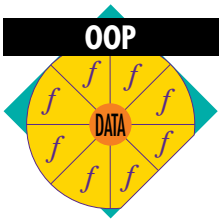

Distributing Objects with Orbix

By Annrai O'Toole

Object Request Broker (ORB) technology is an essential component in developing distributed object-oriented applications. This article discusses the Common Object Request Broker Architecture (CORBA) using examples from Orbix, a commercially available implementation of the CORBA specification from IONA Technologies.

The Object Management Group's® (OMG's®) Common Object Request Broker Architecture (CORBA) is a viable option for attaining open and interworking object-oriented software. CORBA is a set of rules for developing software objects and applications that will interoperate across many system platforms and operating systems.

Several vendors are already providing implementations of CORBA specifications: IBM with Distributed System Object Model (DSOM), Digital, HP™, Sun®, and IONA Technologies. IONA began shipping Orbix, the first complete CORBA implementation with a C++ binding, in June 1993.

The basic CORBA specification published by OMG describes an Interface Definition Language (IDL) and the relationship between the IDL and C, C++, and SmallTalk®. The specification also lists several standard interfaces that must be provided in all ORB implementations. OMG is currently engaged in a process to provide a specification for an IDL to Ada mapping. Other language bindings are expected.

IDL defines interfaces, not how the interfaces are actually implemented. It provides constructs for defining new types, including complex types such as arrays, sequences, structs/records, and unions/variants, as well as the interfaces. The operations supported by interfaces help to define the interface, and inheritance relationships between interfaces can be defined. IDL does not provide executable statements—there are no assignments or statements such as `if`, `while`, or `for`.

IDL makes CORBA an open standard. It defines interfaces that are neutral to any particular programming language. Interfaces to objects can be defined in IDL; they are then automatically transformed into the corresponding interfaces in different programming languages. For example, a C++ client could use a Smalltalk object and vice versa; objects may also be programmed in non-object languages such as C, COBOL, or Ada.

A Simple CORBA Application

Consider a simple banking application in which entities such as banks, bank accounts, and bank clerks can be represented as objects. This example shows inheritance—because there can be different categories of accounts. It also shows distributed programming—because a customer may want to access a bank account remotely from the bank server machine that maintains it.

Similar to C++, the account interface has three components: a read-only attribute that represents the current balance in the account, and two operations to alter the balance. For simplicity, we represent sums of money as floating-point numbers, although this would not be suitable in a commercial application because of potential numeric rounding errors. Figure 1 shows how the basic functionality of a single bank account may be expressed using IDL, and the code produced when an IDL compiler translates it into C++.

The attribute balance has been represented in C++ as a member function returning the value of the balance. If the attribute had not been read-only, there would have been a second member function—taking a float argument and returning a void—to set the balance.

IDL can be translated into C++, but it is generally not possible to translate all C++ type


Annrai O'Toole

declaration constructs into IDL. For example, IDL does not allow overloading of member functions (constructors or destructors), or default and ellipsis arguments. Since IDL is language neutral, any features supported by IDL should be translatable to a wide range of popular object-oriented and non-object-oriented languages.

Both account objects and bank objects to manage the accounts are required. Figure 2 shows the bank interface using IDL and the interface mapped into C++.

References to objects, such as accounts described in IDL, are represented as pointers to classes in the C++ equivalent—shown here as `account*`.

The IDL compiler also builds definitions of the member functions of account and bank classes; each of these definitions builds and sends an appropriate request to a remote server.

Example Client Program

A major benefit of CORBA is its ease in building distributed programs. The following example shows how to use the account and bank classes in a client program that executes remotely from the bank server program, which in turn manages the bank and maintains the accounts.

Figure 3 shows C++ code examples using several specific features of Orbix. ORBs from other vendors provide similar functionality.

Orbix provides the C++ static member function `_bind` call that requests the ORB to search the network for any bank server, activating such a server if necessary. Figure 4 compares a search of the network for a specific bank server to a search of a specific host ("Bank" refers to the name of the bank).

Although header files are ignored here, in practice, the program would include both `<stream.h>` and the C++ header file generated by the IDL compiler containing the C++ equivalent of the bank and account IDL interfaces. Possible error conditions are also ignored in these examples.

The IDL exception-handling mechanism allows standard ORB errors to be described, and also allows object programmers to introduce their own application-specific exceptions. Any IDL-described operation can result in an exception. Exceptions are mapped in C++ as a trailing, defaulted argument to every operation. In fact, the C++ code generated by the IDL compiler for an account is shown in Figure 5.

Because environments are defaulted, they can be ignored when appropriate; that is, when the

Bank Account Expressed in IDL

```
//IDL
interface account {
   read-only attribute float balance;
   void makeLodgement (in float f);
   void makeWithdrawal (in float f);
};
```

Bank Account Translated into C++

```
// C++
class account {
public:
   virtual float balance ();
   virtual void makeLodgement (float f);
   virtual void makeWithdrawal (float f);
};
```

Figure 1. The account IDL interface and its mapping in C++

Bank Interface Using IDL

```
//IDL
interface bank {
   account lookUpAccount (in string customerName);
   account newAccount (in string customerName);
};
```

Bank Interface Mapped into C++

```
// C++
class bank {
public:
    virtual account* lookUpAccount (char* customerName);
    virtual account* newAccount (char* customerName);
};
```

Figure 2. Bank IDL interface and its mapping in C++

```
// C++
main () {
   // bind to *any* bank service
   bank *b = bank::_bind ();

   // create a new account
   account *a = b->newAccount ("Marie");

   // lodge $10
   a->makeLodgement (10.00);

   // output the balance
   cout << "balance is " << a->balance () << endl;
}
```

Figure 3. A simple client application

Searching the Network

```
bank *b = bank::_bind ("Palo Alto:Bank");
```

Searching a Specific Host

```
bank *b = bank::_bind ("Palo Alto:Bank", safeHaven.mil);
```

Figure 4. Controlling what interface object a client application use

```
// C++
class account {
public:
  virtual float balance
     (Environment&env=default_environment);
  virtual void makeLodgement
     (float f,Environment&env=default_environment);
  virtual void makeWithdrawal
     (float f,Environment&env=default_environment);
};
```

Figure 5. Adding CORBA environments to operation signatures

```
// C++
main () {
   TRY {
     // bind to *any* bank service
     bank *b = bank::_bind (IT_X);

     // find an account
       account *a = b->newAccount ("Marie",IT_X);

     // lodge $10
     a->makeLodgement (10.00,IT_X);

     // output the balance
     cout << "balance is " <<
              a->balance (IT_X) <<endl;
  }
  CATCHANY {
     cerr << "Unexpected exception "
            << IT_X << endl;
  }
  ENDTRY
}
```

Figure 6. Adding exception handling to a CORBA application

client and server object are on the same machine or in the same UNIX process. In these cases, there are no exceptional conditions caused by network or machine failure. All predefined identifiers relating to the ORB are scoped with an enclosing class module called CORBA, resulting in the following environment argument (the CORBA scoping is omitted in the code below).

```
CORBA::Environment&
   env=CORBA::default_environment
```

Figure 6 shows several predefined macros in Orbix that simplify catching an exception in the client program.

The TRY macro introduces the (automatic) variable `Environment IT_X`. The Environment class supports `operator <<` on ostreams, so the details of the error can be simply generated.

In Fall 1994, support for C++ exceptions was added in the OMG C++ to IDL mapping adopted by the OMG. This C++ forms part of the CORBA 2.0 specification and will be available from IONA in mid-1995.

Example Server Program

The server program implementation is straightforward. In addition to generating the account and bank C++ classes used by the client programs, the IDL compiler also declares two derived classes of these for use by server programmers, as shown in Figure 7.

Each class inherits from the corresponding C++ class that is also used by the client code. Because one server can be a client of another, it requires access to the code to construct and marshal requests.

The member functions of the `BOAImpl` classes are defined `=0` so that these classes are abstract and cannot be directly instantiated. Server programmers are expected to implement each member function in a derived class. Figure 8 shows implementations of the account and bank classes.

The list of opened accounts is maintained in a serial list. Figure 9 shows how to build a new account.

The `acc` is a derived class of account. We have ignored certain rules used by the ORB and server code for managing dynamically allocated memory. When a request for a `newAccount` arrives, the ORB dynamically allocates a new string for the customer name argument and deletes this string when the request has finished. For this reason, it is important that the server program return a copy of the string to the client application.

```
// now build new account
   account *pa = new acc
      (strdup(customerName), 0.0);
```

Similarly, the ORB recovers the storage occupied by any IDL-defined objects when these are passed to it as return or out parameters of a

request. To avoid losing the newly created account when a pointer to it is returned from the function, the ORB must be instructed to keep it after the request finishes:

```
// now build new account
   account *pa = new acc
      (strdup(customerName), 0.0);
   pa->_duplicate ();
```

The `_duplicate` increments a reference count associated by the ORB with the account object. The ORB will delete the object once its reference count reaches zero.

The following represents the server mainline:

```
main () {
   banker myBank;
   Orbix.impl_is_ready ();
}
```

The mainline is clearly trivial. Incoming requests are directed to a banking object, telling the ORB that we are now ready to take incoming requests. The server is implementing two distinct IDL interfaces: bank and account. The mainline needs to create only a bank; the bank can create accounts as requested.

The `impl_is_ready` call will automatically timeout if consecutive requests are not received within a certain period (approximately five minutes). The timeout value can be explicitly specified in milliseconds:

```
Orbix.impl_is_ready (30*60*1000);
   // 30 minutes
```

Like most calls on the ORB, `impl_is_ready` can raise an exception; for example, the ORB itself cannot find its configuration information. If the `impl_is_ready` eventually times out, it does not raise an exception; timing out is considered the normal outcome of `impl_is_ready`. The following is a more appropriate way to code the mainline:

```
main () {
  banker myBank;
  TRY {
     Orbix.impl_is_ready (IT_X);
  }
  CATCHANY {
     cerr << "Exception:: " << IT_X <<
        endl;
  }
  ENDTRY
}
```

Account Skeleton Class

```
// C++
class accountBOAImpl : public account {
public:
   virtual float balance
       (Environment&env=default_environment)=0;
   virtual void makeLodgement
       (float f,Environment&env=default_environment)=0;
   virtual void makeWithdrawal
       (float f,Environment&env=default_environment)=0;
};
```

Bank Skeleton Class

```
class bankBOAImpl : public bank {
public:
   virtual account* lookUpAccount (char* customerName,
         Environment&env=default_environment)=0;
   virtual account* newAccount (char* customerName,
         Environment&env=default_environment)=0;
};
```

Figure 7. Skeleton classes for bank and account

Account Class

```
class acc : public accountBOAImpl {
   char    *m_name;
   float m_balance;
public:
   acc (char* name, float bal) { // constructor
      m_name = name; m_balance = balance; }

   float balance (Environment&env=
                  default_environment) {
      return m_balance; }

   void makeLodgement (float f,Environment&env=
                   default_environment) {
      m_balance += f;}

   void makeWithdrawal (float f,Environment&env=
                    default_environment) {
      m_balance -= f;}
};
```

Bank Class

```
class banker : public bankBOAImpl {
protected:
   struct accountsList { account   *ac;
      accountsList   *next;
      };
      accountsList   *m_head;

public:
   virtual banker () {m_head = nil;}
   account* lookUpAccount (char* customerName,
        Environment&env=default_environment);
   account* newAccount (char* customerName,
        Environment&env=default_environment);
};
```

Figure 8. Implementation classes for bank and account

```
account* banker::newAccount (char* customerName,
                        Environment&) {
   // check customer's account does
   // not already exist...
   // now build new account
   account *pa = new acc (customerName, 0.0);

   // insert it into the list...
   // finished
   return pa;
}
```

Figure 9. Implementing the newAccount operation

The ORB knows that incoming bank requests should be directed to `myBank`. What would happen if the mainline created several different bank objects—which one of them would be used? These issues are discussed in the next section.

Activation

The ORB uses a registry of servers called the *implementation repository*. Servers must be preregistered with this repository before they can be used with an ORB. Once registered, the ORB will automatically relaunch that server if any client attempts to use it. The ORB also allows the server to be launched manually.

A shell-level command registers a server as follows:

```
% putit bank /usr/users/me/a.out
```

The `putit` command and the implementation repository associate a server name with an executable file. Server names must be unique within a repository, but they can be hierarchically structured. Each server machine node in the network is expected to have access to at least one repository. Different server machines can use different repositories.

Th following example uses the server name Bank:

```
% putit Bank /usr/users/me/a.out
```

When a client does a `bank::_bind()` request, the ORB uses the implementation repository to find a server named bank (and not Bank) If the server name differs from the IDL interface name (Bank differs from bank), then the `_bind` request must name the server explicitly, and the ORB will search the repository for a server called Bank, not bank:

```
bank *b = bank::_bind (":Bank");
```

A client program can also specify a specific object within a server to which it wishes to bind:

```
bank *b = bank::_bind ("Palo Alto:Bank");
```

In this case, the server must name the objects to which it expects clients to bind, using the `_marker` call:

```
main () {
  banker myBank;
  myBank._marker ("Palo Alto");

  banker anotherBank;
  anotherBank._marker ("Walnut Creek");

  TRY {
    Orbix.impl_is_ready (IT_X);
  }
  CATCHANY {
    cerr << "Exception:: " << IT_X <<
       endl;
  }
  ENDTRY
}
```

If a client does not specify a particular object within a server during a bind, then the ORB can choose any type-compatible objects. If a client enters `bank *b = bank::_bind ();`, then either `myBank` or `anotherBank` can be used to satisfy the bind.

Execution Trace

The bank server creates a banker object when it is run. This results in a quiescent server, waiting for incoming requests.

```
main () {
  banker myBank;
  TRY { Orbix.impl_is_ready (IT_X);
```

When the client starts, it first binds to any bank service:

```
main () {
   // Bind to *any* bank service
   bank *b = bank::_bind()
```

The result of the binding is an automatically generated proxy or surrogate object that acts as a stand-in for the remote banker object in the server.

Programmers do not need to be aware of the proxy object since it is managed automatically by the ORB. The client program then asks the bank to open a new account.

```
// find an account
account *a = b->newAccount
   ("Marie",IT_X);
```

Within the server, `banker::newAccount` is called, generating a new `acc` object. The `acc` object is linked to the banker object's list of accounts (via `banker::m_head`). Finally, `newAccount` returns a pointer to the account back to the client. At the client side, a new proxy for the remote pointer is automatically created.

Collocation

For some applications, IDL can be used to define the interfaces between components, even if these components are not distributed. Components of some applications may be distributed, depending on how the application is configured by its installer. Developers of distributed applications may want to temporarily run code with all components bound in the same process address space, such as to aid debugging.

To support these issues, the ability to collocate client and server objects within the same process address space is essential. For collocation to be as efficient as normal object invocations, the underlying ORB should not normally be involved in the invocation path once a binding has been established. In particular, messages should not necessarily be marshaled into and unmarshaled from message buffers. This implies that the mapping from IDL to the target programming language (such as C++) should be perfectly symmetrical between the client and server sides of each interface. Consequently, client and server code can be directly bound together.

Orbix supports collocation by linking with a particular version of the ORB library, which never attempts to locate an object outside the current process address space. A program can be relinked to use the normal form of the ORB library; a runtime test can also be made to determine the library to which the application is currently linked.

If collocation is always to be used for a particular (non-distributed) application, it may be appropriate to ignore the trailing environment values in client calls, thus ignoring system-raised exceptions.

Figure 10 shows a collocatable form of the bank example.

In effect, the initialization of the server mainline has been moved into the client code (a heap allocated object was used instead of an automatic object, which retains the server object after the collocation test). The code in Figure 10 can then be run collocated or distributed (against the original server code). In the collocated case, an execution trace would result in dynamic invocation.

```
// C++
main () {
  // build server object in this process
  // if need be...
  if (Orbix.collocated ()) {
     banker *myBank = new banker (); }
  // bind to *any* bank service
  bank *b = bank::_bind ();
  // create a new account
  account *a = b->newAccount ("Marie");
  // lodge $10
  a->makeLodgement (10.00);
  // output the balance
  cout << "balance is " << a->balance () <<
endl;
}
```

Figure 10. Collatable bank example

Dynamic Invocation

IDL describes the interfaces to an object. In the previous application examples, we examined how to write an IDL interface and then provide some implementation code behind that interface. We showed how to build client code to access that IDL object. The client application used code generated by an IDL compiler. All the code to build and send a request to the IDL interface was compiled statically into the client application at compile time.

Using the IDL compiler in this way limits some applications. The IDL interfaces that a client program can use are determined when the client program is compiled. Therefore, the client code can use only those servers that provide the IDL interfaces selected by the client programmer when building the application. Some application programs and tools require that they can use an indeterminate range of interfaces—such as browsers, management support tools, command-line and interactive interfaces, and interfaces that perhaps are not even conceived at the time the application is developed.

The OMG's CORBA specification also introduces a dynamic invocation interface that allows an application to issue requests for any interface, even if that interface was unknown when the application was compiled. Invocations can be constructed at runtime by specifying the target object reference, the operation name, and the parameters. Invocation is dynamic, because the IDL interfaces used by the client application do

```
TRY {
  r.invoke (IT_X);
}
CATCHANY {
  cerr << "Exception " << IT_X << endl;
  // maybe exit
}
ENDTRY

Object *a;
r >> a;   // new account
```

Figure 11. Simple dynamic invocation

```
//IDL
interface account {
   readonly attribute float balance;
   void makeLodgement (in float f);
   void makeWithdrawal (in float f);
};

interface checkingAccount : account {
   readonly attribute float overdraftLimit;
};

interface bank {
   account lookUpAccount (in string customerName);
   account newAccount (in string customerName);
   checkingAccount newCheckingAccount
         (in string customerName, in float limit);
};
```

Figure 12. Interface inheritance using IDL

not have to be statically determined at the time the program is compiled.

A server receiving an incoming request does not know—or care—whether the client that sent the request used the static or dynamic approach to compose a request. However, the underlying ORB infrastructure must ensure that the request sent by the client is type-safe against what the server expects. Dynamically composed requests must be tagged at runtime with sufficient type information to enable the ORB at the server side to assert type safety at runtime. Clearly, the expense of runtime type assertions should only be incurred for dynamically composed requests.

The CORBA specification provides an Application Programming Interface (API) to allow client code to dynamically compose requests, based on lists of "named values." Each named value is also tagged with type information. A named value list (an NVList) constitutes the arguments, including `out` parameters, to any particular invocation.

CORBA Object Reference

A CORBA object reference (as opposed to a C++ pointer to a C++ object) is defined in CORBA by the IDL interface object. CORBA's object interface is also the implicit root of all other IDL interfaces. For example, both previously discussed IDL interfaces—account and bank—are implicitly derived from Object.

Any CORBA object reference can be translated into a character-string format by using the operation `object_to_string` (invoked on the ORB itself), and back again using `string_to_object`. This is one way in which an object reference can be initialized. The CORBA specification does not directly address obtaining suitable string values that can be converted into object references. For example, they could be obtained via an external name service into which object references had previously been registered.

All CORBA-conformant ORBs support the dynamic invocation API. Because the API is complex to use, Orbix also provides a "veneer" over the API in C++, which makes the API easier to use. In the bank example, the first issue is to build an object reference for the bank server. If we use `_bind` again, this assumes that we know in advance to use the bank interface and can use the `_bind` call generated by the IDL compiler from the bank interface. In general, we do not know in advance what interface we want to use. Instead, we might initialize an object reference from a character string obtained externally to the program, using the C++ constructor for Object[1].

We can now continue with a code example to demonstrate making the first dynamic invocation that will create a new account object. We first create a CORBA request object, then initialize it with the desired target object reference and operation name.

```
// create an account....
Request r (&b, "newAccount");
// strictly,CORBA::Request
```

Next, we insert the arguments. In Orbix, class Request supports a stream-based interface to do this: `r << "Marie"`.

Because C++ is strongly typed, the stream operators can tag the underlying NVlist with the correct type information. In contrast, the C

[1] Object is strictly `CORBA::Object`, but we have omitted `CORBA::` prefixes for brevity.

language interface to the dynamic invocation interface requires programmers to supply their own type tags. It is then possible to invoke and obtain the invocation result—in this case, another object reference for the newly created account, shown in Figure 11.

An exception will be raised during the invocation if the dynamic type checking fails. We can continue to further invocations.

For an IDL-specified attribute such as balance, the operation name used during the dynamic invocation interface must specify whether the value is being sought or set. The example above shows the dynamic approach. It is typical practice to require information about what IDL interfaces a particular (remote) object supports, and to compose messages to it. CORBA specifies further interfaces that assist in this work, specifically a service responsible for managing information about interfaces themselves—the interface repository.

The CORBA specification contains more information about the interface repository.

Inheritance in IDL

Inheritance in IDL allows a new interface to be refined by extending the functionality provided by the earlier one. The new interface inherits or derives from the first. This is similar to the way in which a C++ class can be derived from its base class. Both IDL and C++ support multiple inheritance, allowing an interface (or class) to have several immediate base interfaces (or classes).

Figure 12 shows an example of single inheritance using the bank account.

The new interface `checkingAccount` derives from `account`. We have also added a new operation to interface bank to manufacture `checkingAccounts` (we could also have derived from the interface bank to produce a new interface that supports the new operation; however, this might make the example more difficult to explain).

The C++ class hierarchy produced by the IDL compiler ignores the Environment arguments, shown in Figure 13.

The usage from a client is similar to the previous example. The new server code that results is also predictable from the earlier example (see Figure 14).

Conclusion

The banking example describes what using an ORB might look like from C++, following the specifics of the Orbix ORB, but using the general principles implemented by all CORBA-conformant ORBs.

Writing distributed applications using ORB technology does not require much additional learning once you understand C++. Using IDL to describe interfaces to system components (whether or not they are distributed) and using an ORB to bind components together can offer substantial savings in design, coding, and maintenance.

The CORBA specification provides a baseline technology for distributed object-oriented systems in the computing industry. OMG is currently considering various value-added services that can be standardized within COBRA and used to augment a basic ORB. Services being considered include object life-cycle and relationships, persistence

```
// C++
class account {
public:
   virtual float balance ();
   virtual void makeLodgement (float f);
   virtual void makeWithdrawal (float f);
};

class checkingAccount : public virtual account {
public:
   virtual float overdraftLimit ();
};

class bank {
public:
   virtual account* lookUpAccount (char* customerName);
   virtual account* newAccount (char* customerName);
   virtual checkingAccount* newCheckingAccount
           (char* customerName, float limit);
};
```

Figure 13. C++ mapping for IDL inheritance

```
// C++
main () {
  TRY {
  // bind to *any* bank service
  bank *b = bank::_bind (IT_X);
  // create a new account
  checkingAccount *cA =
    b->newCheckingAccount ("Grady", 1000.00, IT_X);
  // lodge $10
  cA->makeLodgement (10.00, IT_X);
  // output the limit
  cout << "limit is " << cA->overdraftLimit (IT_X)
                      << endl;
}
```

Figure 14. Client application code using inheritance

and storage, event handling, naming and directory services, and transactions.

ORB vendors are also adding features that should help the acceptance of the ORB technology. For example, Orbix allows server programmers to indicate specific-purpose proxies for the services to clients, and provides hooks for transparent filtering and interpretation of invocations.

ORB technology can help compose application components in the same address space, in the same machine, and across different machines including cross-language and cross-operating system bindings.

Annrai O'Toole, IONA Technologies Ltd., 8-34 Percy Place, Ireland. Telephone: 353-1-6686-6522. In the U.S., 1-800-ORBIX4U. E-mail: info@iona.ie. Mr. O'Toole , a co-founder of IONA, is vice president for development at IONA Technologies. He is responsible for technical and business strategy. He has degrees in Engineering from Dublin University and Trinity College and an MSc in Computer Science from Trinity College.