ServantRetentionPolicy: if the policy value is RETAIN, then it keeps one; if the policy value is NON_RETAIN, then no Active Object Map is maintained.

- A default servant may be registered with the POA, for use in case there is no Active Object Map, or if the object is not found in the Active Object Map.

- The application may provide a Servant Manager to make the decision regarding which servant to use. As with the default servant, this method is used if there is no Active Object Map, or if the object is not found in the Active Object Map.

The flow of control depends on the three possible values of the `RequestProcessingPolicy`, and is shown in the following diagram:



An application running for an extended period of time serving a large number of objects can build a very large Active Object Map, so this is a consideration when choosing the

`ServantRetentionPolicy`. On the other hand, if the number of objects is limited, and the same servant will always be used for any given object, the use of an Active Object Map can be efficient. **Consider the potential size of the Active Object Map when choosing the `ServantRetentionPolicy`.**

Choosing an appropriate `RequestProcessingPolicy` usually is a straightforward process, depending on how the design selects the servant to be used for an incoming request.

`LifespanPolicy`

Although the `LifespanPolicy` applies to `POA`s, the effect of the policy applies to object references created by a given `POA`. The impact of the policy is felt when the `ORB` receives a request for an object, but the object's `POA` which created it no longer exists. In those situations:

- If the reference was created by a `POA` whose `LifespanPolicy` is `TRANSIENT`, then the `ORB` fails the request.
- If the reference was created by a `POA` whose `LifespanPolicy` is `PERSISTENT`, then the `ORB` attempts to activate the missing `POA`. The `ORB` tries to create a missing `POA` by calling the parent `POA`'s *adapter activator*. This requires that the missing `POA`'s parent have an adapter activator registered with it.

Although an object reference might be created by a `TRANSIENT` `POA`, it still might be "persistent" in a more general sense. The distinction meant by the `LifespanPolicy` is merely whether or not the `ORB` will attempt to activate a missing `POA`.

Use the `PERSISTENT` `LifespanPolicy` when, for whatever reason, you want to have `POA`s which are not always active, and you want the `ORB` to activate them automatically when requests arrive. This may include situations where the `POA` is used infrequently, and is not kept active between invocations.

`IdUniquenessPolicy`

The `IdUniquenessPolicy` specifies whether there is only one, or possibly more than one, object per servant. If the value is `UNIQUE_ID`, then that each of that `POA`'s servants is used for exactly one object id. If the value is `MULTIPLE_ID`, then that `POA`'s servants support one or more object ids.

Since, without an Active Object Map (that is, without the `RETAIN` policy), servants are selected by default or by a servant manager, the value of `IdUniquenessPolicy` is irrelevant without an Active Object Map. **Use the `UNIQUE_ID` policy value when appropriate to allow a (possible) optimization in the use of the Active Object Map by the `POA`.**

`ImplicitActivationPolicy`

If an application invokes either `servant_to_id()` or `servant_to_reference()` to create an object id or object reference for use with a given servant, and if the specified servant is not active, having `ImplicitActivationPolicy` set to `IMPLICIT_ACTIVATION` will cause the `POA` to activate the specified servant. This only applies when the `RETAIN` and `SYSTEM_ID` policy values are in effect.

Use of `IMPLICIT_ACTIVATION` is mostly a convenience for application developers, since, regardless of the value of the `ImplicitActivationPolicy`, activation can be done explicitly by the application.

`ThreadPolicy`

The `ThreadPolicy` specifies the thread model and reentrancy of the servants. It only matters in multithreaded environments.

- If the value is `ORB_CTRL_MODEL`, the servants are presumed to be thread aware and reentrant. A `POA` may simultaneously call the same servant from different threads.
- If the value is `SINGLE_THREAD_MODEL`, the `POA` will call the servants from only one thread at a time. This is for servants which are non-reentrant or not thread aware.
- The `MAIN_THREAD_MODEL` value is used for some environments which distinguish a special "main thread" from other threads. (Examples include DOS and some RTOS environments.) With this policy value, not only are all calls made from a single thread, but that thread always will be the distinguished, "main" thread.

`ThreadPolicy` is simply an indication whether the servants are reentrant and thread aware, and whether they must executed only on a special main or distinguished thread in certain environments.

Note that `ThreadPolicy` applies not only to servants, but also to servant managers, if any.

Creating and Advertising Object References

From an application standpoint, object references are created by `POA`s. Object references may be created by any application which has an object reference for the `POA` (and maybe for the servant) which will handle the object's requests. Although there are some other ways in which objects might come into existence (implicit activation, and so on), this *Tech Note* will limit discussion to the four basic `POA` operations that most applications will invoke.

- `create_reference()` and `create_reference_with_id()` will create and return new object references. The first is used when the `SYSTEM_ID` policy value applies, and the other is used when the `USER_ID` policy value applies. These operations do not associate the new references with specific servants.
- `activate_object()` and `activate_object_with_id()` will create new objects

associated with specific servants, but will not return the references. To obtain the references for the new objects, one of the `POA` operations `servant_to_reference()` or `id_to_reference()` must be invoked.

These operations need not be done in a servant. An application can them invoke outside of any servant if it has the `POA` and maybe the servant references, and a servant for one object can create objects for another servant. To get the `POA`,

- From within a servant, the application can use `ORB::resolve_initial_references()` to find *"POACurrent"*, which will be an object of type `PortableServer::Current`. That object's `get_POA()` operation will return the `POA` for the current request.
- From outside a servant, the application can use `ORB::resolve_initial_references()` to obtain the Root POA, then use `POA::find_POA()` to find the specific `POA` needed. (This works from within a servant, too.)

In any case, finding the servant (if needed) is language mapping and application dependent. Also, it is not necessarily obvious from the CORBA documentation how to obtain a reference for the `ORB` from within a servant: the operation is the same as for any other application, namely, `CORBA_ORB_init()`.

Once the servant or any other application has an object reference, it must somehow make that reference available to clients. The common mechanisms are summarized in more detail in *Tech Note #20*, but are again listed here with some mention of server (rather than client) considerations:

- *Name service.* The client will need the name of the object, which can be sent as a string using non-CORBA mechanisms.
- *Trading service.* The client will need the attributes of the object, which can be sent using non-CORBA mechanisms.
- *Object identifier.* This method is used for fundamental objects such as the interface repository. This method is not generally applicable to application objects.
- *Object URL.* The simplest way to create an object URL is to invoke the `ORB`'s `object_to_string()` operation, which will return a string which can be understood by another `ORB`'s `string_to_object()` operation. The interoperable string can be transmitted among programs by any ordinary mechanism. Object URLs may also be constructed manually by the application, when the type of URL specifies a commonly understood and well specified technique such as the name service.
  .
- *External Mechanisms.* Details of this mechanism, if used, will depend on the implementation.

Unless the object reference is in string form (as a URL, perhaps created by `ORB::object_to_string`), it is not generally portable among different `ORB`s. Non-string references may be passed among `ORB`s only as arguments to operations mediated by the `ORB`s involved.

Basic Steps

Creating a server application consists of the following basic steps:

1. Design the interfaces.
2. Design the servant and the adapter configuration.
3. Make some decisions about the application environment and implementation.
4. Compile the interface IDL to create the header files, stubs, and other necessary programs and data structures.
5. Design and code the application.
6. Arrange to convey critical information to client application developers.
7. Link and run the application.

The steps common to almost all programming (design review, testing, documentation, and so on) are omitted from this list unless there are special considerations to be brought up for writing CORBA client applications.

Design the Interfaces

The end product of the first step includes, as a minimum, the IDL description of the interfaces and all necessary data definitions (structures and other types) necessary for operation invocation.

A description in IDL is incomplete. Not only was IDL not intended to describe activities, implementations, and other things beyond the scope of object interfaces, it also has limitations in its primary function. There is no way to describe ranges of simple types in IDL, for example. (It can be said that a type is `short`, for instance, but what are the minimum and maximum or specific allowable values?) Do not forget to include all information in the interface description, not merely those things which can be expressed in IDL. At the very least consider a UML description of the system.

Design the Servant and the Adapter Configuration

The first part of servant design usually is to decide the lifetime of an object, and how object state information (if any) is to be kept. When reasonable values for the seven POA policies have become obvious, then this phase of development probably is nearly complete.

Determine Application Environment and Implementation

The remarks in *Tech Note #20* regarding this step are appropriate here.

Compile the Interface IDL

Tech Note #20 lists various items generated by the IDL compiler, possibly relevant to a client application. In addition to those, the IDL compiler generates additional code useful in developing server applications.

Methods for Each Operation

For each operation and attribute, the IDL compiler generates (as required by the specifications) prototypes for the methods. Bionic Buffalo's `france` compiler, in addition, will (if asked) generate a sample method for each operation. The operation of the sample method program is determined by the value of the compile time symbol (macro) `VUT_METHOD_SELECTOR`.

- If `VUT_METHOD_SELECTOR` is undefined, or if it is set to `VUT_METHOD_NO_IMPLEMENT`, then the sample method program will generate a `NO_IMPLEMENT` exception when called.
- If `VUT_METHOD_SELECTOR` is set to `VUT_METHOD_STATELESS`, then the sample method will look up the object reference and object id of the target object.
- If `VUT_METHOD_SELECTOR` is set to `VUT_METHOD_MEMORY_STATE`, then the sample method will look up the object reference and object id of the target object, and will compute a pointer to a state structure in memory. The object id is presumed to include within its structure a pointer to the object's state structure.
- If `VUT_METHOD_SELECTOR` is set to `VUT_METHOD_INDEXED_STATE`, then the sample method will perform a database lookup of the object's state, then it will perform a database update of the object's state. Between the lookup and the update, meaningful work is presumed to be done. Generic database routines are used; non-functioning versions are provided in the libraries, but a developer may substitute functioning routines that use file i/o, a database, or some other mechanism. The object id is presumed to include within its structure the database key. The state structure is marshaled into an opaque sequence of bytes when written to the database.

In any of these cases, the developer may consider the generated sample method to be an illustration, or he or she may use the sample method as the starting point for a real application. These sample methods do not do anything other than housekeeping; a developer must add code to perform some meaningful work from a client's point of view.

In the two options above which deal with state lookup and update (either in memory, or in a database), the IDL compiler looks for a structure called `ObjectState` defined in the interface. If found, that structure is used as the state value structure, otherwise the compiler provides a trivial definition for an `ObjectState` structure. If an interface inherits from another interface, the derived interface's state structure will include the state structure from the base interface.

Servants for Each Interface

For each interface, the IDL compiler generate (as required by the specification) prototypes and initialization and finalization routines for the interface's servant. In addition, Bionic Buffalo's `france` compiler will (if asked) generate a sample servant for each interface. The sample servant employs the sample methods created for each of the operations and attributes of the interface. Thus, by default, the sample servant will raise `NO_IMPLEMENT` for each of its operations and attributes, or the behaviour may be modified by setting the `VUT_METHOD_SELECTOR` macro at compile time.

Installation Routines for Each Interface

For each interface described in the IDL, Bionic Buffalo's `france` IDL compiler will (if asked) generate an installation routine which will install the sample servant and methods, creating (if necessary) a suitably configured `POA`. It will also generate an uninstall routine, to remove them.

Object Creation Routines for Each Interface

For each interface described in the IDL, Bionic Buffalo's `france` IDL compiler will (if asked) generate object creation and destruction routines to create and destroy objects with the specified interface. The object creation and destruction routines will allocate or release state structures as required, including the necessary database operations.

Design and Code the Application

A CORBA server application usually has at least five basic parts:

- An installation routine to set up the `POA`, servants, and related objects (such as servant managers)
- A startup routine to create one or more initial objects, whose references will be passed or advertised to potential clients
- The servants and related objects themselves
- A shutdown routine to destroy objects created by the startup routine
- A uninstallation routine to remove the `POA`, servants, and related objects

Installation

The installation routine must install and configure any `POA`s, servants, servant managers, and other services or objects required by the server application.

1. Use `CORBA_ORB_init()`, if necessary, to obtain a reference for the `ORB` which will own the objects.
2. Use `ORB::resolve_initial_references()` to obtain a reference for the Root `POA`.
3. Iteratively use `POA::find_POA()` to traverse the `POA` tree. Use `POA::create_POA()` to create any needed `POA`s.
4. Create and install any required `AdapterActivator`, `ServantActivator`, and

`ServantLocator` objects at appropriate places in the above process. Note that most language mappings treat these the same as ordinary server objects, so they, in turn, might require other objects to be installed first.

Often, the installation routine is combined with the startup routine, whose description follows.

Startup

Once the servants are in place, the application must create some initial objects for the clients to use. As explained above, there are four basic operations (`create_reference()`, `create_reference_with_id()`, `activate_servant()`, and `activate_servant_with_id()`) to accomplish this. References for these initial objects must then be made available to clients, using one of the above listed mechanisms.

Servants

Servants are, for the most part, coded as are other applications, except for the servant's ability to acquire and to use the `POACurrent` object.

The `POACurrent` object is of type `PortableServer::Current`, and is obtained from the `ORB` by use of `ORB::resolve_initial_references()`. The `POACurrent` object is available only to servants while handling a request, and provides information about the current request to the servant. The `POACurrent` object has four operations to provide that information:

- `get_POA()` returns the `POA` which called the servant (a servant might be used by more than one `POA`)
- `get_servant()` returns the servant (a method might belong to more than one servant)
- `get_object_id()` returns the object id of the target object (that is, the object whose operation is invoked by the request)
- `get_reference()` returns a reference for the target object

Unless `IdUniquenessPolicy = UNIQUE_ID`, the servant methods might be called for different objects, and the `get_object_id()` and `get_reference()` operations are the only way for the servant to determine which object was the target of the call. The servant must maintain a mapping between object id or reference, and the various objects supported by the servant. There are two obvious ways to do this: either a lookup table with object id or reference as an entry, or by encoding some distinguishing information into the object id. Encoding distinguishing information into the object id requires that `IdAssignmentPolicy = USER_ID`. Convenient information to encode includes pointers to data structures in memory, peripheral i/o addresses, names of files, and database keys. (Sometimes it is a good idea to encode a serial number or similar identifier into the object id, in addition to an address or key, to distinguish among different objects which might exist at different times but which coincidentally might have the same address or key. An object reference on a client might persist after an object has been destroyed on the server.)

Character, wide character, string, and wide string arguments passed from client to server (or returned from server to client) may undergo translation by the `ORB` to accommodate differences in character

sets between the two environments.

Shutdown and Uninstallation

Shutting down and uninstalling the servants and objects should not be neglected operations. In addition to releasing resources for other applications, proper termination routines encourage good programming practice: the activity of shutting down often reveals errors which might not show up otherwise until after an application is released to the field.

Arrange to Convey Critical Information to Client Application Developers

Any service, application, or other mechanism used to convey essential information (interfaces, references, *et caetera*) to clients should be started, activated, configured, or otherwise engaged.

Link and Run the Application

The final step is to link and run the application. There are no major differences between client and server applications in this regard. The reader is referred to the equivalent section in *Tech Note #20*.

Bibliography

Please consult the Bibliography of *Tech Note #20*.
