

# Distributed Objects Programming Topics



# Contents

## **Introduction to Distributed Objects** 5

Limitations 5

Organization of This Document 5

## **About Distributed Objects** 7

## **Distributed Objects Architecture** 8

## **Connections and Proxies** 10

NSConnection 10

NSProxy and Subclasses 11

    NSDistantObject 12

    NSProtocolChecker 12

## **Ports and Name Servers** 14

NSPorts and Subclasses 14

    NSMachPort 14

    NSMessagePort 15

    NSSocketPort 15

NSPortNameServer and Subclasses 15

    NSMachBootstrapServer 15

    NSMessagePortNameServer 15

    NSSocketPortNameServer 16

## **Message Encapsulation** 17

NSInvocation 17

NSMethodSignature 17

NSPortCoder 18

NSPortMessage 18

NSDistantObjectRequest 18

## **Vending an Object** 19

## **Getting a Vended Object** 21

---

<b>Configuring a Connection</b>	23
<b>Handling Connection Errors</b>	25
<b>Authenticating Connections</b>	27
<b>Making Substitutions During Message Encoding</b>	28
<b>Using NSInvocation</b>	30
Saving NSInvocation Objects for Later Use	31
Using NSInvocation Objects with Timers	31
<b>Document Revision History</b>	33

# Figures

## **Distributed Objects Architecture** 8

Figure 1      Sending a message to a vended object 8

## **Connections and Proxies** 10

Figure 1      NSConnection objects between a server and two client processes 10

Figure 2      NSConnection objects between a server and two client threads 11

Figure 3      NSConnection objects between two threads 11

# Introduction to Distributed Objects

The Objective-C runtime supports an interprocess messaging solution called “distributed objects.” This mechanism enables a Cocoa application to call an object in a different Cocoa application (or a different thread in the same application). The applications can even be running on different computers on a network.

This programming topic describes the Cocoa classes that form the distributed objects system.

## Limitations

Cocoa’s distributed objects system is available only to Objective-C applications.

## Organization of This Document

The Objective-C language support for distributed objects is described in detail in the “Remote Messaging” section of The Runtime System in *The Objective-C Programming Language*. You should be familiar with it before reading this topic. This topic extends that discussion by describing the Cocoa classes used to implement distributed objects.

The classes are divided into the following categories:

- ["Distributed Objects Architecture"](#) (page 8)
- ["Connections and Proxies"](#) (page 10)
- ["Ports and Name Servers"](#) (page 14)
- ["Message Encapsulation"](#) (page 17)

More detailed discussion and examples of how to use distributed objects are covered in the following tasks:

- ["Vending an Object"](#) (page 19)
- ["Getting a Vended Object"](#) (page 21)
- ["Configuring a Connection"](#) (page 23)
- ["Handling Connection Errors"](#) (page 25)
- ["Authenticating Connections"](#) (page 27)

- ["Making Substitutions During Message Encoding"](#) (page 28)
- ["Using NSInvocation"](#) (page 30)

# About Distributed Objects

Cocoa's distributed objects architecture enables objects in different threads or tasks, perhaps on different machines, to transparently send messages to each other. While there are many ways for threads and tasks to communicate with one another, distributed objects hides the mechanism behind the standard Objective-C messaging mechanism. The syntax of local and remote messages are identical.

Remote messages can be sent synchronously or asynchronously. When sending a synchronous message, the sender waits for a reply, blocking its execution, just like a local message. When sending an asynchronous message, the sender continues executing without waiting for a reply; any response from the remote object is ignored.

Distributed objects can be used to divide a complex task into separate code paths that run independently, but can still cooperate as if they were running together. For example, an application can be divided into a graphical front end and a computational back end. The front end can accept all user input and tell the back end to perform various actions. The back end can handle the “heavy lifting” and inform the front end when it can update the user interface with the results. Because the front and back ends run independently, the front end can continue interacting with the user while the back end is busy.

Distributed objects can also be used to implement distributed computing, or parallel processing. If a large job is split into many smaller jobs that are distributed on a multiprocessor machine or on multiple machines, you can harness the combined computational power of all the processors to complete the job. Distributed objects simplifies the application architecture and the communication between its distributed parts.

Cocoa allows distributed objects to communicate over Mach ports, message ports, and sockets. The first two are restricted to communication on a single machine, but sockets can communicate over large networks, including the internet.

Before an object can receive messages from other processes, it must allocate a communication port and make it available to others. Cocoa provides access to name servers for each supported type of communication port. An object can register its port with the appropriate name server to allow other processes to find it. In the case of sockets, you can also publish the object's socket as a Bonjour network service (see the Cocoa Programming Topic *Bonjour Overview*). Other processes that want to communicate with this object can then look up the object's communication port, requesting it by name, and attach distributed object connections to it.

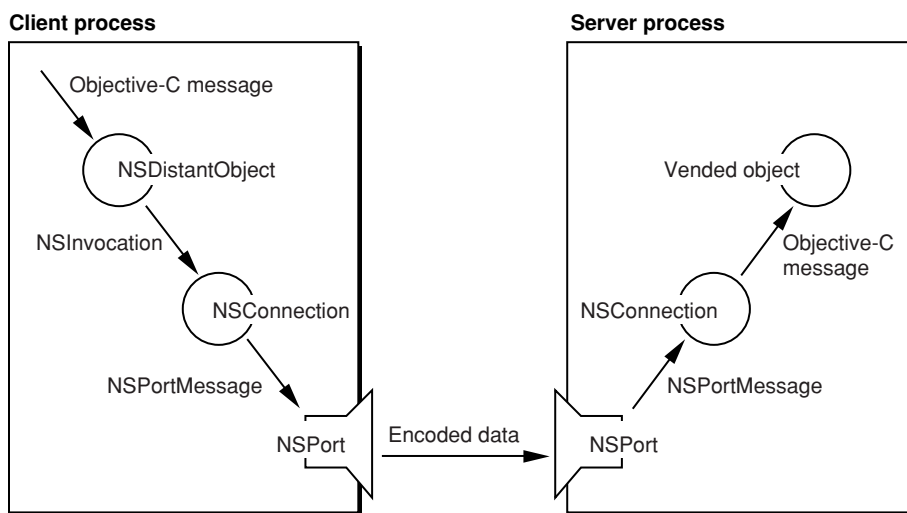
For details on how the Objective-C language and runtime enables distributed objects, see the “Remote Messaging” section of The Runtime System in *The Objective-C Programming Language*.

# Distributed Objects Architecture

Distributed objects operates by having the server process “vend,” or make public, an object to which other client processes can connect. Once a connection is made, the client process invokes one of the vended object’s methods as if the object existed in the client process—the syntax does not change. Cocoa and the Objective-C runtime system handles the necessary transmission of data between the processes.

[Figure 1](#) (page 8) shows many of the objects involved in the distributed objects system and how a message is passed from the client process to the server process. The process goes as follows.

**Figure 1** Sending a message to a vended object



The server process vends an object by attaching it to an **NSConnection** object which contains an **NSPort** object. The port can be registered with an **NSPortNameServer** object to allow easy access by clients wanting to use the vended object. The vended object can be either the real object that implements the methods being provided or an **NSProtocolChecker** proxy object which filters methods based on a protocol before passing methods to the real object.

The client process attaches to the vended object by connecting its own **NSConnection** object to the server’s **NSPort** object (possibly obtained from a port name server) and requesting a proxy of the vended object. The proxy object is an instance of **NSDistantObject**. The client then treats the **NSDistantObject** object as the real object, sending messages normally.



When the client process sends a message to the `NSDistantObject` object, the proxy captures the Objective-C message in the form of an `NSInvocation` object and forwards it to its `NSConnection` object. The `NSConnection` object encodes the `NSInvocation` into an `NSPortMessage` object, using an `NSPortCoder` object, and passes it to an `NSPort` object connected to an `NSPort` object in the server process. The client's port sends the encoded data to the server's port which decodes the data back into an `NSPortMessage` object. The port message is then sent to the `NSConnection` object which converts it into an `NSInvocation` object, using an `NSPortCoder` object. The invocation is finally dispatched as an Objective-C message sent to the vended object. Any return value from the object is passed back through the connection and returned transparently to the client process.

If the vended object is an instance of `NSProtocolChecker`, it tests if the Objective-C message it received conforms to a particular protocol implemented by the real object. If the message is in the protocol, the protocol checker forwards the message to the real object. Otherwise, an exception is raised and returned to the client process.

The client process blocks while the message is dispatched to the server and waits for the remote method request to finish execution, either by returning (with or without a value) or raising an exception. For methods without a return value, the method can be declared with the `oneway` keyword to indicate that the message should be sent asynchronously. The client does not block in that case and continues running once the message is sent.

# Connections and Proxies

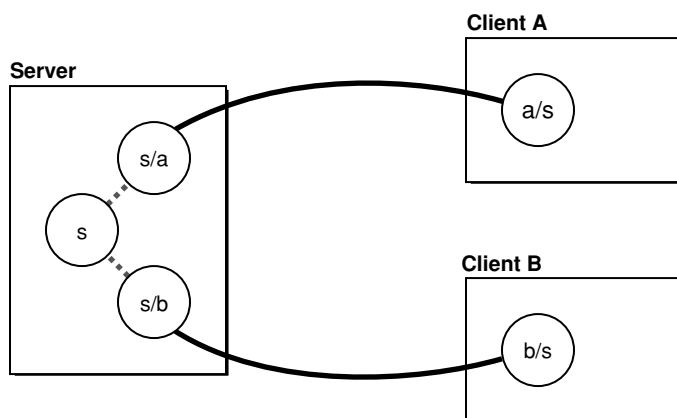
This section describes the highest level components of the distributed objects system: the object that manages the communication (`NSConnection`) and the two proxy objects that stand in for the vended object (`NSDistantObject` and `NSProtocolChecker`).

## NSConnection

`NSConnection` objects manage communication between objects in different threads or tasks, on a single host or over the network. They form the backbone of the distributed objects mechanism, and normally operate in the background. You use `NSConnection` API explicitly when making an object available to other applications, when accessing such a vended object, and when altering default communication parameters; the rest of the time you simply interact with the distributed objects themselves.

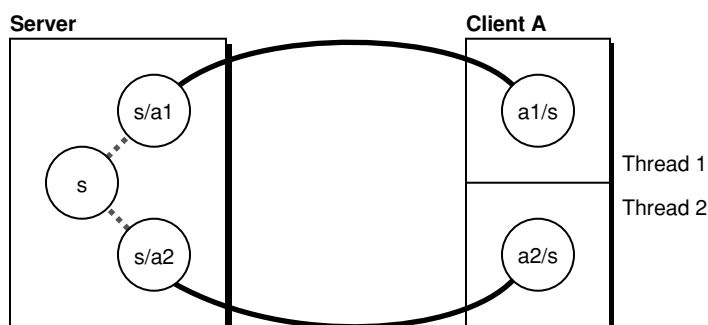
`NSConnection` objects work in pairs, one in each communicating application or thread. A server application has an `NSConnection` object for every client application connected to it, as shown in [Figure 1](#) (page 10) (the connection labeled `s` is used to form new connections, as described in ["Vending an Object"](#) (page 19) and ["Getting a Vended Object"](#) (page 21)). The circles represent `NSConnection` objects, and the labels indicate the application itself and the application it is connected to. For example, in `s/a` the `s` stands for the server and the `a` stands for client A. If a link is formed between clients A and B in this example, two new `NSConnection` objects get created: `a/b` and `b/a`.

**Figure 1** `NSConnection` objects between a server and two client processes



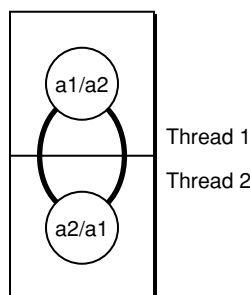
Under normal circumstances, all distributed objects passed between applications are tied through one pair of `NSConnection` objects. `NSConnection` objects cannot be shared by separate threads, though, so for multithreaded applications a separate `NSConnection` object must be created for each thread. This is shown in [Figure 2](#) (page 11).

**Figure 2** `NSConnection` objects between a server and two client threads



Finally, an application can use distributed objects between its own threads to make sending messages thread-safe. This is useful for coordinating work with the Application Kit, for example. [Figure 3](#) (page 11) shows how the `NSConnection` objects are connected. (Note that every thread has its own default `NSConnection` object with which it can vend a single object.) See *Communicating With Distributed Objects* for more details.

**Figure 3** `NSConnection` objects between two threads



## NSProxy and Subclasses

`NSProxy` is an abstract superclass defining an API for objects that act as stand-ins for other objects or for objects that don't exist yet. Typically, a message to a proxy is forwarded to the real object, or causes the proxy to load (or transform itself into) the real object. Subclasses of `NSProxy` can be used to implement transparent distributed messaging (for example, `NSDistantObject`) or for lazy instantiation of objects that are expensive to create.

There are two subclasses of `NSProxy` defined by the distributed objects system. `NSDistantObject` represents the vended object on the client system; it captures messages passed to it and forwards them using an `NSConnection` object to the server process. An `NSProtocolChecker` object can be vended by the server process instead of the real object to filter out any messages that do not conform to a particular protocol. More details of each class are below.

## NSDistantObject

`NSDistantObject` is a concrete subclass of `NSProxy` that defines proxies for objects in other applications or threads. When an `NSDistantObject` object receives a message, in most cases it forwards the message through its `NSConnection` object to the real object in another application, supplying the return value to the sender of the message if one is forthcoming, and propagating any exception back to the invoker of the method that raised it.

`NSDistantObject` adds two useful instance methods to those defined by `NSProxy`. `connectionForProxy` returns the `NSConnection` object that handles the receiver. `setProtocolForProxy:` establishes the set of methods the real object is known to respond to, saving the network traffic required to determine the argument and return types the first time a particular selector is forwarded to the remote proxy. Setting a protocol, though, does not prevent other methods from being sent; they just require network traffic to obtain the method signature. To filter out methods not in the protocol, use an `NSProtocolChecker` instance as the vended object.

There are two kinds of `NSDistantObject`: local proxies and remote proxies. A local proxy is created by an `NSConnection` object the first time an object is sent to another application. It is used by the `NSConnection` object for bookkeeping purposes and should be considered private. The local proxy is transmitted over the network using the `NSCoding` protocol to create the remote proxy, which is the object that the other application uses. `NSDistantObject` defines methods for an `NSConnection` object to create instances, but they are intended only for subclasses to override—you should never invoke them directly. Use `NSConnection`'s `rootProxyForConnectionWithRegisteredName:host:` method, which sets up all the required state for an object-proxy pair.

## NSProtocolChecker

When an object is vended, all of its methods become available to other processes. This may not be desired when vending an object with many methods, only a few of which ought to be remotely accessible. The `NSProtocolChecker` class (a concrete subclass of `NSProxy`) defines an object that restricts the messages that can be sent to another object (referred to as the checker's delegate).

A protocol checker acts as a kind of proxy; when it receives a message that is in its designated protocol, it forwards the message to its target, and consequently appears to be the target object itself. However, when it receives a message not in its protocol, it raises an `NSInvalidArgumentException` to indicate that the message is not allowed, whether or not the target object implements the method.

Typically, an object that is to be distributed (yet must restrict messages) creates an `NSProtocolChecker` object for itself and returns the checker rather than returning itself in response to any messages. The object might also register the checker as the root object of an `NSConnection`.

The object should be careful about vending references to `self`—the protocol checker converts a return value of `self` to indicate the checker rather than the object for any messages forwarded by the checker, but direct references to the object (bypassing the checker) could be passed around by other objects.

# Ports and Name Servers

Ports are the low-level communication channels that transmit and receive the raw data between threads and processes. Ports can be assigned names and advertised to other processes through port name servers. Each type of port has its own port name server.

## NSPorts and Subclasses

An `NSPort` object represents a communication channel to or from another `NSPort` object, which typically resides in a different thread or task. The distributed objects system uses `NSPort` objects to send `NSPortMessage` objects back and forth. You should implement interapplication communication using distributed objects whenever possible, and use `NSPort` objects directly only when necessary.

To receive incoming messages, `NSPort` objects must be added to an `NSRunLoop` as an input source. `NSConnection` objects automatically add their receive port when initialized. See *Run Loops* for more information.

Subclasses of `NSPort` represent particular flavors of data transport from one process to another. The available subclasses are `NSMachPort`, `NSMessagePort`, and `NSSocketPort` and each is described below.

Note that instances of port subclasses cannot be mixed on a particular communication channel. For example, a client cannot connect to a server using `NSMessagePort` if the server only supports connections made with `NSSocketPort`. Also, you cannot transfer instances of `NSMessagePort` in a message to another process over a channel which is using `NSSocketPort` objects as its endpoints; you can only pass `NSSocketPort` objects on such a channel. These restrictions apply to any subclasses of `NSPort`, not just `NSMessagePort` and `NSSocketPort`. However, you are free to create other connections to a server using other subclasses of `NSPort` (assuming the server supports multiple transports) and send instances of that other subclass on that channel.

## NSMachPort

`NSMachPort` is an object wrapper for a Mach port, the fundamental communication port in OS X. `NSMachPort` allows for local (on the same machine) communication only.

To use `NSMachPort` effectively you should be familiar with Mach ports, port access rights, and Mach messages. See the Mach OS documentation for more information.

## NSMessagePort

NSMessagePort is a system-independent implementation of NSPort for sending messages. NSMessagePort allows for local (on the same machine) communication only.

## NSSocketPort

NSSocketPort is a system-independent implementation of NSPort for sending messages over a BSD socket port. NSSocketPort allows for both local and remote communication, but may be more expensive than the other ports for the local case.

## NSPortNameServer and Subclasses

NSPortNameServer provides an object-oriented interface to the port registration service used by the distributed objects system. NSConnection objects use it to contact each other and to distribute objects over the network; you should rarely need to interact directly with an NSPortNameServer object.

You get an NSPortNameServer object by using the `systemDefaultPortNameServer` class method—never allocate and initialize an instance directly. With the default server object you can register an NSPort object under a given name, making it available on the network, and also unregister it so that it cannot be looked up (although other applications that have already looked up the NSPort object can still use it until it becomes invalid).

Each type of NSPort has its own NSPortNameServer subclass as described below.

### NSMachBootstrapServer

This port name server takes and returns instances of NSMachPort.

Port removal functionality is not supported in NSMachBootstrapServer; if you want to cancel a service, you have to destroy the port (invalidate the NSMachPort object given to `registerPort:name:`).

### NSMessagePortNameServer

This port name server takes and returns instances of NSMessagePort.

Port removal functionality is not supported in NSMessagePortNameServer; if you want to cancel a service, you have to destroy the port (invalidate the NSMessagePort object given to `registerPort:name:`).

## NSSocketPortNameServer

This port name server takes and returns instances of `NSSocketPort`.

Port removal functionality is supported by the `removePortForName:` method and should be used to remove invalid socket ports.

Unlike the other port name servers, `NSSocketPortNameServer` can operate over a network. By registering your socket ports, you make them available to other computers on the network without hard-coding the TCP port numbers. Clients just need to know the name of the host running the port name server (and the name of the port).



# Message Encapsulation

This section describes the classes used by the distributed objects system to encapsulate messages passed over a connection. Unless you are getting involved with the low-level details of distributed objects, you should never need to use these classes directly. `NSInvocation` and `NSMethodSignature`, however, have uses outside of distributed objects, so you may encounter them in other situations.

## NSInvocation

An `NSInvocation` object is an Objective-C message rendered static, an action turned into an object. `NSInvocation` objects are used to store and forward messages between objects and between applications, primarily by `NSTimer` and the distributed objects system. An `NSInvocation` object contains all the elements of an Objective-C message: a target, a selector, arguments, and the return value. Each of these elements can be set directly, and the return value is set automatically when the invocation is dispatched.

An `NSInvocation` object can be repeatedly dispatched to different targets; its arguments can be modified between dispatch for varying results; even its selector can be changed to another with the same method signature (argument and return types). This makes it useful for repeating messages with many arguments and variations; rather than retyping a slightly different expression for each message, you modify the `NSInvocation` object as needed each time before dispatching it to a new target.

For examples of how `NSInvocation` is used, see ["Using NSInvocation"](#) (page 30).

## NSMethodSignature

An `NSMethodSignature` object records type information for the arguments and return value of a method. It is used to forward messages that the receiving object does not respond to—most notably in the case of distributed objects. An `NSMethodSignature` object is typically created using `NSObject`'s `methodSignatureForSelector:` instance method. It is then used to create an `NSInvocation` object, which is passed as the argument to a `forwardInvocation:` message to send the invocation on to whatever other object can handle the message. In the default case, `NSObject` invokes `doesNotRecognizeSelector:`, which raises an exception. For distributed objects, the `NSInvocation` object is encoded using the information in the `NSMethodSignature` object and sent to the real object represented by the receiver of the message.

## NSPortCoder

NSPortCoder is a concrete subclass of NSCoder used in the distributed objects system to transmit object proxies (and sometimes objects themselves) between NSConnection objects. An NSPortCoder object is always created and used by an NSConnection object; your code should never need to explicitly create or use one.

## NSPortMessage

An NSPortMessage object defines a low level, operating-system independent type for interapplication (and interthread) messages. NSPortMessage objects are used primarily by the distributed objects system. You should implement interapplication communication using distributed objects whenever possible, and use NSPortMessage objects only when necessary.

An NSPortMessage object has three major parts: the send and receive ports, which are NSPorts that link the sender of the message to the receiver, and the components, which form the body of the message. The components are held as an NSArray object containing NSData and NSPort objects. NSPortMessage's `sendBeforeDate:` message sends the components out through the send port; any replies to the message arrive on the receive port. See the NSPort class specification for information on handling incoming messages.

An NSPortMessage object can be initialized with a pair of NSPort objects and an NSArray instance containing components. An NSPortMessage object's body can contain only NSPort objects or NSData objects. In the distributed objects system the byte/character arrays are usually encoded NSInvocation objects that are being forwarded from a proxy to the corresponding real object.

An NSPortMessage object also maintains a message identifier, which can be used to indicate the class of a message, such as an Objective-C method invocation, a connection request, an error, and so on. Use the `setMsgid:` and `msgid` methods to access the identifier.

## NSDistantObjectRequest

NSDistantObjectRequest objects are used by the Distributed Objects system to help handle invocations between different processes. You should never create NSDistantObjectRequest objects directly. Unless you are getting involved with the low-level details of Distributed Objects, there should never be a need to access an NSDistantObjectRequest object. The distant object request for an incoming message is sent to the connection's delegate if it implements `connection:handleRequest:`.

# Vending an Object

To make an object available to other applications, set it up as the root object of an `NSConnection` object and register the connection by name on the network. This code fragment vends `serverObject`, which is assumed to have a valid value of an object to be vended:

```
/* Assume serverObject has a valid value of an object to be vended. */
NSConnection *theConnection;

theConnection = [NSConnection defaultConnection];
[theConnection setRootObject:serverObject];
if ([theConnection registerName:@"server"] == NO) {
    /* Handle error. */
}
```

This fragment takes advantage of the fact that every thread has a default `NSConnection` object, which can be set up as a server. An `NSConnection` object can vend only one object, so the default `NSConnection` object might not be available. In this case, you can create additional `NSConnection` objects to vend objects with the usual `alloc` and `init` methods.

To advertise the connection to other threads and tasks, this fragment registers `theConnection` under the name “server”. This causes the connection’s default receive port to be registered with the system’s default port name `server` as returned by the `NSPortNameServer` class method `systemDefaultPortNameServer`.

An `NSConnection` object set up this way is called a named connection. A named connection rarely has a channel to any other `NSConnection` object (in [Figure 1](#) (page 10) and [Figure 2](#) (page 11) the named `NSConnection` objects are the circles labeled `s`). When a client contacts the server, a new pair of `NSConnection` objects is created specifically to handle communication between the two.

An `NSConnection` object adds itself to the current `NSRunLoop` instance when it is initialized. In the main thread of an application based on the Application Kit, the run loop is already running, so there is nothing more to do to vend an object. In a secondary thread or an application that does not use the `NSApplication` object, you have to start the run loop explicitly to capture incoming connection requests and messages. This is usually as simple as getting the current thread’s `NSRunLoop` instance and sending it a `run` message:

```
[ [NSRunLoop currentRunLoop] run];
```

See "[Configuring a Connection](#)" (page 23) for more information on setting `NSConnection` objects up to handle requests.

# Getting a Vended Object

An application gets a vended object by creating a proxy, or a stand-in, for that object in its own address space. The proxy forwards messages sent to it through its `NSConnection` object back to the vended object. An application can get a proxy for a vended object in two ways. First, the `rootProxyForConnectionWithRegisteredName:host:` class method returns the proxy directly:

```
id theProxy;
theProxy = [[NSConnection
    rootProxyForConnectionWithRegisteredName:@"server"
    host:nil] retain];
[theProxy setProtocolForProxy:@protocol(ServerProtocol)];
```

This message returns a proxy to the root object of the `NSConnection` object named “server”. The `nil` host name indicates that only the local host is searched for a registered `NSConnection` object; you can specify a specific host name to restrict the server to an identified host.

The invocation of `setProtocolForProxy:` informs the distributed objects system of the set of messages that `theProxy` responds to. Normally, the first time a particular selector is forwarded by a proxy the `NSConnection` object must confirm the argument and return types with the real object. This can add significant overhead to distributed messages. Setting a protocol records this information so that no confirmation is needed for the messages in the protocol, and only the message forwarding costs are incurred.

Another way to get a proxy is to get an `NSConnection` object to the server and then ask for the proxy of its root object:

```
NSConnection *theConnection;
id theProxy;

theConnection = [NSConnection connectionWithRegisteredName:@"server"
    host:nil];
theProxy = [[theConnection rootProxy] retain];
[theProxy setProtocolForProxy:@protocol(ServerProtocol)];
```

This is useful if you need to interact with the `NSConnection` object as well as the proxy. (However, note that `theConnection` is not retained in this example.)

A named `NSConnection` object spawns a child `NSConnection` object to handle communication between two applications (`s` spawning `s/b` and `s/a` in [Figure 1](#) (page 10)). Though the child `NSConnection` object does not have a name, it shares the root object and other configuration attributes of its parent, but not the delegate. You should not register a child `NSConnection` object with a name or change its root object, but you can change its other attributes, as described in ["Configuring a Connection"](#) (page 23).

By default, messages sent to a proxy object are forwarded over the connection synchronously; that is, the sender waits for the message to be processed and a reply received from the remote object. This occurs even for a method with a `void` return type, since the remote object can raise an exception that is passed back to the sender. The local thread or application thus blocks until the message completes execution. To avoid this, you can declare the method type as `oneway void` to cause asynchronous messaging. For more details, see the "Remote Messaging" section of The Runtime System in *The Objective-C Programming Language*.

# Configuring a Connection

You can control some factors of distributed objects communication by configuring `NSConnection` objects. You can set timeouts to limit the amount of time an `NSConnection` object waits on a remote message, set the mode it awaits requests and responses on, and control how an `NSConnection` object manages multiple remote messages. In addition to these parameter settings, you can change an `NSConnection` object's registered name or root object for dynamic alteration of your distributed application.

An `NSConnection` object uses two kinds of timeouts, one for outgoing messages and one for replies. An outgoing network message may take some time to send. Once it goes out, there is usually a delay before any return value arrives. If either of these operations exceeds its timeout, the `NSConnection` object raises an `NSPortTimeoutException`. You can set the values for these timeouts with the `setRequestTimeout:` and `setReplyTimeout:` messages, respectively. By default these timeouts are set to the maximum possible value.

`NSConnection` objects that vend objects await new connection requests in `NSDefaultRunLoopMode` (as defined by the `NSRunLoop` class). When an `NSConnection` object sends a remote message out, it awaits the return value in `NSConnectionReplyMode`. You cannot change this mode, but you can use it to set up `NSTimer` objects or other input mechanisms that need to be processed while awaiting replies to remote messages. Use `addRequestMode:` to add input mechanisms for this mode.

Normally an `NSConnection` object forwards remote messages to their intended recipients as it receives them. If your application returns to the run loop or uses distributed objects either directly or indirectly, it can receive a remote message while it is already busy processing another. Suppose a server is processing a remote message and sends a message to another application through distributed objects. If another application sends a message to the server, its `NSConnection` object immediately forwards it to the intended recipient, even though the server is also awaiting a reply on the outgoing message. This behavior can cause problems if a remote message causes a lengthy change in the server application's state that renders it inconsistent for a time: Other remote messages may interfere with this state, either getting incorrect results or corrupting the state of the server application. You can turn this behavior off with the `setIndependentConversationQueueing:` method, so that only one remote message is allowed to be in effect at any time within the `NSConnection` object's thread. When independent conversation queueing is turned on, the `NSConnection` object forwards incoming remote messages only when no other remote messages are being handled in its thread. This only affects messages between objects, not requests for new connections; new connections can be formed at any time.



**Warning** Because independent conversation queueing causes remote messages to block where they normally do not, it can cause deadlock to occur between applications. Use this method only when you know the nature of the interaction between two applications. Specifically, note that multiple callbacks between the client and server are not possible with independent conversation queueing.

One other way to configure a named `NSConnection` object is to change its name or root object. This effectively changes the object that applications get using the techniques described in ["Getting a Vended Object"](#) (page 21), but does not change the proxies that other applications have already received. You might use this technique to field-upgrade a distributed application with an improved server object class. For example, to install a new server process have the old one change its name, perhaps from "Analysis Server" to "Old Analysis Server". This hides it from clients attempting to establish new connections, but allows its root object to serve existing connections (when those connections close, the old server process exits). In the meantime, launch the new server which claims the name "Analysis Server" so that new requests for analyses contact the updated object.

Note that for inter-host communication, you cannot use the default connection, default `NSPort` subclass, or default port name server. You must use an `NSPort` subclass that supports inter-machine communication, such as `NSSocketPort`. You might configure the server as shown in the following code fragment.

```
NSSocketPort *port = [[NSSocketPort alloc] init];
NSConnection *connection = [NSConnection connectionWithReceivePort:port
sendPort:nil];
[[NSSocketPortNameServer sharedInstance] registerPort:port name:@"doug"];
```

You would then configure the client as follows.

```
NSSocketPort *port = [[NSSocketPortNameServer sharedInstance] portForName:@"doug"
host:@"*"];
NSConnection *connection = [NSConnection connectionWithReceivePort:nil
sendPort:port];
```



# Handling Connection Errors

`NSConnection` objects make use of network resources that can become unavailable at any time, either temporarily or permanently.

Due to heavy network traffic or a busy process, individual messages over a connection may get delayed or lost, causing a timeout error. A timeout error can happen for an outgoing message, meaning the message was never sent to its recipient, or for a reply to a message successfully sent, meaning either that the message failed to reach its recipient or that the reply could not be delivered back to the original sender. `NSConnection` raises an `NSPortTimeoutException` if after a preset time period either the outgoing message is not sent or the reply message is not received. You set the durations for these timeouts with the `NSConnection` instance methods `setRequestTimeout:` and `setReplyTimeout:`. An application can put an exception handler in place for critical messages, and if an `NSPortTimeoutException` is raised it can send the message again, check that the server (or client) is still running or take whatever other action it needs to recover.

In the extreme case, a connection may become permanently severed. When a process using distributed objects crashes, for example, the objects in that process that have been vended to other applications simply cease to exist. In such a case, the `NSConnection` objects handling those objects invalidate themselves and post an `NSConnectionDidDieNotification` to any observers. This notification allows objects to clean up their state as much as possible in the face of an error.

To register for the `NSConnectionDidDieNotification`, add an observer to the default `NSNotificationCenter`:

```
[[NSNotificationCenter defaultCenter] addObserver:proxyUser
      selector:@selector(connectionDidDie:)
      name:NSConnectionDidDieNotification
      object:serverConnection];
```

The fragment above registers the `proxyUser` object to receive a `connectionDidDie:` message when the `serverConnection` object in the application posts an `NSConnectionDidDieNotification`. This allows it to release any proxies it holds and to handle the error as gracefully as possible. See *Notification Programming Topics* for more information on notifications.

One limitation of `NSConnectionDidDieNotification`, however, is that an `NSConnection` object attached to a remote `NSSocketPort` object (the send port) cannot detect when the remote port becomes invalid. This is true even if the remote port is on the same machine. Therefore, an `NSConnection` object cannot post an `NSConnectionDidDieNotification` when the connection is lost. Instead, you must detect the timeout error when the next message is sent and invalidate the `NSSocketPort` object yourself.

In the case of a client-server model wherein the server never messages the client, the server can accumulate `NSConnection` and `NSSocketPort` objects when client applications quit without explicitly logging out from the server. The server, therefore, does not realize that the connection can be closed and released, resulting in memory leaks. One workaround for this situation involves the client vending an object to the server, allowing the server to “ping,” or message, the client if the client has been silent for an excessive period of time. If the client fails to respond, the server can assume the client is no longer alive and it can close the connection.

# Authenticating Connections

An `NSConnection` object can be assigned a delegate, which has two possible responsibilities: approving the formation of new connections, and authenticating messages that pass between `NSConnection` objects.

When a named `NSConnection` object is contacted by a client and forms a child `NSConnection` object to communicate with that client, it sends `connection:shouldMakeNewConnection:` to its delegate first to approve the new connection. If the delegate returns `NO` the connection is refused. This method is useful for limiting the load on a server. It's also useful for setting the delegate of a child `NSConnection` object (since delegates are not shared automatically between parent and child).

Portable Distributed Objects adds message authentication to `NSConnection`'s API. Delegates in different applications can cooperate to validate the messages passing between them by implementing `authenticationDataForComponents:` and `authenticateComponents:withData:`. The first method requests an authentication stamp for an outgoing message, which is used by the second method to check the validity of the message when it is received.

`authenticationDataForComponents:` provides the packaged components for an outgoing network message in the form of `NSData` and `NSPort` objects. The delegate should use only the `NSData` objects to create the authentication stamp, by hashing the data, calculating a checksum, or some other method. The stamp should be small enough not to adversely affect network performance. The delegate in the receiving application receives an `authenticateComponents:withData:` message to confirm the message, and should recalculate the stamp for the components and compare it with the stamp provided. If it returns `YES` the message is forwarded; if it returns `NO`, an `NSFailedAuthenticationException` is raised and a message is logged to the console.

# Making Substitutions During Message Encoding

Like its abstract superclass, `NSCoder`, `NSPortCoder` makes use of substitution methods that allow an object to encode itself as an instance of another class or to replace another object for itself. An object may need to offer a different replacement when being encoded specifically by an `NSPortCoder` object, however, so instead of the generic `classForCoder` and `replacementObjectForCoder:` methods, `NSPortCoder` invokes `classForPortCoder` and `replacementObjectForPortCoder:`. Their default implementations in `NSObject` fall back to the generic methods, providing reasonable default behavior. (`NSPortCoder` does not use a special substitution method for decoding; it simply uses `awakeAfterUsingCoder:` as `NSCoder` does.)

The generic `classForCoder` method is most useful for mapping private subclass hierarchies through a public superclass, which (for example) aids the stability of archives when subclasses are private or subject to change. Since processes communicating at run time typically use the same version of a class library, this mapping is often not needed in distributed objects communication. `classForPortCoder` allows an object to bypass or override the generic `classForCoder` behavior, sending its real class (or simply a different one from the generic case) to the communicating process or thread. If you implement a group of classes that use the generic `classForCoder` method, you should also consider using `classForPortCoder` to handle the special case of encoding with the distributed objects system.

The generic `replacementObjectForCoder:` method offers a standard way to substitute a different instance at encoding time. `replacementObjectForPortCoder:` specifically allows for the substitution of proxies over a distributed objects connection. The receiver of a `replacementObjectForPortCoder:` message can ask the `NSPortCoder` instance whether it should be encoded bycopy or not, and return itself or a proxy as appropriate. `NSObject`'s implementation always returns a proxy, so subclasses that allow bycopy encoding should override `replacementObjectForPortCoder:` to perform at least as this sample does:

```
- (id)replacementObjectForPortCoder:(NSPortCoder *)encoder
{
    if ([encoder isBycopy]) return self;
    return [super replacementObjectForPortCoder:encoder];
}
```

If the `NSPortCoder` object returns `YES` when sent an `isBycopy` message, this example method returns `self`, which results in the receiver being sent an `encodeWithCoder:` message. If the `NSPortCoder` object returns `NO`, this method invokes the superclass's implementation, which typically returns an instance of `NSDistantObject`.

# Using NSInvocation

Creating an `NSInvocation` object requires several steps. Consider this method of the custom class `MyCalendar`:

```
- (BOOL)updateAppointmentsForDate:(NSDate *)aDate
```

`updateAppointmentsForDate:` takes an `NSDate` object as its only argument, and returns YES or NO depending on whether the appointments could be updated without conflicts. The following code fragment sets up an `NSInvocation` object for it:

```
SEL theSelector;
NSMethodSignature *aSignature;
NSInvocation *anInvocation;

theSelector = @selector(updateAppointmentsForDate:);
aSignature = [MyCalendar instanceMethodSignatureForSelector:theSelector];
anInvocation = [NSInvocation invocationWithMethodSignature:aSignature];
[anInvocation setSelector:theSelector];
```

The first two lines get the `NSMethodSignature` object for the `updateAppointmentsForDate:` method. The last two lines actually create the `NSInvocation` object and set its selector. Note that the selector can be set to any selector matching the signature of `updateAppointmentsForDate:`. Any of these methods can be used with `anInvocation`:

```
- (BOOL)clearAppointmentsForDate:(NSDate *)aDate
- (BOOL)isAvailableOnDate:(NSDate *)aDate
- (BOOL)setMeetingTime:(NSDate *)aDate
```

Before being dispatched, `anInvocation` must have its target and arguments set:

```
MyCalendar *userDatebook;    /* Assume this exists. */
NSDate *todaysDate;          /* Assume this exists. */
```

```
[anInvocation setTarget:userDatebook];  
[anInvocation setArgument:&todaysDate atIndex:2];
```

`setArgument:atIndex:` sets the specified argument to the value supplied. Every method has two hidden arguments, the `target` and `selector` (whose indices are 0 and 1), so the first argument that needs to be set is actually at index 2. In this case, `todaysDate` is the `NSDate` argument to `updateAppointmentsForDate:`.

To dispatch the `NSInvocation` object, send an `invoke` or `invokeWithTarget:` message. `invoke` only produces a result if the `NSInvocation` object has a target set. Once dispatched, the `NSInvocation` object contains the return value of the message, which `getReturnValue:` produces:

```
BOOL result;  
  
[anInvocation invoke];  
[anInvocation getReturnValue:&result];
```

`NSInvocation` does not support invocations of methods with either variable numbers of arguments or union arguments.

## Saving NSInvocation Objects for Later Use

Because an `NSInvocation` object does not always need to retain its arguments, by default it does not do so. This can cause object arguments as well as the target to become invalid if they are automatically released. If you plan to cache an `NSInvocation` object or dispatch it repeatedly during the execution of your application, you should send it a `retainArguments` message. This method retains the target and all object arguments, and copies C strings so that they are not lost because another object frees them.

## Using NSInvocation Objects with Timers

Suppose the `NSInvocation` object created above is being used in a time-management application that allows multiple users to set appointments for others, such as group meetings. This application might allow a user's calendar to be automatically updated every few minutes, so that the user always knows what his schedule looks like. Such automatic updating can be accomplished by setting up `NSTimer` objects with `NSInvocation` objects.

Given the `NSInvocation` object above, this is as simple as invoking one `NSTimer` method:

```
[NSTimer scheduledTimerWithInterval:600  
    invocation:anInvocation  
    repeats:YES];
```

This line of code sets up an `NSTimer` object to dispatch an `Invocation` every 10 minutes (600 seconds). Note that an `NSTimer` object always instructs its `NSInvocation` object to retain its arguments; thus, you do not need to send `retainArguments` yourself. See *Timer Programming Topics* for more information on timers.



# Document Revision History

This table describes the changes to *Distributed Objects Programming Topics*.

Date	Notes
2007-06-06	Updated a link to "The Runtime System" article (in The Objective-C Programming Language).
2006-02-07	Clarified configuration of NSConnection for inter-host communication.
2005-04-29	Changed "Rendezvous" to "Bonjour."
2003-04-15	<a href="#">"About Distributed Objects"</a> (page 7) moved from the "Networking" topic.
2002-11-12	Revision history was added to existing topic. It will be used to record changes to the content of the topic.



Apple Inc.  
© 2003, 2007 Apple Inc.  
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

The Apple logo is a trademark of Apple Inc.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications only for Apple-labeled computers.

Apple Inc.  
1 Infinite Loop  
Cupertino, CA 95014  
408-996-1010

Apple, the Apple logo, Bonjour, Cocoa, Mac, Objective-C, and OS X are trademarks of Apple Inc., registered in the United States and other countries.

Even though Apple has reviewed this document, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.