NXApp, Winter 1993 (Volume 1, Issue 1).

Spreading the Wealth: DO and PDO

written by **Dennis Gentry**

The Portable Distributed Objects system (PDO) is a powerful subset of NEXTSTEP technology. It's an extension of Distributed Objects (DO) and is part of the NEXTSTEP development environment. Distributed Objects and Portable Distributed Objects enable developers to efficiently construct, operate, and maintain complex client/server applications in a heterogenous computing environment.

What happens when more people need to use your application than you had initially planned, so that you need to split the processing load across several computers? Or, you want to use NEXTSTEP to build the user interface to a database application, but the database server runs on an HP® server? Or maybe your company needs you to build a groupware application that lets people work together interactively?

What to do? Why, use Distributed Objects and Portable Distributed Objects, of course!

See the Distributed Objects chapter of the *NEXTSTEP 3.2 General Reference* for more details on both DO and PDO.

SHARE AND SHARE ALIKE

The Distributed Object system provides a way to share objects among multiple client and server applications running on separate computers on a network. The *server* application is a collection
of objects that are intended for use by cooperating *client* applications. The server publishes

some
of its objects to make them available to client applications on the same computer and other computers on the network. To the clients, the published objects are messaged as if they were in the same process as the rest of the client. This transparent messaging is much cleaner than previous remote procedure call (RPC) mechanisms. DO preserves the power and benefits of object-oriented programming, even in a distributed application environment.

The Portable Distributed Object system extends the power of Distributed Objects to non-NEXTSTEP computers. It allows a core section of the NEXTSTEP environment to run on other systems. Objects in the PDO environment can communicate over networks with other Portable Distributed Objects and NEXTSTEP objects. The PDO system includes all the parts of NEXTSTEP necessary to run distributed object servers plus some additional common functionality, like NEXTSTEP's file stream functions and portable BuildServer.

GRIEF-FREE CLIENT/SERVER DEVELOPMENT

Compared to other popular RPCs like Sun RPC and Mach RPC, DO and PDO have a number of advantages that make developing with them nearly transparent. They allow you to cleanly design client/server application architectures without the hassles that come with other RPC mechanisms.

Dynamic and simple

One major advantage of DO over previous RPCs is that DO is dynamic. Other RPC systems require you to specify the exact procedures that you'll call remotely. Likewise, they require you to indicate the exact types and sizes of the arguments and return values. When you add a procedure to your RPC project's list of remotely callable procedures you must recompile all affected code on the server and the client. In contrast, DO allows you to send messages to objects that don't exist or haven't even been defined. If a new Distributed Object server implements and exports an object that conforms to some protocol, previously running clients that use that protocol can begin using the new object immediately.

Another advantage is that DO frees you from many memory management concerns. You can't completely ignore memory management because there's no automatic garbage collection in NEXTSTEP. However, if you're just sending and receiving parameters and return values, you generally don't need to explicitly deal with memory as you would with other RPC systems.

Divide and conquer

With some RPC systems, you must always be (painfully) aware that you're writing an RPC program before you start. If your existing single-machine code was not written with RPC in

mind and you later need to scale up your application as your business grows, you'll have to rewrite and extend your program to distribute it across multiple machines. If you're concerned about decent performance with your RPC application, you have even more work to do.

In contrast, you can often take a non-distributed NEXTSTEP application and make it distributed with little trouble. The NEXTSTEP application should already be composed of objects, and distributing your application might involve merely identifying the relevant objects and moving them to a server program.

DO and PDO also benefit from the advantages of object-oriented programming over procedural programming. Because your application is made up of objects, and because of the encapsulation properties of objects, your application will probably be made up of neatly self-contained computational units from the start. These can often be relatively easily distributed across multiple machines because of their clean interfaces to other objects, and they should have reasonable performance in a distributed environment due to locality of reference.

Accessible servers

PDO allows non-NEXTSTEP operating systems to take advantage of Distributed Objects, so that the power and benefits of object-oriented programming and NEXTSTEP are available in a heterogenous distributed environment. PDO allows greater reusability of custom objects developed under NEXTSTEP and doesn't require additional software on NEXTSTEP clients or servers. It lets you vend and use objects remotely as either clients or servers, even on machines that aren't running NEXTSTEP. As a result, you can take advantage of NEXTSTEP's user interface capabilities while using existing server resources.

To find out more about the advantages of object-oriented programming, see ªAn Informal Approach to Object-Oriented Designº in this issue.

CHOOSING BETWEEN DO AND PDO

Ordinarily, most programmers would probably choose to use DO instead of PDO because DO runs under the full NEXTSTEP environment and is therefore more powerful, not to mention simpler to use. For example, the full Application Kit™ is available under NEXTSTEP, but not under PDO. Also, some PDO operating systems don't have the functionality to support preemptive threads that you may need to build your server. (The Distributed Object Event Loop comes with PDO to work around this limitation).

However, in some situations you might consider using Portable Distributed Objects rather than Distributed Objects to build a server for your application:

- A central machine must service many requests.

Your applications have occasional compute- or memory-intensive requests, or need a fail-safe or easily recoverable server.

A non-NEXTSTEP machine is already set up to parcel out a centralized data feed.

You'd like to take advantage of your heterogenous network to perform tasks in parallel.

If you don't have one or more of these requirements, you might find a NEXTSTEP-based DO server more convenient than a PDO server. If your site outgrows your NEXTSTEP server, it's relatively easy to move your server to a Portable Distributed Objects platform.

DISTRIBUTING OBJECTS

Applications take advantage of Distributed Objects by sending ordinary Objective C messages to objects in remote applications. The program that implements and makes an object available for remote use is called the server, and a program that takes advantage of that object by sending it messages is a client. A single application can easily play both the client and server roles.

To set up servers and clients you need to add a few additional lines of code to each cooperating application to specify which applications and objects are involved. In most cases, Distributed Objects and Portable Distributed Objects understand and neatly handle most data types as arguments or return values, including structures, pointers, strings, and, most importantly, objects (**id**s).

The server

To make an object distributed and therefore available to other applications, a server program must first vend the object. Here's a simple application that shares a central stock price data feed.

```
id myServer = [[PriceServer alloc] init];
id myConnection = [NXConnection registerRoot: myServer withName:
"stockPriceServer"];

[myConnection run]; // does not return
```

The NXConnection class provides other, more commonly used methods than **run** that allow the waiting to take place asynchronously. More on this in Multithreaded servers.º

This code instantiates a price server, then registers that server with the network name server as **stockPriceServer**. The last line loops to wait for remote messages. In each application

that will participate in Distributed Objects, you need to include the two lines of code.

The client
To use an object that has been vended, a client looks up the desired server object and stores a handle to it in a local NXProxy object. For example, this line stores the handle in theServer:

```
id theServer = [NXConnection connectToName:"stockPriceServer"];
```

If this line of code returns a non-nil value to theServer, the client may then refer to the stock price server on the server machine as if it were implemented in the client, with only a few exceptions. This is the heart of Distributed Objects. For example:

```
printf("IBM is currently at %d\n", [theServer priceFor:"IBM"]);
```

Passing objects
Probably the most important data type that clients and servers can pass to each other is the **id**. In the example above, the server explicitly vends and the client looks up only one serving object. After that, either the client or the server may pass ids of objects that each wishes to implicitly vend as arguments or return values .

The few non-transparent aspects of Distributed Objects are described in ªAvoiding Pitfalls.º

As long as the client is prepared to handle remote messages via some form of NXConnection **run** message and vends an object to the server in this way, the server may then use objects in the client. Thus, the client and server switch roles. More commonly, the original server would make additional objects available that the client would find useful, without additional setup code overhead.

For example, suppose the stock price server should return more attributes than just the price of the stock. A good way to do this is to have the server return a Stock object that the client can then query for the stock attributes. The client code might look like this:

```
id myStock = [theServer stockFor:"IBM"];
struct tm today = gmtime();
printf("IBM is currently at %d\n", [myStock priceAtTime:today]);
printf("IBM's last dividend was %d\n", [myStock dividend]);
```

Executing the first line implicitly vends a Stock object from the server, accessible through the id mystock. Each of the **printf** commands remotely invokes the stock object in the server, even though the client refers to myStock just like any local object.

Multithreaded servers
In the first example server above, the last line of the program (**[myConnection run]**) never returnsÐit just loops while waiting for incoming remote messages. In most applications a server must do more than simply service remote messages. For example, a real stock price server might also update a database from a real-time data feed. To allow a server to continue with other tasks while it also waits for messages to objects it has vended, use multiple threads. The DO system makes this very easy for Application Kit-based programs with the NXConnection method **runFromAppkit**.

Although for most applications you use Portable Distributed Objects in exactly the same way as Distributed Objects, you can't currently write multithreaded PDO servers. This is because there are no tools for threads in the HP operating system.

For example, the code from the server shown above might be enhanced to look like this:

```
id myServer = [[PriceServer alloc] init];
id myConnection = [NXConnection registerRoot: myServer withName:
"stockPriceServer"];

[myConnection runFromAppkit]; // creates a new thread that waits

/* Code to receive data feed goes here and is executed in the original thread.
*/
```

The **runFromAppkit** method creates a new thread whose sole purpose is to loop, waiting for remote method invocations. **runFromAppkit** is also aware that the Application Kit isn't thread-safe, so it waits to dispatch remote methods until your application is between Application Kit events. If your server doesn't use the Application Kit and requires finer-grained parallelism, other methods let you create threads that dispatch remote methods without waiting for the Application Kit. These methods are documented in the *NEXTSTEP General Reference* book.

AVOIDING PITFALLS
If your application is simple, like the example shown above, you'll find that using Distributed Objects and Portable Distributed Objects is pretty transparent. However, if you're building a more complex, robust application there are a few issues that you must be aware of.

Returning self has new semantics
In Objective C it's common to return the id self to indicate success of a method. This has reasonable performance for local objects, but returning self to a remote caller actually vends the object to which self refers, with all the overhead involved. Unused object vending is not excessively expensive, but for maximum efficiency objects should return a more appropriate type than self.

For example, to indicate success or failure, an object should return a scalar type such as YES or NO instead of self or nil. If the server doesn't need to return a status at all, it can return void and the method call can use the **oneway** keyword. This results in a very fast one-way asynchronous call, meaning that the caller doesn't even have to wait for the remote method to finish.

Network or remote machine failure
Make sure that cooperating programs deal gracefully with the failure of their clients or servers. The exact action an application should take depends on the nature of the cooperating programs, but DO provides a reasonably simple mechanism that allows programs to notice the loss of a cooperating program.

To be notified of the loss of a cooperating program, an object needs to request notification and implement a **senderIsInvalid:** method. When the object is asynchronously notified via this method, it must determine which remote objects have become inaccessible and decide what to do about it.

Non-transparent data types
A few data types can't be passed and returned transparently: unions, void pointers, and pointers inside structures other than ids and char *s. The basic problem with these types is that in general the compiler can't know the size of the data being referenced, so it can't pass the data to a remote program in a meaningful way. Another problem is that the computer on which the remote object is running might deal with the data differently; for example, it might use different byte-ordering. The result is that it's not possible to pass data types whose layout can't be known.

In a future version DO will manage the memory for strings like it currently does for other data types.

There are at least two ways to deal with this limitation: Type-cast pointers, or enclose complex structures in objects and then transmit the objects. You can type-cast pointers to non-recursive structures to work around the void pointer problem, and you can encapsulate

more complex structures in objects. However, if you find yourself often transmitting objects around, you might consider redesigning your application to lessen network traffic.

To transmit object copies instead of vending them, use the new **bycopy** Objective C keyword in the parameter list. Be sure to conform to the NXTransport protocol, which requires that you write three simple methods: **encodeRemotelyFor:freeAfterEncoding:isBycopy:**, **encodeUsing:**, and **decodeUsing:**. The first of these is actually implemented in the Object class. You typically override it with a simple two-line method that uses the isByCopy parameter to decide whether to send a copy of the object or not. If a copy is to be sent, the other two methods cooperate to send the data necessary to create a copy of the object at its new location: Locally **encodeUsing:** packs up the unique data of the object, and on the remote computer **decodeUsing:** unpacks it to instantiate a copy.

Memory management of strings
The current version of DO manages the memory for storing pointers to chars (strings) differently than it does for pointers to other data types. Normally, pointer data is automatically freed when the server returns; however, in the current DO, the server must explicitly free strings when it has finished with them. If you don't free strings in your servers, the memory for those strings is lost.

Performance, deadlock avoidance, and transaction management
For many Distributed Object applications you don't need to worry about optimizing performance, avoiding deadlock, or managing atomic transactions. However, for large distributed applications these issues can become very important. For example, with a larger network and more complex needs, the latent problems you might have in existing DO applications can become more apparent. This isn't all bad, because if problems are apparent you have a better chance of fixing them.

Dealing with these issues properly is beyond the scope of this article. However, consider the inherent complexity involved in writing distributed applications before beginning work on a large distributed application, rather than as an afterthought. For example, to deal with deadlock, be careful to reason about the behavior of cooperating and competing servers to make sure they can never mutually rely on the same resources at the same time in order to make progress. Likewise, to deal with managing atomic transactions, use a two-phase commit protocol.

Realities of servers and networks
When you plan to put compute-intensive tasks on a server, keep perspective on scaling issues.

For example, no current PDO server has the aggregate computing power of 500 or even 10 Pentium-based NEXTSTEP machines. Therefore, if you might eventually decentralize your application, you shouldn't plan to saturate a single central serverÐrather, consider distributing compute-intensive tasks across multiple server machines if possible. The trade-off, of course, is that it can be more difficult and time-consuming to correctly implement your computation for parallel processing.

If you do decide to distribute a task across several computers, keep in mind that the network has a finite bandwidth that can be saturated by a few high-performance machines sending remote messages extensively. Design your application to take advantage of Distributed Objects' facility for moving objects from one machine to another. This can reduce the amount of remote messaging that might otherwise occur.

CONCLUSION

DO and PDO offer you excellent tools for developing client/server applications. Their design also gives you the flexibility to expand applications as NEXTSTEP and PDO become available on more platforms. We hope you'll find they're just what you need to make great applications.

Dennis Gentry is a member of the Developer Support Team. You can reach him by e-mail at **Dennis_Gentry@next.com**.

References

Andleigh, Prabhat, and Michael Gretzinger. *Distributed Object-Oriented Data-Systems Design*. Englewood Cliffs, NJ: Prentice Hall, 1992. ISBN 0-13-174913-7.

Elmasri, Ramez, and Shamkant B. Navathe. *Fundamentals of Database Systems*. Redwood City, CA: Benjamin/Cummings, 1994. ISBN 0-8053-1748-1.

NeXT Computer, Inc. *NEXTSTEP 3.2 General Reference*, vol. II. Palo Alto, CA: Addison Wesley, 1992. ISBN 0-201-62221-1.

NeXT Computer, Inc. *NEXTSTEP 3.2 Release Notes*. Redwood City, CA: NeXT Computer, 1993.

NeXT Computer, Inc. *Object-Oriented Programming and the Objective C Language*. Palo Alto, CA: Addison Wesley, 1993.

NeXT Computer, Inc. *Portable Distributed Objects 1.0 Release Notes*. Redwood City, CA: NeXT Computer, 1993.

Next Article NeXTanswer #1504 **An Informal Approach to Object-Oriented Design**
Table of contents http://www.next.com/HotNews/Journal/NXapp/Winter1994/ContentsWinter1994.html