In July we created a simple example. There was not much to it; a single method `add()` that took two IDL longs and returned a long. One of the problems with teaching and learning CORBA is that it gets complex right from the start, given that it's based on a distribution of clients and servers. Immediately we have to deal with the network. So, let's deal with the network right now.

The network

When we dig a little deeper, we see that there is a great deal going on but that it's straightforward. Our simple calculator server is designed to be called remotely and CORBA specializes in ensuring that we don't have to worry about all those differences between the environment of the client and the server. This remote call by the client to the server has its roots in Remote Procedure Call (RPC) protocol, which has been around since the 1980s. RPC is the result of years of testing with various communication models -- this is just tried and true technology that has been tested in production environments. The CORBA model we are using is based on this model.

Figure 1. The network

The interoperable object reference (IOR)

Let's track our method call. The client must first get what it thinks is an instance of our calculator. It does this by using the `calculatorHelper.java.narrow()` method. The `ior` is a string representation of an Interoperable Object Reference (IOR) that we have retrieved from a file -- `calcref.ior`. This file was written by our server as a way for clients to locate and connect to it.
The method call on the orb\string_to_object() will simply take our ior string and convert it to an object reference. Here is the code from our client, SimpleCalcClient.java:

calculator calc = calculatorHelper.narrow(orb.string_to_object(ior));
System.out.println("Calling into the server");
System.out.println( calc.add(2,3) );

What is in that IOR? The IOR is a data structure providing information on the type, protocol support, and available ORB services. The IOR is created, used, and maintained by the ORB. Many ORB vendors supply a utility to peer inside an IOR. OOC’s (Object Oriented Concepts, Inc.) Orbacus (see Related topics) comes with IORDump.exe, and, if you are using Visibroker, it has PrintIOR.exe. There are also Web sites that will parse your IOR for you; one that I use can be found at the Xerox Parc site (see Related topics). Because I am using Orbacus I will run IORDump against the IOR we created in the SimpleCalc example. I get the following output:

Embedded within the IOR is the type_id, IIOP version, the host address and port number, as well as an object key. The type_id string is the interface type also known as the repository ID format. Essentially, a repository ID is a unique identifier for an interface. This ID could be in the DCE UUID format (which COM programmers are familiar with) or in a local format that you specify. The IIOP version will help the readers of the IOR (usually ORBs) know exactly which format our IOR is in because the OMG is always improving the specifications and each version may need to be read a byte differently ;-) than the previous version. The host address and port number will get us to the ORB that will communicate with the object we desire. The object key and much of the other stuff are built around OMG standards for service-specific information. This is service-specific data that will help the ORB support the server. For example, these proprietary IOR components could encode the ORB type and version or help support an ORB implementation of the OMG Security
Service. Much of the information above specifies character code set conversions so that the client and server can understand one another.

If I run the IOR through the Xerox Parc IOR parser I get this output:

```plaintext
IIOP_ParseCDR:  byte order BigEndian,
    repository id ,
    1 profile
_IIOOP_ParseCDR:  profile 1 is 124 bytes,
    tag 0 (INTERNET),
    BigEndian byte order
(iiop.c:parse_IIOP_Profile):  bo=BigEndian,
    version=1.2,
    hostname=192.168.165.142,
    port=4545,
    object_key=<...1961005816._RootPOA......9G......>
(iiop.c:parse_IIOP_Profile): encoded object key is
<<%AC«19610058
  16%00_RootPOA
  %00%0E%0E9G
  %F8%00%00%00%00>
(iiop.c:parse_IIOP_Profile): non-native cinfo is object key is
<#AB#AC#AB196100
  5816#00_RootPOA
  #00#00#CA#FE#BA
  #BE9G#CB#F8#00#
  #00#00#00>
no trustworthy most-specific-type info;
unrecognized ORB;
reachable with IIOP 1.2 at host "192.168.165.142", port 4545
```

The most important parts of the IOR are those that will help a client connect to the server. Those parts can be seen in the output from the Xerox Parc IOR reader (see Related topics). However, much of the other information is proprietary to Orbacus, and other readers of the IOR are not able to decipher it. These proprietary parts appear as sequences of data attached to the IOR and only the ORB that built the IOR will be able to understand the data.

**The stub**

We now know what comes with the IOR. The goal of the IOR is to enable the client to make a connection to the server so that it can complete a method call. The client must convert the IOR to a realistic object with the Add method that it can call to. This is done using two of the Java files produced from our IDL compiler. The client will first use the calculatorHelper object to narrow our IOR into a _calculatorStub proxy object.

Here is the `narrow()` method generated by the jidl compiler the comes with Orbacus:
public static calculator narrow(org.omg.CORBA.Object _ob_v) {
    if(_ob_v != null) {
        try {
            return (calculator)_ob_v;
        } catch(ClassCastException ex) {
        }
        if(_ob_v._is_a(id())) {
            org.omg.CORBA.portable.ObjectImpl _ob_impl;
            _calculatorStub _ob_stub = new _calculatorStub();
            _ob_impl = (org.omg.CORBA.portable.ObjectImpl)_ob_v;
            _ob_stub._set_delegate(_ob_impl._get_delegate());
            return _ob_stub;
        }
        throw new org.omg.CORBA.BAD_PARAM();
    }
    return null;
}

As you can see, its most important task is to create a new _calculatorStub object. The _calculatorStub acts as a proxy object for the real calculator object that resides over on the server. If you don't know about the proxy pattern, I will vigorously refer you to the "gang of four" Design Patterns book (see Related topics). The proxy pattern is really nothing more than the creation of an object that represents or acts as a substitute for another real object that will eventually complete the call or perform the service. The proxy pattern is an important and often used pattern. It is used throughout all distributed designs. I'm willing to bet you've used this pattern in some form or another, but may never have referred to your design as a proxy pattern.

Once the _calculatorStub is created, it represents the calculator interface for our client. The add method is implemented within our server running in cyberspace at the address defined in the IOR. At this point it is the add() method that gets called. Two points are important to note here: first, we call the add method, so that must be present in some form in the _calculatorStub. Second, notice that our client will be blocked until the call returns, just like any other synchronous method call. This is a request-response protocol and it mimics a single process application. Programming the client and then executing the client using this request-response protocol will seem as normal and natural as the usual programming development environment created with libraries and API calls. This does not mean that there won't be occasions when you want to use an asynchronous call; you can, and the capabilities to make those types of calls are certainly available. I will cover those in a later column.

**Marshalling: GIOP and CDR**

At this point in our architecture we have successfully duped the client into thinking that the service is right there with it. But it is not, and in the next few steps of our trip we have to mold our data and method call into a form that will allow it to be carried over the network and then used on the other side. This is not trivial and the models for this effort have been around for years. You have likely seen the OSI model many times, and in Figure 2 you will see the OSI model planted next to the model that is used by the OMG.
When the client invokes an interface operation, it must send the operation data (in and inout parameters) to the server. The difficulty at this point lies in putting the data into a common format so the server can extract the operation data without misinterpreting or misaligning the data. Because our server could be on any number of different platforms, we should expect architectural differences between the client and the server. CORBA handles this by strictly defining how data is translated or marshalled into a common format. The data is then reconstituted or unmarshalled at the other end of the connection. This is done by representing the data in the most elementary of structures -- a byte stream also known as an octet stream.

The CORBA specification defines an octet stream as "an abstract notion that typically corresponds to a memory buffer that is to be sent to another process or machine over some IPC mechanism or network transport." An IDL octet maps exactly to a Java byte. They are both 8-bit values that will not be marshalled by the client or the server. The ultimate goal of streaming these parameters to a sequence of octets is to produce a basic structure for the exchange of information.

We should now peer into the _calculatorStub generated code. Keep in mind that this was not written by me -- it was generated by the IDL-to-Java compiler that came with Orbacus -- jidl.

```java
//
// IDL:corbasem/gen/calcsimpl/calculator/add:1.0
//
public int add(int _ob_a0, int _ob_a1) {
    System.out.println("Inside _calculatorStub.add()");
    while(true) {
        if(!this._is_local()) {
            org.omg.CORBA.portable.OutputStream out = null;
            org.omg.CORBA.portable.InputStream in = null;
            try {
                out = _request("add", true);
                out.write_long(_ob_a0);
                out.write_long(_ob_a1);
                in = _invoke(out);
                int _ob_r = in.read_long();
                return _ob_r;
            } catch(org.omg.CORBA.portable.RemarshalException _ob_ex) {
                continue;
            } catch(org.omg.CORBA._TranslatorFailedException _ob_ex) {
                throw new RuntimeException(
                    ",
                    new Strings[] {"_ob_ex", "_ob_ex", _ob_ex.toString()})
            }
        }
    }
```
The section to take note of is the one that contains the \_request(), write\_long() calls and the \_invoke() followed by read\_long(). The call to \_request() takes the name of the method to be called and a boolean that shows whether a response is to be expected. It returns an org.omg.CORBA.portable.OutputStream object that has been specified by the CORBA specification. This is necessary for portability because, often, Java classes are downloaded and reliant on common libraries that will be available at the location where they will be run. This is as true for ORBs as it is for IO. Therefore, the CORBA specification defines a more extensive set of portability classes for the Java language than it does for other languages.

General Inter-ORB Protocol (GIOP)

The purpose of the General Inter-ORB Protocol (GIOP) is to define the structures and formats for passing messages in this messy world of heterogeneous computers and their various architectures. If you take the structures and formats of GIOP and apply them over TCP/IP, then you have IIOP. GIOP comes in two versions: 1.0 and 1.1. This means that our messages may have different formats depending upon which version of GIOP they are conforming to.

At this point, we have to take a look at GIOP to understand the manipulations our request will go through to become a properly formatted CORBA request. Although we will be looking closely at the request, the response is simply the mirror image of the request. If you understand how the request works, you will understand the response.

The GIOP request message is broken up into three parts: the GIOP message header, the GIOP Request header, and the Request body. The GIOP message header indicates that this is, in fact, a GIOP message. It contains the GIOP version, message type, message size, and then, depending on whether you are using 1.0, 1.1 or 1.2, either it contains the byte order (GIOP 1.0) or a bit flag field that includes the byte ordering as well as some reserved bit flags. GIOP 1.1 added support for message fragmentation and GIOP 1.2 added support for bidirectional communication. The later versions are all backward-compatible with earlier versions.
Common Data Representation (CDR)

The Common Data Representation (CDR) is the formal mapping of the data types to be used in a CORBA invocation. When the client made the request, it did so without knowing where the request was going or which server would be responding to it. CORBA, as a specification, and GIOP, as the section of the specification that defines message structure and passing, are designed to allow one of possibly many different servers that implement an interface to respond to this request. The specification has to define how the data in that operation will be packaged so that all the potential servers will be able to extract the parameters and invoke the remote operations without any data translation ambiguities. The classic example to this translation problem is a pointer. What does a pointer from the client mean to a server that is running in another process on a different machine? Nothing. Or, how are arguments transmitted between machines with different addressing schemes, big endian vs. little endian? These data types have to be translated into a stream the server can understand and work with. The CORBA specification is obviously quite detailed in the area of Common Data Representation. It is a level of detail we don't need to go into here, but if you would like more information, see the specification or the book *IIOP Complete* by Ruh, Herron and Klinker (see Related topics).

Once all the data is packaged, the information from the IOR is used to create a connection. As you can tell from the structure of the IOR, this will usually entail using TCP as the transport mechanism. However, other transports may also be used (again, see the CORBA specification for details). The ORB daemon is responsible for finding the object implementation specified by the IOR and establishing a connection between the client and the server. Once that connection is made, GIOP defines a set of messages that are used either by the client for requesting, or by the server for responding. The client will send message types Request, LocateRequest, CancelRequest, Fragment, and MessageError. The server can send message types Reply, LocateReply, CloseConnection, Fragment, and MessageError.

If we tear apart a GIOP message, it looks something like this:

```
0x47 0x49 0x4f 0x50 -> GIOP, the key
0x01 0x00 -> GIOP_version
0x00 -> Byte order (big endian)
0x00 -> Message type (Request message)
0x00 0x00 0x00 0x2c -> Message size (44)
0x00 0x00 0x00 0x00 -> Service context
0x00 0x00 0x00 0x01 -> Request ID
0x01 -> Response expected
0x00 0x00 0x00 0x24 -> Object key length in octets (36)
0x00 0x00 0x00 0x04 -> Operation name length (4 octets long)
0x00 0x00 0x00 0x00 -> Value of operation name ("add")
0x20 -> Padding bytes to align next value
```

You should get the picture. This message stream is highly structured. It has to be, in order for the client to create a message the server can translate to the implementation -- no matter how or where that implementation may be running. The server must do the same for the return value
and the parameters that will be used in the response to the client. This message format is an extremely important link in the OMG effort to reach their goals of portability and interoperability. This portability will give you those liberties we talked about in the first column. You don't care about hardware or database or programming language. You just want your information.

IIOP

We're not quite finished yet. GIOP is the guts of CORBA method invocations. GIOP is not based on any particular network protocol, like IPX or TCP/IP. In order to ensure interoperability, the OMG must define GIOP on top of a specific transport that will be supported by all vendors. Interoperability is not provided when there is a detailed and compact message specification, yet all the vendors implement it using a different transport mechanism. Therefore, the OMG standardized the GIOP over the most widely used communication transport platform -- TCP/IP. GIOP plus TCP/IP equals IIOP! It's that simple.

A client that needs to use a published object's services will initiate a connection with the object, using the values in the IOR. We have now come full circle and are back at our IOR. The IOR is critical to IIOP and any client that wants to invoke a method on an object will send a Request message to the host and port address detailed in the IOR. On the host machine, the server process listens at the port for requests and, when they come in, sends those messages to the object. This then requires that the server actively listen for requests.

In the yin and yang of life, everything has its drawbacks, and interoperability and IIOP are no exception. The OMG has IIOP off and running, and it is an improvement over the era when ORB vendors did most of this stuff on their own and there was no server-side portability. But what should we do if we want our server to be location-independent? If the host and port values are embedded in the IOR, this is a problem that will crop up whenever you want to load-balance your objects by moving them from one server to another. The good news is that this problem has been solved; the bad news is that it has been solved differently by each vendor.

In conclusion

For now, load balancing will remain a future topic. If you had a CORBA "experience" a few years ago, say during an evaluation period, and are now taking another look, I think you'll be pleasantly surprised. The specification has gone to great lengths to ensure that the server code you write for one ORB will be portable to another server running a different ORB. The solutions are straightforward and based on the fundamentals of classic protocols that have been around for years. The standard exchange syntax between the client and the server is based on certain requirements detailed by the OMG. The OMG has created more leverage for its specifications by keeping the network addressing protocol (IIOP) separate from the messaging protocol (GIOP). This also ensures that as the information industry changes, which it most certainly will do, CORBA will be able to keep pace with it. What I like most about all this is that, for the CORBA object invocation we have just been talking about to complete successfully, I had no code to write! The network plumbing and marshalling details are wrapped up for us in the ORB and the power of standardizing an interface with IDL. Next month we will look at IDL.
Related topics

- Find out more about CORBA at the OMG Web site.
- Download Orbacus from Object Oriented Concepts, Inc.
- Download Visibroker for a 60-day evaluation from Inprise Corp.
- Learn about Java programming from Bruce Eckel at Mindview.net and from the second edition of Thinking in Java