NXConnection

Inherits From:	NXInvalidationNotifier (Mach Kit) : Object
Conforms To:	NXSenderIsInvalid NXReference (NXInvalidationNotifier)
Declared In:	remote/NXConnection.h

Class Description

The NXConnection class is used to establish a connection that allows objects in one process to send messages to objects in another process, and it defines instances that manage the local side of such a connection.

To establish a connection, some object must first be registered with the Network Name Server using **registerRoot:withName:**. This creates an NXConnection and makes the given root object available (through **connectToName:**) to any application that knows the registered name.

NXConnection objects can also be automatically created by the system. When a proxy is vended to an application, the application doesn't receive a proxy to the proxy. Instead, a new connection is formed if necessary, and the application receives a proxy to the original object. The delegate method **connection:didConnect:** is used to inform the application of the automatic creation of new connections.

An NXConnection maintains a table containing an NXProxy object for every local object that has been vended. It also maintains a table of remote NXProxy objects; these proxies are used to send messages to real objects that exist in other applications. A local NXProxy is created automatically by an NXConnection when a local object is vended to another application. Similarly, a remote NXProxy is created automatically when a remote object is vended to the NXConnection; this remote proxy forwards the messages it receives to its corresponding real object, with the effect that it generally appears to be the real object to the local application.

Running a Connection

When a connection is created, it is able to originate messages, and it sends these messages out to a port known as its *out-port* (available though the **outPort** method). Having sent a message, the connection will generally need to receive a reply message, which comes in over the connection's *in-port*. While it awaits this reply, the connection may dispatch messages in response to other messages that appeared on its in-port. However, once the desired reply is found, the connection will return its thread of control back to the caller, and the connection won't be able to receive unsolicited messages. In order to wait on unsolicited messages, a connection must be run, a process that involves waiting for messages on its in-port. The connection's thread is unavailable for other tasks while it runs. For this reason, there are a variety of run methods that allow a connection to run concurrently from the event loop, in its own thread, or for a limited period of time. The run methods are:

– run
– runWithTimeout:
– runInNewThread
– runFromAppKit
– runFromAppKitWithPriority:

A connection can receive remote messages from connections running in other threads or processes, and it will queue up these messages and dispatch them locally from its own thread. However, you cannot run a connection in one thread and send outgoing two-way messages over that connection from another thread; the process of running the connection has the connection's thread waiting on the in-port, so this port is not available for a return message for the caller's thread.

Instance Variables

id **delegate**

delegate	The connection's delegate

Adopted Protocols

NXSenderIsInvalid	– senderIsInvalid:

Method Types

Establishing a connection	+ connectToName: + connectToName:fromZone: + connectToName:onHost: + connectToName:onHost:fromZone: + connectToPort: + connectToPort:fromZone: + connectToPort:withInPort: + connectToPort:withInPort:fromZone:
Ascertaining connections	+ connections:
Registering an object	+ registerRoot: + registerRoot:fromZone: + registerRoot:withName: + registerRoot:withName:fromZone:
Eliminating references	+ removeObject:
Invalidation	+ unregisterForInvalidationNotification:
Statistics	+ messagesReceived
Timeouts	+ setDefaultTimeout: + defaultTimeout – setInTimeout: – setOutTimeout: – inTimeout – outTimeout
Zone usage	+ setDefaultZone: – defaultZone
Assigning a delegate	– setDelegate: – delegate
Returning port objects	– inPort – outPort
Getting and setting the root object	– rootObject – setRoot:
Imported and exported objects	– remoteObjects – localObjects
Returning a proxy	– getLocal: – newRemote:withProtocol:

Running a connection	– run – runWithTimeout: – runInNewThread – runFromAppKit – runFromAppKitWithPriority:
Freeing an NXConnection instance	– free

Class Methods

connections:

+ **connections:**(List *) *aList*

Adds all the application's NXConnections to the supplied list *aList* (but doesn't delete its prior contents). A reference is added to every connection in the list. Returns *aList*.

connectToName:

+ (NXProxy *)**connectToName:**(const char *)*rootName*

Returns an NXProxy to the object registered with the Network Name Server as *rootName*. This method is a cover for **connectToName:onHost:fromZone:** with a null host-name and using the NXConnection class's default zone.

connectToName:fromZone:

+ (NXProxy *)**connectToName:**(const char *)*rootName* **fromZone:**(NXZone *)*zone*

Returns an NXProxy to the object registered with the Network Name Server as *rootName*. This method is a cover for **connectToName:onHost:fromZone:** with a null host-name and using the specified zone *zone*.

connectToName:onHost:

+ (NXProxy *)**connectToName:**(const char *)*rootName*
onHost:(const char *)*hostName*

Returns an NXProxy to the object registered with the Network Name Server as *rootName*. This method is similar to **connectToName:onHost:fromZone:** using the NXConnection class's default zone.

connectToName:onHost:fromZone:

+ (NXProxy *)**connectToName:**(const char *)*rootName*
onHost:(const char *)*hostName*
fromZone:(NXZone *)*zone*

Returns an NXProxy to the object registered with the Network Name Server as *rootName*, or **nil** if no connection can be established. Functionally, this method can be thought to return that root object. If *hostName* is explicitly specified, this method queries the Network Name Server on *hostName* for the object registered under *rootName*. If *hostName* is NULL, this method queries the Network Name Server on the local host. If *hostName* is "*", this method will query the Network Name Server on each machine on the subnet until it finds an object registered under *rootName*. Note that querying each machine on a subnet can take a bit of time, so if the host is known, it should be specified.

In addition to creating and returning an NXProxy, this method creates an NXConnection. If this connection will be used to receive remote messages (as is the common case), you will need to run it by sending it a variation of the **run** message. A connection that isn't run will dispatch incoming messages only while it awaits a callback in response to a locally initiated message, so unsolicited remote messages will not be handled in a timely manner. To get the connection of the returned proxy (in order to run it), use NXProxy's **connectionForProxy** method.

If *zone* is specified, the objects associated with the new connection will be allocated from that zone; if *zone* is NULL they will be allocated from the NXConnection class's default zone.

See also: + **registerRoot:withName:**, – **runFromAppKit**, – **connectionForProxy** (NXProxy)

connectToPort:

+ (NXProxy *)**connectToPort:**(NXPort *)*aPort*

Returns an NXProxy to the root object for the connection identified with the port *aPort*, or **nil** if no connection can be established. This method is a cover for **connectToPort:fromZone:** using the NXConnection class's default zone.

connectToPort:fromZone:

+ (NXProxy *)**connectToPort:**(NXPort *)*aPort* **fromZone:**(NXZone *) *zone*

Returns an NXProxy to the root object for the connection identified with the port *aPort*, or **nil** if no connection can be established. You can use this method to establish a connection

based on a port you are vended. In other words, you can use this method to establish a connection based on another connection's out-port that is handed to your application.

If *zone* is specified, the objects associated with the new connection will be allocated from that zone; if *zone* is NULL they will be allocated from the NXConnection class's default zone.

See also: + **connectToName:onHost:**, – **outPort**, + **connectToPort:withInPort:**

connectToPort:withInPort:

+ (NXProxy *)**connectToPort:**(NXPort *)*aPort* **withInPort:**(NXPort *)*inPort*

Returns an NXProxy to the root object for the connection identified with the port *aPort*, or **nil** if no connection can be established. This method is a cover for **connectToPort:withInPort:fromZone:** using the NXConnection class's default zone.

connectToPort:withInPort:fromZone:

+ (NXProxy *)**connectToPort:**(NXPort *)*aPort*
withInPort:(NXPort *)*inPort*
fromZone:(NXZone *)*zone*

Returns an NXProxy to the root object for the connection identified with the port *aPort*, or **nil** if no connection can be established. The supplied port *inPort* will be used to receive incoming messages.

If *zone* is specified, the objects associated with the new connection will be allocated from that zone; if *zone* is NULL they will be allocated from the NXConnection class's default zone.

See also: + **connectToName:onHost:**, + **connectToPort:**

defaultTimeout

+ (int)**defaultTimeout**

Returns the default connection timeout interval in milliseconds. The interval is 15000 milliseconds unless set to some other value by **setDefaultTimeout:**. A connection will initially use the default timeout interval for both its input and output ports; however, these values can be changed for any port using the **setInTimeout:** or **setOutTimeout:** method.

defaultZone

+ (NXZone *)**defaultZone**

Returns the default zone for all connections. If a zone isn't specified when a connection is created, memory (and objects) associated with the connection will be allocated from this zone. The default zone is initially set to **NXDefaultMallocZone()**, but can be set to another zone using **setDefaultZone:**.

messagesReceived

+ (int)**messagesReceived**

Returns the number of messages received by all connections in the application. This value can be helpful when you attempt to optimize an application's performance by minimizing remote messages.

registerRoot:

+ **registerRoot:***anObject*

Establishes *anObject* as a root object, creating a new NXConnection if necessary. This method is a cover for **registerRoot:fromZone:** using the NXConnection class's default zone.

registerRoot:fromZone:

+ **registerRoot:***anObject* **fromZone:**(NXZone *)*zone*

Establishes *anObject* as a root object, creating a new NXConnection if necessary. *anObject* isn't advertised by the Network Name Server, though you can allow other objects to access it by vending its in-port to private clients, who can then connect to that port using **connectToPort:**. Returns *anObject*'s NXConnection, which must then receive a variant of the **run** message to receive unsolicited remote messages and forward them to *anObject*.

If *zone* is specified, the objects associated with the new connection will be allocated from that zone; if *zone* is NULL they will be allocated from the NXConnection class's default zone.

See also: + **registerRoot:withName:**, – **runFromAppKit**, – **inPort**

registerRoot:withName:

+ **registerRoot:***anObject* **withName:**(const char *)*name*

Establishes *anObject* as a root object, creating a new NXConnection if necessary. This method is a cover for **registerRoot:withName:fromZone:** using the NXConnection class's default zone.

registerRoot:withName:fromZone:

+ **registerRoot:***anObject*
  withName:(const char *)*name*
  fromZone:(NXZone *)*zone*

Establishes *anObject* as a root object, creating a new NXConnection if necessary. *anObject* is advertised by the Network Name Server with the name *name*. Returns *anObject*'s NXConnection, which must then receive a variant of the **run** message to pass remote messages to **anObject**.

If *zone* is specified, the objects associated with *anObject*'s connection will be allocated from that zone; if *zone* is NULL they will be allocated from the NXConnection class's default zone.

See also: – **runFromAppKit**

removeObject:

+ **removeObject:***anObject*

Removes all proxies to *anObject*. If *anObject* has been vended to clients, the clients hold proxies for it which ought to be removed before *anObject* is destroyed. You will therefore probably need to invoke **removeObject:** in *anObject*'s **free** method to avoid dangling references and memory leaks. Returns **self**.

setDefaultTimeout:

+ **setDefaultTimeout:**(int)*interval*

Sets the default connection time interval to *interval*. A connection initially uses this interval for both its input and output ports; however, these values can be changed for any port using the **setInTimeout:** or **setOutTimeout:** method.

See also: + **defaultTimeout**

setDefaultZone:

+ **setDefaultZone:**(NXZone *)*zone*

Sets the default zone for all connections. If a zone isn't specified when a connection is created, memory (and objects) associated with the connection will be allocated from this zone. The default zone is initially set to **NXDefaultMallocZone()**.

See also: + **defaultZone**

unregisterForInvalidationNotification:

+ **unregisterForInvalidationNotification:***anObject*

Unregisters *anObject* so it won't be notified of the invalidation of any of its connections.

See also: – **unregisterForInvalidationNotification:** (NXInvalidationNotifier),
– **registerForInvalidationNotification:** (NXInvalidationNotifier)

Instance Methods

delegate

– **delegate**

Returns the connection's delegate.

free

– **free**

Removes a reference to the connection. If outstanding references remain, the NXConnection isn't actually freed and this method returns **self**. If no references remain, this method frees the NXConnection and the proxies it maintains and returns **nil**.

getLocal:

– **getLocal:***anObject*

Returns the local NXProxy for *anObject*, or **nil** if *anObject* isn't represented by a local proxy on the receiving NXConnection. Vending *anObject*'s local proxy is essentially the same as vending *anObject* itself except that by vending the local proxy you determine the connection over which *anObject* is referenced.

inPort

– (NXPort *)**inPort**

Returns the connection's in-port, the NXPort used by the connection to receive incoming messages.

inTimeout

– (int)**inTimeout**

Returns the timeout interval (in milliseconds) for incoming messages. A value of –1 means the connection will wait forever for incoming messages.

See also: – **setInTimeout:**, – **outTimeout**

localObjects

– (List *)**localObjects**

Creates and returns a List of the proxies to local objects vended by the connection. The proxies belong to the connection and should not be altered, but the returned List should be freed by the sender of this message.

newRemote:withProtocol:

– **newRemote:**(unsigned int)*anObject* **withProtocol:**(Protocol *)*proto*

Creates and returns a remote proxy for the local object identified by *anObject*. This proxy can then be given to other objects to vend *anObject* over the receiving connection. *anObject* is the **id** of the local object, though you must cast it to an unsigned integer to satisfy the implementation. *proto*, if non-NULL, is used to specify the protocol that *anObject* responds to; performance is increased if the protocol is specified because a round-trip message to fetch argument types (for encoding purposes) is obviated.

outPort

– (NXPort *)**outPort**

Returns the connection's out-port, the NXPort object used to identify the remote port (and connection) that the receiving connection communicates with. This NXPort can be used to create a new connection by **connectToPort:**.

outTimeout

– (int)**outTimeout**

Returns the timeout interval (in milliseconds) for outgoing messages. A value of –1 means outgoing messages will never time out.

See also: – **setOutTimeout:**, – **inTimeout**

remoteObjects

– (List *)**remoteObjects**

Creates and returns a List of the proxies to remote objects maintained by the receiving connection. The proxies belong to the connection and should not be altered, but the returned List should be freed by the sender of this message. If the connection becomes invalid, objects in the application will no longer be able to send remote messages to the objects in this List.

See also: – **localObjects**

rootObject

– **rootObject**

Returns the connection's root object, which is the object returned (by way of a proxy) to other applications when they connect to the NXConnection.

See also: + **registerRoot:withName:**, – **setRoot**

run

– **run**

Runs the connection by waiting for messages and dispatching them. This method runs in the same thread that it was invoked from, and it doesn't return until the connection is invalidated. If the connection becomes invalid, this method returns **self**. This method is a cover for **runWithTimeout:** with an argument of –1.

See also: – **runFromAppKit**, – **runInNewThread**, – **runWithTimeout:**

runFromAppKit

– **runFromAppKit**

Runs the connection by waiting for messages and dispatching them. This method adds the connection's port to those that the DPS client library monitors for messages, at a priority of NX_RUNMODALTHRESHOLD. When a message arrives over the connection, it will be handled between events. The connection isn't really run concurrent to the application, but the effect is close enough to concurrency for most uses.

This method is typically the best way to run a connection that will dispatch messages to objects that use the Application Kit or Window Server, since these objects cannot be messaged from multiple threads. (Note, however, that the connection run from the DPS client library can communicate with connections running in separate threads.)

This method immediately returns **self**.

See also: – **run**, – **runFromAppKitWithPriority:**, – **runInNewThread**, – **runWithTimeout:**

runFromAppKitWithPriority:

– **runFromAppKitWithPriority:**(int)*priority*

Runs the connection by waiting for messages and dispatching them. This method adds the connection's port to those that the DPS client library monitors for messages, at a priority of *priority.* Otherwise this method is identical to **runFromAppKit**.

runInNewThread

– **runInNewThread**

Runs the connection by waiting for messages and dispatching them. This method forks a new thread that invokes the **run** method; it then immediately returns **self**. All messages sent to this connection are dispatched by the new thread. Because the Window Server and Application Kit aren't thread-safe, you shouldn't send messages to a connection in a separate thread that call upon them. If you need some concurrency in a connection that will invoke the Window Server or Application Kit, you should use **runFromAppKit**.

See also: – **runFromAppKit**, – **run**, – **runWithTimeout:**

runWithTimeout:

– **runWithTimeout:**(int)*timeout*

Runs the connection by waiting for messages and dispatching them. This method runs for *timeout* milliseconds or until the connection is invalidated before returning **self**. If *timeout* is (-1) the connection will run forever or until it is invalidated, whichever occurs first.

See also: – **runFromAppKit**, – **runInNewThread**, – **run**

senderIsInvalid:

– **senderIsInvalid:***sender*

Responds to a message that the connection's port has died. This method invalidates the connection, invalidates the proxies to remote objects (which can no longer be accessed), and sends a **free** message to all the local objects vended by the connection that conform to the NXReference protocol, thereby giving up the connection's references to these objects. *sender* is an instance of a private port management class; your code shouldn't send messages to it.

setDelegate:

– **setDelegate:***anObject*

Sets the connection's delegate. Returns **self**.

setInTimeout:

– **setInTimeout:**(int)*timeout*

Sets the connection's timeout for incoming messages to *timeout* milliseconds. This is the amount of time the connection will wait for return parameters, return values, callbacks, and the like. If a message isn't received before the timeout, an exception will be raised. Setting *timeout* to −1 results in an infinite timeout interval. Returns **self**.

See also: + **setDefaultTimeout:**, – **setOutTimeout:**

setOutTimeout:

– **setOutTimeout:**(int)*timeout*

Sets the connection's timeout for outgoing messages to *timeout* milliseconds. This is the amount of time the connection will wait for a message send to succeed. If an outgoing message can't be sent before the timeout, an exception will be raised. Setting *timeout* to –1 results in an infinite timeout interval, and setting it to 0 has the effect that a message will be delivered only if the receiver's port has room. Returns **self**.

See also: + **setDefaultTimeout**, – **setInTimeout**

setRoot:

– **setRoot:***anObject*

Sets the connection's root object to *anObject*. This method should be invoked only for a connection that doesn't have a root object.

See also: – **rootObject**

Methods Implemented By The Delegate

connection:didConnect:

– **connection:**(NXConnection *)*conn* **didConnect:**(NXConnection *)*newConn*

Notifies *conn*'s delegate that a new connection has been established using *conn*'s input port. *newConn* is the NXConnection object that was just created. This method must return the NXConnection object that should be used, which is typically *newConn*; if another connection is returned, the application is responsible for freeing *newConn*.