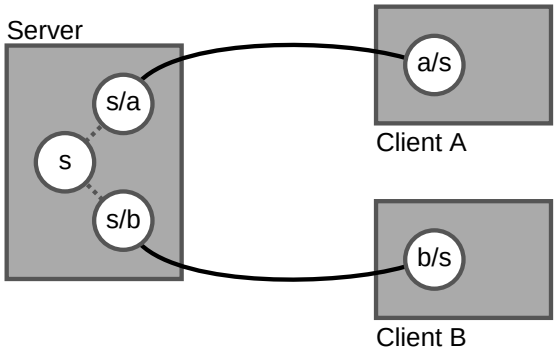
NSConnection

Inherits From:	NSObject
Conforms To:	NSObject (NSObject)
Declared In:	Foundation/NSConnection.h

Class Description

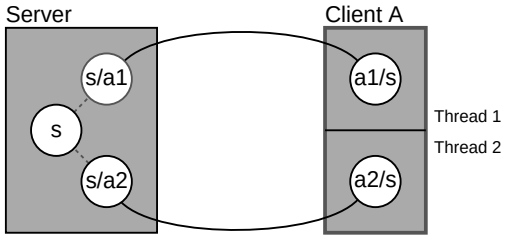
NSConnection objects manage communication between objects in different threads or tasks, on a single host or over the network. They form the backbone of the distributed objects mechanism, and normally operate in the background. You use NSConnection API explicitly when making an object available to other 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.



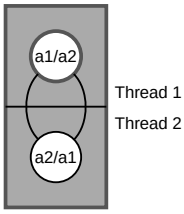
NSConnection objects work in pairs, one in each communicating application or thread. A server application has an NSConnection for every client application connected to it, as shown above (the NSConnection labeled **s** is used to form new connections, as described under “Vending an Object” and “Getting a Vended Object”). The circles represent NSConnection objects, and the labels indicate the application itself and the application it’s 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**.

Note: For NeXT’s implementation the small letters represent NSPort objects. The letter on the left indicates the *receive port*, which messages arrive on; the letter on the right indicates the *send port*, which outgoing messages leave through. Similar mechanisms exist for other OpenStep-compliant systems, but any API exporting them isn’t part of the OpenStep specification. Subclasses of NSConnection that use them (which they typically must do) are therefore not portable among OpenStep implementations.

Under normal circumstances, all distributed objects passed between applications are tied through one pair of NSConnection objects. NSConnection objects can't be shared by separate threads, though, so for multithreaded applications a separate NSConnection must be created for each thread. This is shown in here:



Finally, an application can use distributed objects between its own threads to make sending messages thread-safe (see the following figure). This is useful for coordinating work with the Application Kit, for example.



Vending an Object

To make an object available to other applications, set it up as the root object of an NSConnection and register the NSConnection by name on the network. This code fragment vends **serverObject**:

```
id serverObject;    /* Assume this exists. */
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 can vend only one object, so the default NSConnection might not be available. In this case, you can create additional NSConnections to vend objects with the usual **alloc** and **init** methods.

An NSConnection set up this way is called a *named NSConnection.* A named NSConnection rarely has a channel to any other NSConnection (in the illustrations above the named NSConnection is the circle labeled

s). When a client contacts the server, a new pair of NSConnection objects is created specifically to handle communication between the two. The following sections describe this in more detail.

An NSConnection adds itself to the current NSRunLoop when it's initialized. In an application based on the Application Kit, the NSRunLoop is already running, so there's nothing more to do to vend an object. In an application that doesn't use the NSApplication object, you have to start the NSRunLoop explicitly to capture incoming connection requests and messages. This is usually as simple as getting the current thread's NSRunLoop and sending it a **run** message:

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

See "Configuring an NSConnection" and the NSRunLoop class description for more information on setting NSConnections 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 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:@"*"]
    retain];
[theProxy setProtocolForProxy:@protocol(ServerProtocol)];
```

This message returns a proxy to the root object of the NSConnection named "server". The host name of "*" indicates that any host on the local subnet with such an NSConnection can be used; you can specify a specific host name to restrict the server to an identified host. If the host name is **nil** or is empty then only the local host is searched for a registered NSConnection.

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 to the server and then ask for the proxy of its root object:

```
NSConnection *theConnection;
id theProxy;

theConnection = [NSConnection connectionWithRegisteredName:@"server"
    host:@"*"];
```

```
theProxy = [[theConnection rootProxy] retain];
[theProxy setProtocolForProxy:@protocol(ServerProtocol)];
```

This is useful if you need to interact with the NSConnection as well as the proxy (note, though, that **connection** isn't retained in this example).

A named NSConnection spawns a child NSConnection to handle communication between two applications (**s** spawning **s/b** and **s/a** in the first figure). Though the child NSConnection doesn't have a name, it shares the root object and other configuration attributes of its parent (but not the delegate). You shouldn't register a child NSConnection with a name or change its root object, but you can change its other attributes, as described in the next section.

Forming Connections Between Threads

You can use distributed objects between threads in a single application for thread-safe use of shared objects, such as those in the Application Kit. The Application Kit objects run in the main thread, while special calculations or other computations run in other threads. You can set up distributed objects between threads by registering them under names, but doing so exports the object over the entire network, making them available to other applications. This may not be desirable.

Instead of registering your distributed objects, you can manually set up NSConnections that are linked to each other with NSPorts using the **connectionWithReceivePort:sendPort:** message. In the main (Application Kit) thread, you create an NSConnection with two NSPorts, then switch them to create another NSConnection in a separate thread. Suppose you have an application that uses a Calculator object to perform intensive calculations. It might make sense to have this object run in a separate thread and communicate through an NSConnection. In the following example, the initial work of setting up a Calculator object is done in the NSApplication delegate's **applicationDidFinishLaunching:** method:

```
- (void)applicationDidFinishLaunching:(NSNotification *)note
{
    NSPort *port1;
    NSPort *port2;
    NSArray *portArray;

    port1 = [NSPort port];
    port2 = [NSPort port];

    kitConnection = [[NSConnection alloc] initWithReceivePort:port1
        sendPort:port2];
    [kitConnection setRootObject:self];

    /* Ports switched here. */
    portArray = [NSArray arrayWithObjects:port2, port1, nil];

    [NSThread detachNewThreadSelector:@selector(connectWithPorts:)
        toTarget:[Calculator class] withObject:portArray];
```

```
    return;
}
```

The delegate creates **kitConnection**, which is an instance variable, and sets itself up as the root object so that the Calculator object can find it. To set up the server thread, the delegate packages the NSPorts in an NSArray (note that they're in reverse order) and creates a Calculator object in a separate thread by invoking **detachNewThreadSelector:toTarget:withObject:**. This message starts the new thread by sending **connectWithPorts:** to the Calculator class, which creates the Calculator object that the application uses as an internal server and connects that object to the application delegate:

```
+ (void)connectWithPorts:(NSArray *)portArray
{
    NSAutoreleasePool *pool;
    NSConnection *serverConnection;
    Calculator *serverObject;

    pool = [[NSAutoreleasePool alloc] init];

    serverConnection = [NSConnection
        connectionWithReceivePort:[portArray objectAtIndex:0]
        sendPort:[portArray objectAtIndex:1]];

    serverObject = [[self alloc] init];
    [(id)[serverConnection rootProxy] setServer:serverObject];
    [serverObject release];

    [[NSRunLoop currentRunLoop] run];
    [pool release];
    [NSThread exit];

    return;
}
```

connectWithPorts: is responsible for creating the server object and connecting it to the client thread. This method first creates an NSAutoreleasePool to prevent objects from being leaked if the NSThread ever exits. It then creates **serverConnection** using the two NSPorts from **kitConnection**, and proceeds to set up the server object. Once this is done, it passes the server object back to the client with a **setServer:** message. Finally, it starts the NSRunLoop for its thread. If the NSRunLoop ever stops, the NSAutoreleasePool is cleaned up and the NSThread is made to exit.

The application delegate's **setServer:** message simply sets the protocol and stores the server object:

```
- (void)setServer:(id)anObject
{
    [anObject setProtocolForProxy:@protocol(CalculatorMethods)];
    calculator = (id <CalculatorMethods>)[anObject retain];
    return;
}
```

Because *anObject* arrived through the NSConnections set up between threads, it's really a proxy to the **serverObject** created in **connectWithPorts:** above.

Configuring an NSConnection

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 will wait on a remote message, set the mode it awaits requests and responses on, and control how an NSConnection manages multiple remote messages. In addition to these parameter settings, you can change an NSConnection's registered name or root object for dynamic alteration of your distributed application.

An NSConnection 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's usually a delay before any return value arrives. If either of these operations exceeds its timeout, the NSConnection 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.

NSConnections that vend objects await new connection requests in NSDefaultRunLoopMode (as defined by the NSRunLoop class). When an NSConnection sends a remote message out, it awaits the return value in NSConnectionReplyMode. You can't change this mode, but you can use it to set up NSTimers 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 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's 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 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's thread. When independent conversation queueing is turned on, the NSConnection 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 don't, 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 aren't possible with independent conversation queueing.

One other way to configure a named NSConnection 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,” but doesn’t 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.

The Delegate

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

When a named NSConnection is contacted by a client and forms a child NSConnection 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 (since delegates aren’t shared automatically between parent and child).

Portable Distributed Objects adds message authentication to NSConnection’s OpenStep 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’s 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.

Handling NSConnection Errors

NSConnections make use of network resources that can become unavailable at any time. When a server machine loses power, for example, the objects on that machine that have been vended to other applications simply cease to exist. In such a case, the NSConnections 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 notification, 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 the NSNotification and NSNotificationCenter class specifications for more information on notifications.

A less serious error is a timeout on a remote message. This 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 couldn't be delivered back to the original sender. An application can put an exception handler in place for critical messages, and if a timeout exception is raised it can send the message again, check that the server is still running or take whatever other action it needs to recover.

Method Types

Getting the default instance	+ defaultConnection
Creating instances	+ connectionWithReceivePort:sendPort: – initWithReceivePort:sendPort:
Running the connection in a new thread	– runInNewThread
Vending an object	– registerName: – setRootObject: – rootObject
Getting a remote object	+ connectionWithRegisteredName:host: – rootProxy + rootProxyForConnectionWithRegisteredName:host:
Getting all NSConnections	+ allConnections
Configuring instances	– setRequestTimeout: – requestTimeout – setReplyTimeout: – replyTimeout – setIndependentConversationQueueing: – independentConversationQueueing – addRequestMode: – removeRequestMode: – requestModes – invalidate

Getting ports	– receivePort – sendPort
Getting statistics	– statistics
Setting the delegate	– setDelegate: – delegate

Class Methods

allConnections

+ (NSArray *)**allConnections**

Returns all valid NSConnections in the process.

See also: – **isValid**

connectionWithReceivePort:sendPort:

+ (NSConnection *)**connectionWithReceivePort:**(NSPort *)*receivePort* **sendPort:**(NSPort *)*sendPort*

Returns an NSConnection that communicates using *receivePort* and *sendPort*. See **initWithReceivePort:sendPort:** for more information.

See also: + **defaultConnection**

connectionWithRegisteredName:host:

+ (NSConnection *)**connectionWithRegisteredName:**(NSString *)*name* **host:**(NSString *)*hostName*

Returns the NSConnection whose send port links it to the NSConnection registered under *name* on the host named *hostName*. Returns **nil** if no NSConnection can be found for *name* and *hostName*. The returned NSConnection is a child of the default NSConnection for the current thread (that is, it shares the default NSConnection's receive port).

If *hostName* is **nil** or empty then only the local host is searched for the named NSConnection. If *hostName* is "*" then all hosts on the local subnet are queried for an NSConnection registered under *name;* where there are duplicates the connection is made with an arbitrary host, which is then used for every subsequent request from the local host.

To get the object vended by the NSConnection, use the **rootProxy** instance method. The **rootProxyForConnectionWithRegisteredName:host:** class method immediately returns this object.

See also: + **defaultConnection**

defaultConnection

+ (NSConnection *)**defaultConnection**

Returns the default NSConnection for the current thread, creating it if necessary. The default NSConnection uses a single NSPort for both receiving and sending, and is useful only for vending an object; use the **setRootObject:** and **registerName:** methods to do this.

See also: – **init**

rootProxyForConnectionWithRegisteredName:host:

+ (NSDistantObject *)**rootProxyForConnectionWithRegisteredName:**(NSString *)*name*
host:(NSString *)*hostName*

Returns a proxy for the root object of the NSConnection registered under *name* on the host named *hostName*, or **nil** if that NSConnection has no root object set. Also returns **nil** if no NSConnection can be found for *name* and *hostName*. The NSConnection of the returned proxy is a child of the default NSConnection for the current thread (that is, it shares the default NSConnection's receive port).

If *hostName* is **nil** or empty then only the local host is searched for the named NSConnection. If *hostName* is "*" then all hosts on the local subnet are queried for an NSConnection registered under *name;* where there are duplicates the connection is made with an arbitrary host, which is then used for every subsequent request from the local host.

This method invokes **connectionWithRegisteredName:host:** and sends the resulting NSConnection object a **rootProxy** message.

See also: – **setRootObject:**

Instance Methods

addRequestMode:

– (void)**addRequestMode:**(NSString *)*mode*

Adds *mode* to the set of run loop input modes that the NSConnection uses for connection requests. The default input mode is NSDefaultRunLoopMode. See the NSRunLoop class specification for more information on input modes.

See also: – **addPort:forMode:** (NSRunLoop)

delegate

– (id)**delegate**

Returns the NSConnection's delegate.

See also: – **setDelegate:**

independentConversationQueueing

– (BOOL)**independentConversationQueueing**

Returns YES if the NSConnection handles remote messages atomically, NO otherwise. See "Configuring an NSConnection" in the class description for more information on independent conversation queueing.

See also: – **setIndependentConversationQueueing:**

initWithReceivePort:sendPort:

– (id)**initWithReceivePort:**(NSPort *)*receivePort* **sendPort:**(NSPort *)*sendPort*

Initializes a newly created NSConnection with *receivePort* and *sendPort*. The new NSConnection adds *receivePort* to the current NSRunLoop with NSDefaultRunLoopMode as the mode. If the application doesn't use an NSApplication object to handle events, it needs to run the NSRunLoop with one of its various **run...** messages. Returns **self**.

This method posts an NSConnectionDidInitializeNotification once the connection is initialized.

receivePort and *sendPort* affect initialization as follows:

- If an NSConnection with the same ports already exists, releases the receiver, retains the existing NSConnection, and returns it.
- If an NSConnection exists that uses the same ports, but switched in role, then the new NSConnection communicates with it. Messages sent to a proxy held by either NSConnection are forwarded through the other NSConnection. This rule applies both within and across address spaces.

  This behavior is useful for setting up distributed objects connections between threads within an application. See "Forming Connections Between Threads" in the class description for more information.
- If *receivePort* is **nil**, deallocates the receiver and returns **nil**.
- If *sendPort* is **nil** or if both ports are the same, the NSConnection uses *receivePort* for both sending and receiving, and is useful only for vending an object. Use the **registerName:** and **setRootObject:** instance methods to vend an object.
- If an NSConnection exists that uses *receivePort* as both of its ports, it's treated as the parent of the new NSConnection, and its root object and all of its configuration settings are applied to the new

NSConnection. You should neither register a name for nor set the root object of the new NSConnection. See “Configuring an NSConnection” in the class description for more information.

- If *receivePort* and *sendPort* are different and neither is shared with another NSConnection, the receiver can be used to vend an object as well as to communicate with other NSConnections. However, it has no other NSConnection to communicate with until one is set up.
- *receivePort* can’t be shared by NSConnections in different threads.

This method is the designated initializer for the NSConnection class. Because it isn’t part of the OpenStep specification, subclasses of NSConnection aren’t portable among different OpenStep implementations.

See also: + **defaultConnection**

invalidate

– (void)**invalidate**

Invalidates (but doesn’t release) the receiver. After withdrawing the ports that it has registered with the current run loop, **invalidate** posts an NSConnectionDidDieNotification and then invalidates all remote objects and exported local proxies.

See also: – **isValid**, – **removePort:forMode:** (NSRunLoop), – **requestModes:**

isValid

– (BOOL)**isValid**

Returns NO if the NSConnection is known to be invalid, YES otherwise. An NSConnection becomes invalid when either of its ports becomes invalid, but only notes that it has become invalid when it tries to send or receive a message. When this happens it posts an NSConnectionDidDieNotification to the default notification center.

See also: – **invalidate**, – **isValid** (NSPort)

receivePort

– (NSPort *)**receivePort**

Returns the NSPort that the NSConnection receives incoming network messages on. You can inspect this object for debugging purposes or use it to create another NSConnection, but shouldn’t use it to send or receive messages explicitly. Don’t set the delegate of the receive port; it already has a delegate established by the NSConnection.

See also: – **sendPort**, – **initWithReceivePort:sendPort:**

registerName:

– (BOOL)**registerName:**(NSString *)*name*

Registers the NSConnection under *name* on the local host, returning YES if successful, NO if not (for example, if another NSConnection on the same host is already registered under *name*). Other NSConnections can then contact it using the **connectionWithRegisteredName:host:** and **rootProxyForConnectionWithRegisteredName:host:** class methods.

If the NSConnection was already registered under a name and this method returns NO, the old name remains in effect. If this method is successful, it also unregisters the old name.

To unregister an NSConnection, simply invoke **registerName:** and supply **nil** as the connection name.

See also: – **setRootObject:**

removeRequestMode:

– (void)**removeRequestMode:**(NSString *)*mode*

Removes *mode* from the set of run loop input modes that the NSConnection uses for connection requests.

See also: – **removePort:forMode:** (NSRunLoop)

replyTimeout

– (NSTimeInterval)**replyTimeout**

Returns the timeout interval for replies to outgoing remote messages. If a non-**oneway** remote message is sent and no reply is received by the timeout, an NSPortTimeoutException is raised.

See also: – **requestTimeout**, – **setReplyTimeout:**

requestModes

– (NSArray *)**requestModes**

Returns the set of request modes (as an array of NSStrings) that the NSConnection's receive port is registered for with its NSRunLoop.

See also: – **addRequestMode:**, – **addPort:forMode:** (NSRunLoop), – **removeRequestMode:**

requestTimeout

– (NSTimeInterval)**requestTimeout**

Returns the timeout interval for outgoing remote messages. If a remote message can't be sent before the timeout, an NSPortTimeoutException is raised.

See also: – **replyTimeout**, – **setRequestTimeout:**

rootObject

– (id)**rootObject**

Returns the object that the NSConnection (or its parent) makes available to other applications or threads, or **nil** if there is no root object. To get a proxy to this object in another application or thread, invoke the **rootProxyForConnectionWithRegisteredName:host:** class method with the appropriate arguments.

See also: – **rootProxy**, – **setRootObject:**

rootProxy

– (NSDistantObject *)**rootProxy**

Returns the proxy for the root object of NSConnection's peer in another application or thread. The proxy returned can change between invocations if the peer NSConnection's root object is changed.

Note: If the NSConnection uses separate send and receive ports and has no peer, when you invoke rootProxy it will block for the duration of the reply timeout interval, waiting for a reply.

See also: – **rootObject**

runInNewThread

– (void)**runInNewThread**

Creates and starts a new NSThread and then runs the receiving connection in the new thread. If the newly-created thread is the first to be detached from the current thread, this method posts the notification NSBecomingMultiThreaded with the nil object to the default notification center.

sendPort

– (NSPort *)**sendPort**

Returns the NSPort that the NSConnection sends outgoing network messages through. You can inspect this object for debugging purposes or use it to create another NSConnection, but shouldn't use it to send or

receive messages explicitly. Don't set the delegate of the send port; it already has a delegate established by the NSConnection.

See also: – **receivePort**, – **initWithReceivePort:sendPort:**

setDelegate:

– (void)**setDelegate:**(id)*anObject*

Sets the NSConnection's delegate to *anObject*. Doesn't retain *anObject*.

setIndependentConversationQueueing:

– (void)**setIndependentConversationQueueing:**(BOOL)*flag*

Sets whether the NSConnection handles remote messages atomically. The default is NO: An NSConnection normally forwards remote message to the intended recipients as they come in. See "Configuring an NSConnection" in the class description for more information.

See also: – **independentConversationQueueing**

setReplyTimeout:

– (void)**setReplyTimeout:**(NSTimeInterval)*seconds*

Sets the timeout interval for replies to outgoing remote messages to *seconds*. If a non-**oneway** remote message is sent and no reply is forthcoming by the timeout, an NSPortTimeoutException is raised. The default timeout is the maximum possible value.

See also: – **setRequestTimeout:**, – **replyTimeout**

setRequestMode:

– (void)**setRequestMode:**(NSString *)*mode*

Sets the run loop mode that the NSConnection uses for connection requests to *mode* and reregisters the NSConnection's receive port with the current NSRunLoop. The default request mode is NSDefaultRunLoopMode. See the NSRunLoop class specification for more information on run modes.

See also: – **requestMode**

setRequestTimeout:

– (void)**setRequestTimeout:**(NSTimeInterval)*seconds*

Sets the timeout interval for outgoing remote messages to *seconds*. If a remote message can't be sent before the timeout, an NSPortTimeoutException is raised. The default timeout is the maximum possible value.

See also: – **setReplyTimeout:**, – **requestTimeout**

setRootObject:

– (void)**setRootObject:**(id)*anObject*

Sets the object that the NSConnection makes available to other applications or threads to *anObject*. This only affects new connection requests and **rootProxy** messages to established NSConnections; application that have proxies to the old root object can still send messages through it.

See also: – **rootObject**

statistics

– (NSDictionary *)**statistics**

Returns an NSDictionary containing various statistics for the NSConnection, such as the number of vended objects, the number of requests and replies, and so on. The statistics dictionary should be used only for debugging purposes; see the release notes for more information on its contents.

Methods Implemented by the Delegate

authenticateComponents:withData:

– (BOOL)**authenticateComponents:**(NSArray *)*components*
withData:(NSData *)*authenticationData*

Returns YES if the *authenticationData* provided is valid for *components*, NO otherwise. *components* contains NSData and NSPort objects belonging to an NSPortMessage object. See the NSPortMessage class specification for more information. *authenticationData* should have been created by the delegate of the peer NSConnection with **authenticationDataForComponents:**.

Use this message for validation of incoming messages. An NSConnection raises an NSFailedAuthenticationException on receipt of a remote message that the delegate doesn't authenticate.

authenticationDataForComponents:

– (NSData *)**authenticationDataForComponents:**(NSArray *)*components*

Returns an NSData object to be used as a authentication stamp for an outgoing message. *components* contains the elements of a network message, in the form of NSPort and NSData objects. The delegate should use only the NSData elements to create the authentication stamp. See the NSPortMessage class specification for more information on the components.

If **authenticationDataForComponents:** returns **nil**, an NSGenericException will be raised. If the delegate determines that the message shouldn't be authenticated, it should return an empty NSData object (you can generate an empty NSData object with **[NSData data]**). The delegate on the other side of the connection must then be prepared to accept an empty NSData as the second parameter to **authenticateComponents:withData:** and to handle the situation appropriately.

components will be validated on receipt by the delegate of the peer NSConnection with **authenticateComponents:withData:**.

connection:shouldMakeNewConnection:

– (BOOL)**connection:**(NSConnection *)*parentConnection*
shouldMakeNewConnection:(NSConnection *)*newConnnection*

Returns YES if *parentConnection* should allow *newConnnection* to be created and set up, NO if *parentConnection* should refuse and immediately release *newConnection*. Use this method to limit the amount of NSConnections created in your application or to change the parameters of child NSConnections.

makeNewConnection:sender:

– (BOOL)**makeNewConnection:**(NSConnection *)*newConnection*
sender:(NSConnection *)*parentConnection*

Returns YES if *parentConnection* should allow *newConnnection* to be created and set up, NO if *parentConnection* should refuse and immediately release *newConnection*. Use this method to limit the amount of NSConnections created in your application or to change the parameters of child NSConnections.

This delegate method is obsolete, and shouldn't be used. Use **connection:shouldMakeNewConnection:** instead.

Notifications

NSConnectionDidDieNotification

Posted when the NSConnection is deallocated or when it's notified that its NSPort has become invalid. The notification contains:

Notification Object	The NSConnection object.

The NSConnection object posting this notification is no longer useful, so all receivers should unregister themselves for any notifications involving the NSConnection.

See also: **NSPortDidBecomeInvalidNotification** (NSPort)

 NSConnectionDidInitializeNotification

Posted when the NSConnection is initialized using **initWithReceivePort:sendPort:** (the designated initializer for NSConnection). The notification contains:

Notification Object	The NSConnection object.

See also: initWithReceivePort:sendPort: