OS/2 Developer

The World of Objects

Distributed Objects

BY ROGER SESSIONS

Let's try a quick psychological test. I'll put forth a word, and I want you to say the first word that comes to mind. Are you ready? Here's the word: SOM. Most likely, this word elicited one of three reactions: perplexity, a better object model, or a better C++.

SOM technology was first introduced with OS/2 2.0. The first-ever article about SOM appeared in the 1992 winter issue of *OS/2 Developer* (p.107), written by Nurcan Coskun and myself. Over the following three years, SOM was imbued with a variety of new features, including one that should have guaranteed it a place in the annals of software history.

This critical feature added to SOM 2.0 was the ability to distribute objects. SOM was the first serious commercial implementation of the Object Management Group's CORBA model, which defined an industry standard architecture for distributing objects.

The fact that so few people think of object distribution when they hear the word SOM, is a sad testimony to IBM's marketing ability or lack thereof. IBM has consistently failed to realize the importance of object distribution and has given a muddled story of what SOM is all about.

This reality is SOM's gloomy past. Fortunately, there is some bright news on the horizon. The upcoming release of SOM has been redesigned, with object distribution as the focal point of the release. If IBM can figure out how to market this capability, with half the aplomb that Sun has shown toward marketing Java (a big "if"), we might have a real winner.

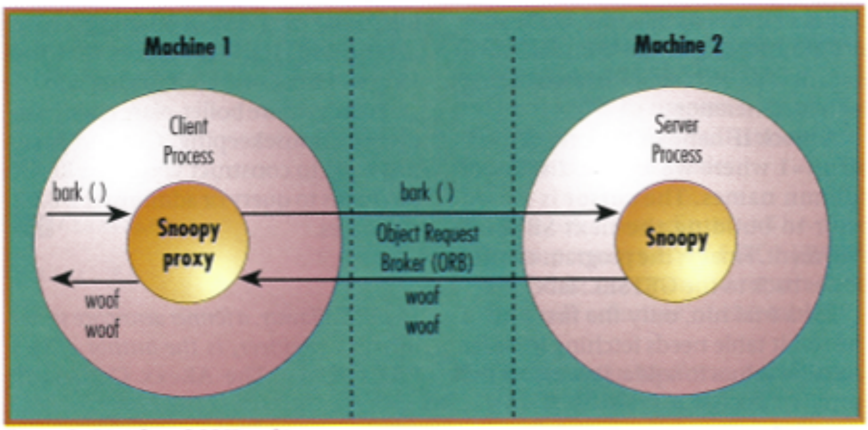

Figure 1: Distributed SOM architecture.

In this column, let's introduce writing distributed objects with SOM. All of the code examples work with the current 2.1 release. To simplify the examples and concepts, we'll use the C language bindings for SOM.

Distributed object applications

Distributed object technology is important, because it's the most natural mechanism we have for doing distributed programming. If we are object-oriented programmers, then we already know about combining behavior and data into little packages that we call objects. With object distribution, we now have the ability to take these little packages and move them to other processes.

From the program's point of view, no difference exists between using local or remote objects. In either case, we'll have an object. That object contains state or data. We interact with the object through well-defined method invocations. Because the state of an object is only of internal concern to the object, we don't care where that state actually resides. Our code works fine, as long as the invoked methods can find the right target object, regardless of whether that object lives in our address space or in another.

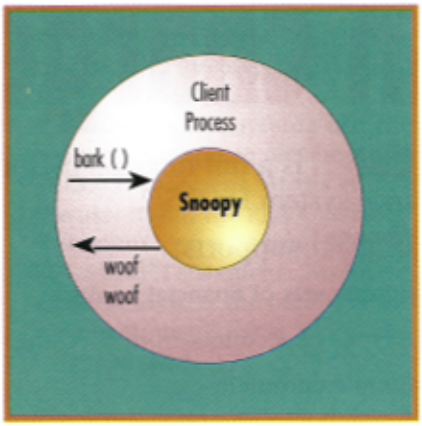

Figure 2: Distributed SOM architecture as seen by client.

It's similar to the concept of remote procedure calls (RPC), which is probably the most widely recognized paradigm for distributed programming. However, procedure calls can only be distributed when procedures have been carefully designed with distribution in mind. Objects are naturally encapsulated, so any well-designed object is a good candidate for being a unit of distribution. Besides, object-oriented programming has a host of advantages over procedural programming.

Any business that organizes its activities over a network of computers is a natural for distributred object technology, especially if these businesses are also under pressure to rapidly respond to changes in the marketplace. Prime examples include banks, insurance companies, and retail outlets, among many others.

Basic SOM definitions

Let's take a closer technical look at this technology. We start by looking at some distributed SOM definitions. Distributed SOM is often referred to as DSOM, but in SOM 3.0 the distinction between SOM and DSOM will likely be de-emphasized.

Object: An object is just that, an object; it knows how to respond to method invocations. If it's a dog object, say Snoopy, it knows how to respond to the bark method.

Proxy: A proxy looks similar to a traditional object, but is just a front for an object. It responds to the same methods as traditional objects, but does so by forwarding the invocation to some actual object. If it's a dog-proxy object, say the Snoopy-proxy, it implements bark by interacting with SOM to forward the bark invocation to the actual Snoopy object. From the client's perspective, the Snoopy-proxy appears to actually be Snoopy. Only SOM knows the difference.

Server Process: All objects live in some process running on some machine. The process in which the actual Snoopy lives is called the server process for Snoopy, or sometimes just the Snoopy Server.

Client Process: All method invocations originate from some process. The process, from which a particular bark invocation originates, is considered the client process for that particular method invocation. Many different client processes can be invoking methods on a given Snoopy.

Object References: A proxy can be turned into an object reference, which can be used to create a new proxy. The new proxy will then be a front for the same object as the original proxy. For example, if we create an object reference from Snoopy-Proxy and then generate a new proxy from that object reference, say Snoopy-proxy2, both Snoopy-proxy and Snoopy-proxy2 will pass their method calls through to the same object, namely, Snoopy. Object references take the physical form of a standard C/C++ string.

Object Request Broker (ORBs): The ORB is an underlying distribution mechanism used to pass information between proxies and objects. It's not visible to the client or object code.

SOMD_OBJECTMGR: **SOMD_ObjectMgr** is a global object available to any distributed SOM process. This global object knows how to, among other things, instantiate objects remotely, create object references from proxies, and create proxies from object references.

Listing 1: C implementation of dog.

```
#ifndef SOM_Module_dog_Source
#define SOM_Module_dog_Source
#endif
#define dog_Class_Source

#include "dog.ih"

SOM_Scope void  SOMLINK setBark(
dog somSelf,
Environment *ev,
string newBark)
{
   dogData *somThis = dogGetData(somSelf);
   _myBark = (string)
        SOMMalloc(strlen(newBark)+1);
   strcpy(_myBark, newBark);

}
SOM_Scope string  SOMLINK bark(
dog somSelf,
Environment *ev)
{
   string returnString;
   dogData *somThis = dogGetData(somSelf);

   returnString = (string)
     SOMMalloc(strlen(_myBark) + 1);
   strcpy(returnString, _myBark);
   return returnString;
}
```

Listing 2: Instantiating Snoopy.

```
#include <somd.h>
#include <dog.h>
#include <stdio.h>

void main()
{
  dog snoopy;
  string IDString;
  FILE *idFile;
  Environment *ev = SOM_CreateLocalEnvironment();

  SOMD_Init(ev);

  snoopy = _somdNewObject(
           SOMD_ObjectMgr, ev, "dog", "");

  IDString = _somdGetIdFromObject(
           SOMD_ObjectMgr, ev, snoopy);

  idFile = fopen("id.dat", "w");
  fprintf(idFile, "%s", IDString);
  fclose(idFile);

  _somdReleaseObject(SOMD_ObjectMgr, ev, snoopy);

  SOMD_Uninit(ev);
  SOM_DestroyLocalEnvironment(ev);
}
```

The architectural relationship between these components is shown in Figure 1. We have a lot of flexibility in configuring client and server processes. For example, they could both be running in a single machine, they could be on two separate OS/2 boxes, or the client could be running on an OS/2 box and the server on an MVS, AIX, or AS/400 box—a configuration particularly likely when the objects must interact with corporate databases.

The architecture, as seen by the client, is much simpler. Clients don't see proxies, ORBs, Servers, or remote machines. From the client perspective, the architecture is as shown in Figure 2.

Sample code

Let's look at one of our standard dogs. The dog definition, *dog.idl*, is:

```
#include <somobj.idl>

 string bark();
  void setBark(
     in string newBark);
  implementation {
    string myBark;
    dllname = "dog.dll";
   };
};
```

This example defines a dog that responds to two methods: **bark**, which returns a string containing the dog's bark, and **setBark**, which is used to tell the dog what its bark is.

The C code implementing these methods is shown in Listing 1. Notice that nothing exists in the dog implementation that knows whether or not the dog is going to be distributed.

The first will instantiate a Snoopy object, the second will tell Snoopy what his bark is, and the third will ask Snoopy to bark. These three different programs are running in three different processes, and the Snoopy object is in a fourth.

In Listing 2, Line 1 includes the distributed SOM header file. Line 12 initializes the distributed SOM run time. Line 14 asks the **MD_ ObjectMgr** to instantiate a new remote object by typing "dog." Although the client believes a "dog" has been returned, what has actually been returned is a proxy to the remotely instantiated dog. Line 16 creates an object reference for Snoopy. Line 17 to19 opens a file by the name of *id.dat* and writes into it Snoopy's object reference. Line 21 tells **SOMD_ObjectMgr** that the proxy is no longer needed. Lines 23 and 24 close down the system.

When the program completes, this client process is no longer active. However, a remote server process is still running, which contains a live Snoopy object that is ready to accept method invocations.

In Listing 3, Snoopy is told what his bark is in Program 2. With Listing 3, many of these lines are obvious by comparison to Program 1. Lines 15 to 17 open the *id.dat* file and read in the object reference. Using shared files is the standard SOM 2.1 out-of-the-box mechanism for sharing object references. SOM 3.0 is expected to introduce a naming service, which will allow processes to share object references via well-known names. It's not difficult to implement a naming-type service using SOM 2.1, but it does require some advanced SOM programming knowledge.

Line 19 asks the **SOMD_ObjectMgr** to create a proxy from the object reference. Line 21 tells Snoopy his bark. Of course, it's actually the Snoopy-proxy that is told, and this proxy passes the information on to the actual Snoopy. Line 23 to 25 releases the proxy and cleans up.

Listing 3: Setting Snoopy's bark.

```
1.  #include <somd.h>
2.  #include <dog.h>
3.  #include <stdio.h>
4.
5.  void main()
6.  {
7.    dog snoopy;
8.    char IDString[200];
9.    FILE *idFile;
10.   boolean done = FALSE;
11.   Environment *ev =
        SOM_CreateLocalEnvironment();
12.
13.   SOMD_Init(ev);
14.
15.   idFile = fopen("id.dat", "r");
16.   fscanf(idFile, "%s", &IDString[0]);
17.   fclose(idFile);
18.
19.   snoopy = _somdGetObjectFromId(
               SOMD_ObjectMgr, ev, IDString);
20.
21.   _setBark(snoopy, ev, "Woof Woof");
22.
23.   _somdReleaseObject(
        SOMD_ObjectMgr, ev, snoopy);
24.   SOMD_Uninit(ev);
25.   SOM_DestroyLocalEnvironment(ev);
26.
27. }
```

Listing 4: Asking Snoopy to bark.

```
1.  #include <somd.h>
2.  #include <dog.h>
3.  #include <stdio.h>
4.
5.  void main()
6.  {
7.    dog snoopy;
8.    char IDString[200];
9.    boolean done = FALSE;
10.   FILE *idFile;
11.   Environment *ev = SOM_CreateLocalEnvironment();
12.
13.   SOMD_Init(ev);
14.
15.   idFile = fopen("id.dat", "r");
16.   fscanf(idFile, "%s", IDString);
17.   fclose(idFile);
18.   snoopy = _somdGetObjectFromId(
        SOMD_ObjectMgr, ev, IDString);
19.
20.   printf("Snoppy says %s\n", _bark(snoopy, ev));
21.
22.   _somdReleaseObject(SOMD_ObjectMgr, ev,
        snoopy);
23.   SOMD_Uninit(ev);
24.   SOM_DestroyLocalEnvironment(ev);
25.
26. }
```

Most of the remote-specific code has to deal with either creating or freeing object proxies. This code is expected to be simplified in SOM 3.0 through a mechanism described as local/remote transparency, meaning that the same client code will work for both local and remote objects. Again, programming techniques are available for local/remote transparency even in SOM 2.1, but they require advanced programming abilities.

When Program 2 starts, it starts as a new client process, independent of the client process from which Program 1 ran. When this program completes, it then exits. The server process containing Snoopy, who now thinks his bark is "Woof Woof," is still running.

The program in Listing 4 is very similar to Program 2 (Listing 3). The only significant difference is line 20, which asks Snoopy (actually the Snoopy-proxy) to bark. The result of this program, which demonstrates how the Snoopy Server has evolved its state as the three different client processes have come and gone, looks like the following:

```
Snoopy says Woof Woof.
```

Reflection

We have looked at a simple example that was meant to demonstrate the manipulation of remote objects. Similar mechanisms can be used for highly complex objects representing, say, store inventory, customers, or bank accounts.

The main difficulty in programming remote objects has to do with the proxy. As we discussed, this difficulty can be ameliorated using advanced programming techniques and is expected to be simplified in 3.0, although the code shown here will still work.

Distributed object technology allows flexible, sophisticated, and distributed applications to be built quickly. With the rapid growth of the Internet, the demand for highly interactive distributed applications is expected to grow rapidly. SOM, with its ubiquitous presence on the IBM product line and beyond, is well positioned to be the major contender in this field.

To date, IBM has been unfocused in the object field as new products come in from every direction with no obvious coordinated overall strategy. Even within the SOM products line, there appear to be multiple directions. If SOM is to be successful, IBM must get focused. It needs to realize that it can't do everything. IBM has great technology for doing object distribution. Object distribution is an important field. Now is the time for IBM to turn its technical strengths into marketing triumphs.

Roger Sessions is president of ObjectWatch Inc., a company specializing in training and consulting in the use of CORBA technologies on IBM platforms. He has spoken at over 30 conferences and has written extensively. His books include Object Persistence: Beyond Object-Oriented Databases, Class Construction in C and C++; Object-Oriented Fundamentals, *and* Reusable Data Structures for C. *Roger also publishes an Internet newsletter called* ObjectWatch on SOM *and can be contacted via e-mail at roger@fc.net.*