

# Need for Bridging

---

*There is a holy, mistaken zeal in politics,  
as well as religion. By persuading others  
we convince ourselves.*

— Junius (18th Century)  
pseudonym of a writer never identified

It is a truism that the best computer systems are built from hybrids of the best technology and tools available at the time. For this reason, the concept of building a distributed system, which contains elements from both CORBA and DCOM, becomes very attractive. It means that you can take several existing applications and combine them into a single distributed application with a minimum of application programming, the most common case being that you end up factoring out pieces of the existing applications and combining them to build a new application.

The decision to combine multiple object systems into a single application gives you the freedom to choose the best attributes from both systems and combine them to build the best possible application that specifically suits your environment. For example, COM/DCOM with its associated ease-of-use development tools might provide the

best solution for building the front-end for your system. This could incorporate components from Visual Basic, commercial ActiveX components, plus components from other COM-aware visual development environments. You could choose to use CORBA for your middle tiers, and back-end servers running on Unix, integrated with Oracle databases, components like MQSeries<sup>1</sup> on MVS mainframes, or OrbixTalk<sup>2</sup> on multiple platforms.

In order to transparently couple components from DCOM and CORBA, some form of bridging software is needed to handle the translation of types and object references between the two systems. As mentioned earlier, DCOM and CORBA are similar at a high level but differ quite considerably the lower down you go, closer to the bare metal. The question arises, what is actually required to build a bi-directional bridge between the two systems? At a basic level, we need to be able to do the following:

- Transparently contact objects in one system from the other.
- Use data types from one system as though they were native types in the other system.
- Maintain identity and integrity of the types as they pass through the bridging software in order to reconstitute them later. For example, if a CORBA object is passed as a parameter into COM, if the same object ever reappears back in the CORBA world, it should reassert self with its original object identifier and not appear as a wrapper to a COM object. (This is described in more detail in later chapters.)

## Background of DCOM/CORBA Bridging

Since early in 1995, work has been ongoing in the industry to provide a solution to this specific problem, which started with the first commercial OLE/CORBA bridge available from IONA Technologies in May 1995 in version 1.3.4 of their Orbix<sup>TM3</sup> product.

As more companies developed bridging software, the OMG, in cooperation with Microsoft, decided to standardize the specification of a bridge to share the knowledge gained from these early developments and allow better interoperability between them. The first specification was ratified at the beginning of

---

<sup>1</sup> MQSeries is a Message Oriented Middleware (MOM) package available from IBM. It is a message queuing system used for asynchronous delivery of messages.

<sup>2</sup> OrbixTalk is IONA Technologies messaging support service for distributed systems. See <http://www.iona.com/products/messaging/index.html>

<sup>3</sup> Orbix<sup>TM</sup> is a CORBA-compliant ORB (Object Request Broker) which is available on a wide range of platforms. (For details, check out <http://www.iona.com>.)

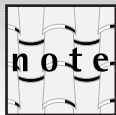
1996, with the release of the “OMG COM/CORBA Interworking Specification, Part A.”<sup>4</sup> The Part A specification outlines a complete description of what it means to bridge between CORBA and COM, and includes all the details necessary to determine if a product correctly addresses the problem.<sup>5</sup>

## Types of Bridging

One thing the OMG specification specifically does not cover is the actual implementation of a COM/CORBA bridge, or the approach to bridging which should be taken. In these respects, there are two main camps, pictured in Figure 4-1.

**Static bridging.** This provides statically generated marshalling code to make the actual call between the object systems. Separate code is needed for each interface that is to be exposed to the other object system. Static bridging also implies that there is an interface-specific package (DLLs, configuration files, etc.) which needs to be deployed with the client application.

**Dynamic bridging.** This provides a single point of access between the two systems which all calls go through. No new marshalling code is required to expose each new interface to the other object system.

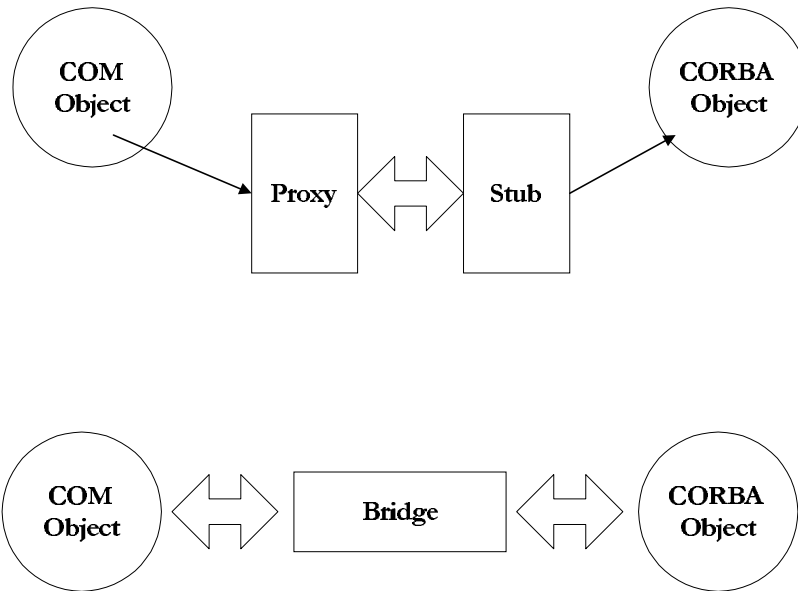


Another approach that has been considered is to provide a low-level protocol bridge. In reality, the level of complexity required to achieve this means it is not a viable option for large systems. There is just not enough semantic information available at that level to make intelligent decisions about any complex data types that are being transferred. Once this extra level of semantics has been added, you end up at much the same level as something like DCOM or CORBA.

In either case, as can be seen from Figure 4-2, a *proxy* is needed on the client side to intercept and pass on the call to the remote machine, with a *stub* on the server side to receive it. (Of course, if the call is being made in-process, it will occur directly between the calling object and the target object, with no proxy or stub required.) Hence, all that is required to provide a bridge between CORBA and DCOM is to provide something which “looks” like it belongs to the current object system but in fact is just passing the calls between one object system and the other.

<sup>4</sup> Part B of the specification addresses the area of bridging to DCOM, which came into beta that year.

<sup>5</sup> It actually provides two mappings, one between COM and CORBA, and another between OLE Automation and CORBA. Two specifications were needed as Automation has several aspects that needed to be addressed separately. (For example, support for a limited set of types, access to only the IUnknown and IDispatch interfaces, etc.)



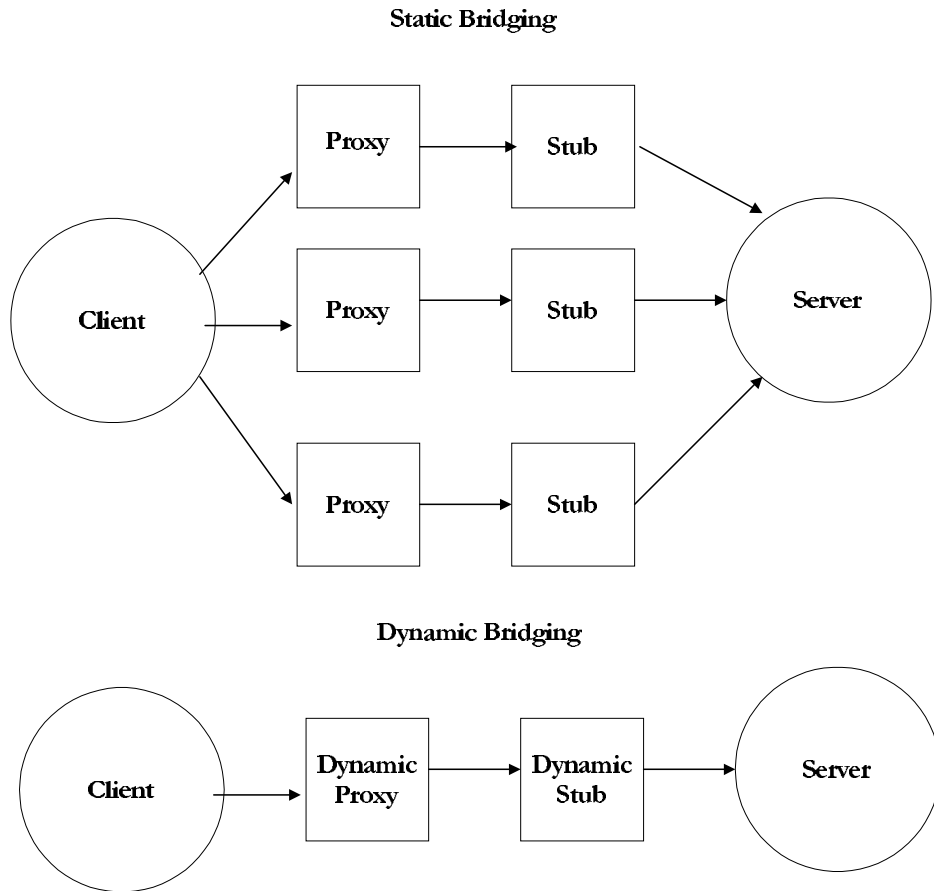
**FIGURE 4.1** Bridge Model—Something Has to Intercept the Call

The connection between two remote machines is normally *buffer-oriented*. The proxy on the client side marshals the parameters into the buffer, and the server side unmarshals them out again. This is true whichever type of bridging is used. The only difference in the two types is the piece of code that does the marshalling. The buffer which is initially passed to the server is normally referred to as the *send* buffer, while the buffer that is passed back with the operation results in it is the *reply* buffer.

### ***Static Bridging***

In this approach, a tool, usually the IDL compiler, takes a file containing a set of IDL definitions and generates static proxy and stub implementations for each interface. A proxy is a piece of code that implements the API of an interface, takes the passed parameters, and encodes them into the send buffer. This buffer is then sent across the machine boundary to the remote server process. A stub on the server side receives this send buffer, decodes the parameters it contains, and passes them on to the requested operation in the implementation object in that server. The stub then takes the results of the operation and encodes them into the reply buffer; this is then passed back to the client proxy. The proxy decodes these results from the buffer and returns them to the client code that invoked the operation in the first place.

In a static bridging approach, there is interface-specific code to handle this encoding/decoding of parameters to and from the request object.



**FIGURE 4.2** Dynamic and Static Bridging

Consider the following pseudo code example:

```
// API for sample method is:
string op1( in long val1,
           out string val2,
           inout short flags);
```

Note the “direction” attributes that indicate how the marshalling code in the proxies and stubs should handle the encoding and decoding of each parameter. Pseudo code for a client proxy would look a little like this:

```
string op1_proxy(long val1, ByRef string val2,
    ByRef short flags) {
    Request req = GetRequestObject()

    // put the parameters into the send buffer
    req.encodeLong(val1)
    req.encodeShort(flags)

    req.invoke()

    // get the results from the reply buffer
    req.decodeString(val2)
    req.decodeShort(flags)

    return req.result
}
```

Pseudo code for a stub would look a like this:

```
void op1_stub(Request ByRef req) {
    // declare parameters to make the up-call
    long v1, string v2, short v3, string result
    // find the target object
    Object obj = GetImplementationObject(req)

    // get the parameters from the send buffer
    req.decodeLong(v1)
    req.decodeShort(v3)

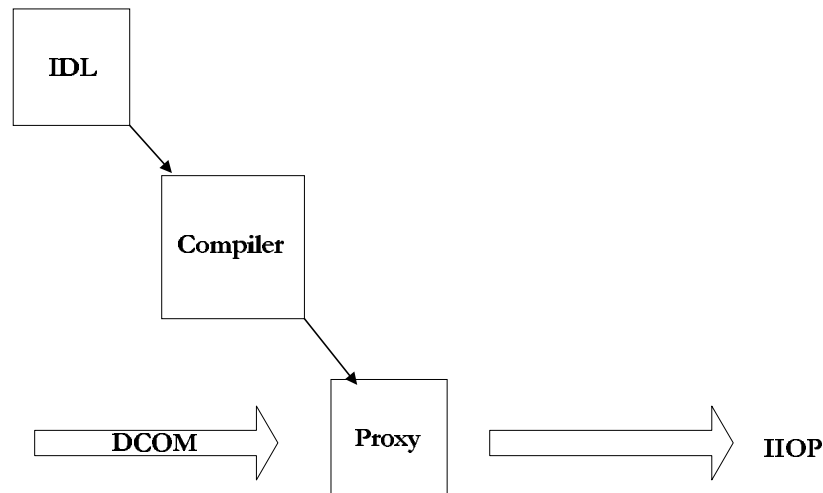
    result = obj.op1(v1, v2, v3)

    // put results into the reply buffer
    req.encodeString(v2)
    req.encodeShort(v3)
    req.encodeStringResult(result)
}
```

As can be seen here, the knowledge of how to make the call (that is, ordering or parameters, etc.) is very specific to the method being called and hence quite efficient. Using a smart request object, the identity of the target object system, DCOM or CORBA, can be hidden from the generated proxy and stub code by keeping it inside the request object implementation. That is, a bridge could present a DCOM request object that actually routes the call to a CORBA server and similarly a CORBA request object would route to a DCOM implementation.

Of the two types of bridging, static bridging is by far the easiest to implement since you have compile-time knowledge of each operation in the interface. Also, in the case of Microsoft IDL you have extra instructions, in the form of MIDL attributes, which can help with marshalling details.

The IDL compilers for both DCOM and CORBA already have the ability to generate static proxies for their own object system. All that is required to implement a static bridge between the two systems is to enhance the code generated by the IDL compiler to have it perform the necessary data type conversions, then encode/decode the parameters using the appropriate wire-level protocol encoding (CDR in the case of CORBA, NDR in the case of DCOM). As we saw in the example above, this can be easily achieved using a smart request object.



**FIGURE 4.3** A Proxy Could Bridge Between Systems

As we already mentioned, the main advantage of static bridging is that there is specific code to handle each operation call. This type of bridge is most efficient where the set of interfaces being used by the system is well-defined, and the layout of the interfaces themselves doesn't change a lot. This is especially true in small projects or pilot systems.

The main disadvantage of this approach is one of maintenance. As we all know, developing the application is really only the first step. It will change over time and, with a static approach to bridging, this means that as the interfaces change, or new interfaces are added, the bridging code needs to be updated.



Interfaces in CORBA are not immutable as they are in COM. That is, in CORBA, it is normal to upgrade a CORBA interface without changing its repository identifier, while in COM it is normal to use a different interface identifier (IID) every time an interface definition is modified.

In reality, maintenance is only a problem when you are required to install the bridging software on the client side of the connection, where hundreds, perhaps thousands of client machines may have to be upgraded. While these situations are not the norm, they can arise. For example, imagine you are required to support Windows 3.11 or NT 3.51 as your client side (platforms where DCOM is not available). In this case, CORBA is the only distribution mechanism available. So the only way to allow unmodified COM applications to have direct access to CORBA servers would be to put the bridging code on the client machines.

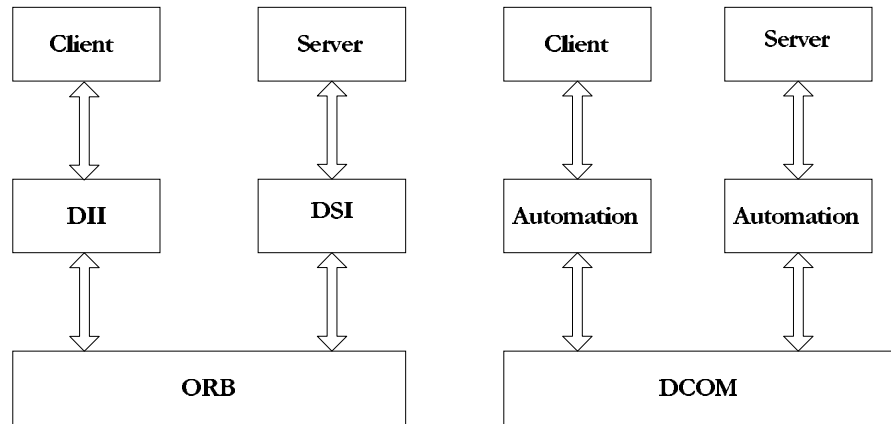
### ***Dynamic Bridging***

Dynamic bridging solves many the problems which hamper static bridging, but as is often the case in these situations, it introduces it own sets of requirements. With dynamic bridging comes the need for generic proxy and stub code capable of handling any method call to any object. The only solution to this is to make use of a dynamic mechanism to build up the request object, transfer it to the remote machine, and have a dynamic mechanism to generically receive this request and route it to the correct implementation object.

CORBA already has the ability to dynamically handle operation calls to interfaces that are only discovered at runtime. It is called the *Dynamic Invocation Interface* (DII) on the client side and the *Dynamic Skeleton Interface* (DSI) on the server side, as shown in Figure 4-4. In fact, in a lot of CORBA ORBs currently available, this mechanism forms the basic building blocks that the statically generated proxies and stubs use. At the time of writing, DCOM only has support for dynamic operation invocation using OLE Automation. While useful, it has the restriction of not supporting complex types such as custom interfaces, user-defined structures, unions, or normal unsigned integers. These dynamic mechanisms for building and invoking remote requests are used mostly by scripting languages, which want to provide access to remote servers.

With the loss of implicit type information in the static proxies/stubs comes the need for runtime type information. Both DCOM and CORBA provide this: DCOM through the system registry and type libraries, and CORBA through its interface repository (IFR).

These dynamic mechanisms work by using the runtime type information for the interfaces to drive the marshalling process. The obvious implication of



**FIGURE 4.4** Dynamic Invocation Mechanisms in CORBA and Automation

this is that the actual task of marshalling and unmarshalling the parameters to the request buffer will be a lot slower than in the static case. This is caused by the need to dynamically look up the type information as the call is being made. The runtime cost of the type information lookup can be reduced dramatically by caching the type information and re-using it for subsequent calls. This caching of type information is explored in the next chapter, “Metatype Information.” Again, consider the psuedo code example from before:

```
// API for method is:
string opl( in long val1,
           out string val2,
           inout short flags);
```

Pseudo code for a generic proxy to handle this would look a little like this:

```
any generic_proxy(string opName, ParameterList args) {

    TypeInformation typeInfo = lookupTypeInfo(opName)
    Request req = CreateRequestObject(typeInfo)

    req.encodeParameters(args)
    req.invoke()
    req.decodeParameters(args)

    return req.result
}
```

Pseudo code for a generic server stub would look a little like this:

```
void generic_stub(Request ByRef req) {  
  
    TypeInfo typeInfo = lookupTypeInfo(req.opName)  
    Object obj = GetImplementation(req.targetObjectID)  
  
    ParameterList params = BuildParamStruct(typeInfo, req)  
    req.result = obj.dispatchMethod(params)  
    req.decodeParameters(typeInfo, args)  
}
```

The basic component of any dynamic bridge is the *dynamic marshaller*. This is the component that uses the type information to ensure that the view of each object it manages is consistent with what is expected by the calling object system. This task is complicated by the need for on-the-fly construction of object references<sup>6</sup> for both object systems.

An on-the-fly object reference will be required for each parameter being passed between the object systems that is an object instance. In this case, the bridge will need to manufacture, or re-use, an object reference for the target object system (this is complicated in DCOM because all objects are transient, that is, they contain machine- and thread-specific information, which the bridge will need to provide).

To be most useful to the widest possible deployment combinations, a dynamic bridge should be capable of being driven from either CORBA type information or DCOM type information. A bridge which has this feature will limit any implicit dependency on either of the object systems.

Where a dynamic bridge comes into its own is in particularly large systems, involving thousands of interfaces, where the rate of development and even deployment of the various pieces of the system is constantly in flux. With a dynamic bridge, there is no need to update the bridge software. Indeed, by allowing the placement of the bridge component to be specified during the deployment of the distributed component, the optimum choice of wire protocol between node (DCOM or CORBA) can be achieved. The location of the bridging software can also be driven by the cost of deployment. For example, DCOM is free for NT4 and Windows 95 and 98 machines, so DCOM could be used to communicate from those machines to the bridge installed on the server machine.

Another significant advantage of a dynamic bridge is that the in-memory footprint, and hence the resource requirement, of processes is significantly reduced by the absence of compiled-in static proxy/stub code for each interface in use. This can make a huge difference for an application that needs to use large numbers of interfaces.

---

<sup>6</sup> An object reference is something that an object system uses to uniquely identify an instance of a given object.