Bionic Buffalo Tech Note #20

Quick Start Guide to Writing CORBA Client Applications

last revised Saturday 2005.02.26

©2005 Bionic Buffalo Corporation. All Rights Reserved.
Tatanka, *TOAD*, and *Bionic Buffalo* are trademarks of Bionic Buffalo Corporation

Introduction

This is a quick introduction to writing CORBA client applications. In this context, a CORBA client application is defined as some application invoking operations on one or more CORBA server objects. The CORBA objects need not be implemented in the same application or even in the same machine. There is no conflict if the same application is both a client and a server. However, writing server applications is covered in a separate companion document, *Tech Note #25: Quick Start Guide to Writing CORBA Server Applications*.

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.

Server Objects, Client Objects, and Pseudo Objects

As explained above, the scope of this *Tech Note* is limited to applications which invoke operations on one or more server objects. Of necessity, however, the application must create certain objects on the client to enable the operation invocations.

The CORBA specification makes a distinction between *pseudo objects* and other, more ordinary, objects. Pseudo objects include the `ORB` itself, as well as more esoteric objects such as those implementing `CORBA::Request`. In spite of this distinction, it is important that an application uses

(with very few exceptions) the same mechanisms to call an object operation, whether the target object is local, remote, ordinary, pseudo, or otherwise classified. To keep things simple, in this *Tech Note*, no distinction is made among the various categories of objects. They all are referred to simply, as "objects".

The main thing to keep in mind is there are certain things you cannot do with pseudo objects. For example, you cannot pass a reference to the `ORB` to another machine, because the `ORB` is considered local, and an application on the other machine cannot necessarily invoke `ORB` operations on your machine. These forbidden things, while not always intuitively obvious, are not likely to get a beginner into too much trouble anyway.

That being said, it is necessary for the purposes of this discussion to distinguish one kind of ordinary object, which is the server object whose operations are to be invoked by the client application being written. Note that "server object" is not a CORBA term of art; it is used here as a term of convenience. A server object, as this *Tech Note* uses the term, is not necessarily on a separate server, and is not necessarily even in a separate application.

Another term which will be used is "target object", a term which is used in the CORBA specifications. A target object is the object upon which a operation is invoked. When an application has selected a specific server object and invokes an operation on that server object, then that server object also becomes a target object.

Basic Steps

Creating a client application consists of the following basic steps:

1. Determine certain fundamental information about the server objects and how they are used.

2. Make some decisions about the application environment and implementation.

3. Compile the interface IDL to create the header files, stubs, and other necessary programs and data structures.

4. Design and code the application.

5. Link the application with the necessary compiler output and support libraries.

6. 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.

Determine Fundamental Information About Server Objects

The key information a client application developer needs regarding server objects include the interface and data definitions as defined by the objects' IDL (interface definition language) description, and how the object is made available, advertised, or to be discovered.

The IDL description is well specified by the CORBA specifications themselves. It embodies a complete description of how the operations are invoked, along with the necessary data definitions (structures, enumerations, and so on), and also describes the exceptions that might occur upon invocation. IDL is beyond the scope of this *Tech Note*. The reader is referred to the CORBA specification for more information. However, in short, IDL is the language that serves as input to an IDL compiler, on which more, below.

In order to invoke an operation on an object, an application needs a reference to that object. The IDL does not define how the client can find the server object and obtain an object reference. The common ways are:

- *Name service.* An object reference is made available on a name service. A name service provides a tree structured directory of objects, similar to that a typical file system directory. To use the object, the client application must have (a) an object reference for the name service, and (b) the name of the object.

- *Trading service.* An object is advertised on a trading service. A trading service is a kind of database of objects, which are indexed by attributes. To use the object, the client must have (a) an object reference for the trading service, and (b) attributes of the object so it can be found using the trading service. Using a trading service may be used when any of several equivalent objects may be acceptable. For example, if there are several servers giving weather information, a trading service may allow an application to find some object that provides the necessary information, but maybe not any specific object.

- *Object identifier*. Certain objects are given character string identifiers, known as object ids. The string can be used to obtain an object reference. This method is used for fundamental objects such as the interface repository. This method is not generally applicable to application objects. To use the object, the client must know the string identifier.

  (Note: CORBA uses the term “object identifier” in two distinct ways. As used above, “object identifier” refers to strings of the type `CORBA::ORB::ObjectId`. The other kind of object identifier refers to opaque sequences of octets, and is defined as `PortableServer::ObjectId`. These two kinds of object identifier are not interchangeable, although in some implementations there may be a relationship between them.)

- *Object URL*. This method is analogous to the use of URLs on web browsers. The `ORB` provides an operation, `string_to_object`, which will return an object reference when given the object URL string. There are various kinds of object URLs. To be resolved, some depend on the existence of other services, such as the name service or trading service. (The `string_to_object` operation takes care of the resolution process, but the service may still be required.) Not all `ORB`s or environments support the resolution of all types of object URL. Sometimes, an object URL string can be constructed on-the-fly by an application according to some rule set. To use the object, the

client application requires the object URL string.

- *External Mechanisms.* Some environments use external or non-CORBA mechanisms. These are dependent on the way the programming and system environments were constructed. For example, the DSM-CC specification for interactive multimedia uses a secondary protocol (the User-Network protocol) to, among other things, provide an initial object reference to the application.

Determine Application Environment and Implementation

Before compiling the IDL, some aspects of the application environment and implementation must be determined. This is because there may be options for the IDL compiler which depend on the environment and implementation, which, in turn, may affect the application code itself.

Most of these considerations are relatively minor, but Bionic Buffalo makes a distinction between two application environments: a standard environment defined by the specification, and a lightweight implementation (the `sudan` product) for very small embedded systems. The required library API for the `sudan` environment is a subset of the library API for minimal CORBA. When the `france` IDL compiler generates code for `sudan`, less code is generated, and some symbols expected by an application in a standard CORBA environment may not resolve. Code generated for `sudan`, however, will run in a standard CORBA environment (at least with Bionic Buffalo's software). Since it is useful when developing application to know when a feature will not be available, it is a good idea to generate for code for `sudan` if you are planning to host it in a very small embedded environment in the future. That way, errors about unresolved and undefined symbols will be encountered early, possibly saving rework later.

Compile the Interface IDL

The IDL compiler generates header files, subroutines, and data structures from the IDL, mapping the IDL into the programming language of choice. The application developer will use this generated code to invoke operations on the server objects.

The code generated by the IDL compiler depends on the programming language. For most common programming languages, there is a formal specification from the Object Management Group defining the mapping to that programming language. For a few common languages (such as Perl) there is no formal specification, and the mapping is defined by the implementor or by informal convention. Even where a formal specification exists, it might be slightly incomplete or ambiguous, so a developer concerned about portability ought to investigate these mapping issues.

Because of the many languages for which there are mappings, it is beyond the scope of this *Tech Note* to consider them all. What follows is a general description of the code generated for the C programming language. The C language serves as a good example because the language itself is simple, and is not object oriented, so the object constructs defined by the mapping specification are a close reflection of the thinking of the OMG regarding what a fundamental CORBA object should be, do, and have. In the case of some object oriented languages, it is sometimes more difficult to see the parts that are CORBA, and the parts that are intrinsic to the language. For example, a Perl object is

inadequate to represent a CORBA object, because, among other things, CORBA objects can be created dynamically, and an application can use a reference to an object which does not yet exist; one cannot do such a thing with Perl objects. Similarly, Perl objects are created at the server's run time and are destroyed when the server process terminates, while CORBA objects may exist before the server is started and may survive the destruction of the server.

For the C mapping, the IDL compiler generates code and data in C source code form, depending on the compiler run time options selected. Such generated information includes:

- For each IDL definition, a static repository definition constant is created.
- For each IDL definition of a type for which a `TypeCode` is defined, a `TypeCode` constant is created.
- For each structure, union, sequence, and exception, an allocation functions function is created. For each sequence, a buffer allocation function is created.
- For each constant, a macro is defined for the constant's value.
- For each `enum` value, a macro is defined as the numeric quantity associated with that value.
- For each data type or interface, a suitable `typedef` is generated
- For each `alias`, the same or equivalent allocation functions, and macros are generated as for the base type, except that the name is based on that of the `alias` name.
- For each operation defined on an interface, a stub subroutine is generated. An application calling the operation on an object calls the stub, which in turn creates a request which is mediated by the ORB. The object reference is the first argument passed to the stub. The other arguments defined by the IDL follow the object reference.
- For each attribute defined on an interface, one or two stub routines are generated, similar to the stub routines generated for operations. A stub for the `_get` operation is generated, and, for attributes which are not `readonly`, a stub for the `_set` operation is also generated.
- In addition to the above items useful to a client, various other items related to the server mapping are created. The generated code relevant to servers only is not considered here.

The generated headers and programs are to be used by applications requiring access to, or use of, the entities defined by the source IDL. In addition to necessary common libraries, the generated code is all that is necessary to write CORBA compliant programs in the chosen programming language.

Design and Code the Application

A CORBA client application usually performs the following activities:

1. Set up the programming environment
2. Acquire references for initial system object or objects
3. Handle possible exceptions
4. Acquire a reference for the server object or objects
5. Do the primary activity of the application
6. Release resources and tear down the programming environment

Set up the programming environment. What constitutes this activity is mostly dependent upon the programming language, but may also involve proprietary considerations. For example, in the C programming language, it is necessary to acquire a pointer to a partially-opaque data structure `CORBA_Environment`, which is used as an argument in subsequent calls to object operations; the OMG C mapping specification doesn't say how this pointer is acquired, leaving it to proprietary mechanisms. Some environments require setting up initial service objects (such as the repository or naming service) in advance of ORB initialization; in other environments, initial service objects are created or set up automatically. For specific set up procedures, it is necessary to consult documentation for the environment you are using.

When programming in C using Bionic Buffalo tools, the `CORBA_Environment` pointer is acquired by the routine `tbt_get_environment()`, prototyped as

```
int tbt_get_environment ( CORBA_Environment ** env ) ;
```

Each thread must have its own copy of the `CORBA_Environment` structure, and the pointers from the above call may not be shared among threads. If a thread makes the above call more than once, it will get the same pointer back each time.

Acquire references for initial system object or objects. As a minimum, every non-trivial application will need a reference to the ORB. The CORBA specification says this will be done using a procedure or subroutine `CORBA_ORB_init()`, but the mapping of that subroutine to each programming language is different. For C, the prototype is

```
typedef char * CORBA_ORBid ;
extern CORBA_ORB CORBA_ORB_init
        ( int * argc,
        char ** argv,
        CORBA_ORBid orb_identifier,
        CORBA_Environment * env ) ;
```

For simple cases, the argument list (`argc` and `argv`) may be left empty in Bionic Buffalo's environment, and the `orb_identifier` may be left empty to select the default `ORB`. (Multiple `ORB`s may be used together by one program; the orb_identifier is used to distinguish among them.) After this subroutine has been invoked successfully, then the application will have an object reference for the `ORB`.

Whether or not additional system object references will be required depends on the application itself, and upon the mechanism which will be used to locate the server objects. For instance, if the server objects are to be located using the name service, then an object reference for the name service object will be needed. In C, this will be done using a call similar to

```
name_service = ( CosNaming_NamingContext )
        CORBA_ORB_resolve_initial_references
        ( orb, ( CORBA_ObjectId ) "NameService", env ) ;
```

The `orb` object reference was acquired in the previous `CORBA_ORB_init` operation, and the `env` pointer was acquired earlier as described. The `resolve_initial_references()` operation is available in all language mappings, but of course its form depends on the rules of the mapping.

Handle possible exceptions. At this point, it is appropriate to consider what happens when a call to an object operation fails. In such cases, an exception will be raised, and the application must recognize that this has happened and respond accordingly.

The mechanisms for exception handling vary from one programming language to the next. Some languages have built in or conventional procedures for using exceptions, and these may be used also for CORBA exceptions. Other languages, such as C, have no such conventions, or the existing mechanisms are inadequate, and the language mapping for each such language will define how that language handles CORBA exceptions.

CORBA exceptions have names and values. The name is defined by the IDL, and (depending on the programming language) may appear to the application programmer as a string containing the repository id of the exception definition. The value is an arbitrary data structure, which may be empty. Exception values may be used to return almost any kind of information back to the application. The data structure is defined in the IDL: although it may contain arbitrary information, it will always be the same for a given exception name.

In the C mapping, a member `_major` of the `CORBA_Environment` structure specifies whether or not an exception has been raised.

> If `env -> _major == CORBA_NO_EXCEPTION`, then there has been no exception.
>
> If `env -> _major == CORBA_SYSTEM_EXCEPTION`, then a system exception has been raised. The system exceptions and their corresponding data structures are defined in the CORBA specification. An application must always be prepared to handle any system exception.
>
> If `env -> _major == CORBA_USER_EXCEPTION`, then a user exception has been raised. User exceptions are defined in the IDL for the operation being called. An application should be prepared for any user exceptions defined in the IDL.

Once a C application has recognized that an exception has been raised, it can use the function `CORBA_exception_id()` to learn the name of the exception. Then one of the two routines

`CORBA_exception_value()` or `CORBA_exception_as_any()` may be used to learn the value of the exception.

For more details on exception handling, consult the language mapping specification for the programming language you are using, or documentation from your `ORB` supplier.

Acquire a reference for the server object or objects. There are, as was explained earlier, various ways to acquire references to server objects. The choice of mechanism will depend on the way the system was designed, and on how the server has chosen to advertise its objects or otherwise to make them available.

The complexity of the various ways to acquire references varies greatly. The mechanism may consist of a single subroutine call (such as `string_to_object()`), or it may involve an iterative procedure or sequence of such procedures. Because of the variety and sometimes complexity, the different mechanisms cannot all be described in this short *Tech Note*. The application programmer is referred to the material listed in the Bibliography below.

Do the primary activity of the application. Once there is in hand a reference to a server object, the application may commence invocation of operations or attribute calls on that object.

Using the stubs generated by the compiler, invocations of operations on objects look the same to the application program as ordinary local subroutine, procedure, or function calls. The stub routine serves as a local proxy for the object, whether that object is in the same application or whether it is remote and accessible only via the network.

For CORBA programming, however, there are some rules and guidelines, however, which will help to avoid some problems and difficulties:

1. Check for exceptions wherever they might occur.

2. Do not look inside or assume knowledge of things which are supposed to be opaque. For example, even if you know how to decompose or duplicate an object reference, do not do it: use the system provided operations instead.

3. In language mappings (such as C) which include memory and other resource allocation and release functions, use the specific, provided functions instead of generic functions. For example, to allocate a structure `X` in C, use `X__alloc()` instead of `malloc()`.

4. In threaded programs, do not share thread-specific system data (such as the `CORBA_Environment` pointer in C) among threads.

5. In threaded (and sometimes other) programs, be aware that calls to object operations may block, even when the object is local and no network activity is required.

6. Do not bypass the language mapping rules. For example, a close reading of the C mapping specification shows that, while IDL `long` is mapped to `CORBA_long`, there is no requirement that

`CORBA_long` map to C `long` (indeed, it may map on some processors to `int`, which may not be the same as `long`.). With typed languages, use the CORBA type for casting and coercion, not the equivalent machine type.

Release resources and tear down the programming environment. This step is often ignored when it should be taken. CORBA programming often has various subtle side effects, especially with regard to resource management, and ignoring the termination and release operations - even when they do not appear relevant to a terminating program - can have a detrimental effect on the performance of other applications.

When an application no longer needs the CORBA environment, resources, and objects, it should return or release those. A terminating application also should call `ORB::shutdown()` and `ORB::destroy()`, to release resources for other programs. (Some `ORB`s maintain resources which are shared among processes, and there may not be immediate or automatic release when the application process terminates.)

Link and Run the Application

The final step is to link and run the application.

Applications built and run using Bionic Buffalo tools will need to link with one or more libraries, depending on the implementation. These will include:

- `tibet` is the fundamental CORBA library. It contains stubs, resource management functions, TypeCodes, repository definitions, and other items defined by the CORBA specification. Effectively, it is the result of compiling the IDL from the specification, along with some “glue” and fundamental definitions to make it all work together and to fill in gaps in the specification. It does not include the implementation of any objects such as the ORB, repository, or other system interfaces.

- `yemen` is a common support library, which includes implementations of some pseudo object interfaces (typecodes, policies, and so on), but does not include implementations of any `ORB` or `ORB` specific objects such as `POA`s.

- The library for the specific `ORB` or `ORB`s to be used by the application. These might be one or more of `sudan`, `gibraltar`, `egypt`, `taiwan`, `mexico`, or `jordan`.

- For any services which are to be provided from within the application process, the relevant service library must be included. For example, if your process will provide a name service (either to itself or to external clients), then `greece` must be linked. On the other hand, if the application only uses the service available in an external process, then (for the most important CORBA services) the stubs and other necessary code will be found in the `tibet` library.

- Various support libraries may also be required. For example, the `nigeria` library is commonly required because it is used to handle character set mediation and translation among various `ORB`s.

When an application is run, the distinction between *application* `ORB`s and *shared-process* `ORB`s must be understood.

> An *application* `ORB` is linked to an application, the same as any other library. The `ORB` runs in the same process as the main application, although it may create separate threads for some activities. Bionic Buffalo application `ORB`s include `sudan`, `gibraltar`, `egypt`, and `taiwan`.
>
> A *shared-process* `ORB` runs in its own process, and serves the requirements of one or more separate and distinct application processes. It uses either networking or inter-process communication to communicate with a library linked to the application processes. Bionic Buffalo shared process `ORB`s include `mexico` and `jordan`.

Since shared process `ORB`s are separate programs, they must be started separately when running an application which uses them. This implies an extra step. Usually, a shared process `ORB` can be started using a subroutine delivered with the `ORB`, so (for testing, at least) the shared process `ORB` can be started by the application itself.

Beyond the Basics

The procedures described above may be used for simple client applications. In addition, CORBA has other facilities which may be of interest to programmers, and which can be used to enhance the above procedures. Some of these facilities include:

- The interface repository is a database of interface, operation, attribute, data structure, and other definitions, which can be described in IDL. Given an object reference, it is possible to obtain the definitions of its operations, attributes, and all related data structures.

- Although an ORB might be usable in real-time applications without them, the real time extensions give an application finer and more direct control over such things as the thread priorities used internally by an ORB.

- The security service can provide authentication, encryption, non repudiation, and other related capabilities. These capabilities can operate transparently, or applications can exercise control over their operation.

- The messaging APIs allow control of various aspects of the way messages are passed among client and server ORBs; these enable control of quality of service, asynchronous messaging (with polling or callback), and specification of routing mechanisms

- Fault tolerance allows replication of servers and automatic or managed recovery from failure conditions. Client applications specify some of the operational parameters in such an environment.

- Transaction processing allows clients to group multiple operation invocations, possibly backing out

of a sequence of operations and “undoing” operations already completed.

The above facilities go beyond simple service additions to CORBA, in that they may directly affect the way other CORBA application programming is done. In other words, they do not simply add new APIs to an implementation, but they affect and alter the behaviour of existing APIs.

Bibliography

Almost all of the procedures described in this *Tech Note* are governed by stable, mature OMG specifications. Nevertheless, minor updates to those specifications are made from time to time. Therefore, while the document numbers in the list below may not be for the most current versions, any newer versions probably will not be greatly different from the versions listed.

- OMG, *Common Object Request Broker Architecture: Core Specification*, document formal/04-03-01. This is the primary document for CORBA 3.x. In addition to the fundamental mechanisms, it also describes the following facilities mentioned above: the interface repository, object URLs, object identifiers, messaging, and fault tolerance.

- OMG, *Naming Service*, document formal/04-10-03.

- OMG, *Trading Object Service*, document formal/00-06-27.

- OMG, *Object Transaction Service*, document formal/03-09-02.

For any of the various language mapping specifications, consult the OMG web site, at *http://www.omg.org*. There is no official specification for the Perl mapping; for that, refer to

- Bionic Buffalo Corporation, *Tibet OMG IDL to Perl Mapping Specification*, 18 January 2005. (Changed greatly from earlier draft versions.)

This *Tech Note* may be reproduced and distributed (including by means of the Internet) without payment of fees or without notification to Bionic Buffalo, as long as it is not changed, altered, or edited in any way. Any distribution or copy must include the entire *Tech Note*, with the original title, copyright notice, and this paragraph. For available *Tech Notes*, please see the Bionic Buffalo web site at *http://www.tatanka.com/doc/technote/index.htm*, or e-mail *query@tatanka.com*. PGP/GnuPG key fingerprint: `a836 e7b0 24ad 3259 7c38 b384 8804 5520 2c74 1e5a`. Most Bionic Buffalo *Tech Notes* are available in both HTML and PDF form.
