

Network Working Group
Internet-Draft
Intended status: Experimental
Expires: 13 August 2022

T. Dreibholz
SimulaMet
9 February 2022

NEAT Sockets API
draft-dreibholz-taps-neat-socketapi-10

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 13 August 2022.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	4
1.1. Conventions	4
2. Initialisation and Clean-Up	4
2.1. nsa_init()	4
2.2. nsa_cleanup()	4
2.3. nsa_map_socket()	5
2.4. nsa_unmap_socket()	5
3. Connection Establishment and Teardown	5
3.1. nsa_socket()	5
3.2. nsa_socketpair()	6
3.3. nsa_close()	6
3.4. nsa_fcntl()	7
3.5. nsa_bind()	7
3.6. nsa_bindx()	8
3.7. nsa_bindn()	9
3.8. nsa_connect()	9
3.9. nsa_connectx()	10
3.10. nsa_connectn()	11
3.11. nsa_listen()	11
3.12. nsa_accept()	12
3.13. nsa_accept4()	12
3.14. nsa_shutdown()	13
4. Options Handling	13
4.1. nsa_getsockopt()	13
4.2. nsa_setsockopt()	14
4.3. nsa_opt_info()	14
5. Security	15
5.1. nsa_set_secure_identity()	15
5.2.	15
6. Input/Output Handling	15
6.1. nsa_write()	15
6.2. nsa_writev()	16
6.3. nsa_pwrite()	16
6.4. nsa_pwrite64()	16
6.5. nsa_pwritev()	17
6.6. nsa_pwritev64()	17
6.7. nsa_send()	17
6.8. nsa_sendto()	18
6.9. nsa_sendmsg()	18
6.10. nsa_sendv()	19
6.11. nsa_read()	20
6.12. nsa_readv()	20
6.13. nsa_pread()	21
6.14. nsa_pread64()	21
6.15. nsa_preadv()	21
6.16. nsa_preadv64()	21

6.17. nsa_recv()	22
6.18. nsa_recvfrom()	22
6.19. nsa_recvmsg()	23
6.20. nsa_recvv()	23
7. Poll and Select	24
7.1. nsa_poll()	24
7.2. nsa_select()	25
8. Address Handling	25
8.1. nsa_getsockname()	25
8.2. nsa_getpeername()	26
8.3. nsa_getladdrs()	26
8.4. nsa_freeladdrs()	27
8.5. nsa_getpaddrs()	27
8.6. nsa_freepaddrs()	28
9. Miscellaneous	28
9.1. nsa_open()	28
9.2. nsa_creat()	28
9.3. nsa_lockf()	28
9.4. nsa_lockf64()	29
9.5. nsa_flock()	29
9.6. nsa_fstat()	29
9.7. nsa_fpathconf()	29
9.8. nsa_fchown()	30
9.9. nsa_fsync()	30
9.10. nsa_fdatasync()	30
9.11. nsa_syncfs()	30
9.12. nsa_dup2()	30
9.13. nsa_dup3()	31
9.14. nsa_dup()	31
9.15. nsa_lseek()	31
9.16. nsa_lseek64()	31
9.17. nsa_truncate()	32
9.18. nsa_truncate64()	32
9.19. nsa_pipe()	32
9.20. nsa_ioctl()	32
10. Code Examples	33
11. Testbed Platform	33
12. Security Considerations	33
13. IANA Considerations	33
14. Acknowledgments	33
15. References	33
15.1. Normative References	33
15.2. Informative References	34
Author's Address	36

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_setsockopt() 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_recv()`

`nsa_recv()` 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_ftruncate64(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) (<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) (<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-transport-usage-09, 26 October 2017, <<https://www.ietf.org/archive/id/draft-ietf-taps-transport-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", Online: <https://www.neat-project.org/>, 2022, <<https://www.neat-project.org/>>.
- [9] Dreibholz, T., "NorNet - A Real-World, Large-Scale Multi-Homing Testbed", Online: <https://www.nntb.no/>, 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., Brunström, A., Dreibholz, T., Hurtig, P., Khademi, N., and M. Tüxen, "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., Brunström, A., Damjanovi, D., Eckert, K. R. E. T., Grinnemo, K., Hansen, A. F., Khademi, N., Mangiante, S., McManus, P., Papastergiou, G., Ros, D., Tüxen, 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., Tüxen, 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., Brunström, A., Dale, Ø., Damjanovi, D., Evensen, K. R., Fairhurst, G., Grinnemo, K., Jones, T., Mangiante, S., Petlund, A., Ros, D., Stenberg, D., Tüxen, 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., Brunström, A., Dale, Ø., Damjanovi, D., Evensen, K. R., Fairhurst, G., Fischer, A., Grinnemo, K., Jones, T., Mangiante, S., Petlund, A., Ros, D., Rüngeler, I., Stenberg, D., Tüxen, 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

Phone: +47-6782-8200
Email: dreibh@simula.no
URI: <https://www.simula.no/people/dreibh>

TSVWG
Internet-Draft
Intended status: Informational
Expires: May 01, 2018

G. Fairhurst
T. Jones
University of Aberdeen
A. Brunstrom
Karlstad University
D. Ros
Simula Research Laboratory
October 30, 2017

The NEAT Interface to Transport Services
draft-fairhurst-taps-neat-00

Abstract

The NEAT System provides an example of a system designed to implement the TAPS Transport Services. This document presents the transport services that the NEAT User API provides to an application or upper-layer protocol. It also describes primitives needed to interface to the NEAT Policy Manager and how policies can be adjusted to match the API behaviour to the properties required by an application or upper-layer protocol using the NEAT User API.

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 <http://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 May 01, 2018.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights

and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
2. The NEAT Context	3
3. NEAT User API Primitives and Events	4
3.1. NEAT Flow Initialisation	4
3.2. NEAT Flow Establishment	5
3.3. NEAT Flow Availability	7
3.4. Writing and reading data	7
3.5. Flow Maintenance Primitives	9
3.6. NEAT Flow Termination	11
3.7. NEAT Error Events	12
4. Security Considerations	12
5. IANA Considerations	12
6. References	12
Appendix A. Revision Information	13
Authors' Addresses	13

1. Introduction

The NEAT (New, Evolutive API and Transport-Layer Architecture for the Internet) [NEAT] System provides a call-back driven API to the network transport layer. It presents a set of transport services [RFC8095] that the NEAT User API provides to an application or upper-layer protocol.

The NEAT System has been implemented in the NEAT User Module. The focus of the present document is on the NEAT User API providing transport services to applications. This utilises a lower interface provided by a Kernel Programming Interface (KPI), to access the traditional Socket API or a transport service implemented in userspace.

Applications that use the NEAT User API can provide information about the features desired from the transport layer and determine the properties of the offered transport service. It is this additional information that enables the NEAT System to move beyond the constraints of the traditional Socket API, since the stack then becomes aware of what the application/user actually desires or requires for each traffic flow. The additional information can be used to automatically identify which transport components (protocol and other transport mechanisms) could be used to realise the required transport service. This can drive the selection by the NEAT System of the best transport components to use and determine how these need to be configured [I-D.grinnemo-taps-he]. In making decisions, the NEAT System can utilise policy information provided at configuration,

previously discovered path characteristics and probing techniques. This can be provided by a policy manager acting below the NEAT User API.

The architecture of the NEAT System is presented in [D1.1]. Some important features of NEAT compared to the existing sockets API are:

- o Event-driven call-back driven interface, enabling applications to be designed so that they respond to events signalling the reception of data blocks, ability to send data blocks, or the successful transmission of data blocks. This concrete API is described in [D2.3].
- o High-level transport interface independent of the selected transport protocol, allowing applications to be written without depending on the features of specific transport protocols, and hence allowing the most suitable transport protocol to be matched to the application, based on the transport features an application requires [RFC8095].
- o Support for unordered/unreliable and reliable transport services.
- o Explicit support for multistreaming and multipath transport protocols and network architectures.
- o A flexible policy framework, allowing applications to describes the properties they expect or require of the transport system and thus enabling the transport services to be configured to match the capabilities of the network that is being used.
- o Ability to work with other network-layer protocols (e.g., network signalling) to realise the required transport service.

The NEAT Library is an open source implementation and is available for download [NEAT-GIT]. This also provides tutorials and examples of code utilising the API and descriptions of the way in the which callback mechanisms can be used to build applications that use this interface. Further documentation for the current NEAT System is available at the NEAT Project web page, [NEAT-DOC].

2. The NEAT Context

Applications interact with the network by sending, receiving and controlling NEAT Flows.

The first step in establishing a flow with the NEAT System is to call a primitive to create and configure a Context. In the remainder of this document, the label P: is used to identify a primitive that may be called for a NEAT Context, and the label E: to identify an event provided by the NEAT System. Each primitive/event is associated with a particular NEAT Context. Most primitives specify the Context and provide a handle to the NEAT Flow upon which they operate, and the primitives and events for manipulating data can only be used after a NEAT Flow has been created.

P: INIT_CTX()

The INIT_CTX primitive sets up the datastructures needed by the NEAT System.

After all network operations are completed it can free the context. It returns a pointer to the newly allocated and initialized NEAT context.

P: FREE_CTX()

The FREE_CTX primitive is called when an created context is no longer needed. It frees the memory associated with the datastructures used by the NEAT System.

3. NEAT User API Primitives and Events

An application using the NEAT System needs to take the following steps to use the network after establishing a context:

1. Initialisation: create a flow by calling P: INIT_FLOW; and then calling P: SET_PROPERTIES to express the application requirements. This is used by the NEAT policy manager. Finally, it needs to bind call-back functions to respond to the events generated by the NEAT System.
2. Establishment / Availability: Connect the NEAT Flow (either actively to a destination endpoint or passively to receive from the network).
3. Writing and reading data: Call primitives to write data or respond to events requesting it to read data.
4. Maintenance: Call maintenance primitives, as needed, to configure attributes of the flow (e.g., while writing reading data).
5. Termination: Close (or abort) the NEAT Flow.

3.1. NEAT Flow Initialisation

An application needs to create and initialise a flow object before it can be used.

P: INIT_FLOW()

The INIT_FLOW primitive creates the essential data structures required to use a NEAT Flow. The application also needs to then call a primitive associate functions with each of the events that it wishes to process.

P: SET_PROPERTIES(property_list)

property_list : A set of flow properties expressed in JSON.

Each NEAT Flow has a set of properties that are set at the flow initialisation time. The SET_PROPERTIES primitive sets properties for the NEAT Flow. Properties are related to Transport Features and Services. For instance: link-layer security, transport-layer security, certificate verification, certificate and key properties can be set at initialisation time are related to a Confidentiality Transport Feature. A flow can also have attributes that can be read by an application using maintenance primitives after a flow has been initialised.

3.2. NEAT Flow Establishment

P: OPEN(destname port [stream_count])

destname : a NEAT-conformant name (which can be a DNS name or a set of IP addresses) to which to connect.

port : port number (integer) or service name (string) to which to connect.

stream_count : the number of requested streams to open (integer). Note that, if this parameter is not used, the system may still use multi-streaming underneath, e.g., by automatically mapping NEAT Flows between the same hosts onto streams of an SCTP association. Using this parameter disables such automatic functionality.

Returns: success or failure. If success, it also returns a handle for a NEAT Flow.

The OPEN primitive opens a flow actively for transports that require a connection handshake (e.g., TCP, SCTP), and opens the flow passively for transports that do not (e.g., UDP, UDP-Lite). Calling P:OPEN alone may not actually have an effect "on the wire", i.e., a P: ACCEPT at the peer may not be triggered by it. Since it is possible that the remote endpoint only returns when data arrives, this may only happen after the local host has called P: WRITE. (This does not result in a problem, since P: ACCEPT does not block).

E: on_connected

The on_connected event indicates a successful connection setup. An

application that receives this event can then use other primitives with this flow.

P: OPEN_WITH_EARLY_DATA(destname port [stream_count] [flow_group]
[stream] [pr_method pr_value] [unordered_flag] data datalen)

destname : defined in the same way as in P: OPEN.

port : defined in the same way as in P: OPEN.

stream_count : defined in the same way as in P: OPEN.

flow_group : defined in the same way as in P: OPEN.

stream : the number of the stream to be used. At the moment this function is called, a connection is still not initialised and the protocol may not be known. If the protocol chosen by the NEAT Selection components supports only one stream, this parameter will be ignored.

pr_method and pr_value : if these parameters are used, then partial reliability is enabled and pr_method must have an integer value from 1 to 2 to specify which method to implement partial reliability is requested. Value 1 means: pr_value specifies a time in milliseconds after which it is unnecessary to send this data block. Value 2 means: pr_value specifies a re-quested maximum number of attempts to retransmit the data block. If the selected NEAT transport does not support partial reliability these parameters will be ignored. (See P: WRITE for more information).

unordered_flag : The data block may be delivered out-of-order if this boolean flag is set. Default: false. If the protocol chosen by the NEAT Selection components does not support unordered delivery, this parameter will be ignored.

data : the data-block to be sent.

datalen : the amount (positive integer) of data supplied in the data-block.

Returns: success or failure. If success, it also returns a handle for a NEAT Flow and the amount of supplied data that was buffered.

The `OPEN_WITH_EARLY_DATA` primitive allows data to be sent at the time when a flow is opened. To accommodate TLS 1.3 [I-D.ietf-tls-tls13] early data and the TCP Fast Open option [RFC7413], application data need to be supplied at the time of opening a NEAT Flow. This primitive opens a flow and sends early data if the protocol supports it. If the protocol chosen does not support early application data. The data will be buffered then sent after connection establishment, similar to calling `P: WRITE`. For this reason, in addition to the parameters of `P: OPEN`, this primitive also needs the same parameters as `P: WRITE`. The supplied data can be delivered multiple times (replayed by the network); an application must take this into account when using this function. This is commonly known as idempotence.

3.3. NEAT Flow Availability

This section describes how an application prepares a flow to accept communication from another NEAT endpoint.

`P: ACCEPT([name] port [stream_count])`

`name` : local NEAT-conformant name (which can be a DNS name or a set of IP addresses) to constrain acceptance of incoming requests to local address(es). If this is missing, requests may arrive at any local address.

`port` : local port number (integer) or service name (string), to constrain acceptance to incoming requests at this port.

`stream_count` : the number of requested streams to open (integer). Default value: 1.

Returns: one or more destination IP addresses, information about which destination IP address is used by default, inbound stream count (= the outbound stream count that was requested on the other side), and outbound stream count (= maximum number of allowed outbound streams).

The `ACCEPT` primitive prepares a NEAT Flow to receive network data. UDP and UDP-Lite do not natively support a POSIX-style accept mechanism; in this case, NEAT emulates this functionality. `P: ACCEPT` can only return once data arrives, not necessarily after the peer has called `P: OPEN` (The callback-based implementation does not have this problem because `P: ACCEPT` does not block).

`E: on_connected`

The `on_connected` event indicates a NEAT peer endpoint has connected, and other primitives can then be used.

3.4. Writing and reading data

The primitives in this section refer to actions that may be performed on an open NEAT Flow, i.e., a NEAT Flow that was either actively established or successfully made available for receiving data. It permits an application to send and receive data-blocks over the API.

E: on_writable

The on_writable event indicates there is buffer space available and the application may write new data using P:WRITE.

P: P: WRITE([stream] [pr_method pr_value] [unordered_flag] data datalen)

stream : the number of the stream to be used (positive integer). This can be omitted if the NEAT Flow contains only one stream.

pr_method and pr_value : if these parameters are used, then partial reliability is enabled and pr_method must have an integer value from 1 to 2 to specify which method to implement partial reliability is requested. Value 1 means: pr_value specifies a time in milliseconds after which it is unnecessary to send this data-block. Value 2 means: pr_value specifies a requested maximum number of attempts to retransmit the data-block. If the selected NEAT transport does not support partial reliability these parameters will be ignored

unordered_flag : The data block may be delivered out-of-order if this boolean flag is set. Default: false. If the protocol chosen by the NEAT Selection components does not support unordered delivery, this parameter will be ignored.

data : The data block to be sent.

datalen : the amount (positive integer) of data supplied in data.

The WRITE primitive provide a NEAT Flow with a data block for transmission to the remote NEAT peer endpoint (with reliability limited by the conditions specified via pr_method, pr_value and the transport protocol used). NEAT Flows can support message delineation as a property of the NEAT Flow that is set via the INIT_FLOW primitive (S. 2.2.1). If a NEAT Flow supports message delineation, the data block is a complete message.

E: on_all_written

The on_all_written event indicates that all data requested to be written using P:WRITE has been sent.

E: on_send_failure

The on_send_failure event may be returned instead of E:on_all_written when the NEAT System was temporarily unable to complete a P:WRITE call, and it not known that all data has been written.

E: on_readable

The on_readable event indicates there is data available for the application that may be read using P:READ.

P: READ()

data : the received data block.

datalen : the amount of data received.

Returns: [unordered_flag] [stream_id] data datalen If a message arrives out of order, this is indicated by unordered_flag. If the underlying transport protocol supports streams, the stream_id parameter is set.

The READ primitive reads a data block from a NEAT Flow into a provided buffer. NEAT Flows can support message delineation as a property of the NEAT Flow that is set via the INIT_FLOW primitive. If a NEAT Flow supports message delineation, the data block is a complete message.

3.5. Flow Maintenance Primitives

The primitives and events below are out-of-band calls that can be issued at any time after a NEAT Flow has been opened and before it has been terminated.

P: CHANGE_TIMEOUT(toval)

toval : the timeout value in seconds.

The CHANGE_TIMEOUT primitive adjusts the time after which a NEAT Flow will terminate if the written data could not be delivered. If this is not called, NEAT will make an automatic default choice for the timeout.

P: SET_PRIMARY(dst_IP_address)

dst_IP_address : the destination IP address that should be used as the primary address.

The SET_PRIMARY primitive is to be used with NEAT Flows that have multiple destination IP addresses, with protocols that do not use load sharing. It should not have an effect otherwise. This will overrule this general per-flow setting. If this is not called, the NEAT System will make an automatic default choice for the destination IP address.

P: SET_LOW_WATERMARK(watermark)

watermark : upper limit of unsent data in the socket buffer, in bytes.

The SET_LOW_WATERMARK primitive allows the application to limit the amount of unsent data in the underlying socket buffer. If set, NEAT will only execute E: WRITABLE when the amount of unsent data falls below the watermark. This allows applications to reduce sender-side queuing delay.

P: SET_MIN_CHECKSUM_COVERAGE(length)

length : The number of bytes that must be covered by the checksum for a datagram to be delivered to the application.

The SET_MIN_CHECKSUM_COVERAGE primitive allows an application to set the minimum acceptable checksum coverage length. This primitive only has effect for a received UDP-Lite datagram. A receiver that receives a UDP-Lite datagram with a smaller coverage length will not hand over the data to the receiving application. This is ignored for other protocols, where all data are fully covered by the checksum.

P: SET_CHECKSUM_COVERAGE(length)

length : sets the number of bytes covered by the checksum on outgoing UDP-Lite datagrams. This is ignored for other protocols, where all data are fully covered by the checksum.

The SET_CHECKSUM_COVERAGE primitive allows an application to set the number of bytes covered by the checksum in a UDP-Lite datagram. This only has effect when the UDP-Lite protocol is selected.

P: SET_TTL(ttl)

ttl : the hop limit to be used for reception.

The SET_TTL primitive sets the minimum IPv4 TTL or IPv6 Hop Count on a datagram that is required for it to be passed to the application.

E: on_network_status_changed

The on_network_status_changed event informs the application that something has happened in the network; it is safe to ignore without harm by many applications. A status code indicates what has happened in accordance with a table that includes at least the following three values: 1) ICMP error message arrived; 2) Excessive retransmissions; 3) one or more destination IP addresses have become available/unavailable.

P: GET_PROPERTY(property)

property : string with a property name.

Returns: value set to the property by the Policy Manager expressed

as JSON.

The GET_PROPERTY primitive allows an application to discover the value assigned to a property by the Policy Manager. Properties are expressed as part of policies and handled by the NEAT Policy Manager and can only be read by an application once a flow has been initialised.

These currently are:

- o Transport parameters: Parameters used (e.g., congestion control mechanism, TCP sysctl parameters, . . .). For example, the choice of congestion control mechanism is likely to depend on the capacity_profile parameter of the INIT_FLOW primitive, if that parameter is used -\u002D but does not specify a concrete congestion control algorithm, which this read- able property returns. More generally, this property gives the application a more concrete view of the choices made by the NEAT System.
- o Interface statistics: Interface MTU, addresses, connection type (link layer), etc.
- o Path statistics: Experienced RTT, packet loss (rate), jitter, throughput, path MTU, etc
- o UsedDSCP: The DSCP assigned to an active NEAT Flow. This may differ from the requested DSCP when the QoS has been mapped by the policy system

3.6. NEAT Flow Termination

This set of primitives and events are related to gracefully or forcefully closing a NEAT Flow, or being informed about this happening.

P: CLOSE()

The CLOSE primitive terminates a NEAT Flow after satisfying all the requirements that were specified regarding the delivery of data that the application has already given to NEAT. If the peer still has data to send, it cannot then be received after this call. Data buffered by the NEAT System that has not yet been given to the network layer will be discarded.

E: on_close

The on_close event informs the application that a NEAT Flow was successfully closed. This can be received at any time for an active NEAT Flow.

P: ABORT()

The ABORT primitive terminates a connection without delivering remaining data.

E: on_aborted

The on_aborted event informs the application that the other side has aborted the NEAT Flow. The event can be received at any time for an active NEAT Flow.

E: on_timeout

The on_timeout event informs the application that the NEAT Flow is aborted because the default timeout has been reached before data could be delivered. This timeout adjusted by the P: CHANGE_TIMEOUT NEAT Flow maintenance primitive. The event can be received at any time for an active NEAT Flow.

3.7. NEAT Error Events

Errors that occur within the NEAT System or that are notified by the network result in an on_error event:

E: on_error

This event notifies a hard or soft error to the upper layer using the NEAT System.

4. Security Considerations

This document is about the design and usage of a transport API. The transport protocols accessed via this API each have security considerations.

The API may be used to request the use of security protocols accessed via the transport API.

5. IANA Considerations

XX RFC ED - PLEASE REMOVE THIS SECTION XXX

This memo includes no request to IANA.

6. References

- [D1.1] Fairhurst, G., Jones, T., Damjanovic, D., Eckert, K., Grinnemo, K., Hansen, A., Mangiante, S., McManus, P., Papastergiou, G., Ros, D., Vyncke, E., Welzl, M. and M. Tuexen, "The NEAT Architecture (D1.1)", 2015, <<https://www.neat-project.org/wp-content/uploads/2016/02/D1.1.pdf>>.

- [D2.3] Khademi, N., Bozakov, Z., Brunstroem, A., Dale, O., Damjanovic, D., Evensen, KR., Fairhurst, G., Fischer, A., Grinnemo, K., Jones, T., Mangiante, S., Petlund, A., Ros, D., Ruengeler, I., Stenberg, D., Tuexen, M., Weinrank, F. and M. Welzl, "The Final Version of Core Transport System (D2.3)", 2017, <<https://www.neat-project.org/wp-content/uploads/2017/10/D2.3.pdf>>.
- [I-D.grinnemo-taps-he] Grinnemo, K., Brunstrom, A., Hurtig, P., Khademi, N. and Z. Bozakov, "Happy Eyeballs for Transport Selection", Internet-Draft draft-grinnemo-taps-he-03, July 2017.
- [I-D.ietf-tls-tls13] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", Internet-Draft draft-ietf-tls-tls13-21, July 2017.
- [NEAT-DOC] Stenberg, D., Weinrank, F., Khademi, N., Dreibholz, T., Jones, T., Bozakov, Z. and O. Dale, "NEAT Programming API Documentation", , <<http://neat.readthedocs.io/>>.
- [NEAT-GIT] "NEAT Source Code Repository", , <<https://github.com/neat-project/neat>>.
- [NEAT] "The EU New, Evolutive API and Transport-Layer Architecture for the Internet (NEAT) Project", 2017, <<https://www.neat-project.org/>>.
- [RFC7413] Cheng, Y., Chu, J., Radhakrishnan, S. and A. Jain, "TCP Fast Open", RFC 7413, DOI 10.17487/RFC7413, December 2014, <<https://www.rfc-editor.org/info/rfc7413>>.
- [RFC8095] Fairhurst, G., Ed., Trammell, B.Ed., and M. Kuehlewind, Ed., "Services Provided by IETF Transport Protocols and Congestion Control Mechanisms", RFC 8095, DOI 10.17487/RFC8095, March 2017, <<https://www.rfc-editor.org/info/rfc8095>>.

Appendix A. Revision Information

-00 This is an individual draft for the IETF community, for consideration by the IETF TAPS WG.

Authors' Addresses

Godred Fairhurst
University of Aberdeen
Department of Engineering
Fraser Noble Building
Aberdeen, AB24 3UE
Scotland

Email: gorry@erg.abdn.ac.uk
URI: <http://www.erg.abdn.ac.uk/>

Tom Jones
University of Aberdeen
Department of Engineering
Fraser Noble Building
Aberdeen, AB24 3UE
Scotland

Email: tom@erg.abdn.ac.uk
URI: <http://www.erg.abdn.ac.uk/>

Anna Brunstrom
Karlstad University
Universitetsgatan 2
Karlstad, 651 88
Sweden

Email: anna.brunstrom@kau.se

David Ros
Simula Research Laboratory
Martin Linges vei 25
1364 Fornebu
Oslo,
Norway

Email: dros@simula.no

TSVWG
Internet-Draft
Intended status: Informational
Expires: May 15, 2018

G. Fairhurst
T. Jones
University of Aberdeen
A. Brunstrom
Karlstad University
D. Ros
Simula Research Laboratory
November 13, 2017

The NEAT Interface to Transport Services
draft-fairhurst-taps-neat-01

Abstract

The NEAT System provides an example of a system designed to implement the TAPS Transport Services. This document presents the transport services that the NEAT User API provides to an application or upper-layer protocol. It also describes primitives needed to interface to the NEAT Policy Manager and how policies can be adjusted to match the API behaviour to the properties required by an application or upper-layer protocol using the NEAT User API.

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 <http://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 May 15, 2018.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components

extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
2. The NEAT Context	4
3. NEAT User API Primitives and Events	5
3.1. NEAT Flow Initialisation	5
3.2. NEAT Flow Establishment	6
3.3. NEAT Flow Availability	7
3.4. Writing and reading data	8
3.5. Flow Maintenance Primitives	10
3.6. NEAT Flow Termination	12
3.7. NEAT Error Events	12
4. Security Considerations	13
5. Acknowledgements	13
6. IANA Considerations	13
7. References	13
Appendix A. Revision Information	14
Authors' Addresses	14

1. Introduction

The NEAT (New, Evolutive API and Transport-Layer Architecture for the Internet) [NEAT] System provides a call-back driven API to the network transport layer. It presents a set of transport services [RFC8095] that the NEAT User API provides to an application or upper-layer protocol.

The NEAT System has been implemented in the NEAT User Module. The focus of the present document is on the NEAT User API providing transport services to applications. This utilises a lower interface provided by a Kernel Programming Interface (KPI), to access the traditional Socket API or a transport service implemented in userspace.

This has been designed to support one-sided deployment, and a NEAT System can therefore exchange data with a variety of transport peers, including:

- o Another endpoint using the NEAT System
- o An endpoint using native TCP, UDP, or UDP-Lite.
- o An endpoint using SCTP (with explicit use of multi-streaming)
- o WebRTC browsers

Applications that use the NEAT User API can provide information about the features desired from the transport layer and determine the properties of the offered transport service. It is this additional information that enables the NEAT System to move beyond the constraints of the traditional Socket API, since the stack then becomes aware of what the application/user actually desires or requires for each traffic flow. The additional information can be used to automatically identify which transport components (protocol and other transport mechanisms) could be used to realise the required transport service. This can drive the selection by the NEAT System of the best transport components to use and determine how these need to be configured [I-D.grinnemo-taps-he]. In making decisions, the NEAT System can utilise policy information provided at configuration, previously discovered path characteristics and probing techniques. This can be provided by a policy manager acting below the NEAT User API.

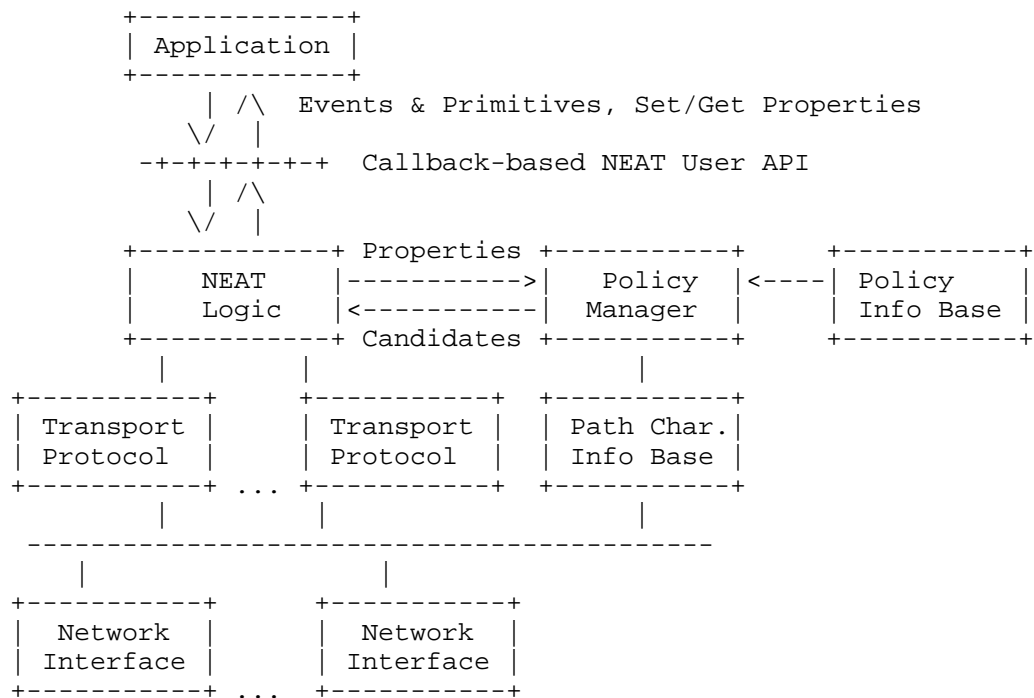


Figure 1: An abstract presentation of the NEAT Architecture and User API.

The architecture of the NEAT System is presented in [D1.1], and depicted in Figure 1. Some important features of NEAT compared to the existing Sockets API are:

- o Event-driven call-back driven interface, enabling applications to be designed to respond to events, such as a signal indicating reception of data blocks, the ability to send data blocks, or the successful transmission of data blocks. This concrete API is described in [D2.3].
- o High-level transport interface, independent of the selected transport protocol, allowing applications to be written without depending on the features of specific transport protocols, and hence allowing the most suitable transport protocol to be matched to the application, based on the transport features an application requires [RFC8095].
- o Support for either unordered/unreliable or reliable transport services.
- o A choice between automatized and explicit support for multistreaming.
- o Explicit support of multipath transport protocols and network architectures.
- o A flexible policy framework, allowing applications to describe the properties they expect or those they require of the transport system and thus enabling the transport services to be configured to match the capabilities of the network that is being used.
- o Ability to work with other network-layer protocols (e.g., network signalling) to realise the required transport service.

The NEAT Library is an open source implementation and is available for download [NEAT-GIT]. This also provides tutorials and examples of code utilising the API and descriptions of the way in which callback mechanisms can be used to build applications that use this interface. Further documentation for the current NEAT System is available at the NEAT Project web page, [NEAT-DOC].

2. The NEAT Context

Applications interact with the network by sending, receiving and controlling NEAT Flows.

The first step in establishing a flow with the NEAT System is to call a primitive to create and configure a Context. In the remainder of this document, the label P: is used to identify a primitive that may be called for a NEAT Context, and the label E: to identify an event provided by the NEAT System. Each primitive/event is associated with a particular NEAT Context. Most primitives specify the Context and provide a handle to the NEAT Flow upon which they operate, and the primitives and events for manipulating data can only be used after a NEAT Flow has been created.

P: INIT_CTX()

The INIT_CTX primitive sets up the datastructures needed by the NEAT System.

After all network operations are completed it can free the context. It returns a pointer to the newly allocated and initialized NEAT context.

P: FREE_CTX()

The FREE_CTX primitive is called when an created context is no longer needed. It frees the memory associated with the datastructures used by the NEAT System.

3. NEAT User API Primitives and Events

An application using the NEAT System needs to take the following steps to use the network after establishing a context:

1. Initialisation: create a flow by calling P: INIT_FLOW; and then calling P: SET_PROPERTIES to express the application requirements. This is used by the NEAT policy manager. Finally, it needs to bind call-back functions to respond to the events generated by the NEAT System.
2. Establishment / Availability: Connect the NEAT Flow (either actively to a destination endpoint or passively to receive from the network).
3. Writing and reading data: Call primitives to write data or respond to events requesting it to read data.
4. Maintenance: Call maintenance primitives, as needed, to configure attributes of the flow (e.g., while writing reading data).
5. Termination: Close (or abort) the NEAT Flow.

3.1. NEAT Flow Initialisation

An application needs to create and initialise a flow object before it can be used.

P: INIT_FLOW()

The INIT_FLOW primitive creates the essential data structures required to use a NEAT Flow. The application also needs to then call a primitive to associate functions with each of the events that it wishes to process.

P: SET_PROPERTIES(property_list)

property_list : A set of flow properties expressed in JSON.

Each NEAT Flow has a set of properties that are set at the flow initialisation time. The SET_PROPERTIES primitive sets properties for the NEAT Flow. Properties are related to Transport Features and Services. For instance: link-layer security, transport-layer security, certificate verification, certificate and key properties can be set at initialisation time are related to a Confidentiality Transport Feature. A flow can also have attributes that can be read by an application using maintenance primitives after a flow has been initialised.

3.2. NEAT Flow Establishment

P: OPEN(destname port [stream_count])

destname : a NEAT-conformant name (which can be a DNS name or a set of IP addresses) to which to connect.

port : port number (integer) or service name (string) to which to connect.

stream_count : the number of requested streams to open (integer). Note that, if this parameter is not used, the system may still use multi-streaming underneath, e.g., by automatically mapping NEAT Flows between the same hosts onto streams of an SCTP association. Using this parameter disables such automatic functionality.

Returns: success or failure. If success, it also returns a handle for a NEAT Flow.

The OPEN primitive opens a flow actively for transports that require a connection handshake (e.g., TCP, SCTP), and opens the flow passively for transports that do not (e.g., UDP, UDP-Lite). Calling P:OPEN alone may not actually have an effect "on the wire", i.e., a P: ACCEPT at the peer may not be triggered by it. Since it is possible that the remote endpoint only returns when data arrives, this may only happen after the local host has called P: WRITE. (This does not result in a problem, since P: ACCEPT does not block).

E: on_connected

The on_connected event indicates a successful connection setup. An application that receives this event can then use other primitives with this flow.

P: OPEN_WITH_EARLY_DATA(destname port [stream_count] [flow_group] [stream] [pr_method pr_value] [unordered_flag] data datalen)

destname : defined in the same way as in P: OPEN.

port : defined in the same way as in P: OPEN.

stream_count : defined in the same way as in P: OPEN.

`flow_group` : defined in the same way as in P: OPEN.

`stream` : the number of the stream to be used. At the moment this function is called, a connection is still not initialised and the protocol may not be known. If the protocol chosen by the NEAT Selection components supports only one stream, this parameter will be ignored.

`pr_method` and `pr_value` : if these parameters are used, then partial reliability is enabled and `pr_method` must have an integer value from 1 to 2 to specify which method to implement partial reliability is requested. Value 1 means: `pr_value` specifies a time in milliseconds after which it is unnecessary to send this data block. Value 2 means: `pr_value` specifies a requested maximum number of attempts to retransmit the data block. If the selected NEAT transport does not support partial reliability these parameters will be ignored. (See P: WRITE for more information).

`unordered_flag` : The data block may be delivered out-of-order if this boolean flag is set. Default: false. If the protocol chosen by the NEAT Selection components does not support unordered delivery, this parameter will be ignored.

`data` : the data-block to be sent.

`datalen` : the amount (positive integer) of data supplied in the data-block.

Returns: success or failure. If success, it also returns a handle for a NEAT Flow and the amount of supplied data that was buffered.

The `OPEN_WITH_EARLY_DATA` primitive allows data to be sent at the time when a flow is opened. To accommodate TLS 1.3 [I-D.ietf-tls-tls13] early data and the TCP Fast Open option [RFC7413], application data need to be supplied at the time of opening a NEAT Flow. This primitive opens a flow and sends early data if the protocol supports it. If the protocol chosen does not support early application data. The data will be buffered then sent after connection establishment, similar to calling P: WRITE. For this reason, in addition to the parameters of P: OPEN, this primitive also needs the same parameters as P: WRITE. The supplied data can be delivered multiple times (replayed by the network); an application must take this into account when using this function. This is commonly known as idempotence.

3.3. NEAT Flow Availability

This section describes how an application prepares a flow to accept communication from another NEAT endpoint.

P: ACCEPT([name] port [stream_count])

name : local NEAT-conformant name (which can be a DNS name or a set of IP addresses) to constrain acceptance of incoming requests to local address(es). If this is missing, requests may arrive at any local address.

port : local port number (integer) or service name (string), to constrain acceptance to incoming requests at this port.

stream_count : the number of requested streams to open (integer). Default value: 1.

Returns: one or more destination IP addresses, information about which destination IP address is used by default, inbound stream count (= the outbound stream count that was requested on the other side), and outbound stream count (= maximum number of allowed outbound streams).

The ACCEPT primitive prepares a NEAT Flow to receive network data. UDP and UDP-Lite do not natively support a POSIX-style accept mechanism; in this case, NEAT emulates this functionality. P: ACCEPT can only return once data arrives, not necessarily after the peer has called P: OPEN (The callback-based implementation does not have this problem because P: ACCEPT does not block).

E: on_connected

The on_connected event indicates a NEAT peer endpoint has connected, and other primitives can then be used.

3.4. Writing and reading data

The primitives in this section refer to actions that may be performed on an open NEAT Flow, i.e., a NEAT Flow that was either actively established or successfully made available for receiving data. It permits an application to send and receive data-blocks over the API.

E: on_writable

The on_writable event indicates there is buffer space available and the application may write new data using P:WRITE.

P: P: WRITE([stream] [pr_method pr_value] [unordered_flag] data datalen)

stream : the number of the stream to be used (positive integer). This can be omitted if the NEAT Flow contains only one stream.

`pr_method` and `pr_value` : if these parameters are used, then partial reliability is enabled and `pr_method` must have an integer value from 1 to 2 to specify which method to implement partial reliability is requested. Value 1 means: `pr_value` specifies a time in milliseconds after which it is unnecessary to send this data-block. Value 2 means: `pr_value` specifies a requested maximum number of attempts to retransmit the data-block. If the selected NEAT transport does not support partial reliability these parameters will be ignored

`unordered_flag` : The data block may be delivered out-of-order if this boolean flag is set. Default: false. If the protocol chosen by the NEAT Selection components does not support unordered delivery, this parameter will be ignored.

`data` : The data block to be sent.

`datalen` : the amount (positive integer) of data supplied in `data`.

The `WRITE` primitive provide a NEAT Flow with a data block for transmission to the remote NEAT peer endpoint (with reliability limited by the conditions specified via `pr_method`, `pr_value` and the transport protocol used). NEAT Flows can support message delineation as a property of the NEAT Flow that is set via the `INIT_FLOW` primitive (S. 2.2.1). If a NEAT Flow supports message delineation, the data block is a complete message.

E: `on_all_written`

The `on_all_written` event indicates that all data requested to be written using `P:WRITE` has been sent.

E: `on_send_failure`

The `on_send_failure` event may be returned instead of `E:on_all_written` when the NEAT System was temporarily unable to complete a `P:WRITE` call, and it not known that all data has been written.

E: `on_readable`

The `on_readable` event indicates there is data available for the application that may be read using `P:READ`.

P: `READ()`

`data` : the received data block.

`datalen` : the amount of data received.

Returns: [`unordered_flag`] [`stream_id`] `data` `datalen`. If a message

arrives out of order, this is indicated by the `unordered_flag`. If the underlying transport protocol supports streams, the `stream_id` parameter is set.

The `READ` primitive reads a data block from a NEAT Flow into a provided buffer. If a NEAT Flow supports message delineation, the data block is a complete message.

3.5. Flow Maintenance Primitives

The primitives and events below are out-of-band calls that can be issued at any time after a NEAT Flow has been opened and before it has been terminated.

P: `CHANGE_TIMEOUT(toval)`

`toval` : the timeout value in seconds.

The `CHANGE_TIMEOUT` primitive adjusts the time after which a NEAT Flow will terminate if the written data could not be delivered. If this is not called, NEAT will make an automatic default choice for the timeout.

P: `SET_PRIMARY(dst_IP_address)`

`dst_IP_address` : the destination IP address that should be used as the primary address.

The `SET_PRIMARY` primitive is to be used with NEAT Flows that have multiple destination IP addresses, with protocols that do not use load sharing. It should not have an effect otherwise. This will overrule this general per-flow setting. If this is not called, the NEAT System will make an automatic default choice for the destination IP address.

P: `SET_LOW_WATERMARK(watermark)`

`watermark` : upper limit of unsent data in the socket buffer, in bytes.

The `SET_LOW_WATERMARK` primitive allows the application to limit the amount of unsent data in the underlying socket buffer. If set, NEAT will only execute E: `WRITABLE` when the amount of unsent data falls below the watermark. This allows applications to reduce sender-side queuing delay.

P: `SET_MIN_CHECKSUM_COVERAGE(length)`

`length` : The number of bytes that must be covered by the checksum for a datagram to be delivered to the application.

The SET_MIN_CHECKSUM_COVERAGE primitive allows an application to set the minimum acceptable checksum coverage length. This primitive only has effect for a received UDP-Lite datagram. A receiver that receives a UDP-Lite datagram with a smaller coverage length will not hand over the data to the receiving application. This is ignored for other protocols, where all data are fully covered by the checksum.

P: SET_CHECKSUM_COVERAGE(length)

length : sets the number of bytes covered by the checksum on outgoing UDP-Lite datagrams. This is ignored for other protocols, where all data are fully covered by the checksum.

The SET_CHECKSUM_COVERAGE primitive allows an application to set the number of bytes covered by the checksum in a UDP-Lite datagram. This only has effect when the UDP-Lite protocol is selected.

P: SET_TTL(ttl)

ttl : the hop limit to be used for reception.

The SET_TTL primitive sets the minimum IPv4 TTL or IPv6 Hop Count on a datagram that is required for it to be passed to the application.

E: on_network_status_changed

The on_network_status_changed event informs the application that something has happened in the network; it is safe to ignore without harm by many applications. A status code indicates what has happened in accordance with a table that includes at least the following three values: 1) ICMP error message arrived; 2) Excessive retransmissions; 3) one or more destination IP addresses have become available/unavailable.

P: GET_PROPERTY(property)

property : string with a property name, expressed as JSON.

Returns: value set to the property returned by the Policy Manager, expressed as JSON.

The GET_PROPERTY primitive allows an application to discover the value assigned to a property by the Policy Manager. Properties are expressed as part of policies and handled by the NEAT Policy Manager and can only be read by an application once a flow has been initialised.

These currently are:

- o Transport parameters: Parameters used (e.g., congestion control mechanism, TCP sysctl parameters, . . .). This property gives the application a more concrete view of the choices that were made.

- o Interface statistics: Interface MTU, addresses, connection type (link layer), etc.
- o Path statistics: Experienced RTT, packet loss (rate), jitter, throughput, path MTU, etc
- o UsedDSCP: The DSCP assigned to an active NEAT Flow. This may differ from the requested DSCP when the QoS has been mapped by the policy system

3.6. NEAT Flow Termination

This set of primitives and events are related to gracefully or forcefully closing a NEAT Flow, or being informed about this happening.

P: CLOSE()

The CLOSE primitive terminates a NEAT Flow after satisfying all the requirements that were specified regarding the delivery of data that the application has already given to NEAT. If the peer still has data to send, it cannot then be received after this call. Data buffered by the NEAT System that has not yet been given to the network layer will be discarded.

E: on_close

The on_close event informs the application that a NEAT Flow was successfully closed. This can be received at any time for an active NEAT Flow.

P: ABORT()

The ABORT primitive terminates a connection without delivering remaining data.

E: on_aborted

The on_aborted event informs the application that the other side has aborted the NEAT Flow. The event can be received at any time for an active NEAT Flow.

E: on_timeout

The on_timeout event informs the application that the NEAT Flow is aborted because the default timeout has been reached before data could be delivered. This timeout adjusted by the P: CHANGE_TIMEOUT NEAT Flow maintenance primitive. The event can be received at any time for an active NEAT Flow.

3.7. NEAT Error Events

Errors that occur within the NEAT System or that are notified by the network result in an `on_error` event:

E: `on_error`

This event notifies a hard or soft error to the upper layer using the NEAT System.

4. Security Considerations

This document is about the design and usage of a transport API. The transport protocols accessed via this API each have security considerations.

The API may be used to request the use of security protocols accessed via the transport API.

5. Acknowledgements

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).

6. IANA Considerations

XX RFC ED - PLEASE REMOVE THIS SECTION XXX

This memo includes no request to IANA.

7. References

[D1.1] Fairhurst, G., Jones, T., Damjanovic, D., Eckert, K., Grinnemo, K., Hansen, A., Mangiante, S., McManus, P., Papastergiou, G., Ros, D., Vyncke, E., Welzl, M. and M. Tuexen, "The NEAT Architecture (D1.1)", 2015, <<https://www.neat-project.org/wp-content/uploads/2016/02/D1.1.pdf>>.

[D2.3] Khademi, N., Bozakov, Z., Brunstroem, A., Dale, O., Damjanovic, D., Evensen, KR., Fairhurst, G., Fischer, A., Grinnemo, K., Jones, T., Mangiante, S., Petlund, A., Ros, D., Ruengeler, I., Stenberg, D., Tuexen, M., Weinrank, F. and M. Welzl, "The Final Version of Core Transport System (D2.3)", 2017, <<https://www.neat-project.org/wp-content/uploads/2017/10/D2.3.pdf>>.

[I-D.grinnemo-taps-he] Grinnemo, K., Brunstrom, A., Hurtig, P., Khademi, N. and Z. Bozakov, "Happy Eyeballs for Transport Selection", Internet-Draft draft-grinnemo-taps-he-03, July 2017.

[I-D.ietf-tls-tls13]

Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", Internet-Draft draft-ietf-tls-tls13-21, July 2017.

[NEAT-DOC]

Stenberg, D., Weinrank, F., Khademi, N., Dreibholz, T., Jones, T., Bozakov, Z. and O. Dale, "NEAT Programming API Documentation", , <<http://neat.readthedocs.io/>>.

[NEAT-GIT]

"NEAT Source Code Repository", , <<https://github.com/neat-project/neat>>.

[NEAT]

"The EU New, Evolutive API and Transport-Layer Architecture for the Internet (NEAT) Project", 2017, <<https://www.neat-project.org/>>.

[RFC7413]

Cheng, Y., Chu, J., Radhakrishnan, S. and A. Jain, "TCP Fast Open", RFC 7413, DOI 10.17487/RFC7413, December 2014, <<https://www.rfc-editor.org/info/rfc7413>>.

[RFC8095]

Fairhurst, G., Ed., Trammell, B.Ed., and M. Kuehlewind, Ed., "Services Provided by IETF Transport Protocols and Congestion Control Mechanisms", RFC 8095, DOI 10.17487/RFC8095, March 2017, <<https://www.rfc-editor.org/info/rfc8095>>.

Appendix A. Revision Information

-00 This is an individual draft for the IETF community, for consideration by the IETF TAPS WG.

-01 Contains corrections to INIT_FLOW; fixes to typos; and includes the Acknowledgment text omitted by mistake in -00.

Authors' Addresses

Godred Fairhurst
University of Aberdeen
Department of Engineering
Fraser Noble Building
Aberdeen, AB24 3UE
Scotland

Email: gorry@erg.abdn.ac.uk
URI: <http://www.erg.abdn.ac.uk/>

Tom Jones
University of Aberdeen
Department of Engineering
Fraser Noble Building
Aberdeen, AB24 3UE
Scotland

Email: tom@erg.abdn.ac.uk
URI: <http://www.erg.abdn.ac.uk/>

Anna Brunstrom
Karlstad University
Universitetsgatan 2
Karlstad, 651 88
Sweden

Email: anna.brunstrom@kau.se

David Ros
Simula Research Laboratory
Martin Linges vei 25
1364 Fornebu
Oslo,
Norway

Email: dros@simula.no

TAPS
Internet-Draft
Intended status: Experimental
Expires: December 3, 2017

K-J. Grinnemo
A. Brunstrom
P. Hurtig
Karlstad University
N. Khademi
University of Oslo
Z. Bozakov
Dell EMC Research Europe
June 2017

Happy Eyeballs for Transport Selection
draft-grinnemo-taps-he-03

Abstract

Ideally, network applications should be able to select an appropriate transport solution from among available transport solutions. However, at present, there is no agreed-upon way to do this. In fact, there is not even an agreed-upon way for a source end host to determine if there is support for a particular transport along a network path. This draft addresses these issues, by proposing a Happy Eyeballs framework. The proposed Happy Eyeballs framework enables the selection of a transport solution that according to application requirements, pre-set policies, and estimated network conditions is the most appropriate one. Additionally, the proposed framework makes it possible for an application to find out whether a particular transport is supported along a network connection towards a specific destination or not.

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 <http://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 December 3, 2017.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Definitions	2
2. Introduction	2
3. Problem Statement	3
4. The Happy Eyeballs Framework	4
5. Design and Implementation Considerations	5
5.1. Candidate List Generation	5
5.2. Caching	7
5.3. Concurrent Connection Attempts	7
6. Example Happy Eyeballs Scenario	7
7. IANA Considerations	8
8. Security Considerations	8
9. Acknowledgements	8
10. References	8
10.1. Normative References	9
10.2. Informative References	9
Authors' Addresses	9

1. Definitions

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 RFC 2119 [RFC2119].

2. Introduction

Information services on the Internet come in varying forms, such as web browsing, email, and on-demand multimedia. The main motivation behind the design of next-generation computer and communications networks is to provide a universal and easy access to these various types of information services on a single multi-service Internet. This means that all forms of communications, e.g., video, voice, data

and control signaling, along with all types of services -- from plain text web pages to multimedia applications -- are bonded in a single-service platform through Internet technology. To enable the next-generation networks, the TAPS Working Group suggests a decoupling between the transport service provided to an application, and the transport stack providing this transport service: An application requests an appropriate transport service on the basis of its transport requirements, and the available transport stack that best meets these requirements is selected. In case the most preferred transport stack is not supported along the network path to the destination, or is not supported by the end host, a less-preferred transport stack is selected instead. As a way to realize the selection of transport stacks, this document suggests a generalization of the Happy Eyeballs (HE) mechanism proposed in Wing et al. [RFC6555] which addresses the selection of complete transport solutions, and which lends itself to arbitrary transport selection criterias. The proposed HE mechanism targets connection-oriented transport solutions, and connectionless transport solutions provided they offer some reasonable way to determine their successful use between endpoints.

The HE mechanism was introduced as a means to promote the use of dual network stacks. Dual-stack client applications should be encouraged to try setting up connections over IPv6 first, and fall back to using IPv4 if IPv6 connection attempts fail. However, serializing tests for IPv6 and IPv4 connectivity can result in large connection latencies. HE for IPv6 minimizes the cost in delay by parallelizing attempts over IPv6 and IPv4. HE has also been proposed as an efficient way to find out the optimal combination of IPv4/IPv6 and TCP/SCTP to use to connect to a server [I-D.wing-tsvwg-happy-eyeballs-sctp]. The HE framework suggested in this document could be seen as a natural continuation of this proposal.

3. Problem Statement

Currently, there is no agreed-upon way for a source end host to select an appropriate transport service for a given application. In fact, there is no common way for a source end-host to find out if a transport stack is supported along a network path between itself and a destination end host. As a consequence, it has become increasingly difficult to introduce new transport stacks, and several applications, including many web applications, run over TCP although there are other transport protocols that better meet the requirements of these applications.

4. The Happy Eyeballs Framework

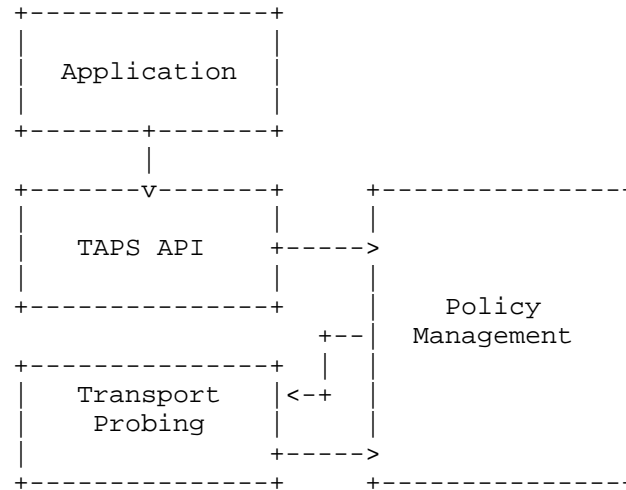


Figure 1: The Happy Eyeballs Framework.

The generalized HE mechanism proposed in this draft is carried out within the framework depicted in Figure 1. It comprises the following steps:

1. The Policy Management component takes as input application requirements from the TAPS API, stored information about previous connection attempts (e.g., whether previous connection attempts succeeded or not), and network conditions and configurations. On the basis of this input and the policies configured in the system, the Policy Management component creates a list of candidate transport solutions, *L*, sorted in decreasing priority order. To be compliant with RFC 6555 [RFC6555], the Policy Management component SHOULD, in those cases there are no policies telling otherwise, following the host's address preference, something which usually means giving preference to IPv6 over IPv4.
2. It is the responsibility of the Transport Probing component to select the most appropriate transport solution. This is done by initiating connection attempts for each transport solution on *L*. To minimize the number of connection attempts that are initiated, the Transport Probing component SHOULD cache the outcome of connection attempts in a repository kept by the Policy Management component. The Policy Management component SHOULD in turn only include those transport solutions on *L* that have not been previously attempted, have valid successful connection-attempt

cache entries, or have previously been attempted but whose cached connection-attempt entries have expired. Cached connection-attempt results SHOULD be valid for a configurable amount of time after which they SHOULD expire and have to be repeated. The transport solutions on L are initiated in priority order. The difference in priority between two consecutive candidates, C1 and C2, is translated according to some criteria to a delay, D. D then governs the delay between the initiation of the connection attempts C1 and C2.

3. After the initiation of the connection attempts, the Transport Probing component waits for the first or winning connection to be established, which becomes the selected transport solution. For the Transport Probing component to be able to efficiently use the connection-attempt cache, already-initiated, non-winning connection attempts SHOULD be given a fair chance to complete. In that way, the connection-attempt cache will be provided with a fairly accurate knowledge of which transport solutions work and does not work against frequently visited transport endpoints. Moreover, it MAY be beneficial to let those transport solutions which have a higher priority than the winning transport solution, live a predetermined amount of time after their establishment, since this enables the reuse of already established connections in later application requests.

5. Design and Implementation Considerations

This section discusses implementation issues that should be considered when a HE mechanism is designed and implemented on the basis of the HE framework proposed in this document.

5.1. Candidate List Generation

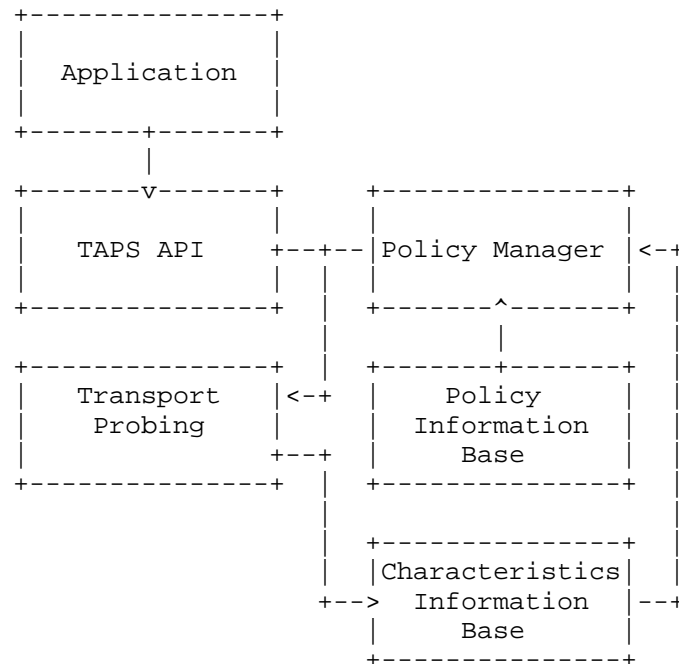


Figure 2: Principle Design of the NEAT Happy Eyeballs Framework.

There are several ways in which the list of candidate transport solutions, L , could be created by the Policy Management component. For example, L could be a list of all available transport solutions in an order that, except for following the host's address preference, is arbitrary; another, more sophisticated, way of creating the list of candidate transport solutions is the one employed by the NEAT System.

The NEAT System is developed as part of the EU Horizon 2020 project, "A New, Evolutive API and Transport-Layer Architecture for the Internet" (NEAT) [NEAT-Webb], and aims to provide a flexible and evolvable transport system that aligns with the charter of the TAPS Working Group. In the NEAT System [NEAT-Git], the HE framework is realized as shown in Figure 2. As follows, the Policy Management component comprises three components in the NEAT HE framework: a Policy Manager (PM), a Policy Information Base (PIB), and a Characteristics Information Base (CIB). PIB is a repository that stores a collection of policies that map application requests to transport solutions, i.e., map application requests to appropriately configured transport protocols, and CIB is a repository that stores information about previous connection attempts, available network

interfaces, supported transport protocols etc. The PM takes as input application requirements from the TAPS API, and information from PIB and CIB. On the basis of this input, the PM creates L.

5.2. Caching

As pointed out in RFC 6555 [RFC6555], a HE algorithm should not waste networking resources by routinely making simultaneous connection attempts. To this end, the HE algorithm should cache the outcome of previous connection attempts to the same peer. The cache lifetime is considered system dependent and should be set on a case-by-case basis. The impact and efficiency of the HE algorithm have been evaluated in [Papastergioul6]. The paper suggests that caching significantly reduces the CPU load imposed by a HE mechanism. It also indicates that the internal-memory footprint of a HE mechanism is essentially the same as for single-flow establishments.

5.3. Concurrent Connection Attempts

As mentioned in Section 4, it is the responsibility of the Transport Probing component to choose the most appropriate transport solution on the list of candidate transport solutions, L. Often this implies that several transport solutions need to be tried out, something which should not be carried out sequentially, but concurrently or partly overlapping depending on the transport-solution priorities. The way this is done is implementation dependent and varies between platforms. The NEAT library [NEAT-Git], which implements the HE framework herein, is built around the libuv asynchronous I/O library [LIBUV] and uses an event-based concurrency model to realize the concurrent initialization of connection attempts. The rationale behind using an event-based concurrency model is at least twofold: The first is that correctly managing concurrency in multi-threaded applications can be challenging with, for example, missing locks or deadlocks. The second is that multi-threading typically offers little or no control over what is scheduled at a given moment in time. Given the complexity of building a general-purpose scheduler that works well in all cases, sometimes the OS will schedule work in a manner that is less than optimal. Those in favor of threads argue that threads are a natural extension of sequential programming in that it maps work to be executed with individual threads. Threads are also a well-known and understood parts of OSes, and are mandatory for exploiting true CPU concurrency.

6. Example Happy Eyeballs Scenario

Consider a scenario in which an IPv6-enabled client using the NEAT System wishes to setup a connection to a server. Assume both the client and server support SCTP and TCP. The Policy Management is

queried about feasible transport solutions to connect to the server. In the NEAT System, this results in PM retrieving information about network connections against this server from the CIB, e.g., supported transport protocols and the outcome of previous connection attempts. In our scenario, the PM learns from the CIB that the server supports SCTP and TCP, and, for the sake of this example, let us assume that the PM is also informed that previous connection attempts against this server, using both SCTP and TCP, were successful. Next, the PM retrieves applicable policies from the PIB, and combines these policies with the previously retrieved CIB information. We assume in this example that the SCTP transport solution has a higher priority than the TCP solution. As a next step, the PM puts together the feasible candidate transport solutions in a list with SCTP over IPv6 placed at the head of the list followed by TCP over IPv6, and supplies this list to the Transport Probing component. The Transport Probing component traverses the candidate list, and initiates a connection attempt with SCTP against the server followed after a short while (governed by the difference in priorities between the SCTP and TCP transport solutions) by a connection attempt with TCP against the server. In our example, assume both connection attempts are successful, however, the SCTP connection attempt completes before the TCP attempt. The Transport Probing component caches in the CIB the SCTP connection attempt as successful, and returns the SCTP connection as the winning connection. When the TCP connection is established some time later, the Transport Probing component caches that connection attempt as successful as well.

7. IANA Considerations

XX RFC ED - PLEASE REMOVE THIS SECTION XXX

This memo includes no request to IANA.

8. Security Considerations

Security will be considered in future versions of this document.

9. Acknowledgements

This work has received funding from 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).

10. References

10.1. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<http://www.rfc-editor.org/info/rfc2119>>.
- [RFC6555] Wing, D. and A. Yourtchenko, "Happy Eyeballs: Success with Dual-Stack Hosts", RFC 6555, DOI 10.17487/RFC6555, April 2012, <<http://www.rfc-editor.org/info/rfc6555>>.

10.2. Informative References

- [I-D.wing-tsvwg-happy-eyeballs-sctp]
Wing, D. and P. Natarajan, "Happy Eyeballs: Trending Towards Success with SCTP", draft-wing-tsvwg-happy-eyeballs-sctp-02 (work in progress), October 2010.
- [LIBUV] libuv -- Asynchronous I/O Made Simple, "<http://libuv.org>", March 2017.
- [NEAT-Git]
A New, Evolutive API and Transport-Layer Architecture for the Internet (NEAT), "<https://github.com/NEAT-project/neat>", March 2017.
- [NEAT-Webb]
NEAT -- A New, Evolutive API and Transport-Layer Architecture for the Internet, "<https://www.neat-project.org>", March 2017.
- [Papastergioul6]
Papastergiou, G., Grinnemo, K-J., Brunstrom, A., Ros, D., Tuexen, M., Khademi, N., and P. Hurtig, "On the Cost of Using Happy Eyeballs for Transport Protocol Selection", July 2016.

Authors' Addresses

Karl-Johan Grinnemo
Karlstad University
Universitetsgatan 2
Karlstad 651 88
Sweden

Phone: +46 54 700 24 40
Email: karl-johan.grinnemo@kau.se

Anna Brunstrom
Karlstad University
Universitetsgatan 2
Karlstad 651 88
Sweden

Phone: +46 54 700 17 95
Email: anna.brunstrom@kau.se

Per Hurtig
Karlstad University
Universitetsgatan 2
Karlstad 651 88
Sweden

Phone: +46 54 700 23 35
Email: per.hurtig@kau.se

Naeem Khademi
University of Oslo
PO Box 1080 Blindern
Oslo N-0316
Norway

Email: naeemk@ifi.uio.no

Zdravko Bozakov
Dell EMC Research Europe
Ovens, Co.
Cork
Ireland

Phone: +353 21 4945733
Email: Zdravko.Bozakov@dell.com

TAPS
Internet-Draft
Intended status: Informational
Expires: April 25, 2018

M. Welzl
S. Gjessing
University of Oslo
October 22, 2017

A Minimal Set of Transport Services for TAPS Systems
draft-ietf-taps-minset-00

Abstract

This draft recommends a minimal set of IETF Transport Services offered by end systems supporting TAPS, and gives guidance on choosing among the available mechanisms and protocols. It is based on the set of transport features given in the TAPS document draft-ietf-taps-transports-usage-08.

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 April 25, 2018.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
2. Terminology	4
3. The Minimal Set of Transport Features	5
3.1. Flow Creation	5
3.2. Flow Connection and Termination	7
3.3. Flow Group Configuration	8
3.4. Flow Configuration	8
3.5. Data Transfer	9
3.5.1. The Sender	9
3.5.2. The Receiver	10
4. An MinSet Abstract Interface	11
4.1. Specification	12
5. Conclusion	17
6. Acknowledgements	18
7. IANA Considerations	18
8. Security Considerations	18
9. References	18
9.1. Normative References	18
9.2. Informative References	19
Appendix A. Deriving the minimal set	21
A.1. Step 1: Categorization -- The Superset of Transport Features	21
A.1.1. CONNECTION Related Transport Features	23
A.1.2. DATA Transfer Related Transport Features	38
A.2. Step 2: Reduction -- The Reduced Set of Transport Features	43
A.2.1. CONNECTION Related Transport Features	44
A.2.2. DATA Transfer Related Transport Features	45
A.3. Step 3: Discussion	46
A.3.1. Sending Messages, Receiving Bytes	46
A.3.2. Stream Schedulers Without Streams	48
A.3.3. Early Data Transmission	49
A.3.4. Sender Running Dry	50
A.3.5. Capacity Profile	50
A.3.6. Security	51
A.3.7. Packet Size	51
Appendix B. Revision information	52
Authors' Addresses	53

1. Introduction

The task of any system that implements TAPS is to offer transport services to its applications, i.e. the applications running on top of TAPS, without binding them to a particular transport protocol. Currently, the set of transport services that most applications use is based on TCP and UDP; this limits the ability for the network

stack to make use of features of other protocols. For example, if a protocol supports out-of-order message delivery but applications always assume that the network provides an ordered bytestream, then the network stack can never utilize out-of-order message delivery: doing so would break a fundamental assumption of the application.

By exposing the transport services of multiple transport protocols, a TAPS system can make it possible to use these services without having to statically bind an application to a specific transport protocol. The first step towards the design of such a system was taken by [RFC8095], which surveys a large number of transports, and [TAPS2] as well as [TAPS2UDP], which identify the specific transport features that are exposed to applications by the protocols TCP, MPTCP, UDP(-Lite) and SCTP as well as the LEDBAT congestion control mechanism. The present draft is based on these documents and follows the same terminology (also listed below).

The number of transport features of current IETF transports is large, and exposing all of them has a number of disadvantages: generally, the more functionality is exposed, the less freedom a TAPS system has to automate usage of the various functions of its available set of transport protocols. Some functions only exist in one particular protocol, and if an application would use them, this would statically tie the application to this protocol, counteracting the purpose of a TAPS system. Also, if the number of exposed features is exceedingly large, a TAPS system might become very hard to use for an application programmer. Taking [TAPS2] as a basis, this document therefore develops a minimal set of transport features, removing the ones that could be harmful to the purpose of a TAPS system but keeping the ones that must be retained for applications to benefit from useful transport functionality.

Applications use a wide variety of APIs today. The transport features in the minimal set in this document must be reflected in **all** network APIs in order for the underlying functionality to become usable everywhere. For example, it does not help an application that talks to a middleware if only the Berkeley Sockets API is extended to offer "unordered message delivery", but the middleware only offers an ordered bytestream. Both the Berkeley Sockets API and the middleware would have to expose the "unordered message delivery" transport feature (alternatively, there may be interesting ways for certain types of middleware to use some transport features without exposing them, based on knowledge about the applications -- but this is not the general case). In most situations, in the interest of being as flexible and efficient as possible, the best choice will be for a middleware or library to expose at least all of the transport features that are recommended as a "minimal set" here.

This "minimal set" can be implemented one-sided with a fall-back to TCP (or UDP, if certain limitations are put in place). This means that a sender-side TAPS system can talk to a non-TAPS TCP (or UDP) receiver, and a receiver-side TAPS system can talk to a non-TAPS TCP (or UDP) sender. For systems that do not have this requirement, [I-D.trammell-taps-post-sockets] describes a way to extend the functionality of the minimal set such that some of its limitations are removed.

2. Terminology

The following terms are used throughout this document, and in subsequent documents produced by TAPS that describe the composition and decomposition of transport services.

Transport Feature: a specific end-to-end feature that the transport layer provides to an application. Examples include confidentiality, reliable delivery, ordered delivery, message-versus-stream orientation, etc.

Transport Service: a set of Transport Features, without an association to any given framing protocol, which provides a complete service to an application.

Transport Protocol: an implementation that provides one or more different transport services using a specific framing and header format on the wire.

Transport Service Instance: an arrangement of transport protocols with a selected set of features and configuration parameters that implements a single transport service, e.g., a protocol stack (RTP over UDP).

Application: an entity that uses the transport layer for end-to-end delivery data across the network (this may also be an upper layer protocol or tunnel encapsulation).

Application-specific knowledge: knowledge that only applications have.

Endpoint: an entity that communicates with one or more other endpoints using a transport protocol.

Connection: shared state of two or more endpoints that persists across messages that are transmitted between these endpoints.

Socket: the combination of a destination IP address and a destination port number.

Moreover, throughout the document, the protocol name "UDP(-Lite)" is used when discussing transport features that are equivalent for UDP and UDP-Lite; similarly, the protocol name "TCP" refers to both TCP and MPTCP.

3. The Minimal Set of Transport Features

Based on the categorization, reduction and discussion in Appendix A, this section describes the minimal set of transport features that is offered by end systems supporting TAPS. This TAPS system is able to fall back to TCP; elements of the system that may prohibit falling back to UDP are marked with "!UDP". To implement a TAPS system that is also able to fall back to UDP, these marked transport features should be excluded.

3.1. Flow Creation

A TAPS flow must be "created" before it is connected, to allow for initial configurations to be carried out. All configuration parameters in Section 3.3 and Section 3.4 can be used initially, although some of them may only take effect when the flow has been connected. Configuring a flow early helps a TAPS system make the right decisions. In particular, the "group number" can influence the TAPS system to implement a TAPS flow as a stream of a multi-streaming protocol's existing association or not.

For flows that use a new "group number", early configuration is necessary because it allows the TAPS system to know which protocols it should try to use (to steer a mechanism such as "Happy Eyeballs" [I-D.grinnemo-taps-he]). In particular, a TAPS system that only makes a one-time choice for a particular protocol must know early about strict requirements that must be kept, or it can end up in a deadlock situation (e.g., having chosen UDP and later be asked to support reliable transfer). As one possibility to correctly handle these cases, we provide the following decision tree (this is derived from Appendix A.2.1 excluding authentication, as explained in Section 8):

- Will it ever be necessary to offer any of the following?

- * Reliably transfer data
- * Notify the peer of closing/aborting
- * Preserve data ordering

Yes: SCTP or TCP can be used.

- Is any of the following useful to the application?
 - * Choosing a scheduler to operate between flows in a group, with the possibility to configure a priority or weight per flow
 - * Configurable message reliability
 - * Unordered message delivery
 - * Request not to delay the acknowledgement (SACK) of a message

Yes: SCTP is preferred.

No:

- Is any of the following useful to the application?
 - * Hand over a message to reliably transfer (possibly multiple times) before connection establishment
 - * Suggest timeout to the peer
 - * Notification of Excessive Retransmissions (early warning below abortion threshold)
 - * Notification of ICMP error message arrival

Yes: TCP is preferred.

No: SCTP and TCP are equally preferable.

No: all protocols can be used.

- Is any of the following useful to the application?
 - * Specify checksum coverage used by the sender
 - * Specify minimum checksum coverage required by receiver

Yes: UDP-Lite is preferred.

No: UDP is preferred.

Note that this decision tree is not optimal for all cases. For example, if an application wants to use "Specify checksum coverage used by the sender", which is only offered by UDP-Lite, and "Configure priority or weight for a scheduler", which is only offered by SCTP, the above decision tree will always choose UDP-Lite, making it impossible to use SCTP's schedulers with priorities between flows in a group. The TAPS system must know which choice is more important for the application in order to make the best decision. We caution implementers to be aware of the full set of trade-offs, for which we recommend consulting the list in Appendix A.2.1 when deciding how to initialize a flow.

Once a flow is created, it can be queried for the maximum amount of data that an application can possibly expect to have reliably transmitted before or during connection establishment (with zero being a possible answer). An application can also give the flow a message for reliable transmission before or during connection establishment (!UDP); the TAPS system will then try to transmit it as early as possible. An application can facilitate sending the message particularly early by marking it as "idempotent"; in this case, the receiving application must be prepared to potentially receive multiple copies of the message (because idempotent messages are reliably transferred, asking for idempotence is not necessary for systems that support UDP-fall-back).

3.2. Flow Connection and Termination

To be compatible with multiple transports, including streams of a multi-streaming protocol (used as if they were transports themselves), the semantics of opening and closing need to be the most restrictive subset of all of them. For example, TCP's support of half-closed connections can be seen as a feature on top of the more restrictive "ABORT"; this feature cannot be supported because not all protocols used by a TAPS system (including streams of an association) support half-closed connections.

After creation, a flow can be actively connected to the other side using "Connect", or it can passively listen for incoming connection requests with "Listen". Note that "Connect" may or may not trigger a notification on the listening side. It is possible that the first notification on the listening side is the arrival of the first data that the active side sends (a receiver-side TAPS system could handle this by continuing to block a "Listen" call, immediately followed by issuing "Receive", for example; callback-based implementations may simply skip the equivalent of "Listen"). This also means that the active opening side is assumed to be the first side sending data.

A TAPS system can actively close a connection, i.e. terminate it after reliably delivering all remaining data to the peer, or it can abort it, i.e. terminate it without delivering remaining data. Unless all data transfers only used unreliable frame transmission without congestion control (i.e., UDP-style transfer), closing a connection is guaranteed to cause an event to notify the peer application that the connection has been closed (!UDP). Similarly, for anything but (UDP-style) unreliable non-congestion-controlled data transfer, aborting a connection will cause an event to notify the peer application that the connection has been aborted (!UDP). A timeout can be configured to abort a flow when data could not be delivered for too long (!UDP); however, timeout-based abortion does not notify the peer application that the connection has been aborted.

Because half-closed connections are not supported, when a TAPS host receives a notification that the peer is closing or aborting the flow (!UDP), the other side may not be able to read outstanding data. This means that unacknowledged data residing in the TAPS system's send buffer may have to be dropped from that buffer upon arrival of a notification to close or abort the flow from the peer.

3.3. Flow Group Configuration

A flow group can be configured with a number of transport features, and there are some notifications to applications about a flow group. Here we list transport features and notifications from Appendix A.2 that sometimes automatically apply to groups of flows (e.g., when a flow is mapped to a stream of a multi-streaming protocol).

Timeout, error notifications:

- o Change timeout for aborting connection (using retransmit limit or time value) (!UDP)
- o Suggest timeout to the peer (!UDP)
- o Notification of Excessive Retransmissions (early warning below abortion threshold)
- o Notification of ICMP error message arrival

Others:

- o Choose a scheduler to operate between flows of a group
- o Obtain ECN field

The following transport features are new or changed, based on the discussion in Appendix A.3:

- o Capacity profile
This describes how an application wants to use its available capacity. Choices can be "lowest possible latency at the expense of overhead" (which would disable any Nagle-like algorithm), "scavenger", and some more values that help determine the DSCP value for a flow (e.g. similar to table 1 in [I-D.ietf-tsvwg-rtcweb-qos]).

3.4. Flow Configuration

Here we list transport features and notifications from Appendix A.2 that only apply to a single flow.

Configure priority or weight for a scheduler

Checksums:

- o Disable checksum when sending
- o Disable checksum requirement when receiving
- o Specify checksum coverage used by the sender
- o Specify minimum checksum coverage required by receiver

3.5. Data Transfer

3.5.1. The Sender

This section discusses how to send data after flow establishment. Section 3.2 discusses the possibility to hand over a message to reliably send before or during establishment.

Here we list per-frame properties that a sender can optionally configure if it hands over a delimited frame for sending with congestion control (!UDP), taken from Appendix A.2:

- o Configurable Message Reliability
- o Ordered message delivery (potentially slower than unordered)
- o Unordered message delivery (potentially faster than ordered)
- o Request not to bundle messages
- o Request not to delay the acknowledgement (SACK) of a message

Additionally, an application can hand over delimited frames for unreliable transmission without congestion control (note that such applications should perform congestion control in accordance with [RFC2914]). Then, none of the per-frame properties listed above have any effect, but it is possible to use the transport feature "Specify DF field" to allow/disallow fragmentation.

Following Appendix A.3.7, there are three transport features (two old, one new) and a notification:

- o Get max. transport frame size that may be sent without fragmentation from the configured interface
This is optional for a TAPS system to offer, and may return an error ("not available"). It can aid applications implementing Path MTU Discovery.
- o Get max. transport frame size that may be received from the configured interface
This is optional for a TAPS system to offer, and may return an error ("not available").

- o Get maximum transport frame size
Irrespective of fragmentation, there is a size limit for the messages that can be handed over to SCTP or UDP(-Lite); because a TAPS system is independent of the transport, it must allow a TAPS application to query this value -- the maximum size of a frame in an Application-Framed-Bytestream. This may also return an error when frames are not delimited ("not available").

There are two more sender-side notifications. These are unreliable, i.e. a TAPS system cannot be assumed to implement them, but they may occur:

- o Notification of send failures
A TAPS system may inform a sender application of a failure to send a specific frame.
- o Notification of draining below a low water mark
A TAPS system can notify a sender application when the TAPS system's filling level of the buffer of unsent data is below a configurable threshold in bytes. Even for TAPS systems that do implement this notification, supporting thresholds other than 0 is optional.

"Notification of draining below a low water mark" is a generic notification that tries to enable uniform access to "TCP_NOTSENT_LOWAT" as well as the "SENDER DRY" notification (as discussed in Appendix A.3.4 -- SCTP's "SENDER DRY" is a special case where the threshold (for unsent data) is 0 and there is also no more unacknowledged data in the send buffer). Note that this threshold and its notification should operate across the buffers of the whole TAPS system, i.e. also any potential buffers that the TAPS system itself may use on top of the transport's send buffer.

3.5.2. The Receiver

A receiving application obtains an Application-Framed Bytestream. Similar to TCP's receiver semantics, it is just a stream of bytes. If frame boundaries were specified by the sender, a receiver-side TAPS system will still not inform the receiving application about them. Within the bytestream, frames themselves will always stay intact (partial frames are not supported - see Appendix A.3.1). Different from TCP's semantics, there is no guarantee that all frames in the bytestream are transmitted from the sender to the receiver,

and that all of them are in the same sequence in which they were handed over by the sender. If an application is aware of frame delimiters in the bytestream, and if the sender-side application has informed the TAPS system about these boundaries and about potentially relaxed requirements regarding the sequence of frames or per-frame reliability, frames within the receiver-side bytestream may be out-of-order or missing.

4. An MinSet Abstract Interface

Here we present the minimum set in the form of an abstract interface that a TAPS system could implement. This abstract interface is derived from the description in the previous section. The primitives of this abstract interface can be implemented in various ways. For example, information that is provided to an application can either be offered via a primitive that is polled, or via an asynchronous notification.

We note that this is just a different form to represent the text in the previous section, and not an abstract API that is recommended to be implemented in this form by all TAPS systems. Specifically, TAPS systems implementing this specific abstract interface would have the following properties:

1. Support one-sided deployment with a fall-back to TCP (or UDP)
2. Offer all the transport features of (MP)TCP, UDP(-Lite), LEDBAT and SCTP that require application-specific knowledge
3. Not offer any of the transport features of these protocols and the LEDBAT congestion control mechanism that do not require application-specific knowledge (to give maximum flexibility to a TAPS system)

This reciprocally means that this is probably not the ideal interface to implement for systems that:

1. Assume that there is a system on both sides -- in this case, richer functionality can be provided (cf. [I-D.trammell-taps-post-sockets]) -- or assume different fall-back protocols than TCP or UDP
2. Use other protocols than (MP)TCP, UDP(-Lite), SCTP or the LEDBAT congestion control mechanism underneath the TAPS interface
3. Want to offer transport features that do not require application-specific knowledge

4.1. Specification

```
CREATE (flow-group-id, reliability, checksum_coverage,  
config_msg_prio, earlymsg_timeout_notifications)  
Returns: flow-id
```

Create a flow and associate it with an existing or new flow group number. The group number can influence the TAPS system to implement a TAPS flow as a stream of a multi-streaming protocol's existing association or not, and the other parameters serve as input to the decision tree described in Section 3.1. The TAPS systems gives no guarantees about honoring any of the requests at this stage, these parameters are just meant to help it to choose and configure a suitable protocol.

PARAMETERS:

flow-group-id: the flow's group number; all other parameters are only relevant when this number is not currently in use by an ongoing flow to the same destination (in which case the flow becomes a member of the existing flow's group and inherits the configuration of the group).

reliability: a boolean that should be set to true when any of the following will be useful to the application: reliably transfer data; notify the peer of closing/aborting; preserve data ordering.

checksum_coverage: a boolean to specify whether it will be useful to the application to specify checksum coverage when sending or receiving.

config_msg_prio: a boolean that should be set to true when any of the following per-message configuration or prioritization mechanisms will be useful to the application: choosing a scheduler to operate between flows in a group, with the possibility to configure a priority or weight per flow; configurable message reliability; unordered message delivery; requesting not to delay the acknowledgement (SACK) of a message.

earlymsg_timeout_notifications: a boolean that should be set to true when any of the following will be useful to the application: hand over a message to reliably transfer (possibly multiple times) before connection establishment; suggest timeout to the peer; notification of excessive retransmissions (early warning below abortion threshold); notification of ICMP error message arrival.

```
(!UDP) CONFIGURE_TIMEOUT (flow-group-id [timeout] [peer_timeout]  
[retrans_notify])
```

This configures timeouts for all flows in a group. Configuration should generally be carried out as early as possible, ideally before flows are connected, to aid the TAPS system's decision taking.

PARAMETERS:

timeout: a timeout value for aborting connections, in seconds
peer_timeout: a timeout value to be suggested to the peer (if possible), in seconds
retrans_notify: the number of retransmissions after which the application should be notified of "Excessive Retransmissions"

CONFIGURE_CHECKSUM (flow-id [send [send_length]] [receive [receive_length]])

This configures the usage of checksums for a flow in a group. Configuration should generally be carried out as early as possible, ideally before the flow is connected, to aid the TAPS system's decision taking. "send" parameters concern using a checksum when sending, "receive" parameters concern requiring a checksum when receiving. There is no guarantee that any checksum limitations will indeed be enforced; all defaults are: "full coverage, checksum enabled".

PARAMETERS:

send: boolean, enable / disable usage of a checksum
send_length: if send is true, this optional parameter can provide the desired coverage of the checksum in bytes
receive: boolean, enable / disable requiring a checksum
receive_length: if receive is true, this optional parameter can provide the required minimum coverage of the checksum in bytes

CONFIGURE_URGENCY (flow-group-id [scheduler] [capacity_profile] [low_watermark])

This carries out configuration related to the urgency of sending data on flows of a group. Configuration should generally be carried out as early as possible, ideally before flows are connected, to aid the TAPS system's decision taking.

PARAMETERS:

scheduler: a number to identify the type of scheduler that should be used to operate between flows in the group (no guarantees given). Future versions of this document will be self contained, but for now we suggest the schedulers defined in [I-D.ietf-tsvwg-sctp-ndata].
capacity_profile: a number to identify how an application wants to use its available capacity. Future versions of this document will

be self contained, but for now choices can be "lowest possible latency at the expense of overhead" (which would disable any Nagle-like algorithm), "scavenger", and some more values that help determine the DSCP value for a flow (e.g. similar to table 1 in [I-D.ietf-tsvwg-rtweb-qos]).

`low_watermark`: a buffer limit (in bytes); when the sender has less than `low_watermark` bytes in the buffer, the application may be notified. Notifications are not guaranteed, and supporting watermark numbers greater than 0 is not guaranteed.

CONFIGURE_PRIORITY (flow-id priority)

This configures a flow's priority or weight for a scheduler. Configuration should generally be carried out as early as possible, ideally before flows are connected, to aid the TAPS system's decision taking.

PARAMETERS:

`priority`: future versions of this document will be self contained, but for now we suggest the priority as described in [I-D.ietf-tsvwg-sctp-ndata].

NOTIFICATIONS

Returns: flow-group-id notification_type

This is fired when an event occurs, notifying the application about something happening in relation to a flow group. Notification types are:

`Excessive Retransmissions`: the configured (or a default) number of retransmissions has been reached, yielding this early warning below an abortion threshold.

`ICMP Arrival` (parameter: ICMP message): an ICMP packet carrying the conveyed ICMP message has arrived.

`ECN Arrival` (parameter: ECN value): a packet carrying the conveyed ECN value has arrived. This can be useful for applications implementing congestion control.

`Timeout` (parameter: s seconds): data could not be delivered for s seconds.

`Close`: the peer has closed the connection. The peer has no more data to send, and will not read more data. Data that is in transit or resides in the local send buffer will be discarded.

`Abort`: the peer has aborted the connection. The peer has no more data to send, and will not read more data. Data that is in transit or resides in the local send buffer will be discarded.

Note that there is no guarantee that this notification will be invoked when the peer aborts.

Drain: the send buffer has either drained below the configured low water mark or it has become completely empty.

Path Change (parameter: path identifier): the path has changed; the path identifier is a number that can be used to determine a previously used path is used again (e.g., the TAPS system has switched from one interface to the other and back).

Send Failure (parameter: frame identifier): this informs the application of a failure to send a specific frame. There can be a send failure without this notification happening.

QUERY_PROPERTIES (flow-group-id property_identifier)
Returns: requested property (see below)

This allows to query some properties of a flow group. Return values per property identifier are:

- o The maximum frame size that may be sent without fragmentation, in bytes (or "not available")
- o The maximum transport frame size that can be sent, in bytes (or "not available")
- o The maximum transport frame size that can be received, in bytes (or "not available")
- o The maximum amount of data that can possibly be sent before or during connection establishment, in bytes (or "not available")

CONNECT (flow-id dst_addr)

Connects a flow. This primitive may or may not trigger a notification (continuing LISTEN) on the listening side. If a send precedes this call, then data may be transmitted with this connect.

PARAMETERS:

dst_addr: the destination transport address to connect to

LISTEN (flow-id)

Blocking passive connect, listening on all interfaces. This may not be the direct result of the peer calling CONNECT - it may also be invoked upon reception of the first block of data. In this case, RECEIVE_FRAME is invoked immediately after.

SEND_FRAME (flow-id frame [reliability] [ordered] [bundle] [delack]
[fragment] [idempotent])

Sends an application frame. No guarantees are given about the preservation of frame boundaries to the peer; if frame boundaries are needed, the receiving application at the peer must know about them beforehand (or the TAPS system cannot fall back to TCP). Note that this call can already be used before a flow is connected. All parameters refer to the frame that is being handed over.

PARAMETERS:

(!UDP) reliability: this parameter is used to convey a choice of: fully reliable, unreliable without congestion control (which is guaranteed), unreliable, partially reliable (how to configure: TBD, probably using a time value). The latter two choices are not guaranteed and may result in full reliability.

(!UDP) ordered: this boolean parameter lets an application choose between ordered message delivery (true) and possibly unordered, potentially faster message delivery (false).

bundle: a boolean that expresses a preference for allowing to bundle frames (true) or not (false). No guarantees are given.

delack: a boolean that, if false, lets an application request that the peer would not delay the acknowledgement for this frame.

fragment: a boolean that expresses a preference for allowing to fragment frames (true) or not (false), at the IP level. No guarantees are given.

(!UDP) idempotent: a boolean that expresses whether a frame is idempotent (true) or not (false). Idempotent frames may arrive multiple times at the receiver (but they will arrive at least once). When data is idempotent it can be used by the receiver immediately on a connection establishment attempt. Thus, if SEND_FRAME is used before connecting, stating that a frame is idempotent facilitates transmitting it to the peer application particularly early.

(!UDP) CLOSE (flow-id)

Closes the flow after all outstanding data is reliably delivered to the peer (if reliable data delivery was requested). In case reliable or partially reliable data delivery was requested earlier, the peer is notified of the CLOSE.

ABORT (flow-id)

Aborts the flow without delivering outstanding data to the peer. In case reliable or partially reliable data delivery was requested earlier (!UDP), the peer is notified of the ABORT.

RECEIVE_FRAME (flow-id buffer)

This receives a block of data. This block may or may not correspond to a sender-side frame, i.e. the receiving application is not informed about frame boundaries (this limitation is only needed for TAPS systems that want to be able to fall back to TCP). However, if the sending application has allowed that frames are not fully reliably transferred, or delivered out of order, then such re-ordering or unreliability may be reflected per frame in the arriving data. Frames will always stay intact - i.e. if an incomplete frame is contained at the end of the arriving data block, this frame is guaranteed to continue in the next arriving data block.

PARAMETERS:

buffer: the buffer where the received data will be stored.

5. Conclusion

By decoupling applications from transport protocols, a TAPS system provides a different abstraction level than the Berkeley sockets interface. As with high- vs. low-level programming languages, a higher abstraction level allows more freedom for automation below the interface, yet it takes some control away from the application programmer. This is the design trade-off that a TAPS system developer is facing, and this document provides guidance on the design of this abstraction level. Some transport features are currently rarely offered by APIs, yet they must be offered or they can never be used ("functional" transport features). Other transport features are offered by the APIs of the protocols covered here, but not exposing them in a TAPS API would allow for more freedom to automate protocol usage in a TAPS system.

The minimal set presented in this document is an effort to find a middle ground that can be recommended for TAPS systems to implement, on the basis of the transport features discussed in [TAPS2]. This middle ground eliminates a large number of transport features because they do not require application-specific knowledge, but instead rely on knowledge about the network or the Operating System. This leaves us with an unanswered question about how exactly a TAPS system should automate using all of these "automatable" transport features.

In some cases, it may be best to not entirely automate the decision making, but leave it up to a system-wide policy. For example, when multiple paths are available, a system policy could guide the decision on whether to connect via a WiFi or a cellular interface. Such high-level guidance could also be provided by application developers, e.g. via a primitive that lets applications specify such preferences. As long as this kind of information from applications is treated as advisory, it will not lead to a permanent protocol binding and does therefore not limit the flexibility of a TAPS system. Decisions to add such primitives are therefore left open to TAPS system designers.

6. Acknowledgements

The authors would like to thank the participants of the TAPS Working Group and the NEAT research project for valuable input to this document. We especially thank Michael Tuexen for help with TAPS flow connection establishment/teardown and Gorrry Fairhurst for his suggestions regarding fragmentation and packet sizes. This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 644334 (NEAT).

7. IANA Considerations

XX RFC ED - PLEASE REMOVE THIS SECTION XXX

This memo includes no request to IANA.

8. Security Considerations

Authentication, confidentiality protection, and integrity protection are identified as transport features by [RFC8095]. As currently deployed in the Internet, these features are generally provided by a protocol or layer on top of the transport protocol; no current full-featured standards-track transport protocol provides all of these transport features on its own. Therefore, these transport features are not considered in this document, with the exception of native authentication capabilities of TCP and SCTP for which the security considerations in [RFC5925] and [RFC4895] apply.

9. References

9.1. Normative References

- [RFC8095] Fairhurst, G., Ed., Trammell, B., Ed., and M. Kuehlewind, Ed., "Services Provided by IETF Transport Protocols and Congestion Control Mechanisms", RFC 8095, DOI 10.17487/RFC8095, March 2017, <<https://www.rfc-editor.org/info/rfc8095>>.
- [TAPS2] Welzl, M., Tuexen, M., and N. Khademi, "On the Usage of Transport Features Provided by IETF Transport Protocols", Internet-draft draft-ietf-taps-transports-usage-08, August 2017.
- [TAPS2UDP] Fairhurst, G. and T. Jones, "Features of the User Datagram Protocol (UDP) and Lightweight UDP (UDP-Lite) Transport Protocols", Internet-draft draft-ietf-taps-transports-usage-udp-07, September 2017.

9.2. Informative References

- [COBS] Cheshire, S. and M. Baker, "Consistent Overhead Byte Stuffing", September 1997, <<http://stuartcheshire.org/papers/COBSforToN.pdf>>.
- [I-D.grinnemo-taps-he] Grinnemo, K., Brunstrom, A., Hurtig, P., Khademi, N., and Z. Bozakov, "Happy Eyeballs for Transport Selection", draft-grinnemo-taps-he-03 (work in progress), July 2017.
- [I-D.ietf-tsvwg-rtcweb-qos] Jones, P., Dhesikan, S., Jennings, C., and D. Druta, "DSCP Packet Markings for WebRTC QoS", draft-ietf-tsvwg-rtcweb-qos-18 (work in progress), August 2016.
- [I-D.ietf-tsvwg-sctp-ndata] Stewart, R., Tuexen, M., Loreto, S., and R. Seggelmann, "Stream Schedulers and User Message Interleaving for the Stream Control Transmission Protocol", draft-ietf-tsvwg-sctp-ndata-13 (work in progress), September 2017.
- [I-D.pauly-taps-transport-security] Pauly, T. and C. Wood, "A Survey of Transport Security Protocols", draft-pauly-taps-transport-security-00 (work in progress), July 2017.

- [I-D.trammell-taps-post-sockets]
Trammell, B., Perkins, C., Pauly, T., Kuehlewind, M., and C. Wood, "Post Sockets, An Abstract Programming Interface for the Transport Layer", draft-trammell-taps-post-sockets-01 (work in progress), September 2017.
- [LBE-draft]
Bless, R., "A Lower Effort Per-Hop Behavior (LE PHB)", Internet-draft draft-tsvwg-le-phb-02, June 2017.
- [RFC2914] Floyd, S., "Congestion Control Principles", BCP 41, RFC 2914, DOI 10.17487/RFC2914, September 2000, <<https://www.rfc-editor.org/info/rfc2914>>.
- [RFC4895] Tuexen, M., Stewart, R., Lei, P., and E. Rescorla, "Authenticated Chunks for the Stream Control Transmission Protocol (SCTP)", RFC 4895, DOI 10.17487/RFC4895, August 2007, <<https://www.rfc-editor.org/info/rfc4895>>.
- [RFC4987] Eddy, W., "TCP SYN Flooding Attacks and Common Mitigations", RFC 4987, DOI 10.17487/RFC4987, August 2007, <<https://www.rfc-editor.org/info/rfc4987>>.
- [RFC5925] Touch, J., Mankin, A., and R. Bonica, "The TCP Authentication Option", RFC 5925, DOI 10.17487/RFC5925, June 2010, <<https://www.rfc-editor.org/info/rfc5925>>.
- [RFC6458] 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>>.
- [RFC6525] Stewart, R., Tuexen, M., and P. Lei, "Stream Control Transmission Protocol (SCTP) Stream Reconfiguration", RFC 6525, DOI 10.17487/RFC6525, February 2012, <<https://www.rfc-editor.org/info/rfc6525>>.
- [RFC7413] Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP Fast Open", RFC 7413, DOI 10.17487/RFC7413, December 2014, <<https://www.rfc-editor.org/info/rfc7413>>.
- [WWDC2015]
Lakhera, P. and S. Cheshire, "Your App and Next Generation Networks", Apple Worldwide Developers Conference 2015, San Francisco, USA, June 2015, <<https://developer.apple.com/videos/wwdc/2015/?id=719>>.

Appendix A. Deriving the minimal set

We approach the construction of a minimal set of transport features in the following way:

1. Categorization: the superset of transport features from [TAPS2] is presented, and transport features are categorized for later reduction.
2. Reduction: a shorter list of transport features is derived from the categorization in the first step. This removes all transport features that do not require application-specific knowledge or cannot be implemented with TCP. !!!TODO discuss UDP
3. Discussion: the resulting list shows a number of peculiarities that are discussed, to provide a basis for constructing the minimal set.
4. Construction: Based on the reduced set and the discussion of the transport features therein, a minimal set is constructed.

The first three steps as well as the underlying rationale for constructing the minimal set are described in this appendix. The minimal set itself is described in Section 3.

A.1. Step 1: Categorization -- The Superset of Transport Features

Following [TAPS2], we divide the transport features into two main groups as follows:

1. CONNECTION related transport features
 - ESTABLISHMENT
 - AVAILABILITY
 - MAINTENANCE
 - TERMINATION
2. DATA Transfer Related transport features
 - Sending Data
 - Receiving Data
 - Errors

We assume that TAPS applications have no specific requirements that need knowledge about the network, e.g. regarding the choice of network interface or the end-to-end path. Even with these assumptions, there are certain requirements that are strictly kept by transport protocols today, and these must also be kept by a TAPS system. Some of these requirements relate to transport features that we call "Functional".

Functional transport features provide functionality that cannot be used without the application knowing about them, or else they violate assumptions that might cause the application to fail. For example, ordered message delivery is a functional transport feature: it cannot be configured without the application knowing about it because the application's assumption could be that messages always arrive in order. Failure includes any change of the application behavior that is not performance oriented, e.g. security.

"Change DSCP" and "Disable Nagle algorithm" are examples of transport features that we call "Optimizing": if a TAPS system autonomously decides to enable or disable them, an application will not fail, but a TAPS system may be able to communicate more efficiently if the application is in control of this optimizing transport feature. These transport features require application-specific knowledge (e.g., about delay/bandwidth requirements or the length of future data blocks that are to be transmitted).

The transport features of IETF transport protocols that do not require application-specific knowledge and could therefore be transparently utilized by a TAPS system are called "Automatable".

Finally, some transport features are aggregated and/or slightly changed in the description below. These transport features are marked as "ADDED". The corresponding transport features are automatable, and they are listed immediately below the "ADDED" transport feature.

In this description, transport services are presented following the nomenclature "CATEGORY.[SUBCATEGORY].SERVICENAME.PROTOCOL", equivalent to "pass 2" in [TAPS2]. We also sketch how some of the TAPS transport features can be implemented by a TAPS system. For all transport features that are categorized as "functional" or "optimizing", and for which no matching TCP and/or UDP primitive exists in "pass 2" of [TAPS2], a brief discussion on how to fall back to TCP and/or UDP is included.

We designate some transport features as "automatable" on the basis of a broader decision that affects multiple transport features:

- o Most transport features that are related to multi-streaming were designated as "automatable". This was done because the decision on whether to use multi-streaming or not does not depend on application-specific knowledge. This means that a connection that is exhibited to an application could be implemented by using a single stream of an SCTP association instead of mapping it to a complete SCTP association or TCP connection. This could be achieved by using more than one stream when an SCTP association is

first established (CONNECT.SCTP parameter "outbound stream count"), maintaining an internal stream number, and using this stream number when sending data (SEND.SCTP parameter "stream number"). Closing or aborting a connection could then simply free the stream number for future use. This is discussed further in Appendix A.3.2.

- o All transport features that are related to using multiple paths or the choice of the network interface were designated as "automatable". Choosing a path or an interface does not depend on application-specific knowledge. For example, "Listen" could always listen on all available interfaces and "Connect" could use the default interface for the destination IP address.

A.1.1. CONNECTION Related Transport Features

ESTABLISHMENT:

- o Connect
Protocols: TCP, SCTP, UDP(-Lite)
Functional because the notion of a connection is often reflected in applications as an expectation to be able to communicate after a "Connect" succeeded, with a communication sequence relating to this transport feature that is defined by the application protocol.
Implementation: via CONNECT.TCP, CONNECT.SCTP or CONNECT.UDP(-Lite).
- o Specify which IP Options must always be used
Protocols: TCP, UDP(-Lite)
Automatable because IP Options relate to knowledge about the network, not the application.
- o Request multiple streams
Protocols: SCTP
Automatable because using multi-streaming does not require application-specific knowledge.
Implementation: see Appendix A.3.2.
- o Limit the number of inbound streams
Protocols: SCTP
Automatable because using multi-streaming does not require application-specific knowledge.
Implementation: see Appendix A.3.2.

- o Specify number of attempts and/or timeout for the first establishment message
Protocols: TCP, SCTP
Functional because this is closely related to potentially assumed reliable data delivery for data that is sent before or during connection establishment.
Implementation: Using a parameter of CONNECT.TCP and CONNECT.SCTP.
Fall-back to UDP: Do nothing (this is irrelevant in case of UDP because there, reliable data delivery is not assumed).
- o Obtain multiple sockets
Protocols: SCTP
Automatable because the usage of multiple paths to communicate to the same end host relates to knowledge about the network, not the application.
- o Disable MPTCP
Protocols: MPTCP
Automatable because the usage of multiple paths to communicate to the same end host relates to knowledge about the network, not the application.
Implementation: via a boolean parameter in CONNECT.MPTCP.
- o Configure authentication
Protocols: TCP, SCTP
Functional because this has a direct influence on security.
Implementation: via parameters in CONNECT.TCP and CONNECT.SCTP.
Fall-back to TCP: With TCP, this allows to configure Master Key Tuples (MKTs) to authenticate complete segments (including the TCP IPv4 pseudoheader, TCP header, and TCP data). With SCTP, this allows to specify which chunk types must always be authenticated. Authenticating only certain chunk types creates a reduced level of security that is not supported by TCP; to be compatible, this should therefore only allow to authenticate all chunk types. Key material must be provided in a way that is compatible with both [RFC4895] and [RFC5925].
Fall-back to UDP: Not possible.
- o Indicate (and/or obtain upon completion) an Adaptation Layer via an adaptation code point
Protocols: SCTP

Functional because it allows to send extra data for the sake of identifying an adaptation layer, which by itself is application-specific.

Implementation: via a parameter in CONNECT.SCTP.

Fall-back to TCP: not possible.

Fall-back to UDP: not possible.

- o Request to negotiate interleaving of user messages
Protocols: SCTP
Automatable because it requires using multiple streams, but requesting multiple streams in the CONNECTION.ESTABLISHMENT category is automatable.
Implementation: via a parameter in CONNECT.SCTP.
- o Hand over a message to reliably transfer (possibly multiple times) before connection establishment
Protocols: TCP
Functional because this is closely tied to properties of the data that an application sends or expects to receive.
Implementation: via a parameter in CONNECT.TCP.
Fall-back to UDP: not possible.
- o Hand over a message to reliably transfer during connection establishment
Protocols: SCTP
Functional because this can only work if the message is limited in size, making it closely tied to properties of the data that an application sends or expects to receive.
Implementation: via a parameter in CONNECT.SCTP.
Fall-back to UDP: not possible.
- o Enable UDP encapsulation with a specified remote UDP port number
Protocols: SCTP
Automatable because UDP encapsulation relates to knowledge about the network, not the application.

AVAILABILITY:

- o Listen

Protocols: TCP, SCTP, UDP(-Lite)

Functional because the notion of accepting connection requests is often reflected in applications as an expectation to be able to communicate after a "Listen" succeeded, with a communication sequence relating to this transport feature that is defined by the application protocol.

ADDED. This differs from the 3 automatable transport features below in that it leaves the choice of interfaces for listening open.

Implementation: by listening on all interfaces via LISTEN.TCP (not providing a local IP address) or LISTEN.SCTP (providing SCTP port number / address pairs for all local IP addresses). LISTEN.UDP(-Lite) supports both methods.

- o Listen, 1 specified local interface
Protocols: TCP, SCTP, UDP(-Lite)
Automatable because decisions about local interfaces relate to knowledge about the network and the Operating System, not the application.
- o Listen, N specified local interfaces
Protocols: SCTP
Automatable because decisions about local interfaces relate to knowledge about the network and the Operating System, not the application.
- o Listen, all local interfaces
Protocols: TCP, SCTP, UDP(-Lite)
Automatable because decisions about local interfaces relate to knowledge about the network and the Operating System, not the application.
- o Specify which IP Options must always be used
Protocols: TCP, UDP(-Lite)
Automatable because IP Options relate to knowledge about the network, not the application.
- o Disable MPTCP
Protocols: MPTCP

Automatable because the usage of multiple paths to communicate to the same end host relates to knowledge about the network, not the application.

- o Configure authentication
Protocols: TCP, SCTP
Functional because this has a direct influence on security.
Implementation: via parameters in LISTEN.TCP and LISTEN.SCTP.
Fall-back to TCP: With TCP, this allows to configure Master Key Tuples (MKTs) to authenticate complete segments (including the TCP IPv4 pseudoheader, TCP header, and TCP data). With SCTP, this allows to specify which chunk types must always be authenticated. Authenticating only certain chunk types creates a reduced level of security that is not supported by TCP; to be compatible, this should therefore only allow to authenticate all chunk types. Key material must be provided in a way that is compatible with both [RFC4895] and [RFC5925].
Fall-back to UDP: not possible.
- o Obtain requested number of streams
Protocols: SCTP
Automatable because using multi-streaming does not require application-specific knowledge.
Implementation: see Appendix A.3.2.
- o Limit the number of inbound streams
Protocols: SCTP
Automatable because using multi-streaming does not require application-specific knowledge.
Implementation: see Appendix A.3.2.
- o Indicate (and/or obtain upon completion) an Adaptation Layer via an adaptation code point
Protocols: SCTP
Functional because it allows to send extra data for the sake of identifying an adaptation layer, which by itself is application-specific.
Implementation: via a parameter in LISTEN.SCTP.
Fall-back to TCP: not possible.
Fall-back to UDP: not possible.
- o Request to negotiate interleaving of user messages

Protocols: SCTP

Automatable because it requires using multiple streams, but requesting multiple streams in the CONNECTION.ESTABLISHMENT category is automatable.

Implementation: via a parameter in LISTEN.SCTP.

MAINTENANCE:

- o Change timeout for aborting connection (using retransmit limit or time value)

Protocols: TCP, SCTP

Functional because this is closely related to potentially assumed reliable data delivery.

Implementation: via CHANGE-TIMEOUT.TCP or CHANGE-TIMEOUT.SCTP.

Fall-back to UDP: not possible (UDP is unreliable and there is no connection timeout).

- o Suggest timeout to the peer

Protocols: TCP

Functional because this is closely related to potentially assumed reliable data delivery.

Implementation: via CHANGE-TIMEOUT.TCP.

Fall-back to UDP: not possible (UDP is unreliable and there is no connection timeout).

- o Disable Nagle algorithm

Protocols: TCP, SCTP

Optimizing because this decision depends on knowledge about the size of future data blocks and the delay between them.

Implementation: via DISABLE-NAGLE.TCP and DISABLE-NAGLE.SCTP.

Fall-back to UDP: do nothing (UDP does not implement the Nagle algorithm).

- o Request an immediate heartbeat, returning success/failure

Protocols: SCTP

Automatable because this informs about network-specific knowledge.

- o Notification of Excessive Retransmissions (early warning below abortion threshold)
Protocols: TCP
Optimizing because it is an early warning to the application, informing it of an impending functional event.
Implementation: via ERROR.TCP.
Fall-back to UDP: do nothing (there is no abortion threshold).

- o Add path
Protocols: MPTCP, SCTP
MPTCP Parameters: source-IP; source-Port; destination-IP; destination-Port
SCTP Parameters: local IP address
Automatable because the usage of multiple paths to communicate to the same end host relates to knowledge about the network, not the application.

- o Remove path
Protocols: MPTCP, SCTP
MPTCP Parameters: source-IP; source-Port; destination-IP; destination-Port
SCTP Parameters: local IP address
Automatable because the usage of multiple paths to communicate to the same end host relates to knowledge about the network, not the application.

- o Set primary path
Protocols: SCTP
Automatable because the usage of multiple paths to communicate to the same end host relates to knowledge about the network, not the application.

- o Suggest primary path to the peer
Protocols: SCTP
Automatable because the usage of multiple paths to communicate to the same end host relates to knowledge about the network, not the application.

- o Configure Path Switchover
Protocols: SCTP
Automatable because the usage of multiple paths to communicate to the same end host relates to knowledge about the network, not the application.

- o Obtain status (query or notification)
Protocols: SCTP, MPTCP
SCTP parameters: association connection state; destination transport address list; destination transport address reachability states; current local and peer receiver window size; current local congestion window sizes; number of unacknowledged DATA chunks; number of DATA chunks pending receipt; primary path; most recent SRTT on primary path; RTO on primary path; SRTT and RTO on other destination addresses; MTU per path; interleaving supported yes/no
MPTCP parameters: subflow-list (identified by source-IP; source-Port; destination-IP; destination-Port)
Automatable because these parameters relate to knowledge about the network, not the application.

- o Specify DSCP field
Protocols: TCP, SCTP, UDP(-Lite)
Optimizing because choosing a suitable DSCP value requires application-specific knowledge.
Implementation: via SET_DSCP.TCP / SET_DSCP.SCTP / SET_DSCP.UDP(-Lite)

- o Notification of ICMP error message arrival
Protocols: TCP, UDP(-Lite)
Optimizing because these messages can inform about success or failure of functional transport features (e.g., host unreachable relates to "Connect")
Implementation: via ERROR.TCP or ERROR.UDP(-Lite).

- o Obtain information about interleaving support
Protocols: SCTP
Automatable because it requires using multiple streams, but requesting multiple streams in the CONNECTION.ESTABLISHMENT category is automatable.
Implementation: via a parameter in GETINTERL.SCTP.

- o Change authentication parameters
Protocols: TCP, SCTP
Functional because this has a direct influence on security.
Implementation: via SET_AUTH.TCP and SET_AUTH.SCTP.
Fall-back to TCP: With SCTP, this allows to adjust key_id, key, and hmac_id. With TCP, this allows to change the preferred outgoing MKT (current_key) and the preferred incoming MKT (rnext_key), respectively, for a segment that is sent on the connection. Key material must be provided in a way that is compatible with both [RFC4895] and [RFC5925].
Fall-back to UDP: not possible.

- o Obtain authentication information
Protocols: SCTP
Functional because authentication decisions may have been made by the peer, and this has an influence on the necessary application-level measures to provide a certain level of security.
Implementation: via GETAUTH.SCTP.
Fall-back to TCP: With SCTP, this allows to obtain key_id and a chunk list. With TCP, this allows to obtain current_key and rnext_key from a previously received segment. Key material must be provided in a way that is compatible with both [RFC4895] and [RFC5925].
Fall-back to UDP: not possible.

- o Reset Stream
Protocols: SCTP
Automatable because using multi-streaming does not require application-specific knowledge.
Implementation: see Appendix A.3.2.

- o Notification of Stream Reset
Protocols: STCP
Automatable because using multi-streaming does not require application-specific knowledge.
Implementation: see Appendix A.3.2.

- o Reset Association
Protocols: SCTP
Automatable because deciding to reset an association does not require application-specific knowledge.
Implementation: via RESETASSOC.SCTP.

- o Notification of Association Reset
Protocols: STCP
Automatable because this notification does not relate to application-specific knowledge.
- o Add Streams
Protocols: SCTP
Automatable because using multi-streaming does not require application-specific knowledge.
Implementation: see Appendix A.3.2.
- o Notification of Added Stream
Protocols: STCP
Automatable because using multi-streaming does not require application-specific knowledge.
Implementation: see Appendix A.3.2.
- o Choose a scheduler to operate between streams of an association
Protocols: SCTP
Optimizing because the scheduling decision requires application-specific knowledge. However, if a TAPS system would not use this, or wrongly configure it on its own, this would only affect the performance of data transfers; the outcome would still be correct within the "best effort" service model.
Implementation: using SETSTREAMSCHEDULER.SCTP.
Fall-back to TCP: do nothing.
Fall-back to UDP: do nothing.
- o Configure priority or weight for a scheduler
Protocols: SCTP
Optimizing because the priority or weight requires application-specific knowledge. However, if a TAPS system would not use this, or wrongly configure it on its own, this would only affect the performance of data transfers; the outcome would still be correct within the "best effort" service model.
Implementation: using CONFIGURESTREAMSCHEDULER.SCTP.
Fall-back to TCP: do nothing.
Fall-back to UDP: do nothing.
- o Configure send buffer size

Protocols: SCTP

Automatable because this decision relates to knowledge about the network and the Operating System, not the application (see also the discussion in Appendix A.3.4).

- o Configure receive buffer (and rwnd) size
Protocols: SCTP
Automatable because this decision relates to knowledge about the network and the Operating System, not the application.
- o Configure message fragmentation
Protocols: SCTP
Automatable because fragmentation relates to knowledge about the network and the Operating System, not the application.
Implementation: by always enabling it with CONFIG_FRAGMENTATION.SCTP and auto-setting the fragmentation size based on network or Operating System conditions.
- o Configure PMTUD
Protocols: SCTP
Automatable because Path MTU Discovery relates to knowledge about the network, not the application.
- o Configure delayed SACK timer
Protocols: SCTP
Automatable because the receiver-side decision to delay sending SACKs relates to knowledge about the network, not the application (it can be relevant for a sending application to request not to delay the SACK of a message, but this is a different transport feature).
- o Set Cookie life value
Protocols: SCTP
Functional because it relates to security (possibly weakened by keeping a cookie very long) versus the time between connection establishment attempts. Knowledge about both issues can be application-specific.

Fall-back to TCP: the closest specified TCP functionality is the cookie in TCP Fast Open; for this, [RFC7413] states that the server "can expire the cookie at any time to enhance security" and section 4.1.2 describes an example implementation where updating the key on the server side causes the cookie to expire. Alternatively, for implementations that do not support TCP Fast Open, this transport feature could also affect the validity of SYN cookies (see Section 3.6 of [RFC4987]).
Fall-back to UDP: do nothing.

- o Set maximum burst
Protocols: SCTP
Automatable because it relates to knowledge about the network, not the application.
- o Configure size where messages are broken up for partial delivery
Protocols: SCTP
Functional because this is closely tied to properties of the data that an application sends or expects to receive.
Fall-back to TCP: not possible.
Fall-back to UDP: not possible.
- o Disable checksum when sending
Protocols: UDP
Functional because application-specific knowledge is necessary to decide whether it can be acceptable to lose data integrity.
Implementation: via SET_CHECKSUM_ENABLED.UDP.
Fall-back to TCP: do nothing.
- o Disable checksum requirement when receiving
Protocols: UDP
Functional because application-specific knowledge is necessary to decide whether it can be acceptable to lose data integrity.
Implementation: via SET_CHECKSUM_REQUIRED.UDP.
Fall-back to TCP: do nothing.
- o Specify checksum coverage used by the sender
Protocols: UDP-Lite

Functional because application-specific knowledge is necessary to decide for which parts of the data it can be acceptable to lose data integrity.

Implementation: via SET_CHECKSUM_COVERAGE.UDP-Lite.

Fall-back to TCP: do nothing.

- o Specify minimum checksum coverage required by receiver
Protocols: UDP-Lite
Functional because application-specific knowledge is necessary to decide for which parts of the data it can be acceptable to lose data integrity.
Implementation: via SET_MIN_CHECKSUM_COVERAGE.UDP-Lite.
Fall-back to TCP: do nothing.
- o Specify DF field
Protocols: UDP(-Lite)
Optimizing because the DF field can be used to carry out Path MTU Discovery, which can lead an application to choose message sizes that can be transmitted more efficiently.
Implementation: via MAINTENANCE.SET_DF.UDP(-Lite) and SEND_FAILURE.UDP(-Lite).
Fall-back to TCP: do nothing. With TCP the sender is not in control of transport message sizes, making this functionality irrelevant.
- o Get max. transport-message size that may be sent using a non-fragmented IP packet from the configured interface
Protocols: UDP(-Lite)
Optimizing because this can lead an application to choose message sizes that can be transmitted more efficiently.
Fall-back to TCP: do nothing: this information is not available with TCP.
- o Get max. transport-message size that may be received from the configured interface
Protocols: UDP(-Lite)
Optimizing because this can, for example, influence an application's memory management.
Fall-back to TCP: do nothing: this information is not available with TCP.

- o Specify TTL/Hop count field
Protocols: UDP(-Lite)
Automatable because a TAPS system can use a large enough system default to avoid communication failures. Allowing an application to configure it differently can produce notifications of ICMP error message arrivals that yield information which only relates to knowledge about the network, not the application.
- o Obtain TTL/Hop count field
Protocols: UDP(-Lite)
Automatable because the TTL/Hop count field relates to knowledge about the network, not the application.
- o Specify ECN field
Protocols: UDP(-Lite)
Automatable because the ECN field relates to knowledge about the network, not the application.
- o Obtain ECN field
Protocols: UDP(-Lite)
Optimizing because this information can be used by an application to better carry out congestion control (this is relevant when choosing a data transmission transport service that does not already do congestion control).
Fall-back to TCP: do nothing: this information is not available with TCP.
- o Specify IP Options
Protocols: UDP(-Lite)
Automatable because IP Options relate to knowledge about the network, not the application.
- o Obtain IP Options
Protocols: UDP(-Lite)
Automatable because IP Options relate to knowledge about the network, not the application.

- o Enable and configure a "Low Extra Delay Background Transfer"
Protocols: A protocol implementing the LEDBAT congestion control mechanism
Optimizing because whether this service is appropriate or not depends on application-specific knowledge. However, wrongly using this will only affect the speed of data transfers (albeit including other transfers that may compete with the TAPS transfer in the network), so it is still correct within the "best effort" service model.
Implementation: via CONFIGURE.LEDBAT and/or SET_DSCP.TCP / SET_DSCP.SCTP / SET_DSCP.UDP(-Lite) [LBE-draft].
Fall-back to TCP: do nothing.
Fall-back to UDP: do nothing.

TERMINATION:

- o Close after reliably delivering all remaining data, causing an event informing the application on the other side
Protocols: TCP, SCTP
Functional because the notion of a connection is often reflected in applications as an expectation to have all outstanding data delivered and no longer be able to communicate after a "Close" succeeded, with a communication sequence relating to this transport feature that is defined by the application protocol.
Implementation: via CLOSE.TCP and CLOSE.SCTP.
Fall-back to UDP: not possible.
- o Abort without delivering remaining data, causing an event informing the application on the other side
Protocols: TCP, SCTP
Functional because the notion of a connection is often reflected in applications as an expectation to potentially not have all outstanding data delivered and no longer be able to communicate after an "Abort" succeeded. On both sides of a connection, an application protocol may define a communication sequence relating to this transport feature.
Implementation: via ABORT.TCP and ABORT.SCTP.
Fall-back to UDP: not possible.
- o Abort without delivering remaining data, not causing an event informing the application on the other side

Protocols: UDP(-Lite)

Functional because the notion of a connection is often reflected in applications as an expectation to potentially not have all outstanding data delivered and no longer be able to communicate after an "Abort" succeeded. On both sides of a connection, an application protocol may define a communication sequence relating to this transport feature.

Implementation: via ABORT.UDP(-Lite).

Fall-back to TCP: stop using the connection, wait for a timeout.

- o Timeout event when data could not be delivered for too long
Protocols: TCP, SCTP
Functional because this notifies that potentially assumed reliable data delivery is no longer provided.
Implementation: via TIMEOUT.TCP and TIMEOUT.SCTP.
Fall-back to UDP: do nothing: this event will not occur with UDP.

A.1.2. DATA Transfer Related Transport Features

A.1.2.1. Sending Data

- o Reliably transfer data, with congestion control
Protocols: TCP, SCTP
Functional because this is closely tied to properties of the data that an application sends or expects to receive.
Implementation: via SEND.TCP and SEND.SCTP.
Fall-back to UDP: not possible.
- o Reliably transfer a message, with congestion control
Protocols: SCTP
Functional because this is closely tied to properties of the data that an application sends or expects to receive.
Implementation: via SEND.SCTP.
Fall-back to TCP: via SEND.TCP. With SEND.TCP, messages will not be identifiable by the receiver.
Fall-back to UDP: not possible.
- o Unreliably transfer a message
Protocols: SCTP, UDP(-Lite)

Optimizing because only applications know about the time criticality of their communication, and reliably transferring a message is never incorrect for the receiver of a potentially unreliable data transfer, it is just slower.
ADDED. This differs from the 2 automatable transport features below in that it leaves the choice of congestion control open.
Implementation: via SEND.SCTP or SEND.UDP(-Lite).
Fall-back to TCP: use SEND.TCP. With SEND.TCP, messages will be sent reliably, and they will not be identifiable by the receiver.

- o Unreliably transfer a message, with congestion control
Protocols: SCTP
Automatable because congestion control relates to knowledge about the network, not the application.
- o Unreliably transfer a message, without congestion control
Protocols: UDP(-Lite)
Automatable because congestion control relates to knowledge about the network, not the application.
- o Configurable Message Reliability
Protocols: SCTP
Optimizing because only applications know about the time criticality of their communication, and reliably transferring a message is never incorrect for the receiver of a potentially unreliable data transfer, it is just slower.
Implementation: via SEND.SCTP.
Fall-back to TCP: By using SEND.TCP and ignoring this configuration: based on the assumption of the best-effort service model, unnecessarily delivering data does not violate application expectations. Moreover, it is not possible to associate the requested reliability to a "message" in TCP anyway.
Fall-back to UDP: not possible.
- o Choice of stream
Protocols: SCTP
Automatable because it requires using multiple streams, but requesting multiple streams in the CONNECTION.ESTABLISHMENT category is automatable. Implementation: see Appendix A.3.2.

- o Choice of path (destination address)
Protocols: SCTP
Automatable because it requires using multiple sockets, but obtaining multiple sockets in the CONNECTION.ESTABLISHMENT category is automatable.
- o Ordered message delivery (potentially slower than unordered)
Protocols: SCTP
Functional because this is closely tied to properties of the data that an application sends or expects to receive.
Implementation: via SEND.SCTP.
Fall-back to TCP: By using SEND.TCP. With SEND.TCP, messages will not be identifiable by the receiver.
Fall-back to UDP: not possible.
- o Unordered message delivery (potentially faster than ordered)
Protocols: SCTP, UDP(-Lite)
Functional because this is closely tied to properties of the data that an application sends or expects to receive.
Implementation: via SEND.SCTP.
Fall-back to TCP: By using SEND.TCP and always sending data ordered: based on the assumption of the best-effort service model, ordered delivery may just be slower and does not violate application expectations. Moreover, it is not possible to associate the requested delivery order to a "message" in TCP anyway.
- o Request not to bundle messages
Protocols: SCTP
Optimizing because this decision depends on knowledge about the size of future data blocks and the delay between them.
Implementation: via SEND.SCTP.
Fall-back to TCP: By using SEND.TCP and DISABLE-NAGLE.TCP to disable the Nagle algorithm when the request is made and enable it again when the request is no longer made. Note that this is not fully equivalent because it relates to the time of issuing the request rather than a specific message.
Fall-back to UDP: do nothing (UDP never bundles messages).

- o Specifying a "payload protocol-id" (handed over as such by the receiver)
Protocols: SCTP
Functional because it allows to send extra application data with every message, for the sake of identification of data, which by itself is application-specific.
Implementation: SEND.SCTP.
Fall-back to TCP: not possible.
Fall-back to UDP: not possible.

- o Specifying a key id to be used to authenticate a message
Protocols: SCTP
Functional because this has a direct influence on security.
Implementation: via a parameter in SEND.SCTP.
Fall-back to TCP: This could be emulated by using SET_AUTH.TCP before and after the message is sent. Note that this is not fully equivalent because it relates to the time of issuing the request rather than a specific message.
Fall-back to UDP: not possible.

- o Request not to delay the acknowledgement (SACK) of a message
Protocols: SCTP
Optimizing because only an application knows for which message it wants to quickly be informed about success / failure of its delivery.
Fall-back to TCP: do nothing.
Fall-back to UDP: do nothing.

A.1.2.2. Receiving Data

- o Receive data (with no message delimiting)
Protocols: TCP
Functional because a TAPS system must be able to send and receive data.
Implementation: via RECEIVE.TCP.
Fall-back to UDP: do nothing (hand over a message, let the application ignore frame boundaries).

- o Receive a message

Protocols: SCTP, UDP(-Lite)
Functional because this is closely tied to properties of the data that an application sends or expects to receive.
Implementation: via RECEIVE.SCTP and RECEIVE.UDP(-Lite).
Fall-back to TCP: not possible.

- o Choice of stream to receive from
Protocols: SCTP
Automatable because it requires using multiple streams, but requesting multiple streams in the CONNECTION.ESTABLISHMENT category is automatable.
Implementation: see Appendix A.3.2.
- o Information about partial message arrival
Protocols: SCTP
Functional because this is closely tied to properties of the data that an application sends or expects to receive.
Implementation: via RECEIVE.SCTP.
Fall-back to TCP: do nothing: this information is not available with TCP.
Fall-back to UDP: do nothing: this information is not available with UDP.

A.1.2.3. Errors

This section describes sending failures that are associated with a specific call to in the "Sending Data" category (Appendix A.1.2.1).

- o Notification of send failures
Protocols: SCTP, UDP(-Lite)
Functional because this notifies that potentially assumed reliable data delivery is no longer provided.
ADDED. This differs from the 2 automatable transport features below in that it does not distinguish between unsent and unacknowledged messages.
Implementation: via SENDFAILURE-EVENT.SCTP and SEND_FAILURE.UDP(-Lite).
Fall-back to TCP: do nothing: this notification is not available and will therefore not occur with TCP.

- o Notification of an unsent (part of a) message
Protocols: SCTP, UDP(-Lite)
Automatable because the distinction between unsent and unacknowledged is network-specific.
- o Notification of an unacknowledged (part of a) message
Protocols: SCTP
Automatable because the distinction between unsent and unacknowledged is network-specific.
- o Notification that the stack has no more user data to send
Protocols: SCTP
Optimizing because reacting to this notification requires the application to be involved, and ensuring that the stack does not run dry of data (for too long) can improve performance.
Fall-back to TCP: do nothing. See also the discussion in Appendix A.3.4.
Fall-back to UDP: do nothing. This notification is not available and will therefore not occur with UDP.
- o Notification to a receiver that a partial message delivery has been aborted
Protocols: SCTP
Functional because this is closely tied to properties of the data that an application sends or expects to receive.
Fall-back to TCP: do nothing. This notification is not available and will therefore not occur with TCP.
Fall-back to UDP: do nothing. This notification is not available and will therefore not occur with UDP.

A.2. Step 2: Reduction -- The Reduced Set of Transport Features

By hiding automatable transport features from the application, a TAPS system can gain opportunities to automate the usage of network-related functionality. This can facilitate using the TAPS system for the application programmer and it allows for optimizations that may not be possible for an application. For instance, system-wide configurations regarding the usage of multiple interfaces can better be exploited if the choice of the interface is not entirely up to the

application. Therefore, since they are not strictly necessary to expose in a TAPS system, we do not include automatable transport features in the reduced set of transport features. This leaves us with only the transport features that are either optimizing or functional.

A TAPS system should be able to fall back to TCP or UDP if alternative transport protocols are found not to work. For many transport features, this is possible -- often by simply not doing anything. For some transport features, however, it was identified that neither a fall-back to TCP nor a fall-back to UDP is possible: in these cases, even not doing anything would incur semantically incorrect behavior. Whenever an application would make use of one of these transport features, this would eliminate the possibility to use TCP or UDP. Thus, we only keep the functional and optimizing transport features for which a fall-back to either TCP or UDP is possible in our reduced set.

In the following list, we precede a transport feature with "T:" if a fall-back to TCP is possible, "U:" if a fall-back to UDP is possible, and "TU:" if a fall-back to either TCP or UDP is possible.

A.2.1. CONNECTION Related Transport Features

ESTABLISHMENT:

- o T,U: Connect
- o T,U: Specify number of attempts and/or timeout for the first establishment message
- o T: Configure authentication
- o T: Hand over a message to reliably transfer (possibly multiple times) before connection establishment
- o T: Hand over a message to reliably transfer during connection establishment

AVAILABILITY:

- o T,U: Listen
- o T: Configure authentication

MAINTENANCE:

- o T: Change timeout for aborting connection (using retransmit limit or time value)
- o T: Suggest timeout to the peer
- o T,U: Disable Nagle algorithm
- o T,U: Notification of Excessive Retransmissions (early warning below abortion threshold)

- o T,U: Specify DSCP field
- o T,U: Notification of ICMP error message arrival
- o T: Change authentication parameters
- o T: Obtain authentication information
- o T,U: Set Cookie life value
- o T,U: Choose a scheduler to operate between streams of an association
- o T,U: Configure priority or weight for a scheduler
- o T,U: Disable checksum when sending
- o T,U: Disable checksum requirement when receiving
- o T,U: Specify checksum coverage used by the sender
- o T,U: Specify minimum checksum coverage required by receiver
- o T,U: Specify DF field
- o T,U: Get max. transport-message size that may be sent using a non-fragmented IP packet from the configured interface
- o T,U: Get max. transport-message size that may be received from the configured interface
- o T,U: Obtain ECN field
- o T,U: Enable and configure a "Low Extra Delay Background Transfer"

TERMINATION:

- o T: Close after reliably delivering all remaining data, causing an event informing the application on the other side
- o T: Abort without delivering remaining data, causing an event informing the application on the other side
- o T,U: Abort without delivering remaining data, not causing an event informing the application on the other side
- o T,U: Timeout event when data could not be delivered for too long

A.2.2. DATA Transfer Related Transport Features

A.2.2.1. Sending Data

- o T: Reliably transfer data, with congestion control
- o T: Reliably transfer a message, with congestion control
- o T,U: Unreliably transfer a message
- o T: Configurable Message Reliability
- o T: Ordered message delivery (potentially slower than unordered)
- o T,U: Unordered message delivery (potentially faster than ordered)
- o T,U: Request not to bundle messages
- o T: Specifying a key id to be used to authenticate a message
- o T,U: Request not to delay the acknowledgement (SACK) of a message

A.2.2.2. Receiving Data

- o T,U: Receive data (with no message delimiting)
- o U: Receive a message
- o T,U: Information about partial message arrival

A.2.2.3. Errors

This section describes sending failures that are associated with a specific call to in the "Sending Data" category (Appendix A.1.2.1).

- o T,U: Notification of send failures
- o T,U: Notification that the stack has no more user data to send
- o T,U: Notification to a receiver that a partial message delivery has been aborted

A.3. Step 3: Discussion

The reduced set in the previous section exhibits a number of peculiarities, which we will discuss in the following. This section focuses on TCP because, with the exception of one particular transport feature ("Receive a message" -- we will discuss this in Appendix A.3.1), the list shows that UDP is strictly a subset of TCP. We can first try to understand how to build a TAPS system that is able to fall back to TCP, and then narrow down the result further to allow that the system can always fall back to either TCP or UDP (which effectively means removing everything related to reliability, ordering, authentication and closing/aborting with a notification to the peer).

Note that, because the functional transport features of UDP are -- with the exception of "Receive a message" -- a subset of TCP, TCP can be used as a fall-back for UDP whenever an application does not need message delimiting (e.g., because the application-layer protocol already does it). This has been recognized by many applications that already do this in practice, by trying to communicate with UDP at first, and falling back to TCP in case of a connection failure.

A.3.1. Sending Messages, Receiving Bytes

When considering to fall back to TCP, there are several transport features related to sending, but only a single transport feature related to receiving: "Receive data (with no message delimiting)" (and, strangely, "information about partial message arrival"). Notably, the transport feature "Receive a message" is also the only non-automatable transport feature of UDP(-Lite) for which no fall-back to TCP is possible.

To support these TCP receiver semantics, we define an "Application-Framed Bytestream" (AFra-Bytestream). AFra-Bytestreams allow senders to operate on messages while minimizing changes to the TCP socket API. In particular, nothing changes on the receiver side - data can be accepted via a normal TCP socket.

In an AFra-Bytestream, the sending application can optionally inform the transport about frame boundaries and required properties per frame (configurable order and reliability, or embedding a request not to delay the acknowledgement of a frame). Whenever the sending application specifies per-frame properties that relax the notion of reliable in-order delivery of bytes, it must assume that the receiving application is 1) able to determine frame boundaries, provided that frames are always kept intact, and 2) able to accept these relaxed per-frame properties. Any signaling of such information to the peer is up to an application-layer protocol and considered out of scope of this document.

For example, if an application requests to transfer fixed-size messages of 100 bytes with partial reliability, this needs the receiving application to be prepared to accept data in chunks of 100 bytes. If, then, some of these 100-byte messages are missing (e.g., if SCTP with Configurable Reliability is used), this is the expected application behavior. With TCP, no messages would be missing, but this is also correct for the application, and the possible retransmission delay is acceptable within the best effort service model. Still, the receiving application would separate the byte stream into 100-byte chunks.

Note that this usage of messages does not require all messages to be equal in size. Many application protocols use some form of Type-Length-Value (TLV) encoding, e.g. by defining a header including length fields; another alternative is the use of byte stuffing methods such as COBS [COBS]. If an application needs message numbers, e.g. to restore the correct sequence of messages, these must also be encoded by the application itself, as the sequence number related transport features of SCTP are no longer provided (in the interest of enabling a fall-back to TCP).

For the implementation of a TAPS system, this has the following consequences:

- o Because the receiver-side transport leaves it up to the application to delimit messages, messages must always remain intact as they are handed over by the transport receiver. Data can be handed over at any time as they arrive, but the byte stream must never "skip ahead" to the beginning of the next message.

- o With SCTP, a "partial flag" informs a receiving application that a message is incomplete. Then, the next receive calls will only deliver remaining parts of the same message (i.e., no messages or partial messages will arrive on other streams until the message is complete) (see Section 8.1.20 in [RFC6458]). This can facilitate the implementation of the receiver buffer in the receiving application, but then such an application does not support message interleaving (which is required by stream schedulers). However, receiving a byte stream from multiple SCTP streams requires a per-stream receiver buffer anyway, so this potential benefit is lost and the "partial flag" (the transport feature "Information about partial message arrival") becomes unnecessary for a TAPS system. With it, the transport feature "Notification to a receiver that a partial message delivery has been aborted" becomes unnecessary too.
- o From the above, a TAPS system should always support message interleaving because it enables the use of stream schedulers and comes at no additional implementation cost on the receiver side. Stream schedulers operate on the sender side. Hence, because a TAPS sender-side application may talk to an SCTP receiver that does not support interleaving, it cannot assume that stream schedulers will always work as expected.

A.3.2. Stream Schedulers Without Streams

We have already stated that multi-streaming does not require application-specific knowledge. Potential benefits or disadvantages of, e.g., using two streams over an SCTP association versus using two separate SCTP associations or TCP connections are related to knowledge about the network and the particular transport protocol in use, not the application. However, the transport features "Choose a scheduler to operate between streams of an association" and "Configure priority or weight for a scheduler" operate on streams. Here, streams identify communication channels between which a scheduler operates, and they can be assigned a priority. Moreover, the transport features in the MAINTENANCE category all operate on associations in case of SCTP, i.e. they apply to all streams in that association.

With only these semantics necessary to represent, the interface to a TAPS system becomes easier if we rename connections into "TAPS flows" (the TAPS equivalent of a connection which may be a transport connection or association, but could also become a stream of an existing SCTP association, for example) and allow assigning a "Group Number" to a TAPS flow. Then, all MAINTENANCE transport features can be said to operate on flow groups, not connections, and a scheduler also operates on the flows within a group.

For the implementation of a TAPS system, this has the following consequences:

- o Streams may be identified in different ways across different protocols. The only multi-streaming protocol considered in this document, SCTP, uses a stream id. The transport association below still uses a Transport Address (which includes one port number) for each communicating endpoint. To implement a TAPS system without exposed streams, an application must be given an identifier for each TAPS flow (akin to a socket), and depending on whether streams are used or not, there will be a 1:1 mapping between this identifier and local ports or not.
- o In SCTP, a fixed number of streams exists from the beginning of an association; streams are not "established", there is no handshake or any other form of signaling to create them: they can just be used. They are also not "gracefully shut down" -- at best, an "SSN Reset Request Parameter" in a "RE-CONFIG" chunk [RFC6525] can be used to inform the peer that of a "Stream Reset", as a rough equivalent of an "Abort". This has an impact on the semantics connection establishment and teardown (see Section 3.2).
- o To support stream schedulers, a receiver-side TAPS system should always support message interleaving because it comes at no additional implementation cost (because of the receiver-side stream reception discussed in Appendix A.3.1). Note, however, that Stream schedulers operate on the sender side. Hence, because a TAPS sender-side application may talk to a native TCP-based receiver-side application, it cannot assume that stream schedulers will always work as expected.

A.3.3. Early Data Transmission

There are two transport features related to transferring a message early: "Hand over a message to reliably transfer (possibly multiple times) before connection establishment", which relates to TCP Fast Open [RFC7413], and "Hand over a message to reliably transfer during connection establishment", which relates to SCTP's ability to transfer data together with the COOKIE-Echo chunk. Also without TCP Fast Open, TCP can transfer data during the handshake, together with the SYN packet -- however, the receiver of this data may not hand it over to the application until the handshake has completed. Also, different from TCP Fast Open, this data is not delimited as a message by TCP (thus, not visible as a ``message``). This functionality is commonly available in TCP and supported in several implementations, even though the TCP specification does not explain how to provide it to applications.

A TAPS system could differentiate between the cases of transmitting data "before" (possibly multiple times) or during the handshake.

Alternatively, it could also assume that data that are handed over early will be transmitted as early as possible, and "before" the handshake would only be used for data that are explicitly marked as "idempotent" (i.e., it would be acceptable to transfer it multiple times).

The amount of data that can successfully be transmitted before or during the handshake depends on various factors: the transport protocol, the use of header options, the choice of IPv4 and IPv6 and the Path MTU. A TAPS system should therefore allow a sending application to query the maximum amount of data it can possibly transmit before (or, if exposed, during) connection establishment.

A.3.4. Sender Running Dry

The transport feature "Notification that the stack has no more user data to send" relates to SCTP's "SENDER DRY" notification. Such notifications can, in principle, be used to avoid having an unnecessarily large send buffer, yet ensure that the transport sender always has data available when it has an opportunity to transmit it. This has been found to be very beneficial for some applications [WWDC2015]. However, "SENDER DRY" truly means that the entire send buffer (including both unsent and unacknowledged data) has emptied -- i.e., when it notifies the sender, it is already too late, the transport protocol already missed an opportunity to send data. Some modern TCP implementations now include the unspecified "TCP_NOTSENT_LOWAT" socket option proposed in [WWDC2015], which limits the amount of unsent data that TCP can keep in the socket buffer; this allows to specify at which buffer filling level the socket becomes writable, rather than waiting for the buffer to run empty.

SCTP allows to configure the sender-side buffer too: the automatable Transport Feature "Configure send buffer size" provides this functionality, but only for the complete buffer, which includes both unsent and unacknowledged data. SCTP does not allow to control these two sizes separately. A TAPS system should allow for uniform access to "TCP_NOTSENT_LOWAT" as well as the "SENDER DRY" notification.

A.3.5. Capacity Profile

The transport features:

- o Disable Nagle algorithm
- o Enable and configure a "Low Extra Delay Background Transfer"
- o Specify DSCP field

all relate to a QoS-like application need such as "low latency" or "scavenger". In the interest of flexibility of a TAPS system, they could therefore be offered in a uniform, more abstract way, where a TAPS system could e.g. decide by itself how to use combinations of LEDBAT-like congestion control and certain DSCP values, and an application would only specify a general "capacity profile" (a description of how it wants to use the available capacity). A need for "lowest possible latency at the expense of overhead" could then translate into automatically disabling the Nagle algorithm.

In some cases, the Nagle algorithm is best controlled directly by the application because it is not only related to a general profile but also to knowledge about the size of future messages. For fine-grain control over Nagle-like functionality, the "Request not to bundle messages" is available.

A.3.6. Security

Both TCP and SCTP offer authentication. TCP authenticates complete segments. SCTP allows to configure which of SCTP's chunk types must always be authenticated -- if this is exposed as such, it creates an undesirable dependency on the transport protocol. For compatibility with TCP, a TAPS system should only allow to configure complete transport layer packets, including headers, IP pseudo-header (if any) and payload.

Security is discussed in a separate TAPS document [I-D.pauly-taps-transport-security]. The minimal set presented in the present document therefore excludes all security related transport features: "Configure authentication", "Change authentication parameters", "Obtain authentication information" and "Set Cookie life value" as well as "Specifying a key id to be used to authenticate a message".

A.3.7. Packet Size

UDP(-Lite) has a transport feature called "Specify DF field". This yields an error message in case of sending a message that exceeds the Path MTU, which is necessary for a UDP-based application to be able to implement Path MTU Discovery (a function that UDP-based applications must do by themselves). The "Get max. transport-message size that may be sent using a non-fragmented IP packet from the configured interface" transport feature yields an upper limit for the Path MTU (minus headers) and can therefore help to implement Path MTU Discovery more efficiently.

This also relates to the fact that the choice of path is automatable: if a TAPS system can switch a path at any time, unknown to an

application, yet the application intends to do Path MTU Discovery, this could yield a very inefficient behavior. Thus, a TAPS system should probably avoid automatically switching paths, and inform the application about any unavoidable path changes, when applications request to disallow fragmentation with the "Specify DF field" feature.

Appendix B. Revision information

XXX RFC-Ed please remove this section prior to publication.

-02: implementation suggestions added, discussion section added, terminology extended, DELETED category removed, various other fixes; list of Transport Features adjusted to -01 version of [TAPS2] except that MPTCP is not included.

-03: updated to be consistent with -02 version of [TAPS2].

-04: updated to be consistent with -03 version of [TAPS2].
Reorganized document, rewrote intro and conclusion, and made a first stab at creating a real "minimal set".

-05: updated to be consistent with -05 version of [TAPS2] (minor changes). Fixed a mistake regarding Cookie Life value. Exclusion of security related transport features (to be covered in a separate document). Reorganized the document (now begins with the minset, derivation is in the appendix). First stab at an abstract API for the minset.

draft-ietf-taps-minset-00: updated to be consistent with -08 version of [TAPS2] ("obtain message delivery number" was removed, as this has also been removed in [TAPS2] because it was a mistake in RFC4960. This led to the removal of two more transport features that were only designated as functional because they affected "obtain message delivery number"). Fall-back to UDP incorporated (this was requested at IETF-99); this also affected the transport feature "Choice between unordered (potentially faster) or ordered delivery of messages" because this is a boolean which is always true for one fall-back protocol, and always false for the other one. This was therefore now divided into two features, one for ordered, one for unordered delivery. The word "reliably" was added to the transport features "Hand over a message to reliably transfer (possibly multiple times) before connection establishment" and "Hand over a message to reliably transfer during connection establishment" to make it clearer why this is not supported by UDP. Clarified that the "minset abstract interface" is not proposing a specific API for all TAPS systems to implement, but it is just a way to describe the minimum set. Author order changed.

Authors' Addresses

Michael Welzl
University of Oslo
PO Box 1080 Blindern
Oslo N-0316
Norway

Phone: +47 22 85 24 20
Email: michawe@ifi.uio.no

Stein Gjessing
University of Oslo
PO Box 1080 Blindern
Oslo N-0316
Norway

Phone: +47 22 85 24 44
Email: steing@ifi.uio.no

TAPS
Internet-Draft
Intended status: Informational
Expires: March 31, 2019

M. Welzl
S. Gjessing
University of Oslo
September 27, 2018

A Minimal Set of Transport Services for End Systems
draft-ietf-taps-minset-11

Abstract

This draft recommends a minimal set of Transport Services offered by end systems, and gives guidance on choosing among the available mechanisms and protocols. It is based on the set of transport features in RFC 8303.

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 March 31, 2019.

Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
2. Terminology	4
3. Deriving the minimal set	5
4. The Reduced Set of Transport Features	6
4.1. CONNECTION Related Transport Features	7
4.2. DATA Transfer Related Transport Features	8
4.2.1. Sending Data	8
4.2.2. Receiving Data	9
4.2.3. Errors	9
5. Discussion	9
5.1. Sending Messages, Receiving Bytes	10
5.2. Stream Schedulers Without Streams	11
5.3. Early Data Transmission	11
5.4. Sender Running Dry	12
5.5. Capacity Profile	13
5.6. Security	13
5.7. Packet Size	14
6. The Minimal Set of Transport Features	14
6.1. ESTABLISHMENT, AVAILABILITY and TERMINATION	14
6.2. MAINTENANCE	18
6.2.1. Connection groups	18
6.2.2. Individual connections	20
6.3. DATA Transfer	20
6.3.1. Sending Data	20
6.3.2. Receiving Data	21
7. Acknowledgements	22
8. IANA Considerations	22
9. Security Considerations	22
10. References	22
10.1. Normative References	22
10.2. Informative References	23
Appendix A. The Superset of Transport Features	25
A.1. CONNECTION Related Transport Features	26
A.2. DATA Transfer Related Transport Features	42
A.2.1. Sending Data	42
A.2.2. Receiving Data	46
A.2.3. Errors	47
Appendix B. Revision information	48
Authors' Addresses	50

1. Introduction

Currently, the set of transport services that most applications use is based on TCP and UDP (and protocols that are layered on top of them); this limits the ability for the network stack to make use of features of other transport protocols. For example, if a protocol

supports out-of-order message delivery but applications always assume that the network provides an ordered bytestream, then the network stack can not immediately deliver a message that arrives out-of-order: doing so would break a fundamental assumption of the application. The net result is unnecessary head-of-line blocking delay.

By exposing the transport services of multiple transport protocols, a transport system can make it possible for applications to use these services without being statically bound to a specific transport protocol. The first step towards the design of such a system was taken by [RFC8095], which surveys a large number of transports, and [RFC8303] as well as [RFC8304], which identify the specific transport features that are exposed to applications by the protocols TCP, MPTCP, UDP(-Lite) and SCTP as well as the LEDBAT congestion control mechanism. LEDBAT was included as the only congestion control mechanism in this list because the "low extra delay background transport" service that it offers is significantly different from the typical service provided by other congestion control mechanisms. This memo is based on these documents and follows the same terminology (also listed below). Because the considered transport protocols conjointly cover a wide range of transport features, there is reason to hope that the resulting set (and the reasoning that led to it) will also apply to many aspects of other transport protocols that may be in use today, or may be designed in the future.

By decoupling applications from transport protocols, a transport system provides a different abstraction level than the Berkeley sockets interface [POSIX]. As with high- vs. low-level programming languages, a higher abstraction level allows more freedom for automation below the interface, yet it takes some control away from the application programmer. This is the design trade-off that a transport system developer is facing, and this document provides guidance on the design of this abstraction level. Some transport features are currently rarely offered by APIs, yet they must be offered or they can never be used. Other transport features are offered by the APIs of the protocols covered here, but not exposing them in an API would allow for more freedom to automate protocol usage in a transport system. The minimal set presented here is an effort to find a middle ground that can be recommended for transport systems to implement, on the basis of the transport features discussed in [RFC8303].

Applications use a wide variety of APIs today. While this document was created to ensure the API developed in the Transport Services (TAPS) Working Group ([I-D.ietf-taps-interface]) includes the most important transport features, the minimal set presented here must be reflected in *all* network APIs in order for the underlying

functionality to become usable everywhere. For example, it does not help an application that talks to a library which offers its own communication interface if the underlying Berkeley Sockets API is extended to offer "unordered message delivery", but the library only exposes an ordered bytestream. Both the Berkeley Sockets API and the library would have to expose the "unordered message delivery" transport feature (alternatively, there may be ways for certain types of libraries to use this transport feature without exposing it, based on knowledge about the applications -- but this is not the general case). Similarly, transport protocols such as SCTP offer multi-streaming, which cannot be utilized, e.g., to prioritize messages between streams, unless applications communicate the priorities and the group of connections upon which these priorities should be applied. In most situations, in the interest of being as flexible and efficient as possible, the best choice will be for a library to expose at least all of the transport features that are recommended as a "minimal set" here.

This "minimal set" can be implemented "one-sided" over TCP. This means that a sender-side transport system can talk to a standard TCP receiver, and a receiver-side transport system can talk to a standard TCP sender. If certain limitations are put in place, the "minimal set" can also be implemented "one-sided" over UDP. While the possibility of such "one-sided" implementation may help deployment, it comes at the cost of limiting the set to services that can also be provided by TCP (or, with further limitations, UDP). Thus, the minimal set of transport features here is applicable for many, but not all, applications: some application protocols have requirements that are not met by this "minimal set".

Note that, throughout this document, protocols are meant to be used natively. For example, when transport features of UDP, or "implementation over" UDP is discussed, this refers to native usage of UDP.

2. Terminology

Transport Feature: a specific end-to-end feature that the transport layer provides to an application. Examples include confidentiality, reliable delivery, ordered delivery, message-versus-stream orientation, etc.

Transport Service: a set of Transport Features, without an association to any given framing protocol, which provides a complete service to an application.

Transport Protocol: an implementation that provides one or more different transport services using a specific framing and header format on the wire.

Application: an entity that uses a transport layer interface for end-to-end delivery of data across the network (this may also be an upper layer protocol or tunnel encapsulation).

Application-specific knowledge: knowledge that only applications have.

End system: an entity that communicates with one or more other end systems using a transport protocol. An end system provides a transport layer interface to applications.

Connection: shared state of two or more end systems that persists across messages that are transmitted between these end systems.

Connection Group: a set of connections which share the same configuration (configuring one of them causes all other connections in the same group to be configured in the same way). We call connections that belong to a connection group "grouped", while "ungrouped" connections are not a part of a connection group.

Socket: the combination of a destination IP address and a destination port number.

Moreover, throughout the document, the protocol name "UDP(-Lite)" is used when discussing transport features that are equivalent for UDP and UDP-Lite; similarly, the protocol name "TCP" refers to both TCP and MPTCP.

3. Deriving the minimal set

We assume that applications have no specific requirements that need knowledge about the network, e.g. regarding the choice of network interface or the end-to-end path. Even with these assumptions, there are certain requirements that are strictly kept by transport protocols today, and these must also be kept by a transport system. Some of these requirements relate to transport features that we call "Functional".

Functional transport features provide functionality that cannot be used without the application knowing about them, or else they violate assumptions that might cause the application to fail. For example, ordered message delivery is a functional transport feature: it cannot be configured without the application knowing about it because the application's assumption could be that messages always arrive in order. Failure includes any change of the application behavior that is not performance oriented, e.g. security.

"Change DSCP" and "Disable Nagle algorithm" are examples of transport features that we call "Optimizing": if a transport system autonomously decides to enable or disable them, an application will not fail, but a transport system may be able to communicate more efficiently if the application is in control of this optimizing

transport feature. These transport features require application-specific knowledge (e.g., about delay/bandwidth requirements or the length of future data blocks that are to be transmitted).

The transport features of IETF transport protocols that do not require application-specific knowledge and could therefore be utilized by a transport system on its own without involving the application are called "Automatable".

We approach the construction of a minimal set of transport features in the following way:

1. Categorization (Appendix A): the superset of transport features from [RFC8303] is presented, and transport features are categorized as Functional, Optimizing or Automatable for later reduction.
2. Reduction (Section 4): a shorter list of transport features is derived from the categorization in the first step. This removes all transport features that do not require application-specific knowledge or would result in semantically incorrect behavior if they were implemented over TCP or UDP.
3. Discussion (Section 5): the resulting list shows a number of peculiarities that are discussed, to provide a basis for constructing the minimal set.
4. Construction (Section 6): Based on the reduced set and the discussion of the transport features therein, a minimal set is constructed.

Following [RFC8303] and retaining its terminology, we divide the transport features into two main groups as follows:

1. CONNECTION related transport features
 - ESTABLISHMENT
 - AVAILABILITY
 - MAINTENANCE
 - TERMINATION
2. DATA Transfer related transport features
 - Sending Data
 - Receiving Data
 - Errors

4. The Reduced Set of Transport Features

By hiding automatable transport features from the application, a transport system can gain opportunities to automate the usage of network-related functionality. This can facilitate using the

transport system for the application programmer and it allows for optimizations that may not be possible for an application. For instance, system-wide configurations regarding the usage of multiple interfaces can better be exploited if the choice of the interface is not entirely up to the application. Therefore, since they are not strictly necessary to expose in a transport system, we do not include automatable transport features in the reduced set of transport features. This leaves us with only the transport features that are either optimizing or functional.

A transport system should be able to communicate via TCP or UDP if alternative transport protocols are found not to work. For many transport features, this is possible -- often by simply not doing anything when a specific request is made. For some transport features, however, it was identified that direct usage of neither TCP nor UDP is possible: in these cases, even not doing anything would incur semantically incorrect behavior. Whenever an application would make use of one of these transport features, this would eliminate the possibility to use TCP or UDP. Thus, we only keep the functional and optimizing transport features for which an implementation over either TCP or UDP is possible in our reduced set.

The following list contains the transport features from Appendix A, reduced using these rules. The "minimal set" derived in this document is meant to be implementable "one-sided" over TCP, and, with limitations, UDP. In the list, we therefore precede a transport feature with "T:" if an implementation over TCP is possible, "U:" if an implementation over UDP is possible, and "T,U:" if an implementation over either TCP or UDP is possible.

4.1. CONNECTION Related Transport Features

ESTABLISHMENT:

- o T,U: Connect
- o T,U: Specify number of attempts and/or timeout for the first establishment message
- o T,U: Disable MPTCP
- o T: Configure authentication
- o T: Hand over a message to reliably transfer (possibly multiple times) before connection establishment
- o T: Hand over a message to reliably transfer during connection establishment

AVAILABILITY:

- o T,U: Listen
- o T,U: Disable MPTCP

- o T: Configure authentication

MAINTENANCE:

- o T: Change timeout for aborting connection (using retransmit limit or time value)
- o T: Suggest timeout to the peer
- o T,U: Disable Nagle algorithm
- o T,U: Notification of Excessive Retransmissions (early warning below abortion threshold)
- o T,U: Specify DSCP field
- o T,U: Notification of ICMP error message arrival
- o T: Change authentication parameters
- o T: Obtain authentication information
- o T,U: Set Cookie life value
- o T,U: Choose a scheduler to operate between streams of an association
- o T,U: Configure priority or weight for a scheduler
- o T,U: Disable checksum when sending
- o T,U: Disable checksum requirement when receiving
- o T,U: Specify checksum coverage used by the sender
- o T,U: Specify minimum checksum coverage required by receiver
- o T,U: Specify DF field
- o T,U: Get max. transport-message size that may be sent using a non-fragmented IP packet from the configured interface
- o T,U: Get max. transport-message size that may be received from the configured interface
- o T,U: Obtain ECN field
- o T,U: Enable and configure a "Low Extra Delay Background Transfer"

TERMINATION:

- o T: Close after reliably delivering all remaining data, causing an event informing the application on the other side
- o T: Abort without delivering remaining data, causing an event informing the application on the other side
- o T,U: Abort without delivering remaining data, not causing an event informing the application on the other side
- o T,U: Timeout event when data could not be delivered for too long

4.2. DATA Transfer Related Transport Features

4.2.1. Sending Data

- o T: Reliably transfer data, with congestion control
- o T: Reliably transfer a message, with congestion control
- o T,U: Unreliably transfer a message
- o T: Configurable Message Reliability

- o T: Ordered message delivery (potentially slower than unordered)
- o T,U: Unordered message delivery (potentially faster than ordered)
- o T,U: Request not to bundle messages
- o T: Specifying a key id to be used to authenticate a message
- o T,U: Request not to delay the acknowledgement (SACK) of a message

4.2.2. Receiving Data

- o T,U: Receive data (with no message delimiting)
- o U: Receive a message
- o T,U: Information about partial message arrival

4.2.3. Errors

This section describes sending failures that are associated with a specific call to in the "Sending Data" category (Appendix A.2.1).

- o T,U: Notification of send failures
- o T,U: Notification that the stack has no more user data to send
- o T,U: Notification to a receiver that a partial message delivery has been aborted

5. Discussion

The reduced set in the previous section exhibits a number of peculiarities, which we will discuss in the following. This section focuses on TCP because, with the exception of one particular transport feature ("Receive a message" -- we will discuss this in Section 5.1), the list shows that UDP is strictly a subset of TCP. We can first try to understand how to build a transport system that can run over TCP, and then narrow down the result further to allow that the system can always run over either TCP or UDP (which effectively means removing everything related to reliability, ordering, authentication and closing/aborting with a notification to the peer).

Note that, because the functional transport features of UDP are -- with the exception of "Receive a message" -- a subset of TCP, TCP can be used as a replacement for UDP whenever an application does not need message delimiting (e.g., because the application-layer protocol already does it). This has been recognized by many applications that already do this in practice, by trying to communicate with UDP at first, and falling back to TCP in case of a connection failure.

5.1. Sending Messages, Receiving Bytes

For implementing a transport system over TCP, there are several transport features related to sending, but only a single transport feature related to receiving: "Receive data (with no message delimiting)" (and, strangely, "information about partial message arrival"). Notably, the transport feature "Receive a message" is also the only non-automatable transport feature of UDP(-Lite) for which no implementation over TCP is possible.

To support these TCP receiver semantics, we define an "Application-Framed Bytestream" (AFra-Bytestream). AFra-Bytestreams allow senders to operate on messages while minimizing changes to the TCP socket API. In particular, nothing changes on the receiver side - data can be accepted via a normal TCP socket.

In an AFra-Bytestream, the sending application can optionally inform the transport about message boundaries and required properties per message (configurable order and reliability, or embedding a request not to delay the acknowledgement of a message). Whenever the sending application specifies per-message properties that relax the notion of reliable in-order delivery of bytes, it must assume that the receiving application is 1) able to determine message boundaries, provided that messages are always kept intact, and 2) able to accept these relaxed per-message properties. Any signaling of such information to the peer is up to an application-layer protocol and considered out of scope of this document.

For example, if an application requests to transfer fixed-size messages of 100 bytes with partial reliability, this needs the receiving application to be prepared to accept data in chunks of 100 bytes. If, then, some of these 100-byte messages are missing (e.g., if SCTP with Configurable Reliability is used), this is the expected application behavior. With TCP, no messages would be missing, but this is also correct for the application, and the possible retransmission delay is acceptable within the best-effort service model (see [RFC7305], Section 3.5). Still, the receiving application would separate the byte stream into 100-byte chunks.

Note that this usage of messages does not require all messages to be equal in size. Many application protocols use some form of Type-Length-Value (TLV) encoding, e.g. by defining a header including length fields; another alternative is the use of byte stuffing methods such as COBS [COBS]. If an application needs message numbers, e.g. to restore the correct sequence of messages, these must also be encoded by the application itself, as the sequence number related transport features of SCTP are not provided by the "minimum set" (in the interest of enabling usage of TCP).

5.2. Stream Schedulers Without Streams

We have already stated that multi-streaming does not require application-specific knowledge. Potential benefits or disadvantages of, e.g., using two streams of an SCTP association versus using two separate SCTP associations or TCP connections are related to knowledge about the network and the particular transport protocol in use, not the application. However, the transport features "Choose a scheduler to operate between streams of an association" and "Configure priority or weight for a scheduler" operate on streams. Here, streams identify communication channels between which a scheduler operates, and they can be assigned a priority. Moreover, the transport features in the MAINTENANCE category all operate on associations in case of SCTP, i.e. they apply to all streams in that association.

With only these semantics necessary to represent, the interface to a transport system becomes easier if we assume that connections may be not only a transport protocol's connection or association, but could also be a stream of an existing SCTP association, for example. We only need to allow for a way to define a possible grouping of connections. Then, all MAINTENANCE transport features can be said to operate on connection groups, not connections, and a scheduler operates on the connections within a group.

To be compatible with multiple transport protocols and uniformly allow access to both transport connections and streams of a multi-streaming protocol, the semantics of opening and closing need to be the most restrictive subset of all of the underlying options. For example, TCP's support of half-closed connections can be seen as a feature on top of the more restrictive "ABORT"; this feature cannot be supported because not all protocols used by a transport system (including streams of an association) support half-closed connections.

5.3. Early Data Transmission

There are two transport features related to transferring a message early: "Hand over a message to reliably transfer (possibly multiple times) before connection establishment", which relates to TCP Fast Open [RFC7413], and "Hand over a message to reliably transfer during connection establishment", which relates to SCTP's ability to transfer data together with the COOKIE-Echo chunk. Also without TCP Fast Open, TCP can transfer data during the handshake, together with the SYN packet -- however, the receiver of this data may not hand it over to the application until the handshake has completed. Also, different from TCP Fast Open, this data is not delimited as a message by TCP (thus, not visible as a ``message``). This functionality is

commonly available in TCP and supported in several implementations, even though the TCP specification does not explain how to provide it to applications.

A transport system could differentiate between the cases of transmitting data "before" (possibly multiple times) or "during" the handshake. Alternatively, it could also assume that data that are handed over early will be transmitted as early as possible, and "before" the handshake would only be used for messages that are explicitly marked as "idempotent" (i.e., it would be acceptable to transfer them multiple times).

The amount of data that can successfully be transmitted before or during the handshake depends on various factors: the transport protocol, the use of header options, the choice of IPv4 and IPv6 and the Path MTU. A transport system should therefore allow a sending application to query the maximum amount of data it can possibly transmit before (or, if exposed, during) connection establishment.

5.4. Sender Running Dry

The transport feature "Notification that the stack has no more user data to send" relates to SCTP's "SENDER DRY" notification. Such notifications can, in principle, be used to avoid having an unnecessarily large send buffer, yet ensure that the transport sender always has data available when it has an opportunity to transmit it. This has been found to be very beneficial for some applications [WWDC2015]. However, "SENDER DRY" truly means that the entire send buffer (including both unsent and unacknowledged data) has emptied -- i.e., when it notifies the sender, it is already too late, the transport protocol already missed an opportunity to send data. Some modern TCP implementations now include the unspecified "TCP_NOTSENT_LOWAT" socket option that was proposed in [WWDC2015], which limits the amount of unsent data that TCP can keep in the socket buffer; this allows to specify at which buffer filling level the socket becomes writable, rather than waiting for the buffer to run empty.

SCTP allows to configure the sender-side buffer too: the automatable Transport Feature "Configure send buffer size" provides this functionality, but only for the complete buffer, which includes both unsent and unacknowledged data. SCTP does not allow to control these two sizes separately. It therefore makes sense for a transport system to allow for uniform access to "TCP_NOTSENT_LOWAT" as well as the "SENDER DRY" notification.

5.5. Capacity Profile

The transport features:

- o Disable Nagle algorithm
- o Enable and configure a "Low Extra Delay Background Transfer"
- o Specify DSCP field

all relate to a QoS-like application need such as "low latency" or "scavenger". In the interest of flexibility of a transport system, they could therefore be offered in a uniform, more abstract way, where a transport system could e.g. decide by itself how to use combinations of LEDBAT-like congestion control and certain DSCP values, and an application would only specify a general "capacity profile" (a description of how it wants to use the available capacity). A need for "lowest possible latency at the expense of overhead" could then translate into automatically disabling the Nagle algorithm.

In some cases, the Nagle algorithm is best controlled directly by the application because it is not only related to a general profile but also to knowledge about the size of future messages. For fine-grain control over Nagle-like functionality, the "Request not to bundle messages" is available.

5.6. Security

Both TCP and SCTP offer authentication. TCP authenticates complete segments. SCTP allows to configure which of SCTP's chunk types must always be authenticated -- if this is exposed as such, it creates an undesirable dependency on the transport protocol. For compatibility with TCP, a transport system should only allow to configure complete transport layer packets, including headers, IP pseudo-header (if any) and payload.

Security is discussed in a separate document [I-D.ietf-taps-transport-security]. The minimal set presented in the present document excludes all security related transport features from Appendix A: "Configure authentication", "Change authentication parameters", "Obtain authentication information" and "Set Cookie life value" as well as "Specifying a key id to be used to authenticate a message". It also excludes security transport features not listed in Appendix A, including content privacy to in-path devices.

5.7. Packet Size

UDP(-Lite) has a transport feature called "Specify DF field". This yields an error message in case of sending a message that exceeds the Path MTU, which is necessary for a UDP-based application to be able to implement Path MTU Discovery (a function that UDP-based applications must do by themselves). The "Get max. transport-message size that may be sent using a non-fragmented IP packet from the configured interface" transport feature yields an upper limit for the Path MTU (minus headers) and can therefore help to implement Path MTU Discovery more efficiently.

6. The Minimal Set of Transport Features

Based on the categorization, reduction, and discussion in Section 3, this section describes a minimal set of transport features that end systems should offer. Any configuration based the described minimum set of transport feature can always be realized over TCP but also gives the transport system flexibility to choose another transport if implemented. In the text of this section, "not UDP" is used to indicate elements of the system that cannot be implemented over UDP. Conversely, all elements of the system that are not marked with "not UDP" can also be implemented over UDP.

The arguments laid out in Section 5 ("discussion") were used to make the final representation of the minimal set as short, simple and general as possible. There may be situations where these arguments do not apply -- e.g., implementers may have specific reasons to expose multi-streaming as a visible functionality to applications, or the restrictive open / close semantics may be problematic under some circumstances. In such cases, the representation in Section 4 ("reduction") should be considered.

As in Section 3, Section 4 and [RFC8303], we categorize the minimal set of transport features as 1) CONNECTION related (ESTABLISHMENT, AVAILABILITY, MAINTENANCE, TERMINATION) and 2) DATA Transfer related (Sending Data, Receiving Data, Errors). Here, the focus is on connections that the transport system offers as an abstraction to the application, as opposed to connections of transport protocols that the transport system uses.

6.1. ESTABLISHMENT, AVAILABILITY and TERMINATION

A connection must first be "created" to allow for some initial configuration to be carried out before the transport system can actively or passively establish communication with a remote end system. As a configuration of the newly created connection, an application can choose to disallow usage of MPTCP. Furthermore, all

configuration parameters in Section 6.2 can be used initially, although some of them may only take effect when a connection has been established with a chosen transport protocol. Configuring a connection early helps a transport system make the right decisions. For example, grouping information can influence the transport system to implement a connection as a stream of a multi-streaming protocol's existing association or not.

For ungrouped connections, early configuration is necessary because it allows the transport system to know which protocols it should try to use. In particular, a transport system that only makes a one-time choice for a particular protocol must know early about strict requirements that must be kept, or it can end up in a deadlock situation (e.g., having chosen UDP and later be asked to support reliable transfer). As an example description of how to correctly handle these cases, we provide the following decision tree (this is derived from Section 4.1 excluding authentication, as explained in Section 9):

- Will it ever be necessary to offer any of the following?

- * Reliably transfer data
- * Notify the peer of closing/aborting
- * Preserve data ordering

Yes: SCTP or TCP can be used.

- Is any of the following useful to the application?

- * Choosing a scheduler to operate between connections in a group, with the possibility to configure a priority or weight per connection
- * Configurable message reliability
- * Unordered message delivery
- * Request not to delay the acknowledgement (SACK) of a message

Yes: SCTP is preferred.

No:

- Is any of the following useful to the application?

- * Hand over a message to reliably transfer (possibly multiple times) before connection establishment
- * Suggest timeout to the peer
- * Notification of Excessive Retransmissions (early warning below abortion threshold)
- * Notification of ICMP error message arrival

Yes: TCP is preferred.

No: SCTP and TCP are equally preferable.

No: all protocols can be used.

- Is any of the following useful to the application?

- * Specify checksum coverage used by the sender
- * Specify minimum checksum coverage required by receiver

Yes: UDP-Lite is preferred.

No: UDP is preferred.

Note that this decision tree is not optimal for all cases. For example, if an application wants to use "Specify checksum coverage used by the sender", which is only offered by UDP-Lite, and "Configure priority or weight for a scheduler", which is only offered by SCTP, the above decision tree will always choose UDP-Lite, making it impossible to use SCTP's schedulers with priorities between grouped connections. Also, several other factors may influence the decisions for or against a protocol -- e.g. penetration rates, the ability to work through NATs, etc. We caution implementers to be aware of the full set of trade-offs, for which we recommend

consulting the list in Section 4.1 when deciding how to initialize a connection.

To summarize, the following parameters serve as input for the transport system to help it choose and configure a suitable protocol:

- o Reliability: a boolean that should be set to true when any of the following will be useful to the application: reliably transfer data; notify the peer of closing/aborting; preserve data ordering.
- o Checksum coverage: a boolean to specify whether it will be useful to the application to specify checksum coverage when sending or receiving.
- o Configure message priority: a boolean that should be set to true when any of the following per-message configuration or prioritization mechanisms will be useful to the application: choosing a scheduler to operate between grouped connections, with the possibility to configure a priority or weight per connection; configurable message reliability; unordered message delivery; requesting not to delay the acknowledgement (SACK) of a message.
- o Early message timeout notifications: a boolean that should be set to true when any of the following will be useful to the application: hand over a message to reliably transfer (possibly multiple times) before connection establishment; suggest timeout to the peer; notification of excessive retransmissions (early warning below abortion threshold); notification of ICMP error message arrival.

Once a connection is created, it can be queried for the maximum amount of data that an application can possibly expect to have reliably transmitted before or during transport connection establishment (with zero being a possible answer) (see Section 6.2.1). An application can also give the connection a message for reliable transmission before or during connection establishment (not UDP); the transport system will then try to transmit it as early as possible. An application can facilitate sending a message particularly early by marking it as "idempotent" (see Section 6.3.1); in this case, the receiving application must be prepared to potentially receive multiple copies of the message (because idempotent messages are reliably transferred, asking for idempotence is not necessary for systems that support UDP).

After creation, a transport system can actively establish communication with a peer, or it can passively listen for incoming connection requests. Note that active establishment may or may not trigger a notification on the listening side. It is possible that the first notification on the listening side is the arrival of the first data that the active side sends (a receiver-side transport system could handle this by continuing to block a "Listen" call,

immediately followed by issuing "Receive", for example; callback-based implementations could simply skip the equivalent of "Listen"). This also means that the active opening side is assumed to be the first side sending data.

A transport system can actively close a connection, i.e. terminate it after reliably delivering all remaining data to the peer (if reliable data delivery was requested earlier (not UDP)), in which case the peer is notified that the connection is closed. Alternatively, a connection can be aborted without delivering outstanding data to the peer. In case reliable or partially reliable data delivery was requested earlier (not UDP), the peer is notified that the connection is aborted. A timeout can be configured to abort a connection when data could not be delivered for too long (not UDP); however, timeout-based abortion does not notify the peer application that the connection has been aborted. Because half-closed connections are not supported, when a host implementing a transport system receives a notification that the peer is closing or aborting the connection (not UDP), its peer may not be able to read outstanding data. This means that unacknowledged data residing in a transport system's send buffer may have to be dropped from that buffer upon arrival of a "close" or "abort" notification from the peer.

6.2. MAINTENANCE

A transport system must offer means to group connections, but it cannot guarantee truly grouping them using the transport protocols that it uses (e.g., it cannot be guaranteed that connections become multiplexed as streams on a single SCTP association when SCTP may not be available). The transport system must therefore ensure that group- versus non-group-configurations are handled correctly in some way (e.g., by applying the configuration to all grouped connections even when they are not multiplexed, or informing the application about grouping success or failure).

As a general rule, any configuration described below should be carried out as early as possible to aid the transport system's decision making.

6.2.1. Connection groups

The following transport features and notifications (some directly from Section 4, some new or changed, based on the discussion in Section 5) automatically apply to all grouped connections:

(not UDP) Configure a timeout: this can be done with the following parameters:

- o A timeout value for aborting connections, in seconds
- o A timeout value to be suggested to the peer (if possible), in seconds
- o The number of retransmissions after which the application should be notified of "Excessive Retransmissions"

Configure urgency: this can be done with the following parameters:

- o A number to identify the type of scheduler that should be used to operate between connections in the group (no guarantees given). Schedulers are defined in [RFC8260].
- o A "capacity profile" number to identify how an application wants to use its available capacity. Choices can be "lowest possible latency at the expense of overhead" (which would disable any Nagle-like algorithm), "scavenger", or values that help determine the DSCP value for a connection (e.g. similar to table 1 in [I-D.ietf-tsvwg-rtcweb-qos]).
- o A buffer limit (in bytes); when the sender has less than the provided limit of bytes in the buffer, the application may be notified. Notifications are not guaranteed, and it is optional for a transport system to support buffer limit values greater than 0. Note that this limit and its notification should operate across the buffers of the whole transport system, i.e. also any potential buffers that the transport system itself may use on top of the transport's send buffer.

Following Section 5.7, these properties can be queried:

- o The maximum message size that may be sent without fragmentation via the configured interface. This is optional for a transport system to offer, and may return an error ("not available"). It can aid applications implementing Path MTU Discovery.
- o The maximum transport message size that can be sent, in bytes. Irrespective of fragmentation, there is a size limit for the messages that can be handed over to SCTP or UDP(-Lite); because the service provided by a transport system is independent of the transport protocol, it must allow an application to query this value -- the maximum size of a message in an Application-Framed-Bytestream (see Section 5.1). This may also return an error when data is not delimited ("not available").
- o The maximum transport message size that can be received from the configured interface, in bytes (or "not available").
- o The maximum amount of data that can possibly be sent before or during connection establishment, in bytes.

In addition to the already mentioned closing / aborting notifications and possible send errors, the following notifications can occur:

- o Excessive Retransmissions: the configured (or a default) number of retransmissions has been reached, yielding this early warning below an abortion threshold.
- o ICMP Arrival (parameter: ICMP message): an ICMP packet carrying the conveyed ICMP message has arrived.
- o ECN Arrival (parameter: ECN value): a packet carrying the conveyed ECN value has arrived. This can be useful for applications implementing congestion control.
- o Timeout (parameter: s seconds): data could not be delivered for s seconds.
- o Drain: the send buffer has either drained below the configured buffer limit or it has become completely empty. This is a generic notification that tries to enable uniform access to "TCP_NOTSENT_LOWAT" as well as the "SENDER DRY" notification (as discussed in Section 5.4 -- SCTP's "SENDER DRY" is a special case where the threshold (for unsent data) is 0 and there is also no more unacknowledged data in the send buffer).

6.2.2. Individual connections

Configure priority or weight for a scheduler, as described in [RFC8260].

Configure checksum usage: this can be done with the following parameters, but there is no guarantee that any checksum limitations will indeed be enforced (the default behavior is "full coverage, checksum enabled"):

- o A boolean to enable / disable usage of a checksum when sending
- o The desired coverage (in bytes) of the checksum used when sending
- o A boolean to enable / disable requiring a checksum when receiving
- o The required minimum coverage (in bytes) of the checksum when receiving

6.3. DATA Transfer

6.3.1. Sending Data

When sending a message, no guarantees are given about the preservation of message boundaries to the peer; if message boundaries are needed, the receiving application at the peer must know about them beforehand (or the transport system cannot use TCP). Note that an application should already be able to hand over data before the transport system establishes a connection with a chosen transport protocol. Regarding the message that is being handed over, the following parameters can be used:

- o Reliability: this parameter is used to convey a choice of: fully reliable with congestion control (not UDP), unreliable without congestion control, unreliable with congestion control (not UDP), partially reliable with congestion control (see [RFC3758] and [RFC7496] for details on how to specify partial reliability) (not UDP). The latter two choices are optional for a transport system to offer and may result in full reliability. Note that applications sending unreliable data without congestion control should themselves perform congestion control in accordance with [RFC8085].
- o (not UDP) Ordered: this boolean parameter lets an application choose between ordered message delivery (true) and possibly unordered, potentially faster message delivery (false).
- o Bundle: a boolean that expresses a preference for allowing to bundle messages (true) or not (false). No guarantees are given.
- o DelAck: a boolean that, if false, lets an application request that the peer would not delay the acknowledgement for this message.
- o Fragment: a boolean that expresses a preference for allowing to fragment messages (true) or not (false), at the IP level. No guarantees are given.
- o (not UDP) Idempotent: a boolean that expresses whether a message is idempotent (true) or not (false). Idempotent messages may arrive multiple times at the receiver (but they will arrive at least once). When data is idempotent it can be used by the receiver immediately on a connection establishment attempt. Thus, if data is handed over before the transport system establishes a connection with a chosen transport protocol, stating that a message is idempotent facilitates transmitting it to the peer application particularly early.

An application can be notified of a failure to send a specific message. There is no guarantee of such notifications, i.e. send failures can also silently occur.

6.3.2. Receiving Data

A receiving application obtains an "Application-Framed Bytestream" (AFra-Bytestream); this concept is further described in Section 5.1). In line with TCP's receiver semantics, an AFra-Bytestream is just a stream of bytes to the receiver. If message boundaries were specified by the sender, a receiver-side transport system implementing only the minimum set of transport services defined here will still not inform the receiving application about them (this limitation is only needed for transport systems that are implemented to directly use TCP).

Different from TCP's semantics, if the sending application has allowed that messages are not fully reliably transferred, or

delivered out of order, then such re-ordering or unreliability may be reflected per message in the arriving data. Messages will always stay intact - i.e. if an incomplete message is contained at the end of the arriving data block, this message is guaranteed to continue in the next arriving data block.

7. Acknowledgements

The authors would like to thank all the participants of the TAPS Working Group and the NEAT and MAMI research projects for valuable input to this document. We especially thank Michael Tuexen for help with connection establishment/teardown, Gorrry Fairhurst for his suggestions regarding fragmentation and packet sizes, and Spencer Dawkins for his extremely detailed and constructive review. This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 644334 (NEAT).

8. IANA Considerations

This memo includes no request to IANA.

9. Security Considerations

Authentication, confidentiality protection, and integrity protection are identified as transport features by [RFC8095]. Often, these features are provided by a protocol or layer on top of the transport protocol; none of the full-featured standards-track transport protocols in [RFC8303], which this document is based upon, provides all of these transport features on its own. Therefore, they are not considered in this document, with the exception of native authentication capabilities of TCP and SCTP for which the security considerations in [RFC5925] and [RFC4895] apply. The minimum requirements for a secure transport system are discussed in a separate document (Section 5 on Security Features and Transport Dependencies of [I-D.ietf-taps-transport-security]).

10. References

10.1. Normative References

[I-D.ietf-taps-transport-security]
Pauly, T., Perkins, C., Rose, K., and C. Wood, "A Survey of Transport Security Protocols", draft-ietf-taps-transport-security-02 (work in progress), June 2018.

- [RFC8095] Fairhurst, G., Ed., Trammell, B., Ed., and M. Kuehlewind, Ed., "Services Provided by IETF Transport Protocols and Congestion Control Mechanisms", RFC 8095, DOI 10.17487/RFC8095, March 2017, <<https://www.rfc-editor.org/info/rfc8095>>.
- [RFC8303] Welzl, M., Tuexen, M., and N. Khademi, "On the Usage of Transport Features Provided by IETF Transport Protocols", RFC 8303, DOI 10.17487/RFC8303, February 2018, <<https://www.rfc-editor.org/info/rfc8303>>.

10.2. Informative References

- [COBS] Cheshire, S. and M. Baker, "Consistent Overhead Byte Stuffing", IEEE/ACM Transactions on Networking Vol. 7, No. 2, April 1999.
- [I-D.ietf-taps-interface] Trammell, B., Welzl, M., Enghardt, T., Fairhurst, G., Kuehlewind, M., Perkins, C., Tiesel, P., and C. Wood, "An Abstract Application Layer Interface to Transport Services", draft-ietf-taps-interface-01 (work in progress), July 2018.
- [I-D.ietf-tsvwg-rtcweb-qos] Jones, P., Dhesikan, S., Jennings, C., and D. Druta, "DSCP Packet Markings for WebRTC QoS", draft-ietf-tsvwg-rtcweb-qos-18 (work in progress), August 2016.
- [LBE-draft] Bless, R., "A Lower Effort Per-Hop Behavior (LE PHB)", Internet-draft draft-tsvwg-le-phb-03, February 2018.
- [POSIX] "IEEE Standard for Information Technology--Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7", IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008), January 2018, <<http://www.opengroup.org/onlinepubs/9699919799/functions/contents.html>>.
- [RFC3758] Stewart, R., Ramalho, M., Xie, Q., Tuexen, M., and P. Conrad, "Stream Control Transmission Protocol (SCTP) Partial Reliability Extension", RFC 3758, DOI 10.17487/RFC3758, May 2004, <<https://www.rfc-editor.org/info/rfc3758>>.

- [RFC4895] Tuexen, M., Stewart, R., Lei, P., and E. Rescorla, "Authenticated Chunks for the Stream Control Transmission Protocol (SCTP)", RFC 4895, DOI 10.17487/RFC4895, August 2007, <<https://www.rfc-editor.org/info/rfc4895>>.
- [RFC4987] Eddy, W., "TCP SYN Flooding Attacks and Common Mitigations", RFC 4987, DOI 10.17487/RFC4987, August 2007, <<https://www.rfc-editor.org/info/rfc4987>>.
- [RFC5925] Touch, J., Mankin, A., and R. Bonica, "The TCP Authentication Option", RFC 5925, DOI 10.17487/RFC5925, June 2010, <<https://www.rfc-editor.org/info/rfc5925>>.
- [RFC6897] Scharf, M. and A. Ford, "Multipath TCP (MPTCP) Application Interface Considerations", RFC 6897, DOI 10.17487/RFC6897, March 2013, <<https://www.rfc-editor.org/info/rfc6897>>.
- [RFC7305] Lear, E., Ed., "Report from the IAB Workshop on Internet Technology Adoption and Transition (ITAT)", RFC 7305, DOI 10.17487/RFC7305, July 2014, <<https://www.rfc-editor.org/info/rfc7305>>.
- [RFC7413] Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP Fast Open", RFC 7413, DOI 10.17487/RFC7413, December 2014, <<https://www.rfc-editor.org/info/rfc7413>>.
- [RFC7496] Tuexen, M., Seggelmann, R., Stewart, R., and S. Loreto, "Additional Policies for the Partially Reliable Stream Control Transmission Protocol Extension", RFC 7496, DOI 10.17487/RFC7496, April 2015, <<https://www.rfc-editor.org/info/rfc7496>>.
- [RFC8085] Eggert, L., Fairhurst, G., and G. Shepherd, "UDP Usage Guidelines", BCP 145, RFC 8085, DOI 10.17487/RFC8085, March 2017, <<https://www.rfc-editor.org/info/rfc8085>>.
- [RFC8260] Stewart, R., Tuexen, M., Loreto, S., and R. Seggelmann, "Stream Schedulers and User Message Interleaving for the Stream Control Transmission Protocol", RFC 8260, DOI 10.17487/RFC8260, November 2017, <<https://www.rfc-editor.org/info/rfc8260>>.
- [RFC8304] Fairhurst, G. and T. Jones, "Transport Features of the User Datagram Protocol (UDP) and Lightweight UDP (UDP-Lite)", RFC 8304, DOI 10.17487/RFC8304, February 2018, <<https://www.rfc-editor.org/info/rfc8304>>.

[SCTP-stream-1]

Weinrank, F. and M. Tuexen, "Transparent Flow Mapping for NEAT", IFIP NETWORKING Workshop on Future of Internet Transport (FIT 2017), June 2017.

[SCTP-stream-2]

Welzl, M., Niederbacher, F., and S. Gjessing, "Beneficial Transparent Deployment of SCTP", IEEE GlobeCom 2011, December 2011.

[WWDC2015]

Lakhera, P. and S. Cheshire, "Your App and Next Generation Networks", Apple Worldwide Developers Conference 2015, San Francisco, USA, June 2015, <<https://developer.apple.com/videos/wwdc/2015/?id=719>>.

Appendix A. The Superset of Transport Features

In this description, transport features are presented following the nomenclature "CATEGORY.[SUBCATEGORY].FEATURENAME.PROTOCOL", equivalent to "pass 2" in [RFC8303]. We also sketch how functional or optimizing transport features can be implemented by a transport system. The "minimal set" derived in this document is meant to be implementable "one-sided" over TCP, and, with limitations, UDP. Hence, for all transport features that are categorized as "functional" or "optimizing", and for which no matching TCP and/or UDP primitive exists in "pass 2" of [RFC8303], a brief discussion on how to implement them over TCP and/or UDP is included.

We designate some transport features as "automatable" on the basis of a broader decision that affects multiple transport features:

- o Most transport features that are related to multi-streaming were designated as "automatable". This was done because the decision on whether to use multi-streaming or not does not depend on application-specific knowledge. This means that a connection that is exhibited to an application could be implemented by using a single stream of an SCTP association instead of mapping it to a complete SCTP association or TCP connection. This could be achieved by using more than one stream when an SCTP association is first established (CONNECT.SCTP parameter "outbound stream count"), maintaining an internal stream number, and using this stream number when sending data (SEND.SCTP parameter "stream number"). Closing or aborting a connection could then simply free the stream number for future use. This is discussed further in Section 5.2.
- o With the exception of "Disable MPTCP", all transport features that are related to using multiple paths or the choice of the network

interface were designated as "automatable". For example, "Listen" could always listen on all available interfaces and "Connect" could use the default interface for the destination IP address.

Finally, in three cases, transport features are aggregated and/or slightly changed from [RFC8303] in the description below. These transport features are marked as "CHANGED FROM RFC8303". These do not add any new functionality but just represent a simple refactoring step that helps to streamline the derivation process (e.g., by removing a choice of a parameter for the sake of applications that may not care about this choice). The corresponding transport features are automatable, and they are listed immediately below the "CHANGED FROM RFC8303" transport feature.

A.1. CONNECTION Related Transport Features

ESTABLISHMENT:

- o Connect
Protocols: TCP, SCTP, UDP(-Lite)
Functional because the notion of a connection is often reflected in applications as an expectation to be able to communicate after a "Connect" succeeded, with a communication sequence relating to this transport feature that is defined by the application protocol.
Implementation: via CONNECT.TCP, CONNECT.SCTP or CONNECT.UDP(-Lite).
- o Specify which IP Options must always be used
Protocols: TCP, UDP(-Lite)
Automatable because IP Options relate to knowledge about the network, not the application.
- o Request multiple streams
Protocols: SCTP
Automatable because using multi-streaming does not require application-specific knowledge (example implementations of using multi-streaming without involving the application are described in [SCTP-stream-1] and [SCTP-stream-2]).
Implementation: see Section 5.2.
- o Limit the number of inbound streams
Protocols: SCTP

Automatable because using multi-streaming does not require application-specific knowledge.

Implementation: see Section 5.2.

- o Specify number of attempts and/or timeout for the first establishment message
Protocols: TCP, SCTP
Functional because this is closely related to potentially assumed reliable data delivery for data that is sent before or during connection establishment.
Implementation: Using a parameter of CONNECT.TCP and CONNECT.SCTP.
Implementation over UDP: Do nothing (this is irrelevant in case of UDP because there, reliable data delivery is not assumed).
- o Obtain multiple sockets
Protocols: SCTP
Automatable because the non-parallel usage of multiple paths to communicate between the same end hosts relates to knowledge about the network, not the application.
- o Disable MPTCP
Protocols: MPTCP
Optimizing because the parallel usage of multiple paths to communicate between the same end hosts can improve performance. Whether to use this feature depends on knowledge about the network as well as application-specific knowledge (see Section 3.1 of [RFC6897]).
Implementation: via a boolean parameter in CONNECT.MPTCP.
Implementation over TCP: Do nothing.
Implementation over UDP: Do nothing.
- o Configure authentication
Protocols: TCP, SCTP
Functional because this has a direct influence on security.
Implementation: via parameters in CONNECT.TCP and CONNECT.SCTP.
With TCP, this allows to configure Master Key Tuples (MKTs) to authenticate complete segments (including the TCP IPv4 pseudoheader, TCP header, and TCP data). With SCTP, this allows to specify which chunk types must always be authenticated.
Authenticating only certain chunk types creates a reduced level of security that is not supported by TCP; to be compatible, this should therefore only allow to authenticate all chunk types. Key

material must be provided in a way that is compatible with both [RFC4895] and [RFC5925].

Implementation over UDP: Not possible (UDP does not offer this functionality).

- o Indicate (and/or obtain upon completion) an Adaptation Layer via an adaptation code point
Protocols: SCTP
Functional because it allows to send extra data for the sake of identifying an adaptation layer, which by itself is application-specific.
Implementation: via a parameter in CONNECT.SCTP.
Implementation over TCP: not possible (TCP does not offer this functionality).
Implementation over UDP: not possible (UDP does not offer this functionality).
- o Request to negotiate interleaving of user messages
Protocols: SCTP
Automatable because it requires using multiple streams, but requesting multiple streams in the CONNECTION.ESTABLISHMENT category is automatable.
Implementation: controlled via a parameter in CONNECT.SCTP. One possible implementation is to always try to enable interleaving.
- o Hand over a message to reliably transfer (possibly multiple times) before connection establishment
Protocols: TCP
Functional because this is closely tied to properties of the data that an application sends or expects to receive.
Implementation: via a parameter in CONNECT.TCP.
Implementation over UDP: not possible (UDP does not provide reliability).
- o Hand over a message to reliably transfer during connection establishment
Protocols: SCTP
Functional because this can only work if the message is limited in size, making it closely tied to properties of the data that an application sends or expects to receive.
Implementation: via a parameter in CONNECT.SCTP.

Implementation over TCP: not possible (TCP does not allow identification of message boundaries because it provides a byte stream service)

Implementation over UDP: not possible (UDP is unreliable).

- o Enable UDP encapsulation with a specified remote UDP port number
Protocols: SCTP
Automatable because UDP encapsulation relates to knowledge about the network, not the application.

AVAILABILITY:

- o Listen
Protocols: TCP, SCTP, UDP(-Lite)
Functional because the notion of accepting connection requests is often reflected in applications as an expectation to be able to communicate after a "Listen" succeeded, with a communication sequence relating to this transport feature that is defined by the application protocol.
CHANGED FROM RFC8303. This differs from the 3 automatable transport features below in that it leaves the choice of interfaces for listening open.
Implementation: by listening on all interfaces via LISTEN.TCP (not providing a local IP address) or LISTEN.SCTP (providing SCTP port number / address pairs for all local IP addresses). LISTEN.UDP(-Lite) supports both methods.
- o Listen, 1 specified local interface
Protocols: TCP, SCTP, UDP(-Lite)
Automatable because decisions about local interfaces relate to knowledge about the network and the Operating System, not the application.
- o Listen, N specified local interfaces
Protocols: SCTP
Automatable because decisions about local interfaces relate to knowledge about the network and the Operating System, not the application.

- o Listen, all local interfaces
Protocols: TCP, SCTP, UDP(-Lite)
Automatable because decisions about local interfaces relate to knowledge about the network and the Operating System, not the application.
- o Specify which IP Options must always be used
Protocols: TCP, UDP(-Lite)
Automatable because IP Options relate to knowledge about the network, not the application.
- o Disable MPTCP
Protocols: MPTCP
Optimizing because the parallel usage of multiple paths to communicate between the same end hosts can improve performance. Whether to use this feature depends on knowledge about the network as well as application-specific knowledge (see Section 3.1 of [RFC6897]).
Implementation: via a boolean parameter in LISTEN.MPTCP.
Implementation over TCP: Do nothing.
Implementation over UDP: Do nothing.
- o Configure authentication
Protocols: TCP, SCTP
Functional because this has a direct influence on security.
Implementation: via parameters in LISTEN.TCP and LISTEN.SCTP.
Implementation over TCP: With TCP, this allows to configure Master Key Tuples (MKTs) to authenticate complete segments (including the TCP IPv4 pseudoheader, TCP header, and TCP data). With SCTP, this allows to specify which chunk types must always be authenticated. Authenticating only certain chunk types creates a reduced level of security that is not supported by TCP; to be compatible, this should therefore only allow to authenticate all chunk types. Key material must be provided in a way that is compatible with both [RFC4895] and [RFC5925].
Implementation over UDP: not possible (UDP does not offer authentication).
- o Obtain requested number of streams
Protocols: SCTP
Automatable because using multi-streaming does not require application-specific knowledge.

Implementation: see Section 5.2.

- o Limit the number of inbound streams
Protocols: SCTP
Automatable because using multi-streaming does not require application-specific knowledge.
Implementation: see Section 5.2.
- o Indicate (and/or obtain upon completion) an Adaptation Layer via an adaptation code point
Protocols: SCTP
Functional because it allows to send extra data for the sake of identifying an adaptation layer, which by itself is application-specific.
Implementation: via a parameter in LISTEN.SCTP.
Implementation over TCP: not possible (TCP does not offer this functionality).
Implementation over UDP: not possible (UDP does not offer this functionality).
- o Request to negotiate interleaving of user messages
Protocols: SCTP
Automatable because it requires using multiple streams, but requesting multiple streams in the CONNECTION.ESTABLISHMENT category is automatable.
Implementation: via a parameter in LISTEN.SCTP.

MAINTENANCE:

- o Change timeout for aborting connection (using retransmit limit or time value)
Protocols: TCP, SCTP
Functional because this is closely related to potentially assumed reliable data delivery.
Implementation: via CHANGE_TIMEOUT.TCP or CHANGE_TIMEOUT.SCTP.
Implementation over UDP: not possible (UDP is unreliable and there is no connection timeout).
- o Suggest timeout to the peer
Protocols: TCP

Functional because this is closely related to potentially assumed reliable data delivery.

Implementation: via `CHANGE_TIMEOUT.TCP`.

Implementation over UDP: not possible (UDP is unreliable and there is no connection timeout).

- o Disable Nagle algorithm
Protocols: TCP, SCTP
Optimizing because this decision depends on knowledge about the size of future data blocks and the delay between them.
Implementation: via `DISABLE_NAGLE.TCP` and `DISABLE_NAGLE.SCTP`.
Implementation over UDP: do nothing (UDP does not implement the Nagle algorithm).
- o Request an immediate heartbeat, returning success/failure
Protocols: SCTP
Automatable because this informs about network-specific knowledge.
- o Notification of Excessive Retransmissions (early warning below abortion threshold)
Protocols: TCP
Optimizing because it is an early warning to the application, informing it of an impending functional event.
Implementation: via `ERROR.TCP`.
Implementation over UDP: do nothing (there is no abortion threshold).
- o Add path
Protocols: MPTCP, SCTP
MPTCP Parameters: source-IP; source-Port; destination-IP; destination-Port
SCTP Parameters: local IP address
Automatable because the choice of paths to communicate between the same end hosts relates to knowledge about the network, not the application.
- o Remove path
Protocols: MPTCP, SCTP

MPTCP Parameters: source-IP; source-Port; destination-IP;
destination-Port

SCTP Parameters: local IP address

Automatable because the choice of paths to communicate between the same end host relates to knowledge about the network, not the application.

- o Set primary path

Protocols: SCTP

Automatable because the choice of paths to communicate between the same end hosts relates to knowledge about the network, not the application.

- o Suggest primary path to the peer

Protocols: SCTP

Automatable because the choice of paths to communicate between the same end hosts relates to knowledge about the network, not the application.

- o Configure Path Switchover

Protocols: SCTP

Automatable because the choice of paths to communicate between the same end hosts relates to knowledge about the network, not the application.

- o Obtain status (query or notification)

Protocols: SCTP, MPTCP

SCTP parameters: association connection state; destination transport address list; destination transport address reachability states; current local and peer receiver window size; current local congestion window sizes; number of unacknowledged DATA chunks; number of DATA chunks pending receipt; primary path; most recent SRTT on primary path; RTO on primary path; SRTT and RTO on other destination addresses; MTU per path; interleaving supported yes/no
MPTCP parameters: subflow-list (identified by source-IP; source-Port; destination-IP; destination-Port)

Automatable because these parameters relate to knowledge about the network, not the application.

- o Specify DSCP field
Protocols: TCP, SCTP, UDP(-Lite)
Optimizing because choosing a suitable DSCP value requires application-specific knowledge.
Implementation: via SET_DSCP.TCP / SET_DSCP.SCTP / SET_DSCP.UDP(-Lite)
- o Notification of ICMP error message arrival
Protocols: TCP, UDP(-Lite)
Optimizing because these messages can inform about success or failure of functional transport features (e.g., host unreachable relates to "Connect")
Implementation: via ERROR.TCP or ERROR.UDP(-Lite).
- o Obtain information about interleaving support
Protocols: SCTP
Automatable because it requires using multiple streams, but requesting multiple streams in the CONNECTION.ESTABLISHMENT category is automatable.
Implementation: via STATUS.SCTP.
- o Change authentication parameters
Protocols: TCP, SCTP
Functional because this has a direct influence on security.
Implementation: via SET_AUTH.TCP and SET_AUTH.SCTP.
Implementation over TCP: With SCTP, this allows to adjust key_id, key, and hmac_id. With TCP, this allows to change the preferred outgoing MKT (current_key) and the preferred incoming MKT (rnext_key), respectively, for a segment that is sent on the connection. Key material must be provided in a way that is compatible with both [RFC4895] and [RFC5925].
Implementation over UDP: not possible (UDP does not offer authentication).
- o Obtain authentication information
Protocols: SCTP
Functional because authentication decisions may have been made by the peer, and this has an influence on the necessary application-level measures to provide a certain level of security.
Implementation: via GET_AUTH.SCTP.

Implementation over TCP: With SCTP, this allows to obtain `key_id` and a chunk list. With TCP, this allows to obtain `current_key` and `rnext_key` from a previously received segment. Key material must be provided in a way that is compatible with both [RFC4895] and [RFC5925].

Implementation over UDP: not possible (UDP does not offer authentication).

- o Reset Stream
Protocols: SCTP
Automatable because using multi-streaming does not require application-specific knowledge.
Implementation: see Section 5.2.
- o Notification of Stream Reset
Protocols: STCP
Automatable because using multi-streaming does not require application-specific knowledge.
Implementation: see Section 5.2.
- o Reset Association
Protocols: SCTP
Automatable because deciding to reset an association does not require application-specific knowledge.
Implementation: via `RESET_ASSOC.SCTP`.
- o Notification of Association Reset
Protocols: STCP
Automatable because this notification does not relate to application-specific knowledge.
- o Add Streams
Protocols: SCTP
Automatable because using multi-streaming does not require application-specific knowledge.
Implementation: see Section 5.2.
- o Notification of Added Stream
Protocols: STCP

Automatable because using multi-streaming does not require application-specific knowledge.

Implementation: see Section 5.2.

- o Choose a scheduler to operate between streams of an association
Protocols: SCTP
Optimizing because the scheduling decision requires application-specific knowledge. However, if a transport system would not use this, or wrongly configure it on its own, this would only affect the performance of data transfers; the outcome would still be correct within the "best effort" service model.
Implementation: using SET_STREAM_SCHEDULER.SCTP.
Implementation over TCP: do nothing (streams are not available in TCP, but no guarantee is given that this transport feature has any effect).
Implementation over UDP: do nothing (streams are not available in UDP, but no guarantee is given that this transport feature has any effect).
- o Configure priority or weight for a scheduler
Protocols: SCTP
Optimizing because the priority or weight requires application-specific knowledge. However, if a transport system would not use this, or wrongly configure it on its own, this would only affect the performance of data transfers; the outcome would still be correct within the "best effort" service model.
Implementation: using CONFIGURE_STREAM_SCHEDULER.SCTP.
Implementation over TCP: do nothing (streams are not available in TCP, but no guarantee is given that this transport feature has any effect).
Implementation over UDP: do nothing (streams are not available in UDP, but no guarantee is given that this transport feature has any effect).
- o Configure send buffer size
Protocols: SCTP
Automatable because this decision relates to knowledge about the network and the Operating System, not the application (see also the discussion in Section 5.4).
- o Configure receive buffer (and rwnd) size

Protocols: SCTP

Automatable because this decision relates to knowledge about the network and the Operating System, not the application.

- o Configure message fragmentation

Protocols: SCTP

Automatable because this relates to knowledge about the network and the Operating System, not the application. Note that this SCTP feature does not control IP-level fragmentation, but decides on fragmentation of messages by SCTP, in the end system.

Implementation: by always enabling it with CONFIG_FRAGMENTATION.SCTP and auto-setting the fragmentation size based on network or Operating System conditions.

- o Configure PMTUD

Protocols: SCTP

Automatable because Path MTU Discovery relates to knowledge about the network, not the application.

- o Configure delayed SACK timer

Protocols: SCTP

Automatable because the receiver-side decision to delay sending SACKs relates to knowledge about the network, not the application (it can be relevant for a sending application to request not to delay the SACK of a message, but this is a different transport feature).

- o Set Cookie life value

Protocols: SCTP

Functional because it relates to security (possibly weakened by keeping a cookie very long) versus the time between connection establishment attempts. Knowledge about both issues can be application-specific.

Implementation over TCP: the closest specified TCP functionality is the cookie in TCP Fast Open; for this, [RFC7413] states that the server "can expire the cookie at any time to enhance security" and section 4.1.2 describes an example implementation where updating the key on the server side causes the cookie to expire. Alternatively, for implementations that do not support TCP Fast

Open, this transport feature could also affect the validity of SYN cookies (see Section 3.6 of [RFC4987]).

Implementation over UDP: not possible (UDP does not offer this functionality).

- o Set maximum burst
Protocols: SCTP
Automatable because it relates to knowledge about the network, not the application.
- o Configure size where messages are broken up for partial delivery
Protocols: SCTP
Functional because this is closely tied to properties of the data that an application sends or expects to receive.
Implementation over TCP: not possible (TCP does not offer identification of message boundaries).
Implementation over UDP: not possible (UDP does not fragment messages).
- o Disable checksum when sending
Protocols: UDP
Functional because application-specific knowledge is necessary to decide whether it can be acceptable to lose data integrity with respect to random corruption.
Implementation: via `SET_CHECKSUM_ENABLED.UDP`.
Implementation over TCP: do nothing (TCP does not offer to disable the checksum, but transmitting data with an intact checksum will not yield a semantically wrong result).
- o Disable checksum requirement when receiving
Protocols: UDP
Functional because application-specific knowledge is necessary to decide whether it can be acceptable to lose data integrity with respect to random corruption.
Implementation: via `SET_CHECKSUM_REQUIRED.UDP`.
Implementation over TCP: do nothing (TCP does not offer to disable the checksum, but transmitting data with an intact checksum will not yield a semantically wrong result).
- o Specify checksum coverage used by the sender

Protocols: UDP-Lite

Functional because application-specific knowledge is necessary to decide for which parts of the data it can be acceptable to lose data integrity with respect to random corruption.

Implementation: via SET_CHECKSUM_COVERAGE.UDP-Lite.

Implementation over TCP: do nothing (TCP does not offer to limit the checksum length, but transmitting data with an intact checksum will not yield a semantically wrong result).

Implementation over UDP: if checksum coverage is set to cover payload data, do nothing. Else, either do nothing (transmitting data with an intact checksum will not yield a semantically wrong result), or use the transport feature "Disable checksum when sending".

- o Specify minimum checksum coverage required by receiver

Protocols: UDP-Lite

Functional because application-specific knowledge is necessary to decide for which parts of the data it can be acceptable to lose data integrity with respect to random corruption.

Implementation: via SET_MIN_CHECKSUM_COVERAGE.UDP-Lite.

Implementation over TCP: do nothing (TCP does not offer to limit the checksum length, but transmitting data with an intact checksum will not yield a semantically wrong result).

Implementation over UDP: if checksum coverage is set to cover payload data, do nothing. Else, either do nothing (transmitting data with an intact checksum will not yield a semantically wrong result), or use the transport feature "Disable checksum requirement when receiving".

- o Specify DF field

Protocols: UDP(-Lite)

Optimizing because the DF field can be used to carry out Path MTU Discovery, which can lead an application to choose message sizes that can be transmitted more efficiently.

Implementation: via MAINTENANCE.SET_DF.UDP(-Lite) and SEND_FAILURE.UDP(-Lite).

Implementation over TCP: do nothing (with TCP, the sending application is not in control of transport message sizes, making this functionality irrelevant).

- o Get max. transport-message size that may be sent using a non-fragmented IP packet from the configured interface

Protocols: UDP(-Lite)

Optimizing because this can lead an application to choose message sizes that can be transmitted more efficiently.

Implementation over TCP: do nothing (this information is not available with TCP).

- o Get max. transport-message size that may be received from the configured interface
Protocols: UDP(-Lite)
Optimizing because this can, for example, influence an application's memory management.
Implementation over TCP: do nothing (this information is not available with TCP).
- o Specify TTL/Hop count field
Protocols: UDP(-Lite)
Automatable because a transport system can use a large enough system default to avoid communication failures. Allowing an application to configure it differently can produce notifications of ICMP error message arrivals that yield information which only relates to knowledge about the network, not the application.
- o Obtain TTL/Hop count field
Protocols: UDP(-Lite)
Automatable because the TTL/Hop count field relates to knowledge about the network, not the application.
- o Specify ECN field
Protocols: UDP(-Lite)
Automatable because the ECN field relates to knowledge about the network, not the application.
- o Obtain ECN field
Protocols: UDP(-Lite)
Optimizing because this information can be used by an application to better carry out congestion control (this is relevant when choosing a data transmission transport service that does not already do congestion control).
Implementation over TCP: do nothing (this information is not available with TCP).

- o Specify IP Options
Protocols: UDP(-Lite)
Automatable because IP Options relate to knowledge about the network, not the application.
- o Obtain IP Options
Protocols: UDP(-Lite)
Automatable because IP Options relate to knowledge about the network, not the application.
- o Enable and configure a "Low Extra Delay Background Transfer"
Protocols: A protocol implementing the LEDBAT congestion control mechanism
Optimizing because whether this feature is appropriate or not depends on application-specific knowledge. However, wrongly using this will only affect the speed of data transfers (albeit including other transfers that may compete with the transport system's transfer in the network), so it is still correct within the "best effort" service model.
Implementation: via CONFIGURE.LEDBAT and/or SET_DSCP.TCP / SET_DSCP.SCTP / SET_DSCP.UDP(-Lite) [LBE-draft].
Implementation over TCP: do nothing (TCP does not support LEDBAT congestion control, but not implementing this functionality will not yield a semantically wrong behavior).
Implementation over UDP: do nothing (UDP does not offer congestion control).

TERMINATION:

- o Close after reliably delivering all remaining data, causing an event informing the application on the other side
Protocols: TCP, SCTP
Functional because the notion of a connection is often reflected in applications as an expectation to have all outstanding data delivered and no longer be able to communicate after a "Close" succeeded, with a communication sequence relating to this transport feature that is defined by the application protocol.
Implementation: via CLOSE.TCP and CLOSE.SCTP.
Implementation over UDP: not possible (UDP is unreliable and hence does not know when all remaining data is delivered; it does also not offer to cause an event related to closing at the peer).

- o Abort without delivering remaining data, causing an event informing the application on the other side
Protocols: TCP, SCTP
Functional because the notion of a connection is often reflected in applications as an expectation to potentially not have all outstanding data delivered and no longer be able to communicate after an "Abort" succeeded. On both sides of a connection, an application protocol may define a communication sequence relating to this transport feature.
Implementation: via ABORT.TCP and ABORT.SCTP.
Implementation over UDP: not possible (UDP does not offer to cause an event related to aborting at the peer).

- o Abort without delivering remaining data, not causing an event informing the application on the other side
Protocols: UDP(-Lite)
Functional because the notion of a connection is often reflected in applications as an expectation to potentially not have all outstanding data delivered and no longer be able to communicate after an "Abort" succeeded. On both sides of a connection, an application protocol may define a communication sequence relating to this transport feature.
Implementation: via ABORT.UDP(-Lite).
Implementation over TCP: stop using the connection, wait for a timeout.

- o Timeout event when data could not be delivered for too long
Protocols: TCP, SCTP
Functional because this notifies that potentially assumed reliable data delivery is no longer provided.
Implementation: via TIMEOUT.TCP and TIMEOUT.SCTP.
Implementation over UDP: do nothing (this event will not occur with UDP).

A.2. DATA Transfer Related Transport Features

A.2.1. Sending Data

- o Reliably transfer data, with congestion control
Protocols: TCP, SCTP

Functional because this is closely tied to properties of the data that an application sends or expects to receive.

Implementation: via SEND.TCP and SEND.SCTP.

Implementation over UDP: not possible (UDP is unreliable).

- o Reliably transfer a message, with congestion control
Protocols: SCTP
Functional because this is closely tied to properties of the data that an application sends or expects to receive.
Implementation: via SEND.SCTP.
Implementation over TCP: via SEND.TCP. With SEND.TCP, message boundaries will not be identifiable by the receiver, because TCP provides a byte stream service.
Implementation over UDP: not possible (UDP is unreliable).
- o Unreliably transfer a message
Protocols: SCTP, UDP(-Lite)
Optimizing because only applications know about the time criticality of their communication, and reliably transferring a message is never incorrect for the receiver of a potentially unreliable data transfer, it is just slower.
CHANGED FROM RFC8303. This differs from the 2 automatable transport features below in that it leaves the choice of congestion control open.
Implementation: via SEND.SCTP or SEND.UDP(-Lite).
Implementation over TCP: use SEND.TCP. With SEND.TCP, messages will be sent reliably, and message boundaries will not be identifiable by the receiver.
- o Unreliably transfer a message, with congestion control
Protocols: SCTP
Automatable because congestion control relates to knowledge about the network, not the application.
- o Unreliably transfer a message, without congestion control
Protocols: UDP(-Lite)
Automatable because congestion control relates to knowledge about the network, not the application.

- o Configurable Message Reliability
Protocols: SCTP
Optimizing because only applications know about the time criticality of their communication, and reliably transferring a message is never incorrect for the receiver of a potentially unreliable data transfer, it is just slower.
Implementation: via SEND.SCTP.
Implementation over TCP: By using SEND.TCP and ignoring this configuration: based on the assumption of the best-effort service model, unnecessarily delivering data does not violate application expectations. Moreover, it is not possible to associate the requested reliability to a "message" in TCP anyway.
Implementation over UDP: not possible (UDP is unreliable).

- o Choice of stream
Protocols: SCTP
Automatable because it requires using multiple streams, but requesting multiple streams in the CONNECTION.ESTABLISHMENT category is automatable. Implementation: see Section 5.2.

- o Choice of path (destination address)
Protocols: SCTP
Automatable because it requires using multiple sockets, but obtaining multiple sockets in the CONNECTION.ESTABLISHMENT category is automatable.

- o Ordered message delivery (potentially slower than unordered)
Protocols: SCTP
Functional because this is closely tied to properties of the data that an application sends or expects to receive.
Implementation: via SEND.SCTP.
Implementation over TCP: By using SEND.TCP. With SEND.TCP, messages will not be identifiable by the receiver.
Implementation over UDP: not possible (UDP does not offer any guarantees regarding ordering).

- o Unordered message delivery (potentially faster than ordered)
Protocols: SCTP, UDP(-Lite)
Functional because this is closely tied to properties of the data that an application sends or expects to receive.
Implementation: via SEND.SCTP.

Implementation over TCP: By using SEND.TCP and always sending data ordered: based on the assumption of the best-effort service model, ordered delivery may just be slower and does not violate application expectations. Moreover, it is not possible to associate the requested delivery order to a "message" in TCP anyway.

- o Request not to bundle messages
Protocols: SCTP
Optimizing because this decision depends on knowledge about the size of future data blocks and the delay between them.
Implementation: via SEND.SCTP.
Implementation over TCP: By using SEND.TCP and DISABLE_NAGLE.TCP to disable the Nagle algorithm when the request is made and enable it again when the request is no longer made. Note that this is not fully equivalent because it relates to the time of issuing the request rather than a specific message.
Implementation over UDP: do nothing (UDP never bundles messages).

- o Specifying a "payload protocol-id" (handed over as such by the receiver)
Protocols: SCTP
Functional because it allows to send extra application data with every message, for the sake of identification of data, which by itself is application-specific.
Implementation: SEND.SCTP.
Implementation over TCP: not possible (this functionality is not available in TCP).
Implementation over UDP: not possible (this functionality is not available in UDP).

- o Specifying a key id to be used to authenticate a message
Protocols: SCTP
Functional because this has a direct influence on security.
Implementation: via a parameter in SEND.SCTP.
Implementation over TCP: This could be emulated by using SET_AUTH.TCP before and after the message is sent. Note that this is not fully equivalent because it relates to the time of issuing the request rather than a specific message.
Implementation over UDP: not possible (UDP does not offer authentication).

- o Request not to delay the acknowledgement (SACK) of a message
Protocols: SCTP
Optimizing because only an application knows for which message it wants to quickly be informed about success / failure of its delivery.
Implementation over TCP: do nothing (TCP does not offer this functionality, but ignoring this request from the application will not yield a semantically wrong behavior).
Implementation over UDP: do nothing (UDP does not offer this functionality, but ignoring this request from the application will not yield a semantically wrong behavior).

A.2.2. Receiving Data

- o Receive data (with no message delimiting)
Protocols: TCP
Functional because a transport system must be able to send and receive data.
Implementation: via RECEIVE.TCP.
Implementation over UDP: do nothing (UDP only works on messages; these can be handed over, the application can still ignore the message boundaries).
- o Receive a message
Protocols: SCTP, UDP(-Lite)
Functional because this is closely tied to properties of the data that an application sends or expects to receive.
Implementation: via RECEIVE.SCTP and RECEIVE.UDP(-Lite).
Implementation over TCP: not possible (TCP does not support identification of message boundaries).
- o Choice of stream to receive from
Protocols: SCTP
Automatable because it requires using multiple streams, but requesting multiple streams in the CONNECTION.ESTABLISHMENT category is automatable.
Implementation: see Section 5.2.
- o Information about partial message arrival
Protocols: SCTP

Functional because this is closely tied to properties of the data that an application sends or expects to receive.

Implementation: via RECEIVE.SCTP.

Implementation over TCP: do nothing (this information is not available with TCP).

Implementation over UDP: do nothing (this information is not available with UDP).

A.2.3. Errors

This section describes sending failures that are associated with a specific call to in the "Sending Data" category (Appendix A.2.1).

- o Notification of send failures
Protocols: SCTP, UDP(-Lite)
Functional because this notifies that potentially assumed reliable data delivery is no longer provided.
CHANGED FROM RFC8303. This differs from the 2 automatable transport features below in that it does not distinguish between unsent and unacknowledged messages.
Implementation: via SENDFAILURE-EVENT.SCTP and SEND_FAILURE.UDP(-Lite).
Implementation over TCP: do nothing (this notification is not available and will therefore not occur with TCP).
- o Notification of an unsent (part of a) message
Protocols: SCTP, UDP(-Lite)
Automatable because the distinction between unsent and unacknowledged does not relate to application-specific knowledge.
- o Notification of an unacknowledged (part of a) message
Protocols: SCTP
Automatable because the distinction between unsent and unacknowledged does not relate to application-specific knowledge.
- o Notification that the stack has no more user data to send
Protocols: SCTP

Optimizing because reacting to this notification requires the application to be involved, and ensuring that the stack does not run dry of data (for too long) can improve performance.

Implementation over TCP: do nothing (see the discussion in Section 5.4).

Implementation over UDP: do nothing (this notification is not available and will therefore not occur with UDP).

- o Notification to a receiver that a partial message delivery has been aborted

Protocols: SCTP

Functional because this is closely tied to properties of the data that an application sends or expects to receive.

Implementation over TCP: do nothing (this notification is not available and will therefore not occur with TCP).

Implementation over UDP: do nothing (this notification is not available and will therefore not occur with UDP).

Appendix B. Revision information

XXX RFC-Ed please remove this section prior to publication.

-02: implementation suggestions added, discussion section added, terminology extended, DELETED category removed, various other fixes; list of Transport Features adjusted to -01 version of [RFC8303] except that MPTCP is not included.

-03: updated to be consistent with -02 version of [RFC8303].

-04: updated to be consistent with -03 version of [RFC8303]. Reorganized document, rewrote intro and conclusion, and made a first stab at creating a real "minimal set".

-05: updated to be consistent with -05 version of [RFC8303] (minor changes). Fixed a mistake regarding Cookie Life value. Exclusion of security related transport features (to be covered in a separate document). Reorganized the document (now begins with the minset, derivation is in the appendix). First stab at an abstract API for the minset.

draft-ietf-taps-minset-00: updated to be consistent with -08 version of [RFC8303] ("obtain message delivery number" was removed, as this has also been removed in [RFC8303] because it was a mistake in

RFC4960. This led to the removal of two more transport features that were only designated as functional because they affected "obtain message delivery number"). Fall-back to UDP incorporated (this was requested at IETF-99); this also affected the transport feature "Choice between unordered (potentially faster) or ordered delivery of messages" because this is a boolean which is always true for one fall-back protocol, and always false for the other one. This was therefore now divided into two features, one for ordered, one for unordered delivery. The word "reliably" was added to the transport features "Hand over a message to reliably transfer (possibly multiple times) before connection establishment" and "Hand over a message to reliably transfer during connection establishment" to make it clearer why this is not supported by UDP. Clarified that the "minset abstract interface" is not proposing a specific API for all TAPS systems to implement, but it is just a way to describe the minimum set. Author order changed.

WG -01: "fall-back to" (TCP or UDP) replaced (mostly with "implementation over"). References to post-sockets removed (these were statements that assumed that post-sockets requires two-sided implementation). Replaced "flow" with "TAPS Connection" and "frame" with "message" to avoid introducing new terminology. Made sections 3 and 4 in line with the categorization that is already used in the appendix and [RFC8303], and changed style of section 4 to be even shorter and less interface-like. Updated reference draft-ietf-tsvwg-sctp-ndata to RFC8260.

WG -02: rephrased "the TAPS system" and "TAPS connection" etc. to more generally talk about transport after the intro (mostly replacing "TAPS system" with "transport system" and "TAPS connection" with "connection". Merged sections 3 and 4 to form a new section 3.

WG -03: updated sentence referencing [I-D.ietf-taps-transport-security] to say that "the minimum security requirements for a taps system are discussed in a separate security document", wrote "example" in the paragraph introducing the decision tree. Removed reference draft-grinnemo-taps-he-03 and the sentence that referred to it.

WG -04: addressed comments from Theresa Enghardt and Tommy Pauly. As part of that, removed "TAPS" as a term everywhere (abstract, intro, ..).

WG -05: addressed comments from Spencer Dawkins.

WG -06: Fixed nits.

WG -07: Addressed Genart comments from Robert Sparks.

WG -08: Addressed one more Genart comment from Robert Sparks.

WG -09: Addressed comments from Mirja Kuehlewind, Alvaro Retana, Ben Campbell, Benjamin Kaduk and Eric Rescorla.

WG -10: Addressed comments from Benjamin Kaduk and Eric Rescorla.

WG -11: Addressed comments from Alissa Cooper.

Authors' Addresses

Michael Welzl
University of Oslo
PO Box 1080 Blindern
Oslo N-0316
Norway

Phone: +47 22 85 24 20
Email: michawe@ifi.uio.no

Stein Gjessing
University of Oslo
PO Box 1080 Blindern
Oslo N-0316
Norway

Phone: +47 22 85 24 44
Email: steing@ifi.uio.no

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 27, 2018

T. Pauly
Apple Inc.
October 24, 2017

Guidelines for Racing During Connection Establishment
draft-pauly-taps-guidelines-01

Abstract

Often, connections created across the Internet have multiple options of how to communicate: address families, specific IP addresses, network attachments, and application and transport protocols. This document describes how an implementation can race multiple options during connection establishment, and expose this functionality through an API.

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 <http://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 April 27, 2018.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of

the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
2. Terminology	3
2.1. Endpoint	3
2.2. Derived Endpoint	3
2.3. Path	3
2.4. Connection	4
3. Connection Establishment Overview	4
4. Structuring Options as a Tree	5
4.1. Branch Types	7
4.1.1. Derived Endpoints	7
4.1.2. Alternate Paths	7
4.1.3. Protocol Options	8
4.2. Branching Order-of-Operations	9
5. Connection Establishment Dynamics	10
5.1. Building the Tree	10
5.2. Racing Methods	11
5.2.1. Delayed Racing	11
5.2.2. Failover	12
5.3. Completing Establishment	12
5.3.1. Determining Successful Establishment	13
6. API Considerations	14
6.1. Handling 0-RTT Data	14
7. Security Considerations	15
8. IANA Considerations	16
9. Acknowledgments	16
10. Informative References	16
Author's Address	17

1. Introduction

Often, connections created across the Internet have multiple options of how to communicate: address families, specific IP addresses, network attachments, and application and transport protocols. If an application chooses to only attempt one of these options, it may fail to connect, or end up using a suboptimal path. If an application chooses to attempt one option after another, waiting for each to fail or time out, a user of the application may need to wait for a very long time before progress is made. And, if an application simultaneously attempts all options, it may unnecessarily consume significant local or network resources.

In order to solve this, applications can employ a method of racing their various connection establishment options. This approach is

commonly used for racing multiple IP address families, the algorithm for which is referred to as "Happy Eyeballs" [I-D.ietf-v6ops-rfc6555bis]. However, the approach can apply more generally.

This document describes how an implementation can race multiple options during connection establishment, and expose this functionality through an API.

2. Terminology

This document uses specific terminology when discussing connection establishment.

2.1. Endpoint

An identifier for a network service. Generally there is a concept of both a local and remote endpoint. Endpoints are the targets of network connections. If an endpoint of a given type cannot be directly used, it should be resolved into one or more endpoints of another type. Examples of endpoint types include:

- o IP address + port
- o Hostname + port
- o Service name + type + domain
- o URI

2.2. Derived Endpoint

A derived endpoint is an endpoint that is not the original target of an API client, but an endpoint created from the original endpoint through transformation or lookup. Derivation may take the form of hostname resolution into addresses, synthesis between address types, or changing to a different endpoint entirely based on a configuration requirement. For example, if a proxy server must be used for a connection, the endpoint that represents the proxy is a derived endpoint.

2.3. Path

A view of network properties that can be used to communicate to an endpoint from the current system. This is sometimes referred to as a Provisioning Domain (PvD) [RFC7556]. The path may include properties of the addresses and routes being used, the network interfaces being

used, and other metadata about the network learned from configuration or negotiation.

2.4. Connection

A flow of data between two endpoints. A connection is created with a target remote endpoint, and a set of parameters indicating client preferences for path selection and protocol options.

3. Connection Establishment Overview

The process of establishing a network connection begins when an application expresses intent to communicate with a remote endpoint (along with any constraints or requirements it may have on the connection). The process can be considered complete once there is at least one set of network protocols that have completed any required setup to the point that it can transmit and receive the application's data.

Looking more closely, connection establishment has three required steps that must be performed by some entity on a system:

1. Identifying the endpoint to which the connection should be established
2. Choosing which path or interface to use
3. Conducting the necessary set of protocol handshakes to establish the connection

The most simple example of this process might involve identifying the single IP address to which the application wishes to connect, using the system's current default interface or path, and starting a TCP handshake to establish a stream to the specified IP address. However, each step may also vary depending on the requirements of the connection: if the endpoint is defined as a hostname and port, then there may be multiple resolved addresses that are available; there may also be multiple interfaces or paths available, other than the default system interface; and some protocols may not need any transport handshake to be considered "established" (such as UDP), while other connections may utilize layered protocol handshakes, such as TLS over TCP.

Whenever an application has multiple options for connection establishment, it can view the set of all individual connection establishment options as a single, aggregate connection establishment. The aggregate set conceptually includes every valid combination of endpoints, paths, and protocols. As an example,

consider an application that initiates a TCP connection to a hostname + port endpoint, and has two valid interfaces available (Wi-Fi and LTE). The hostname resolves to a single IPv4 address on the Wi-Fi network, and resolves to the same IPv4 address on the LTE network, as well as a single IPv6 address. The aggregate set of connection establishment options can be viewed as follows:

```
Aggregate [Endpoint: www.example.com:80] [Interface: Any] [Protocol: TCP]
  |-> [Endpoint: 192.0.2.1:80] [Interface: Wi-Fi] [Protocol: TCP]
  |-> [Endpoint: 192.0.2.1:80] [Interface: LTE] [Protocol: TCP]
  |-> [Endpoint: 2001:DB8::1.80] [Interface: LTE] [Protocol: TCP]
```

Any one of these sub-entries on the aggregate connection attempt would satisfy the original application intent. The concern of this document is the algorithm defining which of these options to try, when, and in what order.

4. Structuring Options as a Tree

When an implementation responsible for connection establishment needs to consider multiple options, it SHOULD logically structure these options as a hierarchical tree. Each leaf node of the tree represents a single, coherent connection attempt, with an Endpoint, a Path, and a set of protocols that can directly negotiate and send data on the network. Each node in the tree that is not a leaf represents a connection attempt that is either underspecified, or else includes multiple distinct options. For example, when connecting on an IP network, a connection attempt to a hostname and port is underspecified, because the connection attempt requires a resolved IP address as its remote endpoint. In this case, the node represented by the connection attempt to the hostname is a parent node, with child nodes for each IP address. Similarly, an application that is allowed to connect using multiple interfaces will have a parent node of the tree for the decision between the paths, with a branch for each interface.

The example aggregate connection attempt above can be drawn as a tree by grouping the addresses resolved on the same interface into branches:

```

      ||
      +=====+
      | www.example.com:80/Any |
      +=====+
      //          \\\
+=====+          +=====+
| www.example.com:80/Wi-Fi | | www.example.com:80/LTE |
+=====+          +=====+
      ||          //          \\\
+=====+ +=====+ +=====+
| 192.0.2.1:80/Wi-Fi | | 192.0.2.1:80/LTE | | 2001:DB8::1.80/LTE |
+=====+ +=====+ +=====+

```

The rest of this document will use a notation scheme to represent this tree. The parent (or trunk) node of the tree will be represented by a single integer, such as "1". Each child of that node will have an integer that identifies it, from 1 to the number of children. That child node will be uniquely identified by concatenating its integer to it's parents identifier with a dot in between, such as "1.1" and "1.2". Each node will be summarized by a tuple of three elements: Endpoint, Path, and Protocol. The above example can now be written more succinctly as:

```

1 [www.example.com:80, Any, TCP]
  1.1 [www.example.com:80, Wi-Fi, TCP]
    1.1.1 [192.0.2.1:80, Wi-Fi, TCP]
  1.2 [www.example.com:80, LTE, TCP]
    1.2.1 [192.0.2.1:80, LTE, TCP]
    1.2.2 [2001:DB8::1.80, LTE, TCP]

```

When an application views this aggregate set of connection attempts as a single connection establishment, it only will use one of the leaf nodes to transfer data. Thus, when a single leaf node becomes ready to use, then the entire connection attempt is ready to use by the application. Another way to represent this is that every leaf node updates the state of its parent node when it becomes ready, until the trunk node of the tree is ready, which then notifies the application that the connection as a whole is ready to use.

A connection establishment tree may be degenerate, and only have a single leaf node, such as a connection attempt to an IP address over a single interface with a single protocol.

```

1 [192.0.2.1:80, Wi-Fi, TCP]

```

A parent node may also only have one child (or leaf) node, such as a when a hostname resolves to only a single IP address.


```
1 [www.example.com:80, Wi-Fi, TCP]
  1.1 [192.0.2.1:80, Wi-Fi, TCP]
```

4.1. Branch Types

There are three types of branching from a parent node into one or more child nodes. Any parent node of the tree **MUST** only use one type of branching.

4.1.1. Derived Endpoints

If a connection originally targets a single endpoint, there may be multiple endpoints of different types that can be derived from the original. The connection library should order the derived endpoints according to application preference and expected performance.

DNS hostname-to-address resolution is the most common method of endpoint derivation. When trying to connect to a hostname endpoint on a traditional IP network, the implementation **SHOULD** send DNS queries for both A (IPv4) and AAAA (IPv6) records if both are supported on the local link. The algorithm for ordering and racing these addresses **SHOULD** follow the recommendations in Happy Eyeballs [I-D.ietf-v6ops-rfc6555bis].

```
1 [www.example.com:80, Wi-Fi, TCP]
  1.1 [2001:DB8::1.80, Wi-Fi, TCP]
  1.2 [192.0.2.1:80, Wi-Fi, TCP]
  1.3 [2001:DB8::2.80, Wi-Fi, TCP]
  1.4 [2001:DB8::3.80, Wi-Fi, TCP]
```

DNS-Based Service Discovery can also provide an endpoint derivation step. When trying to connect to a named service, the client may discover one or more hostname and port pairs on the local network using multicast DNS. These hostnames should each be treated as a branch which can be attempted independently from other hostnames. Each of these hostnames may also resolve to one or more addresses, thus creating multiple layers of branching.

```
1 [term-printer._ipp._tcp.meeting.ietf.org, Wi-Fi, TCP]
  1.1 [term-printer.meeting.ietf.org:631, Wi-Fi, TCP]
    1.1.1 [31.133.160.18.631, Wi-Fi, TCP]
```

4.1.2. Alternate Paths

If a client has multiple network interfaces available to it, such as mobile client with both Wi-Fi and Cellular connectivity, it can attempt a connection over either interface. This represents a branch point in the connection establishment. Like with derived endpoints,

the interfaces should be ranked based on preference, system policy, and performance. Attempts should be started on one interface, and then on other interfaces successively after delays based on expected round-trip-time or other available metrics.

- 1 [192.0.2.1:80, Any, TCP]
 - 1.1 [192.0.2.1:80, Wi-Fi, TCP]
 - 1.2 [192.0.2.1:80, LTE, TCP]

This same approach applies to any situation in which the client is aware of multiple links or views of the network. Multiple Paths, each with a coherent set of addresses, routes, DNS server, and more, may share a single interface. A path may also represent a virtual interface service such as a Virtual Private Network (VPN).

The list of available paths should be constrained by any requirements or prohibitions the application sets, as well as system policy.

4.1.3. Protocol Options

Differences in possible protocol compositions and options can also provide a branching point in connection establishment. This allows clients to be resilient to situations in which a certain protocol is not functioning on a server or network.

This approach is commonly used for connections with optional proxy server configurations. A single connection may be allowed to use an HTTP-based proxy, a SOCKS-based proxy, or connect directly. These options should be ranked and attempted in succession.

- 1 [www.example.com:80, Any, HTTP/TCP]
 - 1.1 [192.0.2.8:80, Any, HTTP/HTTP Proxy/TCP]
 - 1.2 [192.0.2.7:10234, Any, HTTP/SOCKS/TCP]
 - 1.3 [www.example.com:80, Any, HTTP/TCP]
 - 1.3.1 [192.0.2.1:80, Any, HTTP/TCP]

This approach also allows a client to attempt different sets of application and transport protocols that may provide preferable characteristics when available. For example, the protocol options could involve QUIC [I-D.ietf-quic-transport] over UDP on one branch, and HTTP/2 [RFC7540] over TLS over TCP on the other:

- 1 [www.example.com:443, Any, Any HTTP]
 - 1.1 [www.example.com:443, Any, QUIC/UDP]
 - 1.1.1 [192.0.2.1:443, Any, QUIC/UDP]
 - 1.2 [www.example.com:443, Any, HTTP2/TLS/TCP]
 - 1.2.1 [192.0.2.1:443, Any, HTTP2/TLS/TCP]

Another example is racing SCTP with TCP:

```
1 [www.example.com:80, Any, Any Stream]
  1.1 [www.example.com:80, Any, SCTP]
    1.1.1 [192.0.2.1:80, Any, SCTP]
  1.2 [www.example.com:80, Any, TCP]
    1.2.1 [192.0.2.1:80, Any, TCP]
```

Implementations that support racing protocols and protocol options SHOULD maintain a history of which protocols and protocol options successfully established, on a per-network basis. This information can influence future racing decisions to prioritize or prune branches.

4.2. Branching Order-of-Operations

Branch types must occur in a specific order relative to one another to avoid creating leaf nodes with invalid or incompatible settings. In the example above, it would be invalid to branch for derived endpoints (the DNS results for `www.example.com`) before branching between interface paths, since usable DNS results on one network may not necessarily be the same as DNS results on another network due to local network entities, supported address families, or enterprise network configurations. Implementations must be careful to branch in an order that results in usable leaf nodes whenever there are multiple branch types that could be used from a single node.

The order of operations for branching, where lower numbers are acted upon first, SHOULD be:

1. Alternate Paths
2. Protocol Options
3. Derived Endpoints

Branching between paths is the first in the list because results across multiple interfaces are likely not related to one another: endpoint resolution may return different results, especially when using locally resolved host and service names, and which protocols are supported and preferred may differ across interfaces. Thus, if multiple paths are attempted, the overall connection can be seen as a race between the available paths or interfaces.

Protocol options are checked next in order. Whether or not a set of protocol, or protocol-specific options, can successfully connect is generally not dependent on which specific IP address is used. Furthermore, the protocol stacks being attempted may influence or

altogether change the endpoints being used. Adding a proxy to a connection's branch will change the endpoint to the proxy's IP address or hostname. Choosing an alternate protocol may also modify the ports that should be selected.

Branching for derived endpoints is the final step, and may have multiple layers of derivation or resolution, such as DNS service resolution and DNS hostname resolution.

5. Connection Establishment Dynamics

The primary goal of the connection establishment process is to successfully negotiate a protocol stack to an endpoint over an interface--to connect a single leaf node of the tree--with as little delay and as few unnecessary connections attempts as possible. Optimizing these two factors improves the user experience, while minimizing network load.

This section covers the dynamic aspect of connection establishment. While the tree described above is a useful conceptual and architectural model, an implementation does not know what the full tree may become up front, nor will many of the possible branches be used in the common case.

5.1. Building the Tree

The tree of options is built dynamically, out from the original trunk node. Any time that a connection attempt may be made directly to an endpoint without further derivation, and without needing to try alternate paths or protocol options that have not yet been covered by previous branches, the implementation SHOULD treat this as a leaf node and connect directly. Any time that an implementation chooses to branch between multiple options, it SHOULD determine a preferred order between the child nodes based on system policy, expected or historical performance, and application preference.

When multiple paths are available, and permitted by the system's policy, the implementation SHOULD branch between the various paths. The list SHOULD be sorted based on the system policies and routes (which often determine a "default" interface), preferences expressed by the application, and expected performance based on measured or advertised properties of each path.

When multiple protocol options are allowed by an application, and the system and implementation identify valid sets of protocols and protocol options, the implementation SHOULD branch between these sets. This list SHOULD be sorted based on application preference and

expected performance, generally measured in terms of latency and bandwidth.

An implementation will only branch to derive endpoints when necessary. This step involves the most external information, as endpoint derivation is often a process that requires fetching information from the network. Before branching, an implementation must first generate the list of derived endpoints. Once this list is sufficiently populated to continue, the implementation SHOULD sort the list based on preference and expected performance. When these derived endpoints are IP addresses, implementations SHOULD use the algorithm in [RFC6724] to sort the addresses. In cases where additional information can become available after the initial tree has been constructed, the implementation SHOULD update the tree to reflect new information and orderings if none of the leaf nodes are fully established.

5.2. Racing Methods

There are three different approaches to racing the attempts for different nodes of the connection establishment tree:

1. Immediate
2. Delayed
3. Failover

Each approach is appropriate in different use-cases and branch types. However, to avoid consuming unnecessary network resources, implementations SHOULD NOT use immediate racing as a default approach.

The timing algorithms for racing SHOULD remain independent across branches of the tree. Any timers or racing logic is isolated to a given parent node, and is not ordered precisely with regards to other children of other nodes.

5.2.1. Delayed Racing

Delayed racing can be used whenever a single node of the tree has multiple child nodes. Based on the order determined when building the tree, the first child node will be initiated immediately, followed by the next child node after some delay. Once that second child node is initiated, the third child node (if present) will begin after another delay, and so on until all child nodes have been initiated, or one of the child nodes successfully completes its negotiation.

Delayed racing attempts occur in parallel. Implementations SHOULD NOT terminate an earlier child connection attempt upon starting a secondary child.

The delay between starting child nodes SHOULD be based on the properties of the previously started child node. For example, if the first child represents an IP address with a known route, and the second child represents another IP address, the delay between starting the first and second IP addresses can be based on the expected retransmission cadence for the first child's connection (derived from historical round-trip-time). Alternatively, if the first child represents a branch on a Wi-Fi interface, and the second child represents a branch on an LTE interface, the delay should be based on the expected time in which the branch for the first interface would be able to establish a connection, based on link quality and historical round-trip-time.

Any delay SHOULD have a defined minimum and maximum value based on the branch type. Generally, branches between paths and protocols should have longer delays than branches between derived endpoints. The maximum delay should be considered with regards to how long a user is expected to wait for the connection to complete.

If a child node fails to connect before the delay timer has fired for the next child, the next child SHOULD be started immediately.

5.2.2. Failover

If an implementation or application has a strong preference for one branch over another, the branching node may choose to wait until one child has failed before starting the next. Failure of a leaf node is determined by its protocol negotiation failing or timing out; failure of a parent branching node is determined by all of its children failing.

An example in which failover is recommended is a race between a protocol stack that uses a proxy and a protocol stack that bypasses the proxy. Failover is useful in case the proxy is down or misconfigured, but any more aggressive type of racing may end up unnecessarily avoiding a proxy that was preferred by policy.

5.3. Completing Establishment

The process of connection establishment completes when one leaf node of the tree has completed negotiation with the remote endpoint successfully, or else all nodes of the tree have failed to connect. The first leaf node to complete its connection is then used by the application to send and receive data.

It is useful to process success and failure throughout the tree by child nodes reporting to their parent nodes (towards the trunk of the tree). For example, in the following case, if 1.1.1 fails to connect, it reports the failure to 1.1. Since 1.1 has no other child nodes, it also has failed and reports that failure to 1. Because 1.2 has not yet failed, 1 is not considered to have failed. Since 1.2 has not yet started, it is started and the process continues. Similarly, if 1.1.1 successfully connects, then it marks 1.1 as connected, which propagates to the trunk node 1. At this point, the connection as a whole is considered to be successfully connected and ready to process application data

```
1 [www.example.com:80, Any, TCP]
  1.1 [www.example.com:80, Wi-Fi, TCP]
    1.1.1 [192.0.2.1:80, Wi-Fi, TCP]
    1.2 [www.example.com:80, LTE, TCP]
  ...
```

If a leaf node has successfully completed its connection, all other attempts SHOULD be made ineligible for use by the application for the original request. New connection attempts that involve transmitting data on the network SHOULD NOT be started after another leaf node has completed successfully, as the connection as a whole has been established. An implementation MAY choose to let certain handshakes and negotiations complete in order to gather metrics to influence future connections. Similarly, an implementation MAY choose to hold onto fully established leaf nodes that were not the first to establish for use in future connections, but this approach is not recommended since those attempts were slower to connect and may exhibit less desirable properties.

5.3.1. Determining Successful Establishment

Implementations may select the criteria by which a leaf node is considered to be successfully connected differently on a per-protocol basis. If the only protocol being used is a transport protocol with a clear handshake, like TCP, then the obvious choice is to declare that node "connected" when the last packet of the three-way handshake has been received. If the only protocol being used is an "unconnected" protocol, like UDP, the implementation may consider the node fully "connected" the moment it determines a route is present, before sending any packets on the network.

For protocol stacks with multiple handshakes, the decision becomes more nuanced. If the protocol stack involves both TLS and TCP, an implementation MAY determine that a leaf node is connected after the TCP handshake is complete, or it MAY wait for the TLS handshake to complete as well. The benefit of declaring completion when the TCP

handshake finishes, and thus stopping the race for other branches of the tree, is that there will be less burden on the network from other connection attempts. On the other hand, by waiting until the TLS handshake is complete, an implementation avoids the scenario in which a TCP handshake completes quickly, but TLS negotiation is either very slow or fails altogether in particular network conditions or to a particular endpoint.

6. API Considerations

In general, the internal states and nodes of racing connection establishment do not need to be exposed to applications. Instead, this process SHOULD be treated as an abstraction of a single, aggregate connection establishment behind an API. This places some requirements on the API, including:

- o The API must allow the application to specify an un-resolved endpoint as the remote side of the connection, such as a URI or hostname + port. The application also should be able to provide constraints on path selection and protocol features.
- o Any read or write operations cannot take effect until one leaf node has been chosen as the connected node. The API needs to either expose asynchronous reads and writes, or else prohibit reads and writes until the connection is established.
- o The action of starting or initiating the connection may involve many network-bound operations, so this operation SHOULD be asynchronous.
- o Properties of the connection, such as the remote and local addresses, the interface used, and the protocols used, may not be queryable until the connection is established.

6.1. Handling 0-RTT Data

Several protocols allow sending higher-level protocol or application data within the first packet of their protocol establishment, such as TCP Fast Open [RFC7413] and TLS 1.3 [I-D.ietf-tls-tls13]. This approach is referred to as sending Zero-RTT (0-RTT) data. This is a desirable property, but poses challenges to an implementation that uses racing during connection establishment.

If the application has 0-RTT data to send in any protocol handshakes, it needs to provide this data before the handshakes have begun. When racing, this means that the data SHOULD be provided before the process of connection establishment has begun. If the API allows the application to send 0-RTT data, it MUST provide an interface that

identifies this data as idempotent data. In general, 0-RTT data may be replayed (for example, if a TCP SYN contains data, and the SYN is retransmitted, the data will be retransmitted as well), but racing means that different leaf nodes have the opportunity to send the same data independently. If data is truly idempotent, this should be permissible.

Once the application has provided its 0-RTT data, an implementation SHOULD keep a copy of this data and provide it to each new leaf node that is started and for which a 0-RTT protocol is being used.

It is also possible that protocol stacks within a particular leaf node use 0-RTT handshakes without any idempotent application data. For example, TCP Fast Open could use a Client Hello from a TLS as its 0-RTT data, shortening the cumulative handshake time.

0-RTT handshakes often rely on previous state, such as TCP Fast Open cookies, previously established TLS tickets, or out-of-band distributed pre-shared keys (PSKs). Implementations should be aware of security concerns around using these tokens across multiple addresses or paths when racing. In the case of TLS, any given ticket or PSK SHOULD only be used on one leaf node. If implementations have multiple tickets available from a previous connection, each leaf node attempt MUST use a different ticket. In effect, each leaf node will send the same early application data, yet encoded (encrypted) differently on the wire.

7. Security Considerations

See Section 6.1 for security considerations around racing with 0-RTT data.

An attacker that knows a particular device is racing several options during connection establishment may be able to block packets for the first connection attempt, thus inducing the device to fall back to a secondary attempt. This is a problem if the secondary attempts have worse security properties that enable further attacks. Implementations should ensure that all options have equivalent security properties to avoid incentivizing attacks.

Since results from the network can determine how a connection attempt tree is built, such as when DNS returns a list of resolved endpoints, it is possible for the network to cause an implementation to consume significant on-device resources. Implementations SHOULD limit the maximum amount of state allowed for any given node, including the number of child nodes, especially when the state is based on results from the network.

8. IANA Considerations

This document has no request to IANA.

9. Acknowledgments

Thanks to Josh Graessley and Stuart Cheshire for their help in the design of the original implementation of Happy Eyeballs for Apple that began this work.

10. Informative References

[I-D.ietf-quick-transport]

Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quick-transport-07 (work in progress), October 2017.

[I-D.ietf-tls-tls13]

Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-21 (work in progress), July 2017.

[I-D.ietf-v6ops-rfc6555bis]

Schinazi, D. and T. Pauly, "Happy Eyeballs Version 2: Better Connectivity Using Concurrency", draft-ietf-v6ops-rfc6555bis-06 (work in progress), October 2017.

[RFC6724] Thaler, D., Ed., Draves, R., Matsumoto, A., and T. Chown, "Default Address Selection for Internet Protocol Version 6 (IPv6)", RFC 6724, DOI 10.17487/RFC6724, September 2012, <<https://www.rfc-editor.org/info/rfc6724>>.

[RFC7413] Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP Fast Open", RFC 7413, DOI 10.17487/RFC7413, December 2014, <<https://www.rfc-editor.org/info/rfc7413>>.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <<https://www.rfc-editor.org/info/rfc7540>>.

[RFC7556] Anipko, D., Ed., "Multiple Provisioning Domain Architecture", RFC 7556, DOI 10.17487/RFC7556, June 2015, <<https://www.rfc-editor.org/info/rfc7556>>.

Author's Address

Tommy Pauly
Apple Inc.
1 Infinite Loop
Cupertino, California 95014
United States of America

Email: tpauly@apple.com

TAPS Working Group
Internet-Draft
Intended status: Experimental
Expires: April 30, 2018

P. Tiesel
T. Enhardt
A. Feldmann
TU Berlin
October 27, 2017

Socket Intents
draft-tiesel-taps-socketintents-01

Abstract

This document outlines Socket Intents, a concept that allows applications to share their knowledge about upcoming communication and express their performance preferences in a generic, intuitive and, portable way. Using Socket Intents, an application can express what it knows, assumes, expects, or wants regarding its network communication. The information provided by Socket Intents can be used by the network stack to optimize communication in a best-effort way.

Socket Intent can be used to stem against the complexity of exploiting transport diversity, e.g., to automate the choice among multiple paths, provisioning domains or protocols. By shifting this complexity from the application developer to the operating system, it enables the use of these transport features to a wider range of applications.

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 April 30, 2018.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Conventions and Definitions	3
2. Introduction	3
3. Problem Statement	3
4. Socket Intents Concept	4
4.1. Interactions between Socket Intents and QoS	5
5. Socket Intent Types	5
6. Initial Socket Intent Types	6
6.1. Traffic Category	6
6.2. Size to be Sent / Received	7
6.3. Duration	7
6.4. Stream Bitrate Sent / Received	7
6.5. Burstiness	7
6.6. Timeliness	8
6.7. Disruption Resilience	9
6.8. Cost Preferences	9
7. Implementation Guidelines	10
8. Security Considerations	10
8.1. Performance Degradation Attacks	10
8.2. Information Leakage	11
9. IANA Considerations	11
10. Publications History	11
11. Acknowledgements	11
12. References	11
12.1. Normative References	11
12.2. Informative References	12
Appendix A. Usage examples	13
A.1. Example 1	13
A.2. Example 2	13
A.3. Example 3	14
Appendix B. Changes	14
B.1. Since -00	14

Authors' Addresses	15
------------------------------	----

1. Conventions and Definitions

The words "MUST", "MUST NOT", "SHALL", "SHALL NOT", "SHOULD", and "MAY" are used in this document. It's not shouting; when these words are capitalized, they have a special meaning as defined in [RFC2119].

Association Set, Association, Stream, or Message are used as defined in [I-D.tiesel-taps-communitgrany].

2. Introduction

Despite recent advances in the transport area, the adaption of new transport protocols and transport protocol features is slow. In practice, this only happens in limited fields as Web browsers or within datacenters. The same problem occurs for taking advantage of paths or provisioning domains (PvDs). In both cases, the benefits of the new transport diversity come at the cost of an increased complexity that has to be mastered by the application programmer.

To enable transport features like TCP fast open [RFC7413] or to control how MPTCP [RFC6824] creates subflows requires specialized APIs. These APIs are not part of the standard socket API, usually not portable, and not available in many programming languages. Using them often requires profound knowledge of the transport protocol internals.

To use multiple paths, applications usually have to use their own heuristics to select which paths, provisioning domains, or access network to use. Choosing the right path is difficult as their characteristics differ, e.g., regarding performance. Obtaining the necessary information is difficult since it may require special privileges and non-portable APIs.

In all cases mentioned above, an application that wants to take advantage of the available transport diversity is faced with substantially higher complexity regarding network APIs and networking code.

3. Problem Statement

Application programmers opening a communication channel typically know how this channel will be used. There is more information available than the protocol and destination address needed to establish a communication channel: An application developer has an intuition about many aspects of an upcoming communication. These intuition may include:

preferences: whether to optimize for bandwidth, latency, or cost

characteristics: expected packet rates, byte rates or how many bytes will be sent or received.

expectations: towards path availability or packet loss

resiliences: whether the application can gracefully handle certain error cases

These preferences, expectations and other information known about the upcoming communication should be expressible in an intuitive, generic way, that is independent of the network and transport protocol. Its representation should be independent of the actual API used for network communication and should be expressible in whatever API available, e.g., as socket options for BSD sockets or as part of the address resolution configuration for Post Sockets [I-D.trammell-taps-post-sockets].

Socket Intents should enable the OS to adjust the communication channel according to the application's intents in a best-effort fashion: They should provide the information needed to automatically enabling transport features the application can benefit from or help choosing the most suitable (combination) of paths based on the properties of the access networks or PvD (see [RFC7556], Section 6.2) available. The actual implementation is not part of the Socket Intents concept, it is left to an OS policy that may choose the best transport protocol, default parameters and PvDs available and may also try to further optimize wherever possible.

4. Socket Intents Concept

Socket Intents are pieces of information that allow an application to express what they know about the application's communication. They indicate what the application wants to achieve, knows, or assumes in general, intuitive terms. An application can use them to annotate the characteristics, preferences, and intentions it associates with each communication unit. Depending on the API used, Socket Intents can be used on a per Association Set, Association, Stream or, Message level.

Socket Intents are optional information that can be considered in a best-effort manner. Socket Intents do not include requirements, such as reliable in-order delivery. Typical examples include desired transport characteristics, e.g., low delay, high throughput, or minimal cost, as well as expected application behavior, e.g., will send 500 bytes. As this information captures the intents of an

applications and passes them along with the communication socket, we call these pieces of information Socket Intents.

Applications have an incentive to specify their intents as accurately as possible to take advantage of the most suitable existing resources. Applications are expected to selfishly specify their preferences. It is up to the OS's policy to prevent commitment of excessive resources.

4.1. Interactions between Socket Intents and QoS

Socket Intents are not QoS labels, but have an orthogonal meaning. While the purpose of QoS is to specify what an application requires, Socket Intents are used to specify what an application knows or prefers. Therefore,

- o Socket Intents SHALL be purely advisory.
- o Socket Intents MUST NOT be used to derive IntServ / RSVP style guarantees.
- o Socket Intents SHOULD be taken into account on a best-effort basis and MAY be used to derive DiffServ Service Classes as described in [RFC4594].

5. Socket Intent Types

Socket Intents are structured as key-value-pairs.

The key, called short name, specifies the Socket Intent type. It is identified by a string of the lower-case characters [a-z], numbers [0-9] and the separator "-".

The namespace for the short names is partitioned as follows:

- o All Socket Intent type not starting with "x-" or "y-" are managed by an IANA registry. The assignment of new types requires an RFC or expert review (TO BE DECIDED).
- o Socket Intent type starting with "x-" are for experimental use.
- o Private or vendor specific Socket Intent type MUST start with "y-[vendor]-".

Values can be represented as Enum, Int, Float, ASCII-String [RFC0020] or a sequence of the aforementioned data types. Implementations determine how these types are represented on the respective platform.

The data type for the individual Socket Intents are determined by the document defining the Socket Intent and MUST NOT be changed by an implementation. For Enum data types, a list of valid values MUST be provided by the document specifying that intent as well as a default value that is equivalent to not specifying this intent.

6. Initial Socket Intent Types

The following sections contain a list of Socket Intent types and their possible values. Recommended default values for Enum values are marked with an asterisk (*) behind the level name.

6.1. Traffic Category

The Traffic Category describes the dominating traffic pattern of the respective communication unit expected by the application.

Short name: category

Applicability: Association Set, Association, Stream

Data type: Enum

Level	Description
query	Single request / response style workload, latency bound
control	Long lasting low bandwidth control channel, not bandwidth bound
stream	Stream of bytes/messages with steady data rate
bulk	Bulk transfer of large messages, presumably bandwidth bound
mixed*	Don't know or none of the above

Note: Most categories suggest the use of other intents to further describe the traffic pattern anticipated, e.g., the bulk category suggesting the use of the Size to be Sent intent or the stream category suggesting the Stream Bitrate and Duration intents.

6.2. Size to be Sent / Received

This Intent is used to communicate the expected size of a transfer.

Short name: `send_size / recv_size`

Applicability: Association Set, Association, Stream, Message

Data type: Int (bytes)

6.3. Duration

This Intent is used to communicate the expected lifetime of the respective communication unit.

Short name: `duration`

Applicability: Association Set, Association, Stream

Data type: Int (msec)

6.4. Stream Bitrate Sent / Received

This Intent is used to communicate the bitrate of the respective communication unit.

Short name: `send_bitrate / recv_bitrate`

Applicability: Association Set, Association, Stream

Data type: Int (bits/sec)

6.5. Burstiness

This Intent describes the anticipated burst characteristics of the traffic for this communication unit. It expresses how the traffic sent by the application is expected to vary over time, and, consequently, how long sequences of consecutively sent packets will be. Note that the actual burst characteristics of the traffic at the receiver side will depend on the network.

This Intent can provide hints to the application on what the resource usage pattern for this communication unit will look like, which can be useful for balancing the requirements of different application.

Short name: `bursts`

Applicability: Association Set, Association, Stream

Data type: Enum

Level	Description
no_bursts	Application sends traffic at a constant rate
regular_bursts	Application sends bursts of traffic periodically
random_bursts	Application sends bursts of traffic irregularly
bulk	Application sends a bulk of traffic
mixed*	Don't know or none of the above

6.6. Timeliness

This Intent describes the desired delay characteristics for this communication unit. It provides hints for the OS whether to optimize for low delay or for other criteria. There are no hard requirements or implied guarantees on whether these requirements can actually be satisfied.

Short name: timeliness

Applicability: Association Set, Association, Stream, Message

Data type: Enum

Level	Description
stream	Delay and packet delay variation should be kept as low as possible
interactive	Delay should be kept as low as possible, but some variation is tolerable
transfer*	Delay and packet delay variation should be reasonable, but are not critical
background	Delay and packet delay variation is no concern

6.7. Disruption Resilience

This Intent describes how an application deals with disruption of its communication, e.g. connection loss. It communicates how well the application can recover from such disturbance and can have implications on how many resources the OS should allocate to failover techniques for this particular communication unit.

Short name: resilience

Applicability: Association Set, Association, Stream, Message

Data type: Enum

Level	Description
sensitive	Disruptions result in application failure, disrupting user experience
recoverable*	Disruptions are inconvenient for the application, but can be recovered from
resilient	Disruptions have minimal impact for the application

6.8. Cost Preferences

This describes the Intents of an Application towards costs caused by the respective communication unit. It should guide the OS how to handle cost vs. performance and reliability tradeoffs.

Short name: cost

Applicability: Association Set, Association, Stream, Message

Data type: Enum

Level	Description
no_expense	Avoid expensive transports and consider failing otherwise
optimize_cost	Prefer inexpensive transports and accept service degradation
balance_cost*	Do not bias balancing cost and other criteria
ignore_cost	Ignore cost, choose transport solely based on other criteria

Note: the "no_expense" level implicitly asks the OS to fail communication attempts if no inexpensive transports are available.

Application developers MUST be aware that this also no hard requirement and can be ignored or overridden by the OS policy.

7. Implementation Guidelines

Implementations faced with unknown Socket Intent types SHOULD ignore these intents for forward compatibility. The API MAY include a parameter to change this behavior and make specifying unknown Socket Intent types return an error.

Invalid values SHOULD return an error to the application.

For debugging purposes, implementations SHOULD allow to enumerate the Socket Intents that are understood by the implementation. They MAY expose which of the Socket Intents were considered by the implementation.

8. Security Considerations

8.1. Performance Degradation Attacks

We assume that applications specify their preferences in a selfish, but not malicious way and that it is up to the OS to find a compromise between demands.

A malicious application could confuse the OS in a way that leads to scheduling traffic with certain Intents on a more expensive interface, penalizing this traffic, or even rejecting it. The attack vector added by this is negligible: As the malicious application

could also generate the traffic it claims to intend, it already has a much more powerful attack vector.

As a mitigation, the OS could monitor and compare the intents specified with the traffic actually generated and notify the user if the usage of Socket Intents is unusual or defective.

8.2. Information Leakage

Varying the transport or IP layer parameters of packets belonging to different Streams or Messages multiplexed in the same encrypted association might enable an attacker to gain some ground truth about the shares of different kinds of traffic. As this might also be implied by packet timings, application developers might weight the small additional information disclosure against the possible performance gains. Using Socket Intents on Association level can be considered safe.

9. IANA Considerations

The Socket Intents type namespace SHOULD be managed by the IANA registry. Details conforming to [RFC5226] are laid out in Section 5, the initial types for the registry are described in Section 6.

10. Publications History

- o The original idea of Socket Intents was published in [CoNEXT2013].
- o A performance study "Socket Intents: OS Support for Using Multiple Access Networks and its Benefits for Web Browsing" is under submission.

11. Acknowledgements

This work has been supported by Leibniz Prize project funds of DFG - German Research Foundation: Gottfried Wilhelm Leibniz-Preis 2011 (FKZ FE 570/4-1).

12. References

12.1. Normative References

- [RFC0020] Cerf, V., "ASCII format for network interchange", STD 80, RFC 20, DOI 10.17487/RFC0020, October 1969, <<https://www.rfc-editor.org/info/rfc20>>.

- [RFC2119] 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>>.
- [RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", RFC 5226, DOI 10.17487/RFC5226, May 2008, <<https://www.rfc-editor.org/info/rfc5226>>.

12.2. Informative References

- [CoNEXT2013] Schmidt, P., Enghardt, T., Khalili, R., and A. Feldmann, "Socket intents", Proceedings of the ninth ACM conference on Emerging networking experiments and technologies - CoNEXT '13, DOI 10.1145/2535372.2535405, 2013.
- [DASH] International Organization for Standardization, "Dynamic adaptive streaming over HTTP (DASH) - Part 1: Media presentation description and segment formats", Standard ISO/IEC 23009-1:2014, June 2011, <<https://www.iso.org/standard/65274.html>>.
- [I-D.pauly-taps-guidelines] Pauly, T., "Guidelines for Racing During Connection Establishment", draft-pauly-taps-guidelines-01 (work in progress), October 2017.
- [I-D.tiesel-taps-communitgrany] Tiesel, P. and T. Enghardt, "Communication Units Granularity Considerations for Multi-Path Aware Transport Selection", draft-tiesel-taps-communitgrany-01 (work in progress), October 2017.
- [I-D.trammell-taps-post-sockets] Trammell, B., Perkins, C., Pauly, T., Kuehlewind, M., and C. Wood, "Post Sockets, An Abstract Programming Interface for the Transport Layer", draft-trammell-taps-post-sockets-03 (work in progress), October 2017.
- [RFC4594] Babiarz, J., Chan, K., and F. Baker, "Configuration Guidelines for DiffServ Service Classes", RFC 4594, DOI 10.17487/RFC4594, August 2006, <<https://www.rfc-editor.org/info/rfc4594>>.

- [RFC4960] Stewart, R., Ed., "Stream Control Transmission Protocol", RFC 4960, DOI 10.17487/RFC4960, September 2007, <<https://www.rfc-editor.org/info/rfc4960>>.
- [RFC6824] Ford, A., Raiciu, C., Handley, M., and O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses", RFC 6824, DOI 10.17487/RFC6824, January 2013, <<https://www.rfc-editor.org/info/rfc6824>>.
- [RFC7413] Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP Fast Open", RFC 7413, DOI 10.17487/RFC7413, December 2014, <<https://www.rfc-editor.org/info/rfc7413>>.
- [RFC7556] Anipko, D., Ed., "Multiple Provisioning Domain Architecture", RFC 7556, DOI 10.17487/RFC7556, June 2015, <<https://www.rfc-editor.org/info/rfc7556>>.

Appendix A. Usage examples

A.1. Example 1

Consider a cellphone performing an OS upgrade. This process usually implies downloading a large file. This is a bulk transfer for which the application may already know the file size. Timing is typically noncritical and the data can be downloaded as background traffic with minimal cost and power overhead. It would not hurt if the TCP connection was closed during the transfer as the download can be continued.

For this case, the application should set the "Traffic Category" to "bulk", "Timeliness" to "background", and "Application Resilience" to "resilient". In addition, "Message Size to be Received" can be provided. Finally, the application may set the the "Cost Preferences" to "no_expense".

The OS can use this information and therefore may schedule this transfer on a flaky but not traffic-billed WiFi link and may reject the connection attempt if no cheap access link is available.

A.2. Example 2

Consider a user watching non-live video content using MPEG-DASH [DASH]. This usually means fetching a stream of video chunks. The application should know the size of each chunk and may know the bitrate and the duration of each chunk and the whole video. Disconnection of the TCP connection should be avoided because that might have an effect that is visible to the user.

For this case, the application should set the "Traffic Category" to "stream", the "Timeliness" to "stream", and "Application Resilience" to "sensitive". It may also provide the "Stream Bitrate Received" and "Duration" expected. Finally, the application may set the "Cost Preferences" to "balance_cost".

The OS can use this information and, e.g., use MPTCP [RFC6824] if available to schedule the traffic on the cheaper link (e.g., WiFi) while establishing an additional subflow over an expensive link (e.g., LTE). If the desired bandwidth cannot be matched by the cheaper link, the more expensive link can be added to satisfy the desired bandwidth.

If the application would set the "Cost Preferences" to "optimize_cost", the OS would not schedule traffic on the second subflow and the application would reduce the video quality to adapt to the available data rate.

A.3. Example 3

Consider a user managing a remote machine via SSH. This usually involves at least one long-lived console session and possibly file transfers using SCP or rsync multiplexed on the same association (e.g. TCP connection).

For the packets sent for the console session, the application can set the "Traffic Category" to "control", the "Burstiness" to "random bursts", the timeliness to "interactive" and the resilience to "sensitive". For the packets of the file transfers, SSH may set both, the "Traffic Category" and "Burstiness" to "bulk". It may also know the size of the transfer and therefore sets "Message Size to be Sent" or "Message Size to be Received".

Assuming there are transport opportunities supporting multiple streams in a single association (e.g. SCPT [RFC4960]), the OS can use this information to schedule the streams over different links to meet their requirements (latency vs. bandwidth). In case the OS has to use TCP, it can still optimize by disabling TCP Nagle Algorithm for console session related transmissions.

Appendix B. Changes

B.1. Since -00

- o Updates on Terminology (Object -> Message, Flow -> Association)
- o More detailed Socket Intent Types specification

- o Added implementation guidelines
- o Many clarifications
- o Fixed Authors and affiliations

Authors' Addresses

Philipp S. Tiesel
TU Berlin
Marchstr. 23
Berlin
Germany

Email: philipp@inet.tu-berlin.de

Theresa Enghardt
TU Berlin
Marchstr. 23
Berlin
Germany

Email: theresa@inet.tu-berlin.de

Anja Feldmann
TU Berlin
Marchstr. 23
Berlin
Germany

Email: anja@inet.tu-berlin.de

TAPS Working Group
Internet-Draft
Intended status: Informational
Expires: April 30, 2018

B. Trammell
ETH Zurich
C. Perkins
University of Glasgow
T. Pauly
Apple Inc.
M. Kuehlewind
ETH Zurich
C. Wood
Apple Inc.
October 27, 2017

Post Sockets, An Abstract Programming Interface for the Transport Layer
draft-trammell-taps-post-sockets-03

Abstract

This document describes Post Sockets, an asynchronous abstract programming interface for the atomic transmission of messages in an inherently multipath environment. Post replaces connections with long-lived associations between endpoints, with the possibility to cache cryptographic state in order to reduce amortized connection latency. We present this abstract interface as an illustration of what is possible with present developments in transport protocols when freed from the strictures of the current sockets API.

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 April 30, 2018.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	3
2. Abstractions and Terminology	5
2.1. Message Carrier	6
2.2. Message	8
2.3. Association	11
2.4. Remote	11
2.5. Local	12
2.6. Configuration	12
2.7. Transient	13
2.8. Path	14
3. Abstract Programming Interface	15
3.1. Example Connection Patterns	16
3.1.1. Client-Server	16
3.1.2. Client-Server with Happy Eyeballs and 0-RTT establishment	18
3.1.3. Peer to Peer with Network Address Translation	18
3.1.4. Multicast Receiver	18
3.1.5. Association Bootstrapping	19
3.2. API Dynamics	20
4. Implementation Considerations	22
4.1. Protocol Stack Instance (PSI)	23
4.2. Message Framing, Parsing, and Serialization	24
4.3. Message Size Limitations	25
4.4. Back-pressure	25
4.5. Associations, Transients, Racing, and Rendezvous	26
5. Acknowledgments	28
6. References	28
6.1. Normative References	28
6.2. Informative References	28
Appendix A. Open Issues	30
Authors' Addresses	30

1. Introduction

The BSD Unix Sockets API's SOCK_STREAM abstraction, by bringing network sockets into the UNIX programming model, allowing anyone who knew how to write programs that dealt with sequential-access files to also write network applications, was a revolution in simplicity. It would not be an overstatement to say that this simple API is the reason the Internet won the protocol wars of the 1980s. SOCK_STREAM is tied to the Transmission Control Protocol (TCP), specified in 1981 [RFC0793]. TCP has scaled remarkably well over the past three and a half decades, but its total ubiquity has hidden an uncomfortable fact: the network is not really a file, and stream abstractions are too simplistic for many modern application programming models.

In the meantime, the nature of Internet access, and the variety of Internet transport protocols, is evolving. The challenges that new protocols and access paradigms present to the sockets API and to programming models based on them inspire the design elements of a new approach.

Many end-user devices are connected to the Internet via multiple interfaces, which suggests it is time to promote the paths by which two endpoints are connected to each other to a first-order object. While implicit multipath communication is available for these multihomed nodes in the present Internet architecture with the Multipath TCP extension (MPTCP) [RFC6824], MPTCP was specifically designed to hide multipath communication from the application for purposes of compatibility. Since many multihomed nodes are connected to the Internet through access paths with widely different properties with respect to bandwidth, latency and cost, adding explicit path control to MPTCP's API would be useful in many situations.

Another trend straining the traditional layering of the transport stack associated with the SOCK_STREAM interface is the widespread interest in ubiquitous deployment of encryption to guarantee confidentiality, authenticity, and integrity, in the face of pervasive surveillance [RFC7258]. Layering the most widely deployed encryption technology, Transport Layer Security (TLS), strictly atop TCP (i.e., via a TLS library such as OpenSSL that uses the sockets API) requires the encryption-layer handshake to happen after the transport-layer handshake, which increases connection setup latency on the order of one or two round-trip times, an unacceptable delay for many applications. Integrating cryptographic state setup and maintenance into the path abstraction naturally complements efforts in new protocols (e.g. QUIC [I-D.ietf-quic-transport]) to mitigate this strict layering.

To meet these challenges, we present the Post-Sockets Application Programming Interface (API), described in detail in this work. Post is designed to be language, transport protocol, and architecture independent, allowing applications to be written to a common abstract interface, easily ported among different platforms, and used even in environments where transport protocol selection may be done dynamically, as proposed in the IETF's Transport Services working group.

Post replaces the traditional SOCK_STREAM abstraction with a Message abstraction, which can be seen as a generalization of the Stream Control Transmission Protocol's [RFC4960] SOCK_SEQPACKET service. Messages are sent and received on Carriers, which logically group Messages for transmission and reception. For backward compatibility, bidirectional byte stream protocols are represented as a pair of Messages, one in each direction, that can only be marked complete when the sending peer has finished transmitting data.

Post replaces the notions of a socket address and connected socket with an Association with a remote endpoint via set of Paths. Implementation and wire format for transport protocol(s) implementing the Post API are explicitly out of scope for this work; these abstractions need not map directly to implementation-level concepts, and indeed with various amounts of shimming and glue could be implemented with varying success atop any sufficiently flexible transport protocol.

The key features of Post as compared with the existing sockets API are:

- o Explicit Message orientation, with framing and atomicity guarantees for Message transmission.
- o Asynchronous reception, allowing all receiver-side interactions to be event-driven.
- o Explicit support for multistreaming and multipath transport protocols and network architectures.
- o Long-lived Associations, whose lifetimes may not be bound to underlying transport connections. This allows associations to cache state and cryptographic key material to enable fast resumption of communication, and for the implementation of the API to explicitly take care of connection establishment mechanics such as connection racing [RFC6555] and peer-to-peer rendezvous [RFC5245].

- o Transport protocol stack independence, allowing applications to be written in terms of the semantics best for the application's own design, separate from the protocol(s) used on the wire to achieve them. This enables applications written to a single API to make use of transport protocols in terms of the features they provide, as in [I-D.ietf-taps-transports].

This work is the synthesis of many years of Internet transport protocol research and development. It is inspired by concepts from the Stream Control Transmission Protocol (SCTP) [RFC4960], TCP Minion [I-D.iyengar-minion-protocol], and MinimaLT [MinimaLT], among other transport protocol modernization efforts. We present Post as an illustration of what is possible with present developments in transport protocols when freed from the strictures of the current sockets API. While much of the work for building parts of the protocols needed to implement Post are already ongoing in other IETF working groups (e.g. MPTCP, QUIC, TLS), we argue that an abstract programming interface unifying access all these efforts is necessary to fully exploit their potential.

2. Abstractions and Terminology

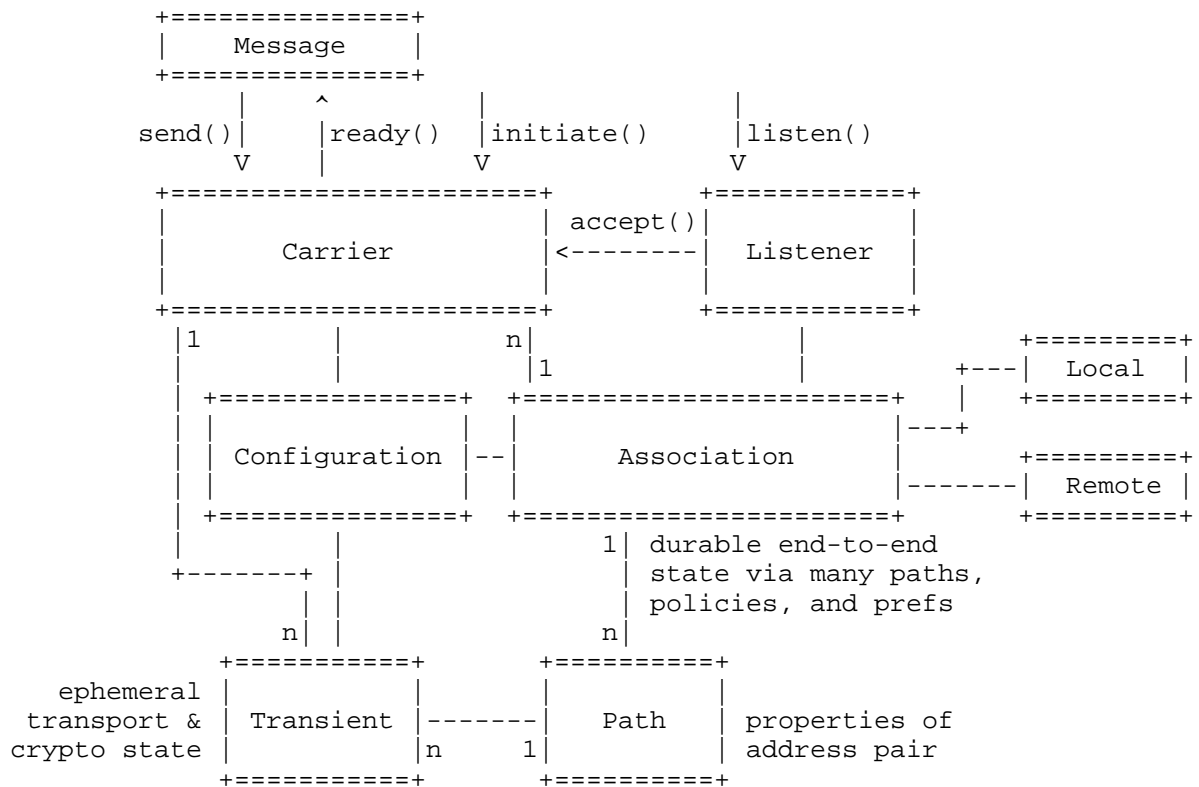


Figure 1: Abstractions and relationships in Post Sockets

Post is based on a small set of abstractions, centered around a Message Carrier as the entry point for an application to the networking API. The relationships among them are shown in Figure Figure 1 and detailed in this section.

2.1. Message Carrier

A Message Carrier (or simply Carrier) is a transport protocol stack-independent interface for sending and receiving messages between an application and a remote endpoint; it is roughly analogous to a socket in the present sockets API.

Sending a Message over a Carrier is driven by the application, while receipt is driven by the arrival of the last packet that allows the Message to be assembled, decrypted, and passed to the application. Receipt is therefore asynchronous; given the different models for asynchronous I/O and concurrency supported by different platforms, it

may be implemented in any number of ways. The abstract API provides only for a way for the application to register how it wants to handle incoming messages.

All the Messages sent to a Carrier will be received on the corresponding Carrier at the remote endpoint, though not necessarily reliably or in order, depending on Message properties and the underlying transport protocol stack.

A Carrier that is backed by current transport protocol stack state (such as a TCP connection; see Section 2.7) is said to be "active": messages can be sent and received over it. A Carrier can also be "dormant": there is long-term state associated with it (via the underlying Association; see Section 2.3), and it may be able to be reactivated, but messages cannot be sent and received immediately. Carriers become dormant when the underlying transport protocol stack determines that an underlying connection has been lost and there is insufficient state in the Association to re-establish it (e.g., in the case of a server-side Carrier where the client's address has changed unexpectedly). Passive close can be handled by the application via an event on the carrier. Attempting to use a carrier after passive close results in an error.

If supported by the underlying transport protocol stack, a Carrier may be forked: creating a new Carrier associated with a new Carrier at the same remote endpoint. The semantics of the usage of multiple Carriers based on the same Association are application-specific. When a Carrier is forked, its corresponding Carrier at the remote endpoint receives a fork request, which it must accept in order to fully establish the new carrier. Multiple Carriers between endpoints are implemented differently by different transport protocol stacks, either using multiple separate transport-layer connections, or using multiple streams of multistreaming transport protocols.

To exchange messages with a given remote endpoint, an application may initiate a Carrier given its remote (see Section 2.4) and local (see Section 2.5) identities; this is an equivalent to an active open. There are four special cases of Carriers, as well, supporting different initiation and interaction patterns, defined in the subsections below.

- o Listener: A Listener is a special case of Message Carrier which only responds to requests to create a new Carrier from a remote endpoint, analogous to a server or listening socket in the present sockets API. Instead of being bound to a specific remote endpoint, it is bound only to a local identity; however, its interface for accepting fork requests is identical to that for fully fledged Carriers.

- o Source: A Source is a special case of Message Carrier over which messages can only be sent, intended for unidirectional applications such as multicast transmitters. Sources cannot be forked, and need not accept forks.
- o Sink: A Sink is a special case of Message Carrier over which messages can only be received, intended for unidirectional applications such as multicast receivers. Sinks cannot be forked, and need not accept forks.
- o Responder: A Responder is a special case of Message Carrier which may receive messages from many remote sources, for cases in which an application will only ever send Messages in reply back to the source from which a Message was received. This is a common implementation pattern for servers in client-server applications. A Responder's receiver gets a Message, as well as a Source to send replies to. Responders cannot be forked, and need not accept forks.

2.2. Message

A Message is the unit of communication between applications. Messages can represent relatively small structures, such as requests in a request/response protocol such as HTTP; relatively large structures, such as files of arbitrary size in a filesystem; and structures of indeterminate length, such as a stream of bytes in a protocol like TCP.

In the general case, there is no mapping between a Message and packets sent by the underlying protocol stack on the wire: the transport protocol may freely segment messages and/or combine messages into packets. However, a message may be marked as immediate, which will cause it to be sent in a single packet when possible.

Content may be sent and received either as Complete or Partial Messages. Dealing with Complete Messages should be preferred for simplicity whenever possible based on the underlying protocol. It is always possible to send Complete Messages, but only protocols that have a fixed maximum message length may allow clients to receive Messages using an API that guarantees Complete Messages. Sending and receiving Partial Messages (that is, a Message whose content spans multiple calls or callbacks) is always possible.

To send a Message, either Complete or Partial, the Message content is passed into the Carrier, and client provides a set of callbacks to know when the Message was delivered or acknowledged. The client of the API may use the callbacks to pace the sending of Messages.

To receive a Message, the client of the API schedules a completion to be called when a Complete or Partial Message is available. If the client is willing to accept Partial Messages, it can specify the minimum incomplete Message length it is willing to receive at once, and the maximum number of bytes it is willing to receive at once. If the client wants Complete Messages, there are no values to tune. The scheduling of the receive completion indicates to the Carrier that there is a desire to receive bytes, effectively creating a "pull model" in which backpressure may be applied if the client is not receiving Messages or Partial Messages quickly enough to match the peer's sending rate. The Carrier may have some minimal buffer of incoming Messages ready for the client to read to reduce latency.

When receiving a Complete Message, the entire content of the Message must be delivered at once, and the Message is not delivered at all if the full Message is not received. This implies that both the sending and receiving endpoint, whether in the application or the carrier, must guarantee storage for the full size of a Message.

Partial Messages may be sent or received in several stages, with a handle representing the total Message being associated with each portion of the content. Each call to send or receive also indicates whether or not the Message is now complete. This approach is necessary whenever the size of the Message does not have a known bound, or the size is too large to process and hold in memory. Protocols that only present a concept of byte streams represent their data as single Messages with unknown bounds. In the case of TCP, the client will receive a single Message in pieces using the Partial Message API, and that Message will only be marked as complete when the peer has sent a FIN.

Messages are sent over and received from Message Carriers (see Section 2.1).

On sending, Messages have properties that allow the application to specify its requirements with respect to reliability, ordering, priority, idempotence, and immediacy; these are described in detail below. Messages may also have arbitrary properties which provide additional information to the underlying transport protocol stack on how they should be handled, in a protocol-specific way. These stacks may also deliver or set properties on received messages, but in the general case a received messages contains only a sequence of ordered bytes. Message properties include:

- o Lifetime and Partial Reliability: A Message may have a "lifetime" - a wall clock duration before which the Message must be available to the application layer at the remote end. If a lifetime cannot be met, the Message is discarded as soon as possible. Messages

without lifetimes are sent reliably if supported by the transport protocol stack. Lifetimes are also used to prioritize Message delivery.

There is no guarantee that a Message will not be delivered after the end of its lifetime; for example, a Message delivered over a strictly reliable transport will be delivered regardless of its lifetime. Depending on the transport protocol stack used to transmit the message, these lifetimes may also be signalled to path elements by the underlying transport, so that path elements that realize a lifetime cannot be met can discard frames containing the Messages instead of forwarding them.

- o **Priority:** Messages have a "niceness" - a priority among other messages sent over the same Carrier in an unbounded hierarchy most naturally represented as a non-negative integer. By default, Messages are in niceness class 0, or highest priority. Niceness class 1 Messages will yield to niceness class 0 Messages sent over the same Carrier, class 2 to class 1, and so on. Niceness may be translated to a priority signal for exposure to path elements (e.g. DSCP code point) to allow prioritization along the path as well as at the sender and receiver. This inversion of normal schemes for expressing priority has a convenient property: priority increases as both niceness and lifetime decrease. A Message may have both a niceness and a lifetime - Messages with higher niceness classes will yield to lower classes if resource constraints mean only one can meet the lifetime.
- o **Dependence:** A Message may have "antecedents" - other Messages on which it depends, which must be delivered before it (the "successor") is delivered. The sending transport uses deadlines, niceness, and antecedents, along with information about the properties of the Paths available, to determine when to send which Message down which Path.
- o **Idempotence:** A sending application may mark a Message as "idempotent" to signal to the underlying transport protocol stack that its application semantics make it safe to send in situations that may cause it to be received more than once (i.e., for 0-RTT session resumption as in TCP Fast Open, TLS 1.3, and QUIC).
- o **Immediacy:** A sending application may mark a Message as "immediate" to signal to the underlying transport protocol stack that its application semantics require it to be placed in a single packet, on its own, instead of waiting to be combined with other messages or parts thereof (i.e., for media transports and interactive sessions with small messages).

Senders may also be asynchronously notified of three events on Messages they have sent: that the Message has been transmitted, that the Message has been acknowledged by the receiver, or that the Message has expired before transmission/acknowledgement. Not all transport protocol stacks will support all of these events.

2.3. Association

An Association contains the long-term state necessary to support communications between a Local (see Section 2.5) and a Remote (see Section 2.4) endpoint, such as trust model information, including pinned public keys or anchor certificates, cryptographic session resumption parameters, or rendezvous information. It uses information from the Configuration (see Section 2.6) to constrain the selection of transport protocols and local interfaces to create Transients (see Section 2.7) to carry Messages; and information about the paths through the network available between them (see Section 2.8).

All Carriers are bound to an Association. New Carriers will reuse an Association if they can be carried from the same Local to the same Remote over the same Paths; this re-use of an Association may imply the creation of a new Transient.

Associations may exist and be created without a Carrier. This may be done if peer cryptographic state such as a pre-shared key is established out-of-band. Thus, Associations may be created without the need to send application data to a peer, that is, without a Carrier. Associations are mutable. Association state may expire over time, after which it is removed from the Association, and Transients may export cryptographic state to store in an Association as needed. Moreover, this state may be exported directly into the Association or modified before insertion. This may be needed to diversify ephemeral Transient keying material from the longer-term Association keying material.

A primary use of Association state is to allow new Associations and their derived Carriers to be quickly created without performing in-band cryptographic handshakes. See [I-D.kuehlewind-taps-crypto-sep] for more details about this separation.

2.4. Remote

A Remote represents information required to establish and maintain a connection with the far end of an Association: name(s), address(es), and transport protocol parameters that can be used to establish a Transient; transport protocols to use; trust model information, inherited from the relevant Association, used to identify the remote

on connection establishment; and so on. Each Association is associated with a single Remote, either explicitly by the application (when created by the initiation of a Carrier) or a Listener (when created by forking a Carrier on passive open).

A Remote may be resolved, which results in zero or more Remotes with more specific information. For example, an application may want to establish a connection to a website identified by a URL `https://www.example.com`. This URL would be wrapped in a Remote and passed to a call to initiate a Carrier. The first pass resolution might parse the URL, decomposing it into a name, a transport port, and a transport protocol to try connecting with. A second pass resolution would then look up network-layer addresses associated with that name through DNS, and store any certificates available from DANE. Once a Remote has been resolved to the point that a transport protocol stack can use it to create a Transient, it is considered fully resolved.

2.5. Local

A Local represents all the information about the local endpoint necessary to establish an Association or a Listener. It encapsulates the Provisioning Domain (PvD) of a single interface in the multiple provisioning domain architecture [RFC7556], and adds information about the service endpoint (transport protocol port), and, per [I-D.pauly-taps-transport-security], cryptographic identities (certificates and associated private keys) bound to this endpoint.

2.6. Configuration

A Configuration encapsulates an application's preferences around Path selection and protocol options.

Each Association has exactly one Configuration, and all Carriers belonging to that Association share the same Configuration.

The application cannot modify the Configuration for a Carrier or Association once it is set. If a new set of options needs to be used, then the application needs a new Carrier or Association instance. This is necessary to ensure that a single Carrier can consistently track the Paths and protocol options it uses, since it is usually not possible to modify these properties without breaking connectivity.

To influence Path selection, the application can configure a set of requirements, preferences, and restrictions concerning which Paths may be selected by the Association to use for creating Transients between a Local and a Remote. For example, a Configuration can

specify that the application prefers Wi-Fi access over LTE when roaming on a foreign LTE network, due to monetary cost to the user.

The Association uses the Configuration's Path preferences as a key part of determining the Paths to use for its Transients. The Configuration is provided as input when examining the complete list of available Paths on the system (to limit the list, or order the Paths by preference). The system's policy will further restrict and modify the Path that is ultimately selected, using other aspects of the Configuration (protocol options and originating application) to select the most appropriate Path.

To influence protocol selection and options, the Configuration contains one or more allowed Protocol Stack Configurations. Each of these is comprised of application- and transport-layer protocols that may be used together to communicate to the Remote, along with any protocol-specific options. For example, a Configuration could specify two alternate, but equivalent, protocol stacks: one using HTTP/2 over TLS over TCP, and the other using QUIC over UDP. Alternatively, the Configuration could specify two protocol stacks with the same protocols, but different protocol options: one using TLS with TLS 1.3 0-RTT enabled and TCP with TCP Fast-Open enabled, and one using TLS with out 0-RTT and TCP without TCP Fast-Open.

Protocol-specific options within the Configuration include trust settings and acceptable cryptographic algorithms to be used by security protocols. These may be configured for specific protocols to allow different settings for each (such as between TLS over TCP and TLS for use with QUIC), or set as default security settings on the Configuration to be used by any protocol that needs to evaluate trust. Trust settings may include certificate anchors and certificate pinning options.

2.7. Transient

A Transient represents a binding between a Carrier and the instance of the transport protocol stack that implements it. As an Association contains long-term state for communications between two endpoints, a Transient contains ephemeral state for a single transport protocol over a one or more Paths at a given point in time.

A Carrier may be served by multiple Transients at once, e.g. when implementing multipath communication such that the separate paths are exposed to the API by the underlying transport protocol stack. Each Transient serves only one Carrier, although multiple Transients may share the same underlying protocol stack; e.g. when multiplexing Carriers over streams in a multistreaming protocol.

Transients are generally not exposed by the API to the application, though they may be accessible for debugging and logging purposes.

2.8. Path

A Path represents information about a single path through the network used by an Association, in terms of source and destination network and transport layer addresses within an addressing context, and the provisioning domain [RFC7556] of the local interface. This information may be learned through a resolution, discovery, or rendezvous process (e.g. DNS, ICE), by measurements taken by the transport protocol stack, or by some other path information discovery mechanism. It is used by the transport protocol stack to maintain and/or (re-)establish communications for the Association.

The set of available properties is a function of the transport protocol stacks in use by an association. However, the following core properties are generally useful for applications and transport layer protocols to choose among paths for specific Messages:

- o Maximum Transmission Unit (MTU): the maximum size of an Message's payload (subtracting transport, network, and link layer overhead) which will likely fit into a single frame. Derived from signals sent by path elements, where available, and/or path MTU discovery processes run by the transport layer.
- o Latency Expectation: expected one-way delay along the Path. Generally provided by inline measurements performed by the transport layer, as opposed to signaled by path elements.
- o Loss Probability Expectation: expected probability of a loss of any given single frame along the Path. Generally provided by inline measurements performed by the transport layer, as opposed to signaled by path elements.
- o Available Data Rate Expectation: expected maximum data rate along the Path. May be derived from passive measurements by the transport layer, or from signals from path elements.
- o Reserved Data Rate: Committed, reserved data rate for the given Association along the Path. Requires a bandwidth reservation service in the underlying transport protocol stack.
- o Path Element Membership: Identifiers for some or all nodes along the path, depending on the capabilities of the underlying network layer protocol to provide this.

Path properties are generally read-only. MTU is a property of the underlying link-layer technology on each link in the path; latency, loss, and rate expectations are dynamic properties of the network configuration and network traffic conditions; path element membership is a function of network topology. In an explicitly multipath architecture, application and transport layer requirements can be met by having multiple paths with different properties to select from. Transport protocol stacks can also provide signaling to devices along the path, but this signaling is derived from information provided to the Message abstraction.

3. Abstract Programming Interface

We now turn to the design of an abstract programming interface to provide a simple interface to Post's abstractions, constrained by the following design principles:

- o Flexibility is paramount. So is simplicity. Applications must be given as many controls and as much information as they may need, but they must be able to ignore controls and information irrelevant to their operation. This implies that the "default" interface must be no more complicated than BSD sockets, and must do something reasonable.
- o Reception is an inherently asynchronous activity. While the API is designed to be as platform-independent as possible, one key insight it is based on is that a Message receiver's behavior in a packet-switched network is inherently asynchronous, driven by the receipt of packets, and that this asynchronicity must be reflected in the API. The actual implementation of receive and event handling will need to be aligned to the method a given platform provides for asynchronous I/O.
- o A new API cannot be bound to a single transport protocol and expect wide deployment. As the API is transport-independent and may support runtime transport selection, it must impose the minimum possible set of constraints on its underlying transports, though some API features may require underlying transport features to work optimally. It must be possible to implement Post over vanilla TCP in the present Internet architecture.

The API we design from these principles is centered around a Carrier, which can be created actively via `initiate()` or passively via a `listen()`; the latter creates a Listener from which new Carriers can be `accept()`ed. Messages may be created explicitly and passed to this Carrier, or implicitly through a simplified interface which uses default message properties (reliable transport without priority or

deadline, which guarantees ordered delivery over a single Carrier when the underlying transport protocol stack supports it).

For each connection between a Local and a Remote a new Carrier is created and destroyed when the connection is closed. However, a new Carrier may use an existing Association if present for the requested Local-Remote pair and permitted by the PolicyContext that can be provided at Carrier initiation. Further the system-wide PolicyContext can contain more information that determine when to create or destroy Associations other than at Carrier initiation. E.g. an Association can be created at system start, based on the configured PolicyContext or also by a manual action of an single application, for Local-Remote pairs that are known to be likely used soon, and to pre-establish, e.g., cryptographic context as well as potentially collect current information about path capabilities. Every time an actual connection with a specific PSI is established between the Local and Remote, the Association learns new Path information and stores them. This information can be used when a new transient is created, e.g. to decide which PSI to use (to provide the highest probably for a successful connection attempt) or which PSIs to probe for (first). A Transient is created when an application actually sends a Message over a Carrier. As further explained below this step can actually create multiple transients for probing or assign a new transient to an already active PSI, e.g. if multi-streaming is provided and supported for these kind of use on both sides.

3.1. Example Connection Patterns

Here, we illustrate the usage of the API for common connection patterns. Note that error handling is ignored in these illustrations for ease of reading.

3.1.1. Client-Server

Here's an example client-server application. The server echoes messages. The client sends a message and prints what it receives.

The client in Figure 2 connects, sends a message, and sets up a receiver to print messages received in response. The carrier is inactive after the Initiate() call; the Send() call blocks until the carrier can be activated.

```
// connect to a server given a remote
func sayHello() {

    carrier := Initiate(local, remote)

    carrier.Send([]byte("Hello!"))
    carrier.Ready(func (msg InMessage) {
        fmt.Println(string([]byte(msg))
        return false
    })
    carrier.Close()
}
```

Figure 2: Example client

The server in Figure 3 creates a Listener, which accepts Carriers and passes them to a server. The server echos the content of each message it receives.

```
// run a server for a specific carrier, echo all its messages
func runMyServerOn(carrier Carrier) {
    carrier.Ready(func (msg InMessage) {
        carrier.Send(msg)
    })
}

// accept connections forever, spawn servers for them
func acceptConnections() {
    listener := Listen(local)
    listener.Accept(func(carrier Carrier) bool {
        go runMyServerOn(carrier)
        return true
    })
}
```

Figure 3: Example server

The Responder allows the server to be significantly simplified, as shown in Figure 4.

```
func echo(msg InMessage, reply Sink) {
    reply.Send(msg)
}

Respond(local, echo)
```

Figure 4: Example responder

3.1.2. Client-Server with Happy Eyeballs and 0-RTT establishment

The fundamental design of a client need not change at all for happy eyeballs [RFC6555] (selection of multiple potential protocol stacks through connection racing); this is handled by the Post Sockets implementation automatically. If this connection racing is to use 0-RTT data (i.e., as provided by TCP Fast Open [RFC7413]), the client must mark the outgoing message as idempotent.

```
// connect to a server given a remote and send some 0-RTT data
func sayHelloQuickly() {

    carrier := Initiate(local, remote)

    carrier.SendMsg(OutMessage{Content: []byte("Hello!"), Idempotent: true}, nil
, nil, nil)
    carrier.Ready(func (msg InMessage) {
        fmt.Println(string([]byte(msg)))
        return false
    })
    carrier.Close()
}
```

3.1.3. Peer to Peer with Network Address Translation

In the client-server examples shown above, the Remote given to the Initiate call refers to the name and port of the server to connect to. This need not be the case, however; a Remote may also refer to an identity and a rendezvous point for rendezvous as in ICE [RFC5245]. Here, each peer does its own Initiate call simultaneously, and the result on each side is a Carrier attached to an appropriate Association.

3.1.4. Multicast Receiver

A multicast receiver is implemented using a Sink attached to a Local encapsulating a multicast address on which to receive multicast datagrams. The following example prints messages received on the multicast address forever.

```
func receiveMulticast(){
    sink = NewSink(local)
    sink.Ready(func (msg InMessage) {
        fmt.Println(string([]byte(msg)))
        return true
    })
}
```

3.1.5. Association Bootstrapping

Here, we show how Association state may be initialized without a carrier. The goal is to create a long-term Association from which Carriers may be derived and, if possible, used immediately. Per [I-D.pauly-taps-transport-security], a first step is to specify trust model constraints, such as pinned public keys and anchor certificates, which are needed to create Remote connections.

We begin by creating shared security parameters that will be used later for creating a remote connection.

```
// create security parameters with a set of trusted certificates
func createParameters(trustedCerts []Certificate) Parameters {
    parameters := Parameters()
    parameters = parameters.SetTrustedCerts(trustedCerts)
    return parameters
}
```

Using these statically configured parameters, we now show how to create an Association between a Local and Remote using these parameters.

```
// create an Association using shared parameters
func createAssociation(local Local, remote Remote, parameters Parameters) Association {
    association := NewAssociation(local, remote, parameters)
    return association
}
```

We may also create an Association with a pre-shared key configured out-of-band.

```
// create an Association using a pre-shared key
func createAssociationWithPSK(local Local, remote Remote, parameters Parameters,
    preSharedKey []byte) Association {
    association := NewAssociation(local, remote, parameters)
    association = association.SetPreSharedKey(preSharedKey)
    return association
}
```

We now show how to create a Carrier from an existing, pre-configured Association. This Association may or may not contain shared cryptographic static between the Local and Remote, depending on how it was configured.

```
// open a connection to a server using an existing Association and send some data,
// which will be sent early if possible.
func sayHelloWithAssociation(association Association) {
    carrier := association.Initiate()

    carrier.SendMsg(OutMessage{Content: []byte("Hello!"), Idempotent: true}, nil, nil)
    carrier.Ready(func (msg InMessage) {
        fmt.Println(string([]byte(msg)))
        return false
    })
    carrier.Close()
}
```

3.2. API Dynamics

As Carriers provide the central entry point to Post, they are key to API dynamics. The lifecycle of a carrier is shown in Figure 5. Carriers are created by active openers by calling `Initiate()` given a Local and a Remote, and by passive openers by calling `Listen()` given a Local; the `.Accept()` method on the listener Carrier can then be used to create active carriers. By default, the underlying Association is automatically created and managed by the underlying API. This underlying Association can be accessed by the Carrier's `.Association()` method. Alternately, an association can be explicitly created using `NewAssociation()`, and a Carrier on the association may be accessed or initiated by calling the association's `.Initiate()` method.

Once a Carrier has been created (via `Initiate()`, `Association.Initiate()`, `NewSource()`, `NewSink()`, or `Listen()/Accept()`), it may be used to send and receive Messages. The existence of a Carrier does not imply the existence of an active Transient or associated transport-layer connection; these may be created when the carrier is, or may be deferred, depending on the network environment, configuration, and protocol stacks available.

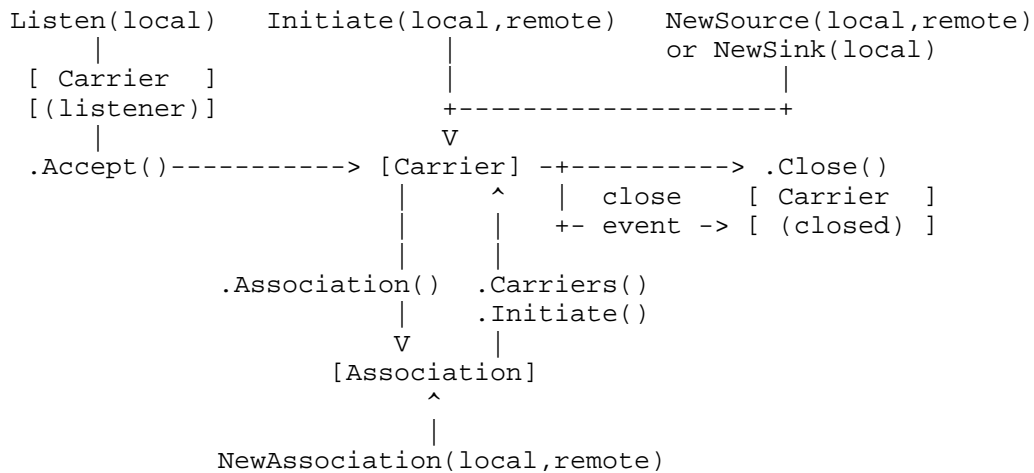


Figure 5: Carrier and Association Life Cycle

Access to more detailed information is possible through accessors on Carriers and Associations, as shown in Figure 6. The set of currently active Transients can be accessed through the Carrier's `.Transients()` methods. The active path(s) used by a Transient can be accessed through the Transient's `.Paths()` method, and the set of all paths for which properties are cached by an Association can be accessed through the Association's `.Paths()` method. The set of active carriers on an association can be accessed through the Association's `.Carriers()` method. Access to transients and paths is not necessary in normal operation; these accessors are provided primarily for logging and debugging purposes.

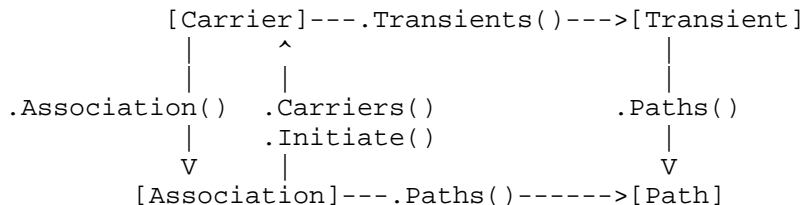


Figure 6: Accessors on Carriers and Associations

Each Carrier has a `.Send()` method, by which Messages can be sent with given properties, and a `.Ready()` method, which supplies a callback for reading Messages from the remote side. `.Send()` is not available on Sinks, and `.Ready()` is not available on Sources. Carriers also provide `.OnSent()`, `.OnAked()`, and `.OnExpired()` calls for binding default send event handlers to the Carrier, and `.OnClosed()` for handling passive close notifications.

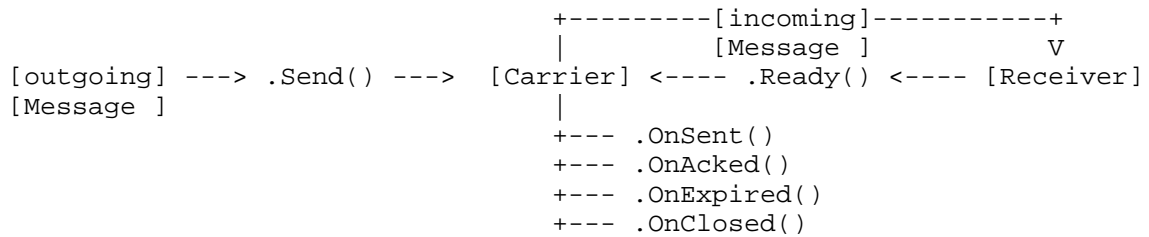


Figure 7: Sending and Receiving Messages and Events

An application may have a global Configuration, as well as more specific Configurations to apply to the establishment of a given Association or Carrier. These Configurations are optional arguments to the Association and Carrier creation calls.

In order to initiate a connection with a remote endpoint, a user of Post Sockets must start from a Remote (see Section 2.4). A Remote encapsulates identifying information about a remote endpoint at a specific level of resolution. A new Remote can be wrapped around some identifying information by via the NewRemote() call. A Remote has a .Resolve() method, which can be iteratively revoked to increase the level of resolution; a call to Resolve on a given Remote may result in one to many Remotes, as shown in Figure 8. Remotes at any level of resolution may be passed to Post Sockets calls; each call will continue resolution to the point necessary to establish or resume a Carrier.

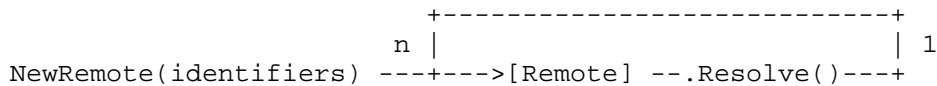


Figure 8: Recursive resolution of Remotes

Information about the local endpoint is also necessary to establish an Association, whether explicitly or implicitly through the creation of a Carrier or Listener. This is passed in the form of a Local (see Section 2.5). A Local is created with a NewLocal() call, which takes a Configuration (including certificates to present and secret keys associated with them) and identifying information (interface(s) and port(s) to use).

4. Implementation Considerations

Here we discuss an incomplete list of API implementation considerations that have arisen with experimentation with prototype implementations of Post.

4.1. Protocol Stack Instance (PSI)

A PSI encapsulates an arbitrary stack of protocols (e.g., TCP over IPv6, SCTP over DTLS over UDP over IPv4). PSIs provide the bridge between the interface (Carrier) plus the current state (Transients) and the implementation of a given set of transport services [I-D.ietf-taps-transports].

A given implementation makes one or more possible protocol stacks available to its applications. Selection and configuration among multiple PSIs is based on system-level or application policies, as well as on network conditions in the provisioning domain in which a connection is made.

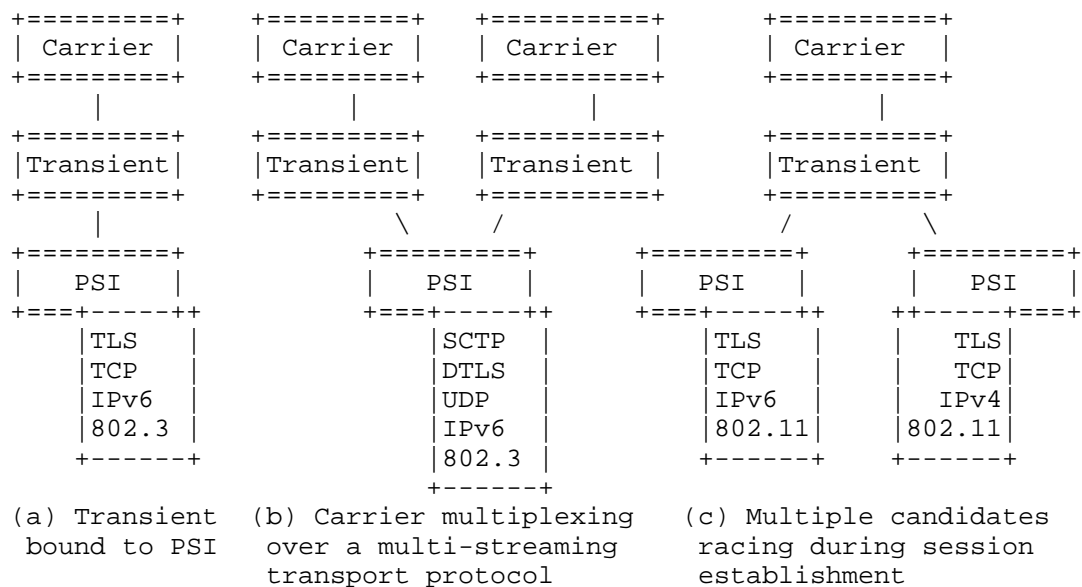


Figure 9: Example Protocol Stack Instances

For example, Figure 9(a) shows a TLS over TCP stack, usable on most network connections. Protocols are layered to ensure that the PSI provides all the transport services required by the application. A single PSI may be bound to multiple Carriers, as shown in Figure 9(b): a multi-streaming transport protocol like QUIC or SCTP can support one carrier per stream. Where multi-streaming transport is not available, these carriers could be serviced by different PSIs on different flows. On the other hand, multiple PSIs are bound to a single transient during establishment, as shown in Figure 9(c). Here, the losing PSI in a happy-eyeballs race will be terminated, and the carrier will continue using the winning PSI.

4.2. Message Framing, Parsing, and Serialization

While some transports expose a byte stream abstraction, most higher level protocols impose some structure onto that byte stream. That is, the higher level protocol operates in terms of messages, protocol data units (PDUs), rather than using unstructured sequences of bytes, with each message being processed in turn. Protocols are specified in terms of state machines acting on semantic messages, with parsing the byte stream into messages being a necessary annoyance, rather than a semantic concern. Accordingly, Post Sockets exposes a message-based API to applications as the primary abstraction. Protocols that deal only in byte streams, such as TCP, represent their data in each direction as a single, long message. When framing protocols are placed on top of byte streams, the messages used in the API represent the framed messages within the stream.

There are other benefits of providing a message-oriented API beyond framing PDUs that Post Sockets should provide when supported by the underlying transport. These include:

- o the ability to associate deadlines with messages, for transports that care about timing;
- o the ability to provide control of reliability, choosing what messages to retransmit in the event of packet loss, and how best to make use of the data that arrived;
- o the ability to manage dependencies between messages, when some messages may not be delivered due to either packet loss or missing a deadline, in particular the ability to avoid (re-)sending data that relies on a previous transmission that was never received.

All require explicit message boundaries, and application-level framing of messages, to be effective. Once a message is passed to Post Sockets, it can not be cancelled or paused, but prioritization as well as lifetime and retransmission management will provide the protocol stack with all needed information to send the messages as quickly as possible without blocking transmission unnecessarily. Post Sockets provides this by handling message, with known identity (sequence numbers, in the simple case), lifetimes, niceness, and antecedents.

Transport protocols such as SCTP provide a message-oriented API that has similar features to those we describe. Other transports, such as TCP, do not. To support a message oriented API, while still being compatible with stream-based transport protocols, Post Sockets must provide APIs for parsing and serialising messages that understand the protocol data. That is, we push message parsing and serialisation

down into the Post Sockets stack, allowing applications to send and receive strongly typed data objects (e.g., a receive call on an HTTP Message Carrier should return an object representing the HTTP response, with pre-parsed status code, headers, and any message body, rather than returning a byte array that the application has to parse itself). This is backwards compatible with existing protocols and APIs, since the wire format of messages does not change, but gives a Post Sockets stack additional information to allow it to make better use of modern transport services.

The Post Sockets approach is therefore to raise the semantic level of the transport API: applications should send and receive messages in the form of meaningful, strongly typed, protocol data. Parsing and serialising such messages should be a re-usable function of the protocol stack instance not the application. This is well-suited to implementation in modern systems languages, such as Swift, Go, Rust, or C++, but can also be implemented with some loss of type safety in C.

4.3. Message Size Limitations

Ideally, Messages can be of infinite size. However, protocol stacks and protocol stack implementations may impose their own limits on message sizing; For example, SCTP [RFC4960] and TLS [I-D.ietf-tls-tls13] impose record size limitations of 64kB and 16kB, respectively. Message sizes may also be limited by the available buffer at the receiver, since a Message must be fully assembled by the transport layer before it can be passed on to the application layer. Since not every transport protocol stack implements the signaling necessary to negotiate or expose message size limitations, these may need to be defined out of band, and are probably best exposed through the Configuration.

A truly infinite message service - e.g. large file transfer where both endpoints have committed persistent storage to the message - is probably best realized as a layer above Post Sockets, and may be added as a new type of Message Carrier to a future revision of this document.

4.4. Back-pressure

Regardless of how asynchronous reception is implemented, it is important for an application to be able to apply receiver back-pressure, to allow the protocol stack to perform receiver flow control. Depending on how asynchronous I/O works in the platform, this could be implemented by having a maximum number of concurrent receive callbacks, or by bounding the maximum number of outstanding, unread bytes at any given time, for example.

4.5. Associations, Transients, Racing, and Rendezvous

As the network has evolved, even the simple act of establishing a connection has become increasingly complex. Clients now regularly race multiple connections, for example over IPv4 and IPv6, to determine which protocol to use. The choice of outgoing interface has also become more important, with differential reachability and performance from multiple interfaces. Name resolution can also give different outcomes depending on the interface the query was issued from. Finally, but often most significantly, NAT traversal, relay discovery, and path state maintenance messages are an essential part of connection establishment, especially for peer-to-peer applications.

Post Sockets accordingly breaks communication establishment down into multiple phases:

- o Gathering Locals

The set of possible Locals is gathered. In the simple case, this merely enumerates the local interfaces and protocols, and allocates ephemeral source ports for transients. For example, a system that has WiFi and Ethernet and supports IPv4 and IPv6 might gather four candidate locals (IPv4 on Ethernet, IPv6 on Ethernet, IPv4 on WiFi, and IPv6 on WiFi) that can form the source for a transient.

If NAT traversal is required, the process of gathering locals becomes broadly equivalent to the ICE candidate gathering phase [RFC5245]. The endpoint determines its server reflexive locals (i.e., the translated address of a local, on the other side of a NAT) and relayed locals (e.g., via a TURN server or other relay), for each interface and network protocol. These are added to the set of candidate locals for this association.

Gathering locals is primarily an endpoint local operation, although it might involve exchanges with a STUN server to derive server reflexive locals, or with a TURN server or other relay to derive relayed locals. It does not involve communication with the remote.

- o Resolving the Remote

The remote is typically a name that needs to be resolved into a set of possible addresses that can be used for communication. Resolving the remote is the process of recursively performing such name lookups, until fully resolved, to return the set of candidates for the remote of this association.

How this is done will depend on the type of the Remote, and can also be specific to each local. A common case is when the Remote is a DNS name, in which case it is resolved to give a set of IPv4 and IPv6 addresses representing that name. Some types of remote might require more complex resolution. Resolving the remote for a peer-to-peer connection might involve communication with a rendezvous server, which in turn contacts the peer to gain consent to communicate and retrieve its set of candidate locals, which are returned and form the candidate remote addresses for contacting that peer.

Resolving the remote is not a local operation. It will involve a directory service, and can require communication with the remote to rendezvous and exchange peer addresses. This can expose some or all of the candidate locals to the remote.

- o Establishing Transients

The set of candidate locals and the set of candidate remotes are paired, to derive a priority ordered set of Candidate Paths that can potentially be used to establish a connection.

Then, communication is attempted over each candidate path, in priority order. If there are multiple candidates with the same priority, then transient establishment proceeds simultaneously and uses the transient that wins the race to be established. Otherwise, transients establishment is sequential, paced at a rate that should not congest the network. Depending on the chosen transport, this phase might involve racing TCP connections to a server over IPv4 and IPv6 [RFC6555], or it could involve a STUN exchange to establish peer-to-peer UDP connectivity [RFC5245], or some other means.

- o Confirming and Maintaining Transients

Once connectivity has been established, unused resources can be released and the chosen path can be confirmed. This is primarily required when establishing peer-to-peer connectivity, where connections supporting relayed locals that were not required can be closed, and where an associated signalling operation might be needed to inform middleboxes and proxies of the chosen path. Keep-alive messages may also be sent, as appropriate, to ensure NAT and firewall state is maintained, so the transient remains operational.

By encapsulating these four phases of communication establishment into the PSI, Post Sockets aims to simplify application development. It can provide reusable implementations of connection racing for TCP,

to enable happy eyeballs, that will be automatically used by all TCP clients, for example. With appropriate callbacks to drive the rendezvous signalling as part of resolving the remote, we believe a generic ICE implementation ought also to be possible. This procedure can even be repeated fully or partially during a connection to enable seamless hand-over and mobility within the network stack.

5. Acknowledgments

Many thanks to Laurent Chuat and Jason Lee at the Network Security Group at ETH Zurich for contributions to the initial design of Post Sockets. Thanks to Joe Hildebrand, Martin Thomson, and Michael Welzl for their feedback, as well as the attendees of the Post Sockets workshop in February 2017 in Zurich for the discussions, which have improved the design described herein.

This work is partially supported by the European Commission under Horizon 2020 grant agreement no. 688421 Measurement and Architecture for a Middleboxed Internet (MAMI), and by the Swiss State Secretariat for Education, Research, and Innovation under contract no. 15.0268. This support does not imply endorsement.

6. References

6.1. Normative References

[I-D.ietf-taps-transports]
Fairhurst, G., Trammell, B., and M. Kuehlewind, "Services provided by IETF transport protocols and congestion control mechanisms", draft-ietf-taps-transports-14 (work in progress), December 2016.

6.2. Informative References

[I-D.ietf-quic-transport]
Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-07 (work in progress), October 2017.

[I-D.ietf-tls-tls13]
Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-21 (work in progress), July 2017.

[I-D.iyengar-minion-protocol]
Jana, J., Cheshire, S., and J. Graessley, "Minion - Wire Protocol", draft-iyengar-minion-protocol-02 (work in progress), October 2013.

- [I-D.kuehlewind-taps-crypto-sep]
Kuehlewind, M., Pauly, T., and C. Wood, "Separating Crypto Negotiation and Communication", draft-kuehlewind-taps-crypto-sep-00 (work in progress), July 2017.
- [I-D.pauly-taps-transport-security]
Pauly, T. and C. Wood, "A Survey of Transport Security Protocols", draft-pauly-taps-transport-security-00 (work in progress), July 2017.
- [I-D.trammell-plus-abstract-mech]
Trammell, B., "Abstract Mechanisms for a Cooperative Path Layer under Endpoint Control", draft-trammell-plus-abstract-mech-00 (work in progress), September 2016.
- [I-D.trammell-plus-statefulness]
Kuehlewind, M., Trammell, B., and J. Hildebrand, "Transport-Independent Path Layer State Management", draft-trammell-plus-statefulness-03 (work in progress), March 2017.
- [MinimalT]
Petullo, W., Zhang, X., Solworth, J., Bernstein, D., and T. Lange, "MinimalT, Minimal-latency Networking Through Better Security", May 2013.
- [NEAT]
Grinnemo, K-J., Tom Jones, ., Gorrry Fairhurst, ., David Ros, ., Anna Brunstrom, ., and . Per Hurtig, "Towards a Flexible Internet Transport Layer Architecture", June 2016.
- [RFC0793]
Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <<https://www.rfc-editor.org/info/rfc793>>.
- [RFC4960]
Stewart, R., Ed., "Stream Control Transmission Protocol", RFC 4960, DOI 10.17487/RFC4960, September 2007, <<https://www.rfc-editor.org/info/rfc4960>>.
- [RFC5245]
Rosenberg, J., "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols", RFC 5245, DOI 10.17487/RFC5245, April 2010, <<https://www.rfc-editor.org/info/rfc5245>>.
- [RFC6555]
Wing, D. and A. Yourtchenko, "Happy Eyeballs: Success with Dual-Stack Hosts", RFC 6555, DOI 10.17487/RFC6555, April 2012, <<https://www.rfc-editor.org/info/rfc6555>>.

- [RFC6698] Hoffman, P. and J. Schlyter, "The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA", RFC 6698, DOI 10.17487/RFC6698, August 2012, <<https://www.rfc-editor.org/info/rfc6698>>.
- [RFC6824] Ford, A., Raiciu, C., Handley, M., and O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses", RFC 6824, DOI 10.17487/RFC6824, January 2013, <<https://www.rfc-editor.org/info/rfc6824>>.
- [RFC7258] Farrell, S. and H. Tschofenig, "Pervasive Monitoring Is an Attack", BCP 188, RFC 7258, DOI 10.17487/RFC7258, May 2014, <<https://www.rfc-editor.org/info/rfc7258>>.
- [RFC7413] Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP Fast Open", RFC 7413, DOI 10.17487/RFC7413, December 2014, <<https://www.rfc-editor.org/info/rfc7413>>.
- [RFC7556] Anipko, D., Ed., "Multiple Provisioning Domain Architecture", RFC 7556, DOI 10.17487/RFC7556, June 2015, <<https://www.rfc-editor.org/info/rfc7556>>.

Appendix A. Open Issues

This document is under active development; a list of current open issues is available at <https://github.com/mami-project/draft-trammell-post-sockets/issues>

Authors' Addresses

Brian Trammell
ETH Zurich
Gloriastrasse 35
8092 Zurich
Switzerland

Email: ietf@trammell.ch

Colin Perkins
University of Glasgow
School of Computing Science
Glasgow G12 8QQ
United Kingdom

Email: csp@csp Perkins.org

Tommy Pauly
Apple Inc.
1 Infinite Loop
Cupertino, California 95014
United States of America

Email: tpauly@apple.com

Mirja Kuehlewind
ETH Zurich
Gloriastrasse 35
8092 Zurich
Switzerland

Email: mirja.kuehlewind@tik.ee.ethz.ch

Chris Wood
Apple Inc.
1 Infinite Loop
Cupertino, California 95014
United States of America

Email: cawood@apple.com