3.3 Release Notes: Distributed Objects

This file contains release notes for the 3.2, 3.1, and 3.0 releases of Distributed Objects. Items specific to the 3.3 release are listed first, and the Release 3.2, 3.1 and 3.0 notes follow.

Known Problems in Release 3.3

These problems exist in Release 3.3:

Reference: 43098

Problem: Reference counting does not work properly in DO.

Description: Reference counting does not work as documented in DO. For instance, references are added each time a server object is passed across a connection (server side) and added to the remote proxy

object only when it is created on the client side.

Workaround: None.

Reference: 49064

Problem: DO can't transfer long doubles.

Description: Transfer of long doubles (by value) doesn't work for any architecture.

Workaround: None.

Problems Fixed in Release 3.3

This problem has been fixed in Release 3.3:

Reference: 20051

Problem: Distributed Objects don't return doubles or structures.

Description: This problem has been fixed, with the exception of some structures 4 bytes or less in size. Returning structures by value works with several caveats:

1. On Intel-based systems (client or server), structures cannot be returned by value. If the structure is increased to greater than 8 bytes, the server will dump core; otherwise, an (apparently) zero-initialized structure will be returned.

2. On Motorola-based systems, structures can be returned by value, but they must be greater than 8 bytes.

3. On PA-RISC systems, 8 byte structures can be returned by value.

Notes Specific to Release 3.2

Known Problems

Reference: 21973

Problem: Server memory not freed for string arguments.

Description: When using Distributed Objects, if you pass a string (using **char *** or **const char ***) as an argument to a Distributed Objects server, the memory allocated by the server for this argument is not freed.

Workaround: None.

Notes Specific to Release 3.1

New Features

Although the documentation says that structures can't be returned, as of Release 3.1, they can be. See bug 20051, below, for more information.

Bugs Fixed in Release 3.1

These bugs have been fixed in Release 3.1:

Reference	30760
Problem	Multithreaded servers don't clean up when connections are freed.
Description	The port remained checked in with the Network Name Server, and the server thread didn't exit.
Reference	30741
Problem	NXPortPortal's **encodeObjectByCopy:** method doesn't encode **nil** correctly.

Description	The result was that any DO routine that returned **nil** using **bycopy** rather than a real object crashed with a bus error.
Reference	30382
Problem	NXConnection's **setOutTimeout:** method has no effect.
Reference	30031
Problem	Bug using proxies without protocols across architectures.
Reference	29887
Problem	Transitive passing of root objects is broken.
Description	Errors occurred if you tried to pass a proxy of a connection's root object to another connection.
Reference	29873
Problem	References to a vended object can persist after the object is freed.
Description	NXConnection's **removeObject:** method should eliminate references to the object passed as the argument to the method. However, if this method was invoked from the object's **free** method (a method that is given special handling in distributed objects), one dangling reference persisted.

Reference	20051
Problem	Distributed Objects don't return doubles or structures.
Description	This bug has been fixed, with the exception of some structures 4 bytes or less in size.

Notes Specific to Release 3.0

These notes were included with the Release 3.0 version of Distributed Objects.

A new facility has been made available that allows programmers to create client-server applications without much fuss. It takes the form of two new classes, NXConnection and NXProxy, which are available in the libsys shared library. These classes allow any two programs to share Objective-C objects and send messages to them independent of their true location.

Overview

DistributedObjects provides a mechanism for distributed programming. Typically, this takes the form of a server program that offers objects (services) to multiple clients, although configurations of pure peers are also common and useful.

The design objectives for DistributedObjects were to

- totally subsume the network aspects of typical RPC programming
- provide the exact programming environment for distributed objects as can be found locally
- provide an object-oriented style of RPC programming
- utilize language mechanisms for efficient network support
- provide support for multithreaded applications

The following sections will illustrate how each of these objectives were met.

Making a DistributedObjects connection

Let's assume the simple case of a single client and server. The client wants to send messages to an object that the server owns. The current implementation of DistributedObjects uses MACH ports as its underlying connection vehicle, and we leverage off of a standard MACH facility called the Network Name Server (NetNameServer) to find the object that the server owns:

```
#import <remote/NXConnection.h>

id ro = [NXConnection connectToName:"ServerExample"];
```

In this example, ro is assigned an NXProxy object that will stand in locally for the object

"vended" by the server, and "ServerExample" is the name used by the server program. The server program has already performed the following code:

```
#import <remote/NXConnection.h>

id          vendedObject = [[YourServerObject alloc] init];
id          roserver = [NXConnection registerRoot:vendedObject withName:"ServerExample"];

[roserver run];        // non-appkit style; blocks
```

The server creates an object of its choice known, in this case, as vendedObject, and registers it with the DistributedObjects subsystem. DistributedObjects registers the MACH port it is using with the Network Name Server so that the client can establish a connection. The server then tells its RemoteObject server object, roserver, to run; that is, to loop awaiting messages from its clients. The messages will be sent to vendedObject and be indistinguishable from local messages sent from elsewhere in the server program (if there were any).

We call the vended object the "root" object because through it, many other objects that the server provides can become accessible. All you have to do is write methods that return these other objects!

If you are writing an AppKit program then you should ask it to service messages for your objects by issuing:

```
[roserver runFromAppKit];
```

instead.

What you can (and can't) do with DistributedObjects

Assuming that the connection discussed in the previous section has been established, what can the client program ask the server program to do with the shared object? Actually, the client can ask it to do anything, since DistributedObjects will *forward* all requests, but the more reasonable question is, what can servers provide to clients in the way of useful Objective-C messages? Here are some examples:

```
[ro aSimpleMessage];          // no parameters
[ro sendAnInteger: 12];       // simple scalars
[ro sendAString:"hello"];
[ro sendAnId: anObject];      // send an arbitrary object
[ro sendAnId: ro];            // send back the shared object!
```

In the last case, we send the id of something the server owns back to the server. In our example, the server knows that object by the value stored in the vendedObject variable, and, indeed, the parameter to the sendAnId: message will have that same value. In other words, DistributedObjects keeps track of the objects sent to and received from other parties and will make sure that your programs see the "right" object. Jumping ahead a little to more advanced features, the client could have sent an NXProxy object that it obtained from some other server X, in which case our example server would have recieved an NXProxy to something vended by Server X. A new connection will have been automatically created between our example server and Server X. Of course, if our server sends *that* NXProxy object back to X, X will see it as the original object sent to our client. In simple terms, you can send objects anywhere and they will always come back to you correctly.

In the case where the client sent anObject, it is possible for the server program to *send a message back to the client*, *in the middle of the server's* sendAnId *implementation*. This is an important feature and should be considered carefully. After the client call completes, the server may wish to call the client back (send a message to an object on the client). In general, this won't work, unless the client is in the middle of sending another message to the server. This is because there has to be a thread of execution waiting to serve incoming requests. The thread executing the client request pauses for the reply and will, charitably, honor any new incoming requests during that time. If the client is willing to yield its thread of control, it can send -run or -runWithTimeout: to the connection.

The last important class of parameters that can be passed are structures:

```
struct   a_struct {
            char aChar;
            int anInt;
            unsigned int bitfield:3;
            enum { red, green, blue } color;
            id anObject;
            char *aString;
            int array[2];
} aStruct = { 'a', 1024, blue, 2, nil, "hello world\n", { 0, 1}};

aStruct.anObject = [Object new];
[ro sendAStruct:aStruct];
```

Although contrived, the example shows that quite complicated things can be passed as parameters.

Servers wouldn't be too useful if they couldn't return interesting values, so DistributedObjects offers the standard mechanisms for so doing:

```
int anInt = [ro getAnInt];
id anId = [ro getAnId];
char *aString = [ro getAString];
double d = [ro getADouble];   // new in 3.1
struct a_struct aStruct = [ro getAStruct]; // new in 3.1
struct a_struct *aStructPointer = [ro getAPtrToStruct];
```

and

```
[ro updateAnInt:&anInt];      // this time, via an inout parameter
[ro updateAnId:&anId];
[ro updateAString:&aString];
[ro updateAStruct:&aStruct];  // caller supplies the storage
```

A server can return a pointer to a structure and DistributedObjects will allocate storage for the structure on the receiving side and return a pointer to it. In the case of the inout parameters, the caller supplies the storage.

It is extremely important to note that we have supplied legal values for the dereferenced parameters (anId has a legal value, aString has a legal value, etc. with the exception of aStruct.anId). This is because DistributedObjects assumes that the server program might want to examine the indirect contents of its passed parameter as part of its function, and will attempt to ship the original contents across. **This may not be what you intend! See the section on Protocols for how to specify** in **or** out **direction exclusively.**

DistributedObjects will allocate memory in order to pass or return strings and structures. It is the responsibility of the recipient to free this memory. Objects

passed by copy also must be freed by the recipient. An object passed by reference will be sent free if the connection breaks or the object is free'd by its recipient. If the object supports the NXReference protocol, however, a reference will be added the first time it is seen on each connection.

Things that won't work

DistributedObjects will not support the following:

unions - DistributedObjects cannot know which element to format and ship.
void * - This is a synonym for an anonymous pointer which is known to be valid in only one address space. We keep it that way.
pointers within structures - DistributedObjects cannot know how deep to recurse when passing a referenced structure so it chooses a simple answer: no recursion.
BOOL * - Distributed Objects treats this incorrectly as a char *.
tiny structures as return values - some structures less than four bytes in size are not returned properly.

Advanced Features

Up to this point in the discussion we have seen many examples of how to use DistributedObjects strictly from a client programmer's perspective, and without any real context, such as the declarations of the object interfaces. One can perfom a

certain amount of code copying and tweaking with a non-zero chance of success, and, in fact, we hope that DistributedObjects is simple enough to use that the non-zero chance is quite high.

But to do significant programming of any nature one must look at the object interfaces and the documentation. This holds true for any objects that are handled by the DistributedObjects system. NeXT has enriched the Objective-C programming language to support better descriptions of programming interfaces, and DistributedObjects supports these enhancements.

The Objective-C language has been extended to support a new method grouping construct known as Protocols. Within a protocol specification, five new keywords have been added. All of these features are of importance to DistributedObjects, so we will discuss them briefly here.

Protocols

A client may specify the expected Protocol that an object will serve upon the completion of a connection. Providing this specification enables more efficient delivery of messages to remote objects by avoiding a "discovery" message per method:

```
id <foo> aFoo = [NXConnection connectToName:"ServerExample"];
[aFoo setProtocolForProxy:@protocol(foo)];
```

A server may restrict the messages served upon a vended object by using the

NXProtocolChecker class:

```
ReadWriteServer *rw = [ReadWriteServer new];
NXProtocolChecker *r = [[NXProtocolChecker alloc] initWithObject:rw
        forProtocol:@protocol(ReadOnly)];
id readServer = [NXConnection registerRoot:r];
```

In the last case, presumably the "ReadOnly" protocol is a subset of the methods that the ReadWriteServer object implements.

Directionality of parameters: *in, out,* and *inout*

In the C programming language, all parameters are passed by value. This works fine for scalars, but pointers present problems. The first problem is whether the pointer points at an array or a single instance, and the second problem is whether the pointer references a valid item upon entry. The historical third problem, whether it is legal to alter the contents of a dereferenced pointer, is addressed by the ANSI `const` declarator.

Objective-C offers three new keywords to address the first and second problems: `in`, `out` and `inout`. An `in` parameter will be copied across from client to server, an `out` parameter will only be copied back, and an `inout` parameter will be copied to and back.

Thus:

```
@protocol foo
```

```
- anINny:(in int *)anInt;
- anOUTty:(out int *)anInt;
- anINnyOUTty:(inout int*)anInt;
@end
```

Pointer parameters are treated as inout by default, and const * parameters are treated as in.

Passing objects on the wire

The default behavior for passing objects from one program to another is to establish an NXProxy object on the client; when the proxy is used locally on the client the Objective-C message is encoded into a MACH message and sent to the server, which decodes and dispatches to the real object. There will be many times when this policy is not desirable, such as when the object is small and won't change over time, when a complete copy of an object is desired for manipulation, or when a container object is passed. In these cases, one wants to pass the implementation of an object in some manner and instantiate a copy on the client side. We describe this latter process as an Object passing itself across the wire (but don't take us too literally!).

Objects that do wish to pass themselves across in this manner must implement the following Transport protocol (from <remote/transport.h>):

```
@protocol Transport
// override standard (NXProxy) formation
- encodeRemotelyFor:(NXConnection *)connection freeAfterEncoding:(BOOL *)flagp
```

```
isBycopy:(BOOL)isBycopy;
// encoding
- encodeUsing:(id <NXEncoding>)portal;
// decoding
- decodeUsing:(id <NXDecoding>)portal;
@end
```

The method encodeUsing is called when the object should encode itself onto the portal and decodeUsing is called on the other end to reconstruct the object. The decodeUsing method should act just like an instance initialization method - storage has been allocated for it but the storage is uninitialized. Another object may be substituted during this reconstruction phase; the substituted object should be returned -- DistributedObjects will free the initially allocated memory.

You will see later that objects don't *always* have to encode themselves, but if they *always* do, they should implement the following method in the following manner:

```
- encodeRemotelyFor:(NXConnection *)connection freeAfterEncoding:(BOOL *)fae
            isBycopy:(BOOL) ibc {
return self;
}
```

The following protocols may be used upon the parameter portal to encode and decode your Object:

```
@protocol NXEncoding
// encode an objc (parameter) type
- encodeData:(void *)data ofType:(const char *)type;
```

```
// encoding methods for transcribing custom objects
- encodeBytes:(const void *)bytes count:(int)count;
- encodeVM:(const void *)bytes count:(int)count;
- encodeMachPort:(port_t)port;
- encodeObject:anObject;            // send a ref to the object across
- encodeObjectBycopy:anObject;   // copy the object across

@end

@protocol NXDecoding
// decode an objc (parameter) type
- decodeData:(void *)d ofType:(const char *)t;

// decoding methods for transcribing custom objects
- decodeBytes:(void *)bytes count:(int)count;
- decodeVM:(void **)bytes count:(int *)count;
- decodeMachPort:(port_t *)pp;
- (id) decodeObject;          // returns decoded object
@end
```

As an example, an Integer object that wanted to pass itself could implement the Transport protocol thus:

```
@interface Integer : Object   <Transport> {
            int          value;
}
...
@end

@implementation Integer
...
```

```
- encodeRemotelyFor:(NXConnection *)connection freeAfterEncoding:(BOOL *)flagp
isBycopy:(BOOL)isBycopy {
        if (isByCopy)
            return self;
        else
            return [super encodeRemoteFor:connection freeAfterEncoding:flagp
                isByCopy:NO];
}

- encodeUsing:(id <NXEncoding>)portal {
    [portal encodeData:&value ofType:"i"];
    return self;
}

- decodeUsing:(id <NXEncoding>)portal {
    [portal decodeData:&value ofType:"i"];
    return self;
}
@end
```

The objects that implement the NXEncoding and NXDecoding protocols hide the nasty details of inter process communication and are of no other utility.

The *bycopy* keyword

In the case of objects, such as containers, that sometimes should instantiate a copy and sometimes not, Objective-C provides the bycopy keyword:

```
- sendAListBycopy:(bycopy id) aList;
```

When a parameter is marked bycopy, DistributedObjects will call that object's encodeRemotelyFor:freeAfterEncoding:isBycopy: method with a YES value for the isBycopy parameter. In those cases where the standard NXProxy treatment is desired, the message should be forwarded to the super class and its result returned.

The *oneway* keyword

There are times in distributed programming when it is known in advance that particular methods will not return to the caller, or that there is no need for a client to wait for the completion of a method as in the normal programming case. For these cases, Objective-C provides the oneway keyword:

```
- (oneway) exit;       // implicitly void
- (oneway) logresult:(in char *) message;
```

Exceptions

DistributedObjects returns exceptions (see NX_RAISE) raised by method implementations.

Multi-threaded programming support

Many servers will wish to be multi-threaded. Some will succeed :-). Servers need to protect any vended objects against shared access, but DistributedObjects is itself

thread-safe and a simple method is provided to invoke several threads serving the same object:

```
while(xtraservers-- > 0)
        [roserver runInNewThread];
[roserver run];        // this one blocks
```

An important restriction when using multiple threads on a single connection is that in general, call-back does not work (unless that call-back is a oneway message). The simplest work-around for this is to build a new connection to the originator, and perhaps dedicate a new thread to that connection:

```
- multithreaded_server_method:(<SomeProtocol>)proxy_to_client {
        NXConnection existing = [proxy_to_client connectionForProxy];
        NXConnection newconn = [NXConnection connectToPort:[existing outPort]];
        id <SomeProtocol> new_proxy = [newconn newRemote:[proxy_to_client nameForProxy]
                withProtocol:@protocol(SomeProtocol)];

        ...
        [new_proxy method_from_SomeProtocol];
}
```

AppKit programming support

When writing single threaded application programs using the AppKit, care must be taken to allow the normal AppKit MACH message processing to occur in conjunction with DistributedObjects. A special interface has been designed to allow single threaded clients and servers to easily share objects within an AppKit context:

Typical use for a Application that acts as a server:

```
id vendedObject = ...;          // something to be exported
NXConnection *c;

c = [NXConnection registerRoot:vendedObject withName:"DOStringServer"];
[c runFromAppKit];
```

Typical use for an app that starts as a client, but that passes out objects that need to be served:

```
id ss = [NXConnection connectToName:"DOStringServer"];

[[ss connectionForProxy] runFromAppKit];
```

DistributedObjects are normally served at the DPS base threshold priority. If this is inappropriate, the runFromAppKitWithPriority:(int)priority method may be used.