Distributed Applications and NeXT’s PDO

PDO brings distributed objects to non-NextStep environments

Dennis Gentry

*Dennis is a member of the developer-support team at NeXT. You can reach him by e-mail at dennis_gentry@next.com.*

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 heterogeneous 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, what happens when you want to use NextStep to build the user interface to a database application, but the database server runs on an HP server? Perhaps your company needs to build a groupware application that lets people work together interactively. DO and PDO were designed for these situations.

**Share and Share Alike**

The DO system provides a way to share objects among multiple client/server applications running on separate computers on a network—distributed over both client and server machines. In this model, the server application consists of 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 on 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 older-generation remote procedure call (RPC) mechanisms. DO preserves the power and benefits of object-oriented programming, even in a distributed application environment.

The PDO 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 PDO and DO objects. The PDO system includes all the parts of NextStep necessary to run distributed-object servers—plus some additional common functionality (such as NextStep's file-stream functions and portable BuildServer). PDO facilitates reusability of objects developed under NextStep and doesn't require additional software on NextStep clients or servers. It lets you use objects remotely as either clients or servers, even on machines that aren't running NextStep.

The components that make up PDO are:

- Objective C, which is the language used to create objects and interact with other objects.
- The Objective C run time, which are extensions to the Objective C language environment that provide for dynamic binding at run time.
- The NextStep Core Classes, which is a library that provides classes such as Object, List, and HashTable.
- The Distributed Object system, which is a set of classes that manages the brokering of server objects and provides transparent connections to remote objects in the network.
- The Transport Mechanism, which is built into the DO classes and provides an independent transport mechanism for communicating between objects over a network.
- The Portable nmserver, which is a portable version of the Mach Network Name Server used in DO to provide object naming over the network.

The first targeted platforms for PDO are mainstream UNIX-server environments such as HP-UX and Solaris. The design of PDO is not restricted to UNIX platforms, so that future support of environments such as NetWare and Windows NT is possible. With regard to the emerging CORBA standard for object interoperability, Next has submitted an Objective C binding for the IDL interface description language so that CORBA clients and servers can be written in Objective C.

**PDO/DO Advantages**

Compared to other popular RPCs (such as 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 usual hassles that come with other RPC mechanisms.

One important aspect of DO is that it 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 DO server implements and exports an object that conforms to some message 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 decide 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 nondistributed NextStep application and make it distributed with little trouble. Your NextStep application already should be composed of objects. In such a case, 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 likely be composed of well-contained computational units from the start. These units often can be relatively easily distributed across multiple machines because of their clean interfaces to other objects, and their locality of reference will foster better performance in a distributed environment.

**Choosing Between DO and PDO**

Ordinarily, most programmers probably would choose to use DO instead of PDO because DO runs under the full NextStep environment and is, therefore, more powerful (and simpler to use). For example, the full Application Kit (a general-purpose application framework) 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 PDO comes with Distributed Object Event Loop to work around this limitation.)

However, you might consider using PDO rather than DO to build a server for your application in situations such as these:

- A central machine must serve 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 heterogeneous 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 PDO platform.

**How to Distribute 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 must add a few additional lines of code to each cooperating application to specify which applications and objects are involved. Both DO and PDO can handle most data types as arguments or return values—including structures, pointers, strings, and, most importantly, objects (ids).

To make an object distributed (and, therefore, available to other applications), a server program must first "vend" (make publicly available) the object. **Example 1(a)** shows a simple application that shares a central stock-price data feed. TheNXConnection class provides other, more commonly used methods than run that allow the waiting to take place asynchronously. **Example 1(a)** instantiates a price server, then registers that server with the network name server stockPriceServer. The last line loops to wait for remote messages. You must include this code in each application that will participate in Distributed Objects.

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. **Example 1(b)** gets a handle and stores it in the variable theServer. If this 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; see **Example 1(c)**. This transparency is at the heart of Distributed Objects.

**Passing Objects**

Perhaps the most important data type that clients and servers can pass to each other is the id. In the example application, 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.

As long as the client is prepared to handle remote messages by way of some form of NXConnection-run message, and as long as the client 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 can make additional objects available that are useful to the client, without additional setup code overhead.

For example, suppose the stock price server needs to 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**Example 2**.
Multithreaded Servers

In Example 1(a), the last line of the program, [myConnection run], never returns--it just loops while waiting for incoming remote messages. However, in most applications, a server must do more than simply service remote messages. For example, a real stock-price server also might update a database from a real-time data feed. Multiple threading is the mechanism that allows a server to continue with other tasks while it also waits for messages to objects it has vended. The DO system makes multithreading very easy for programs built with the Application Kit, using the NXConnection method runFromAppkit.

Although in most applications, you can use PDO objects in exactly the same way as DO 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 DO code from the server shown previously might be enhanced to look like that in Example 3. The runFromAppkit method creates a new thread whose sole purpose is to loop, waiting for remote method invocations. The runFromAppkit method also is 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 here), you'll find that making your application distributed is fairly transparent. However, if you're building a more complex, robust application, you must be aware of a few more issues.

One is that the semantics of returning self are different. In Objective C, it's common to return the idself to indicate success of a method. While this has reasonable performance for local objects, returning self to a remote caller actually vends the object to whichself 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.

Another issue in distributed applications is how to handle a failure in the network or in remote machines. You must 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 must request notification and implement a sendersInvalid: method. When the object is asynchronously notified by way of this method, it must determine which remote objects have become inaccessible, and decide what to do about it.

Another pitfall in making an application distributed is that a few data types can't be passed and returned transparently. These types are 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.

Presently, there are at least two ways to deal with this limitation. One is to type-cast pointers, which works around the void pointer problem, as long as the pointer is to a nonrecursive structure. Another approach is to enclose complex structures into an object and transmit that object. However, if you find yourself often transmitting objects around, you might consider redesigning or repartitioning 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. 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. The encodeUsing: method runs locally and packs up the unique data of the object. On the remote computer, the decodeUsing: method unpacks the object to instantiate a copy.

With regard to strings, a nontransparency arises in the current version of DO, which 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 your server code doesn't free strings, the memory for those strings is lost. In a future version, DO will manage the memory for strings just as it currently does for other data types.

Pitfalls in Scalability

Performance, deadlock avoidance, and transaction management are issues that you don't need to worry about in smaller or simpler DO
applications. However, in large distributed applications, these issues can become very important. With a larger network that has complex needs, latent problems in your existing DO applications may then become apparent. This isn't completely bad news, because once problems are apparent, you have a better chance of fixing them.

Dealing with these advanced issues is beyond the scope of this article. However, it behooves you to consider the inherent complexity involved in writing distributed applications before you begin work on a large-scale distributed application. To deal with deadlock, you must carefully think through 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, you should consider using a two-phase commit protocol.

If you plan to put compute-intensive tasks on a server, keep an eye on scaling issues. For example, no current PDO server has the aggregate computing power of 500, or even ten Pentium-based NextStep machines. Therefore, if you might eventually decentralize your application, you shouldn't plan to saturate a single central server. Rather, consider distributing compute-intensive tasks across multiple server machines. 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 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 the facility in DO 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 excellent tools for developing client/server applications. Their design also provides the flexibility to expand applications as NextStep and PDO become available on more platforms.

References


Example 1: (a) Server code for an application that provides a stock-price data feed; (b) client code that connects to the stock-price server; (c) client code that accesses data from the server.

(a)  id myServer = [[PriceServer alloc] init];
    id myConnection = [NXConnection
registerRoot: myServer
withName: "stockPriceServer"];
    // the following statement does not return
    [myConnection run];

(b)  id theServer = [NXConnection connectToName:"stockPriceServer"];
    // the following line creates a new thread that waits
    [myConnection runFromAppkit];

Example 2: Client code that gets an object from the server and sends messages to that object.

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

Example 3: Server code that uses multiple threads.

id myServer = [[PriceServer alloc] init];
    id myConnection = [NXConnection
registerRoot: myServer
withName: "stockPriceServer"];
    // the following line creates a new thread that waits
    [myConnection runFromAppkit];
// Code to receive the data feed goes here
// and is executed in the original thread...

Copyright © 1994, Dr. Dobb's Journal