6

Distributed Objects

Library: libsys_s.a

Header File Directory: /NextDeveloper/Headers/remote

Introduction

The Distributed Objects system provides a relatively simple way for applications to communicate with one another by allowing them to share Objective C objects, even amongst applications running on different machines across a network. They are useful for implementing client-server and cooperative applications. The Distributed Objects system subsumes the network aspects of typical remote procedure call (RPC) programming, and allow an application to send messages to remote objects using ordinary Objective C syntax.

The Distributed Objects system takes the form of two classes, NXConnection and NXProxy. NXConnection objects are primarily bookkeepers that manage resources passed between applications. NXProxy objects are local objects that represent remote objects. When a remote object is passed to your application, it is passed in the form of a proxy that stands in for the remote object; messages to the proxy are forwarded to the remote object, so for most intents and purposes the proxy can be treated as though it were the object itself. Note that direct access to instance variables of the remote object isn't available through the proxy.

Terminology

In this document, the terms ªserverº and ªclientº are used loosely. Whenever an object in an application is returned to a remote application, the object effectively becomes a server, capable of responding to remote messages. For this document, ªclientº refers to the object originating a remote message, and ªserverº refers the remote object responding to the message. For example, if a database server sends a remote message to a database client, from the perspective of the Distributed Objects system the database server is the client of the message.

Making an Object Available

The Distributed Objects system allows an application to send a message to an object that exists in another application. The message may include most data types (including objects) as arguments, and it may return most data types, again including objects. Clearly, no messages can be sent to a remote application until the local application has gotten a proxy to some object in the remote application. Therefore, in order to bootstrap the communication process, one or more objects must

be made available by name using the Network Name Server. Such an object is known as a *root object*, and is available to any application that knows the registered name of the object. While it is possible to have multiple objects available by name, it is also common to have just one, and to get additional proxies to remote objects in response to messages (to both the root object and objects returned by the root object).

Here is a simple example that shows how to make an instance of the MyServer class available to other applications:

```
id myServer = [[MyServer alloc] init];
id myConnection = [NXConnection registerRoot: myServer
        withName:"exampleServer"];

[myConnection run];
```

The first line creates **myServer**, the object that is to be made available to other applications. The next line registers **myServer** as a root object, available to any application that asks for the object named ªexampleServerº. This method returns an NXConnection object that will dispatch messages sent from remote objects and track resources (such as objects) vended to connecting applications. The last line tells the connection object to begin its process of waiting for messages and dispatching them to the proper receivers. The **run** method shown doesn't return, but there are variations that run the connection concurrently in another thread or pseudo-concurrently from the DPS client routines that dispatch events.

Establishing a Connection

In the example above, an instance of MyServer is made available under the name ªexampleServerº. Another application can get a proxy to the object like this:

```
id server = [NXConnection connectToName:"exampleServer"];
```

When this message is sent, a connection to the application that registered the MyServer object is established. The returned object, **server**, is a proxy to the remote MyServer object. Because **server** forwards messages across the connection to the MyServer object, it can generally be treated as though it were that object.

Connections may also be formed automatically when proxies are passed between applications. For example, imagine that two client objects (call them client A and client B) have connected to a server object. If client B sends its **id** to the server, the server gets a proxy to client B. If client A then asks the server to return client B, the server does this by returning client B, which is actually its proxy. However, client A doesn't receive a proxy to the server's proxy. Instead, a new connection is established between client A and client B, and client A receives its own direct proxy (over the new connection) to client B.

Sending Data Between Applications

The Distributed Objects system can use most data types as message arguments or return values. Here are some examples:

```
[server aSimpleMessage];            // no parameters
[server useAnInteger: 12];          // simple scalars
[server useAnIntByReference: &i];   // sending a pointer
[server useAString:"hello"];        // sending a string
[server useAnId: self];             // send an arbitrary local object
[server useAnotherId: server];      // send back the shared object!
```

In both the **useAnIntByReference:** and **useAString:** methods, a pointer is automatically dereferenced on the client side, and the resulting data is sent to the server. On the server side, space for the data is allocated, and a pointer to the local data is received. The server's allocated copy of

the data is local in scope and will be freed by the system when the server's method returns.

In the **useAnId:** method, the server is passed an **id** to a local object, and the server receives a proxy to that object. In the **useAnotherId:** method, the server is passed the client's proxy to the server. The Distributed Objects system makes sure that the correct object is returned; in this case the server receives the local **id** for itself rather than a proxy.

The Distributed Objects system allows callbacks in the midst of a method implementation. For example, the server can send a message back to the client in the midst of its **useAnId:** implementation. Such a callback doesn't deadlock, and can be useful, but its ramifications must be carefully considered. Methods in the client can be invoked by the server before the client's invocation of the **useAnId:** method returns.

Structures

The Distributed Objects system can utilize structures for both message arguments and as return values, but there are some important limitations. The following example demonstrates that complex structures can be passed as arguments in remote messages:

```
typedef struct {
   char aChar;
   int anInt;
   unsigned int bitfield:3;
   enum { red, green, blue } color;
   id anObject;
   char *aString;
   int array[2];
} exampleStruct;
exampleStruct e = {'a',9,5,green,nil,"Hello",{42,17}};

[server useStructByValue:e];
[server useStructByReference:&e];
```

In general, a structure to be used as a parameter for a remote message can't contain pointers. Pointers are only valid in one address space, so the Distributed Objects system would have to reconstruct the pointer's data on the remote end. The system can't know how deep to recurse when dereferencing pointers, so it implements the simple case and doesn't dereference pointers to most types, with two exceptions. Structures can contain pointers to objects (**id**s) and pointers to character strings. At the time a remote message is sent, these pointers must point to valid data or they must be null pointers, since the system may need to send the pointer's data across the connection in order to yield a valid pointer on the remote side.

Structures can be passed both by value and by reference. In the current implementation, however, structures can only be returned by reference. In other words, a remote method can't return a structure, but it can return a pointer to a structure. If a method returns a structure by reference, memory for the structure is allocated on the caller's side, and the caller is responsible for freeing this memory.

Pointers to Data

The Distributed Objects system can send data by reference as well as by value. Pointers used in remote messages must point to valid data or be null, since they may need to be dereferenced. By default, when you send most data types by reference, the data is copied across the connection so the server can receive a valid local pointer. The data then may or may not be modified, and is copied back across the connection so the client gets any modifications to the data. Needless copying of data is not efficient, so the Distributed Objects system adds three new Objective C keywords Ðfor use in formal protocol declarations onlyÐto determine how data passed by reference should be copied. The keywords are **in**, **out**, and **inout**. **In** arguments are copied from the client to the server, but not copied back. **Out** arguments are not sent the server, but are copied back to the

client, presumably because the server filled in a value. **Inout** arguments are copied in both directions. By default, **const** pointer arguments are treated as **in** parameters, and all other pointer arguments are treated as **inout**. Here are some example definitions showing directionality of arguments:

```
- sendAnInt: (in int *)p;
- receiveAnInt: (out int *)p;
- sendAndReceiveAnInt: (inout int *)p;
```

The system can't tell whether a pointer points to a single data item or to an array; it assumes all **char** pointers point to null-terminated strings and that all other pointers point to single data elements. If you have arrays that must be passed by reference, you might consider encapsulating the data in a custom object or using a subclass of NXData.

Memory Allocation

When you send **in** or **inout** pointer parameters to the server, the system must allocate space for the data on the server side (so that it can supply a pointer valid in the server's address space). This memory is owned by the system and is local to the scope of the server's method; it is freed automatically when the server's method returns.

The Distributed Objects system can allocate client memory for string and structure parameters. To return strings or structures in this manner, you must pass a pointer to a **char** pointer or a pointer to a structure, so that the system can allocate the memory and make the pointer point to it. If the system allocates memory to return data to the client, the client is responsible for freeing this memory. You must be careful about returning data in this manner, because you receive a pointer to an allocated copy of the data if you send the message to a remote object (through a proxy) but you receive a pointer to the data itself (as with ordinary Objective C) if you send the message to a local object. Here is an example that gets a string by sending a **char** pointer by reference, and then frees the string only if it sent the message to a remote object:

```
char *cp;
[anObject getString:&cp];
printf("The string is %s",cp);
if ([anObject isProxy]) free(cp);
```

The Distributed Objects system also allocates memory in the client's address space in order to return a pointer to a structure as a method return value. Again, the client is responsible for freeing this memory.

Types That Don't Work

The Distributed Objects system can't send the following data types:

- UnionsÐThe Distributed Objects system can't distinguish how to correctly encode the data to send it to the server.
- void *ÐThis is a generic pointer, and the system can't correctly dereference it and encode the data.
- Pointers in structures, other than those of type **char *** and **id**.

In addition, remote methods can't return data of type **double** or **struct** (though pointers to structures work). These limitations may be lifted in future implementations.

Sharing Objects

The most important data type that the Distributed Objects system can use in messages, both as

arguments and as return values, is **id**. Objects are usually passed around as proxies, which forward messages to their corresponding real objects and thus appear to be those objects.

Proxies (instances of the NXProxy class) are created automatically when an object is returned to a remote application. To give a client access to a remote object, two proxies are created, one on the server side and one on the client side. The proxy on the server side is known as a local proxy because it tracks a local resource (an object in the proxy's application). A local proxy is used for reference counting by the server's NXConnection object, and to send incoming messages to its corresponding real object. Local proxies are generally hidden from view in the Distributed Objects implementation, and most of their functionings are uninteresting to application developers. More interesting to developers are remote proxies, the objects returned to the client that can generally be treated as though they were the remote objects themselves. These objects receive messages from the client directed to the real object and forward the messages across the connection.

Consider the following code in which a client needs to access a server's list of Widget objects:

```
List *aList;
Widget *aWidget;
aList = [server widgetList];
aWidget = [aList objectAt:0];
```

In the third line, the server returns its list of widgets to the client. The List object exists in the server application, and the client gets a proxy to that List object, which is assigned to **aList**. In the fourth line, the client sends a message to **aList**, and the message is forwarded by the proxy to the actual List object in the server. The List implementation in the server returns the first Widget object in the list. Again, the Widget object is local to the server, so the client receives a proxy to the Widget.

The example above demonstrates that it is very easy to have proxies created. This is an important feature of the Distributed Objects system, but it has performance ramifications that must be considered. Consider the common case where a method in the server returns **self**. The system assumes that you actually intend to return a usable object to the client, so it will return a proxy for the server to the client. If the client's connection doesn't already have a proxy to the server, one will be created. This may or may not be what you intend. It makes most sense to return some non-object type (like **int**) from methods that will be called remotely, unless the object is really intended to be used. (Returning objects isn't horribly expensive, however, and an object is represented by only one proxy on a given connection, even if it is returned many times.)

Reference Counting

With the Distributed Objects system, it is possible for an object to be shared by several applications. Since an object may be in use by many applications, a reference counting scheme may be necessary to insure that an object in use doesn't go away simply because a single application is done with it and frees it. The NXReference protocol is declared to allow objects to implement reference counting. Both the NXConnection and NXProxy classes conform to this protocol in order to know to what extent references are being held. You may wish to make your shared objects conform to this protocol; NXConnection will check if your object conforms to the NXReference protocol before it gives away references to it. If your object conforms to the protocol, a reference is added to the object the first time the object is seen on a connection. Note that a reference is *not* added every time an object is vended, only the first time it is seen on each connection. This works well if the object arrives only once per client application. In other cases, you can add a reference to an object every additional time you receive it, and eliminate the reference (by sending it the free message) every time you are finished with the object.

Object Copies vs. Proxies

While it is often desirable to share an object through the use of proxies, you may occasionally want to pass a copy of an object rather than a proxy. For example, if you have an object that doesn't

change over time, it may be more efficient to pass the object by copy rather than as a proxy; messages to the local copy will require much less overhead than remote messages over a connection. As another example, if an object will be sent many messages before it changes, it may be most efficient to send a copy of the object and send the messages to the copy. This is because sending one large remote message is often more efficient than sending many small remote messages; the overhead of the messaging process is typically much higher than the cost of data transmission.

A new keyword, **bycopy**, has been added to the Objective C language (for use in formal protocol declarations only) to indicate that an object passed as a method parameter ought to be copied rather than passed as a proxy. (The default, without the **bycopy** keyword, is to pass the object as a proxy.)

In the following method declarations, the first method copies the widget across the connection; messages to the copy of the widget will be fast, but changes to the original object will not be reflected in the copy (and vice versa). In the second method, a proxy to widget is given out. The message overhead for remote messages is higher than for messages to a local object, but the widget is truly shared by the applications.

```
- useCopiedWidget: (bycopy in id) widget;
- useSharedWidget: widget;
```

To copy an object over a connection, the receiving application must have a copy of the object's class implementation. This is necessary because the object must be instantiated on the receiving side. Also, an object that is to be copied over a connection must conform to the NXTransport protocol; this protocol defines how an object encodes and decodes itself across a connection. The protocol is as follows:

```
@protocol NXTransport
- encodeUsing:(id <NXEncoding>)portal;
- decodeUsing:(id <NXDecoding>)portal;
- encodeRemotelyFor:(NXConnection *)connection
    freeAfterEncoding:(BOOL *)flagp isBycopy:(BOOL)isBycopy;
@end
```

When an object is to encode itself, it is sent an **encodeUsing:** message where the *portal* argument is an object that conforms to the NXEncoding protocol and thus knows how to encode various data types across a connection. To create the copy of the object on the receiving side, the object is allocated and a **decodeUsing:** message is sent to it. The newly allocated object is *not* initialized, so the **decodeUsing:** implementation generally should invoke the object's designated initializer method. You may occasionally want to substitute another object instead of using the instance that the Distributed Objects system allocated. If you return the substitute object instead of **self**, the substitute object will be used and the system will free the initially allocated memory.

As an example of copying objects, consider the List class, which implements the NXTransport protocol to copy a List object across the connection. The objects in the list are not copied, so the list copy will contain proxies to the objects the real list contains. This behavior may be necessary, because the contents of the list might not conform to the NXTransport protocol and therefore might not be able to be copied. However, if you know the list will only contain objects that conform to the protocol, it may be reasonable to use a list that can be copied, together with its contents, across a connection. The following subclass of List demonstrates exactly this, and shows how a newly allocated object is initialized in the **decodeUsing:** method:

```
@implementation FullCopyList
- encodeUsing:(id <NXEncoding>)portal {
    int i, n = [self count];
    [portal encodeData:&n ofType:"i"];
    for (i = 0; i < n; i++)
        [portal encodeObjectBycopy:[self objectAt:i]];
    return self;
}

- decodeUsing:(id <NXDecoding>)portal {
    int i, n;
    [portal decodeData:&n ofType:"i"];
```

```
    [self initCount:n];
    for (i = 0; i < n; i++)
        [self addObject:[portal decodeObject]];
    return self;
}
@end
```

Determining the Object to Encode

When an object is to be vended to a remote application, the **encodeRemotelyFor:freeAfterEncoding:isBycopy:** method determines what object gets encoded. The default behavior of this method, inherited from the Object class, is to return a local proxy to the object; when the local proxy is encoded, it's received as a remote proxy to the object. However, if this method returns **self**, the NXTransport methods for the object are invoked to copy the object over the connection. The implementation of this method should generally test the value of the **isBycopy** parameter to determine what object to encode:

```
- encodeRemotelyFor:(NXConnection *)connection
    freeAfterEncoding:(BOOL *)flagp
    isBycopy:(BOOL)isBycopy
{
    if (isBycopy) return self;    // encode the object, copying it

    // otherwise, super's behavior is to encode a proxy
    return [super encodeRemotelyFor:connection
                freeAfterEncoding:flagp
                isBycopy:isBycopy];
}
```

Moving an Object Between Applications

It is occasionally useful to move an object from one application to another, and the **encodeRemotelyFor:freeAfterEncoding:isBycopy:** method shown above allows you to do this by setting a flag indicating that the original object is to be freed after encoding and then specifying that the object is to be encoded by copying it across the connection. Note, however, that when you move an object you must be very careful that other applications do not have problems due to the original object getting freed. The following example demonstrates an object that will move every time a reference is given to a remote application.

```
- encodeRemotelyFor:(NXConnection *)connection
    freeAfterEncoding:(BOOL *)flagp
    isBycopy:(BOOL)isBycopy
{
    *flagp = YES;
    return self;
}
```

Asynchronous Messages

By default, remote messages are performed synchronously; execution of the client code doesn't continue until the method in the server returns and the Distributed Objects system sends a reply back to the client (containing the return value if there is one). However, a new keyword for method return values, **oneway**, has been added to the Objective C language to specify asynchronous messages. Like the other new keywords, **oneway** may only be used in a formal protocol declaration. When a client sends an asynchronous message to the server, the method returns to the client immediately. **Oneway** messages implicitly return **void** since the client doesn't wait for a return value from the server. If a method doesn't need to return data and the client doesn't need to stay synchronized to the server, there can be several advantages to **oneway**, asynchronous messages. Because the client continues processing rather than waiting for the server, overall throughput may

increase. Less obviously, **oneway** messages can provide the client with a measure of control over when the client is willing to receive messages back from the server. The server may send a message (like a callback) back to the client anytime the client's connection is running or the client awaits a reply from the server. Occasionally it's unacceptable to receive a callback from the server in the middle of a method implementation (an example might be where the callback is used to clean up and free objects in the client); in such a case you can use **oneway** messages to help insure that the connection is not running and the client won't receive messages until it's ready to do so.

Robust Usage

Although the Distributed Objects system greatly simplifies the sharing of objects, applications that communicate with other applications (distributed applications) are inherently more complex than stand-alone applications. Issues regarding application deaths, communication problems, security, exception handling, and resource allocation must be considered. This section discusses some of the considerations for writing robust distributed applications.

Application Deaths

Distributed applications generally need to know when cooperating applications die. For example, a server application should know when a client application dies (due to an application crash, a system crash, a signal, or other reason) so it can deallocate resources held on the client's behalf, and also avoid sending messages to a client that no longer exists. The Mach operating system tracks all resources held by a process, including the Mach ports used by the Distributed Objects system to send remote messages. The operating system notifies the Distributed Objects system of port deaths when an application dies. The Distributed Objects system, in turn, allows any number of objects to register for notification of the invalidation of the NXConnection object that is used to communicate over its port.

An object must do two things to be notified of the death of a cooperating application:

- It must register for notification of invalidation of the connection to the application.
- It must conform to the NXSenderIsInvalid protocol and take appropriate action when the connection is invalidated.

If the application has no Application object, it must spawn a separate thread to disburse port death notifications. This can be done as follows:

```
[NXPort worryAboutPortInvalidation];
```

Note that in this case, **senderIsInvalid:** messages will be sent from the resultant separate thread, so the receiving object should be thread-safe.

Typically, a new connection is created to vend the first object from one application to another. When your application gets an object in this manner, it should use the returned proxy to get the connection over which the object is accessed, and register for invalidation notification to know when the object becomes inaccessible. The following code gets the proxy for a remote server object and registers for notification of when the server goes away:

```
server = [NXConnection connectToName:REGISTERED_NAME onHost:"*"];
if (server)
{
    NXConnection *    myConnection = [server connectionForProxy];
    [myConnection registerForInvalidationNotification:self];
    [myConnection setDelegate:self];
}
```

In this example, the client also registered itself as the connection's delegate. In this way, the client

can be informed (using the **connection:didConnect:** delegate method) when new connections are automatically created that share **myConnection**'s input port. New, direct connections are formed when proxies are handed between applications. (This eliminates the inefficiency of sending a message over a connection to a proxy that would then forward the message over another connection to the real object.) When a new connection is formed in this manner, the client then has a dependency on the application from which it received the new object, so it should similarly register for invalidation notification on the new connection and it should set the delegate of the new connection appropriately.

If an object registered for a connection's invalidation notification, it receives a **senderIsInvalid:** message from the NXConnection object when the connection is broken (when the connection receives a port death notification indicating an application death, typically). Proper behavior in response to such a notification is nontrivial. The application can examine the NXConnection's list of remote objects (by the **remoteObjects** method) to determine what objects, presumably in use by the application, are no longer accessible. There is no single solution to dealing with application deaths, but a robust architecture is generally one that enables associating a resource to a connection and allows the application to deal with the implications of a broken connection with a cooperating application.

Exceptions

The Distributed Objects system returns exceptions that are raised by method implementations. In other words, if a client sends a message to an object in the server and the implementation in the server raises an exception (see **NX_RAISE**), the exception is forwarded to the client. Also, the Distributed Objects system can raise exceptions in response to communication problems. For this reason, messages to remote objects should generally be bracketed by **NX_DURING**...**NX_ENDHANDLER** constructs. Keep in mind that control isn't returned to a method that doesn't catch an exception that gets raised; for programs using the Application Kit, unhandled exceptions are caught by the Application object's **run** method, which simply continues the event loop.

Memory Leaks

For local messages, returning a pointer to data involves no memory allocation. However, for remote messages, the system must allocate memory to return data, which increases the opportunity to ªleakº memory (in other words, to have allocated memory that has no pointer references, is essentially forgotten and will never be freed). Your application architecture should avoid sending data that needs to be allocated on the client side, or should make it as apparent as possible when data is coming from a remote source. There may still be situations where it isn't immediately obvious whether the recipient of a message is a remote object or not; in this case, if you receive a pointer to data you should check whether the object was a proxy, and if so, take responsibility for freeing the data when you are done. See ªMemory Allocationº earlier in this chapter for an example.

Using Protocols for Efficiency

A message sent to a remote object through a proxy may require two round-trip messages. The first round trip is a request to the real object for its *method signature*, which specifies the types the method requires as arguments. This enables the proxy to encode the data that it has been passed and forward it to the real object. Note that a method signature is not cached; without the use of protocols, it will need to be fetched for every message. The second message (also a round-trip, unless it's a **oneway** message) is used to send the actual message including its encoded arguments, and to return the result.

You can eliminate the need for the first round-trip message by specifying to the proxy the protocol that the corresponding real object conforms to. It's generally known in advance what messages a client will send to a server; the protocol could be as small as a single message a client uses to query

the server or as large as every message the server responds to. When a protocol is specified, the proxy knows the types of the arguments for every message you anticipate sending to the server, and the initial (and somewhat expensive) round-trip message is avoided. If the client sends a message to the server that isn't in the protocol, nothing untoward happens, but an additional round-trip to retrieve the method signature is required. Here is an example of setting the protocol that a client will use to send messages to a server object:

```
@protocol serverMethods
- (int)addClient:(id <clientMethods>)remoteClient;
- getRecordForName:(char *)name
@end

server = [NXConnection connectToName:REGISTERED_NAME onHost:"*"];
if (server)
    [server setProtocolForProxy:@protocol(serverMethods)];
```

Restricting Messages

A key feature of proxies is that they forward any message, including arguments, to the real, remote object. If you return a server object to a remote client, the client can send any message that the server responds to. In fact, the proxy returned to the client will forward any message, whether the server responds to it or not. For security considerations, you might limit the implementation of an object that is to be given out to only methods that the object is willing to receive from remote clients. This is often not practical, however.

An alternative is to group the methods that an object is willing to receive from remote clients into a protocol. You can then use an NXProtocolChecker object (from the Mach Kit) to enforce the protocol. The NXProtocolChecker object forwards all messages in its assigned protocol, but raises an exception for other messages. When an object returns itself as a result of a message forwarded through a protocol checker, the checker substitutes its own **id** for the real object to prevent the sender from receiving an **id** that can receive unchecked messages.

Security

When you register an object with the Network Name Server, it is available to any application that knows the object's name. Because an application must know the object's name, a modicum of security is provided; however, if security is an issue you should not make sensitive objects (or objects capable of providing sensitive objects) available through the Network Name Server. One possible solution is to register only a security validation object with the Network Name Server. This object could require clients to identify themselves as known secure objects before vending sensitive objects.

Multithreaded Applications

The Distributed Objects system is thread-safe. This means that with the proper precautions, the Distributed Objects system can be used to write a multithreaded server. Perhaps more important to application writers is that you can write a server that runs in the main application thread but responds to messages coming from clients running in different threads. This is useful because many parts of the system are not thread-safe and therefore cannot be invoked by clients outside the main thread, but non-thread-safe tasks can be performed on the client's behalf by a server in the main thread. See the discussion of C threads in *NEXTSTEP Operating System Software* for information about which parts of the system are thread-safe.

Relationship to Speaker/Listener

The Speaker and Listener classes in the Application Kit provide a subset of the functionality of the Distributed Objects system. The Distributed Objects system provides a more flexible and dynamic way of communicating between applications. Speaker and Listener are still used by applications to communicate with the Workspace Manager, and will continue to be provided in the near future for backwards compatibility. Nevertheless, the Distributed Objects system is a superior system and should be regarded as a move towards obsoleting the Speaker and Listener classes.