NEAT

A New, Evolutive API and Transport-Layer Architecture for the Internet

H2020-ICT-05-2014
Project number: 644334

Deliverable D1.3
Final Version of Services and APIs

Editor(s): Michael Welzl
Contributor(s): Dragana Damjanovic, Gorry Fairhurst, David Hayes, Tom Jones, David Ros, Michael Tüxen, Felix Weinrank

Work Package: 1 / Use Cases, System Architecture and APIs
Revision: 1.0
Date: October 30, 2017
Deliverable type: R (Report)
Dissemination level: Public

Abstract

This document presents the final version of transport services that the NEAT User API provides to applications. This API reflects the extended functionality that NEAT currently offers. The API also provides primitives to interface to the NEAT Policy Manager; policies can be adjusted to match the API behaviour to the properties required by an application using the NEAT User API. The final API has evolved in concert with documents and feedback in the IETF TAPS Working Group.

The abstract API described here is based on the final analysis and design work done in Work Package 1 of the NEAT Project. This fulfils the requirements of the NEAT use cases, outlined in Deliverable D1.1. The API has evolved and been streamlined following implementation experience from WP2. Some original primitives and events identified in Deliverable D1.2 have been changed or removed; most importantly perhaps, it was determined to be more suitable to implement some functions in the form of a policy rather than as a function call. This document updates D1.2 and it supersedes the API that was presented in that deliverable.

Participant organisation name	**Short name**
Simula Research Laboratory AS *(Coordinator)*	SRL
Celerway Communication AS	Celerway
EMC Information Systems International	EMC
MZ Denmark APS	Mozilla
Karlstads Universitet	KaU
Fachhochschule Münster	FHM
The University Court of the University of Aberdeen	UoA
Universitetet i Oslo	UiO
Cisco Systems France SARL	Cisco

Contents

List of abbreviations

AAA Authentication, Authorisation and Accounting

AAAA Authentication, Authorisation, Accounting and Auditing

API Application Programming Interface

BE Best Effort

BLEST Blocking Estimation-based MPTCP

CC Congestion Control

CCC Coupled Congestion Controller

CDG CAIA Delay Gradient

CIB Characteristics Information Base

CM Congestion Manager

DA-LBE Deadline Aware Less than Best Effort

DAPS Delay-Aware Packet Scheduling

DCCP Datagram Congestion Control Protocol

DNS Domain Name System

DNSSEC Domain Name System Security Extensions

DPI Deep Packet Inspection

DSCP Differentiated Services Code Point

DTLS Datagram Transport Layer Security

ECMP Equal Cost Multi-Path

EFCM Ensemble Flow Congestion Manager

ECN Explicit Congestion Notification

ENUM Electronic Telephone Number Mapping

E-TCP Ensemble-TCP

FEC Forward Error Correction

FLOWER Fuzzy Lower than Best Effort

FSE Flow State Exchange

FSN Fragments Sequence Number

GUE Generic UDP Encapsulation

H1 HTTP/1

H2 HTTP/2

HE Happy Eyeballs

HoLB Head of Line Blocking

HTTP HyperText Transfer Protocol

IAB Internet Architecture Board

ICE Internet Connectivity Establishment

ICMP Internet Control Message Protocol

IETF Internet Engineering Task Force

IF Interface

IGD-PCP Internet Gateway Device – Port Control Protocol

IoT Internet of Things

IP Internet Protocol

IRTF Internet Research Task Force

IW Initial Window

IW10 Initial Window of 10 segments

JSON JavaScript Object Notation

KPI Kernel Programming Interface

LAG Link Aggregation

LAN Local Area Network

LBE Less than Best Effort

LEDBAT Low Extra Delay Background Transport

LRF Lowest RTT First

MBC Model Based Control

MID Message Identifier

MIF Multiple Interfaces

MPTCP Multipath Transmission Control Protocol

MPT-BM Multipath Transport-Bufferbloat Mitigation

MTU Maximum Transmission Unit

NAT Network Address (and Port) Translation

NEAT New, Evolutive API and Transport-Layer Architecture

NIC Network Interface Card

NUM Network Utility Maximization

OF OpenFlow

OS Operating System

OTIAS Out-of-order Transmission for In-order Arrival Scheduling

OVSDB Open vSwitch Database

PCP Port Control Protocol

PDU Protocol Data Unit

PHB Per-Hop Behaviour

PI Policy Interface

PIB Policy Information Base

PID Proportional-Integral-Differential

PLUS Path Layer UDP Substrate

PM Policy Manager

PMTU Path MTU

POSIX Portable Operating System Interface

PPID Payload Protocol Identifier

PRR Proportional Rate Reduction

PvD Provisioning Domain

QoS Quality of Service

QUIC Quick UDP Internet Connections

RACK Recent Acknowledgement

RFC Request for Comments

RSerPool Reliable Server Pooling

RTT Round Trip Time

RTP Real-time Protocol

RTSP Real-time Streaming Protocol

SCTP Stream Control Transmission Protocol

SCTP-CMT Stream Control Transmission Protocol – Concurrent Multipath Transport

SCTP-PF Stream Control Transmission Protocol – Potentially Failed

SCTP-PR Stream Control Transmission Protocol – Partial Reliability

SDN Software-Defined Networking

SDT Secure Datagram Transport

SIMD Single Instruction Multiple Data

SPUD Session Protocol for User Datagrams

SRTT Smoothed RTT

STTF Shortest Transfer Time First

SDP Session Description Protocol

SIP Session Initiation Protocol

SLA Service Level Agreement

SPUD Session Protocol for User Datagrams

STUN Simple Traversal of UDP through NATs

TCB Transmission Control Block

TCP Transmission Control Protocol

TCPINC TCP Increased Security

TLS Transport Layer Security

TSN Transmission Sequence Number

TTL Time to Live

TURN Traversal Using Relays around NAT

UDP User Datagram Protocol

UPnP Universal Plug and Play

URI Uniform Resource Identifier

VoIP Voice over IP

VM Virtual Machine

VPN Virtual Private Network

WAN Wide Area Network

WWAN Wireless Wide Area Network

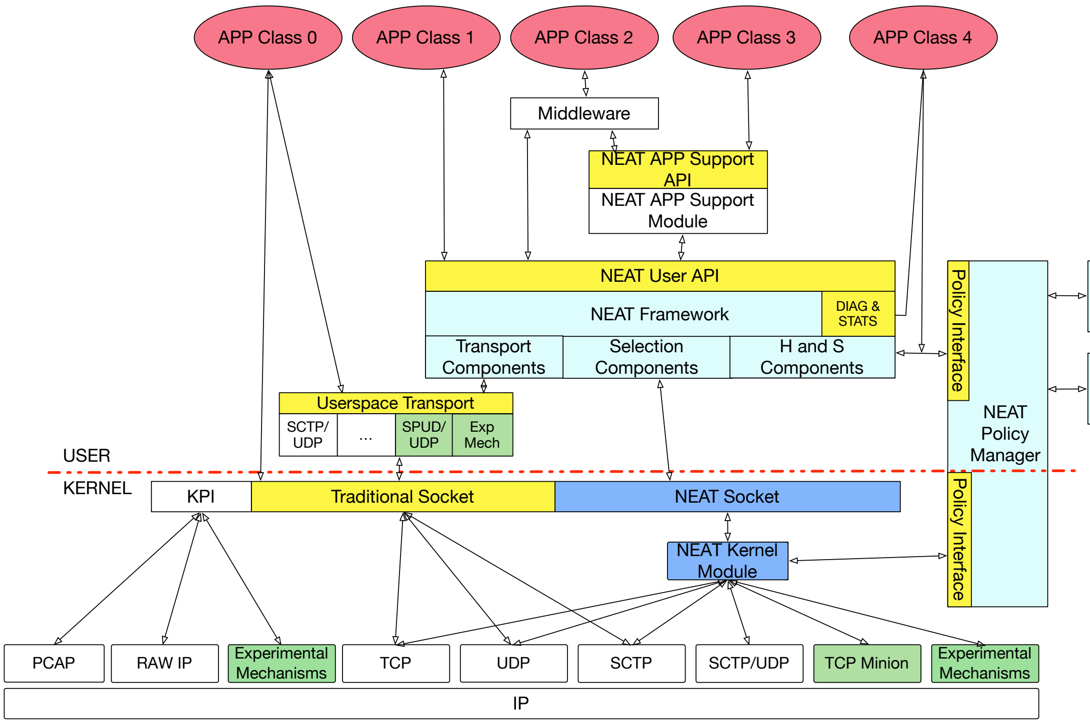

Figure 1: Components and interfaces to the NEAT System, as described in Deliverable D1.1. The NEAT User Module is composed of all the blocks shown in light blue (NEAT Framework, NEAT Transport, NEAT Selection, NEAT Signalling and Handover, and Policy Components) and related APIs (NEAT User API, Policy Interface, Diagnostics and Statistics Interface).

1 Introduction

The NEAT Project has defined a new architecture, presented in Deliverable D1.1 [5] and outlined in Figures 1 and 2, that changes the transport layer interface exposed to Internet applications. By presenting a new API that allows applications to provide information that describes properties of the required service, a NEAT System enables the stack to automatically choose an appropriate protocol. This seemingly simple change can have massive ramifications, because it allows flexible usage of a range of protocol components underneath the new user interface. This can enable the best possible use of the protocols/services that are available end-to-end along a given network path or paths.

This document summarises the final work done in WP1. It provides the conclusion of the architectural analysis in WP1, presenting the final specification of transport services and the final abstract Application Programming Interface (API) for the NEAT System. In the following paragraphs, we describe how this document has evolved from the initial NEAT User API presented in Deliverable D1.2 [12].

The work includes refinements to the architecture and the abstract API, updating the information in D1.2. This original work was guided by the NEAT use cases [5] to realise an API design process based on the IETF TAPS Working Group documents. The focus of the present document is to provide an integrated view of the services and API that can be read together with the final version of the Core transport system in Deliverable D2.3 [7].

The current document follows implementation experience after completing the validation and performance analysis (milestone MS7)—for example, this includes understanding the implications of policy decisions, and experience in integrating NEAT Selection mechanisms.

The number of API primitives and events covered has grown since D1.2, which was in turn largely

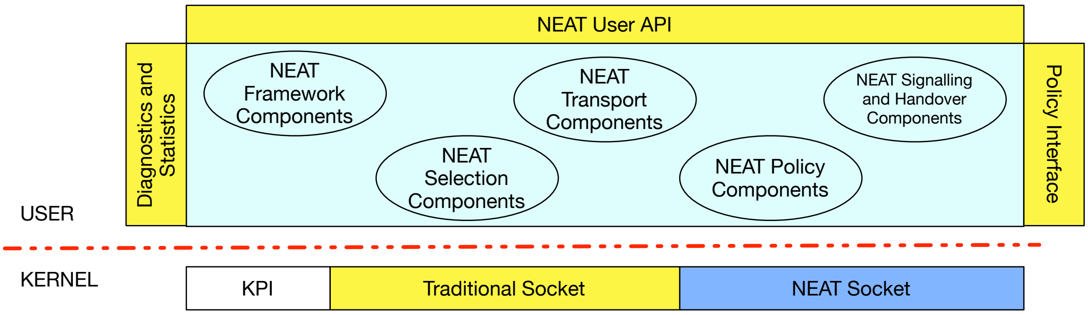

Figure 2: The groups of components and external interfaces used to realise the NEAT User Module, as described in Deliverable D1.1. The NEAT User Module utilises the lower interface provided by a Kernel Programming Interface (KPI), the traditional Socket API or an optional NEAT Socket API. The focus of the present document is on the NEAT User API providing transport services to applications.

based on an earlier version of the TAPS "usage" Internet draft [11], authored by NEAT participants. The present deliverable reflects the many changes as this Internet draft moved towards Internet Consensus and eventual publication as an RFC[1]. It also benefits from the insights from a companion Internet draft describing UDP and UDP-Lite [4], also authored by NEAT participants.

The resulting API is therefore not strictly a superset of the original version in D1.2. The reasons for this are quite diverse, and are explained in detail in Appendix B.

While the Internet drafts cited before were used as input to the present document, there were at least two good reasons *not* to try to incorporate the *whole* content of these drafts into the NEAT User API. From the analysis in [10], these reasons are as follows:

1. Primitives and events that relate to functionality that could under some circumstances be automatically provided underneath the application are not always good to expose, as the application using them then limits the flexibility of the underlying system.

2. Some primitives and events that cannot be replaced with similar functionality from TCP or UDP should not be offered, as they prevent the system from falling back to TCP or UDP. SCTP's "payload protocol-id" is such a function: it is essentially a number that can be transferred out-of-band, "aside" (but logically connected to) an association. TCP cannot do that; hence, if an application explicitly relies on this functionality, it cannot be made to run over TCP if SCTP is not available.

A Minimal Set of Transport Services for IETF TAPS Systems has been produced as a result of work in NEAT, and is documented in a TAPS Internet Draft [10], included in Appendix D. If one was to implement an API using the reasons above as design principles, one would arrive at an API resembling such "minimal set". The NEAT User API as described here implements this minimal set of transport services, but extends this beyond the constraints set by the IETF TAPS working group Charter[2]. Notably, both the NEAT System and the "minimum set" are designed to work one-sided, i.e., a NEAT host can most efficiently talk to another NEAT host, but it can also talk to a NEAT-unaware host via TCP or UDP. Such one-sided deployment greatly facilitates the gradual introduction of the NEAT System into the Internet.

The minimal set in [10] removes multi-streaming from the API altogether, as the decision to use multi-streaming does not require application-specific knowledge (and then, the transport system un-

[1]At the time of writing, an RFC number was not available yet; the status of the draft was: *Submitted to IESG for Publication.*
[2]https://tools.ietf.org/wg/taps/charters

derneath can decide to automatically map application flows onto transport streams). NEAT does implement this functionality; however, in NEAT, streams can also be used directly by the application programmer. This can allow a NEAT application to communicate with a non-NEAT enabled SCTP application.

In some cases, an API primitive or event planned in D1.2 was found to provide functionality that the consortium believed to be better implemented by other means (e.g., specifying the send buffer size vs. using the "low watermark" functionality described in [10]). In other cases, some functionalities were found to be better expressed as system policies. In some other cases, after further analysis the functionality was simply found not to be needed for the industry use cases.

The remainder of the document is structured as follows. Section 2 describes the NEAT User API, defining the set of primitives and events that compose this API. The main body of the document concludes in Section 3. Common NEAT-specific terms are defined in Appendix A. The rationale for the main changes in the API from D1.2 to D1.3 is presented in detail in Appendix B. Appendix C provides a set of examples of policies and how they are used in NEAT. Finally, Appendix D includes a copy of the "Minimal Set" Internet draft at the time of writing, for reference.

2 The NEAT User API

Note: *The description of the NEAT User API presented in this section* **replaces** *that in Deliverable D1.2, reflecting the status at the end of Work Package 1 activities.*

2.1 Overview

The NEAT architecture defines a callback-based design realised as events provided by the NEAT User API. The implementation details of these functions are reviewed in D2.3 [7], together with examples of use with real code. In contrast, this document focusses solely on the *abstract* NEAT User API.

Possible events and primitives related to NEAT flows are described in § 2.2. This covers the communication functionality of the NEAT System. Following the common style in IETF RFCs, these primitives and events are described in an abstract fashion, i.e., the description is *not* bound to a specific programming language. The *semantics* associated with the API primitives and events are fully described here; however, the NEAT implementation, as embodied by the C-language prototype presented in D2.3 [7], may differ in *syntax* from this API.

The NEAT primitives and events can be categorised according to the following taxonomy, based on whether a call pertains to a *NEAT flow* per se or to the *data* carried by such NEAT flow:

- Manipulating a NEAT flow:
  - Initialisation (§ 2.2.1).
  - Establishment (§ 2.2.2).
  - Availability (§ 2.2.3).
  - Maintenance (§ 2.2.4).
  - Termination (§ 2.2.5).
- Manipulating data:

– Writing and reading data (§ 2.2.6).

An application using the NEAT System must take the following steps to utilise the network (see also § 2):

1. Initialisation: a) create a NEAT flow by calling **P:** INIT_FLOW; then b) call **P:** SET_PROPERTIES to express the application's requirements. This is used by NEAT's Policy Manager and is necessary to avoid unwanted outcomes, e.g., to avoid a choice of UDP for an application that requires reliability, or the use of TCP for an application that prefers unordered delivery (these cases are explained in greater detail in Appendix D).

2. Establishment / Availability: Connect (actively or passively) the NEAT flow.

3. Use the flow to transfer data; call maintenance API primitives as needed to configure the flow.

4. Termination: Close (or abort) the NEAT flow.

An example of an application interacting with NEAT is shown in Figure 3.

Table 1 lists all the primitives and events that constitute the NEAT User API. The *Category* column refers to the taxonomy introduced above. The last column points to the relevant section of this document where each component of the API is described in detail.

A NEAT Flow has a set of *properties* which are set at flow initialisation time, and it has *attributes* which can be read by an application once a flow has been initialised (see Table 2). Properties are related to Transport Features. For instance, the *link-layer security*, *transport-layer security*, *certificate verification*, *certificate* and *key* properties set at initialisation time (§ 2.2.1) are related to a *Confidentiality* Transport Feature.

2.1.1 Notation and presentation style

We describe next the notation and presentation style used in the remainder of the document.

Each primitive/event is associated with a particular NEAT flow, and the primitives and events for manipulating data can only be used after a NEAT flow has been created. However, for simplicity, the flow parameters are not shown.

The names of primitives and events are shown in small caps: LIKE THIS. **P:** and **E:** respectively indicate primitives and events. Their parameters are shown *in italics* with optional parameters shown in square brackets: *[like this]*. A triangle (▷) indicates the explanation of a primitive or event.

2.2 API Primitives and Events

The Transport Features offered by the NEAT User API are described as follows:

- Transport Features that require immediate action (or feedback) from NEAT are presented as *primitives*.

- Transport Features that require immediate action from the application are presented as *events*.

- Transport Features that require adjusting properties before a NEAT Flow is opened are presented in the Initialisation category.

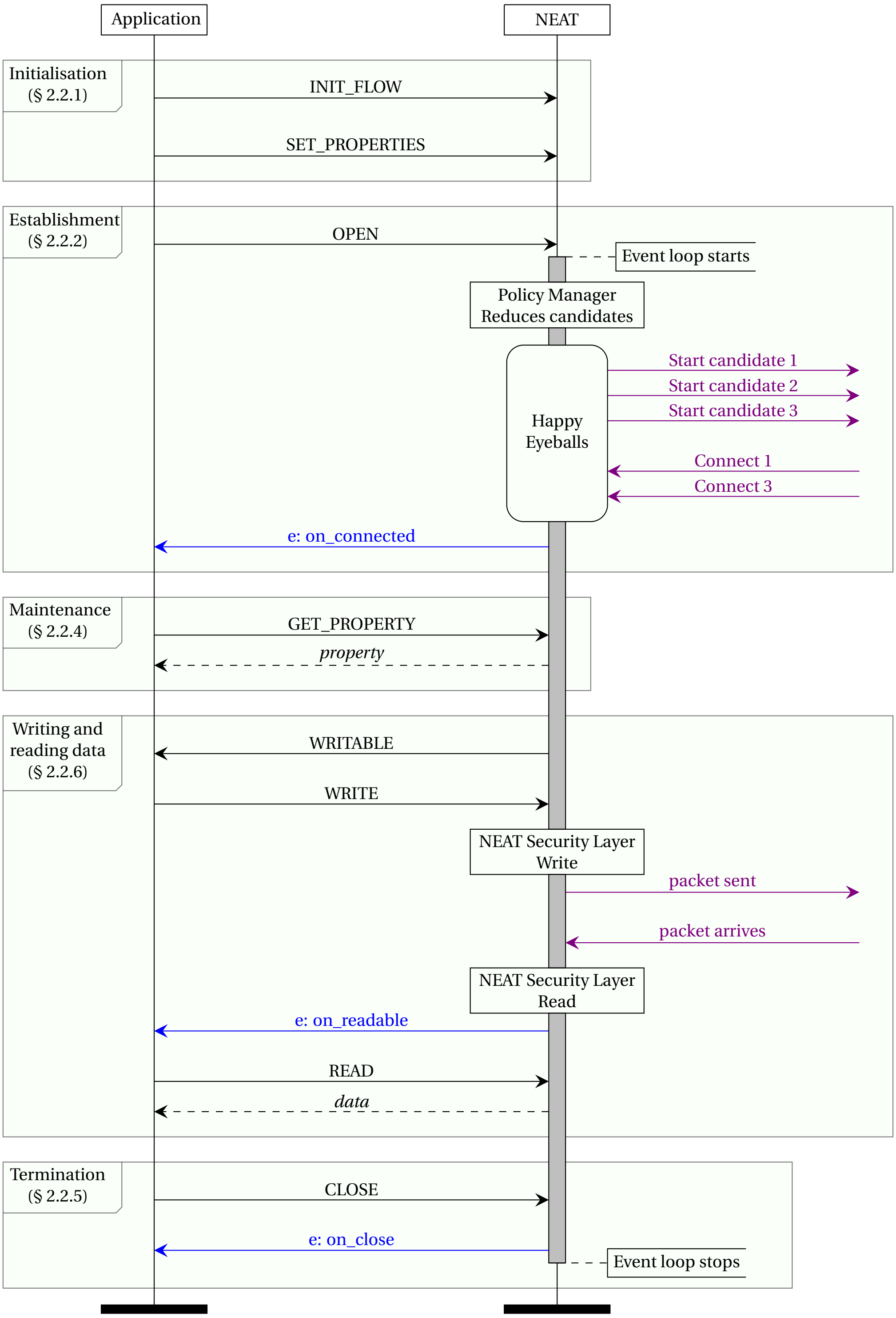

Figure 3: Message Sequence Chart (MSC) illustrating an application making a NEAT-based connection. Messages in blue are specific to our callback based implementation and not part of the more general abstract API described in this document.

For many API primitives and events, syntactical decisions regarding the way they are presented here were guided by the way they have been presented in the related TAPS documents [4, 11] (which,

Table 1: NEAT User API Primitives and Events.

Type	Category	Name	Section
Primitives	NEAT Flow Initialisation	INIT_FLOW SET_PROPERTIES	2.2.1
	NEAT Flow Establishment	OPEN OPEN_WITH_EARLY_DATA	2.2.2
	NEAT Flow Availability	ACCEPT	2.2.3
	NEAT Flow Maintenance	CHANGE_TIMEOUT SET_PRIMARY SET_LOW_WATERMARK SET_MIN_CHECKSUM_COVERAGE SET_CHECKSUM_COVERAGE SET_TTL GET_PROPERTY	2.2.4
	NEAT Flow Termination	CLOSE ABORT	2.2.5
	Writing and reading data	WRITE READ	2.2.6
Events	NEAT Flow Maintenance	NETWORK_STATUS_CHANGE	2.2.4
	NEAT Flow Termination	CLOSE ABORT TIMEOUT	2.2.5
	Writing and reading data	WRITABLE	2.2.6

in turn, depend on IETF RFCs), as well as the system dynamics underlying the respective Transport Features that the API offers. This only defines a method to present an abstract API; it does not limit the NEAT implementation itself, and different implementations of the same abstract API are possible. The NEAT Library implemented in WP2 takes a particular approach — i.e., using the Policy Manager to achieve greater flexibility in use. Other implementations could choose to implement primitives to communicate each property/parameter separately and directly across the API.

2.2.1 NEAT Flow Initialisation

The primitives below are called before the flow is opened.

P: INIT_FLOW()

▷ This primitive must be called before calling **P:** OPEN (§ 2.2.2), **P:** OPEN_WITH_EARLY_DATA (§ 2.2.2) or **P:** ACCEPT (§ 2.2.3), and will return an error otherwise.

P: SET_PROPERTIES(*propertyList*)

propertyList : a JSON String describing the properties of the flow.

▷ This primitive can be called after **P:** INIT_FLOW to express the application's requirements.

Table 2: Examples of Properties and Attributes related to a NEAT Flow. These are controlled and accessed via different API calls, invoked at different times in a NEAT Flow's lifetime, as indicated by the *Category* column.

Category		**Name**	**Section**
NEAT Flow Initialisation		Link-layer security Capacity profile Transport-layer security Peer certificate verification Security certificate Public key NEAT Flow disable handover NEAT ECN Enable NEAT Flow metadata NEAT Flow group NEAT Flow priority DSCP value	2.2.1
NEAT Flow Maintenance	Read-only flow attributes	NEAT transport parameters Interface statistics Path statistics Used DSCP	2.2.4

The (abstract) properties that can be set with this call include:

- **Link-layer security**: Boolean that, if true, requests selection of a local interface that provides some form of link layer security (e.g., to avoid open WiFi networks). Default: false.
- **Capacity profile**: One out of four values defining what kind of dynamic behaviour the NEAT Flow should have: 1) LBE (e.g., LEDBAT [9] congestion control), 2) conservative (e.g., CAIA Delay Gradient congestion control [6]), 3) normal (e.g., TCP-friendly "Reno-like" [1] congestion control), 4) aggressive (e.g., CUBIC [8] congestion control). This is purely advisory, if one of these capacity profiles is requested but is not available, or if this property is not set, the system's default behaviour will be used (e.g., 3 for FreeBSD, 4 for Linux).
- **Transport-layer security**: If this boolean property is included, it specifies a preference for using a secure connection. If true, this means: *must* use a secure connection, whereas false means: *try* to use a secure connection. Default: false.
- **Peer certificate verification**: If this boolean property is used, it specifies a preference for the validation of the peer certificate. A value of true means: must validate, while a value of false means: it will not be validated. Default: true.
- **Security certificate**: This property specifies a file that contains a certificate that is to be used. If this is not specified, no certificate will be used.
- **Public key**: This property specifies a file that contains a public key to be used. If neither this property nor **Security certificate** are specified, no private key is used. If **Public key** is not specified but **Security certificate** is, the private key will be taken from **Security certificate**.

- **NEAT Flow disable handover**: This boolean property allows to disable the "seamless handover" functionality of NEAT. This can be useful for applications that implement their own handover functionality, to avoid function duplication. Default: false.
- **NEAT ECN Enable**: This boolean property indicates a NEAT Flow can initiate use of Explicit Congestion Notification (ECN). Default: true.
- **NEAT Flow metadata**: Information about the flow such as the type and name of the application, the length of the flow in bytes, the expected duration, etc. Default: no information.
- **NEAT Flow group**: This integer number identifies groups of flows—all flows having the same number and the same destination belong to a common group. Flows in one group should obtain common congestion management, allowing a chosen **NEAT Flow priority** (see below) to play out between these flows, e.g., because it is believed that they share the same network bottleneck. The default value is 0.
- **NEAT Flow priority**: This defines a priority value P for the NEAT Flow. The word "priority" here relates to a desired share of the capacity such that an ideal NEAT implementation would assign the NEAT Flow the capacity share $P \times C/sum\_P$, where P = priority, C = total available capacity and sum_P = sum of all priority values that are used for the NEAT Flows in the same NEAT Flow group. The implementation of per-flow priorities is local, meaning that it may yield unexpected behaviour when it interferes with prioritisation inside the network (e.g., when additionally setting a **DSCP value**). The priority setting is purely advisory; no guarantees are given. Default: 1.
- **DSCP value**: The (abstract) DSCP value that the application desires to use for all sent messages of the NEAT Flow. No guarantees are given regarding the actual usage of the DSCP value on packets. Adjusting this property is expected to mostly be useful for datagram services. Care should be taken when adjusting this value, in particular when changing it on an already active flow as this can impact ordering and congestion control [2]. Default: 0.

2.2.2 NEAT Flow Establishment

The two primitives below allow the creation of a NEAT Flow from one transport endpoint to one or more transport endpoints.

P: OPEN(*destname port [stream_count]*)

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

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

stream_count : the number of requested streams to open (integer). Note that, if this parameter is not used, NEAT 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.

▷ This primitive opens a flow—actively for transports that require a connection handshake (e.g., TCP, SCTP), and passively for transports that do not (e.g., UDP, UDP-Lite). Note that 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 peer's **P:** ACCEPT only returns when data arrives, this may only happen after the local host has called **P:** WRITE (NEAT's actual callback-based implementation does not have this problem because its **P:** ACCEPT does not block anyway).

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 in § 2.2.6 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 : data to be sent.

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

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

▷ To accommodate TLS 1.3 early data and the TCP Fast Open option, 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, 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. Note that the supplied data can be delivered multiple times (replayed); an application must take this into account when using this function — this is commonly known as *idempotence*.

2.2.3 NEAT Flow Availability

The primitive below is used to receive incoming communication requests.

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).

▷ This primitive prepares a flow to accept communication from another NEAT endpoint. UDP and UDP-Lite do not natively support a POSIX-style accept mechanism; in this case, NEAT emulates this functionality. Note that **P:** ACCEPT may only return once data arrives, not necessarily after the peer has called **P:** OPEN (NEAT's actual callback-based implementation does not have this problem because its **P:** ACCEPT does not block anyway).

2.2.4 NEAT Flow Maintenance

Primitives and Events 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.

▷ This primitive adjusts the time after which a NEAT Flow will terminate if data could not be delivered. If this primitive 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.

▷ This primitive is meant to be used on NEAT Flows having multiple destination IP addresses, with protocols that do not use load sharing. It should not have an effect otherwise. Note that, in case a contradictory parameter is used when writing data, it will overrule this general per-flow setting. If this primitive 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.

▷ This primitive allows the application to limit the amount of unsent data in the underlying socket buffer. If set, NEAT will only execute **E:** WRITABLE (§ 2.2.6) when the amount of unsent data falls below the watermark. This allows applications to reduce the sender-side queuing delay.

P: SET_MIN_CHECKSUM_COVERAGE(*length*)

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

▷ This primitive allows an application to set the minimum acceptable checksum coverage length 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.

▷ This primitive allows an application to set the number of bytes covered by the checksum in a UDP-Lite datagram.

P: SET_TTL(*ttl*)

ttl : sets the minimum TTL or Hop Count on a datagram before it will be passed to the application.

E: NETWORK_STATUS_CHANGE( )

Returns: status code.

▷ This event informs the application that something has happened in the network; it is safe to ignore without harm by many applications. The 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 the property name.

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

▷ Allows an application to discover the value assigned to a property by the Policy Manager.

Flow maintenance properties The **P:** GET_PROPERTY primitive allows to obtain flow maintenance properties, expressed as part of *policies* and handled by NEAT's Policy Manager. These are properties that either can be adjusted after flow initialisation (§ 2.2.1), or they are *attributes* of a flow that can only be read by an application once a flow has been initialised (*read-only*). These are:

- **NEAT 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** property (§ 2.2.1) if that property is specified — but such property does not indicate a concrete congestion control algorithm, which this readable attribute returns. More generally, this attribute gives the application a more concrete view of the choices made by the NEAT System.

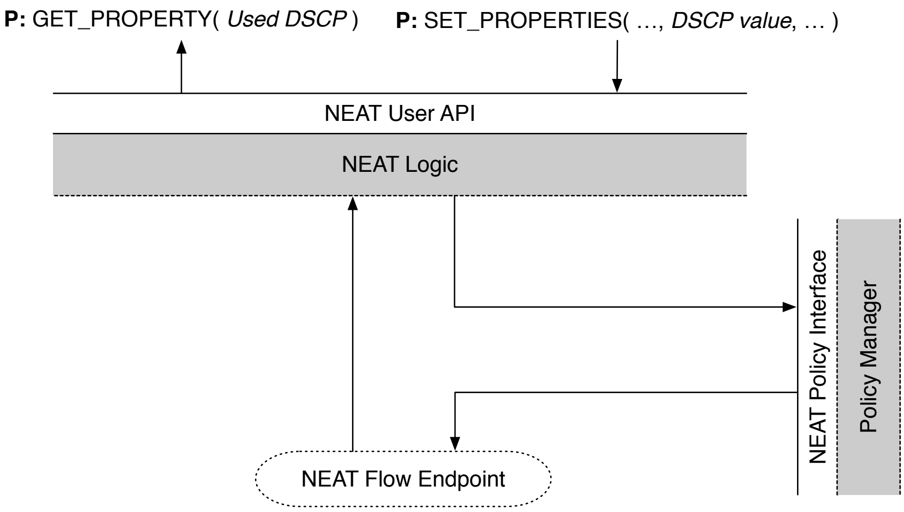

Figure 4: Example of setting and reading flow properties and attributes, respectively, and interaction with NEAT's Policy Manager.

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

Figure 4 provides an example of the relation between the NEAT User API, properties/attributes and the Policy Manager. Suppose the application wants to specify an abstract QoS marking to be used in all of its packets. The application passes this value to NEAT as a **DSCP value** property, via **P:** SET_PROPERTIES. The code implementing the NEAT User API (labeled as "NEAT Logic" in the figure) passes this information to the Policy Manager (PM), via the Policy Interface. The PM instantiates this abstract QoS, using local policy, in a concrete DSCP value that will be used by the NEAT Flow; the mapping from abstract to concrete QoS marking done by the PM could for instance be based on Table 3 (taken from D2.3 [7]). If it wishes to, the application can later query the DSCP value that is actually used (i.e., the **Used DSCP** attribute) via **P:** GET_PROPERTY.

Appendix C provides examples of properties and policies — as actually implemented in the NEAT prototype described in D2.3 [7] — and their expected result.

2.2.5 NEAT Flow Termination

The next primitives and events are related to gracefully or forcefully closing a NEAT Flow, or informing the application about this happening.

P: CLOSE( )

▷ This 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.

Table 3: Possible Abstract QoS to DSCP Mappings in NEAT. Some traffic classes such as Video can have several different capacity requirement levels, the NEAT System exposes these with Very Low, Low, Medium and High capacity requirements. Applications can also request Admitted access, classes that can be guaranteed by the network with policy or dynamic provisioning.

Abstract Name	DSCP Code	DSCP Value
NEAT_QOS_AUDIO_VL	CS1	0x08
NEAT_QOS_AUDIO_L	DF	0x00
NEAT_QOS_AUDIO_M1	EF	0x2E
NEAT_QOS_AUDIO_H1	EF	0x2E
NEAT_QOS_INTERACTIVE_VIDEO_VL	CS1	0x08
NEAT_QOS_INTERACTIVE_VIDEO_L	DF	0x00
NEAT_QOS_INTERACTIVE_VIDEO_M1	AF42	0x24
NEAT_QOS_INTERACTIVE_VIDEO_M2	AF43	0x26
NEAT_QOS_INTERACTIVE_VIDEO_H1	AF41	0x22
NEAT_QOS_INTERACTIVE_VIDEO_H2	AF42	0x24
NEAT_QOS_NON_INTERACTIVE_VIDEO_VL	CS1	0x08
NEAT_QOS_NON_INTERACTIVE_VIDEO_L	DF	0x00
NEAT_QOS_NON_INTERACTIVE_VIDEO_M1	AF32	0x1C
NEAT_QOS_NON_INTERACTIVE_VIDEO_M2	AF33	0x1E
NEAT_QOS_NON_INTERACTIVE_VIDEO_H1	AF31	0x1A
NEAT_QOS_NON_INTERACTIVE_VIDEO_H2	AF32	0x1C
NEAT_QOS_DATA_VL	CS1	0x08
NEAT_QOS_DATA_L	DF	0x00
NEAT_QOS_DATA_M1	AF11	0x0A
NEAT_QOS_DATA_H1	AF21	0x12
NEAT_QOS_BROADCAST	CS3	0x18
NEAT_QOS_REALTIME_INTERACTIVE_DATA	CS4	0x20
NEAT_QOS_IMMERSIVE_AUDIO	AF41	0x22
NEAT_QOS_IMMERSIVE_VIDEO	AF41	0x22
NEAT_QOS_STREAMING	AF31	0x1A
NEAT_QOS_BACKGROUND	CS1	0x08
NEAT_QOS_ADMITTED_AUDIO	EF	0x2E
NEAT_QOS_ADMITTED_VIDEO	AF42	0x24
NEAT_QOS_ADMITTED_IMMERSIVE_AUDIO	AF42	0x24
NEAT_QOS_ADMITTED_IMMERSIVE_VIDEO	AF42	0x24
NEAT_QOS_ADMITTED_DATA	AF42	0x24

E: CLOSE( )

▷ This event informs the application that a NEAT Flow was successfully closed.

P: ABORT( )

▷ This primitive terminates a connection without delivering remaining data.

E: ABORT( )

▷ This event informs the application that the other side has aborted the NEAT Flow.

E: TIMEOUT( )

▷ This event informs the application that the NEAT Flow is aborted because the default timeout

— possibly adjusted by the **P:** CHANGE_TIMEOUT NEAT Flow maintenance primitive (§ 2.2.4) — has been reached before data could be delivered.

2.2.6 Writing and reading data

All primitives in this section refer to an open NEAT Flow, i.e., a NEAT Flow that was either actively established or successfully made available for receiving data.

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 : data to be sent.

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

▷ This primitive gives NEAT a data block for transmission to the other side of the NEAT Flow (with reliability limited by the conditions specified via *pr_method*, *pr_value* and the transport protocol used). If the NEAT Flow supports message delimiting, the data block is a complete message.

P: READ( )

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.

data : received data.

datalen : the amount of data received.

▷ This primitive reads data from a NEAT Flow into a provided buffer. If the NEAT Flow supports message delimiting, the data block is a complete message.

E: WRITABLE( )

▷ This event informs the application that the NEAT Flow is ready to accept new data.

3 Conclusion

This document presented a stable “final” version of the NEAT User API, based on the rationale outlined in Deliverable D1.2, but updated from what NEAT *should* implement (D1.2) to what NEAT currently *does* implement. It reflects the agreement of the NEAT consortium on the exposed functionality of NEAT.

The document reviews the primitives and events related to NEAT Flow initialisation, NEAT Flow establishment, NEAT Flow availability, NEAT Flow maintenance, reading and writing network data and NEAT Flow termination.

IETF documents typically describe their API in terms of a traditional socket-like function, in an abstract, language-independent form; this is the form adopted here. A programmer wishing to view the concrete API in D2.3 [7] is therefore referred to the NEAT Library tutorial and documentation (available from the main NEAT Project web page, at: https://www.neat-project.org/resources). Also, deliverable D2.3 provides examples of code utilising the API and descriptions of the way in which the callback mechanisms can be used.

References

[1] M. Allman, V. Paxson, and E. Blanton, "TCP Congestion Control," RFC 5681 (Draft Standard), Internet Engineering Task Force, Sep. 2009. [Online]. Available: http://www.ietf.org/rfc/rfc5681.txt

[2] D. Black and P. Jones, "Differentiated Services (Diffserv) and Real-Time Communication," RFC 7657 (Informational), Internet Engineering Task Force, Nov. 2015. [Online]. Available: http://www.ietf.org/rfc/rfc7657.txt

[3] ECMA, *ECMA-404: The JSON Data Interchange Format*. Geneva, Switzerland: ECMA (European Association for Standardizing Information and Communication Systems), 2013. [Online]. Available: http://www.ecma-international.org/publications/standards/Ecma-404.htm

[4] G. Fairhurst and T. Jones, "Features of the User Datagram Protocol (UDP) and Lightweight UDP (UDP-Lite) Transport Protocols," Internet Draft draft-fairhurst-taps-transports-usage-udp, May 2017, work in progress. [Online]. Available: http://www.ietf.org/internet-drafts/draft-fairhurst-taps-transports-usage-udp-00.txt

[5] G. Fairhurst (ed.), T. Jones (ed.), Z. Bozakov, A. Brunstrom, D. Damjanovic, T. Eckert, K. Evensen, K.-J. Grinnemo, A. Hansen, N. Khademi, S. Mangiante, P. McManus, G. Papastergiou, D. Ros, M. Tüxen, E. Vyncke, and M. Welzl, "NEAT Architecture," NEAT Project (H2020-ICT-05-2014), Deliverable D1.1, Dec. 2015. [Online]. Available: https://www.neat-project.org/publications/

[6] D. A. Hayes and G. Armitage, "Improved coexistence and loss tolerance for delay based TCP congestion control," in *Proc. of the IEEE Local Computer Networks (LCN)*, Denver, Colorado, USA, Oct. 2010, pp. 24–31. [Online]. Available: http://dx.doi.org/10.1109/LCN.2010.5735714

[7] N. Khademi, Z. Bozakov, A. Brunstrom, O. Dale, D. Damjanovic, K. R. Evensen, G. Fairhurst, A. Fischer, K.-J. Grinnemo, T. Jones, S. Mangiante, A. Petlund, D. Ros, I. Rüngeler, D. Stenberg, M. Tüxen, F. Weinrank, and M. Welzl, "Final Version of Core Transport System," NEAT Project (H2020-ICT-05-2014), Deliverable D2.3, Aug. 2017. [Online]. Available: https://www.neat-project.org/publications/

[8] I. Rhee, L. Xu, S. Ha, A. Zimmermann, L. Eggert, and R. Scheffenegger, "Cubic for fast long-distance networks," Working Draft, IETF Secretariat, Internet-Draft draft-ietf-tcpm-cubic-01, January 2016, http://www.ietf.org/internet-drafts/draft-ietf-tcpm-cubic-01.txt. [Online]. Available: http://www.ietf.org/internet-drafts/draft-ietf-tcpm-cubic-01.txt

[9] S. Shalunov, G. Hazel, J. Iyengar, and M. Kuehlewind, "Low Extra Delay Background Transport (LEDBAT)," RFC 6817 (Experimental), Internet Engineering Task Force, Dec. 2012. [Online]. Available: http://www.ietf.org/rfc/rfc6817.txt

[10] M. Welzl and S. Gjessing, "A minimal set of transport services for TAPS systems," Internet Draft draft-ietf-taps-minset, work in progress, Oct. 2017. [Online]. Available: https://tools.ietf.org/html/draft-ietf-taps-minset

[11] M. Welzl, M. Tüxen, and N. Khademi, "On the usage of transport service features provided by IETF transport protocols," Internet Draft draft-ietf-taps-transports, Aug. 2017, work in progress. [Online]. Available: https://tools.ietf.org/html/draft-ietf-taps-transports-usage

[12] M. Welzl, A. Brunstrom, D. Damjanovic, K. Evensen, T. Eckert, G. Fairhurst, N. Khademi, S. Mangiante, A. Petlund, D. Ros, and M. Tüxen, "First Version of Services and APIs," The NEAT Project (H2020-ICT-05-2014), Deliverable D1.2, Mar. 2016. [Online]. Available: https://www.neat-project.org/publications/

A NEAT Terminology

This appendix defines terminology used to describe NEAT. These terms are used throughout this document.

Application An entity (program or protocol module) that uses the transport layer for end-to-end delivery of data across the network (this may also be an upper layer protocol or tunnel encapsulation). In NEAT, the application data is communicated across the network using the NEAT User API either directly, or via middleware or a NEAT Application Support API on top of the NEAT User API.

Characteristics Information Base (CIB) The entity where path information and other collected data from the NEAT System is stored for access via the NEAT Policy Manager.

NEAT API Framework A callback-based API in NEAT. Once the NEAT base structure has started, using this framework an application can request a connection (create NEAT Flow), communicate over it (write data to the NEAT Flow and read received data from the NEAT Flow) and register callback functions that will be executed upon the occurrence of certain events.

NEAT Application Support Module Example code and/or libraries that provide a more abstract way for an application to use the NEAT User API. This could include methods to directly support a middleware library or an interface to emulate the traditional Socket API.

NEAT Component An implementation of a feature within the NEAT System. An example is a "Happy Eyeballs" component to provide Transport Service selection. Components are designed to be portable (e.g. platform-independent).

NEAT Diagnostics and Statistics Interface An interface to the NEAT System to access information about the operation and/or performance of system components, and to return endpoint statistics for NEAT Flows.

NEAT Flow A flow of protocol data units sent via the NEAT User API. For a connection-oriented flow, this consists of the PDUs related to a specific connection.

NEAT Flow Endpoint The NEAT Flow Endpoint is a NEAT structure that has a similar role to the Transmission Control Block (TCB) in the context of TCP. This is mainly used by the NEAT Logic to collect the information about a NEAT Flow.

NEAT Framework The Framework components include supporting code and data structures needed to implement the NEAT User Module. They call other components to perform the functions required to select and realise a Transport Service. The NEAT User API is an important component of the NEAT Framework; other components include diagnostics and measurement.

NEAT Logic The NEAT Logic is at the core of the NEAT System as part of the NEAT Framework components and is responsible for providing functionalities behind the NEAT User API.

NEAT Policy Manager Part of the NEAT User Module responsible for the policies used for service selection. The Policy Manager is accessed via the (user-space) Policy Interface, portable across platforms. An implementation of the NEAT Policy Manager may optionally also interface to kernel functions or implement new functions within the kernel (e.g. relating to information about a specific network interface or protocols).

NEAT Selection Selection components are responsible for choosing an appropriate transport endpoint and a set of transport components to create a Transport Service Instantiation. This utilises information passed through the NEAT User API, and combines this with inputs from the NEAT Policy Manager to identify candidate services and test the suitability of the candidates to make a final selection.

NEAT Signalling and Handover Signalling and Handover components enable optional interaction with remote endpoints and network devices to signal the service requested by a NEAT Flow, or to interpret signalling messages concerning network or endpoint capabilities for a Transport Service Instantiation.

NEAT System The NEAT System includes all user-space and kernel-space components needed to realise application communication across the network. This includes all of the NEAT User Module, and the NEAT Application Support Module.

NEAT User API The API to the NEAT User Module through which application data is exchanged. This offers Transport Services similar to those offered by the Socket API, but using an event-driven style of interaction. The NEAT User API provides the necessary information to allow the NEAT User Module to select an appropriate Transport Service. This is part of the NEAT Framework group of components.

NEAT User Module The set of all components necessary to realise a Transport Service provided by the NEAT System. The NEAT User Module is implemented in user space and is designed to be portable across platforms. It has five main groupings of components: Selection, Policy (i.e. the Policy Manager and its related information bases and default values), Transport, Signalling and Handover, and the NEAT Framework. The NEAT User Module is a subset of a NEAT System.

Policy Information Base (PIB) The rules used by the NEAT Policy Manager to guide the selection of the Transport Service Instantiation.

Policy Interface (PI) The interface to allow querying of the NEAT Policy Manager.

Stream A set of data blocks that logically belong together, such that uniform network treatment would be desirable for them. A stream is bound to a NEAT Flow. A NEAT Flow contains one or more streams.

Transport Address A transport address is defined by a network-layer address, a transport-layer protocol, and a transport-layer port number.

Transport Feature Short for Transport Service Feature.

Transport Service A set of end-to-end features provided to users, without an association to any given framing protocol, which provides a complete service to an application. The desire to use a specific feature is indicated through the NEAT User API.

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

Transport Service Instantiation An arrangement of one or more transport protocols with a selected set of features and configuration parameters that implements a single Transport Service. Examples include: a protocol stack to support TCP, UDP, or SCTP over UDP with the partial reliability option.

B Reasons for changes from D1.2

It is not surprising that D1.3 includes new API elements with respect to D1.2: the NEAT code base has grown and new functionality has been added. When D1.2 was written, it was impossible to envision every single function that the NEAT consortium may find to be useful further down the road. However, some D1.2 functionality has also been removed or changed. This section explains the reasons for the most significant such changes that were made to the abstract API since Deliverable D1.2.

- **P:** INIT_FLOW parameter *messages*: this boolean parameter specified whether message boundaries are preserved (true) or not (false). We found this unnecessary because data can be handed over as messages in any case, along with properties such as "partial reliability" or "out-of-order", and still be handed over from NEAT to the application as a byte-stream. A receiving application can fully use messaging functionality as long as 1) messages remain intact (i.e., if NEAT begins to hand over a message, it must later continue with the remaining data of the same message until the message is complete), and 2) the receiving application is able to determine frame boundaries inside the received byte-stream on its own. This "Application-Framed Bytestream" (AFra-Bytestream) concept is explained in detail in [10].
- **P:** REQUEST_CAPACITY was removed: the NEAT consortium found it more useful to indicate the capacity needed by the application via policy.
- **E:** RATE_HINT( ), **E:** SLOWDOWN( ) and the NEAT Flow property **Flow metadata privacy** were removed because these events and this property relate to network signaling. For security / privacy reasons, the consortium has decided against the use of network signaling.
- **Optimise for continuous connectivity**: This boolean property was meant to enable or disable mechanisms that try to make communication more robust, perhaps at some cost (e.g., lower throughput). It was removed because the envisioned functionality was related to mobility, which is no longer a main focus of the project's final use cases.
- **NEAT flow disable dynamic enhancement**: This boolean property was meant to allow preventing NEAT from changing the behaviour of a flow on-the-fly. For example, changing the underlying transport protocol during the lifetime of a flow could be prevented with this. This functionality was removed for the sake of simplifying the API, as we did not implement such in-flight protocol changes (these would be mostly relevant in case of mobility).
- **NEAT flow delay budget**: This was a floating point number that would indicate a "delay budget" in milliseconds, to communicate more or less stringent time requirements. This functionality turned out to be unnecessary for Celerway's use case, and was hence not implemented, and removed from the abstract API.
- In D1.2, the property **NEAT flow low latency** allows to specify a desired maximum send buffer size (advisory only). We found that it would be better to replace this functionality with the ability to specify a "low watermark", where draining the buffer below a certain level will provoke an event. This event (**E:** WRITABLE()) has also been added.
- **P:** WRITE parameter *[priority]* was removed because this per-message priority was found to be unnecessary; it could also create consistency problems in conjunction with per-flow priorities.

- In **P:** READ, partial message delivery was removed because it was decided that this is not an important functionality for NEAT, at least for now. It can also get in the way of the "Application-Framed Bytestream" concept, see [10].
- **P:** OPEN_WITH_EARLY_DATA has been added to accommodate the TCP Fast Open option and TLS 1.3 early data.
- **P:** OPEN, **P:** ACCEPT and **P:** CLOSE explanations have been adapted to match the stricter open/-close semantics imposed by transparent stream mapping. For example, when a NEAT Flow is a stream of an already existing association, opening the flow may not have any effect on the wire, and can not be assumed to trigger "accept"; the peer may have to wait until actual user data is transferred.
- Some functions are now taken care of by the policy system instead of API primitives; these were removed and explained in Appendix C, which briefly introduces NEAT's policy system.

C Examples of Policy

The Policy Interface, depicted in Figure 1, is an important part of the NEAT System, taking input parameters read from configuration files and parameters passed via the NEAT User API. Using policy information the Policy Manager of the NEAT System can evaluate and enforce abstract and high-level policies, as introduced in the architecture description in D1.1 [5]. This provides a flexible way to implement features that the application requires or desires, without having to define specific NEAT User API calls for these features.

The Policy Manager is configured using a set of policy *profiles* supplied as JSON-formatted files. When used in this way, the application does not directly interact with the Policy Manager. The information held in the Policy Information Base (PIB) and Characteristics Information Base (CIB) is combined with the profile and is used to make policy decisions. Deliverable D2.3 [7] describes the operation of the Policy Manager.

The NEAT User API allows an application to inspect the outcome of the policy decisions taken by Policy Manager with the **P:** GET_PROPERTY primitive. Figure 4 shows this interaction through the Policy Interface.

Similarly, an application can use the **P:** SET_PROPERTIES primitive to pass application requirements to the Policy Manager (see Figure 4). These requirements are expressed as a JSON message, using the prototype described in D2.3 [7]. The use of messages encoded in JSON strings provides greater flexibility and extensibility, compared with using a pre-defined C-level API.

The operation of NEAT's Policy system and its relation to the *properties* of a NEAT Flow (introduced in § 2.1) is most easily explained using concrete examples. This appendix therefore provides a set of examples of policies and their usage in the NEAT System.

C.1 JSON format

JSON (JavaScript Object Notation) is a language-independent, lightweight data-interchange format, designed to be easy to read and write and is fully described in [3]. The text format of a message is based on a subset of the JavaScript Programming Language.

JSON is built on two structures:

- A collection of name/value pairs.
- An ordered list of values.

with the following tokens defined:

- **Object:** An unordered set of name/value pairs. An object begins with "{" (left brace) and ends with "}" (right brace). Each name is followed by ":" (colon) and the name/value pairs are separated by "," (comma).
- **Array:** An ordered collection of values. An array begins with "[" (left bracket) and ends with "]" (right bracket). Values are separated by "," (comma).
- **Value:** A string in double quotes, or a number, or true or false or null, or an object or an array. These structures can be nested.
- **String:** A sequence of zero or more Unicode characters, wrapped in double quotes, using backslash escapes. A character is represented as a single character string.

C.2 Profiles

The NEAT Policy system makes aggregate policy groups available as *profiles*. Profiles give an application a single point to access a common set of properties at once. The NEAT System has the following Profiles available:

- **default:** Used when none is specified.
- **elephant_flows:** Tagging of flows to mark them as requiring a large capacity.
- **low_latency:** Low Latency Profile.
- **low_latency_tcp:** Low Latency Profile for TCP.
- **reliable_transports:** Predefined set of reliable transport protocols (TCP, SCTP, SCTP/UDP).

Listing 1 is shown below as an example of one of these Profiles.

```
{
    "uid":"low_latency",
    "description":"low latency profile",
    "priority": 1,
    "replace_matched": true,
    "match":{
        "low_latency": {
            "precedence": 1,
            "value": true
        }
    },
    "properties":{
        "RTT": {
            "precedence": 1,
            "value": {"start":0, "end":50},
            "score": 5
        },
        "low_latency_interface": {
            "value": true, "precedence": 1},
        "is_wired_interface": {
            "value": true, "precedence": 2}
    }
}
```

Listing 1: Example of the Low Latency Profile.

C.3 Examples

C.3.1 Default Policy Profile

It is optional for a NEAT Application to use the policy system. When no properties are specified via **P:** SET_PROPERTIES, the NEAT System uses the default policy profile, shown in Listing 2.

```
{
    "uid":"default",
    "description":"default properties: profile matches ANY request",
    "policy_type": "profile",
    "priority": 0,
    "properties":{
        "transport": {
            "value": "reliable",
            "precedence": 0
        },
        "low_latency": {
            "value": false,
            "precedence": 0
        },
        "capacity_profile": {
            "value": "best_effort",
            "precedence": 0
        },
        "high_availability": {
            "value": true,
            "precedence": 0
        },
        "dscp_value": {
            "value": 0,
            "precedence": 0
        },
       "remote_ip": {
            "value": null,
            "precedence": 0,
            "score": 0
        },
       "port": {
            "value": null,
            "precedence": 0,
            "score": 0
        }
    }
}
```

Listing 2: Default Policy Profile.

C.3.2 Example of Transport Selection Properties

The API to Internet Transport that the NEAT System offers makes it possible for an application to leave transport selection to the NEAT System. Listing 3 is an example of the properties sent through the NEAT User API, using the **P:** SET_PROPERTIES call, where the NEAT System is allowed to attempt automatic selection between TCP and SCTP when selecting the transport protocol.

```
{
    "transport": {
```

```
        "value": ["tcp", "sctp"],
        "precedence": 1 }
}
```

Listing 3: JSON properties that select reliable transport protocols.

Listing 5 is an alternative example of an application requesting reliable transport protocols using the pre-defined `transport_profile` profile (shown in Listing 4); that is, passing the properties shown in Listing 5 via **P:** SET_PROPERTIES results in the profile displayed in Listing 4 to be selected, since this profile matches the request expressed via the passed JSON properties.

```
{
    "uid":"reliable_transports",
    "description":"reliable transport protocols profile",
    "policy_type": "profile",
    "priority": 2,
    "replace_matched": true,
    "match":{
        "transport": {"value": "reliable"}
    },
    "properties":[
        [{"transport": { "value": "SCTP", "precedence": 2, "score": 3}},
         {"transport": { "value": "TCP", "precedence": 2, "score": 2}},
         {"transport": { "value": "SCTP/UDP", "precedence": 2, "score": 1}}
         ]
    ]
}
```

Listing 4: Reliable Transport Profile.

```
{
  "transport": {
    "value":"reliable",
    "precedence":2 }
}
```

Listing 5: JSON properties that select the Reliable Transport Profile.

C.3.3 Multihoming Transport Protocol

Listing 6 is an example of properties specifying multihoming, that allows the NEAT System to perform selection between the use of MPTCP and SCTP.

```
{
    "transport": [
        {
            "value": "SCTP",
            "precedence": 1
        },
        {
            "value": "TCP",
            "precedence": 1
```

```
        }
    ],
    "multihoming": {
        "value": true,
        "precedence": 1
    },
    "local_ips": [
        {
            "value": "10.0.2.15",
            "precedence": 1
        },
        {
            "value": "192.168.56.2",
            "precedence": 1
        }
    ]
}
```

Listing 6: JSON properties that select the Multihoming Profile.

D Internet-draft: *A Minimal Set of Transport Services for TAPS Systems*

The following Internet Draft [10], a Working Group Item of the IETF Transport Services working group (TAPS), has been produced by project participants.

TAPS M. Welzl
Internet-Draft S. Gjessing
Intended status: Informational University of Oslo
Expires: April 25, 2018 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

Copyright Notice

Table of Contents

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:

- Disable checksum when sending
- Disable checksum requirement when receiving
- Specify checksum coverage used by the sender
- 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 possiblity 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:

- Configurable Message Reliability
- Ordered message delivery (potentially slower than unordered)
- Unordered message delivery (potentially faster than ordered)
- Request not to bundle messages
- 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:

- 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.

- 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 notifed 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-rtcweb-qos]).

low_watermark: a buffer limit (in bytes); when the sender has less then 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:

- The maximum frame size that may be sent without fragmentation, in bytes (or "not available")
- The maximum transport frame size that can be sent, in bytes (or "not available")
- The maximum transport frame size that can be received, in bytes (or "not available")
- 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 Gorry 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.

- 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:

- 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).

- 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.

- Request multiple streams
  Protocols: SCTP
  Automatable because using multi-streaming does not require application-specific knowledge.
  Implementation: see Appendix A.3.2.

- 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.

- 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.

- 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.

- 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.

- 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:

- 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.

- 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.

- Obtain requested number of streams
  Protocols: SCTP
  Automatable because using multi-streaming does not require application-specific knowledge.
  Implementation: see Appendix A.3.2.

- 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.

- 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.

- 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.

- 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.

- 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.

- 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)

- 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).

- 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.

- Set maximum burst
  Protocols: SCTP
  Automatable because it relates to knowledge about the network, not the application.

- 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.

- 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.

- 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.

- 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 transfering 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 transfering 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 distinugish 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:

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

AVAILABILITY:

- T,U: Listen
- T: Configure authentication

MAINTENANCE:

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

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

TERMINATION:

- T: Close after reliably delivering all remaining data, causing an event informing the application on the other side
- T: Abort without delivering remaining data, causing an event informing the application on the other side
- T,U: Abort without delivering remaining data, not causing an event informing the application on the other side
- 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

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

A.2.2.2. Receiving Data

- T,U: Receive data (with no message delimiting)
- U: Receive a message
- 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).

- T,U: Notification of send failures
- T,U: Notification that the stack has no more user data to send
- 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.

- 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.
- 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 assocations in case of SCTP, i.e. they apply to all streams in that assocation.

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 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

Disclaimer