NEAT

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

H2020-ICT-05-2014
Project number: 644334

Deliverable D1.2

First Version of Services and APIs

Editor(s): Michael Welzl
Contributor(s): Anna Brunstrom, Dragana Damjanovic, Kristian Riktor Evensen, Toerless Eckert, Gorry Fairhurst, Naeem Khademi, Simone Mangiante, Andreas Petlund, David Ros, Michael Tüxen

Work Package: 1 / Use Cases, System Architecture and APIs
Revision: 1.0
Date: March 1, 2016
Deliverable type: R (Report)
Dissemination level: Public

Abstract

NEAT has identified application requirements and associated NEAT service components from use cases in Task 1.1. To fulfill these requirements, the NEAT System must provide certain services, and the NEAT API must contain a set of primitives and events that allow applications to use these services. Because NEAT makes use of the functionality of transport protocols without statically binding them to applications, the NEAT API must also provide (generic) access to the features of the transport protocols that NEAT uses. This document presents generic API primitives and events that map to features of TCP and SCTP, as well as primitives, events and services that relate to NEAT's use cases.

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

Note

This is a public release of work-in-progress describing the initial release of the NEAT Services and APIs. This document is expected to be re-issued in electronic form as the specification is updated and will finally be replaced by the project deliverable D1.3 “Final version of services and APIs”.

Planned functionality to be added in future versions includes:

- Changes to Primitives and Events for UDP and UDP-Lite.
- Primitives and Events that appear in extensions to SCTP.
- Support for automating use of multi-streaming.
- Support for functionality akin to Fast Open for TCP and SCTP.
- Other updates to the API based on experience using the NEAT System.

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

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

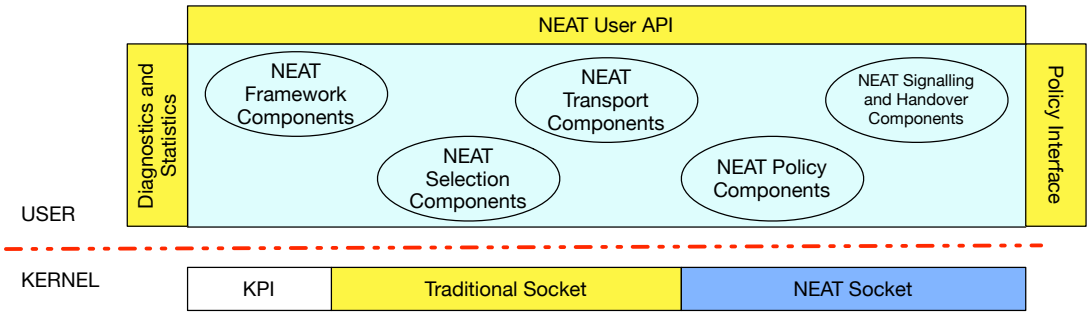


Figure 1: The groups of components and external interfaces used to realise the NEAT User Module. The NEAT User Module utilises the lower interface provided by a Kernel Programming Interface (KPI), the traditional Socket API or an optional NEAT Socket API.

1 Introduction

The NEAT System is designed to offer a flexible and evolvable transport system. Applications interface to the NEAT System through an enhanced API that effectively decouples them from the operation of the transport protocols and the network features being used. In particular, applications provide the NEAT System with information about their traffic requirements. Then, on the basis of these requirements, pre-specified policies, and measured network conditions, NEAT establishes and configures appropriate connections.

This document describes services that the NEAT System will provide together with the API that is offered to the applications using the NEAT System. It specifies the primitives of the upper layer interface of the NEAT User Module[1]. This API, known as the NEAT User API, offers services that are similar to those offered by the traditional socket API, but with the additional benefit that applications can provide parameters (properties) to allow each application to describe the service needed. The NEAT System may then use pre-specified policies to instantiate the required Transport Service.

Figure 1, taken from Deliverable D1.1 [7] shows the placement of the NEAT User API in the NEAT System. The main five NEAT System Component groupings (depicted as ovals) support the functionality provided by NEAT through the NEAT User API. Some of the functionality is quite directly reflected in the API, but much is automatised, and such automatisation can be done better using the rich information—richer than with the Berkeley socket API—that applications provide via the NEAT User API. This automatisation is enabled by extending the Internet's "best effort" service model all the way up to the application [11]: because not many "promises" are made, many possible choices are available for the NEAT system to satisfy application demands. This ability to automatise is enabled by the core idea behind all of NEAT's design, i.e., decoupling applications from the choice of a particular transport protocol.

NEAT works very closely with the Transport Services (TAPS) Working Group in the IETF[2]. As such, most of the documents in TAPS have authors from NEAT and NEAT's design is closely tied to work happening in TAPS. The NEAT Transport Services have been defined as *a set of Transport Service Features* that can be provided by a transport protocol [5]. Examples of Transport Service Features are: reliable delivery, ordered delivery, content privacy to in-path devices, and integrity protection. By zooming into a greater level of detail on what NEAT offers, this document is concerned with these Transport

[1] For more details about NEAT-specific terminology, please refer to Appendix A.

[2] http://tools.ietf.org/wg/taps/; this working group was chartered prior to NEAT following initiatives from several NEAT participants, such as *Birds of a Feather* (BoF) meetings and one Internet Draft predating the creation of the group [12].

Service Features rather than the compound Transport Services offered by transport protocols.

The NEAT User API has been designed in accordance with:

- The Transport Service Features provided by TCP and SCTP, as outlined in draft-ietf-taps-transports-usage [21] (Appendix C).
- The application requirements that were derived from the NEAT use cases in D1.1.
- The architectural design of the NEAT System, which is outlined in D1.1.
- The state-of-the-art, as outlined in the paper [13] included in Appendix B, specifically section IV of the paper (which discusses APIs).

The functionality provided by the NEAT User API is based upon two inputs:

- The functionality of the transport protocols that the NEAT System uses; in the current version these are TCP and SCTP[3].
- The application requirements that were derived from the use cases in D1.1.

The functionality of TCP and SCTP has been summarised in the form of API primitives and events in draft-ietf-taps-transports-usage [21], which provides the input for the first part of the NEAT API. The functionality that is necessary to fulfill the application requirements from the use cases in D1.1 is being implemented by NEAT; the Transport Service Features presented in Section 2.2 of this document (Tables 1 and 2) that are a part of the low-level Core Transport System are associated to Components in Deliverable D2.1 [9].

Figure 2 gives a high-level view of how the API is derived from the use cases and draft-ietf-taps-transports-usage [21], respectively. It shows the roles of draft-ietf-taps-transports-usage [21] as well as Deliverables D1.1 and D1.2 in this process: starting from two different sources, we arrive at one common abstract API definition in this document.

Section 2 presents the Transport Service Features that form the basis for API primitives and events in Section 3. The Transport Service Features from draft-ietf-taps-transports-usage [21] are already presented in the Internet-draft (included in full in Appendix C), and hence only a brief overview of this draft is given in § 2.1. The Transport Service Features that are associated with the use cases in D1.1 were developed as part of Tasks 1.1 and 1.3, and are summarised in § 2.2. In Section 3, after a brief overview of the general API structure and its envisioned usage (§ 3.1), the elements of the API are presented, separating the basic API elements (§ 3.2) from those stemming from the use cases (§ 3.3). Section 4 concludes by outlining planned functionality to be added in the next version of the API (to be reported in Deliverable D1.3).

2 Services provided by NEAT

NEAT services consist of elements that we call “Transport Service Features”, in line with the NEAT terminology and following the terms defined in [5]. Because Transport Service Features are the major concern for API design, these are at the centre of interest in the remainder of this document.

[3] For the moment, SCTP as defined in RFC 4960 [18] only; protocol extensions or alternate APIs such as RFC 6458 [19] will be covered in Deliverable D1.3, “Final Version of Services and APIs”.

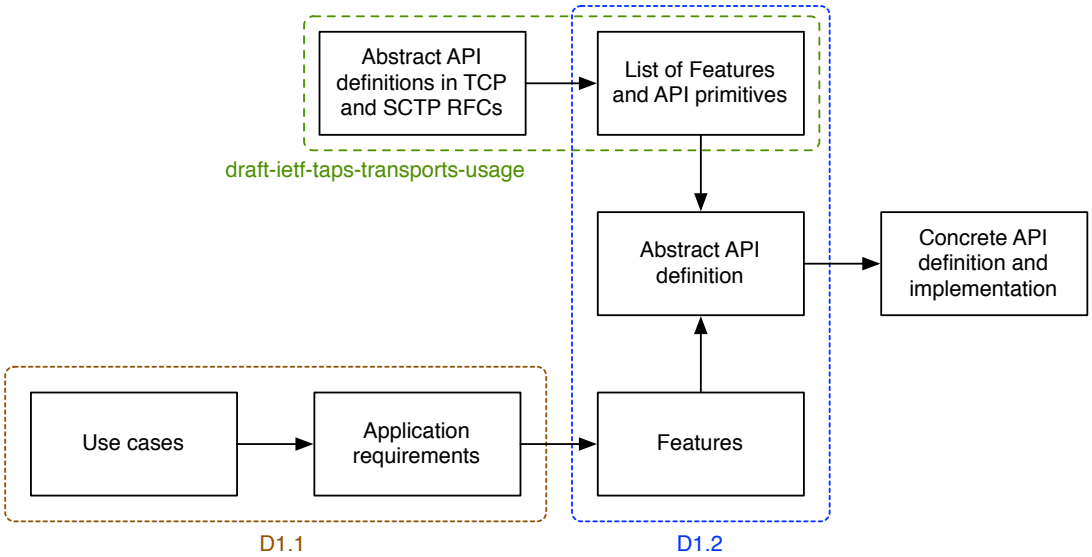


Figure 2: The process of deriving the API from draft-ietf-taps-transports-usage [21] and from NEAT's use cases.

2.1 Transport Service Features from draft-ietf-taps-transports-usage [21]

The Internet Draft draft-ietf-taps-transports-usage [21], written by NEAT participants as part of Task 1.3, derives Transport Services and Transport Service Features provided by transport protocols through an investigation of the IETF RFCs defining these. This method gives a reasonable overview of functionality that is available in most implementations that try to conform to the specification.

The draft follows a three-pass procedure: first, the relevant text pieces of RFCs are identified. Second, the draft presents the primitives and events that are exposed to applications as a means to utilise a protocol's Transport Service Features. Third, an overview of all Transport Service Features is derived, to capture the complete set of Transport Service Features even when some of them are *not* exposed in the API of a particular protocol because they are static properties of the protocol. For example, congestion control is a static property of TCP and SCTP which only stands out as a Transport Service Feature in conjunction with protocols such as UDP which do *not* support it. The TAPS Working Group has accepted draft-ietf-taps-transports-usage [21] with its three-pass method as a Working Group item.

For this document, the most relevant step in the procedure is pass 2. Here we have primitives and events together with some information about parameters and usage. The most important difference between the primitives in draft-ietf-taps-transports-usage [21] and the ones presented in Section 3.2 of this document is that the latter lack a transport protocol in their name—they are generic, allowing the NEAT System to make an automatic choice of the most suitable transport protocol given the policies, environment conditions and requests from the application.

2.2 Transport Service Features from Use Cases in D1.1 [7]

The NEAT consortium has identified a number of Transport Service Features that the NEAT System must provide in order to fulfil the application requirements associated with the NEAT use cases, presented in Deliverable D1.1.

Some Transport Service Features can be associated with an information flow from the NEAT System to the application, whereas others match an information flow in the other direction. Tables 1 and 2

Table 1: Information flow from NEAT to the application.

Use case	Requirement	Transport Service Features	Comments
EMC	Method to determine the transport selected and transport parameters	NEAT selected transport protocol; NEAT transport parameters	Choice of SCTP, TCP, etc. as well as parameters used (e.g. type of congestion control, TCP sysctl parameters, ...)
EMC	Statistics from the network	Interface statistics	Useful for MIF host interface selection or delay improvements (interface MTU, addresses, connection type (link layer), etc.)
Cisco; EMC	Ability to monitor and trace the performance over the network path being used; Latency, packet loss, ..., statistics from the network	Path statistics	Experienced RTT, packet loss (rate), jitter, throughput, path MTU, etc.
Cisco	Capacity hints	Capacity/bandwidth hints for application, upspeeding information	Comes either from the local NEAT endpoint or the remote NEAT endpoint in conjunction with enabled flow metadata signalling (see Table 2); can be boolean or a number (allowed rate in bit/s), depending on what is available
Cisco	Capacity hints	Downspeeding information	E.g., ECN marks passed to the application

present the Transport Service Features together with the use cases and requirements upon which they are based, both in the "upwards" and "downwards" direction. The Requirement column uses text that comes verbatim from D1.1; requirements separated by a semicolon correspond to the case when requirements from two industry use cases are mapped to the same Transport Service Feature. Semicolons are also used to separate multiple Transport Service Features.

In Table 2, all rows can be interpreted as *requests* from the application to NEAT: the application wishes to obtain something and asks the NEAT System for it. For the case where the application just hands over information to the NEAT System so that it can better optimise its behaviour (or signal the information into the network, such that other elements on the path can better optimise their behaviour), handing over the information is by itself a request, and all such requests are covered by the *NEAT flow metadata* Transport Service Feature (sixth row in the table).

3 NEAT User API

There are two main approaches regarding the interaction between a NEAT application and the NEAT User API to efficiently handle multiple concurrent connections (sockets): (1) the *traditional* approach used by the socket API; (2) an event-driven socket approach known as *callback*. On the one hand, with the traditional socket API, one can use e.g. `poll` to address the problem of handling multiple concurrent connections, use `fork` to run a process per socket, or combine these functions (e.g., by using multiple processes/threads that wrap `poll` or a process pool that file descriptors are passed to). On the other hand, the user of a callback-based API hands over a function pointer, and this function is expected to be called back (executed) at some relevant time. In a callback-based program, the readability/writability of a socket is returned using a registered callback function.

The NEAT System has chosen to implement the NEAT User API using callback-based functions,

Table 2: Information flow from the application to NEAT.

Use case	Requirement	Transport Service Features	Comments
Cisco	Naming	Connect to a name	Application only provides a name, not an IP address
Celerway	Network security	Selection of a secure interface	Boolean. E.g., avoid open WiFi
Celerway	Desire for seamless handover/mobility	NEAT flow seamless handover	Boolean (default=false)
Celerway	Robustness: continuous stable Internet connectivity	Optimise for continuous connectivity	Boolean; e.g., use LTE instead of a WiFi hotspot, use "NEAT flow seamless handover"
EMC	Disable dynamic enhancement	NEAT flow disable dynamic enhancement	Boolean; this is about changing the behaviour of a flow on-the-fly. It sets "NEAT flow seamless handover" to false
Cisco; Celerway; EMC	Signal QoS-related and non-QoS-related properties to the network; Metadata about application	NEAT flow metadata	Text (application type/name, flow length (size), duration, etc)
Cisco	Signal QoS-related and non-QoS-related properties to the network	Flow metadata privacy	Boolean per attribute (default=privacy) – signalling via "STUN by DISCUSS" IETF draft or similar PCP extensions; Choice of signalling can also be requested by application (e.g. STUN, PCP, SPUD,..)
Celerway; EMC	Delay budget; Latency expected by the application	NEAT flow delay budget	In ms
Celerway; EMC	Capacity requirements / profile; Capacity requested by the application	NEAT flow capacity	Application needs bit/s for the duration of t seconds (only a hint, no guarantee)
Celerway	Data delivery requirements, capacity requirements / profile	NEAT flow capacity profile	LBE (LEDBAT), conservative (CAIA Delay Gradient algorithm), normal (Reno), aggressive (Cubic)
Cisco; Celerway	Low latency; Data delivery requirements	NEAT flow low latency	Boolean (default = false)
Cisco	QoS	NEAT flow DSCP support	Uses DiffServ code points (DSCP; in-network priority); (maybe) simplified set of DSCP provided by NEAT
Cisco	Per-message priority	Per-message DSCP support	Uses DSCP (in-network priority); (maybe) simplified set of DSCP provided by NEAT. No policy decision here
Cisco	Per-message priority	Per-message priority	Also provides per-message priority (e.g., priority for video I-frames)—translates into send buffer policy, network-based decisions (use "per-message DSCP support"), etc.
Mozilla	Flow coordination	NEAT flow group	Common congestion management (application says which flows should have common congestion control), e.g. for the sake of prioritisation between flows
Mozilla	Flow coordination	NEAT flow priority	Local Priority in a NEAT host
Mozilla	API should provide multi-streaming	Multi-streaming	Just as in SCTP – for e.g. TCP, throw an error
Mozilla; EMC	Transport security; Security needs from the application	NEAT flow security	Parameters for (D)TLS, authentication, support for opportunistic use of encryption and integrity (TCPINC). Application says: "try if possible" or "must use"

because this approach is potentially faster and more scalable in terms of the number of concurrent flows for interfaces between two user-land libraries. A callback approach is becoming the new norm, and to NEAT-enable applications that use the traditional socket API (e.g., `select` or `poll`), a drop-in "shim" library may be developed.

To assist application developers unfamiliar with the callback approach, simple applications will be provided by NEAT developers as sample code that can use NEAT's callback-based User API. In addition, developers who have already implemented existing libraries using callbacks do not need to rewrite their applications to make NEAT callbacks happen (and the file descriptor created by the NEAT System itself is pollable). In addition, most of today's server-side applications are using the callback-based approach, making this a suitable choice for deployment.

3.1 Overview

The NEAT System uses a context structure called `neat_ctx`. A NEAT programmer needs to create it at the beginning of using the NEAT System (by calling NEAT_INIT_CTX,[4] which returns this structure), and destroy it when the application stops using the NEAT System (by calling NEAT_FREE_CTX, which receives the context structure as an argument). To maintain the correct context, all NEAT functions receive the context structure as an argument.

NEAT communication is inherently connection-oriented. A NEAT connection is called a "NEAT flow" (`neat_flow`). This is handled similar to connections in traditional socket programming: it must be created before usage, and destroyed after usage; once it has been created, it is possible to read from a NEAT flow or write to it, and parameters can be adjusted during its existence. This is achieved by using some of the primitives that are available (OPEN, ACCEPT, READ, WRITE, ...) and by registering operations to be performed when certain conditions occur (callbacks). NEAT starts performing these callbacks after its primitive NEAT_START_EVENT_LOOP is called. The `neat_ctx` and `neat_flow` data structures also contain general status information that can be queried.

To summarise, working with the NEAT System requires the following steps:

1. Create the context structure.
2. Create a NEAT flow.
3. Register callback functions for handling events.
4. Start the event loop.
5. Call API primitives as needed.
6. Destroy the NEAT flow.
7. Destroy the context structure.
8. Stop the event loop.

Possible events (to be registered as callbacks in step 3 above) and primitives (step 5) are described in §§ 3.2 and 3.3; 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

[4] The primitives NEAT_INIT_CTX, NEAT_FREE_CTX and NEAT_START_EVENT_LOOP are not described later in this document because they do not map to either draft-ietf-taps-transports-usage [21] or Tables 1 and 2. They are not directly related to any specific Transport Service Features.

description is not bound to a specific programming language. The *semantics* associated with the API primitives and events are fully described here; the final NEAT implementation may however differ in *syntax* from this API. This can include coalescing or splitting primitives and events—for example, instead of using multiple primitives to adjust properties of a NEAT flow, it would also be possible to expose a single SET_PROPERTIES primitive that receives a large data structure containing all the possible elements to be adjusted. The best form is a matter of programming convenience (and one can easily move from one to the other), but this does not change which communication capabilities are made available to the programmer.

NEAT primitives and events are categorised into:

- Manipulating a NEAT flow: *establishment* (e.g., OPEN), *availability* (e.g., ACCEPT), *maintenance* (e.g., CHANGE_TIMEOUT) and *termination* (e.g., CLOSE)
- Manipulating data (e.g., WRITE, READ, SEND_FAILURE).

All primitives/events are always associated with a particular NEAT context and flow, and the primitives/events for manipulating data can only be used when a NEAT flow has been created.

3.1.1 Notation and presentation style

For simplicity, the NEAT context and flow parameters are not shown. The names of primitives and events are shown in small caps: LIKE THIS. **P:** and **E:** indicate primitives and events, respectively. Their parameters are shown *in italics* and optional parameters are shown in square brackets: *[like this]*. A triangle (▷) indicates the explanation of a primitive or event. A right-pointing arrow (→) indicates comments that relate to how the API was derived, but are not a part of the actual API description.

3.2 API Primitives and Events from draft-ietf-taps-transports-usage [21]

The primitives and events in this subsection are based on pass 2 in [21], which lists primitives and events per protocol and tries to present a coherent picture across multiple protocols. In the process of moving from that list to generic API primitives and events, several minor design decisions needed to be made. For example, TCP can be passed a timeout value when opening a connection, whereas SCTP does not. Both protocols allow to adjust the timeout of an existing connection / association. The decision, here, was to ignore the timeout parameter of TCP and only rely on the possibility to adjust it later (i.e., an OPEN call can immediately be followed by a call to CHANGE_TIMEOUT by the NEAT user to obtain an effect similar to using TCP's timeout parameter when opening the flow). Similar types of minor decisions were made everywhere, with the goal of sustaining all the functionality of the protocols underneath the NEAT User API while presenting an API that is easy to use.

An effort was made to minimise protocol-specific behaviour, such as a primitive returning an error because the associated functionality is only available in SCTP or TCP, respectively. The following primitives/events from [21] were *not* included:

- **P:** REQUESTHEARTBEAT: because this primitive only exists in SCTP and it was considered to not be a very useful mechanism to expose.
- **P:** SETPROTOCOLPARAMETERS: because this primitive only exists in SCTP and allows to fine-tune a number of protocol-specific parameters.

- **P:** STATUS: because this functionality is already covered by the `neat_ctx` and `neat_flow` data structures.
- **P:** DISABLE-NAGLE: because this functionality is automatically enabled/disabled depending on the state of the **NEAT flow low latency** property (Table 2).
- **P:** CHANGE-DSCP: because this functionality is covered by the **DSCP value** property of the `neat_flow` data structure.

Moreover, TCP allows to specify whether a specific User Timeout (UTO) value should be advertised to the other side and whether an incoming UTO value should be accepted. This is an optimisation of the API to help in particular cases, and we have not considered this as a part of this first specification.

3.2.1 NEAT Flow Establishment

Active creation of a NEAT flow from one transport endpoint to one or more transport endpoints.

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

> *localname* : a NEAT-conformant name (which can be a DNS name or a set of IP addresses) to specify which local addresses to use; if this is missing, NEAT will make a default choice.
>
> *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). Default value: 1.
>
> **Returns:** success or failure. If success, it also returns a handle for a NEAT flow.

▷ This actively opens a flow.

→ Names are used instead of IP addresses to comply with the Transport Service Feature **Connect to a name** in Table 2. Failing when multi-streaming is not available is required by the Transport Service Feature **Multi-streaming** in Table 2.

3.2.2 NEAT Flow Availability

Preparing to receive incoming communication requests.

P: ACCEPT(*[name] port*)

> *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.
>
> **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 prepares a flow to accept communication from another NEAT endpoint.

3.2.3 NEAT Flow Maintenance

Adjustments to be made to an open NEAT flow, or notifications concerning the NEAT flow. These are out-of-band primitives / events 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 adjusts the time after which a NEAT flow will terminate if data could not be delivered. If this is not called, NEAT will make an automatic default choice.

P: SET_PRIMARY(*dst_IP_address*)

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

▷ This 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 is not called, the NEAT System will make an automatic default choice for the destination IP address.

→ This is derived from SETPRIMARY.SCTP in [21].

E: NETWORK_STATUS_CHANGE( )

Returns: status code.

▷ This 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 address(es) have become available/unavailable.

→ This is derived from ERROR.TCP and STATUS.SCTP (and therein, only NETWORK STATUS CHANGE from pass 1) in [21]. ERROR.TCP informs about soft errors that can be ignored without harm by many applications, including at least the notifications associated with values 1 and 2 of the status code. NETWORK STATUS CHANGE is an SCTP event that informs the user about a destination IP address becoming available/unavailable (value 3). Together with TCP's "soft errors", this is also called "safe to ignore without harm by many applications" because entire communication breakage will eventually also cause a TIMEOUT event (§ 3.2.4).

3.2.4 NEAT Flow Termination

Gracefully or forcefully closing a NEAT flow, or being informed about this event happening.

P: CLOSE( )

▷ This 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.

E: CLOSE( )

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

P: ABORT( )

▷ This terminates a connection without delivering remaining data and, if possible, sends an abort reason to the other side.

E: ABORT( )

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

E: TIMEOUT( )

▷ This informs the application that the NEAT flow is aborted because the default timeout, possibly adjusted by the CHANGE_TIMEOUT NEAT flow maintenance primitive (§ 3.2.3), has been reached before data could be delivered.

3.2.5 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. In addition to the listed parameters, all sending primitives contain a reference to a data block and all receiving primitives contain a reference to available buffer space for the data (both are omitted for readability).

P: WRITE(*[stream] [context] [pr_method pr_value] [dst_IP_address] [unordered_flag] [priority]*)

stream : the number of the stream to be used. This can be omitted if the NEAT flow contains only one stream.

context : a number that can possibly be used later to refer to the correct message when an error is reported.

pr_method and pr_value : if these parameters are used, then *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.

dst_IP_address : the destination IP address of the path that should be preferred, if possible and if there are multiple paths available (see also SET_PRIMARY, § 3.2.3).

unordered_flag : The data block may be delivered out-of-order if this flag is set. Default: false.

priority : This defines a floating point priority value from 0.1 to 1 for the message within the NEAT flow. The word “priority” here relates to a desired share of the capacity. It is recommended to send messages on different streams when they have different priorities. The implementation of per-message priorities is local. The priority setting is purely advisory; no guarantees are given. Default value: 0.5.

▷ This gives NEAT a data block for reliable (possibly limited by the conditions specified via *pr_method* and *pr_value*) transmission to the other side of the NEAT flow. NEAT flows can support message delineation as a property of the NEAT flow that is set via the INIT_FLOW primitive (§ 3.3.1). If a NEAT flow supports message delineation, the data block is a complete message.

→ The *priority* parameter was *not* derived from [21] but introduced here to support the Transport Service Feature **Per-message priority** in Table 2. According to [21], SCTP's `send` call (which many of the functions in this primitive were derived from) also has a `no-bundle_flag` parameter. This was removed here because there is no support for this per-message functionality in the socket API [19]. SCTP's `send` call also includes a payload protocol-id that is handed over to the receiver side. This was removed because NEAT follows the common method of only using port numbers for multiplexing at that layer. There is an additional benefit of the payload protocol-id in that it allows easier identification of packets inside the network; in NEAT, such information is considered metadata and only signalled when allowed by the user (see properties **NEAT flow metadata** and **Flow metadata privacy**, § 3.3.4). For partial reliability, [21] only specifies a "lifetime" parameter for SCTP, which RFC 7496 [20] extends to include the functionality that is captured by the parameters *[pr_method]* and *[pr_value]*, and which we have partially adopted here (it will probably be included in a future version of [21] too).

P: READ(*[stream]*)

stream : a stream number; if this is provided, the call to receive only receives data on one particular stream (else it receives on any stream).

Returns: *[partial_flag p_sequence_number] [unordered_flag u_sequence_number]*

If a partial message arrives, this is indicated by *partial_flag*, and then *p_sequence_number* is provided such that an application can restore the correct order of data blocks that comprise an entire message. If message arrives out of order, this is indicated by *unordered_flag*, and then *u_sequence_number* is provided such that an application can restore the correct order of messages.

▷ This reads data from a NEAT flow into a provided buffer. NEAT flows can support message delineation as a property of the NEAT flow that is set via the INIT_FLOW primitive. If a NEAT flow supports message delineation, the data block is a complete message.

→ At the time of writing, the SCTP code did not provide the ability to read from a specific stream—instead, it returns which stream was read from. This primitive therefore reflects functionality of a planned update to the SCTP code. SCTP's `receive` call also includes a payload protocol-id. This was removed because NEAT follows the common method of only using port numbers for multiplexing at that layer. There is an additional benefit of the payload protocol-id in that it allows easier identification of packets inside the network; in NEAT, such information is considered metadata and only signalled when allowed by the user (see properties **NEAT flow metadata** and **Flow metadata privacy**, § 3.3.4).

E: SEND_FAILURE( )

Returns: *cause_code [context] unsent_or_unacknowledged_msg*

cause_code indicates the reason of the failure, and *context* is the context number if such a number has been provided in WRITE. *unsent_or_unacknowledged_msg* is the message that could not be delivered.

→ This is derived from SENDFAILURE-EVENT.SCTP in [21]. TCP does not have a corresponding event.

3.3 API Primitives and Events from Transport Service Features in Tables 1 and 2

Section 3.1 explains that it does not matter to the functionality provided by NEAT whether the mechanisms are offered as many primitives with only a few parameters each, larger primitives spanning many parameters, or only as properties of the `neat_ctx` and `neat_flow` data structures. A definite decision regarding this aspect of NEAT design is not made in this document. In the previous section, these syntactical decisions were guided by the way primitives and events are presented in [21] (which, in turn, depends on how they are presented in the RFCs that [21] is based upon). For this section however, no such guidance exists. The decisions were made based on the assumed system dynamics underlying the Transport Service Features in Tables 1 and 2:

- Transport Service Features that require adjusting properties before a NEAT flow is opened are presented as part of a special INIT_FLOW primitive (§ 3.3.1).
- Transport Service Features that require immediate action (or feedback) from NEAT are presented as primitives (§ 3.3.2).
- Transport Service Features that require immediate action from the application are presented as events (§ 3.3.3).
- All other Transport Service Features are presented as elements of the `neat_ctx` and `neat_flow` data structures (§ 3.3.4).

This is just a method to present an abstract API; it does not mandate the final design of the reference implementation that is under development by the NEAT consortium.

The following Transport Service Features from Tables 1 and 2 are already covered by primitives or events in § 2.1:

- **Connect to a name**: covered by the OPEN and ACCEPT primitives.
- **Per-message priority**: covered by the *priority* parameter of the WRITE primitive.
- **Multi-streaming**: covered by the OPEN, WRITE and READ primitives.

3.3.1 Adjusting properties of a NEAT flow before it is opened

P: INIT_FLOW(*[messages] [LL_secure_local_interface] [capacity_profile]*
[security [certificate verification] [certificate] [key] [TLS/DTLS version] [cipher suites]])

messages : this boolean parameter specifies whether message boundaries are preserved (true) or not (false). Lack of message boundary preservation means that data are sent as a byte stream.

LL_secure_local_interface : boolean to request selection of a local interface that provides some form of link layer security (e.g., to avoid open WiFi networks).

capacity_profile : one out of four values defining what kind of dynamic behaviour the NEAT flow should have: 1) LBE (e.g., LEDBAT [17] congestion control), 2) conservative (e.g., CAIA Delay Gradient congestion control [8]), 3) normal (e.g., TCP-friendly "Reno-like" [1] congestion control), 4) aggressive (e.g., CUBIC [16] congestion control). This is purely advisory, if one of these capacity profiles is requested but is not available, the system's default behaviour will be used.

security : if this parameter is included, it specifies that a secure connection should be used. It can have two values: 1) must use a secure connection, 2) try to use a secure connection.

certificate_verification : if this parameter is used, it specifies that the peer certificate must be validated. It can have two values: 1) must validate, 2) validation is optional (the validation will be performed but it will not influence the connection establishment; the application can query the NEAT System to discover if verification succeeded or not).

certificate : this specifies a file that contains a certificate to be used.

key : this specifies a file that contains a public key to be used.

TLS/DTLS version : a list of TLS/DTLS versions to be advertised and accepted.

cipher suites : a list of cipher suites to be advertised and accepted.

▷ This primitive is used to initially adjust properties of a NEAT flow. It must be called before calling OPEN or ACCEPT (and will return an error otherwise). If the primitive is not called or a parameter is not used, the following default choices are made:

messages : true.

LL_secure_local_interface : 0.

capacity_profile : system default (e.g., 3 for FreeBSD, 4 for Linux).

security : 2.

certificate_verification : 2.

certificate : if certificate is not specified, no certificate will be used.

key : if key is not specified, no private key is used. If *key* is not specified and *certificate* is, the private key will be taken from *certificate*.

TLS/DTLS : 1.2, 1.1 and 1.0 for TLS and 1.2 and 1.0 for DTLS.

cipher suites : the default list will be defined as a policy.

→ The parameters cover, in sequence, the Transport Service Features **Selection of a secure interface**, **NEAT flow capacity profile** and **NEAT flow security** from Table 2. They were included in the INIT_FLOW call for the following reasons:

LL_secure_local_interface : this mainly influences the OPEN and ACCEPT calls.

capacity_profile : this cannot normally be changed on-the-fly, and quickly changing this would have quite unforeseeable consequences.

security : this is commonly decided at the beginning and may often not be possible to change afterwards. TCPINC is implicit for now.

3.3.2 Features that require immediate action / feedback from NEAT

P: REQUEST_CAPACITY(*rate time*)

rate : a requested sending rate by the application, in bits per second.

time : the duration over which this rate is requested.

▷ This primitive informs the NEAT System that the application needs the provided *rate* for duration *time*, starting at the moment this primitive is called. No guarantees are given, this is purely advisory.

→ This covers the Transport Service Feature **NEAT flow capacity** in Table 2.

3.3.3 Features that require immediate action / feedback from the application

E: RATE_HINT( )

Returns: *[new_rate]*

new_rate is the new maximum allowed sending rate in bit/s.

▷ If *new_rate* is missing, this event only notifies the application that it can send faster. This is purely advisory, NEAT gives no guarantees that e.g. a newly suggested sending rate will not lead to congestion.

→ This covers the Transport Service Feature **Capacity/bandwidth hints for application, upspeeding information** in Table 1.

E: SLOWDOWN( )

Returns: *ECN [new_rate]*

ECN is a boolean. If it is true, the hint to slow down comes from an encountered mark of Explicit Congestion Notification (ECN) [15]. *new_rate* is the new maximum allowed sending rate, in bit/s.

▷ This notifies the application to send more slowly, e.g., because there is congestion in the network. Note that if *ECN* = false, the hint may have been produced by other indications, e.g., packet loss, but it could also have been produced by an encountered ECN-mark that is already reacted upon inside or below NEAT.

→ This covers the Transport Service Feature **Downspeeding information** in Table 1.

3.3.4 Properties of the `neat_ctx` and `neat_flow` data structures

Properties that are *read* by applications are given the same names as the Transport Service Features covered in Table 1:

- **NEAT selected transport protocol**: Choice of SCTP, TCP, etc.
- **NEAT transport parameters**: Parameters used (e.g., congestion control mechanism, TCP sysctl parameters, . . .).
- **Interface statistics**: Interface MTU, addresses, connection type (link layer), etc.

- **Path statistics**: Experienced RTT, packet loss (rate), jitter, throughput, path MTU, etc.
- **Destination IP address information**: one or more destination IP addresses, information about which destination IP address is used by default.

Properties that are *adjusted* by applications are given the same names as the Transport Service Features covered in Table 2:

- **NEAT flow seamless handover**: This boolean property enables or disables the "seamless handover" functionality of NEAT. This can be useful for applications that implement their own handover functionality, to avoid function duplication. Default: false.
- **Optimise for continuous connectivity**: This boolean property enables or disables mechanisms that try to make communication more robust, perhaps at some cost (e.g., lower throughput). Default: false.
- **NEAT flow disable dynamic enhancement**: If this boolean property is set to true, it prevents NEAT from changing the behaviour of a flow on-the-fly. This means that, e.g., it will not use "seamless handover" even if this capability is enabled. Default: false.
- **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.
- **Flow metadata privacy**: This integer number controls the privacy of the information classified under **NEAT flow metadata**. It has the following meanings: 0: do not send metadata into or across the network (keep it only local). 1: freely send metadata into the network using any means. Other values will be reserved in the future for the application to specify the allowed signalling protocol / mechanism. Default: 0.
- **NEAT flow delay budget**: This floating point number indicates a "delay budget" in milliseconds. This can be used to communicate more or less stringent time requirements, such that data of the flow may reside in buffers for a longer or a shorter time. This is purely advisory, NEAT gives no delay guarantees.
- **NEAT flow low latency**: This property consists of a boolean that is used to indicate that a flow has a stronger need for low latency than for high throughput and a desired maximum send buffer size (advisory only). Among other things, this is expected to enable or disable methods that delay data transmission to increase the chance to send a full-sized segment.
- **NEAT flow group**: This integer number identifies groups of flows—all flows having the same number belong to a common group. Flows in one group should obtain common congestion management, allowing a chosen **NEAT flow priority** to play out between these flows, e.g., because it is believed that they share the same network bottleneck.
- **NEAT flow priority**: This defines a floating point priority value from 0.1 to 1 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 changing the **DSCP value**). The priority setting is purely advisory; no guarantees are given. Default: 0.5.

- **DSCP value**: The 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. This covers the Transport Service Features **NEAT flow DSCP support** and **Per-message DSCP support** in Table 2. It maps to CHANGE-DSCP.TCP in [21] and also exists for SCTP according to RFC 6458 [19]. 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].

4 Conclusion

This deliverable has built on the use cases defined in D1.1 and other inputs to identify requirements for the APIs and associated service components to realise a design for a transport system. This analysis has contributed to understanding of the best practice for using the existing transport protocols. The presented API definition is an important part of the NEAT System design.

When work on draft-ietf-taps-transports-usage [21] began, it was assumed that a generic API could be developed from the primitives and events that are described in it. This deliverable proves this assumption right, helping to show that NEAT's goal of enabling efficient protocol-independent communication is attainable. This also makes this deliverable useful as a basis for future inputs to the IETF TAPS Working Group. The API that was developed here is abstract, which makes it language-independent and allows the designer and the reader to focus on the functionality rather than syntactical details. Certainly, creating a protocol-independent transport system with its concrete API is easier when the expected capabilities are laid out in the form of an abstract API, as provided by this document. Here, NEAT functionality was presented in the form of primitives and events, or as readable/changeable properties of a NEAT flow. This decision was made on the basis of the previously defined TCP and SCTP APIs that were used as input to draft-ietf-taps-transports-usage [21] as well as the dynamic nature of the expected underlying functionality. A concrete API may exhibit its functionality in a different way.

The API primitives and functions in Section 3 represent the functions of TCP, SCTP as per RFC 4960 [18] as well as other agreed-upon functionality of the NEAT User API at the time of writing this document. Planned functionality to be added in future versions includes:

- Primitives and Events for UDP and UDP-Lite. The present document is strictly based on the current version of [21] as well as the Use Cases from D1.1. UDP was not considered here—however, there is already some support for UDP within the NEAT prototype implementation, and a recent NEAT-authored Internet draft [6] is a first step towards a similar process to incorporate UDP in the next revision of the API specification.

- Possible Primitives and Events that appear due to the extension of the SCTP part in a future version of draft-ietf-taps-transports-usage [21].

- We plan to automatise the use of multi-streaming such that it does not need to be a functionality that is visible to the application programmer. This means that, in the future, a NEAT Flow could be a TCP connection, an SCTP association or merely a stream inside an already existing SCTP association, for example.

- We plan to include functionality akin to TCP Fast Open [3], which has an impact on the API because it may deliver some data already upon opening a connection (NEAT Flow), yet this data may arrive as a duplicate. Since this functionality is not yet available in SCTP (but under development in NEAT Work Package 3 / Task 3.1), we have postponed its inclusion in the NEAT User API.

- The consortium has not yet decided if/how to exhibit control of sender-side buffers. This decision is not trivial. For instance, we think the functionality provided by the `TCP_NOTSENT_LOWAT` socket option [4] would be useful for a NEAT programmer who intends to minimize latency; usage of this mechanism, which controls the buffer space used for unsent data, is automatised in the NSURLSession and CFNetwork-layer APIs in MacOS X [10], yet the TCP socket API requires a number (the low water mark), so there appears to be a trade-off in either letting the programmer control this value or automatising its use. Moreover, NEAT buffers data internally, and, also for this buffer, it is not yet entirely clear to us whether control of this sender-side buffer should (only?) be automatised.

The API in this deliverable provides a basis for much of the implementation work in NEAT. As the project progresses we expect Task 1.4 to continue architectural analysis based on the design presented in D1.1 and D1.2. This will propose any needed refinements to the architecture, based on implementation experience, updated after completing the validation and performance analysis. This work has already started to contribute to the standards initiatives in Work Package 5, and as this work continues in Task 1.4 we expect this standards work to mature.

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] Y. Cheng, J. Chu, S. Radhakrishnan, and A. Jain, "TCP Fast Open," RFC 7413 (Experimental), Internet Engineering Task Force, Dec. 2014. [Online]. Available: http://www.ietf.org/rfc/rfc7413.txt

[4] E. Dumazet, "tcp: TCP_NOTSENT_LOWAT socket option," LWN.net, Jul. 2013, https://lwn.net/Articles/560082/.

[5] G. Fairhurst, B. Trammell, and M. Kuehlewind, "Services provided by IETF transport protocols and congestion control mechanisms," Internet Draft draft-ietf-taps-transports, Jan. 2016, work in progress. [Online]. Available: https://tools.ietf.org/html/draft-ietf-taps-transports

[6] 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, February 2016, work in progress. [Online]. Available: http://www.ietf.org/internet-drafts/draft-fairhurst-taps-transports-usage-udp-00.txt

[7] 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/

[8] 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

[9] N. Khademi (ed.), Z. Bozakov, A. Brunstrom, D. Damjanovic, K. Evensen, G. Fairhurst, K.-J. Grinnemo, T. Jones, S. Mangiante, G. Papastergiou, D. Ros, M. Tüxen, and M. Welzl, "First Version of Low-Level Core Transport System," NEAT Project (H2020-ICT-05-2014), Deliverable D2.1, Mar. 2016. [Online]. Available: https://www.neat-project.org/publications/

[10] P. Lakhera and S. Cheshire, "Your app and next generation networks," in *Apple Worldwide Developers Conference (WWDC)*, San Francisco, USA, Jun. 2015. [Online]. Available: https://developer.apple.com/videos/wwdc/2015/?id=719

[11] E. Lear, "Report from the IAB Workshop on Internet Technology Adoption and Transition (ITAT)," RFC 7305 (Informational), Internet Engineering Task Force, Jul. 2014. [Online]. Available: http://www.ietf.org/rfc/rfc7305.txt

[12] T. Moncaster (ed.), J. Crowcroft, M. Welzl, D. Ros, and M. Tüxen, "Problem statement: Why the IETF needs defined transport services," Internet Draft draft-moncaster-tsvwg-transport-services, work in progress, Dec. 2013. [Online]. Available: http://tools.ietf.org/html/draft-moncaster-tsvwg-transport-services

[13] G. Papastergiou, A. Brunstrom, G. Fairhurst, K.-J. Grinnemo, P. Hurtig, N. Khademi, D. Ros, M. Tüxen, M. Welzl, D. Damjanovic, and S. Mangiante, "De-ossifying the Internet transport layer: A survey and future perspectives," Under submission, Mar. 2016.

[14] G. Papastergiou, G. Fairhurst, D. Ros, A. Brunstrom, K.-J. Grinnemo, P. Hurtig, N. Khademi, M. Tüxen, M. Welzl, D. Damjanovic, and S. Mangiante, "De-ossifying the Internet transport layer: A survey and future perspectives," *IEEE Communications Surveys and Tutorials*, vol. 19, no. 1, pp. 619–639, 2017. [Online]. Available: http://ieeexplore.ieee.org/document/7738442/

[15] K. Ramakrishnan, S. Floyd, and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP," RFC 3168 (Proposed Standard), Internet Engineering Task Force, Sep. 2001, updated by RFCs 4301, 6040. [Online]. Available: http://www.ietf.org/rfc/rfc3168.txt

[16] 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

[17] 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

[18] R. Stewart, "Stream Control Transmission Protocol," RFC 4960 (Proposed Standard), Internet Engineering Task Force, Sep. 2007, updated by RFCs 6096, 6335, 7053. [Online]. Available: http://www.ietf.org/rfc/rfc4960.txt

[19] R. Stewart, M. Tuexen, K. Poon, P. Lei, and V. Yasevich, "Sockets API Extensions for the Stream Control Transmission Protocol (SCTP)," RFC 6458 (Informational), Internet Engineering Task Force, Dec. 2011. [Online]. Available: http://www.ietf.org/rfc/rfc6458.txt

[20] M. Tuexen, R. Seggelmann, R. Stewart, and S. Loreto, "Additional Policies for the Partially Reliable Stream Control Transmission Protocol Extension," RFC 7496 (Proposed Standard), Internet Engineering Task Force, Apr. 2015. [Online]. Available: http://www.ietf.org/rfc/rfc7496.txt

[21] M. Welzl, M. Tuexen, and N. Khademi, "On the usage of transport service features provided by IETF transport protocols," Internet Draft draft-ietf-taps-transports, Jan. 2016, work in progress. [Online]. Available: https://tools.ietf.org/html/draft-ietf-taps-transports-usage-00.txt

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 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 Paper: *De-ossifying the Internet transport layer: A survey and future perspectives*

The following research paper has been produced by project participants, and has been published in *IEEE Communications Surveys and Tutorials* [14].

De-ossifying the Internet transport layer: A survey and future perspectives

Giorgos Papastergiou, Anna Brunstrom, Gorry Fairhurst, Karl-Johan Grinnemo, Per Hurtig, Naeem Khademi, David Ros, Michael Tüxen, Michael Welzl, Dragana Damjanovic and Simone Mangiante

***Abstract*—It is widely recognized that the Internet transport layer has become ossified, where further evolution has become hard or even impossible. This is a direct consequence of the ubiquitous deployment of middleboxes that hamper the deployment of new transports, aggravated further by the limited flexibility of the Application Programming Interface (API) typically presented to applications. To tackle this problem, a wide range of solutions have been proposed in the literature, each aiming to address a particular aspect. Yet, no single proposal has emerged that is able to enable evolution of the transport layer.**

In this work, after an overview of the main issues and reasons for transport-layer ossification, we review proposed solutions and discuss their potential and limitations. The review is divided into five parts, each covering a set of point solutions for a different facet of the problem space: 1) designing middlebox-proof transports, 2) signaling for facilitating middlebox traversal, 3) enhancing the API between the applications and the transport layer, 4) discovering and exploiting end-to-end capabilities, and 5) enabling user-space protocol stacks. Based on this analysis, we then identify further development needs towards an overall solution. We argue that the development of a comprehensive transport layer framework that is able to facilitate the integration and cooperation of specialized solutions in an application-independent and flexible way is a necessary step toward making the Internet transport architecture truly evolvable. To this end, we identify the requirements that such a framework should fulfill and provide insights for its development.

***Index Terms*—Transport protocols, protocol-stack ossification, API, middleboxes, user-space networking stacks.**

I. Introduction and background

Networks can and do vary significantly in the set of functions they offer and their ability to move data between endpoints. The transport layer operates across the network and is responsible for efficient and robust end-to-end communication between network endpoints. This layer was designed to hide the details and variability of the network service from the applications that need to use it. The Internet's transport layer also contains other functions that are difficult or impossible to provide within a network, such as reliability, verification of delivery, flow control to prevent the application overwhelming the remote endpoint, congestion control to prevent the application from overwhelming the network, etc. People using the Internet mostly run applications that are based on the Transmission Control Protocol, TCP [1], which provides these transport functions.

Some applications need a different set of services to those offered by TCP. For example, a web client may wish to be able to prioritize sub-flows carrying specific objects, a multimedia flow may prefer timeliness to reliable delivery, and IP telephony can be tolerant to packet loss or in some case bit bit errors. There are many cases where TCP simply does not meet the need of applications—yet it ends up being used because it "just works", but not necessarily very well [2]. Applications that do not want the transport semantics of TCP typically just use the User Datagram Protocol, UDP [3]. While UDP provides flexibility that allows any set of services to be defined, every function needed has to be implemented at the application layer.

Some initiatives have developed alternate protocols to TCP, suited for other application types, for instance: the Datagram Congestion Control Protocol (DCCP) [4] was proposed to support streaming multimedia; the Stream Control Transport Protcol (SCTP) [5] originally targeted telephony signaling; UDP-Lite [6] supports error-tolerant audio and video services over wireless links. However, despite being standardized, with available implementations for common platforms, these transports are seldom seen in the general Internet, and TCP and UDP remain the only widely used transports.

A. Transport-layer ossification: overview of issues

Why do developers and users not adopt more modern protocols? It is not because new transports do not meet a real need. The following paragraphs examine the main reasons for this *ossification* of the transport layer.

1) Middleboxes: To become usable, a new transport needs to be made available to applications, requiring upgrades of both the sending and receiving endpoints. However, for a new transport to be adopted, the need to upgrade end-hosts is not the only obstacle to overcome. *Ossification of the network infrastructure* is probably the most significant barrier [7]–[10]: a transport protocol must be able to traverse the network; a new protocol is only useful if it is able to traverse paths on a larger part of the Internet. The ubiquity of middleboxes of a

G. Papastergiou and D. Ros are with Simula Research Laboratory, P.O. Box 134, 1325 Lysaker, Norway. E-mail: {gpapaste,dros}@simula.no.

A. Brunstrom, K.-J. Grinnemo and P. Hurtig are with Karlstad University, Universitetsgatan 2, 651 88 Karlstad, Sweden. E-mail: {anna.brunstrom,karl-johan.grinnemo,per.hurtig}@kau.se.

G. Fairhurst is with University of Aberdeen, AB24 3UE Aberdeen, United Kingdom. E-mail: gorry@erg.abdn.ac.uk.

N. Khademi and M. Welzl are with University of Oslo, P.O. Box 1072 Blindern, 0316 Oslo, Norway. E-mail: {naeemk,michawe}@ifi.uio.no.

M. Tüxen is with Münster University of Applied Sciences, Bismarckstraße 11, 48565 Steinfurt, Germany. E-mail: tuexen@fh-muenster.de.

D. Damjanovic is with Mozilla Corporation. E-mail: ddamjanovic@mozilla.com.

S. Mangiante is with EMC Corporation, Ovens, Co. Cork, Ireland. E-mail: simone.mangiante@emc.com.

variety of forms (from Network Address and Port Translators (NAPTs) to firewalls, accelerators, load-balancers, and a range of portals and more exotic devices) makes it very hard to change the status quo. Performing advanced network functions that go beyond the network layer, middleboxes not only need to understand the semantics of transport layer protocols, but some also tamper with protocol headers and thus violate end-to-end semantics [11]. As a result, any new native transport (layered directly on IP) is doomed to fail to pass through middleboxes until specific explicit support is added for that transport, while new extensions to standard transports (i.e., to TCP and UDP) are also vulnerable to potential middlebox interference [12].

If a protocol (or application) is widely used, then it is likely that there exists a business case to support the protocol. However, the motivation to support a protocol that has not yet reached wide-scale use is much weaker or non-existent. This creates a "tussle", described in [13], or similarly the "vicious circle" described in [2]. Quite simply, a new protocol will not be deployed over the Internet—because to do so would first require a business case, predicated on a user base that already have deployed the protocol. This ossification has resulted in little-to-no use of new transports for the last decade.

Even when TCP or UDP is used, middleboxes still cause significant connectivity problems to applications. For instance, since most NAPTs are built around the traditional client/server application model, they usually break end-to-end connectivity for applications that need direct communication between two arbitrary hosts, such as peer-to-peer applications [14]. While Application Layer Gateways (ALGs) are often used to embed application-specific knowledge into middleboxes to facilitate protocol traversal for particular applications, this solution has significant limitations in terms of deployment and scalability: a separate ALG is needed for each application protocol used (e.g., SIP [15], FTP [16], etc.) and hence all NAPTs need to be updated every time a new application (i.e., a new application protocol) needs to be supported. Security-related manipulation of TCP and UDP traffic performed by corporate firewalls and NAPTs can also cause significant connectivity problems in enterprise environments. Finally, some middleboxes expect only a certain application protocol like HTTP; in the face of such devices, the only solution is to tunnel connections over the supported protocol.

2) Application Programming Interfaces: A flexible and extensible API between the applications and the transport layer is essential for applications to be able to harness the benefits of new transport services. Today, the socket API essentially serves as the omnipresent application networking interface. However, it has become more and more apparent that this API is contributing to the Internet transport-layer ossification problem. Its simplicity may have led to its ubiquity, but has also held back the development of more enhanced APIs. This is evident in the currently ongoing standardization of the SCTP socket API—the SCTP transport protocol incorporates support for multihoming, but it is impossible to export this support through the standard socket API.

The very success of TCP and UDP has therefore led to *ossification of the API presented to applications*. These have now become the only widely available transports. This is reflected in the implementation of the socket API, which ties applications to a priori choices of transport protocol (either TCP or UDP). An application designed to work with one of these transports will need to be changed to support any new transport protocol.

The Internet has been designed so that transports only rely on core network functions, the so-called *Best-Effort* service. This has enabled transports to work across a diverse range of networks without having to know exactly how these provide the network service. However, this does not mean that information about what the transport/application needs from the network would not be helpful to improve the efficiency of the network or to enable the application to receive the most suitable service.

3) Other issues: An evolvable transport layer architecture requires that endpoints are capable of *discovering* if a new transport can be used: An endpoint initiating a communication session must know whether a transport (and any required transport options) are supported both along the end-to-end network path, and by the intended remote endpoint(s).

Except for some one-sided transport-layer mechanisms (e.g., the sender choice of a congestion control algorithm in TCP), the choice of a transport will require not only discovering the set of transports that are available at the remote endpoint, but also when more than one is supported at both ends, there needs to be an *agreement* from both endpoints on the choice of the particular transport.

Many network paths include middleboxes, some of which can, and often will, interfere with transport protocols. Endpoints need to assess whether a particular choice of transport can be safely used over the path.

Finally, one major additional challenge to deploying a new transport protocol is whether the transport protocol is supported across multiple OS platforms (e.g. Linux, FreeBSD, Mac OSX and Windows). Modifying OS kernel code can be costly in terms of deployment effort and often requires an OS update at the sender and/or the receiver to support the new transport, making any development effort platform-dependent.

B. Scope and structure of the paper

A range of point solutions have been proposed in the literature to address the above issues. Each covers a different aspect of the overall problem. In this paper, we review previous and ongoing efforts in the field. Our goal is to provide a better understanding of the pertained research issues, identify the potential and limitations of existing point solutions, and identify the need for further development.

We focus on evolutionary deployment. This restricts our survey to proposals that do *not* require redesigning the Internet architecture from scratch, hence, clean-slate approaches have been ruled as out of scope. Communication middleware is also beyond the scope of this survey, because such middleware usually provides a different communication abstraction to applications, rather than offering transport services different to those of a common networking stack.

Based on our analysis, we argue that proposing solutions in isolation cannot result in an Internet transport layer architecture that is truly evolvable and that a necessary step forward is the development of a comprehensive *transport layer framework* able to facilitate the integration and cooperation of new network and transport functions in an application-independent and flexible way. We therefore identify the requirements that such a framework should fulfill and provide insights for its development.

The remainder of the paper is organized as follows. Sections II to VI provide an overview of previous and ongoing efforts to tackle ossification of the Internet transport layer. The review is divided into five parts, each covering a different aspect of the overall problem:

- § II focuses on ways to design middlebox-proof transports, as a means to overcome the barriers imposed by middleboxes to the using new transport protocols and protocol features.
- § III is devoted to mechanisms that seek to better support end-to-end connectivity by facilitating traversal of middleboxes.
- § IV outlines approaches that aim to enhance the API between applications and the transport layer.
- § V examines approaches that allow endpoints to discover and agree on which protocols are supported along an end-to-end path.
- Lastly, § VI explores techniques for enabling user-space protocol stacks.

Section VII summarizes the survey and taxonomy of point solutions to transport ossification presented in §§ II–VI. Next, Section VIII analyzes the requirements for an evolvable transport framework. Finally, Section IX concludes the paper by identifying future research directions that may assist work in this area.

II. Design of middlebox-proof transports

There have been recent efforts to provide a richer set of transport services to applications than those provided by TCP and UDP within the constrained design space imposed by the ubiquitous deployment of middleboxes. These span two broad research directions: 1) extending TCP to provide a richer set of transport services, while guarding new extensions against potential middlebox interference, and 2) building new application-specific transports on top of UDP or TCP to ensure they transparently pass through existing middleboxes.

A. Extending TCP to offer a richer set of transport services

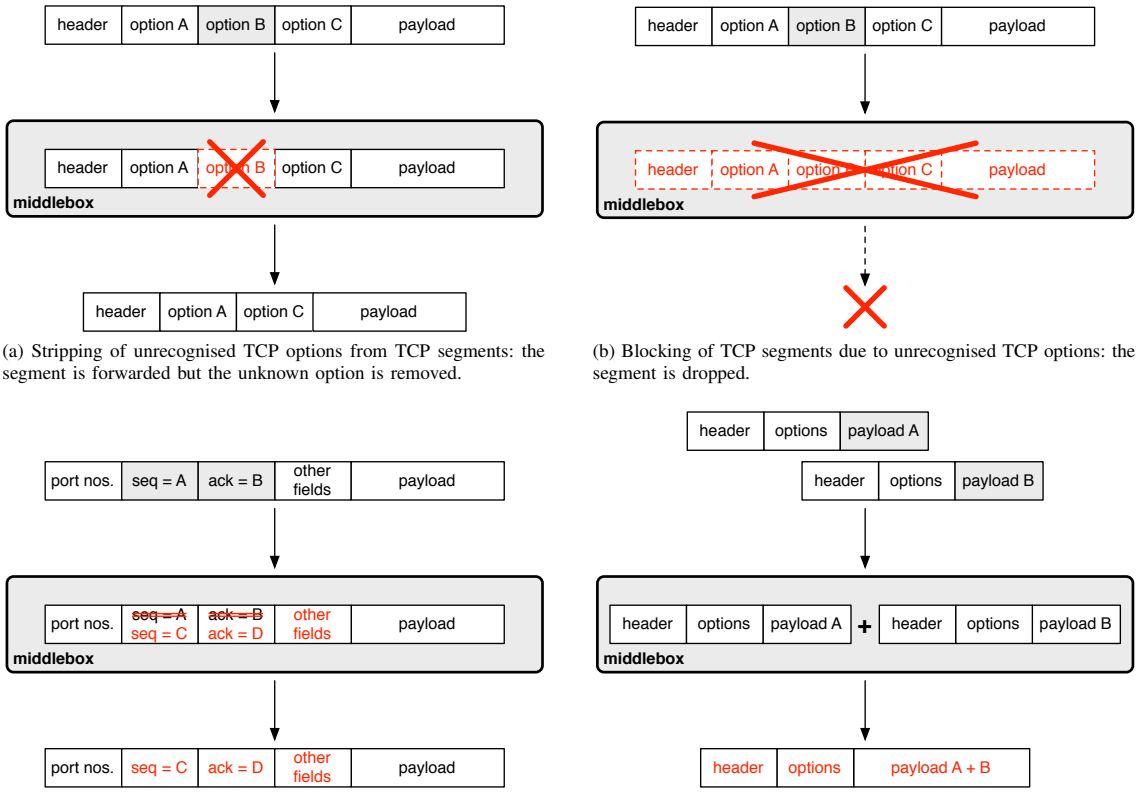
TCP is an extensible protocol. It can negotiate protocol extensions during connection establishment and exchange additional control information throughout the lifetime of a connection. During the last decade, measurement studies have investigated how existing middleboxes interact, either intentionally or unintentionally, with TCP extensions, how prevalent these interactions are, and to what extent they affect TCP's extensibility [10], [12], [17]–[19]. Examples of middlebox behavior (some of which are illustrated in Fig. 1) include: blocking or stripping of unknown TCP options, modification of TCP header fields and options (such as the Initial Sequence Number (ISN) and the Maximum Segment Size (MSS) option), re-segmentation or coalescence of TCP segments, and behavior triggered by "non-stereotypical" TCP communication seen on the wire. These empirical studies provide a first demarcation of the solution space and the first guidelines for designing middlebox-proof TCP extensions [12].

Multipath TCP (MPTCP) [12], [20], [21], Tcpcrypt [22], [23], and Gentle Aggression [24] are prominent examples of TCP extensions whose design was highly influenced by the need to account for known middlebox behavior. Techniques were adopted to guard extended operations against potential middlebox interference. For instance, a fallback strategy to plain TCP is incorporated in all approaches to handle cases where extended operations fail (e.g., when options are stripped from SYN or regular packets, or when payload modification is detected). This ability to fall back to plain TCP assures stability and is considered an important design goal for achieving widespread deployment. Relative sequence numbers are considered when encoding sequencing information within the new options to cope with potential re-writing of sequence numbers. Other techniques include the use of an additional data-level sequence space in MPTCP that allows it to maintain consistent sequence numbering on the wire while ensuring in-order data delivery over multiple subflows. Tcpcrypt was intentionally designed to exclude fields from the authentication header that could be expected to be modified by the path.

Recent work has identified the need for TCP to infer in-path alterations of packet header fields as a way to enable deployment of new TCP functions. Craven et al. [19] proposed TCP HICCUPS, an enhancement that allows TCP to detect packet header manipulation at field-level granularity and take appropriate actions (such as disabling a non-compatible extension) based on the middlebox behavior observed on a path.

TCP has a limited maximum header size. This led the designers of Tcpcrypt to the exchange of encryption information within the TCP payload (i.e., the body of the INIT1 and INIT2 sub-options). This highlights a significant factor that constrains the design space of TCP extensions: The limited space constrains the number and the extent of TCP options that can be simultaneously used by a TCP connection.

Extending the TCP option space has become an active research area that faces similar middlebox-related issues. For instance, Ramaiah [25] presents several middlebox considerations for designs to increase the TCP options space and reviews approaches proposed up to 2012. More recent proposals include TCP Extended Data Offset (EDO) [26], [27], TCP SYN Extended Option Space (SYN-EOS) [28], and Inner Space [29]. TCP EDO extends the option space in all packets except the initial SYN packets (i.e., SYN and SYN/ACK) using a TCP option to override the TCP data offset field, while TCP SYN-EOS complements TCP EDO by extending the option space in SYN packets using an additional out-of-band packet during connection establishment. Inner Space uses a different strategy to extend the option space in every segment, where options are tunneled within the segment payload and a dual handshake procedure is used for assuring backwards com-



(a) Stripping of unrecognised TCP options from TCP segments: the segment is forwarded but the unknown option is removed.

(b) Blocking of TCP segments due to unrecognised TCP options: the segment is dropped.

(c) Rewriting of TCP sequence and acknowledgement numbers. The TCP checksum has to be rewritten as well.

(d) Coalescing of small TCP segments into a single, larger segment: parts of the header are rewritten, and options in the coalesced segment may change.

Fig. 1. Examples of middlebox interactions with TCP.

patibility with legacy servers. These approaches are currently under development and further work is needed to evaluate their deployability.

Experience in the design of MPTCP inspired another possible dimension to the design space: TCP "camouflaging" [30]. This suggests a new transport protocol could operate alongside TCP when the new protocol is disguised to look like TCP on the wire as in Polyversal TCP (PVTCP) [30]. Built upon the MPTCP subflow mechanism, PVTCP allows applications to explicitly customize the transport semantics of each subflow according to their requirements and assures a fallback to plain TCP or MPTCP. It remains to be seen whether the complexity of Polyversal TCP, or similar approaches, will offer a feasible path to deployment.

Although recent advances indicate that TCP continues to be extensible, more detailed and large-scale studies are needed to provide a deeper insight into the prevalence and range of middlebox behaviors. The IAB Workshop on Stack Evolution in a Middlebox Internet (SEMI) [31] identified this need and resulted in the proposed "Measurement and Analysis for Protocols[1]" (MAP) IRTF research group that aims to be a forum for exchange and discussion of insights from such measurements [32].

[1]Formerly known as "How Ossified is the Protocol Stack?" (HOPS).

B. *Using widely deployed transports as substrates*

The broad deployment and support of TCP and UDP in the Internet have led to the proliferation of a new design/deployment model where transport layer innovation occurs on top of these protocols. Typically, such transports are integrated into applications and aim to fulfill specific application requirements.

The choice between TCP and UDP involves a trade-off between design and implementation effort, flexibility and performance. On the one hand, UDP provides a "least-common-denominator" substrate with greater flexibility to control how data are sent over the network. However, building new transports on top of UDP often involves reinventing the wheel for services already offered by TCP (e.g., feature negotiation, congestion control, and reliability) and requires maintaining connection state in middleboxes by sending keep-alive messages that waste capacity and energy [33]. Guidelines for using UDP robustly are given in [34]. On the other hand, TCP is a feature-rich transport protocol that has undergone remarkable evolution over the past decades and can hence offer

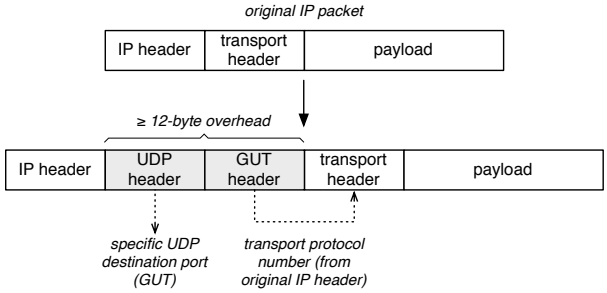
significant performance advantages over UDP [35]. However, TCP does not preserve message boundaries and is unable to support the use of only a subset of the services it provides; providing services that may not be needed can result in significant performance penalties. For example, the TCP in-order delivery service can incur increased end-to-end delays in lossy networks due to head-of-line blocking at the receiver.

The Minion suite of protocols has been proposed to address the above shortcomings of using TCP as a substrate protocol [35]. This was designed to offer an unordered, message-oriented delivery alternative to UDP. Minion is wire-compatible with TCP (or TLS/TCP when secure services are needed), at the expense of using slightly increased capacity. Improvements to the Minion suite, such as support for multiplexing and traffic prioritization have been introduced [36]. Despite its attractive features, the Minion suite has not seen wide-scale use. One reason could be that one of its greatest benefits, the ability to relax the in-order semantics of TCP, requires changes to the TCP stack, and hence is OS-dependent.

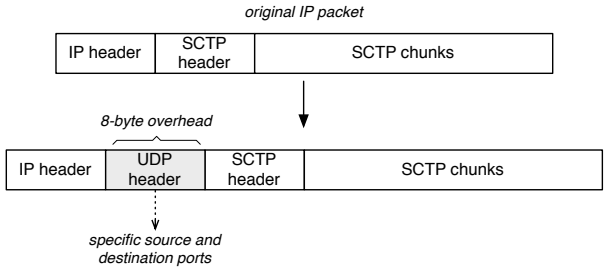
UDP can be used as a lightweight substrate and has been used since the 1990s to carry multimedia traffic with the Real-Time Transport Protocol (RTP) [37]. Characteristic examples using UDP are:

- Google's Quick UDP Internet Connections (QUIC) protocol [38], [39], a UDP-based low-latency alternative to TCP/TLS for SPDY [40] and HTTP/2 [41].
- Adobe's Real Time Media Flow Protocol (RTMFP) [42] enables efficient peer-to-peer multimedia streaming.
- The widely used DTLS [43] protocol provides stream- and datagram-oriented security services over UDP.
- The uTorrent Transport Protocol (uTP) [44], a UDP-based protocol for BitTorrent designed to offer a less-than-best-effort service for peer-to-peer file sharing applications.
- The UDP-based Data Transfer (UDT) protocol [45], [46] designed for efficient transferring of large data volumes over high-speed networks.
- The Structured Stream Transport (SST) protocol [47], a generic approach designed to offer services similar to SCTP [5], such as multistreaming and stream prioritization, over UDP.

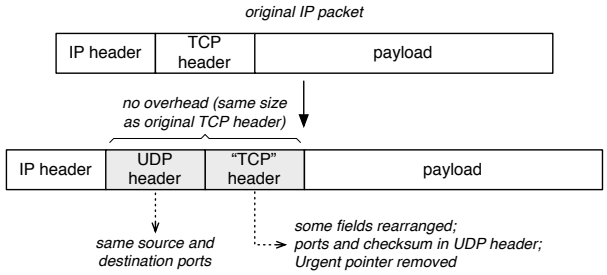
In addition to the above approaches, methods have been standardised by the IETF that encapsulate native protocols such as SCTP [5] and DCCP [4] within UDP [48], [49]. Methods have been proposed for encapsulating TCP over UDP enabling it to traverse network paths where only UDP is supported [50]. There is a large variety of (incompatible) tunnel and encapsulation frameworks that allow protocols to operate over UDP. Generic solutions have been sugested for encapsulating native IP protocols within UDP: Generic UDP Tunneling (GUT) [51] is a simple UDP encapsulation that aims to transparently tunnel native transports over a single well-known UDP port. GUT modifies native IP packets by including an appropriate UDP/GUT header, reconstructing the packets at the receiver. Generic UDP Encapsulation (GUE) [52] is similar to GUT, but focuses on leveraging the capabilities of network devices for handling UDP flows (e.g., load balancing). GUE uses a UDP source port as an inner flow identifier and permits encapsulation of layer-2 and layer-3 protocols. Although generic approaches could allow for more consistent deployment, protocol-specific designs may still be needed to ensure the functionality of the encapsulated protocol is not affected.



(a) Generic UDP Tunneling (GUT) [51], allowing encapsulation of an arbitrary native transport protocol.



(b) SCTP-over-UDP encapsulation [48].



(c) TCP-over-UDP encapsulation [50].

Fig. 2. Examples of transport encapsulation methods based on UDP.

Fig. 2 illustrates some of the UDP-based encapsulation methods just described.

Besides enabling middlebox traversal, UDP encapsulation offers an additional benefit: it allows user-space implementations of native protocols to be a part of applications without requiring special privileges to access the IP layer. The SCTP user-space implementation in [53] also offers this option. However, UDP encapsulation increases protocol overhead due to the additional UDP headers and also affects interoperability as the encapsulated protocol cannot in principle interoperate with the native one. Other potential drawbacks include: additional processing overhead, possibly redundant functionality (e.g., checksums) and increased design complexity due to an additional point of multiplexing.

McQuistin and Perkins [54] approach the problem from a slightly different perspective and suggest, at a conceptual level, to reinterpret the semantics of TCP and UDP to support novel services. They propose reinterpretation of UDP headers as transport identification headers where port numbers become dynamic identifiers of the transport protocol carried in the payload, as well as the relaxation of TCP. The Minion suite discussed above could contribute to this development.

Finally, there is ongoing work [55] to identify the suitability of the DTLS protocol [43] as a sub-transport for providing standardized security to higher-layer transports, along with services similar to that of SPUD (§ III-B), for instance signals to a middlebox to indicate the begining or end of a flow. Huitema et al. [55] identified requirements that need to be fulfilled, including zero-latency setup and low overhead.

III. Signaling for facilitating middlebox traversal

Even when TCP or UDP is used, middleboxes can cause significant connectivity problems to applications. For example, a NAPT can break the end-to-end connectivity for peer-to-peer applications (see Fig. 3) and applications that use control protocols such as SIP [15], RTSP [56], or FTP [16] preventing them from communicating reachability information.

A variety of support protocols and mechanisms have been proposed to improve connectivity across paths with middleboxes. These focus on ways to control middlebox behavior, methods to allow cooperation between endpoints and middleboxes, and methods to facilitate end-to-end connectivity. Such methods may be categorised as either *implicit* or *explicit*.

Implicit control solutions treat middleboxes as black boxes and trigger specific middlebox behaviors using data traffic sent to a well-known third party server. An explicit control solution allows an endpoint to explicitly interact with a middlebox to control or influence its behavior, e.g., to create NAPT mappings or to configure the lifetime for flow state.

A. Implicit middlebox control

Interactive Connection Establishment (ICE) [57] seeks to increase the probability of successful connection by trying a set of implicit control techniques and selecting the one that works best. ICE was developed for middlebox traversal of UDP-based multimedia streams established by an offer/answer protocol (e.g., SIP) and is the middlebox traversal solution used in WebRTC. This utilises the Session Traversal Utilities for NAT (STUN) [58] and the Traversal Using Relays around NAT (TURN) protocol [59]. Ford et at. [14] describe a method for UDP hole punching. A TCP-based extension of ICE [60] adds TCP hole punching and considers UDP encapsulation as an alternative traversal solution for TCP. Techniques for TCP hole punching are presented in [14] and [61]. A TURN relay for TCP is specified in [62].

No single solution is perfect in terms of applicability and performance. For instance, UDP hole punching cannot work with symmetric NATs, TURN uses a relay server and hence can be a performance bottleneck, and TCP hole punching techniques have lower success probability because they depend on specific middlebox behaviors that are not always supported [63].

B. Explicit middlebox control and cooperation

There is a range of approaches that can allow the transport to exchange control information with a middlebox, such as the Universal Plug and Play Internet Gateway Device (UPnP IGD) protocol, the Port Control Protocol (PCP) [64] and its precursor NAT Port Mapping Protocol (NAT-PMP) [65], the Middlebox Communication (MIDCOM) framework [66], and the NAT/Firewall NSIS Signaling Layer Protocol (NSLP) of the NSIS protocol suite [67]. Each solution has its own merits depending on network topology and security requirements, and hence there is no single solution that an application can rely upon to be universally available. For this reason, applications usually resort to use implicit control schemes that do not require additional support by middleboxes. However, no solution can always guarantee traversal.

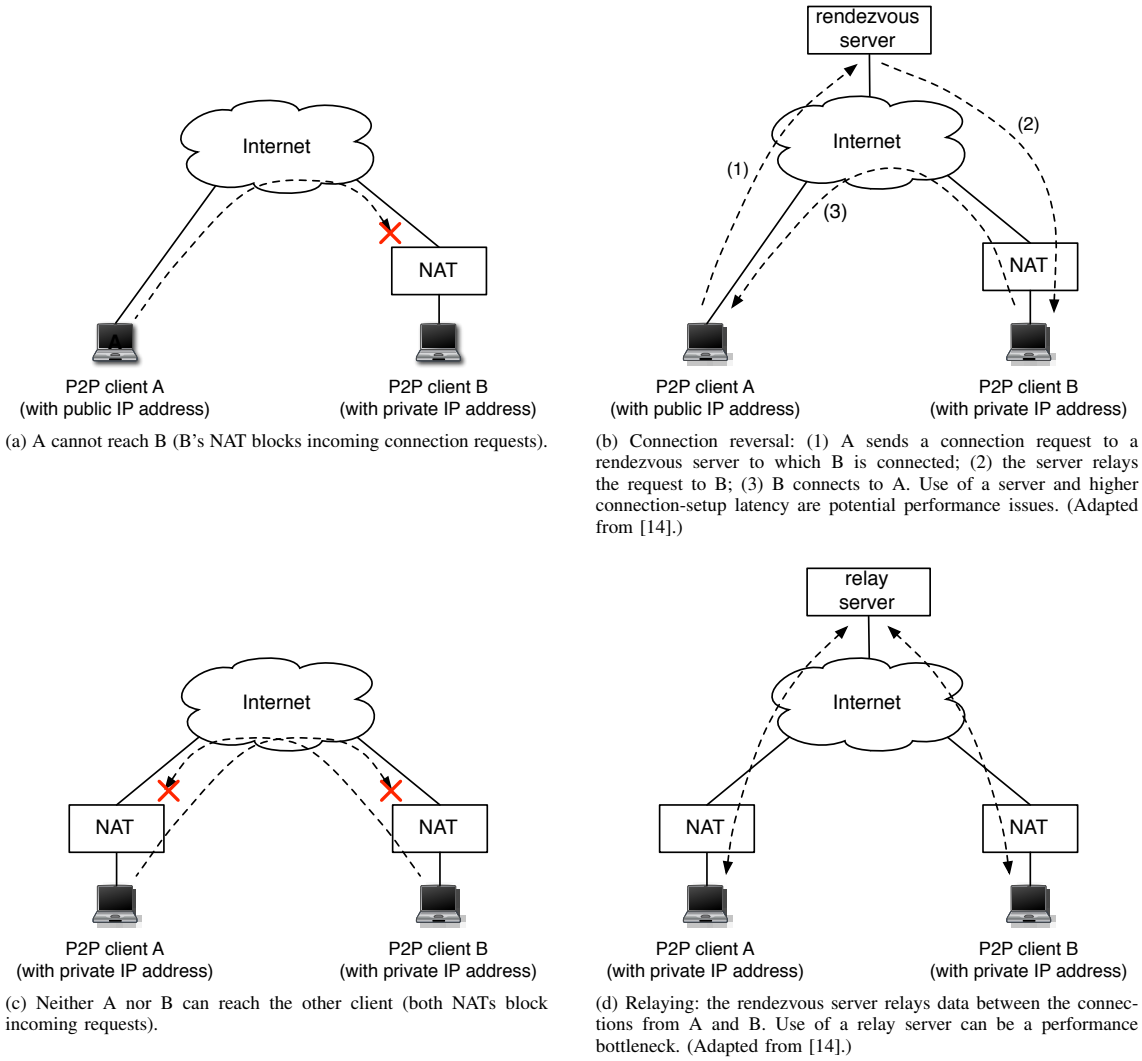
A new form of UDP encapsulation layer could allow explicit cooperation with middleboxes [31], [68], [69]. This approach may help re-instantiate the layer boundary between a hop-by-hop network layer and an end-to-end transport layer [69], by allowing endpoints to control the information exposed to the path (encrypting everything above the UDP header), while still allowing appropriate transport semantics to be explicitly exposed to the path to assist the middlebox in establising and maintaining state. An approach in which the transport protocol encrypts its protocol information can allow the transport to evolve without needing to consider the interference of middleboxes [40].

The Substrate Protocol for User Datagrams (SPUD) prototype [70] is ongoing work that seeks to realize and facilitate middlebox traversal for new transports. SPUD groups the packets of a transport connection into a “tube that can allow network devices on the path to understand basic session semantics (e.g., beginning and end of a flow). SPUD may also enable communication of path information to the sender, and permits explicit endpoint to/from middlebox communication.

SPUD requires support at both endpoints, and only gains benefits from middleboxes when they also implement support for the protocol. Hence it can not be considered a “quick-fix” solution. It has therefore been designed so that the SPUD protocol is useful as a simple encapsulation until support is enabled in middleboxes, enabling incremental deployment [70].

IV. Enhancing the API between the applications and the transport layer

The first part of this section gives an overview of the standard socket API and how it has been extended to support SCTP. The remaining parts consider ways to address some of the major inherent limitations of this API, i.e., those limitations that are believed to contribute to the ossification of the transport layer. We examine some proposed extensions to the standard socket API, and ways to address its current tight coupling between the offered transport service and the underlying transport protocol offering this service.



(a) A cannot reach B (B's NAT blocks incoming connection requests).

(b) Connection reversal: (1) A sends a connection request to a rendezvous server to which B is connected; (2) the server relays the request to B; (3) B connects to A. Use of a server and higher connection-setup latency are potential performance issues. (Adapted from [14].)

(c) Neither A nor B can reach the other client (both NATs block incoming requests).

(d) Relaying: the rendezvous server relays data between the connections from A and B. Use of a relay server can be a performance bottleneck. (Adapted from [14].)

Fig. 3. Examples of connectivity issues due to NATs, and of implicit control techniques to address them.

A. *The socket API*

The socket API is one of the most pervading and longest-lasting interfaces in distributed computing. Conceptually, a socket is an abstraction of a communication endpoint through which an application may send and receive data in much the same way as an open file permits an application to read and write data to a stable storage device such as a hard disk. Applications use socket descriptors to access sockets in the same way that they use file descriptors to access files.

The API was designed from the start to be independent from the underlying protocol stack, as seen in the way that a socket is created: `int socket( int` *domain*, `int` *type*, `int` *protocol* `)`. The *domain* parameter determines the communication domain or protocol family of a socket. Examples of protocol families include: `AF_INET` for the IPv4 Internet domain; `AF_INET6` for the IPv6 Internet domain; and, `AF_UNIX` for the local or Unix domain. The *type* parameter determines the type of a socket, or, more specifically, the semantics for the transport service—e.g., whether the transport service should be stream-oriented, reliable, and connection-oriented (`SOCK_STREAM`), or message-oriented, unreliable, and connectionless (`SOCK_DGRAM`). Finally, the `protocol` parameter lets an application specify which transport protocol to use to provide the transport service specified by the `type` parameter.

Although the socket API comprises a fairly large number of functions, there are less than a dozen core ones. For example, a simple connection-oriented client-server application does not need more than the eight functions listed in Table IV-A. A server application generally executes the first four functions in the order given in the table, while a client application attempts to connect to the server after having created a socket; the `send`

TABLE I
BASIC TCP SOCKET API FUNCTIONS.

Function	Description
`socket`	Creates a new communication endpoint
`bind`	Binds communication endpoint to local IP address and port number
`listen`	Makes a socket listen to incoming connections
`accept`	Blocks a socket until a connection request arrives
`connect`	Makes a connection request
`send`	Sends a message over a connection
`recv`	Receives a message over a connection
`close`	Releases the connection

and `recv` function may be called by both the client and the server. A connection that is no longer needed is closed by the client or server.

The API lets an application control the behavior of a socket through options. The set of options has expanded over time, as usage has evolved. There are essentially three ways to manipulate socket options:

1) The functions `setsockopt` and `getsockopt` provide access to the majority of available socket options.
2) The function `fcntl` is primarily used with non-blocking and asynchronous I/O.
3) The function `ioctl` has traditionally been the way to access implementation-dependent socket attributes.

The socket options accessed via `setsockopt` and `getsockopt` are divided into two levels: The first level are generic, i.e., non-protocol specific, options. For example, the sizes of the socket send (`SO_SNDBUF`) and receive (`SO_RCVBUF`) buffers are generic socket options. The second level comprises protocol-specific options such as those that control the behavior of IP, UDP, and TCP. An example of a well-known, second-level socket option is `TCP_NODELAY`, which determines whether the Nagle algorithm [71] should be enabled.

The deployment of the SCTP transport protocol [5] demanded changes to the socket API. In addition to the services offered by TCP, SCTP supports both multi-homing (i.e., connections comprising several network paths) and multi-streaming (i.e., several independent logical flows over a single connection). These additions required extended versions of several existing socket API functions and a new notification mechanism to enable signaling of transport-level events to an application, such as connection status changes. A good example of how SCTP extended the socket API, is the extended version of `bind`: The normal `bind` socket call only enables for a communication endpoint to bind to a single IP address. SCTP introduces the `sctp_bindx` socket call which lets an application bind to several or all IP addresses on a host.

Since SCTP has its roots in the transport of critical telephony signaling traffic, it had to be able to communicate transport-level events to an application, such as connection availability and remote operational errors. To ensure the SCTP event notification is well aligned with the rest of the socket API, events are enabled by a socket option: `SCTP_EVENTS`. Once enabled, the SCTP stack sends events as a normal messages to the application. An application may distinguish between event notifications and normal messages, by a flag in event notification messages set to `MSG_NOTIFICATION`.

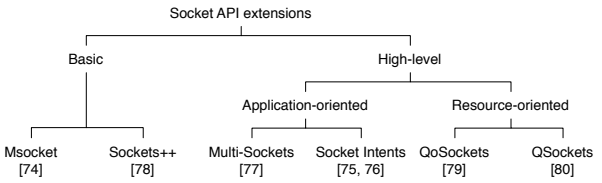


Fig. 4. Extensions to the sockets API.

SCTP also extended the semantics of the socket API by supporting two types of sockets: one-to-one and one-to-many. A one-to-one socket resembles usage by TCP. A one-to-many socket makes it possible for an application to manage several SCTP connections via a single socket. This has advantages for server applications that may use a one-to-many socket to avoid the need to administer each client request through a separate socket.

The example of SCTP has shown that incorporating a transport with different techniques has required updates to the current sockets API. It would seem reasonable to expect similar changes may also be needed to support any additional new transport (or technique). A significant drawback is that this also requires any application that wishes to benefit from using a new technique to be updated to use the new API.

B. More expressive APIs / Extensions to the socket API

Extensions to the sockets API have also been proposed that change the way an application interacts with the transport layer. These may be categorised according to the abstraction level at which the underlying transport services are exposed (Fig. 4). Some proposed extensions, which we call *basic extensions*, only aim to remove perceived limitations and drawbacks of the standard sockets API. For example Msocket [72] makes it possible to have several implementations for each domain, type, and protocol assignment. These proposals provide the same exposure of the transport layer as the standard sockets API.

In contrast, *high-level extensions* hide the implementation of offered transport services from applications. These focus on ways to allow an application to express its quality-of-service (QoS) requirements to the transport layer. Examples include Socket Intents [73], [74] and Multi-Sockets [75] High-level extensions can be further divided into *application-oriented* and *resource-oriented* extensions. Application-oriented extensions let an application express its QoS requirements in terms of application-dependent performance metrics or the characteristics of the traffic it will generate. In contrast, resource-oriented extensions focus on system-wide, network-oriented performance metrics such as packet loss, re-ordering, bitrate, or end-to-end delay. We now present each category of socket API extension and provide examples within these categories.

1) Basic Extensions: If several protocol stacks are available, the standard sockets API does not enable an application

to explicitly select the one to use. The Msocket [72] extension removes this limitation by adding a stack parameter to the `socket` call. In Unix systems, the stack parameter is a device file. This does not have to be the case in other systems, and could refer to a kernel module. Backward compatibility with the standard sockets API is assured by the definition of so-called default stacks: each protocol family is assigned a default stack.

Sockets++ [76] is an object-oriented basic extension that addresses a range of shortcomings with the sockets API. It supports multipoint connections to enable several applications to participate in the same connection. It also supports direct forwarding allowing multimedia applications to request data is directly forwarded from one stream to another. It seeks to minimize parameters in socket calls, e.g., combining domain and protocol parameters in the `socket` call, and to simplify socket API options. Importantly, this extension also enables applications to express their quality-of-service requirements.

2) High-level Extensions: Intentional extensions originated in work for mobile devices with more than one available network interface. It allows applications to inform the API about the traffic they intend to send (e.g., whether it will be latency-sensitive video conferencing traffic or throughput-dependent file transfers). This information enables the transport layer to select the most appropriate network interface, dividing the responsibility for communication between the application and the transport stack.

Intentional networking was first realized in Multi-Sockets [75], allowing an application to use labels to communicate its intents. Labels provided qualitative rather than quantitative information, e.g., to inform the API whether a message unit belongs to an interactive or non-interactive traffic flow, or whether it belongs to a flow that consumes little or much capacity. Conceptually, a multi-socket multiplexes several different labels across a single virtual connection, however, in practice, the proposal instantiated and used actual TCP connections over one or several physical network interfaces.

Socket Intents [73], [74] is a successor to Multi-Sockets, seeking to support multi-homed applications. Socket Intents replaced the labels used in Multi-Sockets by augmenting the sockets API with additional socket options. An implementation of Socket Intents comprises three components: a wrapper library over the standard sockets API, a policy module, and the multi-access manager – a daemon that hosts the policy module. Since creating a single policy that maps different traffic flows to different network interfaces is, in general, not feasible, the Socket Intents API was built as a generic framework with a replaceable policy module.

Resource-oriented sockets API extensions offer communication between themselves and the application. For example, QoSockets [77] enables an application to negotiate its quality-of-service requirements with the transport layer, and for the transport layer to signal violations of these requirements back to the application. The requirements include loss rate, ordered or unordered delivery, end-to-end delay, and jitter. Application- and network-management functions were integrated by adding an interface to a Management Information Base (MIB), and a status interface for connections. These MIBs show how communication resources are allocated and utilized, and enable an application to detect and adapt to quality-of-service violations.

QSockets [78] is another resource-oriented sockets API extension. Similar to QoSockets, QSockets also offers bidirectional communication to the application, enabling applications to obtain detailed quality-of-service feedback. It uses an extended socket API that adds a structure that contains the QoS preferences. The QoS parameters may also be set on a per-packet level by passing a structure to `sendto` calls, allowing per-packet deadlines and the setting of other flags. The API communicates with an in-kernel management module to control an in-kernel scheduler. This exposes functionality to the management module for managing scheduled packet streams. A pluggable scheduling layer allows various QoS scheduling algorithms.

Although no single approach has been adopted by the community, this body of research has shown there are benefits to enriching the transport API to express more than the traditional socket API.

C. Transparent transport protocol selection

The current design of the socket API has a design that focusses on specific support for each transport protocol, each with different needs. Fairhurst et al. [79] provide a recent survey of the services provided by the range of IETF-standardised transports. The present design of the API makes it difficult to introduce any new protocol [80].

These limitations could be overcome by re-designing the way that the API is used, e.g., by: using a protocol-independent mechanism to set parameters; by describing application requirements at a higher level of abstraction (similar to intentional methods); and providing a service-oriented interface between applications and the transport (where applications describe the required services than the protocols to use). The latter would allow transport protocol selection to be dynamically handled at run-time, easing the introduction of new and alternate protocols.

A prototype implementation [81] used a service-oriented API to indicate a combination of inherent properties (reliability, security, etc.) and qualitative properties (expressing tendencies and preferences). The set of inherent transport properties was derived by examining the key transport protocols (TCP/IP, UDP/IP, RDP [82], RDP/IP, XTP [83], XTP/IP, SCTP/IP). Five qualitative properties were also suggested (transmission delay, flow setup delay, network resource usage, host resource usage, and quality). A broker then matched the inherent properties with application requirements to first identify the transport to use, and then used the qualitative properties to optimize the matching.

Welzl [84] identified deployment problems resulting from the complexity of the different protocol APIs and proposed an "Adaptation Layer" that hides protocol details and exposes a common service-oriented interface. This allowed applications to specify their requirements and characteristics. An adaptation layer then sought to provide the best transport service based on available transport protocols and the current network environment. This adaptation layer could also tune protocol parameters and provide additional functions, such as buffering.

Welzl et al. [85] later derived a methodology for constructing a service-oriented transport API. This started with a list of all services provided by SCTP, DCCP and UDP-Lite, and iteratively pruned redundant services or services considered unnecessary, resulting in a list of 23 distinct transport services composed from six different features. This led to a strawman proposal for a protocol-independent version of the socket API, where the selected transport services could be accessed through their service number.

A similar proposal [86] expressed the desired service through a set of requirements, such as packet boundary preservation, authentication or maximum delays. Their adapted socket API used a name similar to a URI [87] to identify the communication peer, removing dependence on IP addresses.

There is a need to standardize any new service-oriented API [84], to ensure that it can have significant impact and becomes used by applications in future. This requires the community of application developers, and transport developers to reach consensus on the set of desirable interface features. Recent IETF work within the Transport Services working group (TAPS) [79], [88] provides a unique opportunity to develop this sort of consensus.

D. Enhancing the API to allow evolution below the transport layer

There is a long history of proposals to support communication between end systems and the network. Proposed solutions can be divided into two broad classes according to their scope: 1) solutions that facilitate middlebox traversal for applications (discussed in § III-B), 2) solutions that focus on communicating information between the network and the endpoints to improve application experience (signalling of QoS requirements, QoS reservations, and indications of capacity changes, of data corruption, of congestion, etc.). However, there are also challenges to finding suitable, scalable, secure and robust signaling mechanisms that can be deployed across the Internet (e.g., [70], [89]–[91]). Finding appropriate methods largely remain an area of research. One issue with deploying mechanisms is that many methods require applications to indicate their needs and how they expect the network to respond. The current socket API does not provide such information, nor have applications typically been designed to utilise such methods, and hence at present these are unlikely to be widely deployed.

A higher-level transport API that places the responsibility for negotiating and using network signaling below the transport API may encourage future applications to utilise new methods as the stack and network introduce them. This technique was adopted by some of the API proposals discussed [78] and could be enabled by the approaches being proposed in [92].

V. Discovery and exploitation of end-to-end capabilities

Some application-layer proposals provide limited support for negotiation of e.g. transport security for unicast, connection-oriented application sessions [93], or transport protocol, port and IP address for multimedia sessions [94]. A more generic approach is for end-points to use a negotiation protocol to exchange protocol-stack information, and to agree on a transport stack (i.e., transport and security protocols to be used, and their options) [95]. Their proposal focused on connection-oriented transports. Minimising latency, by reducing the number of RTTs needed for negotiation, requires changes to the implementations of the transport protocols being negotiated.

In the absence of an explicit end-to-end signaling or a negotiation protocol, the only way for an end-host to discover and (implicitly) agree on the choice of protocol(s) is to *simultaneously try* a set of candidate methods, and choose one a method that works. This "test-and-select" approach, known as *happy eyeballs*, has been proposed both for choosing between transports [96] and between versions of the IP protocol [97]. To the best of our knowledge, only the latter has been implemented in real systems (e.g., [98]), coupled with address-selection algorithms such as [99]. Fig. 5 depicts a possible variant of happy eyeballs for a client to discover SCTP support, both at a server and along the path to the server. A drawback of this kind of technique is it increases both the number of packets sent, the server-side load and (potentially), the amount of state created in middleboxes; hence, it does not scale well with the number of candidates to try. For instance, testing for native SCTP, SCTP-over-UDP and TCP, combined with both IPv4 and IPv6, would in principle require testing six protocol combinations (compared to two in the example). Moreover, happy eyeballs requires careful design of timers, needed to decide when to discard a trial for a given protocol choice. Also, the sequence in which trials are attempted can be important, to avoid systematic bias towards particular protocol choices.

It is important to consider the overhead in the design of a happy eyeballs algorithm, especially the overhead in terms of *added latency* for initiating a session. In general, any transport signaling or feature discovery / negotiation mechanism may incur either additional round-trip times (e.g., if connection attempts are serialized) or waiting delay (e.g., due to waiting for replies to two parallel connection requests). It is therefore essential to cache results to speed-up subsequent trials. For instance, prior knowledge that protocol choice X works with destination D can be used to tune the testing process, by e.g., slightly delaying trials with protocols other than X [96]. Cached information can also inform the happy eyeballs mechanism to give preference to certain choices, e.g., ones expected to offer lower path latency [98].

VI. Enabling user-space protocol stacks

It is possible to run a transport as a user-space library, letting applications use the transport in user-space, rather than the one provided by an OS kernel. This can allow more portability and deployability across multiple OS and hardware platforms. This approach can enable easy introduction and ease testing of new features and protocols (e.g., a simple user-space TCP library (UTCP) used MultiStack [100]).

In many systems, privileges are needed to introduce a new protocol and may not always be granted to the entity trying to

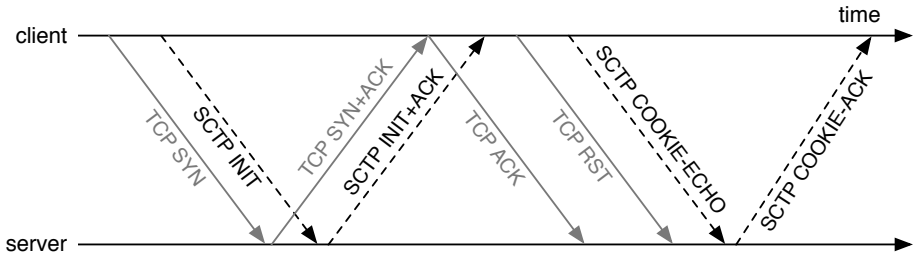


Fig. 5. "Happy eyeballs" technique for the discovery of SCTP support, with SCTP being the preferred choice. The first handshake of the SCTP association succeeds shortly after the TCP connection does, so the latter is aborted.

induce a new transport protocol. User-space transports do not need *root* privileges on the host machine. However, the use of user-space transports presents a range of challenges.

One challenge is that network I/O operations that originate in user-space can incur higher latency compared to network I/O operations handled in the kernel. MutiStack [100] offers a solution that enhances commodity operating systems with support for dedicated user-level network stacks. It can concurrently host a large number of independent stacks, and can fall back to the kernel stack if necessary. MultiStack provides high speed packet I/O at rates up to 10 Gb/s [100], by extending two components: the netmap framework [101] and the VALE software switch [102].

Other libraries can help achieve fast packet I/O in user-space, such as the Data Plane Development Kit (DPDK) [103] and PACKET_MMAP [104]. DPDK [103] is a set of libraries and drivers for fast packet processing mostly in Linux user-space. However, DPDK is not a networking stack and does not provide functions such as Layer-3 forwarding, IPsec, firewalling, etc. PACKET_MMAP [104] seeks to provide efficient raw packet transmission and reception in the Linux kernel using a zero-copy mechanism with a configurable circular buffer, mapped in user-space to minimize the number of system calls.

In addition to user-space TCP [105], there is also a user-space SCTP implementation for all major OS platforms [53] using the FreeBSD kernel sources for SCTP. Since it is not always possible to send data directly over native SCTP (e.g., because not all middleboxes can process SCTP packets), the SCTP user-space implementation in [53] additionally supports the option of encapsulating SCTP packets in UDP.

User-space SCTP [53] is implemented using *raw sockets* in user-space. A raw socket receives or sends raw datagrams (at OSI Layer 3). Whereas *packet sockets* receive or send raw packets at the device driver level (OSI Layer 2). This allows a user to implement protocol modules in user-space on top of the physical layer (e.g. PACKET_MMAP [104]).

Another technique that enables transport protocols to run in user-space is to run the entire kernel (instead of only the transport) as a user-space process, as in User-Mode Linux (UML) [106]. This permits experimenting with new transport protocols implemented in different Linux kernels without interfering with the host Linux setup. UML provides a virtual machine as a single file, potentially with more (virtual) hardware/software resources than the actual host, and can potentially provide limited access to host hardware. A similar approach is followed by LibOS [107], which runs the kernel as a library that can be called by an application. LibOS has been used by NUSE [108] to provide a Linux network stack for user-space applications.

VII. Summary of point solutions

Table II recaps the taxonomy of issues and point solutions to transport-layer ossification described in more detail in the previous sections.

VIII. A way forward: a transport-layer framework

The previous sections have shown that de-ossifying the Internet transport layer to re-enable its evolution is a multi-dimensional problem. This requires the enhancement of multiple components of the end-to-end communication. Several point solutions have been proposed or are underway, each aiming to address a specific aspect of the overall problem. However, there has been little effective integration of techniques that can produce an evolvable transport layer.

For instance, incorporating a new application-level transport within the application's code (e.g., QUIC) to enable new transport services would inevitably require a negotiation service, e.g., a negotiation protocol like the one described in [95] to discover if the transport is supported by the remote peer (e.g., a web server), accompanied with a fall-back strategy for the case where the new transport is not supported[2]. Implementing more advanced transport and network functions, such as dynamic selection and configuration of a transport based on current network state and QoS negotiation, would additionally require the involvement of more components, such as a policy system, measurement modules and network signaling mechanisms, that need to interact with each other.

While various solutions could be partly implemented according to *certain* application needs, this would inevitably result in an application-specific and less flexible implementation, that is neither sufficiently general to support other types of applications nor incrementally upgradable to support new transport and network functions as they become available. This

[2]At the time of writing, the Chrome browser (version 46.0.2490.86) does this by implementing Happy Eyeballs (see § V) between QUIC/UDP and TCP.

TABLE II
SUMMARY OF MAIN ISSUES AND POINT SOLUTIONS TO INTERNET TRANSPORT-LAYER OSSIFICATION.

Type of problem	Family of solutions	Solution approach	Examples of proposals
Middlebox-related hindrances	Design of middlebox-proof transports (§ II)	Extending TCP while guarding against middlebox interference (§ II-A)	MPTCP [12], [20], [21] Tcpcrypt [22], [23] Gentle Aggression [24] Extending TCP's option space [26]–[29] Polyversal TCP [30]
		Using widely deployed transports as substrates (§ II-B)	Minion [35], [36] Encapsulation of specific transport protocols over UDP [48]–[50] Generic encapsulation frameworks [51], [52] Use of DTLS as a substrate [55]
	Signaling for facilitating middlebox traversal (§ III)	Implicit middlebox control (§ III-A)	STUN [58] TURN [59], [62] UDP hole punching [14] NATBLASTER [61] ICE [57], [60]
		Explicit middlebox control (§ III-B)	PCP [64] MIDCOM [66] NSIS Signaling Layer Protocol [67] SPUD [70]
API ossification	Enhancing the API between the applications and the transport layer (§ IV)	More expressive APIs / Extensions to the socket API (§ IV-B)	Msocket [72] Sockets++ [76] Multi-Sockets [75] Socket Intents [73], [74] QoSockets [77] Qsockets [78]
		Transparent transport protocol selection (§ IV-C)	Reuther et al. [81] Welzl [84] Welzl et al. [85] Siddiqui and Mueller [86] TAPS [79], [88]
Lack of local knowledge about path- and remote end-host support	Discovery and exploitation of end-to-end capabilities (§ V)	Explicit negotiation protocols	Rose [93] Rosenberg and Schulzrinne [94] Ford and Iyengar [95]
		Implicit discovery / agreement	Happy eyeballs [96], [97]
End-host deployment issues	Enabling user-space protocol stacks (§ VI)	Transport protocols implemented in user-level libraries	User-space TCP [105] User-space SCTP [53] MultiStack [100]
		Enabling fast packet I/O in user-space	DPDK [103] PACKET_MMAP [104]
		Running an OS kernel in user-space	User-Mode Linux [106] LibOS [107]

would need considerable effort from application developers to re-implement common functions or services that might not be interoperable or efficient. Examples include QUIC in Chrome, RTMFP [42] in Adobe Flash Player, and proprietary protocols in Skype [109] and the WebRTC framework [110].

We argue that a truly evolvable Internet transport architecture requires a necessary step to design and develop a comprehensive and evolutive *transport layer framework* that can facilitate integration and cooperation of transport layer solutions in an application-independent and flexible way.

This would relieve application developers from the burden of changing the application code to introduce new transport or network services and functions, breaking the vicious circle that hampers evolution.

The remainder of this section motivates the requirements for such a framework.

A. Requirements for an evolutive transport layer framework

Based on the discussion in previous sections, we identify a series of requirements that an evolutive transport layer framework should fulfill. We summarize them in five general

categories: 1) API flexibility, 2) Deployability, 3) Extensibility, 4) Guided parameter value selection, and 5) Scalability. We elaborate below on these requirements.

1) API flexibility: As discussed in Section IV, the ossification of the current transport API is a key obstacle that needs to be overcome. Applications using the framework should only interact with it via the API provided by the framework. This API should be able to decouple applications from a priori decisions on underlying protocols and functions. It should also allow to use the framework in the future by providing a simple way for porting existing applications to it. To this end, the API must be flexible, in the sense of the following requirements:

a) Backward compatibility: The API provided by the framework needs to provide backward compatibility to enable evolution from previous versions of the framework without affecting the applications that use the framework.

b) Support of low level configuration: The classical socket API requires detailed usage of the transport protocol stack, where the network and transport protocol need to be specified, and protocol-specific parameters chosen (when values different other than the defaults are needed). The framework should continue to permit this detailed level of configuration.

c) Support of high level configuration: The framework should allow configuration at a high level of abstraction. Mechanisms should describe the needs of an application in a more generic way than required by the classic socket API. Possible needs include message-orientation, preservation of message order, reliability, low latency, mobility support, relative priorities and security features.

An application may assume that it receives the requested service, but should not implicitly receive additional services. This allows the framework to make any further decisions necessary to establish optimal communication with the peer endpoint. As the framework evolves, different choices might lead to a better service without the need to change the application. Finally, multiple levels of abstraction need to be supported.

Recent advances in the development of more expressive, high-level, extensions to the socket API (e.g., Socket Intents and QSockets, § IV-B), and the important ongoing standardization effort of the IETF TAPS working group can provide a basis towards satisfying this requirement.

d) Comprehensibility: The framework must make low level information available to the application and to reveal the decision processes, so that applications know the concrete choices that were made to fulfill the requested abstract requirements. QoS feedback, as provided by QoSockets and QSockets (§ IV-B2), is an example of how such low level information could be of interest to an application.

2) Deployability: The framework should enable fast seamless deployment with as little disruption as possible. The deployability goals translate into the following requirements:

a) Application focus: The evolutionary character of the framework requires support of existing host operating systems. It must be installable, usable and upgradable without specific privileges. This enables the speed of the evolution of the framework to be independent of the speed that operating systems are updated.

b) Host operating system feature tolerance: The framework should not only make use of protocols and features available on the host operating system, but allow integration of additional protocols (e,g., SCTP or DCCP) and features (e.g., caching network or transport information).

To enable easy deployment of new transport protocols and/or transport protocol components, solutions that enable the deployment of user space transport stacks should be supported by the framework. Examples include support for application-level transports (such as uTP and QUIC, § II-B), UDP encapsulation schemes (such as SCTP/UDP encapsulation and GUT, § II-B), and user space implementations of native transports (such as SCTP and TCP, § VI).

c) Peer feature tolerance: It can not be assumed either that all endpoints use the new framework. Even when the framework is supported by all endpoints, it must not be assumed that they use the same version of the framework. This allows for incremental deployment, possibly at the cost of providing less benefit. Similar robustness is required for the protocols and mechanisms used to realize the transport service.

A method that allows implicit or explicit discovery of the set of protocols/mechanisms supported by a remote endpoint could allow the framework to leverage the best common set of available features. Examples of such solutions are the negotiation protocol described in [95] and the happy eyeballs mechanisms (§ V). Feature negotiation and fallback mechanisms can be incorporated within a protocol or a mechanism itself, such as the options mechanism for negotiating TCP extensions and the fallback scheme of MPTCP (§ II-A).

d) Network feature tolerance: The ability to use the framework must not depend on the network support for specific features (e.g., quality of service mechanisms or middlebox interaction), but may utilize these when they are found to be supported.

Support for middlebox-proof transports (§ II) and mechanisms for implicit middlebox control (§ III-A) can be of great value for making the framework independent of the features supported by middleboxes. Additionally, support for "looser" network signaling mechanisms (e.g., SPUD, § III-B) for interacting with network devices can enable a "best effort" use of available network features.

3) Extensibility: The framework must be able to support seamless, independent evolution of the different components:

a) Support of framework evolution: An evolutionary framework must permit addition of new protocols and features in the future.

b) Support of operating system evolution: The interface between the framework and the operating system may change over time to improve the service provided by the framework, including additional protocols and features. This allows moving implementations from the framework to the host operating systems and vice versa as they evolve.

c) Support of network evolution: Some middleboxes may allow an endpoint to signal its needs. Applications should not rely on signaling, but can benefit when this is available, possibly increasing the chance that a path can be used (e.g.,

by explicitly controlling middlebox traversal, § III-B), or even enabling features (such as QoS support) that can benefit the transport (e.g., through the signaling of advisory metadata, § IV-D). It should be assumed that the available methods for interacting with the network (and middleboxes) will evolve over time. The architecture of the framework must therefore allow applications using the framework to benefit from this evolution.

4) Guided parameter value selection: Current transport and network stacks require explicit parameter value selection. For example, an application may choose IPv4 or IPv6 and select DCCP, SCTP, TCP, UDP-Lite or UDP. Furthermore, parameter values can be specified by explicit socket or protocol level socket options. The framework should be able to combine network-wide and local information to select the appropriate parameter values that make the best of available features for satisfying application requirements. Such guided parameter value selection corresponds to the following requirements:

a) Derivation of parameter values: The framework must map high-level requirements provided by the application to the low level parameter values to be used. This parameter selection should be guided by the requirements provided by each application to result in selection of the interfaces to be used, the network protocol, the transport protocol, and the setting of parameter values at each layer. Examples include the policy-based interface selection system of Socket Intents (§ IV-B2) and the run-time service broker in [81] (§ IV-C). The IETF TAPS Working Group is seeking to provide guidance on choosing among available protocols and mechanisms.

b) Dependency on local tools: If possible, tools included in an operating system (for example, link status supervision tools) should provide useful information to the framework when making the decisions and parameter value selections.

c) Dependency on network and peer: Any decision to use a particular protocol must be based on the set of protocols supported by the local and remote endpoints. A prerequisite to using a protocol is that it can communicate over the path between the endpoints, including any middleboxes employed along the path. The framework should support mechanisms for discovering characteristics of the end-to-end path and/or the remote endpoint, such as happy eyeballs, end-to-end signaling and negotiation protocols (§ V).

d) Ability to use time-dependent path information: The final decision to use a candidate protocol can be based on historical information such as whether a protocol or feature was previously supported on the path, but needs to also consider that path characteristics can change over both long time-scales (e.g., due to upgrades or route changes) and short time-scales (due to load balancing over alternate paths, wireless links, etc.). Use of historical information will require components for caching path properties (e.g., caching happy eyeballs results, § V) and which will be able to efficiently store information with diverse lifetime requirements.

e) Agnostic to application protocol: Testing and discovery must be done by the framework and must not require any change to, or specific support by, application protocols.

5) Scalability: The framework must be scalable in a variety of ways:

a) Traffic volume: The framework must limit the impact on CPU load and scale to support a high volume of user traffic (e.g., to support high-speed interfaces). Hardware support should be leveraged whenever possible. At the same time, the framework must not by itself produce control traffic (signaling) that limits scalability.

b) Number of peers: The number of transport associations needed (for example TCP connections or SCTP associations) depend on the use case. The framework must efficiently support a high number of simultaneous transport associations.

c) Size of feature set: Finally, the framework needs to be able to support a variety of combinations of protocols, parameter settings and network interactions. The selection process must therefore be able to select from a large set of possibilities, while providing an acceptable communication setup time.

IX. Future research directions

To conclude, we identify ongoing and forthcoming research efforts that we expect will lead to further developments towards de-ossifying the transport layer.

Considering the approaches discussed so far, it seems that the ossification problem has two main root causes: 1) middleboxes that examine and/or manipulate the contents of packets beyond the IP header make it hard to deploy protocols that these middleboxes do not yet know; 2) the API that is exposed by the socket API ties applications (or the middleware or library that these applications are based upon) to a specific protocol choice. Both sub-problems have been addressed in various ways by research proposals. Unfortunately, some of these proposals are not new, yet it seems that these solutions have had little to no impact on the Internet: the transport layer still appears to consist of only TCP and UDP, often even further constrained to specific port numbers [8]. If anything, the situation seems to have worsened over the years.

There is however some reason for hope that we may be reaching a turning point. At the time of writing, several initiatives are focussing on making such a change possible; these initiatives point at the different open research directions in this space:

- The IETF TAPS Working Group seeks to specify how applications could specify their transport requirements, instead of being tied to a specific protocol, and how a transport system based on such requirements specifications could be constructed[3].
- The IP Stack Evolution Program within the Internet Architecture Board (IAB) provides architectural guidance, and a point of coordination for work at the architectural level to improve the present situation of ossification in the Internet protocol stack[4].
- Current activity around the SPUD protocol at the IETF[5] is striving for better cooperation between end-points and middleboxes in a context of increasing use of encryption.

[3]https://tools.ietf.org/wg/taps/charters
[4]https://www.iab.org/activities/programs/ip-stack-evolution-program/
[5]https://www.ietf.org/mailman/listinfo/spud

- A proposal to create a "Measurement and Analysis for Protocols (MAP)" IRTF Research Group is expected to serve as a forum to exchange insights derived from measuring the Internet, including the possibly to design protocols based on measured path characteristics, rather than conservatively designing just for the worse case.
- The European collaborative research project "NEAT" plans to implement a transport system, following the requirements detailed in § VIII, that will allow transport decisions to be made and verified *at run-time*, instead of design time, based on understanding application needs and the available transport protocols—this is key to breaking the vicious circle and enabling deployment of new transports[6].
- The European collaborative research project "MAMI" is set to perform a large-scale assessment of middlebox behavior, and to use this to inform development of an architecture for middlebox cooperation[7].

ACKNOWLEDGMENT

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 644334 (NEAT). The views expressed are solely those of the author(s).

REFERENCES

[1] J. Postel, "Transmission Control Protocol," RFC 793 (INTERNET STANDARD), Internet Engineering Task Force, Sep. 1981, updated by RFCs 1122, 3168, 6093, 6528. [Online]. Available: http://www.ietf.org/rfc/rfc793.txt

[2] M. Handley, "Why the Internet only just works," *BT Technology Journal*, vol. 24, no. 3, pp. 119–129, 2006.

[3] J. Postel, "User Datagram Protocol," RFC 768 (INTERNET STANDARD), Internet Engineering Task Force, Aug. 1980. [Online]. Available: http://www.ietf.org/rfc/rfc768.txt

[4] E. Kohler, M. Handley, and S. Floyd, "Datagram Congestion Control Protocol (DCCP)," RFC 4340 (Proposed Standard), Internet Engineering Task Force, Mar. 2006, updated by RFCs 5595, 5596, 6335, 6773. [Online]. Available: http://www.ietf.org/rfc/rfc4340.txt

[5] R. Stewart, "Stream Control Transmission Protocol," RFC 4960 (Proposed Standard), Internet Engineering Task Force, Sep. 2007, updated by RFCs 6096, 6335, 7053. [Online]. Available: http://www.ietf.org/rfc/rfc4960.txt

[6] L.-A. Larzon, M. Degermark, S. Pink, L.-E. Jonsson, and G. Fairhurst, "The Lightweight User Datagram Protocol (UDP-Lite)," RFC 3828 (Proposed Standard), Internet Engineering Task Force, Jul. 2004, updated by RFC 6335. [Online]. Available: http://www.ietf.org/rfc/rfc3828.txt

[7] J. Rosenberg, "UDP and TCP as the new waist of the Internet hourglass," Working Draft, IETF Secretariat, Internet-Draft draft-rosenberg-internet-waist-hourglass-00, February 2008, http://www.ietf.org/internet-drafts/draft-rosenberg-internet-waist-hourglass-00.txt. [Online]. Available: http://www.ietf.org/internet-drafts/draft-rosenberg-internet-waist-hourglass-00.txt

[8] L. Popa, A. Ghodsi, and I. Stoica, "HTTP as the narrow waist of the future Internet," in *Proceedings of the ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets)*. ACM, 2010, p. 6.

[9] B. Ford and J. R. Iyengar, "Breaking up the transport logjam." in *Proceedings of the ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets)*, 2008, pp. 85–90.

[10] A. Medina, M. Allman, and S. Floyd, "Measuring the evolution of transport protocols in the Internet," *ACM SIGCOMM Computer Communications Review*, vol. 35, no. 2, pp. 37–52, 2005.

[6] https://www.neat-project.org

[7] https://mami-project.eu

[11] B. Carpenter and S. Brim, "Middleboxes: Taxonomy and Issues," RFC 3234 (Informational), Internet Engineering Task Force, Feb. 2002. [Online]. Available: http://www.ietf.org/rfc/rfc3234.txt

[12] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda, "Is it still possible to extend TCP?" in *Proceedings of ACM IMC*, 2011, pp. 181–194.

[13] D. D. Clark, J. Wroclawski, K. R. Sollins, and R. Braden, "Tussle in cyberspace: defining tomorrow's Internet," in *ACM SIGCOMM Computer Communications Review*, vol. 32, no. 4. ACM, 2002, pp. 347–356.

[14] B. Ford, P. Srisuresh, and D. Kegel, "Peer-to-peer communication across network address translators." in *USENIX Annual Technical Conference, General Track*, 2005, pp. 179–192.

[15] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, "SIP: Session Initiation Protocol," RFC 3261 (Proposed Standard), Internet Engineering Task Force, Jun. 2002, updated by RFCs 3265, 3853, 4320, 4916, 5393, 5621, 5626, 5630, 5922, 5954, 6026, 6141, 6665, 6878, 7462, 7463. [Online]. Available: http://www.ietf.org/rfc/rfc3261.txt

[16] J. Postel and J. Reynolds, "File Transfer Protocol," RFC 959 (INTERNET STANDARD), Internet Engineering Task Force, Oct. 1985, updated by RFCs 2228, 2640, 2773, 3659, 5797, 7151. [Online]. Available: http://www.ietf.org/rfc/rfc959.txt

[17] A. Langley, "Probing the viability of TCP extensions," *URL http://www.imperialviolet. org/binary/ecntest. pdf*, 2008.

[18] G. Detal, B. Hesmans, O. Bonaventure, Y. Vanaubel, and B. Donnet, "Revealing middlebox interference with tracebox," in *Proceedings of ACM IMC*, 2013, pp. 1–8.

[19] R. Craven, R. Beverly, and M. Allman, "A middlebox-cooperative TCP for a non end-to-end Internet," in *Proceedings of ACM SIGCOMM*, 2014, pp. 151–162.

[20] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses," RFC 6824 (Experimental), Internet Engineering Task Force, Jan. 2013. [Online]. Available: http://www.ietf.org/rfc/rfc6824.txt

[21] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, M. Handley *et al.*, "How hard can it be? Designing and implementing a deployable multipath TCP." in *Proceedings of USENIX NSDI*, vol. 12, 2012, pp. 29–29.

[22] A. Bittau, M. Hamburg, M. Handley, D. Mazieres, and D. Boneh, "The case for ubiquitous transport-level encryption." in *USENIX Security Symposium*, 2010, pp. 403–418.

[23] A. Bittau, D. Boneh, M. Hamburg, M. Handley, D. Mazieres, and Q. Slack, "Cryptographic protection of TCP streams (tcpcrypt)," Working Draft, IETF Secretariat, Internet-Draft draft-bittau-tcp-crypt-04, February 2014, http://www.ietf.org/internet-drafts/draft-bittau-tcp-crypt-04.txt. [Online]. Available: http://www.ietf.org/internet-drafts/draft-bittau-tcp-crypt-04.txt

[24] T. Flach, N. Dukkipati, A. Terzis, B. Raghavan, N. Cardwell, Y. Cheng, A. Jain, S. Hao, E. Katz-Bassett, and R. Govindan, "Reducing web latency: the virtue of gentle aggression," in *Proceedings of ACM SIGCOMM*, vol. 43, no. 4, 2013, pp. 159–170.

[25] A. Ramaiah, "TCP option space extension," Working Draft, IETF Secretariat, Internet-Draft draft-ananth-tcpm-tcpoptext-00, March 2012, http://www.ietf.org/internet-drafts/draft-ananth-tcpm-tcpoptext-00.txt. [Online]. Available: http://www.ietf.org/internet-drafts/draft-ananth-tcpm-tcpoptext-00.txt

[26] J. Touch and W. Eddy, "TCP extended data offset option," Working Draft, IETF Secretariat, Internet-Draft draft-ietf-tcpm-tcp-edo-03, April 2015, http://www.ietf.org/internet-drafts/draft-ietf-tcpm-tcp-edo-03.txt. [Online]. Available: http://www.ietf.org/internet-drafts/draft-ietf-tcpm-tcp-edo-03.txt

[27] H. Trieu, J. Touch, and T. Faber, "Implementation of the TCP extended data offset option," USC/ISI, Tech. Rep. ISI-TR-696, March 2015.

[28] J. Touch and T. Faber, "TCP SYN extended option space using an out-of-band segment," Working Draft, IETF Secretariat, Internet-Draft draft-touch-tcpm-tcp-syn-ext-opt-02, April 2015, http://www.ietf.org/internet-drafts/draft-touch-tcpm-tcp-syn-ext-opt-02.txt. [Online]. Available: http://www.ietf.org/internet-drafts/draft-touch-tcpm-tcp-syn-ext-opt-02.txt

[29] B. Briscoe, "Inner space for TCP options," Working Draft, IETF Secretariat, Internet-Draft draft-briscoe-tcpm-inner-space-00, October 2014, http://www.ietf.org/internet-drafts/draft-briscoe-tcpm-inner-space-00.txt. [Online]. Available: http://www.ietf.org/internet-drafts/draft-briscoe-tcpm-inner-space-00.txt

[30] Z. Nabi, T. Moncaster, A. Madhavapeddy, S. Hand, and J. Crowcroft, "Evolving TCP.: how hard can it be?" in *Proceedings of the ACM CoNEXT student workshop*, 2012, pp. 35–36.

[31] B. Trammell and M. Kuehlewind, "Report from the IAB Workshop on Stack Evolution in a Middlebox Internet (SEMI)," RFC 7663 (Informational), Internet Engineering Task Force, Oct. 2015. [Online]. Available: http://www.ietf.org/rfc/rfc7663.txt

[32] IRTF Measurement and Analysis for Protocols (MAP) Proposed Research Group Charter. [Online]. Available: https://datatracker.ietf.org/doc/charter-irtf-maprg/

[33] F. Audet and C. Jennings, "Network Address Translation (NAT) Behavioral Requirements for Unicast UDP," RFC 4787 (Best Current Practice), Internet Engineering Task Force, Jan. 2007, updated by RFC 6888. [Online]. Available: http://www.ietf.org/rfc/rfc4787.txt

[34] L. Eggert, G. Fairhurst, and G. Shepherd, "UDP usage guidelines," Working Draft, IETF Secretariat, Internet-Draft draft-ietf-tsvwg-rfc5405bis, Nov. 2015, https://tools.ietf.org/html/draft-ietf-tsvwg-rfc5405bis. [Online]. Available: https://tools.ietf.org/html/draft-ietf-tsvwg-rfc5405bis

[35] M. F. Nowlan, N. Tiwari, J. Iyengar, S. O. Aminy, and B. Fordy, "Fitting square pegs through round pipes: unordered delivery wire-compatible with TCP and TLS," in *Proceedings of USENIX NSDI*. USENIX Association, 2012, pp. 28–28.

[36] J. Iyengar, S. Cheshire, and J. Graessley, "Minion - wire protocol," Working Draft, IETF Secretariat, Internet-Draft draft-iyengar-minion-protocol-02, October 2013, http://www.ietf.org/internet-drafts/draft-iyengar-minion-protocol-02.txt. [Online]. Available: http://www.ietf.org/internet-drafts/draft-iyengar-minion-protocol-02.txt

[37] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications," RFC 3550 (INTERNET STANDARD), Internet Engineering Task Force, Jul. 2003, updated by RFCs 5506, 5761, 6051, 6222, 7022, 7160, 7164. [Online]. Available: http://www.ietf.org/rfc/rfc3550.txt

[38] J. Roskind, "QUIC: Multiplexed stream transport over UDP," *Google working design document*, 2013.

[39] S. R. Das, "Evaluation of QUIC on web page performance," Ph.D. dissertation, Massachusetts Institute of Technology, 2014.

[40] Google. SPDY: An experimental protocol for a faster web. [Online]. Available: http://www.chromium.org/spdy/spdy-whitepaper

[41] M. Belshe, R. Peon, and M. Thomson, "Hypertext Transfer Protocol Version 2 (HTTP/2)," RFC 7540 (Proposed Standard), Internet Engineering Task Force, May 2015. [Online]. Available: http://www.ietf.org/rfc/rfc7540.txt

[42] M. Thornburgh, "Adobe's Secure Real-Time Media Flow Protocol," RFC 7016 (Informational), Internet Engineering Task Force, Nov. 2013. [Online]. Available: http://www.ietf.org/rfc/rfc7016.txt

[43] E. Rescorla and N. Modadugu, "Datagram Transport Layer Security Version 1.2," RFC 6347 (Proposed Standard), Internet Engineering Task Force, Jan. 2012, updated by RFC 7507. [Online]. Available: http://www.ietf.org/rfc/rfc6347.txt

[44] A. Norberg, "uTorrent transport protocol," *BitTorrent Extension Protocol*, vol. 29, 2009.

[45] Y. Gu and R. L. Grossman, "UDT: UDP-based data transfer for high-speed wide area networks," *Computer Networks*, vol. 51, no. 7, pp. 1777–1799, 2007.

[46] Y. Gu and R. Grossman, "UDTv4: Improvements in performance and usability," in *Networks for Grid Applications*. Springer, 2009, pp. 9–23.

[47] B. Ford, "Structured streams: a new transport abstraction," in *Proceedings of ACM SIGCOMM*, vol. 37, no. 4, 2007, pp. 361–372.

[48] M. Tuexen and R. Stewart, "UDP Encapsulation of Stream Control Transmission Protocol (SCTP) Packets for End-Host to End-Host Communication," RFC 6951 (Proposed Standard), Internet Engineering Task Force, May 2013. [Online]. Available: http://www.ietf.org/rfc/rfc6951.txt

[49] T. Phelan, G. Fairhurst, and C. Perkins, "DCCP-UDP: A Datagram Congestion Control Protocol UDP Encapsulation for NAT Traversal," RFC 6773 (Proposed Standard), Internet Engineering Task Force, Nov. 2012. [Online]. Available: http://www.ietf.org/rfc/rfc6773.txt

[50] S. Cheshire, J. Graessley, and R. McGuire, "Encapsulation of TCP and other transport protocols over UDP," Working Draft, IETF Secretariat, Internet-Draft draft-cheshire-tcp-over-udp-00, July 2013, http://www.ietf.org/internet-drafts/draft-cheshire-tcp-over-udp-00.txt. [Online]. Available: http://www.ietf.org/internet-drafts/draft-cheshire-tcp-over-udp-00.txt

[51] J. Manner, N. Varis, and B. Briscoe, "Generic UDP Tunnelling (GUT)," Working Draft, IETF Secretariat, Internet-Draft draft-manner-tsvwg-gut-02, July 2010, http://www.ietf.org/internet-drafts/draft-manner-tsvwg-gut-02.txt. [Online]. Available: http://www.ietf.org/internet-drafts/draft-manner-tsvwg-gut-02.txt

[52] T. Herbert, L. Yong, and O. Zia, "Generic UDP encapsulation," Working Draft, IETF Secretariat, Internet-Draft draft-ietf-nvo3-gue-00, April 2015, http://www.ietf.org/internet-drafts/draft-ietf-nvo3-gue-00.txt. [Online]. Available: http://www.ietf.org/internet-drafts/draft-ietf-nvo3-gue-00.txt

[53] B. Penoff, A. Wagner, M. Tuxen, and I. Rungeler, "Portable and performant userspace SCTP stack," in *21st International Conference on Computer Communications and Networks (ICCCN)*. IEEE, 2012, pp. 1–9.

[54] S. McQuistin and C. Perkins, "Reinterpreting the transport protocol stack to embrace ossification," in *Report from the IAB Workshop on Stack Evolution in a Middlebox Internet (SEMI)*, 2015. [Online]. Available: https://www.iab.org/activities/workshops/semi/

[55] C. Huitema, E. Rescorla, and J. Iyengar, "DTLS as subtransport protocol," Working Draft, IETF Secretariat, Internet-Draft draft-huitema-tls-dtls-as-subtransport-00, March 2015, http://www.ietf.org/internet-drafts/draft-huitema-tls-dtls-as-subtransport-00.txt. [Online]. Available: http://www.ietf.org/internet-drafts/draft-huitema-tls-dtls-as-subtransport-00.txt

[56] H. Schulzrinne, A. Rao, and R. Lanphier, "Real Time Streaming Protocol (RTSP)," RFC 2326 (Proposed Standard), Internet Engineering Task Force, Apr. 1998. [Online]. Available: http://www.ietf.org/rfc/rfc2326.txt

[57] J. Rosenberg, "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols," RFC 5245 (Proposed Standard), Internet Engineering Task Force, Apr. 2010, updated by RFC 6336. [Online]. Available: http://www.ietf.org/rfc/rfc5245.txt

[58] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing, "Session Traversal Utilities for NAT (STUN)," RFC 5389 (Proposed Standard), Internet Engineering Task Force, Oct. 2008, updated by RFC 7350. [Online]. Available: http://www.ietf.org/rfc/rfc5389.txt

[59] M. Petit-Huguenin, "Traversal Using Relays around NAT (TURN) Resolution Mechanism," RFC 5928 (Proposed Standard), Internet Engineering Task Force, Aug. 2010, updated by RFC 7350. [Online]. Available: http://www.ietf.org/rfc/rfc5928.txt

[60] J. Rosenberg, A. Keranen, B. B. Lowekamp, and A. B. Roach, "TCP Candidates with Interactive Connectivity Establishment (ICE)," RFC 6544 (Proposed Standard), Internet Engineering Task Force, Mar. 2012. [Online]. Available: http://www.ietf.org/rfc/rfc6544.txt

[61] A. Biggadike, D. Ferullo, G. Wilson, and A. Perrig, "NATBLASTER: Establishing TCP connections between hosts behind NATs," in *ACM SIGCOMM Asia Workshop*, vol. 5, 2005.

[62] S. Perreault and J. Rosenberg, "Traversal Using Relays around NAT (TURN) Extensions for TCP Allocations," RFC 6062 (Proposed Standard), Internet Engineering Task Force, Nov. 2010. [Online]. Available: http://www.ietf.org/rfc/rfc6062.txt

[63] S. Guha, K. Biswas, B. Ford, S. Sivakumar, and P. Srisuresh, "NAT Behavioral Requirements for TCP," RFC 5382 (Best Current Practice), Internet Engineering Task Force, Oct. 2008. [Online]. Available: http://www.ietf.org/rfc/rfc5382.txt

[64] D. Wing, S. Cheshire, M. Boucadair, R. Penno, and P. Selkirk, "Port Control Protocol (PCP)," RFC 6887 (Proposed Standard), Internet Engineering Task Force, Apr. 2013, updated by RFCs 7488, 7652. [Online]. Available: http://www.ietf.org/rfc/rfc6887.txt

[65] S. Cheshire and M. Krochmal, "NAT Port Mapping Protocol (NAT-PMP)," RFC 6886 (Informational), Internet Engineering Task Force, Apr. 2013. [Online]. Available: http://www.ietf.org/rfc/rfc6886.txt

[66] P. Srisuresh, J. Kuthan, J. Rosenberg, A. Molitor, and A. Rayhan, "Middlebox communication architecture and framework," RFC 3303 (Informational), Internet Engineering Task Force, Aug. 2002. [Online]. Available: http://www.ietf.org/rfc/rfc3303.txt

[67] M. Stiemerling, H. Tschofenig, C. Aoun, and E. Davies, "NAT/Firewall NSIS Signaling Layer Protocol (NSLP)," RFC 5973 (Experimental), Internet Engineering Task Force, Oct. 2010. [Online]. Available: http://www.ietf.org/rfc/rfc5973.txt

[68] B. Trammell and J. Hildebrand, "Evolving transport in the Internet," *IEEE Internet Computing*, vol. 18, no. 5, pp. 60–64, 2014.

[69] B. Trammell, "Architectural considerations for transport evolution with explicit path cooperation," Working Draft, IETF Secretariat, Internet-Draft draft-trammell-stackevo-explicit-coop-00, September 2015, http://www.ietf.

org/internet-drafts/draft-trammell-stackevo-explicit-coop-00.txt. [Online]. Available: http://www.ietf.org/internet-drafts/draft-trammell-stackevo-explicit-coop-00.txt

[70] J. Hildebrand and B. Trammell, "Substrate Protocol for User Datagrams (SPUD) prototype," Working Draft, IETF Secretariat, Internet-Draft draft-hildebrand-spud-prototype-03, March 2015, http://www.ietf.org/internet-drafts/draft-hildebrand-spud-prototype-03.txt. [Online]. Available: http://www.ietf.org/internet-drafts/draft-hildebrand-spud-prototype-03.txt

[71] J. Nagle, "Congestion control in TCP/IP internetworks," *ACM SIGCOMM Computer Communications Review*, vol. 14, no. 4, pp. 11–17, Oct 1984.

[72] R. Davoli and M. Goldweber, "Msocket: Multiple stack support for the Berkeley socket API," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC)*. New York, NY, USA: ACM, 2012, pp. 588–593. [Online]. Available: http://doi.acm.org/10.1145/2245276.2245390

[73] T. Enghardt, "Socket intents: Extending the socket API to express application needs," Master Thesis Report, Department of Computer Science, Tech. Rep., July 2013.

[74] P. S. Schmidt, T. Enghardt, R. Khalili, and A. Feldmann, "Socket Intents: Leveraging application awareness for multi-access connectivity," in *Proceedings of ACM CoNEXT*, Santa Barbara, California, USA, Dec. 9–12, 2013, pp. 295–300.

[75] B. D. Higgins, A. Reda, T. Alperovich, J. Flinn, T. J. Giuli, B. Noble, and D. Watson, "Intentional networking: opportunistic exploitation of mobile network diversity," in *Proceedings of ACM MOBICOM*. New York, NY, USA: ACM, 2010, pp. 73–84. [Online]. Available: http://doi.acm.org/10.1145/1859995.1860005

[76] S. Bocking, "Sockets++: a uniform application programming interface for basic level communication services," *IEEE Communications Magazine*, vol. 34, no. 12, pp. 114–123, Dec 1996.

[77] P. G. S. Florissi, Y. Yemini, and D. Florissi, "QoSockets: a new extension to the sockets API for end-to-end application QoS management," *Computer Networks*, vol. 35, no. 1, pp. 57–76, 2001.

[78] H. Abbasi, C. Poellabauer, K. Schwan, G. Losik, and R. West, "A quality-of-service enhanced socket API in GNU/Linux," in *Proceedings of the 4th Real-Time Linux Workshop, Boston, Massachusetts*. Citeseer, 2002.

[79] G. Fairhurst, B. Trammell, and M. Kuehlewind, "Services provided by IETF transport protocols and congestion control mechanisms," Working Draft, IETF Secretariat, Internet-Draft draft-ietf-taps-transports-03, February 2015, http://www.ietf.org/internet-drafts/draft-ietf-taps-transports-03.txt. [Online]. Available: http://www.ietf.org/internet-drafts/draft-ietf-taps-transports-03.txt

[80] D. Henrici and B. Reuther, "Service-oriented protocol interfaces and dynamic intermediation of communication services," in *Proceedings of the 2nd IASTED nternational Conference on Communications, Internet and Information Technology (CIIT); Scottsdale, Arizona, USA*, November 2003.

[81] B. Reuther, D. Henrici, and M. Hillenbrand, "Dance: dynamic application oriented network services," in *Proceedings of the 30th Euromicro Conference*. IEEE, 2004, pp. 298–305.

[82] D. Velten, R. Hinden, and J. Sax, "Reliable Data Protocol," RFC 908 (Experimental), Internet Engineering Task Force, Jul. 1984, updated by RFC 1151. [Online]. Available: http://www.ietf.org/rfc/rfc908.txt

[83] T. Strayer ed., "Xpress transport protocol specification, revision 4.0b," XTP Forum Inc., 1998.

[84] M. Welzl, "A case for middleware to enable advanced Internet services," in *Proceedings of Next Generation Network Middleware Workshop (NGNM'04)*, Athens, Greece, May 14, 2004, pp. 20/1–20/5.

[85] M. Welzl, S. Jorer, and S. Gjessing, "Towards a protocol-independent Internet transport API," in *Proceedings of IEEE ICC*, Kyoto, Japan, Jun. 5–9, 2011, pp. 1–6.

[86] A. A. Siddiqui and P. Mueller, "A requirement-based socket API for a transition to future Internet architectures," in *6th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS)*, Palermo, Italy, Jul. 4–6, 2012, pp. 340–345.

[87] T. Berners-Lee, R. Fielding, and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax," RFC 3986 (INTERNET STANDARD), Internet Engineering Task Force, Jan. 2005, updated by RFCs 6874, 7320. [Online]. Available: http://www.ietf.org/rfc/rfc3986.txt

[88] 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-usage, work in progress, Jan. 2016. [Online]. Available: https://tools.ietf.org/html/draft-ietf-taps-transports-usage

[89] M. Kuehlewind and B. Trammell, "SPUD use cases," Working Draft, IETF Secretariat, Internet-Draft draft-kuehlewind-spud-use-cases-00, July 2015, http://www.ietf.org/internet-drafts/draft-kuehlewind-spud-use-cases-00.txt. [Online]. Available: http://www.ietf.org/internet-drafts/draft-kuehlewind-spud-use-cases-00.txt

[90] T. Eckert, R. Penno, A. Choukir, and C. Eckel, "A framework for signaling flow characteristics between applications and the network," Working Draft, IETF Secretariat, Internet-Draft draft-eckert-intarea-flow-metadata-framework-02, October 2013, http://www.ietf.org/internet-drafts/draft-eckert-intarea-flow-metadata-framework-02.txt. [Online]. Available: http://www.ietf.org/internet-drafts/draft-eckert-intarea-flow-metadata-framework-02.txt

[91] S. Floyd, M. Allman, A. Jain, and P. Sarolahti, "Quick-Start for TCP and IP," RFC 4782 (Experimental), Internet Engineering Task Force, Jan. 2007. [Online]. Available: http://www.ietf.org/rfc/rfc4782.txt

[92] IETF Transport Services (TAPS) Working Group Charter. [Online]. Available: https://datatracker.ietf.org/doc/charter-ietf-taps/

[93] M. Rose, "The Blocks Extensible Exchange Protocol Core," RFC 3080 (Proposed Standard), Internet Engineering Task Force, Mar. 2001. [Online]. Available: http://www.ietf.org/rfc/rfc3080.txt

[94] J. Rosenberg and H. Schulzrinne, "An Offer/Answer Model with Session Description Protocol (SDP)," RFC 3264 (Proposed Standard), Internet Engineering Task Force, Jun. 2002, updated by RFC 6157. [Online]. Available: http://www.ietf.org/rfc/rfc3264.txt

[95] B. Ford and J. R. Iyengar, "Efficient cross-layer negotiation." in *Proceedings of the ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets)*, 2009.

[96] D. Wing, A. Yourtchenko, and P. Natarajan, "Happy eyeballs: trending towards success (IPv6 and SCTP)," Working Draft, IETF Secretariat, Internet-Draft draft-wing-http-new-tech-01, August 2010, http://www.ietf.org/internet-drafts/draft-wing-http-new-tech-01.txt. [Online]. Available: http://www.ietf.org/internet-drafts/draft-wing-http-new-tech-01.txt

[97] D. Wing and A. Yourtchenko, "Happy Eyeballs: Success with Dual-Stack Hosts," RFC 6555 (Proposed Standard), Internet Engineering Task Force, Apr. 2012. [Online]. Available: http://www.ietf.org/rfc/rfc6555.txt

[98] D. Schinazi, "Apple and IPv6 — Happy Eyeballs," Email to the IETF v6ops mailing list, Jul. 2015. [Online]. Available: https://www.ietf.org/mail-archive/web/v6ops/current/msg22455.html

[99] D. Thaler, R. Draves, A. Matsumoto, and T. Chown, "Default Address Selection for Internet Protocol Version 6 (IPv6)," RFC 6724 (Proposed Standard), Internet Engineering Task Force, Sep. 2012. [Online]. Available: http://www.ietf.org/rfc/rfc6724.txt

[100] M. Honda, F. Huici, C. Raiciu, J. Araujo, and L. Rizzo, "Rekindling network protocol innovation with user-level stacks," *ACM SIGCOMM Computer Communications Review*, vol. 44, no. 2, pp. 52–58, 2014.

[101] "netmap," http://info.iet.unipi.it/~luigi/netmap/.

[102] L. Rizzo and G. Lettieri, "VALE, a switched Ethernet for virtual machines," in *Proceedings of ACM CoNEXT*, 2012, pp. 61–72.

[103] Intel. Data plane development kit. [Online]. Available: http://www.dpdk.org

[104] U. A. Camar and J. Baudy. PACKET-MMAP. [Online]. Available: https://www.kernel.org/doc/Documentation/networking/packet_mmap.txt

[105] P. Kelsey, "Userspace networking with libuinet," May 2014, presented at the Technical BSD Conference (BSDCan '14). [Online]. Available: https://www.bsdcan.org/2014/schedule/attachments/260_libuinet_bsdcan2014.pdf

[106] J. Dike, *User mode linux*. Prentice Hall Englewood Cliffs, 2006, vol. 2.

[107] H. Tazaki, R. Nakamura, and Y. Sekiya, "Library operating system with mainline Linux network stack," Feb. 2015, presented at NetDev 0.1. [Online]. Available: https://www.netdev01.org/docs/netdev01-tazaki-libos.pdf

[108] "NUSE," https://github.com/libos-nuse/linux-libos-tools.

[109] S. A. Baset and H. G. Schulzrinne, "An analysis of the Skype peer-to-peer Internet telephony protocol," in *Proceedings of IEEE INFOCOM*, Barcelona, Apr. 2006.

[110] A. Bergkvist, D. Burnett, C. Jennings, and A. Narayanan, "WebRTC 1.0: Real-time communication between browsers," *W3C, W3C Working Draft 10, Feb*, 2015.

C Internet-draft: *On the Usage of Transport Service Features Provided by IETF Transport Protocols*

The following Internet Draft [21], a Working Group Item of the IETF Transport Services working group (TAPS), has been produced by project participants and was used to derive the basic API primitives and events in § 3.2.

TAPS
Internet-Draft
Intended status: Informational
Expires: July 11, 2016

M. Welzl
University of Oslo
M. Tuexen
Muenster Univ. of Appl. Sciences
N. Khademi
University of Oslo
January 8, 2016

On the Usage of Transport Service Features Provided by IETF Transport Protocols
draft-ietf-taps-transports-usage-00

Abstract

This document describes how transport protocols expose services to applications and how an application can configure and use the features of a transport service.

Status of this Memo

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

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

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

This Internet-Draft will expire on July 11, 2016.

Copyright Notice

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

Table of Contents

1. Terminology

Transport Service Feature: a specific end-to-end feature that a transport service provides to its clients. Examples include confidentiality, reliable delivery, ordered delivery, message-versus-stream orientation, etc.

Transport Service: a set of transport service 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 Protocol Component: an implementation of a transport service feature within a protocol.

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 of data across the network (this may also be an upper layer protocol or tunnel encapsulation).

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.

Primitive: a function call that is used to locally communicate between an application and a transport endpoint and is related to one or more Transport Service Features.

Parameter: a value passed between an application and a transport protocol by a primitive.

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

2. Introduction

This document presents defined interactions between transport protocols and applications in the form of 'primitives' (function calls). Primitives can be invoked by an application or a transport protocol; the latter type is called an "event". The list of transport service features and primitives in this document is strictly based on the parts of protocol specifications that relate to what the protocol provides to an application using it and how the application interacts with it. It does not cover parts of a protocol that are explicitly stated as optional to implement.

The document presents a three-pass process to arrive at a list of transport service features. In the first pass, the relevant RFC text

is discussed per protocol. In the second pass, this discussion is used to derive a list of primitives that are uniformly categorized across protocols. Here, an attempt is made to present or -- where text describing primitives does not yet exist -- construct primitives in a slightly generalized form to highlight similarities. This is, for example, achieved by renaming primitives of protocols or by avoiding a strict 1:1-mapping between the primitives in the protocol specification and primitives in the list. Finally, the third pass presents transport service features based on pass 2, identifying which protocols implement them.

In the list resulting from the second pass, some transport service features are missing because they are implicit in some protocols, and they only become explicit when we consider the superset of all features offered by all protocols. For example, TCP's reliability includes integrity via a checksum, but we have to include a protocol like UDP-Lite as specified in [RFC3828] (which has a configurable checksum) in the list before we can consider an always-on checksum as a transport service feature. Similar arguments apply to other protocol functions (e.g. congestion control). The complete list of features across all protocols is therefore only available after pass 3.

This document discusses unicast transport protocols. [AUTHOR'S NOTE: we skip "congestion control mechanisms" for now. This simplifies the discussion; the congestion control mechanisms part is about LEDBAT, which should be easy to add later.] Transport protocols provide communication between processes that operate on network endpoints, which means that they allow for multiplexing of communication between the same IP addresses, and normally this multiplexing is achieved using port numbers. Port multiplexing is therefore assumed to be always provided and not discussed in this document.

Some protocols are connection-oriented. Connection-oriented protocols often use an initial call to a specific transport primitive to open a connection before communication can progress, and require communication to be explicitly terminated by issuing another call to a transport primitive (usually called "close"). A "connection" is the common state that some transport primitives refer to, e.g., to adjust general configuration settings. Connection establishment, maintenance and termination are therefore used to categorize transport primitives of connection-oriented transport protocols in pass 2 and pass 3.

3. Pass 1

This first iteration summarizes the relevant text parts of the RFCs

describing the protocols, focusing on what each transport protocol provides to the application and how it is used (abstract API descriptions, where they are available).

3.1. Primitives Provided by TCP

[RFC0793] states: "The Transmission Control Protocol (TCP) is intended for use as a highly reliable host-to-host protocol between hosts in packet-switched computer communication networks, and in interconnected systems of such networks". Section 3.8 in [RFC0793] further specifies the interaction with the application by listing several transport primitives. It is also assumed that an Operating System provides a means for TCP to asynchronously signal the application; the primitives representing such signals are called 'events' in this section. This section describes the relevant primitives.

open: this is either active or passive, to initiate a connection or listen for incoming connections. All other primitives are associated with a specific connection, which is assumed to first have been opened. An active open call contains a socket. A passive open call with a socket waits for a particular connection; alternatively, a passive open call can leave the socket unspecified to accept any incoming connection. A fully specified passive call can later be made active by calling 'send'. Optionally, a timeout can be specified, after which TCP will abort the connection if data has not been successfully delivered to the destination (else a default timeout value is used). [RFC1122] describes a procedure for aborting the connection that must be used to avoid excessive retransmissions, and states that an application must be able to control the threshold used to determine the condition for aborting -- and that this threshold may be measured in time units or as a count of retransmission. This indicates that the timeout could also be specified as a count of retransmission.

Also optional, for multihomed hosts, the local IP address can be provided [RFC1122]. If it is not provided, a default choice will be made in case of active open calls. A passive open call will await incoming connection requests to all local addresses and then maintain usage of the local IP address where the incoming connection request has arrived. Finally, the 'options' parameter is explained in [RFC1122] to allow the application to specify IP options such as source route, record route, or timestamp. It is not stated on which segments of a connection these options should be applied, but probably all segments, as this is also stated in a specification given for the usage of source route (section 4.2.3.8 of [RFC1122]). Source route is the only non-optional IP option in

this parameter, allowing an application to specify a source route when it actively opens a TCP connection.

send: this is the primitive that an application uses to give the local TCP transport endpoint a number of bytes that TCP should reliably send to the other side of the connection. The URGENT flag, if set, states that the data handed over by this send call is urgent and this urgency should be indicated to the receiving process in case the receiving application has not yet consumed all non-urgent data preceding it. An optional timeout parameter can be provided that updates the connection's timeout (see 'open').

receive: This primitive allocates a receiving buffer for a provided number of bytes. It returns the number of received bytes provided in the buffer when these bytes have been received and written into the buffer by TCP. The application is informed of urgent data via an URGENT flag: if it is on, there is urgent data. If it is off, there is no urgent data or this call to 'receive' has returned all the urgent data.

close: This primitive closes one side of a connection. It is semantically equivalent to "I have no more data to send" but does not mean "I will not receive any more", as the other side may still have data to send. This call reliably delivers any data that has already been given to TCP (and if that fails, 'close' becomes 'abort').

abort: This primitive causes all pending 'send' and 'receive' calls to be aborted. A TCP RESET message is sent to the TCP endpoint on the other side of the connection [RFC0793].

close event: TCP uses this primitive to inform an application that the application on the other side has called the 'close' primitive, so the local application can also issue a 'close' and terminate the connection gracefully. See [RFC0793], Section 3.5.

abort event: When TCP aborts a connection upon receiving a "Reset" from the peer, it "advises the user and goes to the CLOSED state." See [RFC0793], Section 3.4.

USER TIMEOUT event: This event, described in Section 3.9 of [RFC0793], is executed when the user timeout expires (see 'open'). All queues are flushed and the application is informed that the connection had to be aborted due to user timeout.

ERROR_REPORT event: This event, described in Section 4.2.4.1 of [RFC1122], informs the application of "soft errors" that can be safely ignored [RFC5461], including the arrival of an ICMP error message or excessive retransmissions (reaching a threshold below the threshold where the connection is aborted).

Type-of-Service: Section 4.2.4.2 of [RFC1122] states that the application layer MUST be able to specify the Type-of-Service (TOS) for segments that are sent on a connection. The application should be able to change the TOS during the connection lifetime, and the TOS value should be passed to the IP layer unchanged. Since then the TOS field has been redefined. A part of the field has been assigned to ECN [RFC3168] and the six most significant bits have been assigned to carry the DiffServ CodePoint, DSField [RFC3260]. Staying with the intention behind the application's ability to specify the "Type of Service", this should probably be interpreted to mean the value in the DSField, which is the Differentiated Services Codepoint (DSCP).

Nagle: The Nagle algorithm, described in Section 4.2.3.4 of [RFC1122], delays sending data for some time to increase the likelihood of sending a full-sized segment. An application can disable the Nagle algorithm for an individual connection.

User Timeout Option: The User Timeout Option (UTO) [RFC5482] allows one end of a TCP connection to advertise its current user timeout value so that the other end of the TCP connection can adapt its own user timeout accordingly. In addition to the configurable value of the User Timeout (see 'send'), [RFC5482] introduces three per-connection state variables that an application can adjust to control the operation of the User Timeout Option (UTO): ADV_UTO is the value of the UTO advertised to the remote TCP peer (default: system-wide default user timeout); ENABLED (default false) is a boolean-type flag that controls whether the UTO option is enabled for a connection. This applies to both sending and receiving. CHANGEABLE is a boolean-type flag (default true) that controls whether the user timeout may be changed based on a UTO option received from the other end of the connection. CHANGEABLE becomes false when an application explicitly sets the user timeout (see 'send').

3.1.1. Excluded Primitives

The 'open' primitive specified in [RFC0793] can be handed optional Precedence or security/compartment information according to [RFC0793], but this was not included here because it is mostly irrelevant today, as explained in [RFC7414].

The 'status' primitive was not included because [RFC0793] describes this primitive as "implementation dependent" and states that it "could be excluded without adverse effect". Moreover, while a data block containing specific information is described, it is also stated that not all of this information may always be available. The 'send' primitive described in [RFC0793] includes an optional PUSH flag which, if set, requires data to be promptly transmitted to the receiver without delay; the 'receive' primitive described in [RFC0793] can (under some conditions) yield the status of the PUSH flag. Because PUSH functionality is made optional to implement for both the 'send' and 'receive' primitives in [RFC1122], this functionality is not included here. [RFC1122] also introduces keep-alives to TCP, but these are optional to implement and hence not considered here. [RFC1122] describes that "some TCP implementations have included a FLUSH call", indicating that this call is also optional to implement. It is therefore not considered here.

3.2. Primitives Provided by SCTP

Section 1.1 of [RFC4960] lists limitations of TCP that SCTP removes. Three of the four mentioned limitations directly translate into a transport service features that are visible to an application using SCTP: 1) it allows for preservation of message delineations; 2) these messages, while reliably transferred, do not require to be in order unless the application wants it; 3) multi-homing is supported. In SCTP, connections are called "association" and they can be between not only two (as in TCP) but multiple addresses at each endpoint.

Section 10 of [RFC4960] further specifies the interaction with the application (which RFC [RFC4960] calls the "Upper Layer Protocol" (ULP)). It is assumed that the Operating System provides a means for SCTP to asynchronously signal the application; the primitives representing such signals are called 'events' in this section. Here, we describe the relevant primitives.

Initialize: Initialize creates a local SCTP instance that it binds to a set of local addresses (and, if provided, port number). Initialize needs to be called only once per set of local addresses.

Associate: This creates an association (the SCTP equivalent of a connection) between the local SCTP instance and a remote SCTP instance. Most primitives are associated with a specific association, which is assumed to first have been created. Associate can return a list of destination transport addresses so that multiple paths can later be used. One of the returned sockets will be selected by the local endpoint as default primary path for sending SCTP packets to this peer, but this choice can be

changed by the application using the list of destination addresses. Associate is also given the number of outgoing streams to request and optionally returns the number of outgoing streams negotiated.

Send: This sends a message of a certain length in bytes over an association. A number can be provided to later refer to the correct message when reporting an error, and a stream id is provided to specify the stream to be used inside an association (we consider this as a mandatory parameter here for simplicity: if not provided, the stream id defaults to 0). An optional maximum life time can specify the time after which the message should be discarded rather than sent. A choice (advisory, i.e. not guaranteed) of the preferred path can be made by providing a socket, and the message can be delivered out-of-order if the unordered flag is set. Another advisory flag indicates whether the application prefers to avoid bundling user data with other outbound DATA chunks (i.e., in the same packet). A payload protocol-id can be provided to pass a value that indicates the type of payload protocol data to the peer.

Receive: Messages are received from an association, and optionally a stream within the association, with their size returned. The application is notified of the availability of data via a DATA ARRIVE notification. If the sender has included a payload protocol-id, this value is also returned. If the received message is only a partial delivery of a whole message, a partial flag will indicate so, in which case the stream id and a stream sequence number are provided to the application.

Shutdown: This primitive gracefully closes an association, reliably delivering any data that has already been handed over to SCTP. A return code informs about success or failure of this procedure.

Abort: This ungracefully closes an association, by discarding any locally queued data and informing the peer that the association was aborted. Optionally, an abort reason to be passed to the peer may be provided by the application. A return code informs about success or failure of this procedure.

Change Heartbeat / Request Heartbeat: This allows the application to enable/disable heartbeats and optionally specify a heartbeat frequency as well as requesting a single heartbeat to be carried out upon a function call, with a notification about success or failure of transmitting the HEARTBEAT chunk to the destination.

Set Protocol Parameters: This allows to set values for protocol parameters per association; for some parameters, a setting can be made per socket. The set listed in [RFC4960] is: RTO.Initial; RTO.Min; RTO.Max; Max.Burst; RTO.Alpha; RTO.Beta; Valid.Cookie.Life; Association.Max.Retrans; Path.Max.Retrans; Max.Init.Retransmits; HB.interval; HB.Max.Burst.

Set Primary: This allows to set a new primary default path for an association by providing a socket. Optionally, a default source address to be used in IP datagrams can be provided.

Status: The 'Status' primitive returns a data block with information about a specified association, containing: association connection state; socket list; destination transport address reachability states; current receiver window size; current 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.

COMMUNICATION UP notification: When a lost communication to an endpoint is restored or when SCTP becomes ready to send or receive user messages, this notification informs the application process about the affected association, the type of event that has occurred, the complete set of sockets of the peer, the maximum number of allowed streams and the inbound stream count (the number of streams the peer endpoint has requested).

DATA ARRIVE notification: When a message is ready to be retrieved via the Receive primitive, the application is informed by this notification.

SEND FAILURE notification / Receive Unsent Message / Receive Unacknowledged Message: When a message cannot be delivered via an association, the sender can be informed about it and learn whether the message has just not been acknowledged or (e.g. in case of lifetime expiry) if it has not even been sent.

NETWORK STATUS CHANGE notification: The NETWORK STATUS CHANGE notification informs the application about a socket becoming active/inactive.

COMMUNICATION LOST notification: When SCTP loses communication to an endpoint (e.g. via Heartbeats or excessive retransmission) or detects an abort, this notification informs the application process of the affected association and the type of event (failure OR termination in response to a shutdown or abort request).

SHUTDOWN COMPLETE notification: When SCTP completes the shutdown procedures, this notification is passed to the upper layer, informing it about the affected assocation.

3.2.1. Excluded Primitives

The 'Receive' primitive can return certain additional information, but this is optional to implement and therefore not considered. With a COMMUNICATION LOST notification, some more information may optionally be passed to the application (e.g., identification to retrieve unsent and unacknowledged data). SCTP "can invoke" a COMMUNICATION ERROR notification and "may send" a RESTART notification, making these two notifications optional to implement. The list provided under 'Status' includes "etc", indicating that more information could be provided. The primitive 'Get SRTT Report' returns information that is included in the information that 'Status' provides and is therefore not discussed. Similarly, 'Set Failure Threshold' sets only one out of various possible parameters included in 'Set Protocol Parameters'. The 'Destroy SCTP Instance' API function was excluded: it erases the SCTP instance that was created by 'Initialize', but is not a Primitive as defined in this document because it does not relate to a Transport Service Feature.

4. Pass 2

This pass categorizes the primitives from pass 1 based on whether they relate to a connection or to data transmission. Primitives are presented following the nomenclature: "CATEGORY.[SUBCATEGORY].PRIMITIVENAME.PROTOCOL". A connection is a general protocol-independent concept and refers to, e.g., TCP connections (identifiable by a unique pair of IP addresses and TCP port numbers) as well as SCTP associations (identifiable by multiple IP address and port number pairs).

Some minor details are omitted for the sake of generalization -- e.g., SCTP's 'close' [RFC4960] returns success or failure, whereas this is not described in the same way for TCP in [RFC0793], but this detail plays no significant role for the primitives provided by either TCP or SCTP.

The TCP 'send' and 'receive' primitives include usage of an "URGENT" mechanism. This mechanism is required to implement the "synch signal" used by telnet [RFC0854], but SHOULD NOT be used by new applications [RFC6093]. Because pass 2 is meant as a basis for the creation of TAPS systems, the "URGENT" mechanism is excluded. This also concerns the notification "Urgent pointer advance" in the

ERROR_REPORT described in Section 4.2.4.1 of [RFC1122].

4.1. CONNECTION Related Primitives

ESTABLISHMENT:
Active creation of a connection from one transport endpoint to one or more transport endpoints.

- CONNECT.TCP:
  Pass 1 primitive / event: 'open' (active) or 'open' (passive) with socket, followed by 'send'
  Parameters: 1 local IP address (optional); 1 destination transport address (for active open; else the socket and the local IP address of the succeeding incoming connection request will be maintained); timeout (optional); options (optional)
  Comments: If the local IP address is not provided, a default choice will automatically be made. The timeout can also be a retransmission count. The options are IP options to be used on all segments of the connection. At least the Source Route option is mandatory for TCP to provide.

- CONNECT.SCTP:
  Pass 1 primitive / event: 'initialize', followed by 'associate'
  Parameters: list of local SCTP port number / IP address pairs (initialize); 1 socket; outbound stream count
  Returns: socket list
  Comments: 'initialize' needs to be called only once per list of local SCTP port number / IP address pairs. One socket will automatically be chosen; it can later be changed in MAINTENANCE.

AVAILABILITY:
Preparing to receive incoming connection requests.

- LISTEN.TCP:
  Pass 1 primitive / event: 'open' (passive)
  Parameters: 1 local IP address (optional); 1 socket (optional); timeout (optional)
  Comments: if the socket and/or local IP address is provided, this waits for incoming connections from only and/or to only the provided address. Else this waits for incoming connections without this / these constraint(s). ESTABLISHMENT can later be performed with 'send'.

- LISTEN.SCTP:
  Pass 1 primitive / event: 'initialize', followed by 'COMMUNICATION UP' notification
  Parameters: list of local SCTP port number / IP address pairs

(initialize)
Returns: socket list; outbound stream count; inbound stream count
Comments: initialize needs to be called only once per list of local SCTP port number / IP address pairs. COMMUNICATION UP can also follow a COMMUNICATION LOST notification, indicating that the lost communication is restored.

MAINTENANCE:
Adjustments made to an open connection, or notifications about it. These are out-of-band messages to the protocol that can be issued at any time, at least after a connection has been established and before it has been terminated (with one exception: CHANGE-TIMEOUT.TCP can only be issued when DATA.SEND.TCP is called).

- CHANGE-TIMEOUT.TCP:
  Pass 1 primitive / event: 'send' combined with unspecified control of per-connection state variables
  Parameters: timeout value (optional); ADV_UTO (optional); boolean UTO_ENABLED (optional, default false); boolean CHANGEABLE (optional, default true)
  Comments: when sending data, an application can adjust the connection's timeout value (time after which the connection will be aborted if data could not be delivered). If UTO_ENABLED is true, the user timeout value (or, if provided, the value ADV_UTO) will be advertised for the TCP on the other side of the connection to adapt its own user timeout accordingly. UTO_ENABLED controls whether the UTO option is enabled for a connection. This applies to both sending and receiving. CHANGEABLE controls whether the user timeout may be changed based on a UTO option received from the other end of the connection; it becomes false when 'timeout value' is used.

- CHANGE-TIMEOUT.SCTP:
  Pass 1 primitive / event: 'Change HeartBeat' combined with 'Set Protocol Parameters'
  Parameters: 'Change HeartBeat': heartbeat frequency; 'Set Protocol Parameters': Association.Max.Retrans (whole association) or Path.Max.Retrans (per socket)
  Comments: Change Heartbeat can enable / disable heartbeats in SCTP as well as change their frequency. The parameter Association.Max.Retrans defines after how many unsuccessful heartbeats the connection will be terminated; thus these two primitives / parameters together can yield a similar behavior to CHANGE-TIMEOUT.TCP.

- DISABLE-NAGLE.TCP:
  Pass 1 primitive / event: not specified
  Parameters: one boolean value
  Comments: the Nagle algorithm delays data transmission to increase the chance to send a full-sized segment. An application must be able to disable this algorithm for a connection. This is related to the no-bundle flag in DATA.SEND.SCTP.

- REQUESTHEARTBEAT.SCTP:
  Pass 1 primitive / event: 'Request HeartBeat'
  Parameters: socket
  Returns: success or failure
  Comments: requests an immediate heartbeat on a path, returning success or failure.

- SETPROTOCOLPARAMETERS.SCTP:
  Pass 1 primitive / event: 'Set Protocol Parameters'
  Parameters: RTO.Initial; RTO.Min; RTO.Max; Max.Burst; RTO.Alpha; RTO.Beta; Valid.Cookie.Life; Association.Max.Retrans; Path.Max.Retrans; Max.Init.Retransmits; HB.interval; HB.Max.Burst

- SETPRIMARY.SCTP:
  Pass 1 primitive / event: 'Set Primary'
  Parameters: socket
  Returns: result of attempting this operation
  Comments: update the current primary address to be used, based on the set of available sockets of the association.

- ERROR.TCP:
  Pass 1 primitive / event: 'ERROR_REPORT'
  Returns: reason (encoding not specified); subreason (encoding not specified)
  Comments: soft errors that can be ignored without harm by many applications; an application should be able to disable these notifications. The reported conditions include at least: ICMP error message arrived; Excessive Retransmissions.

- STATUS.SCTP:
  Pass 1 primitive / event: 'Status' and 'NETWORK STATUS CHANGE' notification
  Returns: data block with information about a specified association, containing: association connection state; socket list; destination transport address reachability states; current receiver window size; current 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. The NETWORK STATUS CHANGE notification informs the application about a socket

becoming active/inactive.

- CHANGE-DSCP.TCP:
  Pass 1 primitive / event: not specified
  Parameters: DSCP value
  Comments: This allows an application to change the DSCP value.
  For TCP this was originally specified for the TOS field [RFC1122],
  which is here interpreted to refer to the DSField [RFC3260].

TERMINATION:
Gracefully or forcefully closing a connection, or being informed
about this event happening.

- CLOSE.TCP:
  Pass 1 primitive / event: 'close'
  Comments: this terminates the sending side of a connection after
  reliably delivering all remaining data.

- CLOSE.SCTP:
  Pass 1 primitive / event: 'Shutdown'
  Comments: this terminates a connection after reliably delivering
  all remaining data.

- ABORT.TCP:
  Pass 1 primitive / event: 'abort'
  Comments: this terminates a connection without delivering
  remaining data and sends an error message to the other side.

- ABORT.SCTP:
  Pass 1 primitive / event: 'abort'
  Parameters: abort reason to be given to the peer (optional)
  Comments: this terminates a connection without delivering
  remaining data and sends an error message to the other side.

- TIMEOUT.TCP:
  Pass 1 primitive / event: 'USER TIMEOUT' event
  Comments: the application is informed that the connection is
  aborted. This event is executed on expiration of the timeout set
  in CONNECTION.ESTABLISHMENT.CONNECT.TCP (possibly adjusted in
  CONNECTION.MAINTENANCE.CHANGE-TIMEOUT.TCP).

- TIMEOUT.SCTP:
  Pass 1 primitive / event: 'COMMUNICATION LOST' event
  Comments: the application is informed that the connection is
  aborted. this event is executed on expiration of the timeout that
  should be enabled by default (see beginning of section 8.3 in
  [RFC4960]) and was possibly adjusted in

CONNECTION.MAINTENANCE.CHANGE-TIMEOOUT.SCTP.

- ABORT-EVENT.TCP:
  Pass 1 primitive / event: not specified.

- ABORT-EVENT.SCTP:
  Pass 1 primitive / event: 'COMMUNICATION LOST' event
  Returns: abort reason from the peer (if available)
  Comments: the application is informed that the other side has aborted the connection using CONNECTION.TERMINATION.ABORT.SCTP.

- CLOSE-EVENT.TCP:
  Pass 1 primitive / event: not specified.

- CLOSE-EVENT.SCTP:
  Pass 1 primitive / event: 'SHUTDOWN COMPLETE' event
  Comments: the application is informed that CONNECTION.TERMINATION.CLOSE.SCTP was successfully completed.

4.2. DATA Transfer Related Primitives

All primitives in this section refer to an existing connection, i.e. a connection that was either established or made available for receiving data. In addition to the listed parameters, all sending primitives contain a reference to a data block and all receiving primitives contain a reference to available buffer space for the data.

- SEND.TCP:
  Pass 1 primitive / event: 'send'
  Parameters: timeout (optional)
  Comments: this gives TCP a data block for reliable transmission to the TCP on the other side of the connection. The timeout can be configured with this call whenever data are sent (see also CONNECTION.MAINTENANCE.CHANGE-TIMEOUT.TCP).

- SEND.SCTP:
  Pass 1 primitive / event: 'Send'
  Parameters: stream number; context (optional); life time (optional); socket (optional); unordered flag (optional); no-bundle flag (optional); payload protocol-id (optional)
  Comments: this gives SCTP a data block for reliable transmission to the SCTP on the other side of the connection (SCTP association). The 'stream number' denotes the stream to be used. The 'context' number can later be used to refer to the correct message when an error is reported. The 'life time' specifies a time after which this data block will not be sent. The 'socket'

can be used to state which path should be preferred, if there are multiple paths available (see also CONNECTION.MAINTENANCE.SETPRIMARY.SCTP). The data block can be delivered out-of-order if the 'unordered flag' is set. The 'no-bundle flag' can be set to indicate a preference to avoid bundling. The 'payload protocol-id' is a number that will, if provided, be handed over to the receiving application.

- RECEIVE.TCP:
  Pass 1 primitive / event: 'receive'.

- RECEIVE.SCTP:
  Pass 1 primitive / event: 'DATA ARRIVE' notification, followed by 'Receive'
  Parameters: stream number (optional)
  Returns: stream sequence number (optional), partial flag (optional)
  Comments: if the 'stream number' is provided, the call to receive only receives data on one particular stream. If a partial message arrives, this is indicated by the 'partial flag', and then the 'stream sequence number' must be provided such that an application can restore the correct order of data blocks that comprise an entire message.

- SENDFAILURE-EVENT.SCTP:
  Pass 1 primitive / event: 'SEND FAILURE' notification, optionally followed by 'Receive Unsent Message' or 'Receive Unacknowledged Message'
  Returns: cause code; context; unsent or unacknowledged message (optional)
  Comments: 'cause code' indicates the reason of the failure, and 'context' is the context number if such a number has been provided in DATA.SEND.SCTP, for later use with 'Receive Unsent Message' or 'Receive Unacknowledged Message', respectively. These primitives can be used to retrieve the complete unsent or unacknowledged message if desired.

5. Pass 3

This section presents the superset of all transport service features in all protocols that were discussed in the preceding sections, based on the list of primitives in pass 2 but also on text in pass 1 to include features that can be configured in one protocol and are static properties in another. Again, some minor details are omitted for the sake of generalization -- e.g., TCP may provide various different IP options, but only source route is mandatory to

implement, and this detail is not visible in the Pass 3 feature "Specify IP Options".

[AUTHOR'S NOTE: the list here looks pretty similar to the list in pass 2 for now. This will change as more protocols are added. For example, when we add UDP, we will find that UDP does not do congestion control, which is relevant to the application using it. This will have to be reflected in pass 1 and pass 2, only for UDP. In pass 3, we can then derive "no congestion control" as a transport service feature of UDP; however, since it would be strange to call the lack of congestion control a feature, the natural outcome is then to list "congestion control" as a feature of TCP and SCTP.]

5.1. CONNECTION Related Transport Service Features

ESTABLISHMENT:
Active creation of a connection from one transport endpoint to one or more transport endpoints.

- Specify IP Options
  Protocols: TCP

- Request multiple streams
  Protocols: SCTP

- Obtain multiple sockets
  Protocols: SCTP

AVAILABILITY:
Preparing to receive incoming connection requests.

- Listen, 1 specified local interface
  Protocols: TCP, SCTP

- Listen, N specified local interfaces
  Protocols: SCTP

- Listen, all local interfaces (unspecified)
  Protocols: TCP, SCTP

- Obtain requested number of streams
  Protocols: SCTP

MAINTENANCE:
Adjustments made to an open connection, or notifications about it.
NOTE: all features except "set primary path" in this category apply

to one out of multiple possible paths (identified via sockets) in SCTP, whereas TCP uses only one path (one socket).

- Change timeout for aborting connection (using retransmit limit or time value)
  Protocols: TCP, SCTP

- Control advertising timeout for aborting connection to remote endpoint
  Protocols: TCP

- Disable Nagle algorithm
  Protocols: TCP, SCTP
  Comments: This is not specified in [RFC4960] but in [RFC6458].

- Request an immediate heartbeat, returning success/failure
  Protocols: SCTP

- Set protocol parameters
  Protocols: SCTP
  SCTP parameters: RTO.Initial; RTO.Min; RTO.Max; Max.Burst; RTO.Alpha; RTO.Beta; Valid.Cookie.Life; Association.Max.Retrans; Path.Max.Retrans; Max.Init.Retransmits; HB.interval; HB.Max.Burst
  Comments: in future versions of this document, it might make sense to split out some of these parameters -- e.g., if a different protocol provides means to adjust the RTO calculation there could be a common feature for them called "adjust RTO calculation".

- Notification of Excessive Retransmissions (early warning below abortion threshold)
  Protocols: TCP

- Notification of ICMP error message arrival
  Protocols: TCP

- Status (query or notification)
  Protocols: SCTP
  SCTP parameters: association connection state; socket list; socket reachability states; current receiver window size; current 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; socket becoming active / inactive

- Set primary path
  Protocols: SCTP

- Change DSCP
  Protocols: TCP
  Comments: This is described to be changeable for SCTP too in [RFC6458].

TERMINATION:
Gracefully or forcefully closing a connection, or being informed about this event happening.

- Close after reliably delivering all remaining data, causing an event informing the application on the other side
  Protocols: TCP, SCTP
  Comments: A TCP endpoint locally only closes the connection for sending; it may still receive data afterwards.

- Abort without delivering remaining data, causing an event informing the application on the other side
  Protocols: TCP, SCTP
  Comments: In SCTP a reason can optionally be given by the application on the aborting side, which can then be received by the application on the other side.

- Timeout event when data could not be delivered for too long
  Protocols: TCP, SCTP
  Comments: the timeout is configured with CONNECTION.MAINTENANCE "Change timeout for aborting connection (using retransmit limit or time value)".

5.2. DATA Transfer Related Transport Service Features

All features in this section refer to an existing connection, i.e. a connection that was either established or made available for receiving data. Reliable data transfer entails delay -- e.g. for the sender to wait until it can transmit data, or due to retransmission in case of packet loss.

5.2.1. Sending Data

All features in this section are provided by DATA.SEND from pass 2. DATA.SEND is given a data block from the application, which we here call a "message".

- Reliably transfer data
  Protocols: TCP, SCTP

- Notifying the receiver to promptly hand over data to application
  Protocols: TCP
  Comments: This seems unnecessary in SCTP, where data arrival causes an event for the application.

- Message identification
  Protocols: SCTP

- Choice of stream
  Protocols: SCTP

- Choice of path (destination address)
  Protocols: SCTP

- Message lifetime
  Protocols: SCTP

- Choice between unordered (potentially faster) or ordered delivery
  Protocols: SCTP

- Request not to bundle messages
  Protocols: SCTP

- Specifying a "payload protocol-id" (handed over as such by the receiver)
  Protocols: SCTP

5.2.2. Receiving Data

All features in this section are provided by DATA.RECEIVE from pass 2. DATA.RECEIVE fills a buffer provided to the application, with what we here call a "message".

- Receive data
  Protocols: TCP, SCTP

- Choice of stream to receive from
  Protocols: SCTP

- Message identification
  Protocols: SCTP
  Comments: In SCTP, this is optionally achieved with a "stream sequence number". The stream sequence number is always provided in case of partial message arrival.

- Information about partial message arrival
  Protocols: SCTP
  Comments: In SCTP, partial messages are combined with a stream sequence number so that the application can restore the correct order of data blocks an entire message consists of.

5.2.3. Errors

This section describes sending failures that are associated with a specific call to DATA.SEND from pass 2.

- Notification of unsent messages
  Protocols: SCTP

- Notification of unacknowledged messages
  Protocols: SCTP

6. Acknowledgements

The authors would like to thank (in alphabetical order) Bob Briscoe, David Hayes, Gorry Fairhurst, Karen Nielsen and Joe Touch for providing valuable feedback on this document. This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 644334 (NEAT). The views expressed are solely those of the author(s).

7. IANA Considerations

XX RFC ED - PLEASE REMOVE THIS SECTION XXX

This memo includes no request to IANA.

8. Security Considerations

Security will be considered in future versions of this document.

9. References

9.1. Normative References

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981,

<http://www.rfc-editor.org/info/rfc793>.

[RFC1122] Braden, R., Ed., "Requirements for Internet Hosts - Communication Layers", STD 3, RFC 1122, DOI 10.17487/RFC1122, October 1989, <http://www.rfc-editor.org/info/rfc1122>.

[RFC4960] Stewart, R., Ed., "Stream Control Transmission Protocol", RFC 4960, DOI 10.17487/RFC4960, September 2007, <http://www.rfc-editor.org/info/rfc4960>.

[RFC5482] Eggert, L. and F. Gont, "TCP User Timeout Option", RFC 5482, DOI 10.17487/RFC5482, March 2009, <http://www.rfc-editor.org/info/rfc5482>.

9.2. Informative References

[FA15] Fairhurst, Ed., G., Trammell, Ed., B., and M. Kuehlewind, Ed., "Services provided by IETF transport protocols and congestion control mechanisms", draft-fairhurst-taps-transports-08.txt (work in progress), December 2015.

[RFC0854] Postel, J. and J. Reynolds, "Telnet Protocol Specification", STD 8, RFC 854, DOI 10.17487/RFC0854, May 1983, <http://www.rfc-editor.org/info/rfc854>.

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

[RFC3168] Ramakrishnan, K., Floyd, S., and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP", RFC 3168, DOI 10.17487/RFC3168, September 2001, <http://www.rfc-editor.org/info/rfc3168>.

[RFC3260] Grossman, D., "New Terminology and Clarifications for Diffserv", RFC 3260, DOI 10.17487/RFC3260, April 2002, <http://www.rfc-editor.org/info/rfc3260>.

[RFC3828] Larzon, L-A., Degermark, M., Pink, S., Jonsson, L-E., Ed., and G. Fairhurst, Ed., "The Lightweight User Datagram Protocol (UDP-Lite)", RFC 3828, DOI 10.17487/RFC3828, July 2004, <http://www.rfc-editor.org/info/rfc3828>.

[RFC5461] Gont, F., "TCP's Reaction to Soft Errors", RFC 5461, DOI 10.17487/RFC5461, February 2009,

<http://www.rfc-editor.org/info/rfc5461>.

[RFC6093] Gont, F. and A. Yourtchenko, "On the Implementation of the TCP Urgent Mechanism", RFC 6093, DOI 10.17487/RFC6093, January 2011, <http://www.rfc-editor.org/info/rfc6093>.

[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, <http://www.rfc-editor.org/info/rfc6458>.

[RFC7414] Duke, M., Braden, R., Eddy, W., Blanton, E., and A. Zimmermann, "A Roadmap for Transmission Control Protocol (TCP) Specification Documents", RFC 7414, DOI 10.17487/RFC7414, February 2015, <http://www.rfc-editor.org/info/rfc7414>.

Appendix A. Overview of RFCs used as input for pass 1

TCP: [RFC0793], [RFC1122], [RFC5482]
SCTP: [RFC4960], planned: [RFC6458]

Appendix B. How to contribute

This document is only concerned with transport service features that are explicitly exposed to applications via primitives. It also strictly follows RFC text: if a feature is truly relevant for an application, the RFCs better say so and in some way describe how to use and configure it. Thus, the approach to follow for contributing to this document is to identify the right RFCs, then analyze and process their text.

Experimental RFCs are excluded, and so are primitives that MAY be implemented (by the transport protocol). To be included, the minimum requirement level for a primitive to be implemented by a protocol is SHOULD. If [RFC2119]-style requirements levels are not used, primitives should be excluded when they are described in conjunction with statements like, e.g.: "some implementations also provide" or "an implementation may also". Briefly describe excluded primitives in a subsection called "excluded primitives".

Pass 1: Identify text that talks about primitives. An API specification, abstract or not, obviously describes primitives -- but note that we are not *only* interested in API specifications. The text describing the 'send' primitive in the API specified in

[RFC0793], for instance, does not say that data transfer is reliable. TCP's reliability is clear, however, from this text in Section 1 of [RFC0793]: "The Transmission Control Protocol (TCP) is intended for use as a highly reliable host-to-host protocol between hosts in packet-switched computer communication networks, and in interconnected systems of such networks."

For the new pass 1 subsection about the protocol you're describing, it is recommendable to begin by copy+pasting all the relevant text parts from the relevant RFCs, then adjust terminology to match the terminology in Section 1 and adjust (shorten!) phrasing to match the general style of the document. Try to formulate everything as a primitive description to make the primitive description as complete as possible (e.g., the "SEND.TCP" primitive in pass 2 is explicitly described as reliably transferring data); if there is text that is relevant for the primitives presented in this pass but still does not fit directly under any primitive, use it as an introduction for your subsection. However, do note that document length is a concern and all the protocols and their services / features are already described in [FA15].

Pass 2: The main goal of this pass is unification of primitives. As input, use your own text from Pass 1, no exterior sources. If you find that something is missing there, fix the text in Pass 1. The list in pass 2 is not done by protocol ("first protocol X, here are all the primitives; then protocol Y, here are all the primitives, ..") but by primitive ("primitive A, implemented this way in protocol X, this way in protocol Y, ..."). We want as many similar pass 2 primitives as possible. This can be achieved, for instance, by not always maintaining a 1:1 mapping between pass 1 and pass 2 primitives, renaming primitives etc. Please consider the primitives that are already there and try to make the ones of the protocol you are describing as much in line with the already existing ones as possible. In other words, we would rather have a primitive with new parameters than a new primitive that allows to send in a particular way.

Please make primitives fit within the already existing categories and subcategories. For each primitive, please follow the style:

- PRIMITIVENAME.PROTOCOL:
  Pass 1 primitive / event:
  Parameters:
  Returns:
  Comments:

The entries "Parameters", "Returns" and "Comments" may be skipped if a primitive has no parameters, no described return value or no

comments seem necessary, respectively. Optional parameters must be followed by "(optional)". If a default value is known, provide it too.

Pass 3: the main point of this pass is to identify features that are the result of static properties of protocols, for which all protocols have to be listed together; this is then the final list of all available features. For this, we need a list of features per category (similar categories as in pass 2) along with the protocol supporting it. This should be primarily based on text from pass 2 as input, but text from pass 1 can also be used. Do not use external sources.

Appendix C. Revision information

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

-00 (from draft-welzl-taps-transports): this now covers TCP based on all TCP RFCs (this means: if you know of something in any TCP RFC that you think should be addressed, please speak up!) as well as SCTP, exclusively based on [RFC4960]. We decided to also incorporate [RFC6458] for SCTP, but this hasn't happened yet. Terminology made in line with [FA15]. Addressed comments by Karen Nielsen and Gorry Fairhurst; various other fixes. Appendices (TCP overview and how-to-contribute) added.

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

Michael Tuexen
Muenster University of Applied Sciences
Stegerwaldstrasse 39
Steinfurt 48565
Germany

Email: tuexen@fh-muenster.de

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

Email: naeemk@ifi.uio.no

Disclaimer