Using Distributed Objects with the Enterprise Objects Framework

OPENSTEP's Distributed Objects system provides a relatively simple way for applications to communicate with one another by allowing them to share Objective C objects, even among applications running on different machines across a network. Distributed objects are particularly useful for implementing client-server and cooperative applications.

As shipped with OPENSTEP, the Distributed Objects system allows you to pass messages and objects that descend from the Object class back and forth between applications. The Enterprise Objects Framework augments this capability, allowing you to pass messages and objects that descend from NSObject as well. Note that these added capabilities do not alter the support for working with subclasses of Object in a distributed environment; you now can pass messages and objects that descend from either root class.

Similarly, you can run the Enterprise Objects Framework access layer on a Portable Distributed Objects (PDO) system to use it as a server for database applications. Of course, you can't use any of the interface layer (such as the model builder) on a PDO system, since this depends on the OPENSTEP graphical user interface. Only a OPENSTEP system can run both layers.

The material in this document supplements that found in the Distributed Objects introduction, found in **/NextLibrary/Documentation/NextDev/Old_PDO_Reference/DistributedObjects/IntroDistObjects.pdf**.

Strategies for Using Distributed Enterprise Objects

There are a number of distinct strategies that you can employ when using the Enterprise Objects Framework in a distributed environment. In particular, two are particularly well supported: that of using a notification server, and that of using a compute server.

Using a Notification Server

A notification server is used when you have multiple clients that need to coordinate access to a database. Each client runs the Enterprise Objects Framework in its entirety: both the access and the interface layers—or just the access layer if it's a PDO system. They each work directly with the database (through the Enterprise Objects Framework), but use a notification server to ensure that they all remain synchronized.

The notification server is a separate task, running either on one of the client machines or on a separate machine altogether. Each client registers with the notification server, and then informs the notification server whenever the client makes a change to the database (typically, messages to the notification server are sent from EODatabaseChannel or EOAdaptorContext delegate methods). The notification server in turn then notifies each client of the change.

Messages are passed between the clients and the notification server using the Distributed Objects system. Because the messages need only indicate that the database has changed (perhaps also indicating which objects were affected; see "Passing Unique Identifiers," below), message traffic between applications is minimized and performance is thus enhanced.

For an example of how to construct a notification server, see the example code that is supplied with the Enterprise Objects Framework (in **/NextDeveloper/Examples/EnterpriseObjects/DistributedEO**).

Using a Compute Server

The notification server has the disadvantage of putting all of the computing burden on each client. Often, you have a high-powered compute server that you want clients to be able to take advantage of. This server might do database queries to determine the set of records that a client is interested in, or it might perform calculations on the records in the database.

When using a compute server, the server should have (at a minimum) the Enterprise Objects Framework's access layer installed. Each client would be running the access layer and, if it is capable, the interface layer.

When the client sends a request to the compute server (using the Distributed Objects system), the server interacts with the database (using the Framework's access layer methods) to determine the result. If the result is not made up of a set of objects already in the database server—if, for example, the result is a matrix that indicates the risk factors associated with a set of securities—the compute server can pass the result to the client directly. If the result is a set of enterprise objects—the set of securities that meet a particular set of risk criteria, for instance—the server can pass unique identifiers for those objects to the client, who then fetches the records directly (see "Passing Unique Identifiers," below). This allows the client to take advantage of the compute server without having to suffer the performance penalty that is incurred when passing enterprise objects between applications.

Passing Unique Identifiers

While you are free to pass enterprise objects between applications, you should consider the impact that this will have on your application's performance. The Enterprise Objects Framework is optimized to move data very rapidly between the database and the end user. By splitting your application so that a server acts as a data source and one or more clients, running the Framework's interface layer, obtain enterprise objects from that data source using the Distributed Objects system, you end up negating any performance advantage you might have gained by enlisting the added power of a separate server.

Imagine, for instance, that you have a data source running on one machine and a browser displaying the objects from that data source running on another, client, machine. When the user scrolls the browser, hundreds of messages will be sent back and forth between the client and server, all of which have to be carried by the Distributed Objects system.

A general solution to this problem is to pass not the enterprise objects themselves, but a unique identifier for each enterprise object (generally, made up of the object's entity and primary key), encoded into an NSString. When a client or server receives one of these unique identifiers, it can look for the object in its uniquing table (using EODatabase's **objectForPrimaryKey:entity:** method). If the enterprise object is not currently known to the client or server receiving the unique identifier, it can simply fetch it from the database using the methods provided by the Enterprise Objects Framework. This solution is used by both of the preceding two scenarios to minimize the number of messages that are passed between applications.

Passing unique identifiers also helps you to work around another problem that can arise when passing enterprise objects directly between applications. Suppose that you have two clients, each passing enterprise objects back and forth to a data source running on a server. Now suppose the clients each fetch the same enterprise object from the server. Each client will put that object into its uniquing table. But if one client then passes the object to the other, the receiving client will not automatically recognize that it already possesses the object and will wind up with two distinct objects that represent the same data in the database. And these two objects can rapidly get out of sync.

A unique identifier that is made up of an enterprise object's entity name and primary key, however, would have the same value on each client. When the identifier is passed from one client to the other, the receiving client can quickly ascertain that it already possesses the indicated object.

Establishing a Connection

Because of Foundation's new autorelease mechanism (which is used by all objects that inherit from NSObject), you must ensure that there's an autorelease pool in the run loop on both the server and client sides of the distributed object connection. This is easily done by registering the server using the methods supplied by NXAutoreleaseConnection (NXAutoreleaseConnection is a subclass of NXConnection that adds no new methods of its own). For example:

```
id aServer;
id aConnection;

aServer = [[Server alloc] init];
aConnection = [NXAutoreleaseConnection registerRoot:aServer
   withName:SERVER_NAME];
if (aConnection) {
   [aConnection run];
   [aServer release];
} else {
   fprintf(stderr, "Couldn't register server - exiting.\n");
    exit(-1);
}
```

In the above example, **Server** is a subclass of NSObject and **SERVER_NAME** is a C string that contains the name to be assigned to the server.

Here's the code on the client side that connects to the server:

```
id aServer = nil;
unsigned int i;

for(i = 0; i < 10; i++){
        /* in case it doesn't connect right away... */
    aServer = [NXAutoreleaseConnection connectToName:SERVER_NAME
        onHost:HOST_NAME];
    if(aServer)
       break;
    sleep(2);
}

if(!aServer){
    fprintf(stderr, "time out !\n");
    exit(1);
}
```

In the above excerpt, **SERVER_NAME** is a C string that contains the name of the server to connect to, and **HOST_NAME** is a C string that identifies the host on which the server is running.

When a client establishes a connection to an NSObject on the server, **connectToName:onHost:** returns a proxy that has been retained (not autoreleased). Once the client is finished with the object returned by **connectToName:onHost:**, it must send **release** or **autorelease** to it. Similarly, the server should release the connection object when it's no longer needed.

Note: Sending NSObjects over an NXConnection (as opposed to an NXAutoreleaseConnection) results in memory leaks.

Passing Objects

Suppose you have a method on the server named **supplyObject** that returns to the client an autoreleased object that's a subclass of NSObject:

```
- (NSObject *)supplyObject
{
    id anObject = [[anNSObjectSubclass alloc] init];
```

```
        return [anObject autorelease];
    }
```

The object on the server is retained. On the client side, the proxy is autoreleased.

Every time an NSObject is passed by reference from a server to a client, the source object's reference count is incremented. When the proxy on the client disappears (due to a **release** or **autorelease** message), the appropriate number of **release** messages are sent to the object on the server.

Restrictions on the Objects Passed

NSObjects are passed *by reference* except for the following, which are passed by copy:

* NSString
* NSNumber
* NSData

Note: NSMutableStrings on the server are passed as NSStrings to the client; on the client, they aren't mutable.

While you can share NSObjects through the use of proxies, messages to a local copy require much less overhead (and are thus quicker) than remote messages over a connection. Thus, you may want to avoid passing subclasses of NSObject (other than NSString, NSNumber, and NSData) if your application is sensitive to performance. Do not use the **bycopy** keyword to force other subclasses of NSObject to be passed by copy. If you really need this behavior, you must implement the methods in the NXTransport protocol for your custom NSObjects.

For an example that illustrates how to pass NSArray and NSDictionary objects, see the Foundation extension code that is supplied with the Enterprise Objects Framework (**FoundationExtensions.[hm]** in **/NextDeveloper/Examples/EnterpriseObjects/DistributedEO/DEOClient.subproj/ DOExtensions.subproj**. Also see /**NextLibrary/Documentation/NextDev/Old_PDO_Reference/DistributedObjects/IntroDistObjects.pdf**.

Copying Proxies

Don't send **copy** or **mutableCopy** to proxies. The proxy that you receive for the object's copy is autoreleased, which is contrary to the way that **copy** operates on local objects. Additionally, in the process of duplicating the object on the server the copy receives an extra **retain**, which ultimately results in a memory leak when you release the proxy.

Keeping Track of Clients

Often, it's handy for a server process to keep track of its current clients. You typically use an NSMutableArray on the server, storing proxies to the clients in the array. If a connection is then broken (due to the connection object being released, for instance), you remove the proxy from the array. When doing so, however, avoid using NSMutableArray's **removeObject:** method; this method uses **isEqual:**, which causes the entire array to be transferred to each client for comparison. Instead, use **removeObjectIdenticalTo:**.