Hedge your bets:

Integrating, Distributing applications via CORBA

By the Semaphore staff.

CORBA lets you integrate diverse back-end applications with language and platform independence, and realize object-oriented benefits.

Your business need is to combine the functionality of several applications effectively, providing an overall problem solution. The risk in traditional application integration is that it typically involves low-level network and operating system programming, making the resulting system difficult to maintain and even harder to extend.

Distributed Object Computing (DOC) minimizes the risk. It introduces a new paradigm for application integration in which applications and services are viewed as objects with well-defined interfaces. Client applications only access the objects through a well-defined interface, and need only code to that interface.

DOC makes systems created extremely flexible. Client applications can be added at any time. An object's implementation can be changed as needed — as long as an interface to an object doesn't change, the client application needn't be rebuilt. Applications modeled as objects can be reused. Need a new one? Use a resembling existing object, inherit from it, then extend it. The development cycle will be shorter and the system less expensive to develop and easier to maintain.

Why CORBA?

Application Integration Method	CORBA's advantage
Remote Procedure Calls	Follows the object-oriented model
Object Oriented DBMS	Flexibility to add non-DBMS future applications
Microsoft COM/DCOM	Supports both PC- and non-PC platforms
Enterprise JavaBeans	Language independence

The OMG, OMA and CORBA

The driving force behind the DOC movement is the Object Management Group (OMG). Founded in 1989 to formulate a standard framework for distributed object computing and related services, OMG membership has grown to over 800 companies.

The first OMG document was the Object Management Architecture (OMA). It specifies the overall object model that is used, and it is the basis for the Common Object Request Broker Architecture and Specification (CORBA) document. The CORBA spec describes components of the DOC framework and interfaces

CORBA implementations must support. Several companies are shipping implementations of the CORBA spec. Others offer developers tools that aid in building CORBA-based distributed systems.

The Object Request Broker (ORB) is the heart of the system. An ORB delivers requests from client applications to server applications. The client's interface is completely independent of the object's implementation. The underlying ORB implementation is unimportant to distributed system developers. The key is that the interfaces to the ORB and to objects built using the ORB are well defined, providing a uniform framework across the entire distributed environment and making applications built using an ORB very portable across diverse platforms.

The Client View

Clients invoke operations on *object references.* The ORB manages these references. The implementation manages the object itself, providing a great deal of flexibility from the application writer's standpoint. How the object is represented and stored is completely up to the application developer. The ORB provides the mechanism to associate object references with the object. Clients manipulate object references, or *proxy objects*. The server maintains the real object.

The CORBA specification provides two operation invocation interfaces, the static invocation interface (SII) and the dynamic invocation interface (DII). They serve the same purpose. The one to use depends on how much information the client application writer has at compile time. SII is a very simple interface, just an invocation on a C++ or Java object, and is the more commonly used.

DII is used when the client application doesn't know all of the information needed for a method invocation at compile time. The client application must write a little more code, but gains flexibility. The client application basically creates a request object that contains information such as the object to invoke on, the operation to invoke, and the parameter list and then invokes the "send" operation on the request object. A message goes to the server for implementation.

CORBA supports synchronous or deferred synchronous requests. The former block the client application until the result is returned. In a deferred synchronous request, the actual invocation returns immediately, and the client application must poll for a response. This allows the client to perform other work while the server processes the request. Deferred synchronous invocations are useful when the client application must handle other events while the request is being satisfied. (An example would be a windowing system. If a client application were blocked on invocation the window system would appear to hang until the method completes.)

The IDL

After initial analysis and design for a distributed system, specifying the interfaces your objects expose is the first step in implementing it. Interfaces represent contracts between client and server applications. CORBA's abstract notation for describing interfaces is the Interfaces Definition Language (IDL). IDL is language independent. Its syntax

is similar to that of C++, and includes constructs for modules, interfaces, operations, attributes, user-defined types, and exceptions. *IDL says nothing about the implementation of the interface*. CORBA products have different ways of handling implementation descriptions. All CORBA products generate language-specific bindings based on the IDL definitions. The C language mapping was the first specified by CORBA, and the OMG has approved C++, Java and Smalltalk bindings.

IDL definitions can be stored in an interface repository, which clients using the DII can use to do runtime discovery of new operations on an interface. Clients can also use the repository to perform type checking.

The Server View

The server manages the implementation side of the system. From the programmer's perspective, it's more complicated than the client side. The CORBA spec is very precise with respect to the client side. On the implementation side it's less so.
An implementation accesses the ORB through the Basic Object Adapter (BOA), which should be supported by all CORBA implementations. The BOA provides basic implementation functionality, including creating and managing object references, activating and deactivating implementations and objects, and method dispatching.

Basically, a server creates an object reference using the BOA create operation, supplying an interface definition and an implementation definition. Interface and implementation definitions can be retrieved from a repository. Or as a convenience, the ORB may generate them. The server then activates the implementation, telling the ORB that the implementation is ready to respond to client requests. The server goes into an event loop, waiting to dispatch invocations. The event loop isn't specified as part of the BOA interface; however all ORB vendors provide one. Most allow their loops to be integrated with other event dispatching models, such as X and MS Windows. The real work occurs once client requests arrive at the implementation.. Access is performed to one or more legacy and heritage applications, databases and other back end resources. The finer points of this interaction are for another article. What is important is that the invoking client is working with an object interface to complete its tasks. The details of application integration are encapsulated from the client.

This is an oversimplification. Many other issues (in which each ORB product provides varying degrees of support) are involved. These include event processing, caching, security, concurrency control, and threading. Implementing ORB servers is highly dependent on the system, the language, and the particular ORB product.

CORBA vs. Other Application Integration Methods

There's several ways to skin the cat. Other methods of application integration include Remote Procedure Calls (RPC), using an object oriented database management system (OODBMS), Microsoft's COM/DCOM and Enterprise Java Beans (EJBs).

Remote Procedure Calls

At first, CORBA appears to be very similar to RPC. The difference lies in the model. RPC's are inherently procedural. CORBA follows an object-oriented model. CORBA provides more of a framework for integration. RPCs aren't less useful for application integration; the underlying model just differs.

Object-Oriented Database Management System

An OODBMS work best in very data-centric applications. Systems centered on information stored in a database are probably best integrated using an OODBMS. Those with broader requirements (for example, peer-to-peer communication) are best integrated using CORBA. A database centric solution doesn't allow integrating non-database applications. If initial system requirements call only for database integration, you may want to consider using an ORB to provide future application integration flexibility.

Microsoft COM/DCOM

CORBA and Microsoft's COM/DCOM offer similar approaches to application integration. While Microsoft has a strong object strategy for the PC world, it hasn't addressed the application integration problem on non-PC platforms. CORBA provides a solution on both PC and non-PC platforms, and it can certainly be used to integrate systems across these platforms. OMG has published a specification for COM to CORBA interoperability to take advantage of both COM on PC platforms and CORBA on other platforms. Several vendors have implemented this spec, a “best of both worlds’ solution for developers.

Enterprise Java Beans

EJB also offers opportunities for application integration. By definition, integration points must be Java compatible. This works well in Java-only environments. Application integration requiring multi-language access is less feasible. Porting technologies such as (Object Database Connectivity) ODBC to JDBC have helped in this area. The EJB spec contains several artifacts from the CORBA world (naming, transactions, etc). As noted below, the EJB and CORBA worlds are merging, also allowing for a multiple technology solution.

The Future

The CORBA 3.0 specification gives us a view into CORBA's future. It adds Internet integration, quality of service control and the CORBA component architecture. Internet integration includes a firewall specification for Internet Inter-ORB Object Protocol (IIOP) traffic and an interoperable Naming Service, which defines URL naming formats. Quality of service comprises an asynchronous messaging service with policies for quality control, a minimum CORBA specification aimed at embedded systems, real-time CORBA and a fault tolerant CORBA request for proposals. The CORBA component architecture includes a container for transactions, security and persistence, integration with EJBs, thus allowing multi-language component development and a software distribution format for CORBA components.

Wrapping it up

Application integration isn't simple and straightforward. If you must develop a totally new solution for each system, it's downright painful. Distributed object computing with CORBA helps to

minimize the amount of new code developed yet maximize system flexibility.

Distributed object computing may not be the best solution for every application integration job, but it provides an approach to solving a familiar problem that could save time, effort, and ultimately money.

A Coding Example

Let's implement a distributed license checking system as an example. Define an interface called "License". We'd like to have an operation "check" that will verify if a user is licensed for a given product. Here are the IDL definitions for our example:

```
interface License {
unsigned short check (in string user, in
string          product,in long
productID);
};
```

This is a very simple example. In practice, systems may contain hundreds, even thousands of IDL definitions.

Using a C++ mapping the client code would look like this:

```
License_ptr          anObject;
unsigned short       aResult;

// retrieve the license object reference
here
// make the call
   try {
        aResult = anObject-
>check("a_user",

“my_app", 534123);
   }
   catch {...} // check and handle errors
here
```

For now, assume we already have a license object. A client typically obtains an object reference as the result of another invocation, or through some external mechanism such as a name service. The code above looks pretty straightforward, doesn't it? When the call is made, the ORB does the work. It will attempt to find a server in the environment and deliver the request. If it can't find a server, it may start one on behalf of the client. The server then executes the method and returns the result. What could be easier?

Semaphore's home page is [*http://www.sema4usa.com*](http://www.sema4usa.com).