The omniORB2 User's Guide

Sai-Lai Lo
(*email: sll@orl.co.uk*)
Olivetti & Oracle Research Laboratory

Note: this document is very incomplete at the moment! More chapters will be added to document the ORB's APIs and its internals.

13 Mar, 1997

Contents

Chapter 1

Introduction

OmniORB2 is an Object Request Broker (ORB) that implements the 2.0 specification of the Common Object Request Broker Architecture (CORBA) [OMG96a]. This user guide tells you how to use omniORB2 to develop CORBA applications. It assumes a basic understanding of CORBA.

In this chapter, we give an overview of the main features of omniORB2 and what you need to do to setup your environment to run omniORB2.

1.1 Features

1.1.1 CORBA 2 compliant

OmniORB2 implements the Internet Inter-ORB Protocol (IIOP). This protocol provides omniORB2 the means of achieving interoperability with the ORBs implemented by other vendors. In fact, this is the native protocol used by omniORB2 for the communication amongst its objects residing in different address spaces. Moreover, the IDL to C++ language mapping provided by omniORB2 conforms to the latest revision of the CORBA specification.

1.1.2 Multithreading

OmniORB2 is fully multithreaded. To achieve low IIOP call overhead, unnecessary call-multiplexing is eliminated. At any time, there is at most one call in-flight in each communication channel between two address spaces. To do so without limiting the level of concurrency, new channels connecting the two address spaces are created on demand and cached when there are more concurrent calls in progress. Each channel is served by a dedicated thread. This arrangement provides maximal concurrency and eliminates any thread switching in either of the address spaces to process a call. Furthermore, to maximize the throughput in processing large call arguments, large data elements are sent as soon as they are processed while the other arguments are being marshalled.

1.1.3 Portability

At ORL, the ability to target a single source tree to multiple platforms is very important. This is difficult to achieve if the IDL to C++ mapping for these platforms are different. We avoid this problem by making sure that only one IDL to C++ mapping is used. We run several flavours of Unices, Windows NT, Windows 95 and our in-house developed systems for our own hardware. OmniORB2 have been ported to all these platforms. **The IDL to C++ mapping for these targets are all the same**.

OmniORB2 uses real C++ exceptions and nested classes. We stay with the CORBA specification's standard mapping as much as possible and do not use the alternative mappings for C++ dialects. The only exception is the mapping of **modules** to **C++ classes** instead of **namespaces**.

OmniORB2 relies on the native thread libraries to provide the multithreading capability. A small class library (omnithread [Richardson96a]) is used to encapsulated the (possibly different) APIs of the native thread libraries. In the application code, it is recommended but not mandatory to use this class library for thread management. It should be easy to port omnithread to any platform that either supports the POSIX thread standard or has a thread package that supports similar capabilities.

1.1.4 Missing features

OmniORB2 is not (yet) a complete implementation of the CORBA core. The following is a list of the missing features.

- The Typcode and the Any type is not supported. Support for these types will be added shortly.
- The BOA only support the persistent server activation policy. Other dynamic activation and deactivation policies are not supported.
- The Dynamic Invocation Interface is not supported.
- The Dynamic Skeleton Interface is not supported.
- OmniORB2 does not has its own Interface Repository.

These features may be implemented in the short to medium term. It is best to check out the latest status on the omniORB2 home page (`http://www.orl.co.uk/omniORB/omniORB.html`).

1.2 Setting Up Your Environment

After you have unpacked the distribution, read all the README files at the top level of the directory tree. These files contain essential information on installing, building and using omniORB2 on the supported platforms.

The following is a checklist of what you have to do:

1. Setup the naming service. An implementation of the COS Naming Service, called omniNames, is provided in this distribution. If you want to use the service, you have to start it up first. Consult the document "The OMNI Naming Service" for details. When omniNames starts up, it writes the stringified object reference for its root context on standard error. This is needed by the omniORB2 runtime. See below for how to configure the runtime. You can also use other naming service implementations provided that you can obtain the stringified object reference for its root context.

2. Configure the omniORB2 runtime. At startup the omniORB runtime tries to read the configuration file `omniORB.cfg` to obtain the object reference to the root context of the Naming Service. This object reference is returned by the call `resolve_initial_references("NameService")`.

   (a) On Unix platforms, omniORB2 looks for the environment variable `OMNIORB_CONFIG`. If this variable is defined, it contains the pathname of the omniORB2 configuration file. If the variable is not set, omniORB2 will use the compiled-in pathname (`/etc/omniORB.cfg`) to locate the file.

   (b) On Win32 platforms (Windows NT, Windows '95), omniORB2 first checks the environment variable (`OMNIORB_CONFIG`) to obtain the pathname of the configuration file. If this is not set, it then attempts to obtain configuration data in the system registry. It searches for the data under the key `HKEY_LOCAL_MACHINE\SOFTWARE\ORL\omniORB\2.0`

   The format of the entry is the word `NAMESERVICE` followed by space and the stringified IOR all on one line. For example:

```
NAMESERVICE IOR:000000000000002049444c3a436f734e616d696e672f4e616d696e674
36f6e746578743a312e300000000001000000000000002c000100000000000012776962626c
652e776f62626c652e636f6d0004d20000000c3371b8c09528a18700000001
```

   Aternatively, the stringified IOR can be placed in the system registry on Win32 platforms, in the (string) value `NAMESERVICE`, under the key `HKEY_LOCAL_MACHINE\SOFTWARE\ORL\omniORB\2.0`.

3. Compiler flags. You should be able to build the whole distribution using the makefiles provided. The makefiles are configured to supply a set of preprocessor defines that are necessary to compile omniORB2 programs. The preprocessor defines are needed because the same set of header files are used for all platforms. If you are to incorporate omniORB2 into your own development environment, these are the necessary preprocessor defines:

Platform	Preprocessor Defines
Sun Solaris 2.x	`-D__sunos__ -D__sparc__ -D__OSVERSION__=5 -DSVR4 -DUsePthread -D_REENTRANT -D__OMNIORB2__`
Digital Unix 3.2	`-D__osf1__ -D__alpha__ -D__OSVERSION__=3 -D_REENTRANT -D__OMNIORB2__`
x86 Linux 2.0 with linuxthreads 0.5	`-D__linux__ -D__i86__ -D__OSVERSION__=2 -D_REENTRANT`
Windows NT	`-D__NT__ -MD -GX -D_X86_ -D__OMNIORB2__`

The makefiles are good examples on how omniORB2 programs are built. Please study them before you try to incorporate omniORB2 into other development environments.

Chapter 2

The Basics

In this chapter, we go through three examples to illustrate the practical steps to use omniORB2. By going through the source code of each example, the essential concepts and APIs are introduced. If you have no previous experience with using CORBA, you should study this chapter in detail. There are pointers to other essential documents you should be familiar with.

If you have experience with using other ORBs, you should still go through this chapter because it provides important information about the features and APIs that are necessarily omniORB2 specific. For instance, the object implementation skeleton is covered in section 2.4.2.

2.1 The Echo Object Example

Our example is an object which has only one method. The method simply echos the argument string. We have to:

1. define the object interface in IDL;
2. use the IDL compiler to generate the stub code[1];
3. provide the object implementation;
4. write the client code.

The source code of this example is included in the last section of this chapter. The files are also included in the distribution. The README file in the example directory contains instructions on how to build and run the programs.

2.2 Specifying the Echo interface in IDL

We define an object interface, called Echo, as follows:

[1]The stub code is the C++ code that provides the object mapping as defined in the CORBA 2.0 specification.

```
interface Echo {
    string echoString(in string mesg);
};
```

If you are new to IDL, you can learn about its syntax in Chapter 3 of the CORBA specification 2.0 [OMG96a].

For the moment, you only need to know that the interface consists of a single operation, echoString, which takes a string as an argument and returns a copy of the same string.

The interface is written in a file, called `echo.idl`.

For simplicity, the interface is defined in the global IDL namespace. This practice should be avoided for the sake of object reusuability. If every CORBA developer defines their interfaces in the global IDL namespace, there is a danger of name clashes between two independently defined interfaces. Therefore, it is better to qualify your interfaces by defining them inside `module` names. Of course, this does not eliminate the chance of a name clash unless some form of naming convention is agreed globally. Nevertheless, a well-chosen module name can help a lot.

2.3 Generating the C++ stubs

From the IDL file, we use the IDL compiler to produce the C++ mapping of the interface. The IDL compiler for omniORB2 is called `omniidl2`. Given the IDL file, `omniidl2` produces two stub files: a C++ header file and a C++ source file. For example, from the file `echo.idl`, the following files are produced:

- `echo.hh`
- `echoSK.cc`

2.4 A Quick Tour of the C++ stubs

The C++ stubs conform to the mapping defined in the CORBA 2.0 specification (chapter 16-18). It is important to understand the mapping before you start writing any serious CORBA applications.

Before going any further, it is worth knowing what the mapping looks like.

2.4.1 Object Reference

The use of an object interface denotes an object reference. For the example interface Echo, the C++ mapping for its object reference is `Echo_ptr`. The type is defined in echo.hh. The relevant section of the code is reproduced below:

```
class Echo;
typedef Echo* Echo_ptr;
```

```
class Echo : public virtual omniObject, public virtual CORBA::Object {
public:

  virtual char *  echoString ( const char *  mesg ) = 0;
  static Echo_ptr _nil();
  static Echo_ptr _duplicate(Echo_ptr);
  static Echo_ptr _narrow(CORBA::Object_ptr);

  ... // methods generated for internal use
};
```

In a compliant application, the operations defined in an object interface should **only** be invoked via an object reference. This is done by using arrow ("→") on an object reference. For example, the call to the operation `echoString` would be written as `obj→echoString(mesg)`.

It should be noted that the concrete type of an object reference is opaque, i.e. you must not make any assumption about how an object reference is implemented. In our example, even though `Echo_ptr` is implemented as a pointer to the class `Echo`, it should not be used as a C++ pointer, i.e. conversion to void*, arithmetic operations, and relational operations, including test for equality using **operation==** must not be performed on the type.

In addition to `echoString`, the mapping also defines three static member functions in the class Echo: `_nil`, `_duplicate`, and `_narrow`. Note that these are operations on an object reference.

The `_nil` function returns a nil object reference of the Echo interface. The following call is guaranteed to return TRUE:

```
CORBA::Boolean true_result = CORBA::is_nil(Echo::_nil());
```

Remember, `CORBA::is_nil()` is the only compliant way to check if an object reference is nil. You should not use the equality operator==.

The `_duplicate` function returns a new object reference of the Echo interface. The new object reference can be used interchangeably with the old object reference to perform an operation on the same object.

All CORBA objects inherit from the generic object `CORBA::Object`. `CORBA::Object_ptr` is the object reference for `CORBA::Object`. Any object reference is therefore conceptually inherited from `CORBA::Object_ptr`. In other words, an object reference such as `Echo_ptr` can be used in places where a `CORBA::Object_ptr` is expected.

The `_narrow` function takes an argument of the type `CORBA::Object_ptr` and returns a new object reference of the Echo interface. If the actual (runtime) type of the argument object reference can be widened to `Echo_ptr`, `_narrow` will return a valid object reference. Otherwise it will return a nil object reference.

To indicate that an object reference will no longer be accessed, you can call the `CORBA::release` operation. Its signature is as follows:

```
class CORBA {
   static void release(CORBA::Object_ptr obj);
   ... // other methods
};
```

You should not use an object reference once you have called `CORBA::release`. This is because the associated resources may have been deallocated. Notice that we are referring to the resources associated with the object reference and **not the object implementation**. Here is a concrete example, if the implementation of an object resides in a different address space, then a call to `CORBA::release` will only caused the resources associated with the object reference in the current address space to be deallocated. The object implementation in the other address space is unaffected.

As described above, the equality operator== should not be used on object references. To test if two object references are equivalent, the member function `_is_equivalent` of the generic object `CORBA::Object` can be used. Here is an example of its usage:

```
Echo_ptr A;
...            // initialized A to a valid object reference
Echo_ptr B = A;
CORBA::Boolean true_result = A->_is_equivalent(B);
// Note: the above call is guaranteed to be TRUE
```

You have now been introduced to most of the operations that can be invoked via `Echo_ptr`. The generic object `CORBA::Object` provides a few more operations and all of them can be invoked via `Echo_ptr`. These operations deal mainly with CORBA's dynamic interfaces. You do not have to understand them in order to use the C++ mapping provided via the stubs. For details, please read the CORBA specification [OMG96a] chapter 17.

Since object references must be released explicitly, their usage is prone to error and can lead to memory leakage. The mapping defines the **object reference variable** type to make life easier. In our example, the variable type `Echo_var` is defined[2].

The `Echo_var` is more convenient to use because it will automatically release its object reference when it is deallocated or when assigned a new object reference. For many operations, mixing data of type `Echo_var` and `Echo_ptr` is possible without any explicit operations or castings [3]. For instance, the operation `echoString` can be called using the arrow ("$\rightarrow$") on a `Echo_var`, as one can do with a `Echo_ptr`.

The usage of `Echo_var` is illustrated below:

```
Echo_var a;
```

[2] In omniORB2, all object reference variable types are instantiated from the template type _CORBA_ObjRef_Var.

[3] However, the implementation of the type conversion operator() between Echo_var and Echo_ptr varies slightly among different C++ compilers, you may need to do an explicit casting when the compiler complains about the conversion being ambiguous.

```
Echo_ptr p = ...     // somehow obtain an object reference

a = p;               // a assumes ownership of p, must not use p anymore

Echo_var b = a;      // implicit _duplicate

p = ...              // somehow obtain another object reference

a = Echo::_duplicate(p);     // release old object reference
                             // a now holds a copy of p.
```

2.4.2 Object Implementation

Unlike the client side of an object, i.e. the use of object references, the CORBA specification 2.0 deliberately leave many of the necessary functionalities to implement an object unspecified. As a consequence, it is very unlikely the implementation code of an object on top of two different ORBs can be identical. However, most of the code are expected to be portable. In particular, the body of an operation implementation can normally be ported with no or little modification.

OmniORB2 uses C++ inheritance to provide the skeleton code for object implementation. For each object interface, a skeleton class is generated. In our example, the skeleton class `_sk_Echo` is generated for the Echo IDL interface. An object implementation can be written by creating an implementation class that derives from the skeleton class.

The skeleton class `_sk_Echo` is defined in `echo.hh`. The relevant section of the code is reproduced below.

```
class _sk_Echo :  public virtual Echo {
public:
  _sk_Echo(const omniORB::objectKey& k);
  virtual char *  echoString ( const char *  mesg ) = 0;
  Echo_ptr        _this();
  void            _obj_is_ready(BOA_ptr);
  void            _dispose();
  BOA_ptr         _boa();
  omniORB::objectKey _key();
  ... // methods generated for internal use
};
```

The code fragment shows the only member functions that can be used in the object implementation code. Other member functions are generated for internal use only. **Unless specified otherwise, the description below is omniORB2 specific.** The functions are:

echoString it is through this abstract function that an implementation class provides the implementation of the `echoString` operation. Notice that its signature is the same as the `echoString` function that can be invoked via the `Echo_ptr` object reference. **The signature of this function is specified by the CORBA specfication**.

_this this function returns an object reference for the target object. The returned value must be deallocated via `CORBA::release`. See 2.7 for an example of how this function is used.

_obj_is_ready this function tells the Basic Object Adaptor[4] (BOA) that the object is ready to serve. Until this function is called, the BOA would not serve any incoming calls to this object. See 2.7 for an example of how this function is used.

_dispose this function tells the BOA to dispose of the object. The BOA will stop serving incoming calls of this object and remove any resources associated with it. See 2.7 for an example of how this function is used.

_boa this function returns a reference to the BOA that serves this object.

_key this function returns the key that the ORB used to identify this object. The type `omniORB::objectKey` is opaque to application code. The function `omniORB::keyToOctetSequence` can be used to convert the key to a sequence of octets.

2.5 Writing the object implementation

You define an implementation class to provide the object implementation. There is little constraint on how you design your implementation class except that it has to inherit from the stubs' skeleton class and to implement all the abstract functions defined in the skeleton class. Each of these abstract functions corresponds to an operation of the interface. They are hooks for the ORB to perform upcalls to your implementation.

Here is a simple implementation of the Echo object.

```
class Echo_i : public virtual _sk_Echo {
public:
  Echo_i() {}
  virtual ~Echo_i() {}
  virtual char * echoString(const char *mesg);
};

char *
Echo_i::echoString(const char *mesg) {
  char *p = CORBA::string_dup(mesg);
  return p;
}
```

There are three points to note here:

Storage Responsibilities A string, which is used as an IN argument and the return value of `echoString`, is a variable size data type. Other examples of variable size data types include sequences, type "any", etc. For these data types, you must be clear about who's responsibility to allocate and release their associated storage. As a rule of thumb, the client (or the caller to the implementation functions) owns the storage of all IN arguments, the object implementation (or the

[4]The interface of a BOA is described in chapter 8 of the CORBA specification.

callee) must copy the data if it wants to retain a copy. For OUT arguments and return values, the object implementation allocates the storage and passes the ownership to the client. The client must release the storage when the variables will no longer be used. For details, please refer to Table 24-27 of the CORBA specification.

Multi-threading As omniORB2 is fully multithreaded, multiple threads may perform the same upcall to your implementation concurrently. It is up to your implementation to synchronise the threads' accesses to shared data. In our simple example, we have no shared data to protect so no thread synchronisation is necessary.

Instantiation You must not instantiate an implementation as automatic variables. Instead, you should always instantiate an implementation using the new operator, i.e. its storage is allocated on the heap. The reason behind this restriction will become clear in section 2.7.

2.6 Writing the client

Here is an example of how a `Echo_ptr` object reference is used.

```
void
hello(CORBA::Object_ptr obj)
{
  Echo_var e = Echo::_narrow(obj);                // line 1

  if (CORBA::is_nil(e)) {                         // line 2
    cerr << "hello: cannot invoke on a nil object reference.\n" << endl;
    return;
  }

  CORBA::String_var src = (const char*) "Hello!";  // line 3
  CORBA::String_var dest;                          // line 4

  dest = e->echoString(src);                       // line 5

  cerr << "I said,\"" << src << "\"."
       << " The Object said,\"" << dest <<"\"" << endl;
}
```

Briefly, the function `hello` accepts a generic object reference. The object reference (`obj`) is narrowed to `Echo_ptr`. If the object reference returned by `Echo::_narrow` is not nil, the operation `echoString` is invoked. Finally, both the argument to and the return value of `echoString` are printed to cerr.

The example also illustrates how T_var types are used. As it was explained in the previous section, T_var types take care of storage allocation and release automatically when variables of the type are assigned to or when the variables go out of scope.

In line 1, the variable `e` takes over the storage responsibility of the object reference returned by `Echo::_narrow`. The object reference is released by the destructor of `e`. It is called automatically when the function returns. Line 2 and 5 shows how a

`Echo_var` variable is used. As said earlier, `Echo_var` type can be used interchangeably with `Echo_ptr` type.

The argument and the return value of `echoString` are stored in `CORBA::String_var` variable `src` and `dest` respectively. The strings managed by the variables are deallocated by the destructor of `CORBA::String_var`. It is called automatically when the function returns. Line 5 shows how `CORBA::String_var` variables are used. They can be used in place of a string (for which the mapping is `char*`)[5]. As used in line 3, assigning a constant string (`const char*`) to a `CORBA::String_var` causes the string to be copied. On the otherhand, assigning a `char*` to a `CORBA::String_var`, as used in line 5, causes the latter to assume the ownership of the string[6].

Under the C++ mapping, T_var types are provided for all the non-basic data types. It is obvious that one should use automatic variables whenever possible both to avoid memory leak and to maximize performance. However, when one has to allocate data items on the heap, it is a good practice to use the T_var types to manage the heap storage.

2.7 Example 1 - Colocated Client and Implementation

Having introduced the client and the object implementation, we can now describe how to link up the two via the ORB. In this section, we describe an example in which both the client and the object implementation are in the same address space. In the next two sections, we shall describe the case where the two are in different address spaces.

The code for this example is reproduced below:

```
int
main(int argc, char **argv)
{
  CORBA::ORB_ptr orb = CORBA::ORB_init(argc,argv,"omniORB2");   // line 1
  CORBA::BOA_ptr boa = orb->BOA_init(argc,argv,"omniORB2_BOA"); // line 2

  Echo_i *myobj = new Echo_i();                                 // line 3
  myobj->_obj_is_ready(boa);                                    // line 4

  boa->impl_is_ready(0,1);                                      // line 5

  Echo_ptr myobjRef = myobj->_this();                           // line 6
  hello(myobjRef);                                              // line 7
  CORBA::release(myobjRef);                                     // line 8

  myobj->_dispose();                                            // line 9
  return 0;
}
```

[5] A conversion operator() of CORBA::String_var converts a CORBA::String_var to a char*.

[6] Please refer to the CORBA specification 16.7 for the details of the String_var mapping. Other T_var types are also covered in chapter 16.

The example illustrates several important interactions among the ORB, the object implementation and the client. Here are the details:

2.7.1 ORB/BOA initialisation

line 1 The ORB is initialised by calling the `CORBA::ORB_init` function. The function uses the 3rd argument to determine which ORB should be returned. To use omniORB2, this argument must either be "omniORB2" or NULL. If it is NULL, there must be an argument, -ORBid "omniORB2", in `argv`. Like any command-line arguments understood by the ORB, it will be removed from argv when `CORBA::ORB_init` returns. Therefore, an application is not required to handle any command-line arguments it does not understand. If the ORB identifier is not "omniORB2", the initialisation will fail and a nil `ORB_ptr` will be returned. If supplied, omniORB2 also reads the configuration file `omniORB.cfg`. Among other things, the file provides a list of initial object references. One example of these object references is the naming service. Its use will be discussed in section 2.9.1. If any error occurs during the processing of the configuration file, the system exception CORBA::INITIALIZE is raised.

line 2 The BOA is initialised by calling the ORB's `BOA_init`. The 3rd argument must either be "omniORB2_BOA" or NULL. If it is NULL, then `argv` must contain an argument, -BOAid "omniORB2_BOA". If the BOA identifier is not "omniORB2_BOA", the initialisation will fail and a nil BOA_ptr will be returned. Like `ORB_init`, any command-line arguments understood by `BOA_init` will be removed from `argv`.

2.7.2 Object initialisation

line 3 An instance of the Echo object is initialised using the `new` operator.

line 4 The object's `_obj_is_ready` is called. This function informs the BOA that this object is ready to serve. Until this function is called, the BOA will not accept any invocation on the object and will not perform any upcall to the object.

line 5 The BOA's `impl_is_ready` is called. This function tells the BOA the implementation is ready. After this call, the BOA will accept IIOP requests from other address spaces. There are 2 points to note here:

1. `boa`→`impl_is_ready` can be called any time after `BOA_init` is called (line 2). In other words, object instances can be initialised and advertised to the BOA before or after this function is called.
2. The 2nd argument[7] to `impl_is_ready` tells the ORB whether this call should be non-blocking. The default value of this argument is FALSE(0) and the call will block indefinitely within the ORB. If there are more things the main thread should do after it calls `impl_is_ready`, as it is the case in this example, the non-blocking option (TRUE=1) should be specified. Whether the main thread blocks in this call or not, the ORB is not affected

[7] The 1st argument is a pointer to the implementation definition and is always ignored by omniORB2.

because its functions are provided by other threads spawned internally. Notice that the signature of `impl_is_ready` in the CORBA specification does not have the 2nd argument[8]. Therefore, calling `impl_is_ready` with the non-blocking option is omniORB2 specific.

2.7.3 Client invocation

line 6 The object reference is obtained from the implementation by calling `_this`. Like any object reference, the return value of _this must be released by `CORBA::release` when it is no longer needed.

line 7 Call `hello` with this object reference. The argument is widened implicitly to the generic object reference `CORBA::Object_ptr`.

line 8 Release the object reference.

One of the important characteristic of an object reference is that it is completely location transparent. A client can invoke on the object using its object reference without any need to know whether the object is colocated in the same address space or resided in a different address space.

In case of colocated client and object implementation, omniORB2 is able to short-circuit the client calls to direct calls on the implementation methods. The cost of an invocation is reduced to that of a function call. This optimisation is applicable **not only** to object references returned by the _this function but to any object references that are passed around within the same address space or received from other address spaces via IIOP calls.

2.7.4 Object disposal

line 9 To dispose of an object implementation and release all the resources associated with it, the `_dispose` function is called. In fact, this is the **only** clean way to get rid of an object implementation. Even though the object is created using the new operator in the application code, the application should never call the delete operator on the object directly.

Once an application calls `_dispose` on an object implementation, the pointer to the object should not be used any more. At the time the `_dispose` call is made, there may be other threads invoking on the object, omniORB2 ensures that all these calls are completed before removing the object from its internal tables and releasing the resources associated with it. The storage associated with the object is released by omniORB2 using the delete operator. This is why all object implementation should be initialised using the new operator (section 2.5).

[8] The CORBA specification does not specify when `impl_is_ready` should return. Many ORB vendors choose to implement `impl_is_ready` as blocking until a certain time-out value is exceeded. In a single threaded implementation this is necessary to give the ORB the time to serve incoming requests.

The disposal of an object implementation by omniORB2 may also be deferred when **colocated** clients continue to hold on to copies of the object's reference[9]. This behavior is to prevent the short-circuited calls from the clients to fail unpredictably.

To summarise, an application can make no assumption as to when the object is disposed by omniORB2 after the `_dispose` call returns. If it is necessary to have better control on when to stop serving incoming requests, the work should be done by the object implementation itself, such as by keeping track of the current serving state.

2.8 Example 2 - Different Address Spaces

In this example, the client and the object implementation reside in two different address spaces. The code of this example is almost the same as the previous example. The only difference is the extra work need to be done to pass the object reference from the object implementation to the client.

The simplest (and quite primitive) way to pass an object reference between two address spaces is to produce a stringified version of the object reference and to pass this string to the client as a command-line argument. The string is then converted by the client into a proper object reference. This method is used in this example. In the next example, we shall introduce a better way of passing the object reference using the COS Naming Service.

2.8.1 Object Implementation: Generating a Stringified Object Reference

The `main` function of the object implementation side is reproduced below. The full listing (`eg2_impl.cc`) can be found at the end of this chapter.

```
int
main(int argc, char **argv)
{
  CORBA::ORB_ptr orb = CORBA::ORB_init(argc,argv,"omniORB2");
  CORBA::BOA_ptr boa = orb->BOA_init(argc,argv,"omniORB2_BOA");

  Echo_i *myobj = new Echo_i();
  myobj->_obj_is_ready(boa);

  {
    Echo_var myobjRef = myobj->_this();
    CORBA::String_var p;

    p = orb->object_to_string(myobjRef);            //line 1

    cerr << "'" << (char*)p << "'" << endl;
  }

  boa->impl_is_ready();    // block here indefinitely
```

[9] Object references held by clients in other address spaces will not prevent the object implementation from being disposed of. If these clients invoke on the object after it is disposed, the system exception INV_OBJREF is raised.

```
                          // See the explanation in example 1
  return 0;
}
```

The stringified object reference is obtained by calling the ORB's function `_object_to_string` (line 1). This is a sequence starting with the signature "IOR:" and followed by a hexadecimal string. All CORBA 2.0 compliant ORBs are able to convert the string into its internal representation of a so-called Interoperable Object Reference (IOR). The IOR contains the location information and a key to uniquely identify the object implementation in its own address space[10]. From the IOR, an object reference can be constructed.

2.8.2 Client: Using a Stringified Object Reference

The stringified object reference is passed to the client as a command-line argument. The client uses the ORB's function `string_to_object` to convert the string into a generic object reference (`CORBA::Object_ptr`). The relevant section of the code is reproduced below. The full listing (`eg2_clt.cc`) can be found at the end of this chapter.

```
try {
  CORBA::Object_var obj = orb->string_to_object(argv[1]);
  hello(obj);
}
catch(CORBA::COMM_FAILURE& ex) {
  ... // code to handle communication failure
}
```

2.8.3 Catching System Exceptions

When omniORB2 detects an error condition, it may raise a system exception. The CORBA specification defines a series of exceptions covering most of the error conditions that an ORB may encounter. The client may choose to catch these exceptions and recover from the error condition[11]. For instance, the code fragment, shown in section 2.8.2, catches the system exception COMM_FAILURE which indicates that communication with the object implementation in another address space has failed.

All system exceptions inherit from the class `CORBA::SystemException`. With compilers that support RTTI[12][13], a single catch `CORBA::SystemException` will catch all the different system exceptions thrown by omniORB2.

When omniORB2 detects an internal inconsistency that is most likely to be caused by a bug in the runtime, it raises the exception `omniORB::fatalException`. When this exception is raised, it is not sensible to proceed with any operation that involves

[10] Notice that the object key is not globally unique across address spaces.

[11] If a system exception is not caught, the C++ runtime will call the `terminate` function. This function is defaulted to abort the whole process and on some system will cause a core file to be produced.

[12] Run Time Type Identification

[13] A noticeable exception is the GNU C++ compiler (version 2.7.2). It doesn't support RTTI unless the compilation flag -frtti is specified. The omniORB2 runtime is not compiled with the -frtti flag. It is said that RTTI will be properly supported in the upcoming version 2.8.

the ORB's runtime. It is best to exit the program immediately. The exception structure carries by `omniORB::fatalException` contains the exact location (the file name and the line number) where the exception is raised. You are strongly encourage to file a bug report and point out the location.

2.8.4 Lifetime of an Object Implementation

It may be obvious but it has to stated that an object implementation exists only for the duration of the process's lifetime. When the same program is run again, a different instance of the object implementation is created. More significantly, **the IOR, and hence the object reference, of this instance is different from that of the previous run**.

For instance, if you look at the stringified object reference produced by the program `eg2_impl` in different runs, they are all different. The implication is that you cannot store away the stringified object reference and expect to be able to use it again later when the original program run has terminated.

For system services and other applications, it may be desirable to have "persistent" object implementations. The objects are "persistent" in the sense that they can be contacted using the same IOR when they are instantiated in different program runs. To provide this functionality, omniORB2 needs to be provided with two pieces of information: the (network) location and the object key. The details of how this can be done will be described in the later part of this manual.

Alternatively, an indirection from textual pathnames to object references can be used. Applications can register object implementations at runtime to a naming service and bind them to fixed pathnames. Clients can bind to the object implementations at runtime by asking the naming service to resolve the pathnames to the object references. CORBA defines a naming service, which is a component of the Common Object Services (COS) [OMG96b], that can be used for this purpose. The next section describes an example of how to use the COS Naming Service.

2.9 Example 3 - Using the COS Naming Service

In this example, the object implementation uses the COS Naming Service [OMG96b] to pass on the object reference to the client. This method is by-far more practical than using stringified object references. The full listing of the object implementation (`eg3_impl.cc`) and the client (`eg3_clt.cc`) can be found at the end of this chapter.

The object reference is bound to the pathname "**test**/**Echo**"[14]. The pathname consists of the context **test** and the object name **Echo**. Both the context and the object name has an attribute **kind**. This attribute is a string that is intended to be used to describe the name in a syntax-independent way. The naming service does not interpret, assign, or manage these values. However both the name and the kind attribute must match

[14] A pathname, or in the Naming Service's terminology- a *compound name*, is a sequence of textual names. Each name component except the last one is bound to a naming context. A naming context is analogous to a directory in a filing system, it can contain names of object references or other naming contexts. The last name component is bound to an object reference. Note: '/' is purely a notation to separate two components in the pathname. It does not appear in the *compound name* that is registered with the Naming Service.

for a name lookup to succeed. In this example, the **kind** values for **test** and **Echo** are chosen to be "my_context" and "Object" respectively. This is an arbitrary choice for there is no standardised set of kind values.

2.9.1 Obtaining the Root Context Object Reference

The initial contact with the Naming Service can be established via what we called the **root** context. The object reference to the root context is provided by the ORB and can be obtained by calling `resolve_initial_references`. The following code fragment shows how it is used:

```
CORBA::ORB_ptr orb = CORBA::ORB_init(argc,argv,"omniORB2");

CORBA::Object_var initServ;
initServ = orb->resolve_initial_references("NameService");

CosNaming::NamingContext_var rootContext;
rootContext = CosNaming::NamingContext::_narrow(initServ);
```

Remember, omniORB2 constructs its internal list of initial references at initialisation time using the information provided in the configuration file `omniORB.cfg`. If this file is not present, the internal list will be empty and `resolve_initial_references` will raise a CORBA::ORB::InvalidName exception.

2.9.2 The Naming Service Interface

It is beyond the scope of this chapter to describe in detail the Naming Service interface. You should consult the CORBAservices specification [OMG96b] (chapter 3). The code listed in `eg3_impl.cc` and `eg3_clt.cc` are good examples of how the service can be used. Please spend time to study the examples carefully.

2.10 Source Listing

2.10.1 echo_i.cc

```
// echo_i.cc - This source code demonstrates an implmentation of the
//             object interface Echo. It is part of the three examples
//             used in Chapter 2 "The Basics" of the omniORB2 user guide.
//
#include <string.h>
#include "echo.hh"

class Echo_i : public virtual _sk_Echo {
public:
  Echo_i() {}
  virtual ~Echo_i() {}
  virtual char * echoString(const char *mesg);
};

char *
Echo_i::echoString(const char *mesg) {
  char *p = CORBA::string_dup(mesg);
  return p;
}
```

2.10.2 greeting.cc

```
// greeting.cc - This source code demonstrates the use of an object
//               reference by a client to perform an operation on an
//               object. It is part of the three examples used
//               in Chapter 2 "The Basics" of the omniORB2 user guide.
//
#include <iostream.h>
#include "echo.hh"

void
hello(CORBA::Object_ptr obj)
{
  Echo_var e = Echo::_narrow(obj);

  if (CORBA::is_nil(e)) {
    cerr << "hello: cannot invoke on a nil object reference.\n" << endl;
    return;
  }

  CORBA::String_var src = (const char*) "Hello!"; // String literals are not
                                                  // const char*. Must do
                                                  // explicit casting to
                                                  // force the use of the copy
                                                  // operator=().
  CORBA::String_var dest;

  dest = e->echoString(src);

  cerr << "I said,\"" << src << "\"."
       << " The Object said,\"" << dest <<"\"" << endl;
}
```

2.10.3 eg1.cc

```
// eg1.cc - This is the source code of example 1 used in Chapter 2
//          "The Basics" of the omniORB2 user guide.
//
//          In this example, both the object implementation and the
//          client are in the same process.
//
// Usage: eg1
//
#include <iostream.h>
#include "echo.hh"

#include "echo_i.cc"
#include "greeting.cc"

int
main(int argc, char **argv)
{
  CORBA::ORB_ptr orb = CORBA::ORB_init(argc,argv,"omniORB2");
  CORBA::BOA_ptr boa = orb->BOA_init(argc,argv,"omniORB2_BOA");

  Echo_i *myobj = new Echo_i();
  // Note: all implementation objects must be instantiated on the
  // heap using the new operator.

  myobj->_obj_is_ready(boa);
  // Tell the BOA the object is ready to serve.
  // This call is omniORB2 specific.
  //
  // This call is equivalent to the following call sequence:
  //     Echo_ptr myobjRef = myobj->_this();
  //     boa->obj_is_ready(myobjRef);
  //     CORBA::release(myobjRef);

  boa->impl_is_ready(0,1);
  // Tell the BOA we are ready and to return immediately once it has
  // done its stuff. It is omniORB2 specific to call impl_is_ready()
  // with the extra 2nd argument- CORBA::Boolean NonBlocking,
  // which is set to TRUE (1) in this case.

  Echo_ptr myobjRef = myobj->_this();
  // Obtain an object reference.
  // Note: always use _this() to obtain an object reference from the
  //       object implementation.

  hello(myobjRef);

  CORBA::release(myobjRef);
  // Dispose of the object reference.

  myobj->_dispose();
  // Dispose of the object implementation.
```

```
  // This call is omniORB2 specific.
  // Note: *never* call the delete operator or the dtor of the object
  //       directly because the BOA needs to be informed.
  //
  // This call is equivalent to the following call sequence:
  //     Echo_ptr myobjRef = myobj->_this();
  //     boa->dispose(myobjRef);
  //     CORBA::release(myobjRef);

  return 0;
}
```

2.10.4 eg2_impl.cc

```
// eg2_impl.cc - This is the source code of example 2 used in Chapter 2
//               "The Basics" of the omniORB2 user guide.
//
//               This is the object implementation.
//
// Usage: eg2_impl
//
//        On startup, the object reference is printed to cerr as a
//        stringified IOR. This string should be used as the argument to
//        eg2_clt.
//
#include <iostream.h>
#include "omnithread.h"
#include "echo.hh"

#include "echo_i.cc"

int
main(int argc, char **argv)
{
  CORBA::ORB_ptr orb = CORBA::ORB_init(argc,argv,"omniORB2");
  CORBA::BOA_ptr boa = orb->BOA_init(argc,argv,"omniORB2_BOA");

  Echo_i *myobj = new Echo_i();
  myobj->_obj_is_ready(boa);

  {
    Echo_var myobjRef = myobj->_this();
    CORBA::String_var p = orb->object_to_string(myobjRef);
    cerr << "'" << (char*)p << "'" << endl;
  }

  boa->impl_is_ready();
  // Tell the BOA we are ready. The BOA's default behaviour is to block
  // on this call indefinitely.

  return 0;
}
```

2.10.5 eg2_clt.cc

```
// eg2_clt.cc - This is the source code of example 2 used in Chapter 2
//              "The Basics" of the omniORB2 user guide.
//
//              This is the client. The object reference is given as a
//              stringified IOR on the command line.
//
// Usage: eg2_clt <object reference>
//
#include <iostream.h>
#include "echo.hh"

#include "greeting.cc"

extern void hello(CORBA::Object_ptr obj);

int
main (int argc, char **argv)
{
  CORBA::ORB_ptr orb = CORBA::ORB_init(argc,argv,"omniORB2");
  CORBA::BOA_ptr boa = orb->BOA_init(argc,argv,"omniORB2_BOA");

  if (argc < 2) {
    cerr << "usage: eg2_clt <object reference>" << endl;
    return 1;
  }

  try {
    CORBA::Object_var obj = orb->string_to_object(argv[1]);
    hello(obj);
  }
  catch(CORBA::COMM_FAILURE& ex) {
    cerr << "Caught system exception COMM_FAILURE, unable to contact the "
         << "object." << endl;
  }
  catch(omniORB::fatalException& ex) {
    cerr << "Caught omniORB2 fatalException. This indicates a bug is caught "
         << "within omniORB2.\nPlease send a bug report.\n"
         << "The exception was thrown in file: " << ex.file() << "\n"
         << "                              line: " << ex.line() << "\n"
         << "The error message is: " << ex.errmsg() << endl;
  }
  catch(...) {
    cerr << "Caught a system exception." << endl;
  }

  return 0;
}
```

2.10.6 eg3_impl.cc

```
// eg3_impl.cc - This is the source code of example 3 used in Chapter 2
//               "The Basics" of the omniORB2 user guide.
//
//               This is the object implementation.
//
// Usage: eg3_impl
//
//        On startup, the object reference is registered with the
//        COS naming service. The client uses the naming service to
//        locate this object.
//
//        The name which the object is bound to is as follows:
//              root  [context]
//               |
//              text  [context] kind [my_context]
//               |
//              Echo  [object]  kind [Object]
//

#include <iostream.h>
#include "omnithread.h"
#include "echo.hh"

#include "echo_i.cc"

static CORBA::Boolean bindObjectToName(CORBA::ORB_ptr,CORBA::Object_ptr);

int
main(int argc, char **argv)
{
  CORBA::ORB_ptr orb = CORBA::ORB_init(argc,argv,"omniORB2");
  CORBA::BOA_ptr boa = orb->BOA_init(argc,argv,"omniORB2_BOA");

  Echo_i *myobj = new Echo_i();
  myobj->_obj_is_ready(boa);

  {
    Echo_var myobjRef = myobj->_this();
    if (!bindObjectToName(orb,myobjRef)) {
      return 1;
    }
  }

  boa->impl_is_ready();
  // Tell the BOA we are ready. The BOA's default behaviour is to block
  // on this call indefinitely.

  return 0;
}
```

```
static
CORBA::Boolean
bindObjectToName(CORBA::ORB_ptr orb,CORBA::Object_ptr obj)
{
  CosNaming::NamingContext_var rootContext;

  try {
    // Obtain a reference to the root context of the Name service:
    CORBA::Object_var initServ;
    initServ = orb->resolve_initial_references("NameService");

    // Narrow the object returned by resolve_initial_references()
    // to a CosNaming::NamingContext object:
    rootContext = CosNaming::NamingContext::_narrow(initServ);
    if (CORBA::is_nil(rootContext))
      {
        cerr << "Failed to narrow naming context." << endl;
        return 0;
      }
  }
  catch(CORBA::ORB::InvalidName& ex) {
    cerr << "Service required is invalid [does not exist]." << endl;
    return 0;
  }


  try {
    // Bind a context called "test" to the root context:

    CosNaming::Name contextName;
    contextName.length(1);
    contextName[0].id   = (const char*) "test";    // string copied
    contextName[0].kind = (const char*) "my_context"; // string copied
    // Note on kind: The kind field is used to indicate the type
    // of the object. This is to avoid conventions such as that used
    // by files (name.type -- e.g. test.ps = postscript etc.)

    CosNaming::NamingContext_var testContext;
    try {
      // Bind the context to root, and assign testContext to it:
      testContext = rootContext->bind_new_context(contextName);
    }
    catch(CosNaming::NamingContext::AlreadyBound& ex) {
      // If the context already exists, this exception will be raised.
      // In this case, just resolve the name and assign testContext
      // to the object returned:
      CORBA::Object_var tmpobj;
      tmpobj = rootContext->resolve(contextName);
      testContext = CosNaming::NamingContext::_narrow(tmpobj);
      if (CORBA::is_nil(testContext)) {
        cerr << "Failed to narrow naming context." << endl;
        return 0;
      }
```

```
    }

    // Bind the object (obj) to testContext, naming it Echo:
    CosNaming::Name objectName;
    objectName.length(1);
    objectName[0].id   = (const char*) "Echo";   // string copied
    objectName[0].kind = (const char*) "Object"; // string copied


    // Bind obj with name Echo to the testContext:
    try {
      testContext->bind(objectName,obj);
    }
    catch(CosNaming::NamingContext::AlreadyBound& ex) {
      testContext->rebind(objectName,obj);
    }
    // Note: Using rebind() will overwrite any Object previously bound
    //       to /test/Echo with obj.
    //       Alternatively, bind() can be used, which will raise a
    //       CosNaming::NamingContext::AlreadyBound exception if the name
    //       supplied is already bound to an object.

    // Amendment: When using OrbixNames, it is necessary to first try bind
    // and then rebind, as rebind on it's own will throw a NotFoundexception if
    // the Name has not already been bound. [This is incorrect behaviour -
    // it should just bind].
  }
  catch (CORBA::COMM_FAILURE& ex) {
    cerr << "Caught system exception COMM_FAILURE, unable to contact the "
         << "naming service." << endl;
    return 0;
  }
  catch (omniORB::fatalException& ex) {
    throw;
  }
  catch (...) {
    cerr << "Caught a system exception while using the naming service."<< endl;
    return 0;
  }
  return 1;
}
```

2.10.7 eg3_clt.cc

```
// eg3_clt.cc - This is the source code of example 3 used in Chapter 2
//              "The Basics" of the omniORB2 user guide.
//
//              This is the client. It uses the COSS naming service
//              to obtain the object reference.
//
// Usage: eg3_clt
//
//
//        On startup, the client lookup the object reference from the
//        COS naming service.
//
//        The name which the object is bound to is as follows:
//              root  [context]
//               |
//              text  [context] kind [my_context]
//               |
//              Echo  [object]  kind [Object]
//

#include <iostream.h>
#include "echo.hh"

#include "greeting.cc"

extern void hello(CORBA::Object_ptr obj);

static CORBA::Object_ptr getObjectReference(CORBA::ORB_ptr orb);

int
main (int argc, char **argv)
{
  CORBA::ORB_ptr orb = CORBA::ORB_init(argc,argv,"omniORB2");
  CORBA::BOA_ptr boa = orb->BOA_init(argc,argv,"omniORB2_BOA");

  try {
    CORBA::Object_var obj = getObjectReference(orb);
    hello(obj);
  }
  catch(CORBA::COMM_FAILURE& ex) {
    cerr << "Caught system exception COMM_FAILURE, unable to contact the "
         << "object." << endl;
  }
  catch(omniORB::fatalException& ex) {
    cerr << "Caught omniORB2 fatalException. This indicates a bug is caught "
         << "within omniORB2.\nPlease send a bug report.\n"
         << "The exception was thrown in file: " << ex.file() << "\n"
         << "                             line: " << ex.line() << "\n"
         << "The error message is: " << ex.errmsg() << endl;
  }
  catch(...) {
```

```
    cerr << "Caught a system exception." << endl;
  }

  return 0;
}

static
CORBA::Object_ptr
getObjectReference(CORBA::ORB_ptr orb)
{
  CosNaming::NamingContext_var rootContext;

  try {
    // Obtain a reference to the root context of the Name service:
    CORBA::Object_var initServ;
    initServ = orb->resolve_initial_references("NameService");

    // Narrow the object returned by resolve_initial_references()
    // to a CosNaming::NamingContext object:
    rootContext = CosNaming::NamingContext::_narrow(initServ);
    if (CORBA::is_nil(rootContext))
      {
        cerr << "Failed to narrow naming context." << endl;
        return CORBA::Object::_nil();
      }
  }
  catch(CORBA::ORB::InvalidName& ex) {
    cerr << "Service required is invalid [does not exist]." << endl;
    return CORBA::Object::_nil();
  }


  // Create a name object, containing the name test/context:
  CosNaming::Name name;
  name.length(2);

  name[0].id   = (const char*) "test";       // string copied
  name[0].kind = (const char*) "my_context"; // string copied
  name[1].id   = (const char*) "Echo";
  name[1].kind = (const char*) "Object";
  // Note on kind: The kind field is used to indicate the type
  // of the object. This is to avoid conventions such as that used
  // by files (name.type -- e.g. test.ps = postscript etc.)


  CORBA::Object_ptr obj;
  try {
    // Resolve the name to an object reference, and assign the reference
    // returned to a CORBA::Object:
    obj = rootContext->resolve(name);
  }
  catch(CosNaming::NamingContext::NotFound& ex)
    {
```

```
      // This exception is thrown if any of the components of the
      // path [contexts or the object] aren't found:
      cerr << "Context not found." << endl;
      return CORBA::Object::_nil();
    }
  catch (CORBA::COMM_FAILURE& ex) {
    cerr << "Caught system exception COMM_FAILURE, unable to contact the "
         << "naming service." << endl;
    return CORBA::Object::_nil();
  }
  catch(omniORB::fatalException& ex) {
    throw;
  }
  catch (...) {
    cerr << "Caught a system exception while using the naming service."<< endl;
    return CORBA::Object::_nil();
  }
  return obj;
}
```

Chapter 3

The omniORB2 API

In this chapter, we introduce the omniORB2 API. The purpose of this API is to provide access points to omniORB2 specific functionalities that are not covered by the CORBA specification. Obviously, if you use this API in your application, that part of your code is not going to be portable to run unchanged on other vendors' ORBs. To make it easier to identify omniORB2 dependent code, this API is defined under the name space "omniORB"[1].

3.1 ORB and BOA initialization options

`CORBA::ORB_init` accepts the following command-line arguments:

-ORBid "omniORB2" The identifier supplied must be "omniORB2".

-ORBtraceLevel <**level**> This option is described in section 3.2.

-ORBstrictIIOP <**1 or 0**> This option when set instructs the runtime to treat any incoming IIOP message as an error if it has a header message size that is larger than the actual body size. By default, this option is not set to allow omniORB2 to interoperate with some ill-behaved IIOP implementations.

`BOA_init` accepts the following command-line arguments:

-BOAid "omniORB2_BOA" The identifier supplied must be "omniORB2_BOA".

-BOAiiop_port <**port number**> This option tells the BOA which TCP/IP port to use to accept IIOP calls. If this option is not specified, the BOA will use an arbitrary port assigned by the operating system.

As defined in the CORBA specification, any command-line arguments understood by the ORB/BOA will be removed from `argv` when the initialisation functions return. Therefore, an application is not required to handle any command-line arguments it does not understand.

[1] omniORB is a class name if the C++ compiler does not support the namespace keyword.

3.2 Run-time Tracing and Diagnostic Messages

OmniORB2 uses the C++ iostream `cerr` to output any tracing and diagnostic messages. Some or all of these messages can be turned-on/off by setting the variable `omniORB::traceLevel`. The type definition of the variable is:

```
CORBA::ULong omniORB::traceLevel = 1;  // The default value is 1
```

At the moment, the following trace levels are defined:

level 0 turn off all tracing and informational messages

level 1 informational messages only

level 2 the above plus configuration information

level 5 the above plus notifications when server threads are created or communication endpoints are shutdown

level 10 the above plus execution traces

The variable can be changed by assignment inside your applications. It can also be changed by specifying the command-line option: `-ORBtraceLevel` <`level`>. For instance:

```
$ eg2_impl -ORBtraceLevel 5
```

3.3 Object Keys

OmniORB2 uses a data type `omniORB::objectKey` to uniquely identify each object implementation. This is an opaque data type and can only be manipulated by the following functions:

```
void omniORB::generateNewKey(omniORB::objectKey &k);
```

`omniORB::generateNewKey` returns a new `objectKey`. The return value is guaranteed to be unique among the keys generated during this program run. On the platforms that have a realtime clock and unique process identifiers, a stronger assertion can be made, i.e. the keys are guaranteed to be unique among all keys ever generated on the same machine.

```
const unsigned int omniORB::hash_table_size;
int omniORB::hash(omniORB::objectKey& k);
```

`omniORB::hash` returns the hash value of an `objectKey`. The value returned by this function is always between 0 and `omniORB:hash_table_size - 1` inclusively.

```
omniORB::objectKey omniORB::nullkey();
```

`omniORB::nullkey` always returns the same `objectKey` value. This key is guaranteed to hash to 0.

```
int operator==(const omniORB::objectKey &k1,const omniORB::objectKey &k2);
int operator!=(const omniORB::objectKey &k1,const omniORB::objectKey &k2);
```

`ObjectKeys` can be tested for equality using the overloaded `operator==` and `operator!=`.

```
omniORB::seqOctets*
omniORB::keyToOctetSequence(const omniORB::objectKey &k1);

omniORB::objectKey
omniORB::octetSequenceToKey(const omniORB::seqOctets& seq);
```

`omniORB::keyToOctetSequence` takes an `objectKey` and returns its externalised representation in the form of a sequence of octets. The same sequence can be converted back to an `objectKey` using `omniORB::octetSequenceToKey`. If the supplied sequence is not an `objectKey`, `omniORB::octetSequenceToKey` raises a `CORBA::MARSHAL` exception.

3.4 Trapping omniORB2 Internal Errors

```
class fatalException {
public:
    const char *file() const;
    int line() const;
    const char *errmsg() const;
};
```

When omniORB2 detects an internal inconsistency that is most likely to be caused by a bug in the runtime, it raises the exception `omniORB::fatalException`. When this exception is raised, it is not sensible to proceed with any operation that involves the ORB's runtime. It is best to exit the program immediately. The exception structure carries by `omniORB::fatalException` contains the exact location (the file name and the line number) where the exception is raised. You are strongly encourage to file a bug report and point out the location.

Bibliography

[OMG96a] *The Common Object Request Broker: Architecture and Specification*, Revision 2.0, OMG, Updated July 1996.

[OMG96b] *CORBAservices: Common Object Services Specification*, OMG, Updated July 1996.

[Richardson96a] *The OMNI Thread Abstraction*, Tristan Richardson, ORL, 22 October 1996.

[Richardson96b] *The OMNI Development Environment Version 4.0*, Tristan Richardson, ORL, 5 November 1996.