Network Working Group
Internet-Draft
Intended status: Experimental
Expires: 18 March 2026

T. Dreibholz
SimulaMet

14 September 2025

NEAT Sockets API
draft-dreibholz-taps-neat-socketapi-17

Abstract

This document describes a BSD Sockets-like API on top of the callback-based NEAT User API. This facilitates porting existing applications to use a subset of NEAT's functionality.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 18 March 2026.

Table of Contents

1. Introduction

The NEAT project [12], [13], [5], [3], [8] wants to achieve a complete redesign of the way in which Internet applications interact with the network. Our goal is to allow network "services" offered to applications - such as reliability, low-delay communication or security - to be dynamically tailored based on application demands, current network conditions, hardware capabilities or local policies, and also to support the integration of new network functionality in an evolutionary fashion.

This document describes the NEAT Sockets API on top of the callback-based NEAT User API [4]. It provides a BSD Sockets-like API that facilitates porting existing applications to use a subset of NEAT's functionality. For further information on NEAT, see also [12], [13], [14], [15], [16].

1.1. Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [1].

2. Initialisation and Clean-Up

2.1. nsa_init()

nsa_init() is used to explicitly initialise the NEAT Sockets API. In the usual case, however, the NEAT Sockets API is automatically initialized when creating a NEAT socket. Explicit initialisation may only be necessary in a multi-threaded program, in order to avoid parallel initialisation calls.

Function Prototype:

int nsa_init()

Return Value:

nsa_init() returns the new NEAT socket descriptor, or -1 in case of error. The error code will be set in the errno variable.

2.2. nsa_cleanup()

nsa_cleanup() is used to free all resources allocated by NEAT. Note, that the NEAT Sockets API is automatically initialized when creating a NEAT socket.

Function Prototype:

void nsa_cleanup()

2.3. nsa_map_socket()

nsa_map_socket() is used to map a system socket descriptor into the NEAT socket descriptor space. This is useful for using NEAT API functions as wrapper to calls on non-NEAT sockets. Mapped socket descriptors can be unmapped by using nsa_unmap_socket().

Function Prototype:

int nsa_map_socket(int systemSD, int neatSD)

Arguments:

systemSD: System socket descriptor.

neatSD: Desired NEAT socket descriptor; -1 for automatic allocation.

Return Value:

nsa_map_socket() returns the new NEAT socket descriptor, or -1 in case of error. The error code will be set in the errno variable.

2.4. nsa_unmap_socket()

nsa_unmap_socket() is used to unmap a system socket descriptor from the NEAT socket descriptor space.

Function Prototype:

int nsa_unmap_socket(int neatSD)

Arguments:

neatSD: NEAT socket descriptor.

3. Connection Establishment and Teardown

3.1. nsa_socket()

nsa_socket() creates a new NEAT socket. The NEAT socket can either be a wrapper around the NEAT User API (if properties are specified) or be a wrapper around a system socket (if no properties are specified).

Function Prototype:

```
int nsa_socket(int domain, int type, int protocol,
               const char* properties)
```

Arguments:

domain: Domain for system socket (e.g. AF_INET).

type: Type for system socket (SOCK_SEQPACKET).

protocol: Protocol for system socket (IPPROTO_SCTP).

properties: Properties for NEAT Core socket.

Return Value:

nsa_socket() returns the NEAT socket descriptor in case of success, or -1 in case of error. The error code will be set in the errno variable.

See the socket() documentation for details.

3.2. nsa_socketpair()

nsa_socketpair() is a wrapper around the socketpair() call, returning NEAT socket descriptors instead. Note, that socketpair() only supports AF_UNIX sockets, i.e. this function is just a wrapper for the system function.

Function Prototype:

```
int nsa_socketpair(int domain, int type, int protocol,
                   const char* properties)
```

See the socketpair() documentation for details.

3.3. nsa_close()

nsa_close() closes a given NEAT socket.

Function Prototype:

```
int nsa_close(int sockfd)
```

Arguments:

sockfd: NEAT socket descriptor.

nsa_close() returns 0 in case of success, or -1 in case of error. The error code will be set in the errno variable.

See the close() documentation for details.

3.4. nsa_fcntl()

nsa_fcntl() manipulates a given NEAT socket.

Function Prototype:

int nsa_fcntl(int sockfd, int cmd, ...)

Arguments:

sockfd: NEAT socket descriptor.

cmd: Command.

...: Command-specific arguments.

nsa_fcntl() returns a command-specific value.

For NEAT sockets, the following commands are specified:

F_GETFL: Obtain value of the socket descriptor status flags. For NEAT sockets, the flag O_NONBLOCK specifies whether the socket is non-blocking. By default, it is blocking (i.e. O_NONBLOCK is not set).

F_SETFL: Set value of the socket descriptor status flags. For NEAT sockets, the flag O_NONBLOCK specifies whether the socket is non-blocking. By default, it is blocking (i.e. O_NONBLOCK is not set). F_SETFL can then be used to change the blocking mode.

See the fcntl() documentation for details.

3.5. nsa_bind()

nsa_bind() binds a given NEAT socket to a given address. Note: this function is provided as legacy wrapper, and it is RECOMMENDED to use nsa_bindn() instead. Note further, that nsa_bind() also supports a single address only (i.e. no multi-homing). nsa_bindx() SHOULD be used instead to support multi-homing.

Function Prototype:

```
int nsa_bind(int sockfd,
             const struct sockaddr* addr, socklen_t addrlen,
             struct neat_tlv* opt, const int optcnt)
```

Arguments:

sockfd: NEAT socket descriptor.

addr: Address to bind to.

addrlen: Length of the address structure "addr".

opt: NEAT options (NULL, if there are none).

optcnt: Number of NEAT options provided by "opt".

nsa_bind() returns 0 in case of success, or -1 in case of error. The error code will be set in the errno variable.

See the bind() documentation for details.

3.6. nsa_bindx()

nsa_bindx() binds a given NEAT socket to a given set of addresses. Note: this function is provided as legacy wrapper, and it is RECOMMENDED to use nsa_bindn() instead.

Function Prototype:

```
int nsa_bindx(int sockfd, const struct sockaddr* addrs, int addrcnt,
              int flags,
              struct neat_tlv* opt, const int optcnt)
```

Arguments:

sockfd: NEAT socket descriptor.

addrs: Addresses to bind to.

addrcnt: Number of addresses in "addr".

flags: Optional flags (0, if there are none).

opt: NEAT options (NULL, if there are none).

optcnt: Number of NEAT options provided by "opt".

nsa_bindx() returns 0 in case of success, or -1 in case of error. The error code will be set in the errno variable.

See the sctp_bindx() documentation for details.

3.7. nsa_bindn()

nsa_bindn() binds a given NEAT socket to a given port. NEAT takes care of handling local addresses.

Function Prototype:

```
int nsa_bindn(int sockfd, uint16_t port, int flags,
              struct neat_tlv* opt, const int optcnt)
```

Arguments:

sockfd: NEAT socket descriptor.

port: Port number to bind to.

flags: Optional flags (0, if there are none).

opt: NEAT options (NULL, if there are none).

optcnt: Number of NEAT options provided by "opt".

nsa_bindn() returns 0 in case of success, or -1 in case of error. The error code will be set in the errno variable.

3.8. nsa_connect()

nsa_connect() connects a given NEAT socket to a given remote address. Note: this function is provided as legacy wrapper, and it is RECOMMENDED to use nsa_connectn() instead. Note further, that nsa_connect() also supports a single address only (i.e. no multi-homing). nsa_connectx() SHOULD be used instead to support multi-homing.

Function Prototype:

```
int nsa_connect(int sockfd,
                const struct sockaddr* addr, socklen_t addrlen,
                struct neat_tlv* opt, const int optcnt)
```

Arguments:

sockfd: NEAT socket descriptor.

addr: Address to connect to.

addrlen: Length of the address structure "addr".

opt: NEAT options (NULL, if there are none).

optcnt: Number of NEAT options provided by "opt".

nsa_connect() returns 0 in case of success, or -1 in case of error. The error code will be set in the errno variable.

See the connect() documentation for details.

3.9. nsa_connectx()

nsa_connectx() connects a given NEAT socket to a given set of remote addresses. Note: this function is provided as legacy wrapper, and it is RECOMMENDED to use nsa_connectn() instead.

Function Prototype:

```
int nsa_connectx(int sockfd,
                 const struct sockaddr* addrs, int addrcnt,
                 neat_assoc_t* id,
                 struct neat_tlv* opt, const int optcnt)
```

Arguments:

sockfd: NEAT socket descriptor.

addrs: Addresses to connect to.

addrcnt: Number of addresses in "addr".

id Pointer to store association ID to (not used yet, use NULL!).

opt: NEAT options (NULL, if there are none).

optcnt: Number of NEAT options provided by "opt".

nsa_connectx() returns 0 in case of success, or -1 in case of error. The error code will be set in the errno variable.

See the sctp_connectx() documentation for details.

3.10. nsa_connectn()

nsa_connectn() connects a given NEAT socket to a given remote name and port. The remote name is resolved by NEAT to corresponding remote addresses.

Function Prototype:

```
int nsa_connectn(int sockfd, const char* name, const uint16_t port,
                 neat_assoc_t* id,
                 struct neat_tlv* opt, const int optcnt)
```

Arguments:

sockfd: NEAT socket descriptor.

name: Remote name to connect to.

port: Remote port number to connect to.

id Pointer to store association ID to (not used yet, use NULL!).

opt: NEAT options (NULL, if there are none).

optcnt: Number of NEAT options provided by "opt".

nsa_connectn() returns 0 in case of success, or -1 in case of error. The error code will be set in the errno variable.

3.11. nsa_listen()

nsa_listen() marks a given NEAT socket as listening socket, i.e. accepting incoming connections.

Function Prototype:

```
int nsa_listen(int sockfd, int backlog)
```

Arguments:

sockfd: NEAT socket descriptor.

backlog: Defines the maximum length to which the queue of pending connections for "sockfd" may grow.

nsa_listen() returns 0 in case of success, or -1 in case of error. The error code will be set in the errno variable.

See the listen() documentation for details.

3.12. nsa_accept()

nsa_accept() extracts the first connection request in the queue of pending connections for a listening NEAT socket, creates a new connected socket, and returns a new NEAT socket descriptor referring to that socket.

Function Prototype:

```
int nsa_accept(int sockfd, struct sockaddr* addr, socklen_t* addrlen)
```

Arguments:

sockfd: NEAT socket descriptor.

addr: Pointer to storage space to store the peer's primary address to (or NULL, if address is not needed).

addrlen: Pointer to variable with size of the storage in "addr" (or NULL, if address is not needed).

nsa_accept() returns the new NEAT socket descriptor in case of success, or -1 in case of error. The error code will be set in the errno variable. In case of success, the peer's primary address is stored in "addr", if there is sufficient space. The variable pointer to by "addrlen" will then contain the actual address size.

See the accept() documentation for details.

3.13. nsa_accept4()

nsa_accept4() extracts the first connection request in the queue of pending connections for a listening NEAT socket, creates a new connected socket, and returns a new NEAT socket descriptor referring to that socket. If successful, and flags!=0, nsa_accept4() furthermore makes the new socket non-blocking (SOCK_NONBLOCK flag) and/or close-on-exec (SOCK_CLOEXEC flag). For flags==0, the behaviour is identical to nsa_accept().

Function Prototype:

```
int nsa_accept4(int sockfd,
                struct sockaddr* addr, socklen_t* addrlen,
                int flags)
```

Arguments:

sockfd: NEAT socket descriptor.

addr: Pointer to storage space to store the peer’s primary address to (or NULL, if address is not needed).

addrlen: Pointer to variable with size of the storage in "addr" (or NULL, if address is not needed).

nsa_accept4() returns the new NEAT socket descriptor in case of success, or -1 in case of error. The error code will be set in the errno variable. In case of success, the peer’s primary address is stored in "addr", if there is sufficient space. The variable pointer to by "addrlen" will then contain the actual address size.

See the accept() documentation for details.

3.14. nsa_shutdown()

nsa_shutdown() shuts down the connection of a given NEAT socket.

Function Prototype:

```
int nsa_shutdown(int sockfd, int how)
```

Arguments:

sockfd: NEAT socket descriptor.

how: Not used for NEAT sockets (set to SHUT_RDWR).

nsa_shutdown() returns 0 in case of success, or -1 in case of error. The error code will be set in the errno variable.

See the shutdown() documentation for details.

4. Options Handling

4.1. nsa_getsockopt()

nsa_getsockopt() gets a socket option of a given NEAT socket.

Function Prototype:

```
int nsa_getsockopt(int sockfd, int level, int optname,
                   void* optval, socklen_t* optlen)
```

Arguments:

sockfd: NEAT socket descriptor.

level: Option level.

optname: Option number.

optval: Buffer to store option value to.

optlen: Pointer to variable with length of the buffer in "optval".

nsa_getsockopt() returns 0 in case of success, or -1 in case of error. The error code will be set in the errno variable.

See the getsockopt() documentation for details.

4.2. nsa_setsockopt()

nsa_getsockopt() sets a socket option of a given NEAT socket.

Function Prototype:

```
int nsa_setsockopt(int sockfd, int level, int optname,
                   const void* optval, socklen_t optlen)
```

Arguments:

sockfd: NEAT socket descriptor.

level: Option level.

optname: Option number.

optval: Buffer with option value to set.

optlen: Length of buffer with option value.

nsa_setsockopt() returns 0 in case of success, or -1 in case of error. The error code will be set in the errno variable.

See the setsockopt() documentation for details.

4.3. nsa_opt_info()

nsa_opt_info() gets a socket option of a given NEAT socket.

Function Prototype:

```
int nsa_opt_info(int sockfd, neat_assoc_t id,
                 int opt, void* arg, socklen_t* size)
```

Arguments:

sockfd: NEAT socket descriptor.

id: Association identifier (0 in case of 1:1-style sockets).

opt: Option number.

arg: Buffer to store option value to.

size: Pointer to variable with length of the buffer in "arg".

nsa_opt_info() returns 0 in case of success, or -1 in case of error. The error code will be set in the errno variable.

See the sctp_opt_info() documentation for details.

5. Security

5.1. nsa_set_secure_identity()

TBD.

5.2. ...

TBD.

6. Input/Output Handling

6.1. nsa_write()

nsa_write() sends data over a given connected NEAT socket. For NEAT sockets, nsa_write() is equal to nsa_send() with "flags" set to 0.

Function Prototype:

```
ssize_t nsa_write(int fd, const void* buf, size_t len)
```

Arguments:

fd: NEAT socket descriptor.

buf: Data to send.

len: Length of data to send.

nsa_write() returns the number of sent bytes in case of success, or -1 in case of error. The error code will be set in the errno variable.

See the write() documentation for details.

6.2. nsa_writev()

nsa_writev() sends data over a given connected NEAT socket. The data is provided by an iovec structure.

Function Prototype:

ssize_t nsa_writev(int fd, const struct iovec* iov, int iovcnt)

Arguments:

sockfd: NEAT socket descriptor.

iov: Data to send provided as iovec structures.

iovcnt: Number of provided iovec structures.

nsa_writev() returns the number of sent bytes in case of success, or -1 in case of error. The error code will be set in the errno variable.

See the writev() documentation for details.

6.3. nsa_pwrite()

nsa_pwrite() is a wrapper around the pwrite() call, using a NEAT socket descriptor instead.

Function Prototype:

ssize_t nsa_pwrite(int fd, const void* buf, size_t len, off_t offset)

See the pwrite() documentation for details.

6.4. nsa_pwrite64()

nsa_pwrite64() is a wrapper around the pwrite64() call, using a NEAT socket descriptor instead.

Function Prototype:

```
ssize_t nsa_pwrite(int fd, const void* buf, size_t len,
                   off64_t offset)
```

See the pwrite64() documentation for details.

6.5. nsa_pwritev()

nsa_pwritev() is a wrapper around the pwritev() call, using a NEAT socket descriptor instead.

Function Prototype:

```
ssize_t nsa_pwritev(int fd, const struct iovec* iov, int iovcnt,
                    off_t offset)
```

See the pwritev() documentation for details.

6.6. nsa_pwritev64()

nsa_pwritev64() is a wrapper around the pwritev64() call, using a NEAT socket descriptor instead.

Function Prototype:

```
ssize_t nsa_pwritev(int fd, const struct iovec* iov, int iovcnt,
                    off64_t offset)
```

See the pwritev64() documentation for details.

6.7. nsa_send()

nsa_send() sends data over a given connected NEAT socket.

Function Prototype:

```
ssize_t nsa_send(int sockfd, const void* buf, size_t len, int flags)
```

Arguments:

sockfd: NEAT socket descriptor.

buf: Data to send.

len: Length of data to send.

flags: Optional flags (0, if there are none).

nsa_send() returns the number of sent bytes in case of success, or -1 in case of error. The error code will be set in the errno variable.

See the send() documentation for details.

6.8. nsa_sendto()

nsa_sendto() is a wrapper around the sendto() call, using NEAT socket descriptors instead. Note: this function is provided as legacy wrapper, and it is RECOMMENDED to use nsa_send() instead. On NEAT sockets, a provided destination address is ignored.

Function Prototype:

```
ssize_t nsa_sendto(int sockfd, const void* buf, size_t len,
                   int flags,
                   const struct sockaddr* to, socklen_t tolen)
```

Arguments:

sockfd: NEAT socket descriptor.

buf: Data to send.

len: Length of data to send.

flags: Optional flags (0, if there are none).

to: Address to send data to (ignored for NEAT sockets).

tolen: Length of address to send data to (ignored for NEAT sockets).

nsa_sendto() returns the number of sent bytes in case of success, or -1 in case of error. The error code will be set in the errno variable.

See the send() documentation for details.

6.9. nsa_sendmsg()

nsa_sendmsg() sends data over a given connected NEAT socket. The data and control information is provided by a msghdr structure. On NEAT sockets, a provided destination address is ignored.

Function Prototype:

```
ssize_t nsa_sendmsg(int sockfd, const struct msghdr* msg, int flags)
```

Arguments:

sockfd: NEAT socket descriptor.

msg: Data to send and corresponding control information as msghdr structure.

flags: Optional flags (0, if there are none).

nsa_sendmsg() returns the number of sent bytes in case of success, or -1 in case of error. The error code will be set in the errno variable.

See the sendmsg() documentation for details.

6.10. nsa_sendv()

nsa_sendv() sends data over a given connected NEAT socket. The data and control information is provided by iovec and info structures. On NEAT sockets, a provided destination address is ignored.

Function Prototype:

```
ssize_t nsa_sendv(int sockfd, struct iovec* iov, int iovcnt,
                  struct sockaddr* to, int tocnt,
                  void* info, socklen_t infolen,
                  unsigned int infotype, int flags)
```

Arguments:

sockfd: NEAT socket descriptor.

iov: Data to send provided as iovec structures.

iovcnt: Number of provided iovec structures.

to: Address(es) to send data to (ignored for NEAT sockets).

tocnt: Number of of addresses to send data to (ignored for NEAT sockets).

info: Control information.

infolen: Length of control information.

infotype: Type of control information.

flags: Optional flags (0, if there are none).

nsa_sendv() returns the number of sent bytes in case of success, or -1 in case of error. The error code will be set in the errno variable.

See the sctp_sendv() documentation for details.

6.11. nsa_read()

nsa_read() reads data from a given connected NEAT socket. For NEAT sockets, nsa_read() is equal to nsa_recv() with "flags" set to 0.

Function Prototype:

ssize_t nsa_read(int fd, void* buf, size_t len)

Arguments:

fd: NEAT socket descriptor.

buf: Buffer to store read data to.

len: Length of the storage buffer.

nsa_read() returns the number of read bytes in case of success, 0 in case of connection shutdown, or -1 in case of error. The error code will be set in the errno variable.

See the read() documentation for details.

6.12. nsa_readv()

nsa_readv() reads data from a given connected NEAT socket. The data information buffers are provided by an iovec structure.

Function Prototype:

ssize_t nsa_readv(int fd, const struct iovec* iov, int iovcnt)

Arguments:

fd: NEAT socket descriptor.

iov: Data to send provided as iovec structures.

iovcnt: Number of provided iovec structures.

nsa_readv() returns the number of read bytes in case of success, 0 in case of connection shutdown, or -1 in case of error. The error code will be set in the errno variable.

See the readv() documentation for details.

6.13. nsa_pread()

nsa_pread() is a wrapper around the pread() call, using a NEAT socket descriptor instead.

Function Prototype:

```
ssize_t nsa_pread(int fd, void* buf, size_t len, off_t offset)
```

See the pread() documentation for details.

6.14. nsa_pread64()

nsa_pread64() is a wrapper around the pread64() call, using a NEAT socket descriptor instead.

Function Prototype:

```
ssize_t nsa_pread(int fd, void* buf, size_t len, off_t offset)
```

See the pread64() documentation for details.

6.15. nsa_preadv()

nsa_preadv() is a wrapper around the preadv() call, using a NEAT socket descriptor instead.

Function Prototype:

```
ssize_t nsa_preadv(int fd, const struct iovec* iov, int iovcnt,
                   off64_t offset)
```

See the preadv() documentation for details.

6.16. nsa_preadv64()

nsa_preadv64() is a wrapper around the preadv64() call, using a NEAT socket descriptor instead.

Function Prototype:

```
ssize_t nsa_preadv(int fd, const struct iovec* iov, int iovcnt,
                   off64_t offset)
```

See the preadv64() documentation for details.

6.17. nsa_recv()

nsa_recv() reads data from a given connected NEAT socket.

Function Prototype:

```
ssize_t nsa_recv(int sockfd, void* buf, size_t len, int flags)
```

Arguments:

sockfd: NEAT socket descriptor.

buf: Buffer to store read data to.

len: Length of the storage buffer.

flags: Optional flags (0, if there are none).

nsa_recv() returns the number of read bytes in case of success, 0 in case of connection shutdown, or -1 in case of error. The error code will be set in the errno variable.

See the recv() documentation for details.

6.18. nsa_recvfrom()

nsa_recvfrom() reads data from a given connected NEAT socket. The peer's sending address of the data (if possible and useful for underlying transport protocol) is obtained as well. Note: this function is provided as legacy wrapper, and it is RECOMMENDED to use nsa_recv() instead.

Function Prototype:

```
ssize_t nsa_recvfrom(int sockfd, void* buf, size_t len, int flags,
                     struct sockaddr* from, socklen_t* fromlen)
```

sockfd: NEAT socket descriptor.

buf: Buffer to store read data to.

len: Length of the storage buffer.

flags: Optional flags (0, if there are none).

from: Pointer to storage space to store the peer's primary address to (or NULL, if address is not needed).

fromlen: Pointer to variable with size of the storage in "from" (or NULL, if address is not needed).

nsa_recvfrom() returns the number of read bytes in case of success, 0 in case of connection shutdown, or -1 in case of error. The error code will be set in the errno variable. In case of success, the peer's sending address (if possible and useful for underlying transport protocol) may be stored in "from", if there is sufficient space. The variable pointer to by "fromlen" will then contain the actual address size.

See the recvfrom() documentation for details.

6.19. nsa_recvmsg()

nsa_recvmsg() reads data from a given connected NEAT socket. The data and control information buffers are provided by a msghdr structure.

Function Prototype:

ssize_t nsa_recvmsg(int sockfd, struct msghdr* msg, int flags)

Arguments:

sockfd: NEAT socket descriptor.

msg: Data to send and corresponding control information as msghdr structure.

flags: Optional flags (0, if there are none).

nsa_recvmsg() returns the number of read bytes in case of success, 0 in case of connection shutdown, or -1 in case of error. The error code will be set in the errno variable.

See the recvmsg() documentation for details.

6.20. nsa_recvv()

nsa_recvv() reads data from a given connected NEAT socket. The data and control information buffers are provided by iovec and info structures.

Function Prototype:

```
ssize_t nsa_recvv(int sockfd, struct iovec* iov, int iovcnt,
                  struct sockaddr* from, socklen_t* fromlen,
                  void* info, socklen_t* infolen,
                  unsigned int* infotype, int* msg_flags)
```

Arguments:

sockfd: NEAT socket descriptor.

iov: Data to send provided as iovec structures.

iovcnt: Number of provided iovec structures.

from: Pointer to storage space to store the peer's primary address to (or NULL, if address is not needed).

fromlen: Pointer to variable with size of the storage in "from" (or NULL, if address is not needed).

info: Pointer to storage space for control information.

infolen: Pointer to variable with length of control information.

infotype: Pointer to variable for storing the control information type to.

flags: Pointer to variable with optional flags.

nsa_recvv() returns the number of sent received in case of success, or -1 in case of error. The error code will be set in the errno variable.

See the sctp_recvv() documentation for details.

7. Poll and Select

7.1. nsa_poll()

nsa_poll() waits for activity (input/output/error/...) on a set of given NEAT sockets.

Function Prototype:

```
int nsa_poll(struct pollfd* ufds, const nfds_t nfds, int timeout)
```

Arguments:

ufds: NEAT socket descriptor and requested activity for each NEAT socket.

nfds: Number of sockets given by "ufds".

timeout: Timeout in milliseconds.

nsa_poll() returns the number of NEAT sockets with activity in case of success, 0 in case of timeout, or -1 in case of error. The error code will be set in the errno variable.

See the poll() documentation for details.

7.2. nsa_select()

nsa_select() is a wrapper around the select() call, using NEAT socket descriptors instead. Note: this function is provided as legacy wrapper, and it is RECOMMENDED to use nsa_poll() instead.

Function Prototype:

```
int nsa_select(int n,
               fd_set* readfds, fd_set* writefds, fd_set* exceptfds,
               struct timeval* timeout)
```

See the select() documentation for details.

8. Address Handling

8.1. nsa_getsockname()

nsa_getsockname() obtains the first local address of a socket. Note: this function is provided as legacy wrapper, and it is RECOMMENDED to use nsa_getladdrs() instead to support multi-homed transport protocols!

Function Prototype:

```
int nsa_getsockname(int sockfd,
                    struct sockaddr* name, socklen_t* namelen)
```

Arguments:

sockfd: NEAT socket descriptor.

name: Storage space for the address.

namelen: Pointer to variable with the storage space’s size.

Return Value:

nsa_getsockname() returns 0 in case of success (with the actual address size stored into the "namelen" variable), or -1 in case of error. The error code will be set in the errno variable.

See the getsockname() documentation for details.

8.2. nsa_getpeername()

nsa_getpeername() obtains the first remote address of a connected socket. Note: this function is provided as legacy wrapper, and it is RECOMMENDED to use nsa_getpaddrs() instead to support multi-homed transport protocols!

Function Prototype:

```
int nsa_getpeername(int sockfd,
                    struct sockaddr* name, socklen_t* namelen)
```

Arguments:

sockfd: NEAT socket descriptor.

name: Storage space for the address.

namelen: Pointer to variable with the storage space’s size.

Return Value:

nsa_getpeername() returns 0 in case of success (with the actual address size stored into the "namelen" variable), or -1 in case of error. The error code will be set in the errno variable.

See the getpeername() documentation for details.

8.3. nsa_getladdrs()

nsa_getladdrs() obtains the local addresses of a socket. The storage space for the addresses will be automatically allocated and needs to be freed by nsa_freeladdrs().

Function Prototype:

```
int nsa_getladdrs(int sockfd, neat_assoc_t id,
                  struct sockaddr** addrs)
```

Arguments:

sockfd: NEAT socket descriptor.

id: Association identifier (0 in case of 1:1-style sockets).

addrs: Pointer to variable to store pointer to addresses to.

nsa_getladdrs() returns the number of addresses stored into a newly allocated space. The pointer to this space is stored into the variable provided by "addrs". In case of error, -1 is returned, and the error code will be set in the errno variable.

8.4. nsa_freeladdrs()

nsa_freeladdrs() frees addresses obtained by nsa_getladdrs().

Function Prototype:

```
void nsa_freeladdrs(struct sockaddr* addrs)
```

Arguments:

addrs: Pointer to addresses to be freed.

8.5. nsa_getpaddrs()

nsa_getpaddrs() obtains the remote addresses of a connected socket. The storage space for the addresses will be automatically allocated and needs to be freed by nsa_freepaddrs().

Function Prototype:

```
int nsa_getpaddrs(int sockfd, neat_assoc_t id,
                  struct sockaddr** addrs)
```

Arguments:

sockfd: NEAT socket descriptor.

id: Association identifier (0 in case of 1:1-style sockets).

addrs: Pointer to variable to store pointer to addresses to.

nsa_getpaddrs() returns the number of addresses stored into a newly allocated space. The pointer to this space is stored into the variable provided by "addrs". In case of error, -1 is returned, and the error code will be set in the errno variable.

8.6. nsa_freepaddrs()

nsa_freepaddrs() frees addresses obtained by nsa_getpaddrs().

Function Prototype:

void nsa_freepaddrs(struct sockaddr* addrs)

Arguments:

addrs: Pointer to addresses to be freed.

9. Miscellaneous

This section contains miscellaneous wrapper functions, mostly around file I/O. Since Unix file descriptors are used together with socket descriptors in functions like poll(), select(), etc., it is necessary to wrap functions handling file descriptors as well.

9.1. nsa_open()

nsa_open() is a wrapper around the open() call, returning a NEAT socket descriptor instead.

Function Prototype:

int nsa_open(const char* pathname, int flags, mode_t mode)

See the open() documentation for details.

9.2. nsa_creat()

nsa_creat() is a wrapper around the creat() call, returning a NEAT socket descriptor instead.

Function Prototype:

int nsa_creat(const char* pathname, mode_t mode)

See the creat() documentation for details.

9.3. nsa_lockf()

nsa_lockf() is a wrapper around the lockf() call, using a NEAT socket descriptor instead.

Function Prototype:

int nsa_lockf(int fd, int cmd, off_t len)

See the lockf() documentation for details.

9.4. nsa_lockf64()

nsa_lockf64() is a wrapper around the lockf64() call, using a NEAT socket descriptor instead.

Function Prototype:

int nsa_lockf(int fd, int cmd, off64_t len)

See the lockf64() documentation for details.

9.5. nsa_flock()

nsa_flock() is a wrapper around the flock() call, using a NEAT socket descriptor instead.

Function Prototype:

int nsa_flock(int fd, int operation)

See the flock() documentation for details.

9.6. nsa_fstat()

nsa_fstat() is a wrapper around the fstat() call, using a NEAT socket descriptor instead.

Function Prototype:

int nsa_fstat(int fd, struct stat* buf)

See the fstat() documentation for details.

9.7. nsa_fpathconf()

nsa_fpathconf() is a wrapper around the fpathconf() call, using a NEAT socket descriptor instead.

Function Prototype:

long nsa_fpathconf(int fd, int name)

See the fpathconf() documentation for details.

9.8. nsa_fchown()

nsa_fchown() is a wrapper around the fchown() call, using a NEAT socket descriptor instead.

Function Prototype:

```
int nsa_fchown(int fd, uid_t owner, gid_t group)
```

See the fchown() documentation for details.

9.9. nsa_fsync()

nsa_fsync() is a wrapper around the fsync() call, using a NEAT socket descriptor instead.

Function Prototype:

```
int nsa_fsync(int fd)
```

See the fsync() documentation for details.

9.10. nsa_fdatasync()

nsa_fdatasync() is a wrapper around the fdatasync() call, using a NEAT socket descriptor instead.

Function Prototype:

```
int nsa_fdatasync(int fd)
```

See the fdatasync() documentation for details.

9.11. nsa_syncfs()

nsa_syncfs() is a wrapper around the syncfs() call, using a NEAT socket descriptor instead.

Function Prototype:

```
int nsa_syncfs(int fd)
```

See the syncfs() documentation for details.

9.12. nsa_dup2()

nsa_dup2() is a wrapper around the dup2() call, using a NEAT socket descriptor instead.

Function Prototype:

int nsa_dup2(int oldfd, int newfd)

See the dup2() documentation for details.

9.13. nsa_dup3()

nsa_dup3() is a wrapper around the dup3() call, using a NEAT socket descriptor instead.

Function Prototype:

int nsa_dup3(int oldfd, int newfd, int flags)

See the dup3() documentation for details.

9.14. nsa_dup()

nsa_dup() is a wrapper around the dup() call, using a NEAT socket descriptor instead.

Function Prototype:

int nsa_dup(int oldfd)

See the dup() documentation for details.

9.15. nsa_lseek()

nsa_lseek() is a wrapper around the lseek() call, using a NEAT socket descriptor instead.

Function Prototype:

off_t nsa_lseek(int fd, off_t offset, int whence)

See the lseek() documentation for details.

9.16. nsa_lseek64()

nsa_lseek64() is a wrapper around the lseek64() call, using a NEAT socket descriptor instead.

Function Prototype:

off_t nsa_lseek(int fd, off64_t offset, int whence)

See the lseek64() documentation for details.

9.17. nsa_truncate()

nsa_truncate() is a wrapper around the truncate() call, using a NEAT socket descriptor instead.

Function Prototype:

```
int nsa_ftruncate(int fd, off_t length)
```

See the truncate() documentation for details.

9.18. nsa_truncate64()

nsa_truncate64() is a wrapper around the truncate64() call, using a NEAT socket descriptor instead.

Function Prototype:

```
int nsa_ftruncate(int fd, off64_t length)
```

See the truncate64() documentation for details.

9.19. nsa_pipe()

nsa_pipe() is a wrapper around the pipe() call, returning NEAT socket descriptors instead.

Function Prototype:

```
int nsa_pipe(int fds[2])
```

See the pipe() documentation for details.

9.20. nsa_ioctl()

nsa_ioctl() is a wrapper around the ioctl() call, using a NEAT socket descriptor instead.

Function Prototype:

```
int nsa_ioctl(int fd, int request, const void* argp)
```

See the ioctl() documentation for details.

10. Code Examples

Running code examples can be found in the NEAT Git repository, with some tutorial material in [10], [11]:

URL: https://github.com/NEAT-project/neat (https://github.com/NEAT-project/neat)

Branch: dreibh/neat-socketapi (https://github.com/NEAT-project/neat/tree/dreibh/neat-socketapi)

Directory: socketapi/examples/ (https://github.com/NEAT-project/neat/tree/dreibh/neat-socketapi/socketapi/examples)

11. Testbed Platform

A large-scale and realistic Internet testbed platform with support for the multi-homing feature of the underlying SCTP and MPTCP protocols is NorNet. A description of NorNet is provided in [6], [7], some further information can be found on the project website [9].

12. Security Considerations

Security considerations for the SCTP sockets API are described in [2].

13. IANA Considerations

This document does not require IANA actions.

14. Acknowledgments

This work was partially funded by the European Union’s Horizon 2020 research and innovation programme under grant agreement No. 644334 (NEAT). The views expressed are solely those of the author(s).

The author would like to thank David Ros, Michael Welzl, and Xing Zhou for their support.

15. References

15.1. Normative References

[1] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[2] Stewart, R., Tuexen, M., Poon, K., Lei, P., and V. Yasevich, "Sockets API Extensions for the Stream Control Transmission Protocol (SCTP)", RFC 6458, DOI 10.17487/RFC6458, December 2011, <https://www.rfc-editor.org/info/rfc6458>.

[3] Gjessing, S. and M. Welzl, "A Minimal Set of Transport Services for TAPS Systems", Work in Progress, Internet-Draft, draft-gjessing-taps-minset-05, 20 June 2017, <https://www.ietf.org/archive/id/draft-gjessing-taps-minset-05.txt>.

[4] Fairhurst, G., "The NEAT Interface to Transport Services", Work in Progress, Internet-Draft, draft-fairhurst-taps-neat-00, 30 October 2017, <http://www.ietf.org/internet-drafts/draft-fairhurst-taps-neat-00.txt>.

[5] Welzl, M., Tuexen, M., and N. Khademi, "On the Usage of Transport Features Provided by IETF Transport Protocols", Work in Progress, Internet-Draft, draft-ietf-taps-transports-usage-09, 26 October 2017, <https://www.ietf.org/archive/id/draft-ietf-taps-transports-usage-09.txt>.

15.2. Informative References

[6] Dreibholz, T., "NorNet Building an Inter-Continental Internet Testbed based on Open Source Software", Proceedings of the LinuxCon Europe, 5 October 2016, <https://simula.no/file/linuxcon2016-presentationpdf/download>.

[7] Gran, E. G., Dreibholz, T., and A. Kvalbein, "NorNet Core A Multi-Homed Research Testbed", Computer Networks, Special Issue on Future Internet Testbeds Volume 61, Pages 75-87, ISSN 1389-1286, DOI 10.1016/j.bjp.2013.12.035, 14 March 2014, <https://www.simula.no/file/simulasimula2236pdf/download>.

[8] Dreibholz, T., "NEAT A New, Evolutive API and Transport-Layer Architecture for the Internet", 2022, <https://www.neat-project.org/>.

[9] Dreibholz, T., "NorNet A Real-World, Large-Scale Multi-Homing Testbed", 2022, <https://www.nntb.no/>.

[10] Dreibholz, T., "NEAT Tutorial at Hainan University: Getting Started with NEAT", Invited Talk at Hainan University, College of Information Science and Technology (CIST), 18 December 2017, <https://www.simula.no/file/haikou2017-neat-tutorialpdf/download>.

[11] Dreibholz, T., "A Practical Introduction to NEAT at Hainan University", Invited Talk at Hainan University, College of Information Science and Technology (CIST), 17 April 2017, <https://www.simula.no/file/haikou2017-neat-introductionpdf/download>.

[12] Weinrank, F., Grinnemo, K., Bozakov, Z., Brunstrm, A., Dreibholz, T., Hurtig, P., Khademi, N., and M. Txen, "A NEAT Way to Browse the Web", Proceedings of the ACM, IRTF and ISOC Applied Networking Research Workshop (ANRW) Pages 33-34, ISBN 978-1-4503-5108-9, DOI 10.1145/3106328.3106335, 15 July 2017, <https://www.simula.no/file/anrw17-final13pdf/download>.

[13] Fairhurst, G., Jones, T., Bozakov, Z., Brunstrm, A., Damjanovi, D., Eckert, K. R. E. T., Grinnemo, K., Hansen, A. F., Khademi, N., Mangiante, S., McManus, P., Papastergiou, G., Ros, D., Txen, M., Vyncke, E., and M. Welzl, "NEAT Architecture", Number D1.1, 1 December 2015, <https://www.neat-project.org/wp-content/uploads/2016/02/D1.1.pdf>.

[14] Welzl, M., Damjanovi, D., Fairhurst, G., Hayes, D., Jones, T., Ros, D., Txen, M., and F. Weinrank, "Final Version of Services and APIs", Deliverable D1.3, 30 October 2017, <https://www.neat-project.org/wp-content/uploads/2015/05/D1.3.pdf>.

[15] Khademi, N., Bozakov, Z., Brunstrm, A., Dale, ., Damjanovi, D., Evensen, K. R., Fairhurst, G., Grinnemo, K., Jones, T., Mangiante, S., Petlund, A., Ros, D., Stenberg, D., Txen, M., Weinrank, F., and M. Welzl, "NEAT Core Transport System, with both Low-level and High-level Components", Number D2.2, 14 March 2017, <https://www.neat-project.org/wp-content/uploads/2017/03/D2.2-public.pdf>.

[16] Khademi, N., Bozakov, Z., Brunstrm, A., Dale, ., Damjanovi, D., Evensen, K. R., Fairhurst, G., Fischer, A., Grinnemo, K., Jones, T., Mangiante, S., Petlund, A., Ros, D., Rngeler, I., Stenberg, D., Txen, M., Weinrank,

F., and M. Welzl, "Final Version of Core Transport System", Deliverable D2.3, 31 August 2017, <https://www.neat-project.org/wp-content/uploads/2017/10/D2.3.pdf>.

Author’s Address

Thomas Dreibholz
Simula Metropolitan Centre for Digital Engineering
Pilestredet 52
0167 Oslo
Norway
Email: dreibh@simula.no
URI: https://www.simula.no/people/dreibh