MiscINETSocket

Inherits From: MiscSocket : Object

Declared In: MiscINETSocket.h

Class Description

This class encapsulates sockets in the Internet domain, and offers operations specific to such sockets. See the documentation for the MiscSocket class for more information on sockets and socket types.

Sockets and Connections

The reader should keep in mind that the text which follows is generalization and simplification to make it more useful as an introduction to Internet sockets.

A network connection (in the {TCP,UDP}/IP protocol model) is distinguished by four pieces of information: the local address, the local port number, the remote address, and the remote port number. The addresses are each one of the addresses of the local and remote hosts involved in the connection (the remote host may be the same host as the local host, but by convention the host to which a connection is made is the remote host). There are two general states in which a socket is useful for network connections: bound and connected (a third state, closed, isn't very useful if you want to do something with the socket). To bind a socket is to assign a local address and local port number to the socket, but no remote address or port. To connect a socket means to form a connection between two hosts, the remote address and port to which to connect are specified, and the system fills in a local address and chooses a local port number.

A bound socket (often referred to as a *passive* socket in the literature) acts something like an electrical

outlet for network connections; a connection formed to a remote host must connect to a socket which has been bound on that host, just as when you want some toast, you must plug your toaster into an electrical outlet (plugging it into the sink doesn't get you toast). This analogy doesn't go much further, however; when an un-bound, un-connected socket forms a connection to a bound socket, a new socket for the remote end of the connection is created on the remote host, a new socket which is in the connected state. The bound socket remains as before, waiting to accept new connections. A socket which is connected (referred to as an *active* socket) can participate in the transfer of data on the connection. Data can be read and written from the socket in much the same way as it can be read and written from a file (with some limitations and differences; you can't seek on a socket, for instance). The system takes care of getting the data to and from the remote host, using whatever transfer method you have specified: datagrams, streams, etc. (this is called the *type* of the socket in this documentation; see the documentation for the MiscSocket class for more information).

Notes on this Class

Instances of the class are reusable, in that they may be sent multiple initialization messages. This may allow an application to avoid some memory allocation and deallocation in some cases. The

initialization messages of this class are **init**, **connectTo:port:type:**, **connectTo:service:type:**, **openServerPort:type:**, and **openServerService:**. These last four, while not conforming to the init- naming convention, are called initialization method because they behave just like an initialization method and can be used as such. Each of those last four methods calls the **init** method as one of the first things it does±this closes the instance if it is open, and initializes it in preparation for forming a connection or listening for connections. Additionally, they free the instance if there is an error (and return nil).

There are four class methods that send and receive datagrams, the "**sendDgram:**..." methods. These methods have been implemented to provide efficient datagram operations, when a small number of datagrams need to be sent by an application. If an application needs to send more than two datagrams to the same destination in a short period of time, a forming a connection and sending on it would be faster.

Methods implemented in this class generally return self on success and nil upon failure, or a small integer if there is a timeout parameter. When nil (or -1) is returned, the value of the *errno* global variable can be examined for the cause; values that this variable may take on are described in the

header file <sys/errno.h> and the UNIX manual page *intro*(2). The list of possible values in errno is meant to be extensive, but not exhaustive. Asynchronous errors on the socket, such as ECONNRESET (remote host has forced the connection to be closed), may result at nearly any time, and may not be related to the operation just attempted. If a method is successful, the previous value of *errno* is preserved.

Other references

Here are a few good references on sockets and network communication:

Comer, Douglas E. *Internetworking with TCP/IP, Volume 1: Principles, Protocols, and Architecture*. Prentice-Hall, Englewood Cliffs, New Jersey. 1991.

Comer, Douglas E. and David L. Stevens. *Internetworking with TCP/IP, Volume 3: Client-Server Programming and Applications* (BSD Socket edition). Prentice-Hall, Englewood Cliffs, New Jersey. 1993.

Stevens, W. R. *UNIX Network Programming*. Prentice-Hall, Englewood Cliffs, New Jersey. 1990.

Instance Variables

MiscINETAddress ***localAddress**	Local end point address of current connection
int **localPortNum**	Local end point port number of current connection
MiscINETAddress ***remoteAddress**	Remote end point address of current connection
int **remotePortNum**	Remote end point port number of current connection

Method Types

Initializing instances	- init
Initializing a connected socket	- connectTo:port:type: - connectTo:service:type:
Initializing a bound socket	- openServerPort:type: - openServerService:type:

Accepting connections	- acceptNewConnection:timeout:
Copying instances	- copyFromZone:
Sending/receiving data	- receiveData:length: - receiveData:length:timeout:from:port: - receiveData:length:timeout:toNext: - sendData:length: + sendDgram:length:to:port: + sendDgram:length:to:port:timeout:retries:withReply:length: + sendDgram:length:to:service: + sendDgram:length:to:service:timeout:retries:withReply:length:
Closing connection endpoints	- shutdownLocalEnd - shutdownRemoteEnd
Assessing accessibility	+ ping:timeout:useECHO:
Listening for connections	+ runServer:target:action:fork:loop:

Accessing socket options	- debugEnabled - delayEnabled - enableDebug: - enableDelay: - enableKeepAlive: - keepAliveEnabled - lingerTime - setLingerTime:
Information about a socket	- dataAvailable - localAddress - localPortNum - remoteAddress - remotePortNum - socketError

Class Methods

ping:timeout:useECHO:

+ **ping:**(MiscINETAddress *)addr
  timeout:(unsigned int)ms
  useECHO:(BOOL)aBool

Sends a message to the host specified by *addr* and waits for a reply. The message can either be the usual "ping" message, using the ICMP protocol, or it can be a datagram sent to the ECHO port on the host *addr*. Only the superuser can send ICMP messages, so *aBool* will most often be YES for user programs. The amount of time to wait in milliseconds can be specified with the *ms* parameter; a zero value means no timeout (infinite wait). Only one message is sent; the application is responsible for trying again if desired. Returns 1 on success (message sent and valid reply received), 0 on timeout, -1 on error (or invalid response). On error, the global *errno* may contain the following values: 0 (response received was invalid), EMFILE (the process is out of descriptors), ENFILE (the system descriptor table is full), ENOBUFS (system lacks resources), EDESTADDRREQ (addr was nil); *for the*

ICMP option only: EACCES (permission denied); *for the ECHO option only*: ENOENT (echo service unknown).

runServer:target:action:fork:loop:

+ **runServer:**(id)sockets
  target:(id)target
  action:(SEL)action
  fork:(BOOL)fFlag
  loop:(BOOL)lFlag

This method implements a fairly generic and typical server "event loop". Typically, a server initializes some sockets to the ports to which it wants to listen, and goes into a loop, servicing requests for service. In the case of a stream socket bound to a port, this means accepting new connections from that socket, and handling requests received from the new connection. Bound datagram sockets are simply listened to, and requests read directly from them. This method encapsulates this activity, allowing the author of a server to ignore many of these details for many typical servers.

The *sockets* parameter is either a socket instance to which to listen, or an instance of HashTable (or subclass thereof) of socket instances, which are indexed by their socket descriptors. The socket (or sockets) are all listened to simultaneously (not in a busy wait). When there is activity on one or more sockets, the *target* is sent the *action* message with the socket instance in question as the only parameter. In the case of a stream socket, the new connection is accepted and the newly created socket for this new connection is passed as the parameter, and this instance is freed when the *action* method returns. There are no provisions for adding sockets dynamically (while this method is executing) to the list of sockets monitored. If *lFlag* is true, the server then goes back to listening for activity, looping forever (or until an error occurs). The *fFlag* parameter dictates whether or not the server forks a child to handle the service request; this is desirable for stream-oriented servers where the connection will be long-lived; after the fork, the child performs the *action* method (and exits when that method returns). If *fFlag* is false, no forking occurs (the *action* method could do the fork itself, depending on the type of socket the activity occurred on, for instance). There are also no provisions for timeout on the length of the waiting for activity.

Does not return if *lFlag* is true and no error occur, otherwise returns self on success. On error, nil is returned, and the global *errno* may contain the following values: EINVAL(*target* does not respond to

action, or *sockets* is neither an instance of MiscSocket (or a subclass) or HashTable (or a subclass), or the socket(s) are all closed), EBADF (one of the sockets specified has closed), ECONNABORTED (one of the sockets is closing), EAGAIN (the fork failed).

sendDgram:length:to:port:

+ **sendDgram:**(void *)data
  length:(int)dlen
  to:(MiscINETAddress *)addr
  port:(int)portNum

Sends a datagram containing data of length *dlen* starting at memory pointed to by *data* to the remote host specified by *addr* and port specified by *portNum*. Returns self on success, nil on error. Since datagrams do not guarantee reliable delivery in the current implementation, the success or failure of the remote host to receive the message is not indicated (all errors are local host errors). On error, the global *errno* may contain the following values: EMFILE (the process is out of descriptors), ENFILE (the system descriptor table is full), ENOBUFS (system lacks resources), EFAULT (invalid *data*

pointer), EMSGSIZE (message size too large), EDESTADDRREQ (*addr* was nil).

sendDgram:length:to:port:timeout:retries:withReply:length:

+ (int)**sendDgram:**(void *)data
  length:(int)dlen
  to:(MiscINETAddress *)addr
  port:(int)portNum
  timeout:(unsigned int)ms
  retries:(int)retries
  withReply:(void *)repl
  length:(int *)rlen

Sends a datagram containing data of length *dlen* starting at memory pointed to by *data* to the remote host specified by *addr* and port specified by *portNum* and waits for a reply. The amount of time to wait in milliseconds can be specified with the *ms* parameter; a zero value means no timeout (infinite wait). The method will make multiple attempts if *retries* is greater than zero. Note that the timeout

value is for each attempt, not for all attempts cumulatively. Returns 1 on success, 0 on timeout, and -1 on error. Since datagrams do not guarantee reliable delivery in the current implementation, the success or failure of the remote host to receive the message is not indicated (the remote host may have received all attempts). On error, the global *errno* may contain the following values: EMFILE (the process is out of descriptors), ENFILE (the system descriptor table is full), ENOBUFS (system lacks resources), EFAULT (invalid *data* pointer), EMSGSIZE (message size too large), EDESTADDRREQ (*addr* was nil).

sendDgram:length:to:service:

+ **sendDgram:**(void *)data
  length:(int)dlen
  to:(MiscINETAddress *)addr
  service:(const char *)service

Translates the service (which may be a service name or a port number) specified by *service* into a port number, and calls **sendDgram:length:to:port:**. In addition to the error conditions of that method, nil

is returned if *service* cannot be mapped into a port number. Additionally, on error, the global *errno* may contain the following value: ENOENT (service unknown).

sendDgram:length:to:service:timeout:retries:withReply:length:

+ (int)**sendDgram:**(void *)data
  length:(int)dlen
  to:(MiscINETAddress *)addr
  service:(const char *)service
  timeout:(unsigned int)ms
  retries:(int)retries
  withReply:(void *)repl
  length:(int *)rlen

Translates the service (which may be a service name or a port number) specified by *service* into a port number, and calls **sendDgram:length:to:port:timeout:retries:withReply:length:**. In addition to the error conditions of that method, -1 is returned if *service* cannot be mapped into a port number.

Additionally, on error, the global *errno* may contain the following value: ENOENT (service unknown).

Instance Methods

acceptNewConnection:timeout:

- (int)**acceptNewConnection:**(MiscINETSocket **)newSocket **timeout:**(unsigned int)ms;

Waits for a connection request to arrive at the socket represented by the receiver. The socket must be of stream type that has been bound locally with the **openServerPort:type:** or **openServerService:type:** methods. When a new connection arrives, a new instance (of stream type) representing the new connection is created and returned by reference in *newSocket*. If the timeout period expires or an error occurs, the value pointed to by *newSocket* is set to nil. The receiver is not modified. The amount of time to wait in milliseconds can be specified with the *ms* parameter; a zero value means no timeout (infinite wait). Returns 1 on success, 0 on timeout, and -1 on error. On error, the global *errno* may contain the following values: EBADF (socket is closed), EINVAL (socket not of

stream type), ECONNABORTED (socket is closing).

connectTo:port:type:

- **connectTo:**(MiscINETAddress *)addr
  port:(int)portNum
  type:(int)aType

Closes the socket instance if it is currently open, and makes a connection to the host specified by *addr* at the port indicated. Returns self if successful. The receiver is freed and nil is returned if *aType* is not valid (for instance, a raw socket is requested and the process does not have the appropriate permissions), *portNum* is less than zero, or a connection to the remote host and port could not be made. On error, the global *errno* may contain the following values: ESOCKTNOSUPPORT (invalid type value), EPROTONOSUPPORT (type not supported for Internet sockets), EMFILE (the process is out of descriptors), ENFILE (the system descriptor table is full), EACCES (permission is denied), ENOBUFS (system lacks resources), EADDRNOTAVAIL (local address/port not available), EADDRINUSE (remote address/port already in use), ETIMEDOUT (connection attempt timed out),

ECONNREFUSED (remote end refused connection), ENETUNREACH (network is unreachable), EHOSTUNREACH (host is unreachable), EDESTADDRREQ (*addr* was nil).

connectTo:service:type:

- **connectTo:**(MiscINETAddress *)addr
  service:(const char *)service
  type:(int)aType

Translates the service (which may be a service name or a port number) specified by *service* into a port number, and calls **connectTo:port:type:**. Returns self if successful. In addition to the error conditions of that method, the receiver is freed and nil is returned if *service* cannot be mapped into a port number, or *aType* specifies a raw socket type. Additionally, on error, the global *errno* may contain the following values: ESOCKTNOSUPPORT (socket type is raw), ENOENT (service unknown), EADDRNOTAVAIL (local service port not available), EADDRINUSE (remote service port already in use).

copyFromZone:

- **copyFromZone:**(NXZone *)zone

Returns a copy of the receiver. See the cautionary message in this class's superclass documentation of this method for more information.

dataAvailable

- (BOOL)**dataAvailable**

Returns YES if there is data to be read from the socket, or NO if there isn't or the socket is closed.

debugEnabled

- (BOOL)**debugEnabled**

Returns YES if the delay option on the socket has been enabled, or NO if it has been disabled (the default), or the socket is closed.

delayEnabled

- (BOOL)**delayEnabled**

Returns NO if the delay option on the socket has been disabled, or YES if it has been enabled (the default), or the socket is closed.

enableDebug:

- **enableDebug:**(BOOL)aBool

Sets the state of the debug option on a socket, if the socket is not closed, and returns self. This option can be used to enable debugging in the underlying protocol layers and turn on the output of debugging information from them.

enableDelay:

- **enableDelay:**(BOOL)aBool

Sets the state of the delay option on a stream socket, if the socket is not closed, and returns self. Stream connections usually transmit data when it is received from the application. If this option is enabled (which it is by default), whenever there is data that has been sent, but not yet acknowledged by the other end of the connection, data that is received from the application is buffered (if the amounts are small) to be sent when the acknowledgement arrives. For a few clients or servers, this delay may be unacceptable, and this option can be disabled. If the socket is not of stream type, does nothing.

enableKeepAlive:

- **enableKeepAlive:**(BOOL)aBool

Sets the state of the keep-alive option on a stream socket, if the socket is not closed, and returns self. If this option is enabled, periodic messages will sent between the lower protocol levels on the local and remote hosts to ensure that the connection remains active. If the remote end of the connection does not respond to the messages, the system considers the connection to be broken, and the process will receive a SIGPIPE signal to notify it of this. Note that the socket object is not closed automatically if the connection fails; the application must attend to that itself. By default, this option is disabled. If

the socket is not of stream type, does nothing.

init

- **init**

Places the instance in the closed (initialized) state, and returns self. This method may be applied multiple times to an instance, so that instances may be reused. This method is the designated initializer of MiscINETSocket instances. Subclasses should implement their own version of this method (beginning with [super init]) to initialize its instance variables. To maintain the multiple-initialization property, the method should check the value of instance variables and free allocated objects and memory as part of the initialization, and extended **init-** methods in the class should call [self init] first thing in the method.

keepAliveEnabled

- (BOOL)**keepAliveEnabled**

Returns YES if the keep-alive option on the socket has been enabled, or NO if it has been disabled (the default), or the socket is closed.

lingerTime

- (int)**lingerTime**

Returns the linger time as set by **setLingerTime:**, or 0 if linger has been disabled (the default), or the socket is closed.

localAddress

- (MiscINETAddress *)**localAddress**

Returns the address of the local end of the socket, or nil if the socket is not open nor connected or bound. On error, the global *errno* may contain the following values: EBADF (socket is closed), ENOBUFS (system lacks resources).

localPortNum

- (int)**localPortNum**

Returns the port number of the local end of the socket, or -1 if the socket is not open nor connected or bound. On error, the global *errno* may contain the following values: EBADF (socket is closed), ENOBUFS (system lacks resources).

openServerPort:type:

- **openServerPort:**(int *)portNum
  type:(int)aType

Closes the instance and reopens it, bound to the port specified. *portNum* should have been initialized to the desired port, or zero if any port is acceptable (one will be assigned by the system). The port is bound to all addresses of the local host. Returns self if successful. The instance is freed and nil is returned if *aType* is not valid (for instance, a raw socket is requested and the process does not have the appropriate permissions), the value pointed to by *portNum* is less than zero, or the socket could not be

bound to the local port (for instance, if it is already in use). The global *errno* may contain the following values: ESOCKTNOSUPPORT (invalid type value), EPROTONOSUPPORT (type not supported for Internet sockets), EMFILE (the process is out of descriptors), ENFILE (the system descriptor table is full), EACCES (permission is denied), ENOBUFS (system lacks resources), EADDRNOTAVAIL (port not available), EADDRINUSE (port already in use).

openServerService:type:

- **openServerService:**(const char *)service
  type:(int)aType

Translates the service (which may be a service name or a port number) specified by *service* into a port number, and calls **openServerPort:type:**. Returns self if successful. In addition to the error conditions of that method, the receiver is freed and nil is returned if *service* cannot be mapped into a port number, or *aType* specifies a raw socket type. Additionally, on error, the global *errno* may contain the following values: ESOCKTNOSUPPORT (socket type is raw), ENOENT (service unknown), EADDRNOTAVAIL (service port not available), EADDRINUSE (service port already in use).

receiveData:length:

- (int)**receiveData:**(void *)data
  length:(int *)dlen

This method is a cover for **receiveData:length:timeout:from:port:** with an infinite timeout and which ignores the remote address and port number from which the data was received. Returns the same values and has the same error conditions as that method.

receiveData:length:timeout:from:port:

- (int)**receiveData:**(void *)data
  length:(int *)dlen
  timeout:(unsigned int)ms
  from:(MiscINETAddress *)addr
  port:(int *)port

Waits for data to become available on the socket, then returns the data in the memory pointed to by

data. The integer pointed to by *dlen* should have been initialized to the maximum amount of data desired; on return, it contains the amount of data actually returned. For datagram sockets, an entire message is returned; if the maximum size specified is less than the size of the message, the excess (of the message) is discarded by the system. For stream sockets, whatever data is currently available (up to the specified maximum) is returned; any data beyond the maximum desired amount will be received on the next (and subsequent) receive(s). The amount of time to wait in milliseconds can be specified with the *ms* parameter; a zero value means no timeout (infinite wait). If the caller does not desire to know either (or both) of *addr* (the source address) or *port* (the source port), nil and NULL (respectively) may be passed for these parameters; normally, they would only be of interest to servers using datagram ports, where no connection is established. The method returns -1 on error, 0 if the timeout period expires before any data is received, and 1 if data is read. On error, the global *errno* may contain the following values: EBADF (socket is closed), EFAULT (invalid *data* pointer), ENOTCONN (tried to receive from a bound stream socket), EPIPE (remote host has closed connection). Note that the socket object is closed if the connection fails (EPIPE).

receiveData:length:timeout:toNext:

- (int)**receiveData:**(void *)data
  length:(int *)dlen
  timeout:(unsigned int)ms
  toNext:(unsigned char)sentinel

Waits for data to become available on the socket, then returns the data in the memory pointed to by *data*. The integer pointed to by *dlen* should have been initialized to the maximum amount of data desired; on return, it contains the amount of data actually returned. The data returned is the data that is available, up to and including the character *sentinel*. Note that no nul-terminator is written after the data (i.e., it should not be immediately used as a string). Available characters after sentinel will be read on the next (and subsequent) receive(s). The intent of this method is to provide a simple message-boundary capability to stream sockets; it will not work on any other type of socket. The amount of time to wait in milliseconds can be specified with the *ms* parameter; a zero value means no timeout (infinite wait). Returns -1 on error, 0 if the timeout period expires before a *sentinel* character is received, and 1 if data is read. Note that on error or timeout, some data may have been read and returned in *data*; the value at *dlen* should be checked. On error, the global *errno* may contain the following values: EBADF (socket is closed), EFAULT (invalid *data* pointer), ENOTCONN (tried to

receive from a bound stream socket), EPIPE (remote host has closed connection), EINVAL (socket not of stream type). Note that the socket object is closed if the connection fails (EPIPE).

remoteAddress

- (MiscINETAddress *)**remoteAddress**

Returns the address of the remote end of the socket, or nil if the socket is not open nor connected. On error, the global *errno* may contain the following values: EBADF (socket is closed), ENOTCONN (socket not connected), ENOBUFS (system lacks resources).

remotePortNum

- (int)**remotePortNum**

Returns the port number of the remote end of the socket, or -1 if the socket is not open nor connected. On error, the global *errno* may contain the following values: EBADF (socket is closed), ENOTCONN (socket not connected), ENOBUFS (system lacks resources).

sendData:length:

- **sendData:**(void *)data
  length:(int)dlen

Sends *dlen* bytes starting at *data* to the remote host if the socket is open. If the socket was opened for datagram service and the amount of data is small enough, the data written will be one datagram; if the amount of data is too large for one datagram, the system will break it into multiple datagrams. The datagram(s) may or may not arrive intact and unduplicated. For stream sockets, the data is appended to the stream of data at the remote end of the connection. The remote process will not necessarily read the data in the same chunks as it is presented to the system at the local end (there are no message boundaries with stream sockets). Returns self on success, or nil upon error. On error, the global *errno* may contain the following values: EBADF (socket is closed), ENOTCONN (send to a bound stream socket), EDESTADDRREQ (send to a bound datagram socket), EPIPE (remote host has closed connection), EMSGSIZE (message size too large), ESHUTDOWN (local end of socket has been shutdown). Note that the socket object is closed if the connection fails (EPIPE).

setLingerTime:

- **setLingerTime:**(int)secs

When a stream socket is closed, there may be unsent data in the local cache, waiting to be sent. If a positive timeout period in seconds has been set with this method and there is unsent data, closing the socket will block the process (or thread) until all data is transmitted or the timeout period expires. If the parameter *secs* is non-positive, then linger is disabled; in this case, the close operation will be performed so that the process continues as quickly as possible. In either case, the (system internal) stream connection is not actually closed until all data has been transmitted, or TCP gives up (say, due to a network failure). Linger is disabled by default. If the socket is closed, this method does nothing. Returns self.

shutdownLocalEnd

- **shutdownLocalEnd**

Closes the local end of a connected socket. Further sends on the socket will be disallowed, and the

remote end will not receive any more data (after the local send and remote receive buffers of the systems have been emptied). Returns self on success. If the socket is not connected, nil is returned. On error, the global *errno* may contain the following values: EBADF (socket is closed), ENOTCONN (socket not connected). The inherited **close** method should be used to close a connection completely.

shutdownRemoteEnd

- **shutdownRemoteEnd**

Closes the remote end of a connected socket. Further receives on the socket will be disallowed (after current the local receive buffers have emptied), and the remote end will not be able to send any more data. Returns self on success. If the socket is not connected, nil is returned. On error, the global *errno* may contain the following values: EBADF (socket is closed), ENOTCONN (socket not connected). The inherited **close** method should be used to close a connection completely.

socketError

- (int)**socketError**

Returns the value of the error status register of the socket, and clears the register. Some errors on sockets happen asynchronously with operations on the socket. For example, when a socket is closed the instance is reinitialzed, and the socket descriptor is freed. But there may be data in buffers in the lower protocol levels of the system which has not yet been sent. For a stream socket, providing reliable delivery, the system will continue to (try and) send this data before closing the connection. If after some timeout period the system gives up, the error condition is placed in the socket's error status register, if the socket descriptor has not been freed. (Ironically, in this example the error status is cannot be accessed because the descriptor *has* been freed.) If there is no error in the register, or the socket is not open, zero is returned.