neat

NEAT

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

H2020-ICT-05-2014
Project number: 644334

Deliverable D2.2

Core Transport System, with both Low-level and High-level Components

Editor(s): Naeem Khademi
Contributor(s): Zdravko Bozakov, Anna Brunstrom, Øystein Dale, Dragana Damjanovic, Kristian Riktor Evensen, Gorry Fairhurst, Karl-Johan Grinnemo, Tom Jones, Simone Mangiante, Andreas Petlund, David Ros, Daniel Stenberg, Michael Tüxen, Felix Weinrank, Michael Welzl

Work Package: 2 / Core Transport System
Revision: 1.0
Date: March 14, 2017
Deliverable type: R (Report)
Dissemination level: Public

Abstract

This document presents the core transport system in NEAT, as used for development of the reference implementation of the NEAT System. The document describes the components necessary to realise the basic Transport Services provided by the NEAT User API, with the description of a set of NEAT building blocks and their related design choices. The design of this core transport system takes into consideration the Transport Services and the API (defined in Task 1.3) and in close coordination with the overall architecture (Task 1.2).

To realise the Transport Services provided by the API, a set of transport functionalities has to be provided by the NEAT Core Transport System. These functionalities take the form of several building blocks, or *NEAT Components*, each representing an associated implementation activity. Some of the components are needed to ensure the basic operation of the NEAT System—e.g., a *NEAT Flow Endpoint*, a callback-based *NEAT API Framework*, the *NEAT Logic* and the functionality to *Connect to a name*. Additional components are needed for: (a) ensuring connectivity, by means of mechanisms for discovery of path support for different protocols; (b) supporting end-to-end security; (c) the ability to apply different policies to influence the decision-making process of the transport system; (d) providing other important functionalities (e.g., a user-space SCTP stack, or gathering statistics for users or system administrators).

This document updates Deliverable D2.1; in particular, the descriptions of NEAT components presented here correspond to the implementation status at the time of writing, and as such they replace those in D2.1.

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

Contents

List of Abbreviations **6**

1 Introduction **10**
1.1 Overview of the NEAT Architecture 10
1.2 Overview of the services provided by the NEAT User API 12
1.3 Overview of the components required to provide the services 12
1.4 Overview of component interaction during connection setup 14

2 Coding with the NEAT User API **16**
2.1 NEAT User API Tutorial 17
2.1.1 What is NEAT? 17
2.1.2 Contexts and Flows 17
2.1.3 Properties 18
2.1.4 Asynchronous API 18
2.1.5 A minimal server 19
2.1.6 A minimal client 22
2.1.7 Tying the client and server together 24
2.2 Summary of the benefits of coding with the NEAT User API 25

3 Core Transport Functions **28**
3.1 NEAT Framework Components 29
3.1.1 NEAT Flow Endpoint 29
3.1.2 NEAT API Framework (callback) 34
3.1.3 NEAT Logic 39
3.1.4 Connect to a name 41
3.1.5 NEAT Flow Endpoint Statistics 42
3.2 NEAT Transport Components 45
3.2.1 NEAT-integrated SCTP user-space stack 46
3.2.2 Middlebox Traversal 48
3.2.3 Local flow scheduling priority 48
3.2.4 Security 51
3.3 NEAT Selection Components 54
3.3.1 Happy Eyeballs 54
3.3.2 Happy Apps (application-level feedback mechanisms) 56
3.4 NEAT Policy Components 59
3.4.1 NEAT Policy Manager 60
3.4.2 Policy Information Base (PIB) 64
3.4.3 Characteristics Information Base (CIB) 66

4 NEAT reference material **71**
4.1 NEAT tutorial 71
4.2 Additional online documentation 71
4.3 Virtual machines 71

5 Conclusions **72**

References **74**

A NEAT Terminology **75**

B NEAT API Reference **77**

B.1 Optional arguments . . . 77
B.1.1 Specifying no optional arguments . . . 77
B.1.2 Optional argument macros . . . 77
B.1.3 Optional argument tags . . . 78
B.2 Properties . . . 79
B.2.1 Application property reference . . . 79
B.2.2 Inferred properties . . . 80
B.3 Callbacks . . . 81
B.3.1 Example callback flow . . . 82
B.3.2 Callback reference . . . 82
B.4 Error codes . . . 83
B.5 API functions . . . 83
B.5.1 `neat_init_ctx` . . . 83
B.5.2 `neat_free_ctx` . . . 84
B.5.3 `neat_new_flow` . . . 84
B.5.4 `neat_set_property` . . . 84
B.5.5 `neat_get_property` . . . 85
B.5.6 `neat_open` . . . 86
B.5.7 `neat_accept` . . . 87
B.5.8 `neat_read` . . . 88
B.5.9 `neat_write` . . . 88
B.5.10 `neat_shutdown` . . . 89
B.5.11 `neat_close` . . . 90
B.5.12 `neat_abort` . . . 90
B.5.13 `neat_set_operations` . . . 91
B.5.14 `neat_change_timeout` . . . 92
B.5.15 `neat_set_primary_dest` . . . 92
B.5.16 `neat_secure_identity` . . . 93
B.5.17 `neat_set_checksum_coverage` . . . 93
B.5.18 `neat_set_qos` . . . 94
B.5.19 `neat_set_ecn` . . . 94
B.5.20 `neat_start_event_loop` . . . 95
B.5.21 `neat_stop_event_loop` . . . 95
B.5.22 `neat_get_backend_fd` . . . 96
B.5.23 `neat_get_backend_timeout` . . . 96
B.5.24 `neat_get_stats` . . . 96
B.5.25 `neat_log_level` . . . 97
B.5.26 `neat_log_file` . . . 97

C Paper: *NEAT: A Platform- and Protocol-Independent Internet Transport API* **98**

D Paper: *On the Cost of Using Happy Eyeballs for Transport Protocol Selection* **108**

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

1 Introduction

The NEAT System aims to change the transport layer interface in a way that Internet applications may specify and select a variety of Transport Services, instead of specifying a transport protocol. The Transport Services to be provided by the NEAT System and the API needed to achieve the above goal are outlined in Deliverable D1.2 [26]. The NEAT *Core Transport System* plays a vital role in translating those Transport Services exposed by the NEAT User API into protocol-level function calls, as well as in supporting a variety of transport-layer mechanisms that provide such Transport Services[1]. We denote as Core Transport System the set of building blocks necessary to provide NEAT Transport Services described in Deliverable D1.2. These include mechanisms for ensuring end-to-end connectivity, discovery of path support for protocol(s) chosen by the NEAT System, end-to-end security, ability to select different system policies depending on the application or network scenarios, and the ability to expose connection-level or system-wide statistics to an application. It also includes the support for necessary transport protocols in user-space (e.g., SCTP support by usrsctp), and essential functionalities for using multi-streaming and multi-homing.

This document is an update to D2.1 and reports on the core transport system actually being implemented in NEAT, which we term the *NEAT Library*. The rest of this section provides a short overview of the NEAT architecture introduced in Deliverable D1.1 [14] (§ 1.1). It also discusses the Transport Services provided by NEAT (§ 1.2). Then, it briefly presents the building blocks, or *NEAT Components*, that comprise the NEAT core transport system (§ 1.3) and how they interact during a connection setup attempt (§ 1.4).

Section 2 describes how applications can be built to use the NEAT User API. The section provides a tutorial on how to use the NEAT User API along with a simple client/server example using the NEAT library, and summarises some benefits of NEAT for application developers when compared with the traditional socket API. Section 3 discusses each of the NEAT components in detail, identifies the Transport Services they provide as well as their dependency on other components in the NEAT System, and provides examples of their operation. Section 4 provides references to related documentation and code examples. Then, Section 5 draws conclusions from this document and points at the final work to be done in Work Package 2.

To close the document, Appendix B provides a detailed reference of the NEAT User API. The paper attached in Appendix C [20] summarises some of the topics dealt with in this document, in particular the benefits offered by NEAT in different contexts, and also discusses the relation between NEAT and Internet standards activities. Finally, Appendix D contains a paper [22] with our performance evaluation results of the Happy Eyeballs mechanism used by NEAT (§ 3.3.1).

1.1 Overview of the NEAT Architecture

The NEAT architecture has been described in detail in Deliverable D1.1. In this section we provide a summary of such description in order to put our implementation work into context. The NEAT System is a layered architecture that provides a flexible and evolvable transport system. The applications and middleware served by the NEAT System utilise a new NEAT User API that abstracts network transport. The NEAT System can provide Transport Services in a way that allows the best transport protocol to be used by an application without the application having to handle selection from application code.

The main part of the NEAT System is the NEAT User Module, depicted in Figure 1. It provides a

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

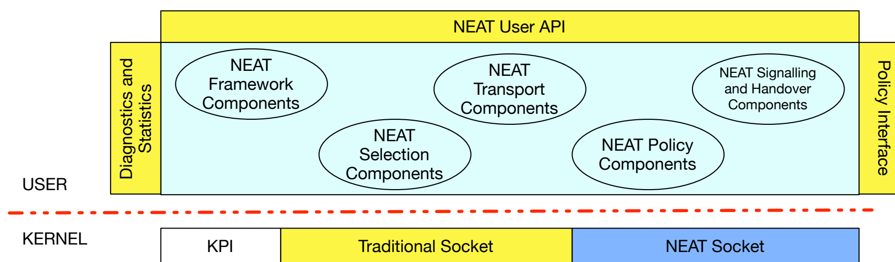


Figure 1: Component groups and interfaces used to realise the NEAT User Module.

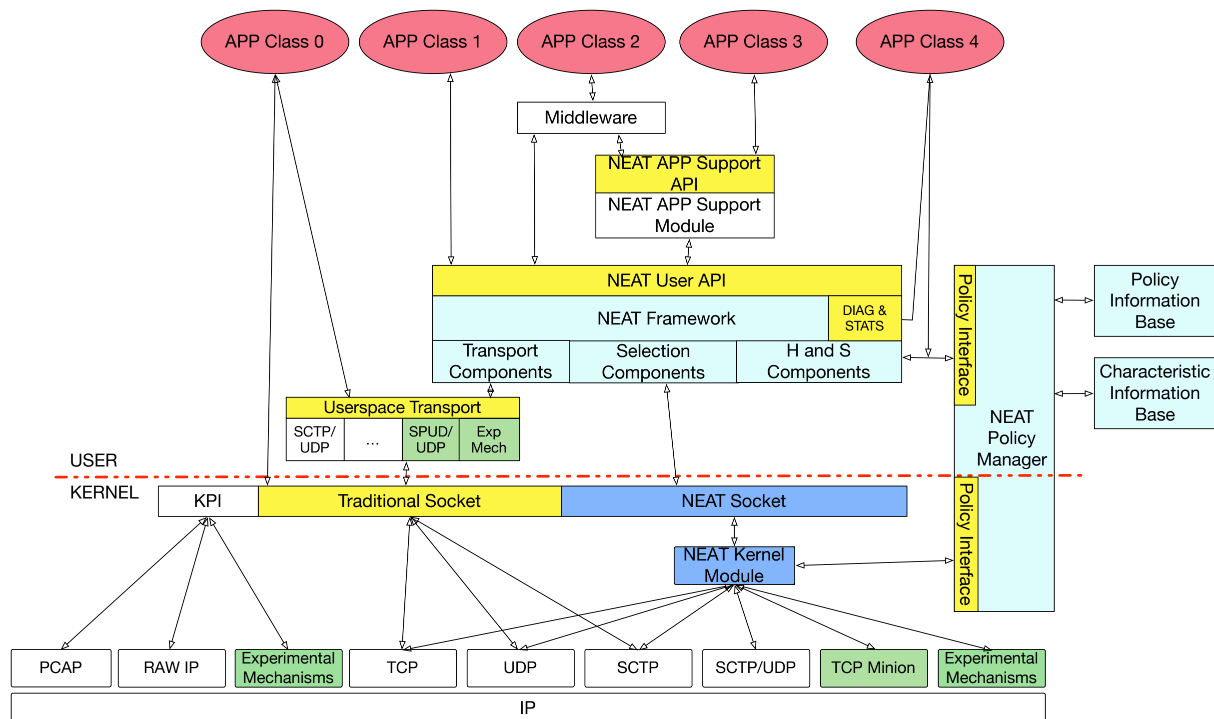


Figure 2: Components and interfaces to the NEAT System. The NEAT User Module is composed of all the blocks shown in light blue (NEAT Framework, NEAT Transport, NEAT Selection, NEAT Signalling and Handover, and Policy Components) and related APIs (NEAT User API, Policy Interface, Diagnostics and Statistics Interface).

set of components necessary to realise a Transport Service provided by the NEAT System. It is implemented in user-space and is intended to be portable across a wide range of platforms.

Figure 2 provides a more detailed overview of the different parts of the NEAT System and its interfaces. Applications access the NEAT System via a NEAT User API and its associated interfaces. The NEAT User API 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 System to select an appropriate Transport Service.

The NEAT User API provides the interface to the NEAT User Module. This API and its associated Diagnostics and Statistics Interface are formally one part of a group of components that comprise the NEAT Framework. Other components in this group are responsible for the most basic functions of the NEAT User Module.

A group of components are responsible for the Selection of the Transport Service, these use the services of the NEAT Policy Manager, which provides high-level components that inform selection

and enforce policy for decisions. The policy information is combined with the information passed via the NEAT User API and can be updated in this architecture by probing/signalling mechanisms to complete selection of the protocols and mechanisms needed to realise the required Transport Service.

The components required to configure and manage the Transport Service are also part of the NEAT User Module. Some protocols (such as TCP and UDP) are typically provided by the kernel of the platform OS. Other transport protocols are provided in user-space, but may optionally also be provided by the kernel. A key goal of the NEAT System is to offer Transport Services in the same way regardless of how the transport protocols have been implemented or how they are offered by the network stack. The NEAT User Module can utilise optional signalling components, implemented in the NEAT Signalling and Handover components.

The NEAT System can evolve to incorporate new and experimental transports. It allows applications to take advantage of new functionality as it becomes available across the Internet and will fall back and emulate features required by applications when other alternatives are not available.

The Kernel interfaces and experimental mechanisms, highlighted in Figure 2 in dark blue and green respectively, are optional components of the NEAT System.

The layered design of the NEAT System enables it to offer optimised transports to applications that would normally have to supply compatibility layers or the entire transport as a library.

1.2 Overview of the services provided by the NEAT User API

Internet drafts from the IETF TAPS working group [13, 25], co-authored by NEAT participants, define a Transport Service as an end-to-end service provided to an application by the transport layer, and a Transport Service Feature as a specific end-to-end feature that a Transport Service provides to its users.

Deliverable D1.2 [26] presents two sets of Transport Service Features: (a) Features derived from draft-ietf-taps-transports-usage [25]; and (b) Features derived from use cases in D1.1 [14]. Set (a) includes Transport Service Features that can be utilised using primitives and events derived from transport-protocol APIs. This includes TCP, MPTCP, SCTP, UDP and UDP-Lite protocols. Set (b) includes Transport Service Features that stem from application requirements of the use cases in D1.1 and are composed of two groups: (1) Features that are associated with the information passed from the NEAT System to the application; and (2) Features that are associated with the information passed from the application to the NEAT System.

The core transport system in NEAT is the minimal set of components necessary to implement these Transport Service Features. We summarise them next and provide a more elaborate description in Section 3.

1.3 Overview of the components required to provide the services

To offer the Transport Service Features presented in D1.2 [26], a set of components are needed for the NEAT core transport system. The core building blocks from D2.1 are updated in this document, and their final version will be provided in D2.3.

Based on the sets of NEAT Components defined in D1.1, the NEAT core components are categorised into:

- **NEAT Framework components:** a set of components that provide the most basic functionality required to run a NEAT System. These include the following building blocks: a *NEAT Flow End-*

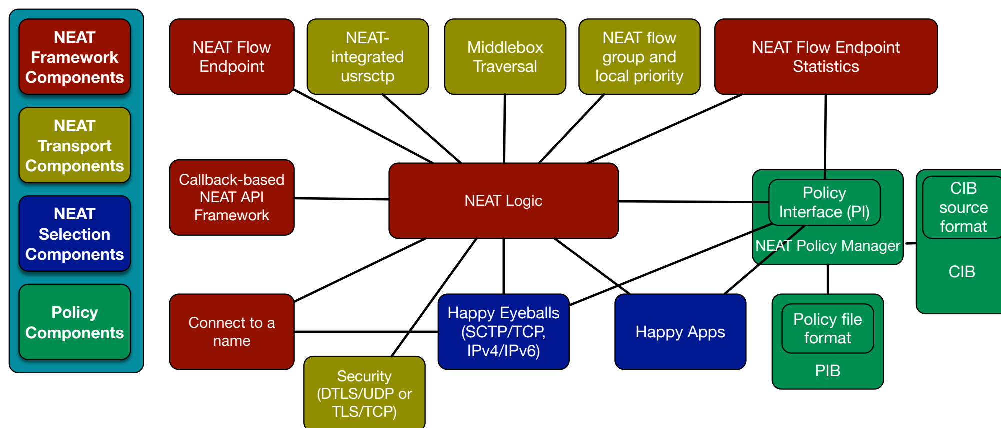


Figure 3: Building blocks of the NEAT core transport system. Each colour denotes a different component grouping.

point, a callback-based *NEAT API Framework*, *NEAT Logic*, the ability to *Connect to a name* and *NEAT Flow Endpoint Statistics*.

- **NEAT Transport components:** a set of components responsible for providing the functions to configure and manage the NEAT Transport Service for a particular NEAT Flow. These include the possibility to use DTLS over UDP (and DTLS over SCTP in the future) as well as TLS over TCP when *Security* is being requested. In addition, the *NEAT-integrated usrsctp* component provides support for the SCTP protocol in user-land in NEAT. Moreover, *Local flow scheduling priority* provides a framework to use priorities among different NEAT Flows in the NEAT send scheduler.

- **NEAT Selection components:** these components are responsible for selecting an appropriate transport endpoint and a set of protocols/mechanisms. These include building blocks for path support discovery using *Happy Eyeballs* mechanisms at the transport layer. Happy Eyeballs can be done between different transport protocols (e.g., SCTP/TCP) or IP versions (IPv4/IPv6). NEAT also provides similar functionality using a feedback mechanism at the application layer named *Happy Apps*.

- **NEAT Policy components:** a set of components providing the possibility to manage and apply different policies. These building blocks include: a *NEAT Policy Manager* (PM), a *Policy Information Base* (PIB), a *Characteristics Information Base* (CIB) and one or multiple *CIB sources*. The NEAT Policy Manager provides a Policy Interface (PI) to communicate with the PIB and CIB(s) and maintains policies defined by the application developer or system developer using a predefined file format.

Figure 3 illustrates the set of core system components and their potential dependency on each other. As seen on the list above, such components are found in four out of the five component groupings in Figure 1 (Framework, Transport, Selection, and Policy); NEAT Signalling and Handover components are part of the Extended Transport System under development in Work Package 3, and therefore out of the scope of this document.

1.4 Overview of component interaction during connection setup

This section presents an overview of NEAT operations from the moment a connection setup attempt is made at the API level until the connection handle is returned to the user. This process has been depicted in Figure 4, a simplified workflow showing how NEAT components interact when opening a flow. Further details on the workings of each component will be provided in Section 3.

A connection is set up in NEAT as shown in Figure 4, as follows (numbers in the arrows correspond to the step numbers below):

1. The application specifies properties of the communication with `neat_set_property` and the application calls `neat_open` in the NEAT User API.

2. The NEAT Logic sends application properties and inferred properties to the Policy Manager using a JSON object in order to query for feasible transport candidates, based on the destination domain name. This is done for pre-filtering purposes, to avoid running DNS lookups for non-feasible candidates since this may increase setup latency.

3. The Policy Manager finds available transport candidates based on the available policies in PIB and cached information in CIB.

4. The Policy Manager replies to NEAT Logic's query with an initial set of transport candidates eligible for name resolution (e.g., DNS lookup).

5. The NEAT Logic initiates one or more name resolution requests to the Name Resolver and the Name Resolver replies with resolved addresses, then the NEAT Logic inserts these into each candidate.

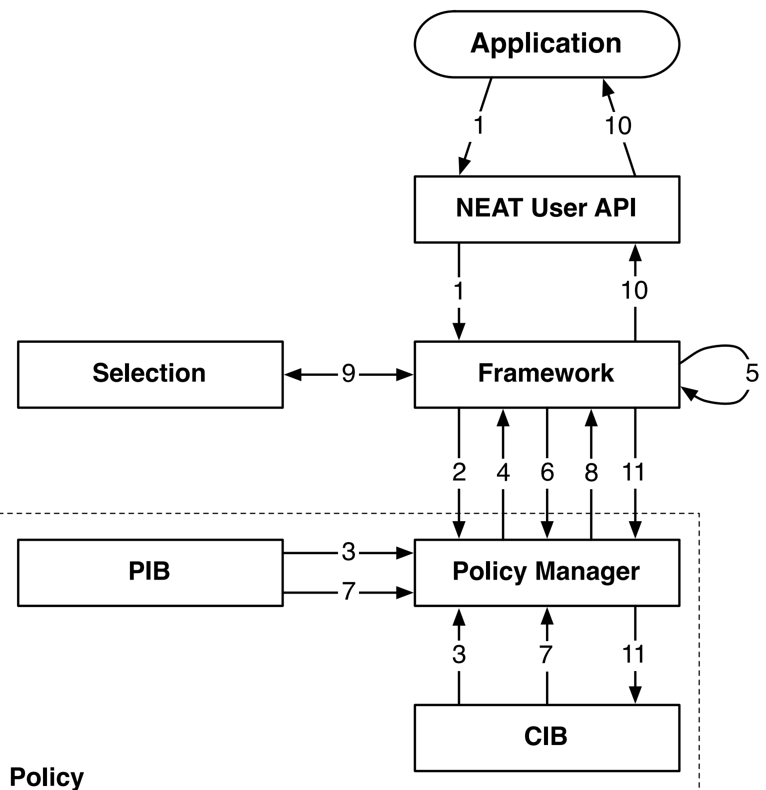
6. The NEAT Logic makes a second call to the Policy Manager, asking it to build candidates for attempting flow establishment based on the outcome of name resolution.

7. The Policy Manager builds candidates, with priorities given by policy and available CIB information. A candidate consists of the following:

   - Transport protocol
   - Interface
   - Port
   - Local address
   - Remote address
   - Priority
   - Application properties

8. The Policy Manager returns the list of suitable candidates that the NEAT Logic should use to establish a flow. If one or more of the application properties are specified as desired (i.e., precedence 1), multiple candidates may be generated with different settings for that property.

9. The NEAT Logic generates a list of Happy Eyeballs candidates and initiates the Happy Eyeballs algorithm. The Happy Eyeballs module tries to connect each candidate. The delay between each



1. Request to open flow & pass application requirements
2. Query PM about feasible transport candidates based on destination domain name
3. PM determines available transport candidates that fulfil policy (PIB) and cached information (CIB)
4. Return ranked list of feasible transport candidates as pre-filter for address resolution
5. Resolve addresses
6. Query PM about feasible transport candidates for resolved destination address
7. PM builds candidates, assigning priorities based on PIB/CIB matches
8. Return ranked list of feasible transport candidates for flow establishment
9. Do Happy Eyeballs with candidates, according to specified priorities
10. Return handle to selected transport solution
11. Cache results from Happy Eyeballs in the CIB

Figure 4: Simplified workflow showing how NEAT components interact when opening a flow.

candidate is determined from the priority of the candidate. A lower value implies a higher priority. The connection may be handled by either the operating system using its own implementation of the protocol, or using a user-space implementation of the protocol. A handle to the first connection that connects successfully and meets all required properties set by the application is returned to the NEAT Logic along with the outcome of Happy Eyeballs results for other transport candidates.

10. The NEAT Logic starts polling the socket internally, and reports back to the application that a connection has been established using the `on_connected` callback if it has been specified using `neat_set_operations`.

11. The NEAT Logic makes a third call to the Policy Manager, asking for the results from Happy Eyeballs to be cached in the CIB.

Now that the NEAT Flow is open for use, NEAT Logic will report that the flow is readable or writable if the respective `on_readable` or `on_writable` callbacks have been specified with `neat_set_operations`. The application can close the flow with `neat_close`.

The following provides a step-by-step example of how the mapping of application properties to the transport candidates is done by the Policy Manager internally:

a. A NEAT application requests the `reliable_transport` and `low_latency` properties (step #1 in Figure 4). In addition, for the sake of simplicity let us assume the application has provided a `remote_IP` property, and thus no DNS resolution is necessary (i.e., steps #2–5 are not needed).

b. The PM is asked to check the profiles and policies available in the PIB (step #6 in Figure 4).

    i) A profile matching `reliable_transport` generates candidates for TCP and SCTP connections. The SCTP candidate is assigned a higher score than TCP.

    ii) A low latency profile adds the socket options `TCP_NODELAY:1` and `SCTP_NODELAY:1`, respectively, to the candidates.

    iii) The low latency policy further dictates that only wired interfaces should be used for the connection.

c. From the CIB repository the PM infers that, for the requested destination, SCTP is only supported when going through interface `eth1` (step #7 in Figure 4).

d. The PM returns the following ranked candidates to HE (step #8 in Figure 4):

    i) `Transport: SCTP, SCTP_nodelay:1, interface:eth1`

    ii) `Transport: TCP, TCP_nodelay:1, interface:eth0`

    iii) `Transport: TCP, TCP_nodelay:1, interface:eth1`

2 Coding with the NEAT User API

Network application programmers need to understand how to program against the NEAT User API, and the way in which programs can take advantage of the services offered by a NEAT System.

Before describing the core transport functions, this section provides a tutorial that describes how an application uses the NEAT User API. It introduces the new API, which is simpler and more flexible

than the existing sockets API. Key concepts are then explained, such as the role of NEAT contexts, NEAT flows and flow properties. A minimal client and server example is finally presented to illustrate how network programming tasks can be eased using this API. In the example, snippets of code illustrate the main concepts of the API. This tutorial is publicly available online[2] at: http://neat.readthedocs.io/en/latest/tutorial.html.

Finally, this section presents an example that summarises the benefits of using the asynchronous and non-blocking NEAT User API when a developer writes an application against it.

2.1 NEAT User API Tutorial

2.1.1 What is NEAT?

NEAT is a library for networked applications, intended to replace existing socket APIs with a simpler, more flexible API. Additionally, NEAT enables endpoints to make better decisions as to how to utilize the available network resources and adapts based on the current condition of the network.

With NEAT, applications are able to specify the service they want from the transport layer. NEAT will determine which of the available protocols fit the requirements of the application and tune all the relevant parameters to ensure that the application gets the desired service from the transport layer, based on knowledge about the current state of the network when this information is available.

NEAT enables applications to be written in a protocol-agnostic way, thus allowing applications to be future-proof, leveraging new protocols as they become available, with minimal to no change. Further, NEAT will try to connect with different protocols if possible, making it able to gracefully fall back to another protocol if it turns out that the most optimal protocol is unavailable, for example, because of a middlebox such as a firewall. A connection in the NEAT API will only fail if all protocols satisfying the requirements of the application are unable to connect, or if no available protocol can satisfy the requirements of the application.

Most operating systems support the same protocols. However, the same protocol may often have a slightly different API on different operating systems. NEAT provides the same API on all supported operating systems, which is currently Linux, FreeBSD, OpenBSD, NetBSD, and OS X. The availability of a protocol depends on whether the protocol is supported by the OS or if NEAT is compiled with support for a user-space stack that implements the protocol.

2.1.2 Contexts and Flows

The most important concept in the NEAT API is that of the *flow*. A flow is similar to a socket in the traditional Berkely Socket API. It is a bidirectional link used to communicate between two applications, on which data may be written to or read from. Further, just like a socket, a flow uses some transport layer protocol to communicate.

However, one important difference is that a flow is not as strictly tied to the underlying transport protocol in the same way a socket is. In fact, a flow may be created without even specifying which transport protocol to use. This is not possible with a socket.

The same applies to modifying options on sockets. Setting the same kind of option on two sockets with different protocols in the traditional socket API requires `setsockopt` calls with different protocol IDs, option names, and sometimes even values with different units. The `setsockopt` calls also vary depending on what system you are on. This is not the case with NEAT. As long as the desired option is

[2] The version included here has been edited slightly.

available for the protocol in use, the API for setting that option is the same for all protocols, and on all operating systems supported by NEAT.

A *context* is a common environment for multiple flows. Along with flows, it contains several services that are used by the flows internally in NEAT, such as a DNS resolver and a Happy Eyeballs implementation. Flows within a context are polled together. A flow may only belong to the context in which it is created, and it cannot be transferred to a different context. Most applications need only one context.

2.1.3 Properties

Different types of applications have different requirements and desires to the services provided by the transport layer. An application for real-time communication may require the communication to have properties such as low latency, high bandwidth, quality of service, and have less strict requirements with regards to reliable delivery. Losing a packet or bit errors may be less critical to these applications. A web browser, on the other hand, might require communication that is (partially) ordered and error-free. A BitTorrent application might only require the ability to send packets to some destination with a minimum amount of effort, and not at the expense of other applications with stricter bandwidth requirements.

With the traditional socket API, the application requirements dictate the choice of which protocol to use. With NEAT, this is not the case. NEAT enables applications to specify the properties[3] of the communication instead of specifying which protocol to use. Some properties may be required; other properties may be desired, but not mandatory.

Based on the properties, NEAT will determine which protocols can support the requirements of the application and the options to set for each protocol. This mapping of application properties to a list of potential transport candidates is done with the help of the Policy Manager (PM) (§ 3.4.1).

Consequently, NEAT System will try to establish a connection by trying each of the transport candidates until one connection succeeds, a method known as Happy Eyeballing.

The ability to specify properties instead of protocols allows applications to take advantage of available protocols where possible. By Happy Eyeballing, NEAT ensures that applications are able to cope with different network configurations, and gracefully fall back to another protocol if necessary should the most desirable protocol not be available for whatever reason.

2.1.4 Asynchronous API

The NEAT API is asynchronous and non-blocking. Once the execution is transferred to NEAT, it will poll the sockets internally, and, when an event happens, execute the appropriate callback in the application. This creates a more natural way of programming communicating applications than with the traditional socket API.

The three most important callbacks in the NEAT API are `on_connected`, `on_readable` and `on_writable`, which may be set per flow. The `on_connected` callback will be executed once the flow has connected to a remote endpoint, or a flow has connected to a server listening for incoming connections. The `on_writable` and `on_readable` callbacks are executed once data may be written to or read from the flow without blocking.

Figure 5 depicts a sufficient callback flow for most applications.

[3]A list of NEAT properties (for `neat_ctx` and `neat_flow` data structures) are defined in § 3.3.4 of Deliverable 1.2.

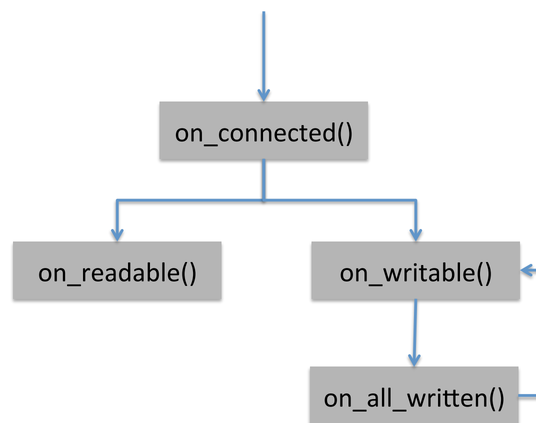


Figure 5: NEAT callback flow.

2.1.5 A minimal server

To get started using the NEAT API, we will write a small server that will send `Hello, this is NEAT!` to any client that connects to it. Later, we will write a similar client, before modifying this server so that it works with the client.

We can summarize the functionality as follows:

- When a client connects, start writing when the flow is writable.
- When a flow is writable, write `Hello, this is NEAT!` to it.
- When the flow has finished writing, close it.

Pay close attention to how easily this natural description can be implemented using the NEAT API.

Here are the includes that should be put on top of the file:

```
#include <neat.h>

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
```

We will start writing the main function of our server. The first thing we need to do is to declare a few variables:

```
main(int argc, char *argv[])
{
    struct neat_ctx *ctx;
    struct neat_flow *flow;
    struct neat_flow_operations ops;
```

And initialize them:

```
    ctx  = neat_init_ctx();
    flow = neat_new_flow(ctx);
    memset(&ops, 0, sizeof(ops));
```

We are already familiar with the flow and the context. `neat_init_ctx` is used to initialize the context, and `neat_new_flow` creates a new flow within the context. The `neat_flow_operations` struct is used to tell NEAT what to do when certain events occur. We will use that next to tell which function we want NEAT to call when a client connects:

```
    ops.on_connected = on_connected;
    neat_set_operations(ctx, flow, &ops);
```

The function `on_connected` has not been defined yet, we will do that later. Now that we have told NEAT what to do with a connecting client, we are ready to accept incoming connections.

```
    if (neat_accept(ctx, flow, 5000, NULL, 0)) {
        fprintf(stderr, "neat_accept failed\n");
        return EXIT_FAILURE;
    }
```

This will instruct NEAT to start listening to incoming connections on port 5000. The flow passed to `neat_accept` is cloned for each accepted connection. The last two parameters are used for optional arguments. This example does not use them.

The last function call we will do in main will be the one that starts the show:

```
    neat_start_event_loop(ctx, NEAT_RUN_DEFAULT);

    return EXIT_SUCCESS;
}
```

When this function is called, NEAT will start doing work behind the scenes. When called with the `NEAT_RUN_DEFAULT` parameter, this function will not return until all flows have closed and all events have been handled. It is also possible to run NEAT without having NEAT capture the main loop. Our final main function looks like this:

```
main(int argc, char *argv[])
{
    struct neat_ctx *ctx;
    struct neat_flow *flow;
    struct neat_flow_operations ops;

    ctx  = neat_init_ctx();
    flow = neat_new_flow(ctx);
    memset(&ops, 0, sizeof(ops));

    // ops.on_readable  = on_readable;
    ops.on_connected = on_connected;
    neat_set_operations(ctx, flow, &ops);

    if (neat_accept(ctx, flow, 5000, NULL, 0)) {
        fprintf(stderr, "neat_accept failed\n");
        return EXIT_FAILURE;
    }

    neat_start_event_loop(ctx, NEAT_RUN_DEFAULT);
```

```
    return EXIT_SUCCESS;
}
```

We have now filled in the main function of our server application. It is time to start working on the callbacks that NEAT will use. The first callback we need is `on_connected`.

```
static neat_error_code
on_connected(struct neat_flow_operations *ops)
{
```

From the functional description above, we know that we need to write to connecting clients when this becomes possible. The callback contains a parameter that is a pointer to a `neat_flow_operations` struct, which we can use to update the active callbacks of the flow. We set the `on_writable` callback so that we can start writing when the flow becomes writable:

```
    ops->on_writable     = on_writable;
```

It is also good practice to set the `on_all_written` callback when setting the `on_writable` callback:

```
    ops->on_all_written = on_all_written;
```

The change is applied by calling `neat_set_operations`, just as in the main function:

```
    neat_set_operations(ops->ctx, ops->flow, ops);

    return NEAT_OK;
}
```

Next, we write the `on_writable` callback:

```
static neat_error_code
on_writable(struct neat_flow_operations *ops)
{
```

Here, we call the function that will send our message:

```
    neat_write(ops->ctx, ops->flow, "Hello, this is NEAT!", 20, NULL, 0);
    return NEAT_OK;
}
```

Here we specify the data to send and the length of the data. As with the `neat_accept` function, `neat_write` takes optional parameters. We do not need to set any optional parameters for this call either, so again we pass `NULL` and `0`.

The final callback we need to implement is the `on_all_written` callback:

```
static neat_error_code
on_all_written(struct neat_flow_operations *ops)
{
```

Here, we call `neat_close` to close the flow:

```
    neat_close(ops->ctx, ops->flow);
    return NEAT_OK;
}
```

This is the final piece of our server. You may now compile and run the server. You can use the tool `socat` to test it. The following output should be observed:

```
$ socat STDIO TCP:localhost:5000
Hello, this is NEAT!
$ socat STDIO SCTP:localhost:5000
Hello, this is NEAT!
```

2.1.6 A minimal client

Next, we want to implement a client that will send the message `"Hi!"` after connecting to a server, and then receive a reply from the server. A fair amount of the code will be similar to the server we wrote above, so you may make a copy of the code for the server and use that as a starting point for the client.

We will make two additions and one change to the main function. First, since we are connecting to a server, we change the `neat_accept` call to `neat_open` instead:

```
    if (neat_open(ctx, flow, "127.0.0.1", 5000, NULL, 0)) {
        fprintf(stderr, "neat_open failed\n");
        return EXIT_FAILURE;
    }
```

Next, we will specify a few properties for the flow:

```
static char *properties = "{\n\
    \"transport\": [\n\
        {\n\
            \"value\": \"SCTP\",\n\
            \"precedence\": 1\n\
        },\n\
        {\n\
            \"value\": \"TCP\",\n\
            \"precedence\": 1\n\
        }\n\
    ]\n\
}";
```

These properties will tell NEAT that it can select either SCTP or TCP as the transport protocol. The properties are applied with `neat_set_properties`, which may be done at any point between `neat_new_flow` and `neat_open`.

Finally, we add `neat_free_ctx` after `neat_start_event_loop`, so that NEAT may free any allocated resources and exit gracefully. The complete main function of the client will look like this:

```
int
main(int argc, char *argv[])
{
    struct neat_ctx *ctx;
    struct neat_flow *flow;
    struct neat_flow_operations ops;

    ctx  = neat_init_ctx();
    flow = neat_new_flow(ctx);
    memset(&ops, 0, sizeof(ops));
```

```
    ops.on_connected = on_connected;
    neat_set_operations(ctx, flow, &ops);

    neat_set_property(ctx, flow, properties);

    if (neat_open(ctx, flow, "127.0.0.1", 5000, NULL, 0)) {
        fprintf(stderr, "neat_open failed\n");
        return EXIT_FAILURE;
    }

    neat_start_event_loop(ctx, NEAT_RUN_DEFAULT);

    neat_free_ctx(ctx);

    return EXIT_SUCCESS;
}
```

Leave the `on_connected` callback similar to the server.

We change the `on_writable` callback to send "`Hi!`" instead:

```
static neat_error_code
on_writable(struct neat_flow_operations *ops)
{
    neat_write(ops->ctx, ops->flow, "Hi!", 3, NULL, 0);
    return NEAT_OK;
}
```

The `on_all_written` callback should not close the flow, but instead stop writing and set the `on_readable` callback:

```
static neat_error_code
on_all_written(struct neat_flow_operations *ops)
{
    ops->on_readable = on_readable;
    ops->on_writable = NULL;
    neat_set_operations(ops->ctx, ops->flow, ops);
    return NEAT_OK;
}
```

Finally, we will write an `on_readable` callback for the client. We allocate some space on the stack to store the received data, and use a variable to store the length of the received message. If the `neat_read` call returns successfully, we print the message. Finally, we stop the internal event loop in NEAT, which will eventually cause the call to `neat_start_event_loop` in the main function to return. The `on_readable` callback should look like this:

```
static neat_error_code
on_readable(struct neat_flow_operations *ops)
{
    uint32_t bytes_read = 0;
    char buffer[32];
```

```
    if (neat_read(ops->ctx, ops->flow, buffer, 31, &bytes_read, NULL, 0) == NEAT_OK)
         {
        buffer[bytes_read] = 0;
        fprintf(stdout, "Read %u bytes:\n%s", bytes_read, buffer);
    }

    neat_close(ops->ctx, ops->flow);
    neat_stop_event_loop(ops->ctx);

    return NEAT_OK;
}
```

And there we have our finished client! You can test it with `socat`:

```
$ socat TCP-LISTEN:5000 STDIO
```

When you run the client, you should see `Hi!` show up in the output from `socat`. You can type a short message followed by pressing return, and it should show up in the output on the client.

2.1.7 Tying the client and server together

A few small changes are required on the server to make the client and server work together. In the `on_connected` callback, the server should set the `on_readable` callback instead of the `on_writable` callback. An `on_readable` callback should be added and read the incoming message from the client, and set the `on_writable` callback.

The callbacks for the updated server are as follows:

```
static neat_error_code
on_readable(struct neat_flow_operations *ops)
{
    uint32_t bytes_read = 0;
    char buffer[32];

    if (neat_read(ops->ctx, ops->flow, buffer, 31, &bytes_read, NULL, 0) == NEAT_OK)
         {
        buffer[bytes_read] = 0;
        fprintf(stdout, "Read %u bytes:\n%s\n", bytes_read, buffer);
    }

    ops->on_readable = NULL;
    ops->on_writable = on_writable;
    ops->on_all_written = on_all_written;
    neat_set_operations(ops->ctx, ops->flow, ops);

    return NEAT_OK;
}

static neat_error_code
on_writable(struct neat_flow_operations *ops)
{
    neat_write(ops->ctx, ops->flow, "Hello, this is NEAT!", 20, NULL, 0);
```

```
    ops->on_writable = NULL;
    ops->on_readable = NULL;
    ops->on_all_written = on_all_written;
    neat_set_operations(ops->ctx, ops->flow, ops);

    return NEAT_OK;
}

static neat_error_code
on_all_written(struct neat_flow_operations *ops)
{
    ops->on_readable = NULL;
    ops->on_writable = NULL;
    ops->on_all_written = NULL;
    neat_set_operations(ops->ctx, ops->flow, ops);

    neat_shutdown(ops->ctx, ops->flow);

    return NEAT_OK;
}

static neat_error_code
on_connected(struct neat_flow_operations *ops)
{
    ops->on_readable = on_readable;
    neat_set_operations(ops->ctx, ops->flow, ops);

    return NEAT_OK;
}
```

2.2 Summary of the benefits of coding with the NEAT User API

This section summarises the simplifications possible when a developer writes an application using the asynchronous and non-blocking NEAT User API.

A key simplification is that the NEAT User API offers a uniform way to access networking functionality, independent from the underlying network protocol or operating system, resulting in portable network applications. Many common network programming tasks like address resolution, buffer management, encryption, connection establishment and handling are built into the NEAT Library and can be used by any application that uses NEAT.

The callback interface is implemented using the libuv [2] library which provides asynchronous I/O across multiple platforms. Coding is also simplified because the NEAT User API executes callbacks in the application when an event from the underlying transport happens, creating a more natural and less error-prone way of network programming than with the traditional socket API. This differs from the code written against the socket API since the callbacks and event loop are handled internally, whereas the socket-based code needs to maintain the event loop and perform the polling.

Our experience with various applications ported to use NEAT shows a reduction of the networking

code size by ≈ 20% for each application[4], as the library streamlines a number of connection establishment steps. For example, the single function call `neat_open` requests name resolution and all other functions required before communication can start, hiding complex boilerplate code. Ported applications remain fully interoperable with regular TCP/IP-based implementations, while being able to take advantage of NEAT functions. Besides, they can benefit from support for alternative transports, when available, relieving programmers from dealing with fallbacks between protocols.

To illustrate some benefits of developing applications against the NEAT API, we have developed a simple example application called FileStreamer [1]. This application sends multiple files from a server to a client using the SCTP protocol. Leveraging the multi-streaming functionality in SCTP, different files can be transferred on different SCTP streams simultaneously.

Our code contains two examples on the server side, one with NEAT (`server_neat.c`) and one without NEAT, purely written with the traditional socket API (`server_sockets.c`). The client side (file receiver) is identical in both cases—NEAT is able to operate one-sided regardless of the support for it on the other end point, which may ease deployment and simplify migration of applications.

Listings 1 and 2 provide code snippets from FileStreamer, showing how a listening socket is set up with NEAT and with the socket API, respectively. As it can be seen in these listings, address resolution, creating and binding to a socket and their related error handling are all done internally by NEAT. Also, using the traditional socket API, SCTP has to be selected explicitly, and the `SCTP_INITMSG` socket option has to be initialized with the number of streams the application wishes to use (lines 42 to 48 of Listing 2). A similar setting without any protocol-related parameter configuration is achieved by setting the protocol-agnostic `NEAT_TAG_STREAM_COUNT` value and the property that expresses the requirement to use SCTP using `neat_set_property` (lines 24 and 30 of Listing 1).

In addition, the socket-based code differs from the code with NEAT in that the former needs to handle itself the polling and event loop (`do_poll` function in `server_sockets.c`, not shown here).

```
1  int
2  setup_neat(const char* port, int file_count, char *file_names[])
3  {
4      int rc = 0;
5      struct neat_ctx *ctx = NULL;
6      struct neat_flow *flow = NULL;
7      struct neat_flow_operations ops;
8      NEAT_OPTARGS_DECLARE(1);
9
10     NEAT_OPTARGS_INIT();
11     memset(&ops, 0, sizeof(ops));
12
13     if ((ctx = neat_init_ctx()) == NULL)
14         return -1;
15
16     if ((flow = neat_new_flow(ctx)) == NULL) {
17         rc = -1;
18         goto error;
19     }
20
21     no_of_files = file_count;
```

[4] The code for several of these applications is available at https://github.com/NEAT-project/neat/tree/master/examples.

```
22     files = file_names;
23 
24     NEAT_OPTARG_INT(NEAT_TAG_STREAM_COUNT, file_count);
25 
26     ops.on_connected = on_connected;
27     ops.on_error = on_error;
28     neat_set_operations(ctx, flow, &ops);
29 
30     neat_set_property(ctx, flow, "{\"transport\": [{\"value\": \"SCTP\", \"
           precedence\": 2}]}");
31 
32     if (neat_accept(ctx, flow, 5001, NEAT_OPTARGS, NEAT_OPTARGS_COUNT) != NEAT_OK)
33         goto error;
34 
35     neat_start_event_loop(ctx, NEAT_RUN_DEFAULT);
36 
37 error:
38     if (ctx)
39         neat_free_ctx(ctx);
40 
41     return rc;
42 }
```

Listing 1: Setting up a listening socket with NEAT.

```
1 int
2 setup_listen_socket(const char* port, int file_count, char* file_names[])
3 {
4     struct addrinfo hints;
5     struct addrinfo *result, *rp;
6     int sfd, s;
7 
8     memset(&hints, 0, sizeof(struct addrinfo));
9     hints.ai_family    = AF_UNSPEC;
10    hints.ai_socktype  = SOCK_STREAM;
11    hints.ai_flags     = AI_PASSIVE;
12    hints.ai_protocol  = IPPROTO_SCTP;
13    hints.ai_canonname = NULL;
14    hints.ai_addr      = NULL;
15    hints.ai_next      = NULL;
16 
17    s = getaddrinfo(NULL, port, &hints, &result);
18    if (s != 0) {
19        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(s));
20        return -1;
21    }
22 
23    for (rp = result; rp != NULL; rp = rp->ai_next) {
24        sfd = socket(rp->ai_family, rp->ai_socktype,
25                     rp->ai_protocol);
```

```
26         if (sfd == -1)
27             continue;
28 
29         if (bind(sfd, rp->ai_addr, rp->ai_addrlen) == 0)
30             break;                  /* Success */
31 
32         close(sfd);
33     }
34 
35     if (rp == NULL) {
36         fprintf(stderr, "Could not bind\n");
37         return -1;
38     }
39 
40     freeaddrinfo(result);
41 
42     struct sctp_initmsg initmsg;
43     memset(&initmsg, 0, sizeof(initmsg));
44 
45     initmsg.sinit_num_ostreams = file_count;
46 
47     if (setsockopt(sfd, IPPROTO_SCTP, SCTP_INITMSG, &initmsg, sizeof(initmsg)) < 0)
48         return -1;
49 
50     if (listen(sfd, SOMAXCONN) < 0) {
51         return -1;
52     }
53 
54     listen_fd = sfd;
55     files = file_names;
56     no_of_files = file_count;
57 
58     return 0;
59 }
```

Listing 2: Setting up a listening socket with the socket API.

3 Core Transport Functions

> ***Note:*** *The descriptions of NEAT components presented in this section* **replace** *those in Deliverable D2.1, reflecting the current status of the implementation.*

This section presents a detailed description of the transport functionalities required to realise the NEAT core transport system based on the components introduced in Section 1.3. Each component is described in detail and Transport Service Features they provide are specified when applicable[5]. Indicative examples of their operation are given where relevant, and relationships among these building

[5] The Transport Service Features mentioned in this deliverable are identified in Tables 1 and 2 of Deliverable 1.2 [26].

blocks are identified. This document does not aim to provide all the implementation details, rather, it intends to present the design choices that have been made. For a more comprehensive reference of the NEAT User API, please refer to Appendix B[6]. When appropriate, snippets of sample code in C are used for better presentation in this section.

3.1 NEAT Framework Components

To run a NEAT System a minimum set of basic building blocks has to be implemented, comprising the NEAT Framework components. This translates into being able to create a NEAT Flow and connect to a host using a domain name address, as well as the ability to translate the functionalities behind the NEAT User API into appropriate function calls, e.g., to different protocols, mechanisms, etc.

Setting up a NEAT Flow can be done by using an event-based, user-space *NEAT library* that implements a callback-based API (NEAT API Framework). Once a NEAT Flow is initialised, it will contain a structure that keeps all of its relevant information during its lifetime (NEAT Flow Endpoint). A NEAT Flow can be assigned to one or more domain names as well as IP addresses (Connect to a name). The functionalities behind the NEAT User API requested for the initialised NEAT Flow require code that "glues" together different components (NEAT Logic). This is not a monolithic chunk of code separated from other components, but rather code that is scattered throughout other components as well as the NEAT User API. Finally, gathering statistics and information about the operation of the system is necessary for diagnostics and performance monitoring (NEAT Flow Endpoint Statistics). The operation of each of these components is presented in the rest of this section.

3.1.1 NEAT Flow Endpoint

The NEAT Flow Endpoint is a NEAT structure that has a role similar to that of the Transmission Control Block (TCB) in the context of TCP [10]. This provides access to the additional flow information that extends the transport components and realises a transport independent interface. Through the NEAT Logic parts of this information are used by most of the other building blocks (e.g., Policy Manager, Happy Eyeballs, Security, etc.).

A NEAT Flow Endpoint `neat_flow` structure corresponds to a single operating system socket or to a stream if multistreaming is used (only for multistreaming capable protocols, e.g., SCTP) and keeps the information about the socket or stream that is relevant during the NEAT Flow's lifetime (Listing 3). This includes:

- A structure that holds information about the underlying OS socket as well as the pointer to the OS file descriptor or a user-space SCTP socket. This information includes: socket family (e.g. AF_INET, AF_INET6), socket type (e.g., SOCK_DGRAM, SOCK_STREAM), protocols (e.g., IPPROTO_UDP, IPPROTO_TCP, IPPROTO_SCTP)
- Remote peer domain name.
- Server certificates in PEM format [8, 18, 19, 21].
- Remote port.
- Remote socket address.

[6] Also available online at: http://neat.readthedocs.io/en/latest/index.html.

- Local address or addresses.

```
1 struct neat_flow
2 {
3     struct neat_pollable_socket *socket;
4     TAILQ_HEAD(neat_listen_socket_head, neat_pollable_socket) listen_sockets;
5
6     const char *name;
7     char *server_pem;
8     uint16_t port;
9     const struct sockaddr *sockAddr; // raw unowned pointer into resolver_results
10    const char *local_address;
```

Listing 3: NEAT Flow Endpoint structure.

It also contains information relevant to the NEAT System operations (e.g., DNS resolver outcome), flow statistics and desired properties set by an application (e.g., QoS, ECN, etc.) and includes (Listing 4):

- Address resolver results, i.e., DNS results.
- Flow statistics: this includes the path MTU, slow-start threshold, round-trip time (RTT), etc.
- Requested properties for the NEAT Flow: these are properties that an application has specified when the flow has been created via the NEAT User API. They are in JSON format (see Listing 18). The Policy Manager translates these properties into a list of candidates.
- Candidate list: it is used by the Happy Eyeballs component (§ 3.3.1) to create concrete socket configuration probes.
- Connection attempt count: the number of different configurations the Happy Eyeballs component has tried.
- QoS: the QoS encoding to be used with this flow.
- ECN: a value to indicate if use of ECN should be attempted.
- Group: the group ID that this flow belongs to. This is used for the coupled congestion control (§ 3.2.3).
- Flow priority: the priority of the given NEAT flow relative to the other flows in the same group. Must be between 0.1 and 1.0.
- Congestion control: The congestion control algorithm to be used for this flow.

```
11    struct neat_resolver_results *resolver_results;
12    struct neat_flow_statistics flow_stats;
13    json_t *properties;
14    struct neat_he_candidates *candidate_list;
15    uint8_t heConnectAttemptCount;
16
17    uint8_t qos;
```

```
18      uint8_t ecn;
19
20      uint32_t group;
21      float priority;
22
23      const char *cc_algorithm;
```

Listing 4: NEAT Flow Endpoint structure (continued).

The NEAT System provides a callback-based API to the application. Pointers to the callback functions are kept in the NEAT Flow Endpoint structure (Listing 5). Their usage will be clarified in § 3.1.2.

```
24      struct neat_flow_operations *operations;
```

Listing 5: NEAT Flow Endpoint structure (continued).

The flow endpoint structure is used as the main hub for communication with the underlying socket. Therefore it contains the pointers to the NEAT base loop structure `neat_ctx`, functions for accessing the underlying socket, and NEAT Flow internal flags and buffers (the write buffer facilitates preservation of message boundaries if the selected transport protocol is message-based), see Listing 6.

```
25      // NEAT base loop structure:
26      struct neat_ctx *ctx;
27
28      // Functions for accessing the underlying socket:
29      neat_read_impl readfx;
30      neat_write_impl writefx;
31      neat_accept_impl acceptfx;
32      neat_connect_impl connectfx;
33      neat_close_impl closefx;
34      neat_close2_impl close2fx;
35      neat_listen_impl listenfx;
36      neat_shutdown_impl shutdownfx;
37
38      // NEAT internal flags:
39      unsigned int hefirstConnect : 1;
40      int firstWritePending : 1;
41      int acceptPending : 1;
42      int isPolling : 1;
43      int ownedByCore : 1;
44      int everConnected : 1;
45      int isDraining : 1;
46      unsigned int isServer : 1;
47      unsigned int isSCTPMultihoming : 1;
48
49      // Write buffer:
50      struct neat_message_queue_head bufferedMessages;
```

Listing 6: NEAT Flow Endpoint structure (continued).

SCTP needs additional internal flow states and variables shown in Listing 7. This includes additional information needed if user-space SCTP or multistreaming are configured.

```
51     // The memory buffer for reading. Used of SCTP reassembly.
52     unsigned char    *readBuffer;     // memory for read buffer
53     size_t           readBufferSize;          // amount of received data
54     size_t           readBufferAllocation;  // size of buffered allocation
55     int readBufferMsgComplete; // it contains a complete user message
56
57     unsigned int streams_requested;
58
59 #if defined(USRSCTP_SUPPORT)
60     neat_accept_usrsctp_impl acceptusrsctpfx;
61 #endif
62
63
64 #ifdef SCTP_MULTISTREAMING
65     unsigned int                    multistream_check        : 1;
66     unsigned int                    multistream_shutdown     : 1;
67     unsigned int                    multistream_reset_in     : 1;
68     unsigned int                    multistream_reset_out    : 1;
69
70     uv_timer_t                      *multistream_timer;
71     uint16_t                        multistream_id;
72     LIST_ENTRY(neat_flow)           multistream_next_flow;
73
74     struct neat_read_queue_head     multistream_read_queue;
75     size_t                          multistream_read_queue_size;
76
77     neat_flow_states                multistream_state;
78 #endif
79 }
```

Listing 7: NEAT Flow Endpoint structure (continued).

Some examples of the operation: Listing 8 shows an example of how the NEAT Flow Endpoint structure is used. After DNS is resolved a list of candidates is handed over to the Happy Eyeballs component and this component will try to establish connections using some or all of the given candidates. After the first candidate is connected, the NEAT system has all socket parameters and these parameters are stored in the `neat_flow` structure. The other candidates are released. This is done in the `he_connected_cb` callback function. For simplicity, some parts of the code are omitted.

The `he_connected_cb` function is called when a connection is established or an error occurs. The function parameter `uv_poll_t` holds pointer to the candidate, i.e. a `neat_he_candidate` structure. The `neat_he_candidate` structure holds a pointer to the corresponding `neat_flow` structure. When the first candidate is successfully connected, the `neat_flow` structure is filled with the necessary information (lines 34-47). When the other candidates are connected, the corresponding socket is closed (lines 50-63).

```
1 static void
2 he_connected_cb(uv_poll_t *handle, int status, int events)
3 {
4     struct neat_he_candidate *candidate = handle->data;
```

```
5       struct neat_flow *flow = candidate->pollable_socket->flow;
6       struct neat_he_candidates *candidate_list = flow->candidate_list;
7
8       assert(candidate);
9       assert(candidate->pollable_socket);
10      assert(flow);
11
12      int so_error = 0;
13      unsigned int len = sizeof(so_error);
14      if (getsockopt(candidate->pollable_socket->fd, SOL_SOCKET, SO_ERROR, &so_error,
            &len) < 0) {
15
16          neat_log(NEAT_LOG_DEBUG, "Call to getsockopt for fd %d failed: %s",
                candidate->pollable_socket->fd, strerror(errno));
17
18          uv_poll_stop(handle);
19          uv_close((uv_handle_t*)handle, free_he_handle_cb);
20
21          neat_io_error(candidate->ctx, flow, NEAT_ERROR_INTERNAL);
22          return;
23      }
24      status = so_error;
25      neat_log(NEAT_LOG_DEBUG, "%s - Connection status: %d", __func__, status);
26
27      if (flow->hefirstConnect && (status == 0)) {
28          // This is the fastest candidate.
29          // Change the flow internal state after one of the candidates was
                successfully connected.
30          flow->hefirstConnect = 0;
31
32          assert(flow->socket);
33
34          flow->socket->fd = candidate->pollable_socket->fd;
35          flow->socket->flow = flow;
36          flow->socket->handle                = handle;
37          flow->socket->handle->data          = flow->socket;
38          flow->socket->family                = candidate->pollable_socket->family;
39          flow->socket->type                  = candidate->pollable_socket->type;
40          flow->socket->stack                 = candidate->pollable_socket->stack;
41          flow->socket->write_size            = candidate->pollable_socket->write_size
                ;
42          flow->socket->write_limit           = candidate->pollable_socket->
                write_limit;
43          flow->socket->read_size             = candidate->pollable_socket->read_size;
44          flow->socket->sctp_explicit_eor     = candidate->pollable_socket->
                sctp_explicit_eor;
45
46          flow->everConnected = 1;
47          flow->isPolling = 1;
```

```
48
49     } else {
50         // This is not the first candidate so close the socket.
51         neat_log(NEAT_LOG_DEBUG, "%s - NOT first connect", __func__);
52
53         uv_poll_stop(handle);
54         uv_close((uv_handle_t*)handle, free_he_handle_cb);
55
56         neat_log(NEAT_LOG_DEBUG, "%s:Release candidate", __func__);
57         TAILQ_REMOVE(candidate_list, candidate, next);
58         free(candidate->pollable_socket->dst_address);
59         free(candidate->pollable_socket->src_address);
60         free(candidate->pollable_socket);
61         free(candidate->if_name);
62         json_decref(candidate->properties);
63         free(candidate);
64     }
65 }
```

Listing 8: Use of the NEAT Flow Endpoint structure.

Provided Transport Service Feature(s): This building block is part of the most basic functionality of the NEAT System and does not relate to any specific application requirement.

Related building blocks:

- NEAT Logic (§ 3.1.3).

3.1.2 NEAT API Framework (callback)

The NEAT System implements a callback-based API based on the libuv [2] library to provides portable asynchronous I/O across multiple platforms. The base of the NEAT System is an event loop that needs to be initialised before any NEAT functionality can be accessed. NEAT uses libuv [2] as an event library. Once the NEAT base structure has started, 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.

The NEAT System offers several basic functions for accessing a NEAT Flow (Listing 9):

- Creating and closing a NEAT Flow, i.e., a NEAT Flow Endpoint structure (this structure is described in § 3.1.1).
- Setting callback functions.
- Opening a connection to a remote host: this function starts the NEAT Logic for selecting and creating the most adequate Transport Service. The outcome of this asynchronous call can be an `on_connected` event in case of success, or an `on_error` event if a failure occurs. Besides a remote host name and a port number, this function can take some additional parameters such as the flow priority, a local address to be used, requesting multihoming, etc. These additional parameters are optional.

- Read and write to a NEAT Flow and through it to the underlying socket: the return values correspond to the return values of the operating system function calls, or of the TLS/DTLS function calls if secure communication is established.
- Functions for getting and setting properties: using these functions, an application can set or change transport requirements (e.g., reliable transport, low latency, etc.). The properties are in a JSON format, their format is presented in Listing 18.
- Opening a NEAT Flow for listening (i.e., server-side socket): this triggers an `on_connected` event if a new connection request is received or an `on_error` event in the case of an error. Besides a port number, a local address to be used and the maximal number of streams can be specified. These parameters are optional.
- Getting flows statistics (e.g., the round-trip time, the slow-start threshold, etc.).
- Querying the flow for local and remote peer address.
- Setting a user specified timeout, i.e., `TCP_USER_TIMEOUT`.
- Setting some parameters that are specific to certain protocols, e.g., `neat_set_primary_dest` only used with SCTP and `neat_set_checksum_coverage` only used for UDP and UDPLite.
- Setting server certificates (i.e., `neat_secure_identity`).
- Setting additional desired parameters, such as QoS and ECN.

```
1 struct neat_flow *neat_new_flow(struct neat_ctx *ctx);
2
3 neat_error_code neat_shutdown(struct neat_ctx *ctx, struct neat_flow *flow);
4 neat_error_code neat_close(struct neat_ctx *ctx, struct neat_flow *flow);
5 neat_error_code neat_abort(struct neat_ctx *ctx, struct neat_flow *flow);
6
7 neat_error_code neat_set_operations(struct neat_ctx *ctx, struct neat_flow *flow,
8                                    struct neat_flow_operations *ops);
9 neat_error_code neat_open(struct neat_ctx *ctx, struct neat_flow *flow,
10                          const char *name, const char *port,
11                          struct neat_tlv optional[], unsigned int opt_count);
12 neat_error_code neat_read(struct neat_ctx *ctx, struct neat_flow *flow,
13                          unsigned char *buffer, uint32_t amt, uint32_t *actualAmt,
14                          struct neat_tlv optional[], unsigned int opt_count);
15 neat_error_code neat_write(struct neat_ctx *ctx, struct neat_flow *flow,
16                           const unsigned char *buffer, uint32_t amt,
17                           struct neat_tlv optional[], unsigned int opt_count);
18
19 neat_error_code neat_get_property(struct neat_ctx *ctx, struct neat_flow *flow,
20                                  const char* name, void *ptr, size_t *size);
21 neat_error_code neat_set_property(struct neat_ctx *ctx, struct neat_flow *flow,
22                                  const char* properties);
23
24 neat_error_code neat_accept(struct neat_ctx *ctx, struct neat_flow *flow,
25                            uint16_t port, struct neat_tlv optional[],
```

```
26                             unsigned int opt_count);
27
28 neat_error_code neat_get_stats(struct neat_ctx *ctx, char **neat_stats);
29
30 int neat_getlpaddrs(struct neat_ctx *ctx, struct neat_flow *flow,
31                     struct sockaddr** addrs, const int local);
32
33 neat_error_code neat_change_timeout(struct neat_ctx *ctx, struct neat_flow *flow,
34                                     unsigned int seconds);
35
36 neat_error_code neat_set_primary_dest(struct neat_ctx *ctx, struct neat_flow *flow,
37                                       const char *name);
38 neat_error_code neat_set_checksum_coverage(struct neat_ctx *ctx,
39                                            struct neat_flow *flow,
40                                            unsigned int send_coverage,
41                                            unsigned int receive_coverage);
42
43 neat_error_code neat_secure_identity(struct neat_ctx *ctx, struct neat_flow *flow,
44                                      const char *filename);
45
46 neat_error_code neat_set_qos(struct neat_ctx *ctx, struct neat_flow *flow,
47                              uint8_t qos);
48 neat_error_code neat_set_ecn(struct neat_ctx *ctx, struct neat_flow *flow,
49                              uint8_t ecn);
```

Listing 9: NEAT API functions.

The NEAT API offers multiple run-time events that call the corresponding functions if registered. The set of events that are triggered are given in Listing 10.

```
1 neat_flow_operations_fx on_connected;
2 neat_flow_operations_fx on_error;
3 neat_flow_operations_fx on_readable;
4 neat_flow_operations_fx on_writable;
5 neat_flow_operations_fx on_all_written;
6 neat_flow_operations_fx on_network_status_changed;
7 neat_flow_operations_fx on_aborted;
8 neat_flow_operations_fx on_timeout;
9 neat_flow_operations_fx on_close;
10 neat_cb_send_failure_t on_send_failure;
11 neat_cb_flow_slowdown_t on_slowdown;
12 neat_cb_flow_rate_hint_t on_rate_hint;
```

Listing 10: NEAT callback functions.

In the following, we present a simple illustration of the NEAT Flow connection establishment and maintenance. Each application needs to initialise the base NEAT structure and before a communication can start its event loop needs to be started. An application creates a NEAT Flow Endpoint for each connection. The application specifies its requirements by utilising the `neat_set_property` and `neat_get_property` functions of the NEAT User API called for each individual NEAT Flow Endpoint. When the application requirements are specified, the connection establishment can be re-

quested by calling the `neat_open` function (corresponding to the OPEN primitive from D1.2 [26]). The function takes a host name and a port number as parameters and maybe some optional parameters. On the server side, the `neat_accept` function (corresponding to the ACCEPT primitive from D1.2 [26]) will be called with parameters: port number and local address the socket should be listening to, and optionally additional parameters.

These two function calls will trigger a set of actions inside the NEAT System. The specified application requirements will be used by the NEAT Logic and the corresponding building blocks (e.g., Policy Manager and Happy Eyeballs) to select and probe selected socket configurations. If a socket that satisfies the application requirements has been successfully connected, an `on_connected` callback function, if registered, will be invoked. Otherwise the `on_error` callback function will be invoked.

The application can register callback functions for `on_socket_readable` and `on_socket_writable` events which will translate into poll parameters for the underlying socket. The functions will be executed if the corresponding event applies.

The NEAT System buffers data that needs to be written. This is necessary to facilitate preservation of message boundaries if the selected transport protocol is message-based. The `on_all_written` event is triggered when all buffered data is written out to the OS socket.

In case of an error (e.g., NEAT internal error, socket error, socket being closed, etc.) an `on_error` event callback function will be invoked.

Some examples of the operation: Listing 11 presents a simple example of setting callback functions and waiting for a callback function to be called. In function `main` a NEAT base loop structure and a NEAT Flow are created (lines 21 and 28). Callback functions `on_error` and `on_connected` are set in lines 59–60. `neat_open` is called in line 70 and if it does not return an error the NEAT loop will be started (line 71). In case of an error the `on_error` function will be invoked, otherwise `on_connected` will be invoked which sets callback functions for `on_all_written` and `on_readable`.

```
1
2  static struct neat_flow_operations ops;
3
4  /*
5      Error handler
6  */
7  static neat_error_code on_error(struct neat_flow_operations *opCB)
8  {
9      exit(EXIT_FAILURE);
10 }
11
12 static neat_error_code on_connected(struct neat_flow_operations *opCB)
13 {
14     opCB->on_all_written = on_all_written;
15     opCB->on_readable = on_readable;
16     return NEAT_OK;
17 }
18
19 int main(int argc, char *argv[])
20 {
21     if ((ctx = neat_init_ctx()) == NULL) {
```

```
22          debug_error("could not initialise context");
23          result = EXIT_FAILURE;
24          goto cleanup;
25      }
26
27      // new neat flow
28      if ((flow = neat_new_flow(ctx)) == NULL) {
29          debug_error("neat_new_flow");
30          result = EXIT_FAILURE;
31          goto cleanup;
32      }
33
34      char *config_property = "{\
35          \"transport\": [\
36              {\
37                  \"value\": \"SCTP\",\
38                  \"precedence\": 1\
39              },\
40              {\
41                  \"value\": \"SCTP/UDP\",\
42                  \"precedence\": 1\
43              },\
44              {\
45                  \"value\": \"TCP\",\
46                  \"precedence\": 1\
47              }\
48          ]\
49      }";
50
51      // set properties
52      if (neat_set_property(ctx, flow, config_property)) {
53          fprintf(stderr, "%s - error: neat_set_property\n", __func__);
54          result = EXIT_FAILURE;
55          goto cleanup;
56      }
57
58      // set callbacks
59      ops.on_connected = on_connected;
60      ops.on_error = on_error;
61
62      if (neat_set_operations(ctx, flow, &ops)) {
63          debug_error("neat_set_operations");
64          result = EXIT_FAILURE;
65          goto cleanup;
66      }
67
68      // wait for on_connected or on_error to be invoked
69      // The last 2 arguments are the remote server name and port number.
70      if (neat_open(ctx, flow, argv[argc - 2], argv[argc - 1]) == NEAT_OK) {
```

```
71          neat_start_event_loop(ctx, NEAT_RUN_DEFAULT);
72      } else {
73          debug_error("neat_open");
74          result = EXIT_FAILURE;
75          goto cleanup;
76      }
77
78      ...
79  }
```

Listing 11: NEAT API Framework example.

Provided Transport Service Feature(s): This building block is part of the most basic functionality of the NEAT System and it does not relate to any specific Transport Service Feature.

Related building blocks:

- NEAT Logic (§ 3.1.3).

3.1.3 NEAT Logic

The NEAT Logic component 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. It orchestrates and "glues" together different components. NEAT Logic is not a monolithic piece of code separated from other components, but its code is scattered throughout other components as well as the NEAT User API.

Requests made via the NEAT User API are translated into function calls to the Policy Manager or other building blocks; for instance, calls to select the transport protocols to be instantiated, or calls to Handover and Signalling after receiving a set of candidates from the Policy Manager. Transport protocols are configured via the relevant NEAT Transport components. NEAT Logic dispatches different decisions returned by the Policy Manager by translating them into certain function calls related to NEAT components—e.g., by calling the Happy Eyeballs function(s) for SCTP/TCP or IPv6/IPv4. In simpler terms, it glues different building blocks of the NEAT System together and makes them operational in one uniform system.

NEAT Logic also maps the primitives and events exposed to the application (listed in § 3.3 of Deliverable D1.2 [26]) to the primitives provided by each transport protocol.

A detailed sequence of NEAT Logic operation with regards to the connection setup in NEAT is presented in Section 1.4 and therefore we avoid repeating it in here for the sake of brevity.

Some examples of the operation: An example of NEAT Logic operation is when the `neat_open` call, corresponding to the OPEN primitive (see D1.2 [26]), is used to open a NEAT Flow. The OPEN primitive does not specify any specific transport protocol. After processing the optional arguments passed in the `neat_tlv` struct (e.g., priority, flow group or congestion control), and initialising the NEAT Flow's address name and port, it would send the properties to the Policy Manager using the `send_properties_to_pm()` function[7]. This corresponds to the first query to the Policy Manager as depicted in Figure 4 and explained in Section 1.4.

[7] The flow properties should be already set by `neat_set_property()` prior to the `neat_open` function call.

```
1  neat_error_code
2  neat_open(neat_ctx *ctx, neat_flow *flow, const char *name, uint16_t port,
3            struct neat_tlv optional[], unsigned int opt_count)
4  {
5  ...
6
7      HANDLE_OPTIONAL_ARGUMENTS_START()
8          OPTIONAL_INTEGER(NEAT_TAG_STREAM_COUNT, stream_count)
9          OPTIONAL_INTEGER(NEAT_TAG_FLOW_GROUP, group)
10         OPTIONAL_FLOAT(NEAT_TAG_PRIORITY, priority)
11         OPTIONAL_STRING(NEAT_TAG_CC_ALGORITHM, cc_algorithm)
12     HANDLE_OPTIONAL_ARGUMENTS_END();
13
14 ...
15
16     flow->name = strdup(name);
17     if (flow->name == NULL)
18         return NEAT_ERROR_OUT_OF_MEMORY;
19     flow->port = port;
20     flow->group = group;
21     flow->priority = priority;
22
23 ...
24
25     send_properties_to_pm(ctx, flow);
26     return NEAT_OK;
27 }
```

Listing 12: NEAT open function.

Provided Transport Service Feature(s): There are no specific Transport Service Features associated to this building block. However, the operation of NEAT Logic is essential for other building blocks to provide their Transport Service Features.

Related building blocks:

- NEAT Flow Endpoint Statistics (§ 3.1.5).
- Middlebox Traversal (§ 3.2.2).
- NEAT Flow Endpoint (§ 3.1.1).
- NEAT API Framework (callback) (§ 3.1.2).
- Connect to a name (§ 3.1.4).
- Happy Eyeballs (§ 3.3.1).
- Security (§ 3.2.4).
- NEAT Policy Manager (via Policy Interface) (§ 3.4.1).

3.1.4 Connect to a name

The higher-level API offered by NEAT provides transport-independent name resolution. This approach not only avoids dependencies on specific network technology, but is also essential to support multiple active network interfaces. The NEAT system address resolver is named *Connect to a name*. NEAT will by default resolve domains using the public DNS servers provided by Google and OpenDNS (both v4 and v6). If a system-wide resolver file is found, then the list is extended with the servers contained in this file.

Currently we support resolving names using plain DNS (i.e., no DNSSEC), but the component is designed in such a way that extending the functionality is easy. One can for example imagine that a new name resolution scheme, protocol or technique will be introduced later on. An IP literal can also be provided, in which case no translation will be performed.

The resolver supports multi-homed hosts. When the NEAT resolve function is called, the query will by default be sent over all available (interface, address) tuples. Only A records are requested over IPv4 addresses, while AAAA records are requested over the available IPv6 addresses. A challenge on multi-homed hosts, which cannot be solved within NEAT, is to make sure that we use the correct interface/address when communicating with a DNS server. This is an issue when for example a host is connected to overlapping networks, or when the DNS servers acquired through DHCP have public IP addresses. There is currently no standardized syntax for how to specify which local interface/address should be used to communicate with a DNS server. We designed, implemented and shared a solution for the popular open-source resolver *dnsmasq*[8], but the discussions with the dnsmasq community have not progressed yet.

Interface/address tuples used by NEAT are stored in a list which is dynamically updated based on events generated by the OS. All major OS support mechanisms for generating events when addresses/network interfaces are added/removed. The mechanisms differ based on OS, so small shims are needed to support different operating systems. However, the core code (and the content of the list) is platform-independent.

The interface/address list is stored in the `neat_ctx` object introduced in § 3.1.1 and is available for use by all other building blocks. For example the address monitoring functionality, combined with an internal notification subsystem, will make reacting to interfaces going up/down more efficient across all blocks.

In summary, the following features are provided by the *Connect to a name* component:

- *Asynchronous DNS lookup:* name resolving will not block the calling application.
- *Address monitoring:* (interface, address) tuples are stored in a dynamically updated list, which is available to all building blocks.
- *Multi-homing support:* the resolver will resolve names using all (interface, address) tuples on a host by default.
- *Private network marking:* names resolving to internal addresses will be marked and can be easily filtered.

Some examples of the operation: In order to use the resolver, a developer has to create the NEAT context first using the `neat_init_ctx` call. Then, the `neat_resolver_init` function has to be

[8] http://www.thekelleys.org.uk/dnsmasq/doc.html

called in order to set up the resolver. This function is passed two function pointers, one that will be called when the resolver finishes (or times out), and one called when the resolver can be released.

After these two functions have been called, it is simply a matter of calling `neat_getaddrinfo`. This function works very much similar to the POSIX-compliant `getaddrinfo`. In other words, it is possible to limit which address family and transport protocol for the resolver to query/return. One change from the normal `getaddrinfo` is that returning multiple transport protocols is supported.

When the resolving is done (or has failed, e.g., due to a timeout), the provided callback function is called. If successful, this function is passed a list of all (interface, source address, destination address) tuples. For example these can be used by a developer to connect to the desired host, or the list will serve as input to the Happy Eyeballs component.

Provided Transport Service Feature(s): Connect to a name.

Related building blocks:

- Connect to a name has no dependencies on other building blocks except NEAT Logic (§ 3.1.3), but several building blocks may depend on the offered functionality. One example is Happy Eyeballs (§ 3.3.1) that must be provided with a set of source/destination addresses to probe for IPv4/IPv6 and transport-protocol connectivity.

3.1.5 NEAT Flow Endpoint Statistics

NEAT not only automates important network decisions for applications, it can also help understand how these decisions were taken. By making this flow information available to the user in a consistent form, it eases the burden of identifying the root causes of any network problems.

The NEAT Flow Endpoint Statistics component is responsible for maintaining information about the current state of the NEAT System, and for gathering usage statistics for both the overall NEAT System and the respective NEAT Flows. This information resembles the information provided by `netstat` in a traditional socket stack, but at the NEAT Flow level of detail. It can be used for application-level diagnostic purposes and for allowing applications to monitor the performance of application flows (e.g., measuring the throughput of NEAT Flows) to take decisions based on provided detailed information.

The information provided by the NEAT Flow Endpoint Statistics building block can be divided into three sets: 1) *current NEAT state*, 2) *NEAT Flow statistics*, and 3) *NEAT System statistics*. The *current NEAT state* set of information provides a detailed view of the current state of the NEAT System. It provides a list of all NEAT Flows that are currently open on the NEAT System along with details about their configuration. The *NEAT Flow statistics* set of information contains usage statistics information for each NEAT Flow that has been created since the instantiation of the NEAT System. The *NEAT System statistics* set of information contains system-wide usage statistics of the NEAT System. Some examples of these three categories are listed below:

- *Current NEAT state:* flow ID, flow creation time, local name, local transport address(es), destination name, destination transport address(es), send queue size, protocol state, transport parameters (e.g., Nagle, DSCP, timeout, etc.), flow properties, interface(s) in use.
- *NEAT Flow statistics:* number of bytes sent/received, number of messages sent/received, number of messages dropped locally, number of handovers, connection duration, creation time.

- *NEAT System statistics:* total number of bytes sent/received, total number of messages sent/received, total number of messages dropped locally, total number of opened/accepted/closed/aborted connections, minimum/average/maximum flow duration, statistics on failures (errors) reported by the NEAT System.

The number and content of the elements returned after a call to the statistics interface will vary between NEAT instances, and even within the lifetime of one NEAT context. Different operating systems, for example, support different transport options, meaning that the NEAT state will look different between operating systems. Also, the protocol used for a particular flow will be different between connections, meaning that a different set of available options will have to be presented each time the statistics module is called. This calls for the statistics module to use a format that supports such dynamic content. In line with the standards chosen for the PM, the statistics module therefore employs a JSON format supporting different elements in a nested structure of objects. Listing 13 shows an example of how statistics are presented in this format. An important consideration is to make sure that the JSON statistics elements are extended/evolved in a backwards compatible way, such that applications will always be able to collect the statistics they expect in the required format.

```
1 {
2   "flow1": {
3     "flow number": 1,
4     "sock_protocol": 6,
5     "remote_host": "localhost",
6     "readSize": 87380,
7     "bytes sent": 2712,
8     "socket type": 1,
9     "tcpstats": {
10      "reordering": 3,
11      "retransmits": 0,
12      "rtt": 19,
13      "pmtu": 65535,
14      "rttvar": 8,
15      "ssthresh": 2147483647,
16      "rcv_ssthresh": 43690,
17      "snd_cwnd": 10
18    },
19    "port": 8080,
20    "writeSize": 16384,
21    "bytes received": 1337
22  },
23  "flow2": {
24    "flow number": 1,
25    "sock_protocol": 6,
26    "remote_host": "localhost",
27    "readSize": 87380,
28    "bytes sent": 6,
29    "socket type": 1,
30    "tcpstats": {
31      "reordering": 3,
32      "retransmits": 0,
```

```
33         "rtt": 19,
34         "pmtu": 65535,
35         "rttvar": 8,
36         "ssthresh": 2147483647,
37         "rcv_ssthresh": 43690,
38         "snd_cwnd": 10
39       },
40       "port": 8080,
41       "writeSize": 16384,
42       "bytes received": 0
43     },
44     "number of flows": 2,
45     "global bytes sent": 2718,
46     "global bytes received": 1337
47 }
```

Listing 13: NEAT Statistics JSON format example.

Applications that want to take advantage of this additional information provided by the NEAT System (i.e., Class-4 applications in Figure 2) can access it through the Diagnostics and Statistics Interface. The scope of the information maintained by the NEAT Flow Endpoint Statistics is local to the application that uses a particular NEAT System instance and is maintained within the application context as long as the application is running. In contrast to other types of statistical information collected from other components of the NEAT System (e.g., the interface and path statistics collected by CIB sources) this information is not stored in any CIB and is not shared among any other NEAT System instances that may be running on the same physical machine. In order to build a tool to collect statistics from many NEAT contexts, to present server-wide statistics, the functionality may be extended to also provide statistics through a local socket interface to users with root privileges.

Some examples of the operation: Listing 14 shows an example of how the statistics can be called by an application. In the example, the JSON string is simply printed to the standard output, but normally the application would parse the string, pick the relevant elements and process the content. Since the statistics are passed by reference, the reserved memory needs to be freed by the calling application. This is done for efficiency reasons, to save system resources.

```
1 static void
2 print_neat_stats(neat_ctx *mgr)
3 {
4     neat_error_code error;
5
6     char* stats = NULL;
7     error = neat_get_stats(mgr, &stats);
8     if (error != NEAT_OK){
9         printf("NEAT ERROR: %i\n", (int)error);
10        return;
11    } else if (stats != NULL) {
12        printf("json %s\n", stats);
13    }
14
```

```
15     free(stats);
16 }
```

Listing 14: Simple application code that prints NEAT Statistics.

An application can leverage the information provided by the NEAT Flow Endpoint Statistics in order to verify that the provided Transport Services are consistent with the requested features/properties, and also to trace decisions made by the NEAT System throughout the progress of NEAT Flows which are normally not intended to be reported back to the application. For example, an application can periodically request current state information in order to be aware of any handover decisions (in case seamless handover is enabled) or changes in transport protocol parameters made by the NEAT System.

Furthermore, the statistical information provided by this building block, combined with the other types of statistics (e.g., path and interface statistics) that are also exposed through the Diagnostics and Statistics Interface, can allow applications to monitor the actual application and network performance in order to implement more specialised functionalities. For example this statistical information may inform application decisions on controlling the behaviour of other applications (e.g., controlling interactions with an SDN controller) in order to optimise performance.

Provided Transport Service Feature(s): This building block is meant mainly for diagnostic purposes and therefore does not relate to any specific Transport Service Features.

Related building blocks:

- NEAT Logic (§ 3.1.3).
- Policy Manager (via Policy Interface) (§ 3.4.1).

3.2 NEAT Transport Components

The NEAT User API offers applications seamless access to the Transport Service Features of standardised transport protocols, while ensuring packets get through the network in the presence of non-supportive middleboxes, or challenging network paths, and providing a path to seamlessly introduce new transport protocols or transport protocol features (e.g., a Less-Than-Best Effort (LBE) Transport Service that considers data-delivery deadlines). While the *selection* of transport protocols are handled by the NEAT Selection Components, the NEAT Transport Components are responsible for *configuring and managing* the transport protocols.

NEAT Transport components offer a consistent abstraction over underlying transport services, e.g., implementing virtual accept for UDP, thus providing a single API to applications. NEAT Transport components allow applications to use transport stacks implemented natively in operating systems and stacks implemented in userspace, such as the usrsctp stack (§ 3.2.1).

The NEAT Transport components manage the combination of protocol parameters used (e.g., use of Nagle and other socket options to create a low-delay TCP service) and the transport protocol components that are enabled (e.g., activation of SCTP-PR or SCTP-PF respectively to create a partial reliability service or a service supporting fast path failover).

In addition, NEAT Transport components implement the necessary functionalities to assign different priorities to each NEAT Flow within a flow group. These priorities then could be used by different

mechanisms at the transport level (§ 3.2.3)—e.g., coupled congestion control in TCP and per-stream scheduling priorities in SCTP.

3.2.1 NEAT-integrated SCTP user-space stack

The *usrsctp* SCTP userland stack [7] presents a possibility to use SCTP as transport protocol when the OS does not support kernel SCTP (like Windows, NetBSD or Mac OS X), and also with Linux when SCTP is not loaded explicitly.

Usrsctp has been ported from the FreeBSD SCTP kernel sources and is still kept in sync. A fully featured API has been provided to easily port applications.

As SCTP is an attractive alternative to TCP and UDP, it should be provided for all platforms that NEAT runs on. Therefore, we forked the original usrsctp repository and adapted it to the needs of the NEAT System.

Thread Support: When using usrsctp the transport is performed by raw sockets whose I/O is handled by threads. NEAT uses libuv as support library for asynchronous I/O. In order to also use this library for usrsctp the handling of one thread per socket had to be substituted by a single-threaded algorithm. An option was introduced to usrsctp to allow the user to either use thread support or not. In the latter case new API functions were also necessary to open the sockets from "outside" usrsctp. Now an IPv4 and an IPv6 raw socket are opened from NEAT at startup and their handling is handed over to libuv. A new timer is also managed by libuv that polls the SCTP event loop.

Whenever data arrive from the network, one of the sockets become readable. When the timer expires, libuv is informed and triggers a callback function in NEAT that calls a function in usrsctp to receive the data waiting at the socket.

The opposite direction, i.e., the sending of data is done directly by writing the complete message including the IP header to the socket.

UDP encapsulation: SCTP is not as widely deployed as TCP and UDP, therefore it can sometimes be blocked by middleboxes. To enable SCTP transport nevertheless, SCTP packets can be encapsulated in a UDP datagram and sent as UDP payload to the receiver.

To enable this feature in NEAT two more sockets (UDP/IPv4 and UDP/IPv6) have to be opened, which are handled in the same way as the raw sockets. One advantage of UDP encapsulation is that no root privileges are needed which is the case when raw sockets are used. That is the reason why often SCTP/UDP is preferred over pure SCTP.

The user can choose to send data over SCTP/UDP instead of SCTP by choosing SCTP/UDP as transport property. NEAT then opens a usrsctp socket and sets the socket option `SCTP_REMOTE_UDP_ENCAPS_PORT` to specify the UDP encapsulation port. Afterwards the connection is handled like a normal SCTP association.

Handling Upcalls: In addition to the sockets that handle the traffic between usrsctp and the network, a usrsctp socket is needed to manage the data transfer between NEAT and usrsctp. This interface is defined by the usrsctp socket API.

In the NEAT framework the sockets between NEAT and the transport layer are managed by libuv, i.e., libuv polls for a socket status change and calls a callback function whenever the socket becomes readable, writable or when an error occurs. Libuv expects sockets to be represented by file descriptors,

however, the opening of a usrsctp socket returns a pointer to a struct. Therefore, libuv cannot be used to handle usrsctp sockets. The communication between usrsctp and its upper layer is either handled by callback functions or directly by using send and receive calls whenever it seems adequate. In accordance with the libuv callback functionality we implemented a new upcall handling. An upcall function can be registered that is called whenever the socket becomes readable, writable or when an error occurs. To ensure stability the function is only triggered at the end of the handling of an incoming packet. Now, it is up to the user to call the usrsctp send and receive functions. The callback function has to handle all events concerning I/O processes and call the corresponding operations that were registered by the user.

Multihoming for SCTP: An outstanding feature of SCTP is the use of multiple paths for one association, called multihoming. This means that during one connection separate paths between the peers can be used either as backup in case of a path failure or for sharing the load between them.

A first step to support multihoming in NEAT for SCTP and usrsctp is to bind several addresses to one socket and thus reduce the number of candidates that have to be tested by Happy Eyeballs.

The candidates created by Happy Eyeballs are scanned and only those are taken where the source address corresponds to one of the addresses the user wants to send from. Then for each destination address all source addresses are assembled and bound with the `bindx` command which reduces the number of candidates to the number of destination addresses.

To enable the user to use multihoming and set the source addresses to be bound, two properties have been introduced, one to enable multihoming and one to set the addresses as a comma-separated list. The destination addresses can also be listed like this.

ICMPv4 and ICMPv6 Support: Using Happy Eyeballs implies that many potential connections might be tested for several protocols on different platforms. To speed up this process it is necessary to get the information immediately, whether the destination is available for a specific protocol or port or whether it is reachable at all.

To be able to interpret ICMP messages that are sent in case of a failure by the destination these messages must be handed over to the transport protocol. Special raw sockets for ICMPv4 and ICMPv6 have to be opened that can receive ICMP datagrams. These messages are forwarded to SCTP to be interpreted so that the adequate measures can be taken.

Support for ICMPv4 and ICMPv6 has been integrated in usrsctp, so that the upper layer is notified when a connection is aborted because of an ICMP *destination unreachable* message. For the integration in NEAT, two additional sockets have to be opened and handled by libuv, and the upcoming connection failure indication has to be interpreted correctly.

Examples of the operation: All existing examples in NEAT repository will use usrsctp instead of kernel SCTP, if the `USRSCTP_SUPPORT` option is set for NEAT. UDP encapsulation can be chosen by selecting SCTP/UDP as transport property. A new example program has been added to illustrate the use of multihoming, i.e., setting the necessary properties. The extended userland stack is available via https://github.com/NEAT-project/usrsctp-neat.

Provided Transport Service Feature(s): This building block is part of the most basic functionality of the NEAT System and it does not relate to any specific Transport Service Feature.

Related building blocks:

- NEAT Logic (§ 3.1.3).

3.2.2 Middlebox Traversal

To allow NEAT-based applications to operate also in a peer-to-peer scenario, middlebox traversal has to be considered. Modern Web Browsers support WebRTC, which allows a peer-to-peer communication supporting audio, video and data channels. The data channels provide a flexible message-oriented communication, allowing the user to control the reliability and message sequence preservation. DTLS is used to provide security. WebRTC uses STUN to determine the set of usable addresses, ICE to ensure reachability and TURN in case of very restrictive communication possibilities. These are the state of the art protocols for implementing middlebox traversal.

Therefore, it seems appropriate to integrate the data channel protocol stack used by WebRTC into NEAT. This provides the middlebox traversal capabilities and the flexibility of the data channels can be used to provide the required services for NEAT flows. Using the WebRTC data channel stack within NEAT should also allow NEAT applications to communicate with applications running in a modern Web Browser using WebRTC.

The work on integrating the WebRTC stack has started recently, but there is no running code yet.

Provided Transport Service Feature(s): There are no specific Transport Service Features associated to this building block.

Related building blocks:

- NEAT Logic (§ 3.1.3).
- Happy Eyeballs (§ 3.3.1).

3.2.3 Local flow scheduling priority

The NEAT System provides a way to prioritise the data transfer between NEAT Flows. This is performed by assigning different priorities to the flows belonging to a certain flow group.

The rationale behind using flow groups is to couple flows that might share a bottleneck and control their sending rate (either by scheduling or by setting the transport's congestion window) in such a way that they can achieve their share of bandwidth without necessarily competing with each other over the available capacity. The decision to choose a certain flow group is taken at the API level and by the user (e.g., for flows that share the same source and destinations).

The flow group is an integer-type number that can be assigned to a certain NEAT flow and is specified by the user as an optional argument tag of `NEAT_TAG_FLOW_GROUP`. NEAT flows with the same flow group will be coupled together and their respective data will be transferred according to their relative priority within the group, based on the underlying mechanism used at the transport protocol level. This priority is defined by a float-type optional argument tag of `NEAT_TAG_PRIORITY`. The optional arguments are set using the `NEAT_OPTARG_INT` and `NEAT_OPTARG_FLOAT` macros respectively and are passed to `neat_open` upon opening a NEAT flow.

Multiple flows can be created and be assigned the same flow group number. Let us assume that the kernel supports the Coupled Congestion Control (CCC) mechanism [17], and that Happy Eyeballs

returns a TCP connection for each flow. In this case, the priorities are used to couple the congestion controllers of NEAT flows (i.e., TCP connections) belonging to the same flow group by giving them sending rates (i.e., congestion windows, *cwnds*) proportional to their priorities. If the kernel does not support the CCC mechanism, or if Happy Eyeballs returns SCTP handles for some flows but TCP for others, the priorities are ignored by NEAT and the cwnd values are determined by each flow's congestion control mechanism individually. The CCC mechanism is already implemented for the FreeBSD kernel[9].

In case the handles returned by happy-eyeballing use SCTP, each NEAT flow would transparently map into a SCTP stream within a SCTP association. As a result all NEAT flows using the same SCTP association between the same end-points would use a common congestion controller. The priorities are therefore translated into per-stream SCTP scheduling priorities as defined in [24]. The integration of prioritised transfer for SCTP in NEAT is still an ongoing work in the project. Deliverable D3.2 [16] elaborates on the underlying CCC mechanisms used for flow-group prioritised transfer for TCP and SCTP.

Some examples of the operation: Assume a scenario where there are two flows, one with `NEAT_TAG_PRIORITY` of `0.75` starting at $t = 3$ s, and a second flow with `NEAT_TAG_PRIORITY` of `0.25` starting at $t = 30$ s, with both being assigned the same flow group `NEAT_TAG_FLOW_GROUP` as `1`.

To test this, we adapted the *tneat*[10] traffic-generation tool in order to accept flow group priority and number as command-line arguments as shown in Listing 15 (`-w` for priority and `-g` for flow group). Each tneat_fg instance corresponds to a single NEAT flow and generates traffic by sending a certain number of messages to a specified host (`-n` and `-l` specify the number and length of messages respectively).

```
1  #! /usr/bin/env sh
2  COUPLED_FLOWS=1
3
4  sleep 3
5  if [ $COUPLED_FLOWS == 1 ]; then
6      ./neat/examples/tneat_fg -p 5001 -g 1 -w 0.75 -l 1000 -n 100000 10.0.0.5 &
7      sleep 30
8      ./neat/examples/tneat_fg -p 5002 -g 1 -w 0.25 -l 1000 -n 15000 10.0.0.5 &
9  else
10     ./neat/examples/tneat_fg -p 5001 -g 0 -w 1.0 -l 1000 -n 115000 10.0.0.5 &
11 fi
12
13 sleep 120
14 echo "$0 DONE"
```

Listing 15: Shell script using tneat_fg command-line with different priorities in a flow group.

Listing 16 shows how the above flow group and priorities as well as CCC algorithm (`newreno_afse`) are passed to the NEAT System as optional arguments via `neat_open` in `tneat_fg.c`.

```
1 ...
2 static float config_priority = 1.0f;
```

[9] Kernel support code is available at https://naeemk@bitbucket.org/naeemk/freebsd11-ccc.git.

[10] Tneat is presented in Section 3.1.4 of Deliverable D4.1 [9]. Code for the adapted version of tneat currently resides in `neat/examples/tneat_fg.c` in the `oystedal/dscp` branch of the public NEAT code repository, at https://github.com/NEAT-project/neat.

```
3 static uint16_t config_group = 0;
4 ...
5 int
6 main(int argc, char *argv[])
7 {
8 ...
9 NEAT_OPTARGS_DECLARE(2);
10 NEAT_OPTARGS_INIT();
11 ...
12     while ((arg = getopt(argc, argv, "l:n:p:w:g:P:R:T:v:")) != -1) {
13         switch(arg) {
14         ...
15         case 'g':
16             config_group = atoi(optarg);
17             break;
18         case 'w':
19             config_priority = atof(optarg);
20             break;
21         ...
22         }
23     }
24     ...
25     if (config_active) {
26         if (config_group) {
27              NEAT_OPTARG_INT(NEAT_TAG_FLOW_GROUP, config_group);
28              NEAT_OPTARG_FLOAT(NEAT_TAG_PRIORITY, config_priority);
29              NEAT_OPTARG_STRING(NEAT_TAG_CC_ALGORITHM, "newreno_afse");
30         }
31         if (neat_open(ctx, flow, argv[optind], config_port, NEAT_OPTARGS,
               NEAT_OPTARGS_COUNT) == NEAT_OK) {
32             neat_start_event_loop(ctx, NEAT_RUN_DEFAULT);
33         } else {
34             fprintf(stderr, "%s - neat_open failed\n", __func__);
35             result = EXIT_FAILURE;
36             goto cleanup;
37         }
38     }
39 ...
40 cleanup:
41     if (ctx != NULL) {
42         neat_free_ctx(ctx);
43     }
44     exit(result);
45 }
```

Listing 16: Assigning flow group, priority and CCC selection in `tneat_fg.c`.

Figure 6 presents the results obtained with the example in Listing 15, using a real-life test on a 10 Mbps bottleneck link with RTT=100 ms (typical for access link scenarios), with a FreeBSD machine acting as the sender and using NewReno as the underlying TCP congestion control mechanism with

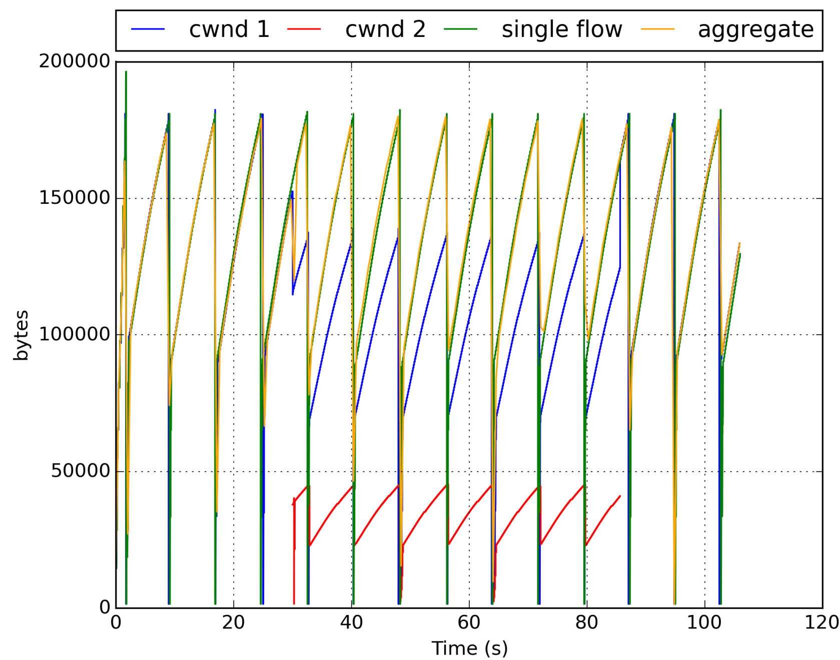


Figure 6: The *cwnd* (in bytes) plot of two TCP flows using coupled congestion control with priorities compared to a single TCP flow scenario. The *aggregate* line depicts the sum of cwnds in the two-flow scenario.

coupled congestion control enabled.

It can be seen in Figure 6 that between $t = 30$ s and $t = 85$ s (the duration when two flows coexist), flow 1 gets 3/4 of the cwnd while flow 2 gets 1/4 of the cwnd, on the average. It can also be seen that the two flows are coupled, as they increase and decrease their cwnds at (close to) the same time. Outside of this interval—when flow 2 either has not started yet or has terminated—flow 1 gets all of the cwnd.

Provided Transport Service Feature(s):

- NEAT flow group.
- NEAT flow priority.

Related building blocks:

- NEAT Logic (§ 3.1.3).

3.2.4 Security

The Security component of the NEAT System offers end to end transport security, including encryption, integrity, and authentication, for NEAT applications. Applications that, for reasons of backwards compatibility, cannot require full security may still utilize opportunistic security though this is not yet available.

The NEAT System provides secure connections using the TLS [11] and DTLS [23] protocols. TLS is used over TCP and DTLS for SCTP and UDP(-Lite). More precisely, the possible combinations that NEAT may provide are: TLS/TCP/IP, DTLS/UDP/IP, DTLS/SCTP/UDP/IP for a user-space SCTP stack, and DTLS/SCTP/IP for a kernel-level SCTP stack. The OpenSSL library [4] that offers TLS and DTLS

support is used in the current implementation, and TLS/TCP/IP is supported. The implementation of both DTLS/UDP/IP on all supported platforms and DTLS/SCTP/IP for a kernel-level SCTP stack on Linux and FreeBSD, are in progress. DTLS/SCTP/UDP/IP will be supported on FreeBSD as well while the SCTP/UDP/IP combination is provided by the FreeBSD kernel. DTLS/SCTP/UDP/IP for a user-space SCTP stack requires properties currently not supported by OpenSSL, i.e., the BIO memory access interface for OpenSSL [3] does not provide a message preserving memory access which is required by SCTP, furthermore OpenSSL provides an interface that requires a kernel socket.

The use of the Security building block depends on application requirements and configured policies. Transport security can be configured in one of the following ways:

- A secure connection is requested including a certificate verification.
- A secure connection is requested with an optional certificate verification: the Security component will perform a certificate verification, but even if it fails the NEAT Flow will consider the connection to be successfully established. An application can query the NEAT Flow to discover whether the certificate has been verified or not. This is currently not available, but it will be implemented as a part of this project.
- A secure connection is requested without a certificate verification: a certificate verification will not be performed. This is currently not available, but it will be implemented as a part of this project.
- A secure connection is optional: if a secure connection cannot be established the NEAT System will not return an error, instead it will establish a new insecure connection. This will add an additional delay. This is currently not available but it is supported by the design.
- A non-secure connection is requested: this option will not involve the Security component.

This corresponds to two NEAT Flow properties:

- *Use security:* required, optional, or do not use a secure connection.
- *Certificate verification:* must be verified, verification is optional, or do not verify. This property is only relevant if a secure connection is used.

A list of trusted Certification Authorities (CA) is currently hardcoded. Future work should enable a trusted list to be specified as a policy. The server certificates and private key files are set using `neat_secure_identity` and the function accepts only the PEM format [8, 18, 19, 21].

SSL3 and RC4 ciphers are not allowed because they do not provide enough security. TLS and DTLS can be limited to advertise and accept only certain TLS/DTLS versions and cipher suites—e.g., advertise only TLS 1.2 and the TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 cipher suite. The TLS/DTLS versions and cipher suites to be accepted will be defined as policies. To extend the flexibility of the NEAT System, an application can further limit or extend these lists for each individual request by setting the corresponding NEAT Flow property. This is currently not available.

Security is added in the NEAT architecture as an additional layer above the operating system socket. The NEAT Logic calls into the Security building block to perform data encryption and decryption before data is written to/read from an operating system socket. During a connection establishment the Security building block performs TLS or DTLS handshake and certificate verification depending on application requirements.

The NEAT System can be extended to use TCPINC [6] although this is not part of the project.

Some examples of the operation: The security component is used automatically by the NEAT Logic if a secure connection is required. An application needs to specify the corresponding property. Listing 17 shows a simple example of how to request a secure connection to a peer. This example is based on Listing 11 and an application only needs to change flow properties to request a secure connection.

```
1 int main(int argc, char *argv[])
2 {
3     if ((ctx = neat_init_ctx()) == NULL) {
4         debug_error("could not initialise context");
5         result = EXIT_FAILURE;
6         goto cleanup;
7     }
8
9     // new neat flow
10    if ((flow = neat_new_flow(ctx)) == NULL) {
11        debug_error("neat_new_flow");
12        result = EXIT_FAILURE;
13        goto cleanup;
14    }
15
16    char *config_property = "{\
17        \"transport\": [\
18            {\
19                \"value\": \"SCTP\",\
20                \"precedence\": 1\
21            },\
22            {\
23                \"value\": \"SCTP/UDP\",\
24                \"precedence\": 1\
25            },\
26            {\
27                \"value\": \"TCP\",\
28                \"precedence\": 1\
29            }\
30        ]\
31        \"security\": {\
32            \"value\": true,\
33            \"precedence\": 2\
34        }\
35    }";
36
37    // set properties
38    if (neat_set_property(ctx, flow, config_property)) {
39        fprintf(stderr, "%s - error: neat_set_property\n", __func__);
40        result = EXIT_FAILURE;
41        goto cleanup;
42    }
43   ...
```

Listing 17: Use of the NEAT Security componnent.

Algorithm 1 NEAT Happy Eyeballs Algorithm

```
1:  procedure HAPPYEYEBALLSCOMPONENT(in listOfCandidates : list of transport solutions)
2:      Require: listOfCandidates is sorted in priority order and len(listOfCandidates) > 0
3:      currentCandidate ← listOfCandidates.first()
4:      repeat
5:          if getPriority(currentCandidate) > 0 then
6:              delta ← convertToTimeInterval(getPriority(currentCandidate))
7:              scheduleAt( now() + delta, doAsynchConnectionAttempt(currentCandidate, connectionCallback))
8:          else
9:              doAsynchConnectionAttempt(currentCandidate, connectionCallback)
10:         end if
11:         currentCandidate ← listOfCandidates.nextCandidate(currentCandidate)
12:     until currentCandidate = endOfList(listOfCandidates)
13: end procedure

14: procedure CONNECTIONCALLBACK(in candidate : transport solution, out connection : transport connection)
15:     if connection ≠ NONE then
16:         policyManager.cacheResultConnectionAttempt(candidate, SUCCESS)
17:     else
18:         policyManager.cacheResultConnectionAttempt(candidate, FAILURE)
19:     end if
20: end procedure
```

Provided Transport Service Feature(s):

- NEAT flow security.

Related building blocks:

- NEAT Logic (§ 3.1.3).

3.3 NEAT Selection Components

The NEAT Selection components provide functions that map the requirements provided by the application to one or more transport endpoints and a set of transport components that can realise the required Transport Service. These functions are provided now by the Happy Eyeballs (§ 3.3.1) and Happy Apps (§ 3.3.2) components. While Happy Eyeballs provides the necessary transport selection functionalities at the transport layer using the information provided by the transport, Happy Apps provides a mean for selection of appropriate protocol and configurations using feedback from the application when such feedback is unavailable at the transport layer level.

3.3.1 Happy Eyeballs

The Happy Eyeballs (HE) building block is part of the NEAT User Module and comprises one of the NEAT Selection components. A description of the main parts of this building block is the subject of an IETF Internet Draft, authored by project participants [15].

In the first part of the NEAT selection process (i.e., steps 2–8 in Figure 4), the Policy Manager combines requirements from an application obtained through the NEAT User API, with available transport protocols, transport-protocol parameters, and feasible transport endpoints, i.e., IP addresses and port numbers. Together, they are used to create a list of candidate *transport solutions*. Each transport solution on the list has a priority. The priority is a positive integer with zero being the highest priority. The list is sorted in ascending order on the basis of the priority.

The pseudo-code for the Happy Eyeballs building block is presented in Algorithm 1. The component takes as input the list of candidate transport solutions created in the first part of the NEAT

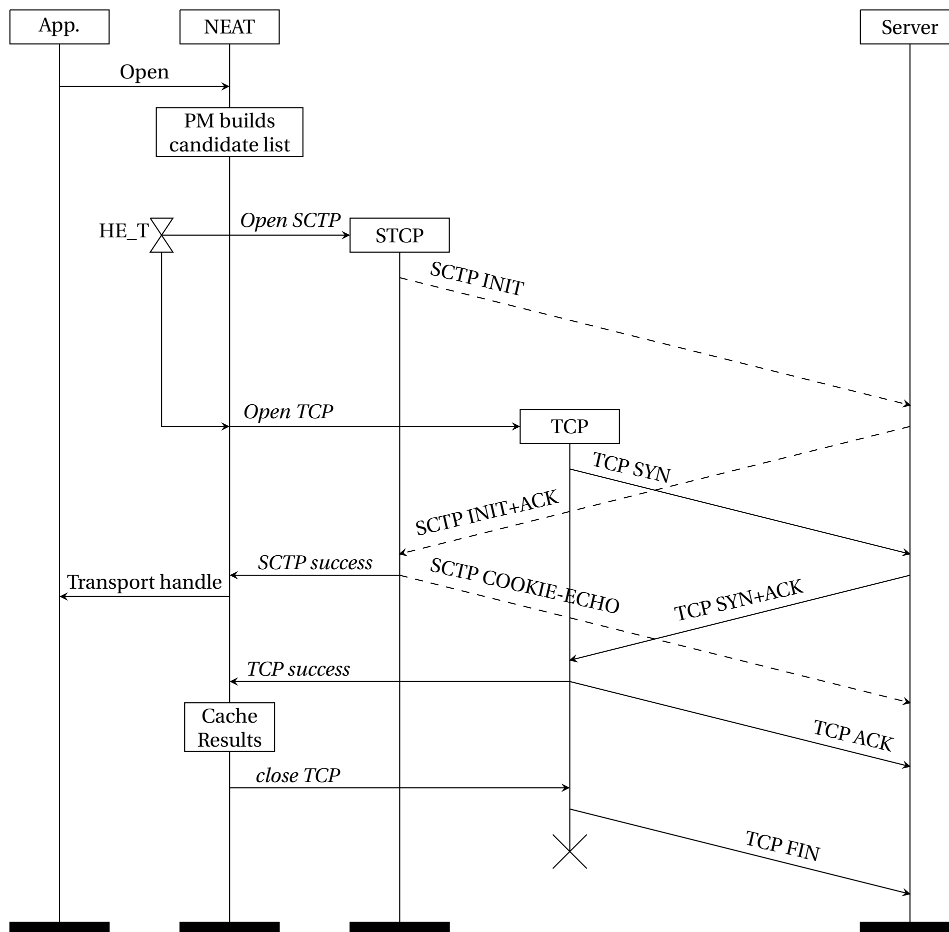


Figure 7: Message sequence chart illustrating the NEAT Happy Eyeballs transport selection process when selecting between TCP and SCTP, with SCTP preferred.

selection process. It concurrently tries out each of the candidate transport solutions in the list and finally returns a handle to the first successfully established connection.

As remarked in RFC 6555 [27], a Happy Eyeballs algorithm should not waste networking resources by routinely making simultaneous connection attempts. To this end, the Happy Eyeballs component instructs the Policy Manager to *cache* the outcome of previous connection attempts. Cached connection attempts are valid for a pre-set time after which they become invalid and have to be repeated. The caching lifetime is at present hardcoded to a fixed value, but in the final version of the library will be part of the system configuration. Our experimental work [22] (see Appendix D) suggests a significant reduction in terms of CPU load with caching. We observed that the CPU load decreases linearly with increasing cache hit-rate, and resulted in a more than 40% reduction of CPU load for unencrypted traffic and almost a 20% reduction for encrypted traffic. In terms of memory, our work reported in [22] concluded that Happy Eyeballs only has a marginal impact on kernel memory usage.

Some examples of the operation: As an example of how Happy Eyeballs works, consider the scenario illustrated in Figure 7. Both the client and server support TCP and SCTP. The Policy Manager

puts together a list of candidate transport solutions. The NEAT Logic calls Happy Eyeballs, which traverses the candidate list, and makes asynchronous connection attempts for each candidate transport solution. Since the TCP transport solution has a lower priority than the SCTP transport solution, the connection attempt for the TCP transport solution is delayed with respect to the one for SCTP. The length of the delay depends on the priority. At present, the delay is computed as a hardcoded value times the priority, however, eventually it will be dynamically adjusted. In the callback routine that is invoked when a connection attempt returns, the outcome of the connection attempt (success or failure) is cached by the Policy Manager through the Policy Interface. Since both connection attempts succeed, they are cached as successful connection attempts. Moreover, since the SCTP connection attempt completes before the TCP connection attempt, Happy Eyeballs returns the SCTP connection.

Provided Transport Service Feature(s):

- NEAT selected transport protocol.

Related building blocks:

- NEAT Logic (§ 3.1.3).
- NEAT Policy Manager (via Policy Interface) (§ 3.4.1).
- Connect to a name (§ 3.1.4).

3.3.2 Happy Apps (application-level feedback mechanisms)

Happy Apps (HA) is part of the NEAT User Module and offers selection mechanisms when the underlying transport protocol does not provide the signals required by the NEAT Logic. Non-connected network transports such as UDP and UDP-Lite do not provide mechanisms via the socket API to signal if traffic is making it to the end host, instead it is up to the application level protocol to decided if it is "happy" with the current flows provided by the NEAT System. The application is the only entity in a datagram transport using UDP(-Lite) that knows the reception state of the receiver, and hence whether a path is operating as required.

The Happy Apps mechanisms allow applications to take advantage of network properties that might not always work while still using the single NEAT User API.

Happy Apps can support fallback queries for different types of network issue, specified to NEAT with policy requests. Currently *QoS Fallback*, further explained below, is specified and implemented as a fallback mechanism in the NEAT System, but there are many possible Fallback Services an application could subscribe to. Some examples are:

- **Abstract QoS Fallback with DiffServ:** The NEAT System offers a mechanism that allows the application to signal its QoS requirements and match these to expected network support for QoS. This can allow NEAT policy to be utilised to select the most appropriate DSCP marking (or other QoS service). The mapping can take advantage of local configuration data, and QoS information provided by the network. By providing fallback through potential candidates it can help ensure that application traffic makes its way through the network with the most appropriate QoS available. This can be useful when a specific feature is blocked by some network boxes on the path.

Abstracting selection to the stack also allows the stack to utilise other path information that it gathers (via a CIB source utilising the Policy Manager) to help make this selection, and enables evolution of the selection process independent of the application code.

- **Custom SDN Provisioning Requests:** An application running in an SDN requires a large amount of bandwidth to perform a backup operation. Depending on network load, either the standard interface is used or a special link can be allocated. The application manages to maintain the required rate at the start of the session, but the bandwidth falls off. When the application signals it is "unhappy" to the NEAT System, the SDN Controller dynamically allocates an assured link for the application to use.

- **Firewall Traversal:** An application requires a service that is operated behind a firewall that only accepts admitted traffic. The application uses NEAT to connect to the firewall, but the traffic is stopped, NEAT fallbacks to a protocol that supports negotiation with the firewall and authenticates the application allowing traffic through.

The Happy Apps system is intended to be run throughout the application's lifetime; path changes that break certain network parameters can occur at any time. The NEAT System improves the API by giving applications a way to keep flows alive with mostly automatic fallback.

It is important to note that the Happy Apps mechanism may not be able to make any further changes to network parameters on behalf of the NEAT-based application, e.g., the application falls through all the selected QoS values without finding one that satisfies its requirements. In this case HA will remain in the final selected state until the application either decides to timeout and shutdown or it attempts to vary parameters itself.

Abstract QoS: NEAT provides mechanisms for transports to use QoS within the network. QoS is implemented in NEAT by setting the DSCP bits in the IP TOS field. Applications built with the socket API can access this field via a `setsockopt`, NEAT exposes the QoS field directly in the NEAT Flow and handles parameter setting in the socket stack on behalf of the application.

Applications using NEAT can set a DSCP value directly or they can use the *Abstract QoS* facilities offered by the NEAT System. NEAT offers abstract QoS types to give application developers access to QoS with a higher level API.

The abstract QoS values in NEAT allow application developers to request a QoS service which is comparable to the requirements of the application they are building. Rather than an application developer directly requesting a code point such as EF (Expedited Forwarding), with the NEAT System an application developer can request a high level `NEAT_QOS_AUDIO_H1` (a class representing high bandwidth audio traffic) and the NEAT System will preform the mapping down to the concrete QoS.

NEAT Abstract QoS uses the Policy Manager when mapping between the Abstract QoS value and the concrete DSCP mark to be used for the packet. Selection with the Policy Manager allows the NEAT System to offer dynamic mapping from abstract to concrete QoS while using information generated by other flows. Table 1 shows the default map from abstract QoS to concrete DSCP values in the NEAT System.

Some examples of the operation: As an example of how Happy Apps works, consider an application that requests QoS on a network path which drops all traffic marks with the AF4x DSCP. The NEAT application requests a NEAT Abstract QoS service of `NEAT_QOS_INTERACTIVE_VIDEO_H1` (Interactive

Table 1: Possible Abstract QoS to DSCP Mappings in NEAT. Some traffic classes such as Video can have several different capacity requirement levels, the NEAT System exposes these with Very Low, Low, Medium and High capacity requirements. Applications can also request Admitted access, classes that can be guaranteed by the network with policy or dynamic provisioning.

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

Video, High Bandwidth) and sets the `feedback_query` callback. The Policy Manager evaluates the requested QoS and generates a concrete mapping to a code point, for this example AF41.

The NEAT System automatically adds the QoS fallback property, and sets up the Happy Apps mechanism because the application has requested an abstract QoS type.

NEAT Logic sets the resolved DSCP on the socket underlying the application's NEAT flow. Each datagram generated by the application via `neat_write` will generate an IP packet with the specified DSCP set. The NEAT Application can now work through its own protocol, reading and writing against the NEAT flow as desired.

The `feedback_query` callback will be triggered by the NEAT System when the feedback query interval time expires, by the default policy triggers this after around 1 second. The NEAT Application must service the callback based on its own state. Because the network drops the traffic from the NEAT Application, the application cannot say it is "happy" with the current setup and returns `NEAT_ERROR`.

NEAT Logic receives the negative application feedback, the established DSCP mark is fed into the Policy Manager so that signal can be used to avoid this for any later connections. The NEAT Logic requests another concrete DSCP mark to use for the flow, in this example the next mark to try is AF31.

The application continues to perform writes with the new QoS value, this time the mark is able to transverse the network. The `feedback_query` callback is triggered again, this time the application is seeing return data from its peer and is able to signal success by returning `NEAT_OK`.

Provided Transport Service Feature(s):

- NEAT flow DSCP Support.

Related building blocks:

- NEAT Logic (§ 3.1.3).
- NEAT Policy Manager (via Policy Interface) (§ 3.4.1).

3.4 NEAT Policy Components

The NEAT Policy components allow the definition of policies which shape the attributes and constraints of each new given NEAT flow based on the properties requested by applications as well as known system and network characteristics. As a result the NEAT System can seamlessly adapt to a wide range of scenarios using administrative policies as well as information collected about the host and the attached networks.

The Policy components are comprised of the following building blocks, depicted in Fig. 8 and described in the sequel:

- Policy Manager (PM).
- Policy Information Base (PIB).
- Characteristics Information Base (CIB).

For each new connection requested by a NEAT-enabled application the PM is responsible for generating a list of candidate transport solutions which meet the requirements defined by the application. To this end, the PM takes into account all known information about the network stored in the CIB as well as any applicable policies defined in the PIB.

The CIB acts as a repository storing information about available interfaces, supported protocols, network properties and current/previous connections between endpoints (generated from passively or actively acquired network metrics—see CIB sources in § 3.4.3).

The PIB acts as a repository that contains a collection of policies, where each policy consists of a set of rules linking a set of matching requirements to a set of preferred or mandatory transport characteristics. Policies can be added by the system administrator, external entities or applications, and have different priorities. In addition to policies the PIB also contains so-called *profiles*, which are policies that are applied *before* the CIB lookup. Profiles are typically used to translate high-level properties into sets of concrete, low-level properties. Besides the time point of execution, policies and profiles are functionally identical.

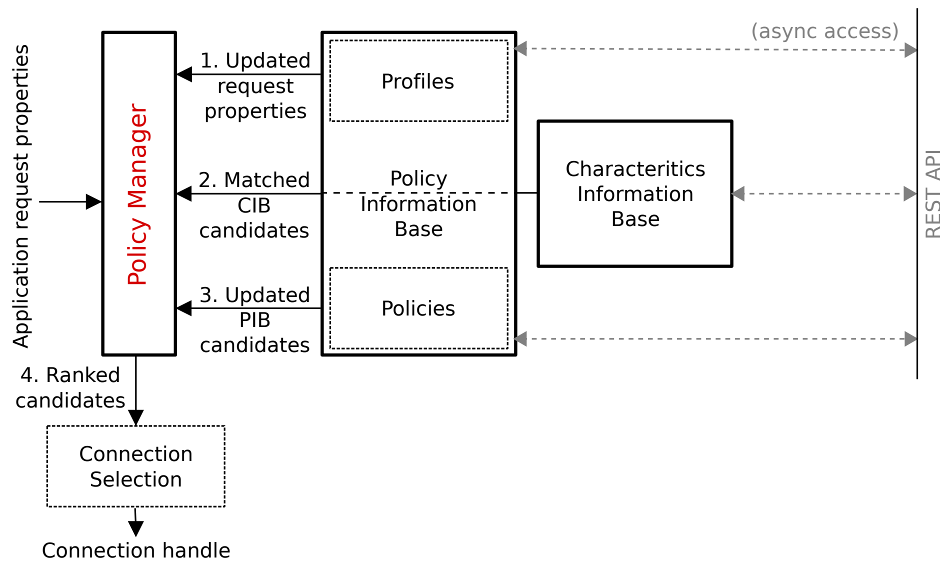


Figure 8: NEAT Policy components and their interactions.

The CIB and PIB entries are parsed to compute a list of potential candidates where each candidate contains an interface to be used, the transport protocol and associated options as well as other characteristics of the network.

The PM is responsible for implementing a strategy which (as far as possible) satisfies the given application requirements and installed policies while taking into account the knowledge about the available network resources. To achieve this the PM needs to prioritise and resolve any conflicting policies, adapting them to changing network information. The PM may optionally manage thresholds for when a CIB change triggers a new policy. Policies do not update within a NEAT Flow's lifetime, the PM is invoked only when a new NEAT Flow starts.

3.4.1 NEAT Policy Manager

The NEAT Policy Manager compiles all policies available at the host into a single set of valid rules. Conflicting policies are resolved based on the priority of categories: the application local policies over the global policies and external system policies over NEAT System policies. Global NEAT policies are combined during NEAT initialisation and serve as the default. Application local policies override the default policies only for the application by which these were installed. There is no lifetime associated with policies. They do not expire and have to be explicitly removed.

For each application the set of valid rules compiled from policies is static and will not be changed during runtime. The PM may optionally implement some level of access control. If more NEAT components require this functionality the addition of a NEAT-wide AAA building block may be considered. The NEAT Policy Manager can expose diagnostic information about the installed and/or active policies. These statistics may be bundled with output from the NEAT Flow Endpoint Statistics (§ 3.1.5), together with statistics from CIB, to enable useful debugging.

PM inputs: The PM requires three types of inputs:

- *Application properties:* an array of *NEAT properties* (see below) describing the attributes that a NEAT-enabled application desires for a newly opened NEAT Flow. The PM uses the approach

described in this section (under **Some examples of the operation**, below) to extract the most suitable candidates for the property requirements. The NEAT property array is passed to the Policy Manager by the NEAT Logic through the Policy Interface.

- *Policies:* they are installed by OS developers, vendors or applications into a predefined location in the file system, or transferred through the PM REST API. Profiles are a special type of policies which are used to expand high-level properties into more concrete properties *before* the CIB lookup. Policies and Profiles use an identical syntax using the JSON format [12] (the policy format is described in § 3.4.2).
- *CIB nodes:* they provide information about transport and interface characteristics in a predefined location in the file system or transferred through the PM REST API. Each CIB node is generated and maintained by so-called *CIB sources*, which may be system local or remote. The CIB repository is generated by all available CIB nodes which form a graph structure. Effectively the CIB expands the branches of this structure to create rows containing potential connection candidates.

PM outputs: The output of the PM is a ranked JSON list containing a set of candidate transport solutions and parameters for use by the NEAT Logic. Each candidate is an array of NEAT properties. Each candidate includes at least the local interface, the local and remote IP addresses and ports and the transport protocol. In addition, each candidate in the output list contains information indicating:

- Which application requirements (properties) are satisfied for the given interface/protocol/destination tuple.
- Which of the properties specified by the related evaluated policies have been verified and applied (added to the candidate).
- Which of the properties specified by the related evaluated policies the PM was not able to verify.

NEAT Properties: The NEAT policy system is based around the notion of NEAT properties. These are essentially `key|value` tuples describing attributes used by the components of the NEAT Policy Manager. NEAT properties are used to express attributes describing the local host and the attached networks, user requirements, or constraints imposed by policies. Examples include interface names, types, supported protocols and parameters, or topology metrics (e.g., available bandwidth).

Each property has a `key` string and a `value`. Currently property values can be

1. A *single* boolean, integer, float or string value, e.g., `2`, `true`, or `"TCP"`.
2. A *set* of values, e.g., `[100, 200, 300, "foo"]`.
3. A numeric *range*, e.g., `1-10`.

Each property is further associated with a `precedence` which identifies the *importance* of the property. Specifically, the precedence indicates if the property may be modified by the Policy Manager logic or if it is immutable. Currently two property precedence levels are defined in order of decreasing priority:

- `[immutable]`, precedence 2: these are mandatory properties whose value is required—e.g., by the application or a policy—and cannot be changed.

- `(requested)`, precedence 1: these are optional properties whose value may be overwritten. A mismatch of such properties will result in a penalty in the ranking within the PM. Such penalties are recorded as the score of the property.

A property's `score` attribute is used to indicate the weight of the property with respect to other properties. Whenever two properties are compared and match, the score of the result is the sum of the individual scores. As a consequence the PM can use the score to determine the most suitable NEAT connection `candidate` for a given request.

In the sequel we use the following shorthand notation: we separate the property key/value pair by the "|" character and indicate the property's precedence by the bracket types shown above. We append the score to the brackets and omit it if zero. For example

1. `[transport|TCP]+2` : the transport protocol *must* be TCP.
2. `(MTU|1500,2000,9000)` : one of the specified MTU values *should* be chosen if possible.
3. `(capacity|10-1000)+1` : the interface capacity *should* be within the numeric range specified by the integers.

For all externally facing interactions with the PM, NEAT properties are encoded using the JSON format. For example the following JSON object encodes an array containing the properties `[transport|TCP]+2`, `(MTU|1500,2000,9000)` and `(capacity|10-1000)+1`:

```
1 {
2 "domain_name" : {"value": "www.google.com", "precedence": 2},
3 "port": {"value": 80, "precedence": 2},
4 "local_interface": {"value": "eth0", "precedence": 2},
5 "local_ip": {"value": "10.10.2.14", "precedence": 2},
6 "transport": {"value": "TCP", "precedence":2, "score":2},
7 "MTU": {"value": [1500,2000,9000], "precedence":1},
8 "capacity": {"value": {"start":10, "end":1000}, "precedence":1, "score":1}
9 }
```

Listing 18: JSON encoded NEAT properties.

Property Operations: Any two NEAT properties are considered equal if their keys are identical and the intersection of their values is a non-empty set. Precedence and scores are ignored when testing for equality. A comparison of two properties yields a boolean result.

For instance, the comparison `[transport|TCP]+1 == (transport|TCP)+3` is true. Set and range value attributes are also considered equal if their values overlap, for example, `[transport|TCP,UDP,MPTCP] == [transport|TCP]`, or `[latency|1-100] == [latency|55]`.

In the course of a lookup in the PM, properties from various sources will be compared and their values may be *updated*. A property may only be updated by another property with the same key. A property's value will *only* be updated by another property whose precedence is greater or equal than itself—in which case it inherits the precedence of the updating property—and if both properties are not immutable (highest precedence). A property update is considered successful if the ranges of the associated properties overlap; the resulting updated property will contain the intersection of the two ranges.

If the above conditions are not satisfied, the update will fail and invalidate entire associated candidate.

As an example, if the immutable property `[transport|STCP]+2` is requested by an application and this property clashes with the immutable property `[transport|TCP]` in a certain connection candidate, the candidate will be discarded.

Policy Interface (PI): The Policy Interface exposes a set of programming function calls that NEAT components may invoke to make requests to the NEAT Policy Manager. The JSON format was selected for the Policy Interface to achieve a decoupling of the PM from the rest of the NEAT System. In fact, the PM components described in this section may become optional or may even be executed outside the host using them in a future version of the NEAT System, e.g., for less powerful mobile devices.

The communication interface between the Policy Manager and other NEAT components is currently implemented using Unix domain sockets. The socket `neat_pm_socket` is used to receive requests from the application and serve back a list of connection candidates to the NEAT Logic. The application request is passed as a string containing a JSON object with a set of NEAT properties, as illustrated in Listing 18. The request should at least contain a destination DNS name and port, or a destination IP, destination port, and a local interface and IP.

Similarly the PM response contains a configurable number of candidates encoded as a JSON list, wherein each list element is a JSON object containing the properties associated with a particular candidate. The list is order by the total score of all candidate properties.

Two additional Unix sockets, `neat_pib_socket` and `neat_cib_socket`, are available for adding new policies and CIB nodes to the PIB and CIB respectively (in addition to the filesystem interface).

Additionally the PM exposes a REST API which is intended to allow external applications, such as SDN controllers, to query the contents of the PIB/CIB and to populate these with new entries. If this optional API is started, the PM starts listening for HTTP connections on a predefined port (45888 by default). Applications may then access the following addresses using HTTP's GET/PUT semantics (using JSON):

/pib (GET) lists all policies installed in the host.

/pib/{uid} (GET/PUT) retrieve or upload a policy with a specific Unique Identifier (UID).

/cib (GET) lists all CIB nodes installed in the host.

/cib/{uid} (GET/PUT) retrieve or upload a CIB node with a specific UID.

/cib/rows (GET) retrieve all rows of the CIB repository.

Some examples of the operation: An example of the current Policy Manager workflow is given below.

1. The Policy Manager receives a query from the NEAT Logic through the Policy Interface. The query contains a list of properties (compiled in JSON format) originating from the application requirements and the NEAT Logic (e.g., DNS lookup result). An example is provided in Listing 19.

```
1 {
2     "domain_name": {"value": "www.debian.com", "precedence": 2 },
3     "port": { "value": 80, "precedence": 2 },
4     "local_ip": { "value": "10.73.64.61", "precedence": 2},
5     "interface": { "value": "en7", "precedence": 2},
```

```
6     "flow_size_bytes": { "value": 1000, "precedence": 2 },
7     "bulk_transfer": { "value": true },
8     "low_latency": { "value": false },
9     "transport": { "value": "reliable", "precedence": 2, "score": 2}
10 }
```

Listing 19: JSON encoded PM query.

2. The PM queries the PIB for profiles, which are used to map high level properties, such as `bulk_transfer|True`, to one or more concrete properties, such as `interface|eth0`, and `wireless|False`.

3. The PM performs a CIB lookup and receives an initial set of transport option candidates which (as far a possible) fulfil the query properties (i.e., by filtering the CIB by available interface types, path characteristics over relevant interfaces). Each candidate is a list containing the matched query properties as well as the associated properties of the potential connection (e.g., supported interface features, TCP variants, cached remote endpoint capabilities). See § 3.4.3 for more details.

4. For each candidate the PM performs a PIB lookup. The lookup yields a set of policies which match against the current candidate properties. Each policy may extend the candidate attribute list with additional optional (requested) properties. Further it may append mandatory (immutable) properties (e.g., “do not use 3G for bulk data”). See § 3.4.2 for more details.

5. The PM prunes all candidates which do not satisfy the required properties.

6. The PM ranks the candidates based on the total score of the individual candidate properties.

7. Finally the ranked candidate list is returned to the NEAT Logic. This information may be used by the NEAT Logic to carry out the final connection selection.

Provided Transport Service Feature(s): While it does not actively offer any Transport Service Feature, this building block is indirectly involved in providing many Transport Service Features by other building blocks—e.g., Selection of a secure interface, NEAT flow delay budget, NEAT flow low latency, etc. can be mapped to policy attributes and combined into rules to define appropriate policies.

Related building blocks:

- CIB (§ 3.4.3).
- PIB (§ 3.4.2).

3.4.2 Policy Information Base (PIB)

This building block defines the PIB repository that stores policies in the NEAT System and is accessed by the Policy Manager.

Essentially NEAT policies are based on the following logic:

MATCH <set of properties to match against> → PROPERTIES <set of mandatory or optional properties appended to flow candidate>.

The list of match conditions always implies that conditions are evaluated by performing an AND operation, i.e., all conditions must be true for the policy to be triggered. To define an OR relationship between conditions, multiple policies with a different set of conditions must be created.

A policy is defined in the JSON format [12] and typically contains the following attributes:

- `uid`: Each policy has a unique identifier to avoid conflicts and ambiguities.
- `description`: optional human-readable description of the policy.
- `priority`: integer for priority level of the policy. Higher number means higher priority.
- `match`: JSON object containing a set of properties to be matched against each candidate in order to trigger the policy. A policy is activated only if all of these properties successfully match properties in the corresponding candidate.
- `properties`: JSON object containing the properties that the policy aims to apply to the candidate. These properties may be optional of immutable. If the candidate cannot satisfy any one of the immutable properties in the list, it must be removed from the candidate list. If the PM cannot determine if a required attribute is satisfied, it must indicate this in the candidate list and let the NEAT Logic fall back to a default behaviour.

The policy UID is mandatory while the remaining attributes may be omitted if empty. A policy may optionally include attributes that will be defined in future versions. The JSON format offers a sufficient level of freedom to extend and modify the policy definition.

Policies may also be used to represent relationships between properties in a simpler and human-readable way. For example, `low_latency:true` implies `rtt_less_than:50` and `interface_latency_less_than:20`.

We consider the following policy categories, each forming a separate *PIB domain*:

- *Global NEAT:* generic set coming from NEAT operations. It can only change (extended in order not to break compatibility) at the next update of the NEAT System.
- *Application specific:* set by each application (e.g., Mozilla Firefox). It can only change at the next update or installation of the application which it belongs to.
- *Operating system specific:* set by OS (e.g., Linux, FreeBSD, ...). It can only change at the next update or installation of the OS.
- *Vendor specific:* set by the end-host manufacturer (e.g., a handset maker). It can only change at the next update of the firmware on the end host.

Users can override some of them by setting corresponding options in user API calls.

For persistence, in the current PIB implementation each policy is stored as a file in a predefined directory. Proposed locations for policy files in the Linux OS are:

- `/etc/neat/policy/OS/`
- `/etc/neat/policy/vendor/`
- `/etc/neat/policy/application/`
- `$HOME/.neat/policy/application/`

In other OSes similar paths where the user executing NEAT is allowed access will be used. The files are imported into the PIB and compiled into a repository which is managed by the PM. Note that the PIB information is not intended to be accessed directly by the NEAT Policy components. Read/write access to the PIB/CIB information is implemented using the existing OS user/access mechanisms, through Unix sockets, or through the PM REST API. Optionally the internal state of the PM may be exposed through an entry in `procfs` when available in Linux-like OSes (or a similar interface in other OSes), or appended to the PM output JSON structure.

Provided Transport Service Feature(s): There are no specific Transport Service Features associated to this building block.

Some examples of the operation: A sample of the proposed policy file format is given in Listing 20.

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

Listing 20: Policy file example.

Related building blocks:

- NEAT Policy Manager (§ 3.4.1).

3.4.3 Characteristics Information Base (CIB)

The Characteristics Information Base (CIB) is a repository that stores information about hosts (e.g., available interfaces, supported protocols), connections (e.g., parameters used by previously established transport sessions, hosts currently communicating) and the network (e.g., path properties). CIB

entries provide measured information, protocol details and capabilities about network entities used in the NEAT System, specifically in the policy decision phase.

The CIB is populated from multiple inputs generated by so-called *CIB sources*, such as components of the NEAT System or external applications. A *CIB source* is defined as any module which provides an input for the CIB in the correct format accepted by the CIB. Each CIB source generates *CIB nodes*, which are JSON data objects used to populate the CIB, comprised of a set of attributes for a given resource. CIB sources may populate the CIB through two methods: by creating files stored in a pre-defined folder in the host's filesystem, the location of which is OS dependent, or through a REST API listening on a port of the host.

Some mechanisms to populate the CIB are already implemented in OSes as statistics/measurement tools and will be made available as default CIB sources. Another class of CIB sources is provided by NEAT building blocks such as Happy Eyeballs (§ 3.3.1) which store discovered transport protocols and parameters supported along paths in the CIB for future reuse. External CIB sources may be provided as helper applications by device and OS vendors or third parties developing modules for active or passive measurements, statistics and metadata collection. The PM will use a pull mechanism to access any information stored in the CIB whenever a new NEAT Flow is initiated.

In the initial CIB implementation, trust is managed by relying on existing OS roles and permissions: CIB sources are allowed to create and update files in the CIB repository folder as long as the OS user executing the task is allowed to write in that folder. Future versions of the CIB may switch to a more complex authentication and trust management method.

CIB architecture: Conceptually, each entry or *row* of the CIB is comprised of an arbitrary number of *NEAT Properties*. Rows are composed from one or more *CIB nodes* represented as JSON objects—and, e.g., stored as files in the CIB directory or received through the REST API. These JSON objects are generated by various CIB sources. CIB nodes span a multitree structure, where each path from a tree root to a tree leaf constitutes a row of the CIB.

A minimal example of the CIB node defining a local interface is provided in Listing 21.

```
1 {
2     "uid": "eth0",
3     "root": true,
4     "priority": 4,
5     "properties": {
6         "interface": {"value": "eth0", "precedence":2},
7         "capacity": {"value": 10000, "precedence":2},
8         "local_ip": {"value": "10.10.2.1", "precedence":2},
9         "is_wired": {"value": true, "precedence":2},
10        "MTU": {"value": {"start":50, "end":9000}}
11    }
12 }
```

Listing 21: Example CIB node, showing the CIB format.

Each CIB node includes a `uid` key and a set of `properties` to be included in the CIB row.

Extending Existing CIB Nodes: Additionally, CIB sources may generate CIB nodes which reference and extend pre-existing CIB nodes. This feature is used to inject network characteristics collected by

external sources (e.g., controllers). For example, the following two CIB nodes, in Listings 22 and 23 reference the node defined in Listing 21:

```
{
    "uid": "eth0_remote_1",
    "description": "information about remote endpoint 1",
    "priority": 2,
    "link": true,
    "match" : [
      {"uid": {"value": "eth0"}}
    ],
    "properties": {
        "remote_ip": {"value": "8.8.8.8", "precedence":2, "score": 2},
        "remote_port": {"value": "80", "precedence":2, "score": 1}
    }
}
```

Listing 22: Example of CIB node referencing the node in Listing 21. Referencing is done via the `uid` value in the `match` attribute (`eth0`).

```
{
    "uid": "eth0_remote_2",
    "description": "information about remote endpoint 2",
    "priority": 2,
    "link": true,
    "match" : [
      {"interface": {"value": "eth0"}, "local_ip": {"value": "10.10.2.1"}}
    ],
    "properties": {
        "remote_ip": {"value": "8.8.4.4.", "precedence":2, "score": 2},
    }
}
```

Listing 23: Example of CIB node referencing the node in Listing 21. Referencing is done via the `interface` and `local_ip` properties in the `match` attribute (`eth0`).

If a CIB node contains a `link` attribute that is set to `true` the CIB will attempt to match any property or `uid` listed in the `match` attribute, and will append the new properties to the corresponding CIB node. Both examples above will be linked to the interface `eth0`, defined in the first CIB node. To reference multiple CIBs, the match attribute list can contain multiple JSON objects. The priority attribute is used to resolve overlapping properties.

Essentially, the CIB internally constructs a directed graph using all available CIB nodes. To generate the CIB rows, the graph is resolved starting at each node which has the attribute `root` set to `true` (see CIB node in Listing 21), generating paths (i.e., rows) by traversing the graph in the reverse direction of the edges.

Hence, from the above CIB nodes the CIB will generate the following two rows:

```

1: {"interface": {"value": "eth0", "precedence":2}, "capacity": {"value": 10000, "
    precedence":2}, "local_ip": {"value": "10.10.2.1", "precedence":2}, "is_wired":
    {"value": true, "precedence":2}, "MTU": {"value": {"start":50, "end":9000}, "
```

```
    remote_ip": {"value": "8.8.8.8", "precedence":2, "score": 2}, "remote_port": {"
    value": "80", "precedence":2, "score": 1}}
3
4 2: {"interface": {"value": "eth0", "precedence":2}, "capacity": {"value": 10000, "
    precedence":2}, "local_ip": {"value": "10.10.2.1", "precedence":2}, "is_wired":
    {"value": true, "precedence":2}, "MTU": {"value": {"start":50, "end":9000}, "
    remote_ip": {"value": "8.8.4.4.", "precedence":2, "score": 2}}
```

Listing 24: CIB rows generated from the CIB nodes defined in Listings 21 to 23.

Extending CIB Rows: Finally, CIB sources have the option of generating CIB nodes which augment existing CIB *rows* with additional properties. This can be useful to annotate specific CIB rows with historical information, e.g., collected from previous connections. To achieve this the `link` attribute is set to `false`:

```
1 {
2     "uid": "historic info",
3     "description": "appended properties to CIB rows matching all match properties.",
4     "priority": 10,
5     "timstamp": 1476104788,
6     "link": false,
7     "match": [
8       { "interface": {"value": "eth0"},
9          "local_ip": {"value": "10.10.2.1"},
10         "remote_ip": {"value": "8.8.8.8"}}
11      ],
12     "properties": [{
13         "remote_port": {"value": 8080, "precedence":1},
14         "local_port": {"value": 56674, "precedence":1},
15         "transport": {"value": "TCP", "precedence":1},
16         "cached": {"value": true, "precedence":2, "score":5},
17         "cache_ttl": {"value": 300, "precedence":1},
18         "cache_status": {"value": "connection_success", "precedence":2, "description
             ":"could be failed, NA, etc."}
19     }]
20 }
```

Listing 25: Extending CIB rows: any row matching all properties in the `match` attribute will be augmented with the properties in the `properties` attribute.

For this example the CIB will match row 1 in Listing 24 (i.e., the one with the `remote_ip` property set to `8.8.8.8`), and insert a new row which includes the additional properties.

```
1 3: {"interface": {"value": "eth0", "precedence":2}, "capacity": {"value": 10000, "
    precedence":2}, "local_ip": {"value": "10.10.2.1", "precedence":2}, "is_wired":
    {"value": true, "precedence":2}, "MTU": {"value": {"start":50, "end":9000}, "
    remote_ip": {"value": "8.8.8.8", "precedence":2, "score": 2}, "remote_port": {"
    value": 8080, "precedence":1}, "local_port": {"value": 56674, "precedence":1}, "
    transport": {"value": "TCP", "precedence":1}, "cache_ttl": {"value": 300, "
    precedence":1}, "cached_connection_status": {"value": "success", "precedence":2,
```

```
"score":5}}
```

Listing 26: CIB row 1 extended by the CIB node defined in Listing 25.

CIB nodes can be continuously updated—e.g., by a module monitoring the local OS capabilities or an external SDN controller. Further, whenever a new NEAT Flow is established, the Happy Eyeballs module caches the properties of the selected candidate by generating a corresponding CIB node referencing a local interface. These cache entries expire after a fixed time once the connection has been closed (time to live, TTL). The value of the TTL for remote CIB entries will be defined based on the initial experiences with the PM.

CIB lookup workflow: CIB lookup requests are arrays of NEAT properties including the address of the NEAT destination. Within the PM the request will contain the application requirements as well as any properties appended by the profile PIB. For each lookup request, the CIB compares the properties in the request against the properties contained in each row of the CIB. The lookup is successful whenever *all* required (precedence: 2) request properties match the properties in a CIB row. In this case, the properties of the corresponding CIB row are appended to the request, and the resulting property array becomes a candidate.

As a result, after the CIB lookup, the PM receives a list of candidates which fulfill the application requirements as well as the constraints and attributes imposed by the local system and the attached networks stored in the CIB. Candidates are ranked by the sum of the scores of the individual properties. However, only candidate properties which have been evaluated (i.e., matched) during the lookup are used to calculate the score, in order to ensure that properties about which nothing is known to the CIB do not influence the ranking.

Figure 4 illustrates how the CIB lookup process fits within the overall PM workflow.

Envisioned examples of NEAT-provided CIB sources include:

- Statistics and metadata provided by the operating system (e.g., network interface, socket statistics, battery drain, etc.).
- Statistics about path support from completed transport sessions and Happy Eyeballs (e.g., transport support and IP version).
- Path characteristics derived from various passive and active measurement techniques (e.g., network controller, network probes).
- Interface metadata (e.g., signal strength, type).

While some of these CIB sources have already been implemented (interface stats, network controller path information), additional CIB sources may be developed as NEAT is deployed in new scenarios.

Provided Transport Service Feature(s): There are no specific Transport Service Features associated to this building block.

Related building blocks:

- NEAT Policy Manager (§ 3.4.1).

4 NEAT reference material

A key element to a successful implementation project is proper documentation, ease of use of the code, and reference material like tutorials and pre-packaged examples. The NEAT consortium believes that the public community—e.g., network application developers, middleware developers, and students—should be able to learn how to use the NEAT library in the most simple and straightforward way. Therefore, the NEAT code is complemented by online documentation as well as compiling/installation files and instructions; also, API reference documentation as well as a tutorial have been made publicly available. In addition, a series of code examples can be found throughout this document with regards to each transport component. The NEAT documentation is maintained up-to-date as the library evolves using the *readthedocs* Content Management System [5].

4.1 NEAT tutorial

A tutorial on how to use the NEAT library, explaining the main concepts used in NEAT (e.g., flows, contexts, properties, etc.) and including a step-by-step minimal client and server example is available at http://neat.readthedocs.io/en/latest/tutorial.html. Section 2.1 of this document provides a copy of such tutorial at the time of writing.

4.2 Additional online documentation

The NEAT API reference documentation describes a set of API functions exposed to the user. It also explains the callback mechanism and the related choices used in NEAT, as well as the error codes, the optional arguments, and properties. It is available at http://neat.readthedocs.io/en/latest/index.html and a to-date instance of it is included in Appendix B of this document.

A reference to NEAT's coding style can be found at http://neat.readthedocs.io/en/latest/internal/codingstyle.html. It is based on the coding style used in the Linux kernel.

4.3 Virtual machines

To enable interested developers to quickly evaluate NEAT, the NEAT library, including installation instructions, sample test applications and documentation, is provided within ready-to-deploy virtual machine (VM) environments. These NEAT VMs support both VMware and VirtualBox hypervisors and provide a self-contained environment showcasing the NEAT library. Each VM contains a pre-configured version of the NEAT library, built from a recent snapshot of the Github repository. NEAT VMs are being made publicly available to download via https://www.neat-project.org/resources/ to facilitate usage and adoption by interested developers. Users can evaluate and test NEAT in a safe and supported machine and they can clone it as many times as needed to simulate multiple hosts in complex topologies. The list of OS distributions (Long Term Support versions are preferred) that will be made available through the NEAT VM repository is listed in Table 2.

Each VM contains a recent binary version of the NEAT library that has successfully passed the automated Buildbot tests[11]. The Policy Manager is configured to run as a daemon, using a minimal set of pre-configured policies and CIB entries. In addition, the VMs contain the NEAT sources in the directory `~/neat` which is linked to the NEAT Github repository enabling users to available to pull

[11] These are described in detail in Deliverable D4.1 [9].

Table 2: OS distributions used for NEAT VMs.

Operating System	**Version**
Ubuntu	16.04 LTS
Debian	9.0 (stretch)
FreeBSD	11
NetBSD	7.0.2

updated development code or push contributions. The images also include all package dependencies required to build and debug NEAT, as well as compiler suites and example NEAT code.

5 Conclusions

This document has presented the NEAT Core Transport System; the components necessary to provide NEAT Transport Services in the API described in Deliverable D1.2 [26]. These core components are grouped into four classes of building blocks: Framework, Transport, Selection and Policy. This was done as part of the design and development efforts undertaken in Tasks 2.1 and 2.2.

In Section 1 we provided a short overview of the NEAT architecture (§ 1.1) and elaborated on the Transport Services provided by the NEAT User API (§ 1.2). We also introduced core transport components required to provide such Transport Services (§ 1.3). Finally, we provided a step-by-step summary of the NEAT operation workflow, from the moment a connection setup attempt is made at the API level by the application until the connection handle is returned to the application (§ 1.4).

Section 2 presented a detailed tutorial on how to use the NEAT User API which contains a full example of a client/server application using NEAT (§ 2.1). In addition, we summarised how easier and simpler network programming becomes using the NEAT User API in contrast to the traditional Berkeley socket API (§ 2.2). Built-in functionalities underneath the NEAT User API that are transparent to the application programmer and are supported across multiple OS platforms (e.g., name resolution, transport protocol selection, and fallback mechanisms) allow a much simpler approach to network programming than the traditional socket API. Use of the callback-based NEAT API allows for a more streamlined code in network applications.

In Section 3 we discussed in detail each of the core components, and explained how they operate and identified their internal dependencies as implemented in the NEAT library, using examples of use or operation and code snippets in the C language.

Lastly, Section 4 presented the available documentation of the NEAT core transport system—this includes an online NEAT tutorial, the NEAT API reference, and VMs pre-packaged with NEAT support.

Work Package 2 of the NEAT Project has finalised the first prototype implementation of the core transport system building blocks described in this document (milestone MS8). The consortium will continue to adjust this prototype to enhance its performance and functionalities during the third year of the project. Task 2.3 will address any required minor adjustments to match the final services and APIs derived in Task 1.4 of WP1. The final outcome of such efforts will be reported in Deliverable D2.3.

In summary, thanks to their platform- and protocol-independent nature, the NEAT core transport system and its User API as outlined in this document provide a simple, flexible, and easy way for application programmers to take advantage of the transport services provided by Internet transport protocols.

References

[1] Filestreamer. [Online]. Available: https://github.com/oystedal/filestreamer

[2] libuv library. [Online]. Available: http://libuv.org/

[3] Openssl bio_s_mem manual. [Online]. Available: https://wiki.openssl.org/index.php/Manual:BIO_s_mem(3)

[4] OpenSSL library. [Online]. Available: https://www.openssl.org/

[5] Read the docs. [Online]. Available: https://readthedocs.org

[6] TCP increased security working group. [Online]. Available: https://datatracker.ietf.org/wg/tcpinc/charter/

[7] usrsctp — a portable sctp userland stack. [Online]. Available: https://github.com/sctplab/usrsctp/

[8] D. Balenson, "Privacy Enhancement for Internet Electronic Mail: Part III: Algorithms, Modes, and Identifiers," RFC 1423 (Historic), Internet Engineering Task Force, Feb. 1993. [Online]. Available: http://www.ietf.org/rfc/rfc1423.txt

[9] Z. Bozakov, S. Mangiante, A. Brunstrom, D. Damjanovic, G. Fairhurst, A. Hansen, T. Jones, N. Khademi, A. Petlund, , D. Ros, D. Stenberg, M. Tüxen, and F. Weinrank, "NEAT-based applications and first version of NEAT-based tools," The NEAT Project (H2020-ICT-05-2014), Deliverable D4.1, Mar. 2017.

[10] V. Cerf, Y. Dalal, and C. Sunshine, "Specification of Internet Transmission Control Program," RFC 675 (Historic), Internet Engineering Task Force, Dec. 1974, obsoleted by RFC 7805. [Online]. Available: http://www.ietf.org/rfc/rfc675.txt

[11] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2," RFC 5246 (Proposed Standard), Internet Engineering Task Force, Aug. 2008, updated by RFCs 5746, 5878, 6176, 7465, 7507, 7568, 7627, 7685, 7905, 7919. [Online]. Available: http://www.ietf.org/rfc/rfc5246.txt

[12] ECMA. (2013) The JSON data interchange format. [Online]. Available: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

[13] G. Fairhurst, B. Trammell, and M. Kuehlewind, "Services Provided by IETF Transport Protocols and Congestion Control Mechanisms," RFC 8095 (Informational), Internet Engineering Task Force, Mar. 2017. [Online]. Available: http://www.ietf.org/rfc/rfc8095.txt

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

[15] K.-J. Grinnemo, A. Brunstrom, P. Hurtig, and N. Khademi, "Happy Eyeballs for Transport Selection," Internet Draft draft-grinnemo-taps-he-02, Feb. 2017, work in progress. [Online]. Available: https://tools.ietf.org/html/draft-grinnemo-taps-he-02

[16] K.-J. Grinnemo, A. Brunstrom, G. Fairhurst, D. Hayes, P. Hurtig, N. Khademi, D. Ros, I. Rüngeler, M. Tüxen, F. Weinrank, and M. Welzl, "Final Report on Transport Protocol Enhancements," The NEAT Project (H2020-ICT-05-2014), Deliverable D3.2, Mar. 2017.

[17] S. Islam, M. Welzl, K. Hiorth, D. Hayes, Ø. Dale, G. Armitage, and S. Gjessing, "How to Control a TCP: Minimally-Invasive Multi-Flow Congestion Management," Under submission, Feb. 2017.

[18] B. Kaliski, "Privacy Enhancement for Internet Electronic Mail: Part IV: Key Certification and Related Services," RFC 1424 (Historic), Internet Engineering Task Force, Feb. 1993. [Online]. Available: http://www.ietf.org/rfc/rfc1424.txt

[19] S. Kent, "Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management," RFC 1422 (Historic), Internet Engineering Task Force, Feb. 1993. [Online]. Available: http://www.ietf.org/rfc/rfc1422.txt

[20] N. Khademi, D. Ros, M. Welzl, Z. Bozakov, A. Brunstrom, G. Fairhurst, K.-J. Grinnemo, D. Hayes, P. Hurtig, T. Jones, S. Mangiante, M. Tüxen, and F. Weinrank, "NEAT: A Platform- and Protocol-Independent Internet Transport API," *IEEE Communications Magazine*, Feb. 2017, to appear.

[21] J. Linn, "Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures," RFC 1421 (Historic), Internet Engineering Task Force, Feb. 1993. [Online]. Available: http://www.ietf.org/rfc/rfc1421.txt

[22] G. Papastergiou, K.-J. Grinnemo, A. Brunstrom, D. Ros, M. Tüxen, N. Khademi, and P. Hurtig, "On the cost of using Happy Eyeballs for transport protocol selection," in *Proceedings of the 2016 Applied Networking Research Workshop (ANRW)*, Berlin, Jul. 2016, pp. 45–51.

[23] E. Rescorla and N. Modadugu, "Datagram Transport Layer Security Version 1.2," RFC 6347 (Proposed Standard), Internet Engineering Task Force, Jan. 2012, updated by RFCs 7507, 7905. [Online]. Available: http://www.ietf.org/rfc/rfc6347.txt

[24] R. Stewart, M. Tuexen, S. Loreto, and R. Seggelmann, "Stream Schedulers and User Message Interleaving for the Stream Control Transmission Protocol," Internet Draft draft-ietf-tsvwg-sctp-ndata, Oct. 2016, work in progress. [Online]. Available: https://tools.ietf.org/html/draft-ietf-tsvwg-sctp-ndata-08

[25] 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, Mar. 2017, work in progress. [Online]. Available: https://tools.ietf.org/html/draft-ietf-taps-transports-usage-03.txt

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

[27] 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

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 NEAT API Reference

B.1 Optional arguments

Some of the functions in the NEAT API, such as `neat_open`, `neat_read` and `neat_write`, take optional arguments. These are sometimes used to pass optional arguments to functions, and sometimes used to return additional values from the function. Optional arguments are passed as an array of the struct `neat_tlv` and an integer specifying the length of this array. `neat_tlv` is defined as follows:

```
struct neat_tlv {
    neat_tlv_tag  tag;
    neat_tlv_type type;

    union {
        int   integer;
        char *string;
        float real;
    } value;
};
```

Listing 27: `neat_tlv` struct.

An optional argument takes the form of a tag name, a type, and a value of either a string, integer or a floating point number. The tag specifies which optional argument the value belongs to, and the type asserts the type of the value passed as this argument. An error will be raised of the type is different than what the function expects.

You may either work with this struct directly, or you may use the preprocessor macros explained later in this document.

B.1.1 Specifying no optional arguments

To specify no optional arguments, simply pass `NULL` as the `optargs` parameter and `0` as the `opt_count` parameter of the function.

B.1.2 Optional argument macros

These are the optional argument macros used in NEAT:

- `NEAT_OPTARGS_DECLARE(max)`: Declare the necessary variables to use the rest of these macros. Allocates (on the stack) an array of length max and an integer for storing the number of optional arguments specified. `NEAT_OPTARGS_MAX` may be used as the default array size.
- `NEAT_OPTARGS_INIT()`: Initializes the variables declared by `NEAT_OPTARGS_DECLARE`. May also be used to reset the (number of) optional arguments back to `0`.
- `NEAT_OPTARG_INT(tag,value)`: Specify the tag and the value of an optional argument that takes an integer.
- `NEAT_OPTARG_FLOAT(tag,value)`: Specify the tag and the value of an optional argument that takes a floating point number.

- NEAT_OPTARG_STRING(tag,value): Specify the tag and the value of an optional argument that takes a string.
- NEAT_OPTARGS: Represents the array of optional arguments. Specify this macro as the optarg parameter.
- NEAT_OPTARG_COUNT: Stores the number of the optional arguments specified so far with NEAT_OPTARG_INT, NEAT_OPTARG_FLOAT or NEAT_OPTARG_STRING. This count is reset by NEAT_OPTARGS_INIT(). Specify this macro as the opt_count argument.

B.1.3 Optional argument tags

These are the optional argument tags used in NEAT:

- NEAT_TAG_STREAM_ID (integer): Specifies the ID of the stream which the data should be written to, or which stream the data was read from.
- NEAT_TAG_STREAM_COUNT (integer): Specifies the number of stream to create. Only used with protocols that support multiple streams.
- NEAT_TAG_FLOW_GROUP (integer): Specifies the flow group this flow belongs to.
- NEAT_TAG_PRIORITY (float): Specifies the priority of this flow relative to other flows in the flow group.
- NEAT_TAG_CC_ALGORITHM (string): Specifies the name of the (TCP) congestion control algorithm that will be used by this flow. A system default will be used if the specified algorithm is not available.

Currently unused tags:

- NEAT_TAG_LOCAL_NAME
- NEAT_TAG_SERVICE_NAME
- NEAT_TAG_CONTEXT
- NEAT_TAG_PARTIAL_RELIABILITY_METHOD
- NEAT_TAG_PARTIAL_RELIABILITY_VALUE
- NEAT_TAG_PARTIAL_MESSAGE_RECEIVED
- NEAT_TAG_PARTIAL_SEQNUM
- NEAT_TAG_UNORDERED
- NEAT_TAG_UNORDERED_SEQNUM
- NEAT_TAG_DESTINATION_IP_ADDRESS

Example:

```
1   NEAT_OPTARGS_DECLARE(NEAT_OPTARGS_MAX);
2   NEAT_OPTARGS_INIT();
3   NEAT_OPTARG_INT(NEAT_TAG_STREAM_COUNT, 5);
4   neat_open(ctx, flow, "127.0.0.1", 8000, NEAT_OPTARGS, NEAT_OPTARGS_COUNT);
```

B.2 Properties

A property in NEAT may either express a requirement or it may express a desire from the application with regards to the service provided by the transport layer.

A property takes the form of a JSON object. A set of properties is contained within one JSON object. Below is an example of a JSON object with one property:

```
{
    "property_name": {
        value: "property_value",
        precedence: 1
    }
}
```

Note that all examples of properties will be specified inside a JSON object.

Properties have a name, a value, and a precedence. A string is always used for the name of a property. The value of a property may be either a boolean, a string, an integer, a floating point number, an array, or an interval. Each property expects only one specific type.

The properties are sent to the Policy Manager (if present), which will return a list containing a list of candidates, which is ordered by how good the candidate matches the request from the application. Each candidate specifies a given setting for each property. NEAT will use the properties specified in each candidate when trying to set up a new connection.

Some properties are set by NEAT based on parameters to function calls. Other properties must be set manually with the `neat_set_property` function.

B.2.1 Application property reference

These are properties that may be set by the application.

transport Type: Array

Specifies an array of transport protocols that may be used. An application may specify either one protocol with precedence 2, or multiple protocols with precedence 1.

Note: May not be queried with `neat_get_property` before execution of the `on_connected` callback. When querying this property, the returned value is a string describing the actual transport in use.

Note: Applications should avoid specifying the protocol(s) to use directly, and instead rely on the Policy Manager to make a decision on what protocol(s) to use based on other properties. The transport property should only be used for systems without a Policy Manager, or if the choice of transport protocol is strictly mandated by the application protocol.

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

```
12 }
```

Listing 28: Example 1: multiple protocols

```
1 {
2     "transport": [
3         {
4             "value": "UDP",
5             "precedence": 2
6         }
7     ]
8 }
```

Listing 29: Example 2: one protocol

Available protocols:

- SCTP
- SCTP/UDP (SCTP tunneled over UDP)
- TCP
- UDP
- UDP-Lite

Security Type: Boolean

Specifies whether the connection should be encrypted or not. With precedence 2, NEAT will only report the connection as established if it was able to connect and the (D)TLS handshake succeeds. With precedence 1, NEAT may still attempt to establish an unencrypted connection.

B.2.2 Inferred properties

These are properties that are inferred during connection setup and subsequently sent to the Policy Manager. Applications should not set these directly.

interfaces Type: Array

Specifies a list of available interfaces that may be used for connections. The Policy Manager may not use all interfaces in this list.

This property is inferred during the `neat_open` call. Do not set this property manually.

domain_name Type: String

Specifies the name of the remote endpoint to connect to with the `neat_open` call.

This property is inferred from the name parameter of `neat_open` call. Do not set this property manually.

port Type: Integer

This property is inferred from the `neat_open` and `neat_accept` calls. Do not set this property manually.

B.3 Callbacks

Callbacks are used in NEAT to signal events to the application. They are used to inform the application when a flow is readable, writable, or an error has occurred.

Most callbacks have the following syntax:

```
1 neat_error_code
2 on_event_name(neat_flow_operations *ops)
3 {
4     return NEAT_OK; // or some error code
5 }
```

Listing 30: NEAT callback syntax.

The struct `neat_flow_operations` is defined as follows:

```
1 struct neat_flow_operations
2 {
3     void *userData;
4 
5     neat_error_code status;
6     int stream_id;
7     struct neat_ctx *ctx;
8     struct neat_flow *flow;
9 
10    neat_flow_operations_fx on_connected;
11    neat_flow_operations_fx on_error;
12    neat_flow_operations_fx on_readable;
13    neat_flow_operations_fx on_writable;
14    neat_flow_operations_fx on_all_written;
15    neat_flow_operations_fx on_network_status_changed;
16    neat_flow_operations_fx on_aborted;
17    neat_flow_operations_fx on_timeout;
18    neat_flow_operations_fx on_close;
19    neat_cb_send_failure_t on_send_failure;
20    neat_cb_flow_slowdown_t on_slowdown;
21    neat_cb_flow_rate_hint_t on_rate_hint;
22 };
```

Listing 31: `neat_flow_operations` struct.

- `userData`: Applications may freely store a pointer in this field.
- `status`: Reports any errors associated with the flow.
- `stream_id`: For flows that use explicit multi-streaming. Specifies which stream the event is related to, if any.
- `ctx`: Pointer to the context the flow belongs to.
- `flow`: Pointer to the flow on which the event happened.

Callbacks are set by assigning the function pointer to the struct passed to the callback and then calling `neat_set_operations`. A NULL pointer may be used to indicate that the callback should no longer be called.

B.3.1 Example callback flow

For most applications it will be sufficient to use the following callback flow as in Figure 5. See the tutorial in Section 2.1 for more details.

B.3.2 Callback reference

Here is a list of callbacks used by NEAT.
`on_connected`: Called whenever an outgoing connection has been established with `neat_open`, or an incoming connection has been established with `neat_accept`.

`on_error`: Called whenever an error occurs when processing the flow. Errors are considered critical.

`on_readable`: Called whenever the flow can be read from without blocking. NEAT does not permit blocking reads.

`on_writable`: Called whenever the flow can be written to without blocking. NEAT does not permit blocking writes.

`on_all_written`: Called when all previous data sent with `neat_write` has been completely written. Does not signal that the flow is writable. Applications may use this callback to re-enable the `on_writable` callback.

`on_network_status_changed`: Inform application that something has happened in the network. This also includes flow endpoints going up, which will subsequently trigger `on_connected` if that callback is set. *Only available when using SCTP.*

`on_aborted`: Called when the remote end aborts the flow. Available for flows using TCP or SCTP.

`on_timeout`: Called if sent data is not acknowledged within the time specified with `neat_change_timeout`. *Currently only available for TCP on Linux.*

`on_close`: Called when the graceful connection shutdown has completed. Only available when using SCTP or TCP. Note that when using TCP, this callback is called when the `close()` system call is made, as TCP implementations currently does not provide any more accurate way of signalling this.

`on_send_failure`: Defined as:

```
1 void
2 on_send_failure(struct neat_flow_operations *flowops, int context, const unsigned
    char *unsent)
3 {
4 }
```

Called to inform the application that the returned message `unsent` could not be transmitted. The failure reason as reported by the transport protocol is returned in the standard status code, as an abstracted NEAT error code. If the message was tagged with a context number, it is returned in context. Only available for SCTP. Flows using TCP may use timeouts instead.

`on_slowdown`: Not currently implemented. Defined as:

```
1 void
2 on_slowdown(struct neat_flow_operations *ops, int ecn, uint32_t rate)
3 {
4 }
```

Inform the application that the flow has experienced congestion and that the sending rate should be lowered. If `rate` is non-zero, it is an estimate of the new maximum sending rate. `ecn` is a boolean indicating whether this notification was triggered by an ECN mark.

`on_rate_hint`: Not currently implemented.

Defined as:

```
void
on_rate_hint(struct neat_flow_operations *ops, uint32_t new_rate)
{
}
```

Called to inform the application that it may increase its sending rate. If `new_rate` is non-zero, it is an estimate of the maximum sending rate.

B.4 Error codes

These are the error codes used by NEAT library.

- `NEAT_OK`: Signals that no error has occurred. Equals to `0`.
- `NEAT_ERROR_WOULD_BLOCK`: Signals that the operation could not be performed because it would block the process. NEAT does not permit blocking operations.
- `NEAT_ERROR_BAD_ARGUMENT`: Signals that one or more arguments given to the function was invalid or incorrect. This also includes optional arguments.
- `NEAT_ERROR_IO`: Signals that an internal I/O operation in NEAT has failed.
- `NEAT_ERROR_DNS`: Signals that there was an error performing DNS resolution.
- `NEAT_ERROR_INTERNAL`: Signals that there was an error internally in NEAT.
- `NEAT_ERROR_SECURITY`: Signals that there was an error setting up an encrypted flow.
- `NEAT_ERROR_UNABLE`: Signals that NEAT is not able to perform the requested operation.
- `NEAT_ERROR_MESSAGE_TOO_BIG`: Signals that the provided buffer space is not sufficient for the received message.
- `NEAT_ERROR_REMOTE`: Signals that there was an error on the remote endpoint.
- `NEAT_ERROR_OUT_OF_MEMORY`: Signals that NEAT is not able to allocate enough memory to complete the requested operation.

B.5 API functions

B.5.1 neat_init_ctx

- Syntax:

```
struct neat_ctx *neat_init_ctx();
```

- Parameters: None.

- Return values: Returns a pointer to a newly allocated and initialized NEAT context. Returns NULL if a context could not be allocated.
- Remarks: None.
- Examples: None.
- See also: `neat_free_ctx` and `neat_new_flow`

B.5.2 neat_free_ctx

- Syntax:

```
1 void neat_free_ctx(struct neat_ctx *ctx);
```

- Parameters: `ctx`: Pointer to the NEAT context to free.
- Return values: None
- Remarks: If there are any flows still kept in this context, those will be freed and closed as part of this operation.
- Examples: None.
- See also: `neat_close` and `neat_init_ctx` and `neat_new_flow`

B.5.3 neat_new_flow

- Summary: Allocate and initialize a new NEAT flow.
- Syntax:

```
1 neat_flow *neat_new_flow(struct neat_ctx *ctx);
```

- Parameters: `ctx`: Pointer to a NEAT context.
- Return values: Returns a pointer to a new flow. Returns NULL on error.
- Remarks: None.
- Examples: None.
- See also: `neat_open` and `neat_close`

B.5.4 neat_set_property

- Summary: Set the properties of a NEAT flow.
- Syntax:

```
1 neat_error_code neat_set_property(
2     struct neat_ctx *ctx,
3     struct neat_flow *flow,
4     const char *properties);
```

- Parameters:
  - `ctx`: Pointer to a NEAT context.
  - `flow`: Pointer to a NEAT flow.
  - `properties`: Pointer to a JSON-encoded string containing the flow properties.
- Return values: Returns `NEAT_OK` if the properties were set successfully. Returns `NEAT_ERROR_BAD_ARGUMENT` if the JSON-encoded string is malformed.
- Remarks: Properties are applied when a flow connects.
- Examples: None.
- See also: None.

B.5.5 neat_get_property

- Summary: Query the properties of a flow. Returns value only, not precedence.
- Syntax:

```
1 neat_error_code neat_get_property(struct neat_ctx *ctx,
2                                   struct neat_flow *flow,
3                                   const char *name,
4                                   void *ptr,
5                                   size_t *size);
```

- Parameters:
  - `ctx`: Pointer to a NEAT context.
  - `flow`: Pointer to a NEAT flow.
  - `name`: Name of the property to query.
  - `ptr`: Pointer to a buffer where the property value may be stored.
  - `size`: Pointer to an integer containing the size of the buffer pointed to by `ptr`. Updated to contain the size of the property upon return.
- Return values: Returns `NEAT_OK` if the property existed and there was sufficient buffer space available. The size parameter is updated to contain the actual size. Returns `NEAT_ERROR_MESSAGE_TOO_BIG` if there was not sufficient buffer space. The size parameter is updated to contain the required buffer size. Returns `NEAT_ERROR_UNABLE` if the property does not exist.
- Remarks: Applications may pass `0` as the size parameter to query the size of the property.
- Examples:

```
1     size_t bufsize = 0;
2     char buffer = NULL;
3
4     if (neat_get_property(ctx, flow, "transport", buffer, &bufsize) ==
          NEAT_ERROR_MESSAGE_TOO_BIG) {
```

```
5         buffer = malloc(bufsize);
6         if (buffer && neat_get_property(ctx, flow, "transport", buffer, &
              bufsize) == NEAT_OK) {
7             printf("Transport: %s\n", buffer);
8         }
9         if (buffer)
10            free(buffer);
11    } else {
12        printf("\tTransport: Error: Could not find property \"transport\"\n");
13    }
```

- See also: Properties and `neat_set_property`

B.5.6 neat_open

- Summary: Open a neat flow and connect it to a given remote name and port.
- Syntax:

```
1 neat_error_code neat_open(struct neat_ctx *ctx,
2                           struct neat_flow *flow,
3                           const char *name,
4                           uint16_t port,
5                           struct neat_tlv optional[],
6                           unsigned int opt_count);
```

- Parameters:
  - `ctx`: Pointer to a NEAT context.
  - `flow`: Pointer to a NEAT flow.
  - `name`: The remote name to connect to.
  - `port`: The remote port to connect to.
  - `optional`: An array containing optional parameters.
  - `opt_count`: The length of the array containing optional parameters.
- Optional parameters:
  - `NEAT_TAG_STREAM_COUNT` (integer): The number of streams to open, for protocols that supports multistreaming
  - `NEAT_TAG_FLOW_GROUP` (integer): The group ID that this flow belongs to. For use with coupled congestion control.
  - `NEAT_TAG_PRIORITY` (float): The priority of this flow relative to the other flows. Must be between 0.1 and 1.0.
  - `NEAT_TAG_CC_ALGORITHM` (string): The congestion control algorithm to use for this flow.
- Return values: Returns `NEAT_OK` if the flow opened successfully. Returns `NEAT_ERROR_OUT_OF_MEMORY` if the function was unable to allocate enough memory.

- Remarks: Callbacks can be specified with `neat_set_operations`. The `on_connected` callback will be invoked if the connection established successfully. The `on_error` callback will be invoked if NEAT is unable to connect to the remote endpoint.
- Examples:

```
1 neat_open(ctx, flow, "bsd10.fh-muenster.de", 80, NULL, 0);
```

- See also: `neat_read` and Optional arguments

B.5.7 `neat_accept`

- Summary: Listen to incoming connections on a given port on one or more protocols.
- Syntax:

```
1 neat_error_code neat_accept(struct neat_ctx *ctx,
2                             struct neat_flow *flow,
3                             uint16_t port,
4                             struct neat_tlv optional[],
5                             unsigned int opt_count);
```

- Parameters:
  - `ctx`: Pointer to a NEAT context.
  - `flow`: Pointer to a NEAT flow.
  - `port`: The local port to listen for incoming connections on.
  - `optional`: An array containing optional parameters.
  - `opt_count`: The length of the array containing optional parameters.
- Optional parameters:
  - `NEAT_TAG_STREAM_COUNT` (integer): The number of streams to accept, for protocols that supports multistreaming.
- Return values: Returns `NEAT_OK` if NEAT is listening for incoming connections on at least one protocol. Returns `NEAT_ERROR_UNABLE` if there is no appropriate protocol available for the flow properties that was specified. Returns `NEAT_ERROR_BAD_ARGUMENT` if flow is pointing to a flow that is already opened or listening for incoming connections. Returns `NEAT_ERROR_BAD_ARGUMENT` if `NEAT_TAG_STREAM_COUNT` is less than `1`. Returns `NEAT_ERROR_OUT_OF_MEMORY` if the function was unable to allocate enough memory.
- Remarks: Callbacks can be specified with `neat_set_operations`. The `on_connected` callback will be invoked if the connection established successfully. The `on_error` callback will be invoked if NEAT is unable to connect to the remote endpoint. Which protocols to listen to is determined by the flow properties.
- Examples:

```
1 neat_accept(ctx, flow, 8080, NULL, 0);
```

- See also: `neat_open` and Optional arguments

B.5.8 neat_read

- Summary: Read data from a neat flow. Should only be called from within the `on_readable` callback specified with `neat_set_operations`.
- Syntax:

```
1 neat_error_code neat_read(struct neat_ctx *ctx,
2                           struct neat_flow *flow,
3                           unsigned char *buffer,
4                           uint32_t amount,
5                           uint32_t *actual_amount,
6                           struct neat_tlv optional[],
7                           unsigned int opt_count);
```

- Parameters:
  - `ctx`: Pointer to a NEAT context.
  - `flow`: Pointer to a NEAT flow.
  - `buffer`: Pointer to a buffer where read data may be stored.
  - `amount`: The size of the buffer pointed to by buffer.
  - `actual_amount`: The amount of data actually read from the transport layer.
  - `optional`: An array containing optional parameters.
  - `opt_count`: The length of the array containing optional parameters.
- Optional parameters: This function uses optional parameters for some return values.
  - `NEAT_TAG_STREAM_ID` (integer): The ID of the stream will be written to this parameter.
- Return values: Returns `NEAT_OK` if data was successfully read from the transport layer. Returns `NEAT_ERROR_WOULD_BLOCK` if this call would block. Returns `NEAT_ERROR_MESSAGE_TOO_BIG` if the buffer is not sufficiently large. This is only returned for protocols that are message based, such as UDP, UDP-Lite and SCTP. Returns `NEAT_ERROR_IO` if the connection is reset.
- Remarks: This function should only be called from within the `on_readable` callback specified with `neat_set_operations`, as this is the only way to guarantee that the call will not block. NEAT does not permit a blocking read operation. The `actual_amount` value is set to `0` when this function returns error.
- Examples: None.
- See also: `neat_write` and Optional arguments

B.5.9 neat_write

- Summary: Write data to a neat flow. Should only be called from within the `on_writable` callback specified with `neat_set_operations`.
- Syntax:

```
1 neat_error_code neat_write(struct neat_ctx *ctx,
2                            struct neat_flow *flow,
3                            const unsigned char *buffer,
4                            uint32_t amount,
5                            struct neat_tlv optional[],
6                            unsigned int opt_count);
```

- Parameters:
  - ctx: Pointer to a NEAT context.
  - flow: Pointer to a NEAT flow.
  - buffer: Pointer to a buffer containing the data to be written.
  - amount: The size of the buffer pointed to by buffer.
  - optional: An array containing optional parameters.
  - opt_count: The length of the array containing optional parameters.
- Optional parameters:
  - NEAT_TAG_STREAM_ID (integer): The ID of the stream the data will be written to.
- Return values: Returns NEAT_OK if data was successfully written to the transport layer. Returns NEAT_ERROR_BAD_ARGUMENT if the specified stream ID is negative. Returns NEAT_ERROR_OUT_OF_MEMORY if NEAT is unable to allocate memory. Returns NEAT_ERROR_WOULD_BLOCK if this call would block. Returns NEAT_ERROR_IO if an I/O operation failed.
- Remarks: This function should only be called from within the on_writable callback specified with neat_set_operations, as this is the only way to guarantee that the call will not block. NEAT does not permit a blocking write operation. Invalid stream IDs are silently ignored.
- Examples: None.
- See also: neat_read and Optional arguments

B.5.10 neat_shutdown

- Summary: Initiate a graceful shutdown of this flow. All previously written data will be sent. Data can still be read from the flow.
- Syntax:

```
1 neat_error_code neat_shutdown(struct neat_ctx *ctx,
2                               struct neat_flow *flow);
```

- Parameters:
  - ctx: Pointer to a NEAT context.
  - flow: Pointer to the NEAT flow to be shut down.
- Return values: Returns NEAT_OK if the flow was shut down successfully. Returns NEAT_ERROR_IO if NEAT was unable to shut the flow down successfully.

- Remarks: None.
- Examples: None.
- See also: `neat_close`

B.5.11 `neat_close`

- Summary: Close this flow and free all associated data.
- Syntax:

```
1 neat_error_code neat_close(struct neat_ctx *ctx,
2                            struct neat_flow *flow);
```

- Parameters:
  - `ctx`: Pointer to a NEAT context.
  - `flow`: Pointer to the NEAT flow to be closed.
- Return values: Returns `NEAT_OK`.
- Remarks: None.
- Examples: None.
- See also: `neat_shutdown`

B.5.12 `neat_abort`

- Summary: Abort this flow and free all associated data.
- Syntax:

```
1 neat_error_code neat_abort(struct neat_ctx *ctx,
2                            struct neat_flow *flow);
```

- Parameters:
  - `ctx`: Pointer to a NEAT context.
  - `flow`: Pointer to the NEAT flow to be aborted.
- Return values: Returns `NEAT_OK`.
- Remarks: Calls `neat_close` internally.
- Examples: None.
- See also: `neat_shutdown` and `neat_close`

B.5.13 `neat_set_operations`

- Syntax:

```
neat_error_code neat_set_operations(
    struct neat_ctx *ctx,
    struct neat_flow *flow,
    struct neat_flow_operations *ops);
```

- Parameters:
  - `ctx`: Pointer to a NEAT context.
  - `flow`: Pointer to a NEAT flow.
  - `ops`: Pointer to a struct that defines the operations/callbacks for this flow.
- Return values: Returns `NEAT_OK`.
- Remarks: struct `neat_flow_operations` is defined as follows:

```
struct neat_flow_operations
{
    void *userData;

    neat_error_code status;
    int stream_id;
    neat_flow_operations_fx on_connected;
    neat_flow_operations_fx on_error;
    neat_flow_operations_fx on_readable;
    neat_flow_operations_fx on_writable;
    neat_flow_operations_fx on_all_written;
    neat_flow_operations_fx on_network_status_changed;
    neat_flow_operations_fx on_aborted;
    neat_flow_operations_fx on_timeout;
    neat_flow_operations_fx on_close;
    neat_cb_send_failure_t on_send_failure;
    neat_cb_flow_slowdown_t on_slowdown;
    neat_cb_flow_rate_hint_t on_rate_hint;

    struct neat_ctx *ctx;
    struct neat_flow *flow;
};
```

- Examples:

```
struct neat_flow_operations ops;
ops.on_readable = on_readable;
ops.on_writable = on_writable;
neat_set_operations(ctx, flow, ops);
```

B.5.14 neat_change_timeout

- Summary: Change the timeout of the flow. Data that is sent may remain un-acked for up to a given number of seconds before the connection is terminated and a timeout is reported to the application.
- Syntax:

```
1 neat_error_code
2 neat_change_timeout(struct neat_ctx *ctx, struct neat_flow *flow,
3                     unsigned int seconds);
```

- Parameters:
  - `ctx`: Pointer to a NEAT context.
  - `flow`: Pointer to a NEAT flow.
  - `seconds`: The number of seconds after which un-acked data will cause a timeout to be reported.
- Return values: Returns `NEAT_OK` if the timeout was successfully changed. Returns `NEAT_ERROR_UNABLE` if attempting to use this function on a system other than Linux, or on flow that is not using TCP. Returns `NEAT_ERROR_BAD_ARGUMENT` if the timeout value is too large or if the specified flow is not opened. Returns `NEAT_ERROR_IO` if NEAT was unable to set the timeout.
- Remarks: Only available on Linux for flows using TCP.
- Examples: None.

B.5.15 neat_set_primary_dest

- Summary: For multihomed flows, set the primary destination address.
- Syntax:

```
1 neat_error_code neat_set_primary_dest(struct neat_ctx *ctx,
2                                       struct neat_flow *flow,
3                                       const char* address);
```

- Parameters:
  - `ctx`: Pointer to a NEAT context.
  - `flow`: Pointer to a NEAT flow.
  - `address`: The remote address to use as the primary destination address.
- Return values: Returns `NEAT_OK` if the primary destination address was set successfully. Returns `NEAT_ERROR_UNABLE` if the flow is not using SCTP as the transport protocol. Returns `NEAT_ERROR_BAD_ARGUMENT` if the provided address is not a literal IP address.
- Remarks: Currently only available for SCTP.
- Examples: None.

B.5.16 `neat_secure_identity`

- Summary: Specify a certificate and key to use for secure connections.
- Syntax:

```
1 neat_error_code neat_secure_identity(
2     struct neat_ctx *ctx,
3     struct neat_flow *flow,
4     const char *filename);
```

- Parameters:
  - `ctx`: Pointer to a NEAT context.
  - `flow`: Pointer to a NEAT flow.
  - `filename`: Path to the PEM file containing the certificate and key.
- Return values: Returns `NEAT_OK`.
- Remarks: None.
- Examples: None.

B.5.17 `neat_set_checksum_coverage`

- Summary: Set the checksum coverage for messages sent or received on this flow.
- Syntax:

```
1 neat_error_code neat_set_checksum_coverage(
2     struct neat_ctx *ctx,
3     struct neat_flow *flow,
4     unsigned int send_coverage,
5     unsigned int receive_coverage);
```

- Parameters:
  - `ctx`: Pointer to a NEAT context.
  - `flow`: Pointer to a NEAT flow.
  - `send_coverage`: UDP-Lite: The number of bytes covered by the checksum when sending messages. UDP: Ignored.
  - `receive_coverage`: UDP-Lite: The lowest number of bytes that must be covered by the checksum on a received message. UDP: See below.
- Return values: Returns `NEAT_OK` if the checksum coverage was set successfully. Returns `NEAT_ERROR_UNABLE` if the checksum coverage cannot be set, either because the value is invalid, or because the protocol does not support it.
- Remarks: Only available for flows using UDP or UDP-Lite. Checksum verification may be enabled/disabled on the receive side for flows using UDP. Specifying a non-zero value for `receive_coverage` will enable it; specifying `0` will disable it.
- Examples: None.

B.5.18 neat_set_qos

- Summary: Set the Quality-of-Service class for this flow.
- Syntax:

```
1 neat_error_code neat_set_qos(struct neat_ctx *ctx,
2                              struct neat_flow *flow,
3                              uint8_t qos);
```

- Parameters:
  - ctx: Pointer to a NEAT context.
  - flow: Pointer to a NEAT flow.
  - qos: The QoS class to use for this flow.
- Return values: Returns NEAT_OK if the QoS class was set successfully. Returns NEAT_ERROR_UNABLE if NEAT was not able to set the requested QoS class.
- Remarks: None.
- Examples: None.

B.5.19 neat_set_ecn

- Summary: Set the Explicit Congestion Notification value for this flow.
- Syntax:

```
1 neat_error_code neat_set_ecn(struct neat_ctx *ctx,
2                              struct neat_flow *flow,
3                              uint8_t ecn);
```

- Parameters:
  - ctx: Pointer to a NEAT context.
  - flow: Pointer to a NEAT flow.
  - ecn: The ECN value to use for this flow.
- Return values: Returns NEAT_OK if the QoS class was set successfully. Returns NEAT_ERROR_UNABLE if NEAT was not able to set the requested ECN value.
- Remarks: None.
- Examples: None.

B.5.20 neat_start_event_loop

- Summary: Starts the internal event loop within NEAT.
- Syntax:

```
1 neat_error_code neat_start_event_loop(struct neat_ctx *ctx, neat_run_mode
      run_mode);
```

- Parameters:
  - ctx: Pointer to a NEAT context.
  - run_mode: The mode of which the event loop in NEAT should execute. May be one of either NEAT_RUN_DEFAULT, NEAT_RUN_ONCE, or NEAT_RUN_NOWAIT.
- Return values: Returns NEAT_OK if the NEAT executed with no error. Returns an error value if the internal event loop in NEAT was stopped due to an error.
- Remarks: This function does not return when executed with NEAT_RUN_DEFAULT. When executed with NEAT_RUN_ONCE, NEAT will poll for I/O, and then block *unless* there are pending callbacks within NEAT that are ready to be processed. These callbacks may be internal. When executed with NEAT_RUN_NOWAIT, NEAT will poll for I/O and execute any pending callbacks. If there are no pending callbacks, it returns after polling.
- Examples: None.
- See also: neat_stop_event_loop and neat_get_backend_fd

B.5.21 neat_stop_event_loop

- Summary: Stops the internal NEAT event loop.
- Syntax:

```
1 int neat_stop_event_loop(struct neat_ctx *ctx);
```

- Parameters:
  - ctx: Pointer to a NEAT context.
- Return values: None.
- Remarks: Once called, no further events will be processed and no callbacks will be called until neat_start_event_loop is called again.
- Examples: None.
- See also: neat_start_event_loop

B.5.22 neat_get_backend_fd

- Syntax:

```
1 int neat_get_backend_fd(struct neat_ctx *ctx);
```

- Parameters:
  - ctx: Pointer to a NEAT context.
- Return values: Returns the file descriptor of the event loop used internally by NEAT. May be polled to check for any new events.
- Remarks: Note that embedding this event loop inside another event loop may not be supported on all systems.
- Examples: None.
- See also: neat_start_event_loop

B.5.23 neat_get_backend_timeout

- Summary: Return the timeout that should be used when polling the backend file descriptor.
- Syntax:

```
1 int neat_get_backend_timeout(struct neat_ctx *ctx);
```

- Parameters:
  - ctx: Pointer to a NEAT context.
- Return values: Returns the number of milliseconds on which a poll operation may at most be blocked on the backend file descriptor from libuv before the NEAT event loop should be executed again to take care of timer events within NEAT.
- Remarks: The client_http_run_once example demonstrates the use of this function.
- Examples: None.
- See also: neat_get_backend_fd

B.5.24 neat_get_stats

- Summary: Return statistics from a NEAT context.
- Syntax:

```
1 neat_error_code neat_get_stats(
2     struct neat_ctx *ctx,
3     char **json_stats);
```

- Parameters:
  - ctx: Pointer to a NEAT context.

  - json_stats: Pointer to an address where address of the statistics may be written.
- Return values: Returns NEAT_OK.
- Remarks: The statistics is output in JSON format. The caller is responsible for freeing the buffer containing the statistics.
- Examples: None.

B.5.25 neat_log_level

- Summary: Set the log-level of the NEAT library.
- Syntax:

```
1 void neat_log_level(uint8_t level)
```

- Parameters:
  - level: Log level of the log entry
    * NEAT_LOG_OFF
    * NEAT_LOG_ERROR
    * NEAT_LOG_WARNING
    * NEAT_LOG_ERROR
    * NEAT_LOG_DEBUG
- Return values: None.
- Examples:

```
1 neat_log_level(NEAT_LOG_ERROR);
```

- See also: neat_log_file

B.5.26 neat_log_file

- Summary: Sets the name of the log file.
- Syntax:

```
1 uint8_t neat_log_file(const char* file_name)
```

- Parameters:
  - file_name: Name of the NEAT logfile. If set to NULL, NEAT writes the log output to stderr.
- Return values: RETVAL_SUCCESS: success; RETVAL_FAILURE: failure
- Examples:

```
1 neat_log_file("disaster.log");
```

- See also: neat_log_level

C Paper: *NEAT: A Platform- and Protocol-Independent Internet Transport API*

The following research paper [20] has been produced by project participants, and has been *accepted for publication* in IEEE Communications Magazine.

Preprint (accepted version)

To appear in IEEE Communications Magazine
http://www.comsoc.org/commag/

Until published, please cite as:

N. Khademi, D. Ros, M. Welzl, Z. Bozakov, A. Brunstrom, G. Fairhurst, K.-J. Grinnemo, D. Hayes, P. Hurtig, T. Jones, S. Mangiante, M. Tüxen, and F. Weinrank. "NEAT: A Platform- and Protocol-Independent Internet Transport API". *IEEE Communications Magazine*, accepted for publication, March 2017.

© 2017 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

IEEE COMMUNICATIONS MAGAZINE, VOL. X, NO. X, JUNE 2017 2

NEAT: A Platform- and Protocol-Independent Internet Transport API

Naeem Khademi, David Ros, Michael Welzl, Zdravko Bozakov, Anna Brunstrom, Gorry Fairhurst, Karl-Johan Grinnemo, David Hayes, Per Hurtig, Tom Jones, Simone Mangiante, Michael Tüxen, and Felix Weinrank

***Abstract*—The sockets Applications Programming Interface (API) has become the standard way that applications access the transport services offered by the Internet Protocol stack. This paper presents NEAT, a user-space library that can provide an alternate transport API. NEAT allows applications to request the service they need using a new design that is agnostic to the specific choice of transport protocol underneath. This not only allows applications to take advantage of common protocol machinery, but also eases introduction of new network mechanisms and transport protocols. The paper describes the components of the NEAT library and illustrates the important benefits that can be gained from this new approach. NEAT is a software platform for developing advanced network applications that was designed in accordance with the standardization efforts on Transport Services (TAPS) in the Internet Engineering Task Force (IETF), but its features exceed the envisioned functionality of a TAPS system.**

I. Introduction

For more than three decades, the Internet's transport layer has essentially supported just two protocols and the original design of the sockets API offered only two Transport Services to applications. One service provided stream-oriented in-order reliable delivery, manifested in TCP, and the other a message-based unordered unreliable delivery, manifested in UDP.

Today, more than three decades later, these are the only two transport protocols commonly offered by operating systems to applications. UDP-based applications are used for a wide variety of datagram services from service discovery to interactive multimedia, while TCP became the dominant protocol for Internet services from web browsing to file sharing and video content delivery. While their success has often been attributed to the robustness of these protocols, during the last decades new service requirements have emerged that are beyond what TCP can deliver or UDP can offer—examples include: an interactive multimedia application may prefer to prioritize low latency over strictly reliable delivery of data, but could use partially-reliable delivery to improve quality while ensuring timeliness, or an application may be designed to take advantage of multihoming when this is available. UDP has also emerged as a substrate upon which user-space transport protocols are being developed—many customized for specific applications (e.g., the QUIC protocol), where much effort can be expended re-implementing common transport functions.

A handful of protocols have been proposed to provide Transport Services beyond those of TCP and UDP; most notably, SCTP, DCCP and UDP-Lite. However none of these have seen widespread use or universal deployment. The reason behind this is often attributed to *ossification* of the Internet's transport layer, where further evolution has become close to impossible. This has two major aspects:

- **Inflexibility of the current socket API:** Application programmers need to specify *transport protocol-specific* configurations to request a desired service. This binding to protocols inevitably requires programmers to recode their applications to take advantage of any new transport protocol. It also introduces complexity when there is a need to customize for different network scenarios, and choose appropriate transport protocol-specific parameters.
- **Deployment vicious circle:** New protocols and mechanisms cannot be expected to work in unmodified networks. Some equipment may need to be reconfigured, updated or replaced to deploy a new protocol. Developers seeking to use new protocols simply find they cannot be relied upon to work across the Internet. Because the current socket API requires application developers to specifically choose a certain protocol, they therefore tend to avoid using a protocol other than TCP or UDP, knowing that any others are likely to be unsuccessful for many network paths. This chicken-and-egg situation has made it hard for unused transport protocols to become deployed in the Internet—even if they would provide a better service to some applications.

In this paper, we introduce the *NEAT Library*. This is a software library built above the socket API to provide networking applications with a new API offering platform- and protocol-independent access to Transport Services. NEAT is, to the best of our knowledge, the first prototype implementation of IETF standardization efforts on Transport Services (TAPS), which we will discuss in Section V. NEAT and its related standardization efforts in TAPS can re-enable the evolution of the Internet's transport layer because they break the deployment vicious circle; NEAT's flexible, customizable API makes it easy to define and use novel services on top of the socket API, seamlessly leveraging new transport protocols

Naeem Khademi and Michael Welzl are with the Department of Informatics, University of Oslo, Norway. E-mail: {naeemk, michawe}@ifi.uio.no.
David Ros and David Hayes are with Simula Research Laboratory, Norway. E-mail: {dros, davidh}@simula.no.
Zdravko Bozakov and Simone Mangiante are with Dell EMC, Ireland. E-mail: {Zdravko.Bozakov, Simone.Mangiante}@dell.com.
Anna Brunstrom, Karl-Johan Grinnemo and Per Hurtig are with Karlstad University, Sweden. E-mail: {anna.brunstrom, karl-johan.grinnemo, per.hurtig}@kau.se.
Gorry Fairhurst and Tom Jones are with the University of Aberdeen, Aberdeen, United Kingdom. E-mail: {tom, gorry}@erg.abdn.ac.uk.
Michael Tüxen and Felix Weinrank are with Münster University of Applied Sciences, Germany. E-mail: {tuexen, weinrank}@fh-muenster.de.

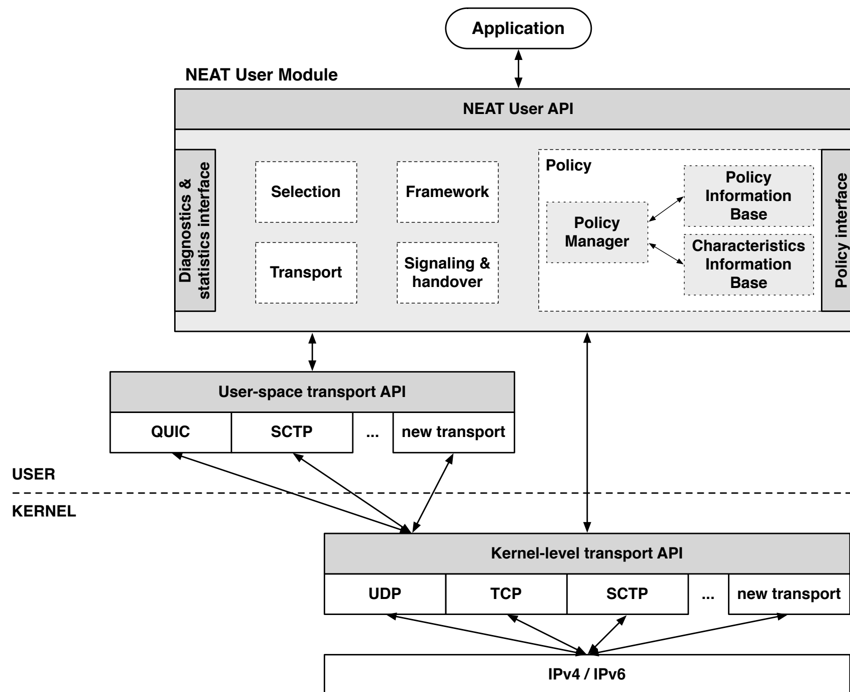


Fig. 1. The architecture of the NEAT System.

as they become available. This, in turn, may create a shift in the traffic pattern seen by vendors and administrators of middleboxes that could at some point lead them to support such traffic. Section IV presents several examples of benefits that NEAT offers to applications.

II. Background

TCP and UDP are a part of the kernel of almost all operating systems and are also supported by nearly all middleboxes. During its lifetime, TCP has been substantially improved. However, the evolution of TCP has had to deal with constraints. Changes to packet format have to consider that middleboxes might block or limit communication. Therefore, a fallback mechanism to the old packet format has become a part of such protocol extensions. New protocol mechanisms (e.g., congestion control or loss recovery mechanisms) mostly focus on single-sided changes to allow faster deployment—but the speed of deployment is still limited by the software development cycle of operating systems.

In addition, having a feature available on an operating system does not imply that it is made available to an application running with user privileges; new features are often disabled by default and turning them on requires special privileges since it has host-wide consequences.

Because UDP provides only minimal services (port numbers and a checksum), it is possible to use it as substrate to implement transport protocols on top of it to introduce features; this approach has become increasingly common. This leads to every UDP-based application to some extent needing to implement the same core set of functions [1]. However, it also leads to per-application protocol stacks, where transport protocols cannot easily be moved between applications (and making this possible is often not in the interest of the application developer). Developing an efficient transport protocol is a difficult task which requires a number of features to be re-implemented again and again. UDP-based transport protocols have also done nothing to fix the general architectural problem: the socket API's protocol binding remains, typically with a choice between only TCP and UDP.

Specific applications can require services not provided by TCP. One example is the transport of signaling messages in telephony signaling networks. This is used to transfer mostly small messages and requires a high level of fault tolerance. When a protocol stack for this application was developed, a new transport protocol, SCTP [2], was created to fulfill these specific requirements. It was possible to deploy SCTP in these networks because there were no middleboxes, and kernel implementations for the operating systems used in telephony signaling networks were developed.

Currently, the IETF and the World Wide Web Consortium (W3C) are developing WebRTC, a technology for real-time multimedia communication directly between web browsers. Non-media communication using SCTP is also supported; to facilitate deployment across arbitrary Internet paths, SCTP runs over UDP. Google has developed QUIC, a UDP-based transport protocol with features including fast connection setup, cross-layer optimized security, and a modern congestion control and loss recovery mechanism. If QUIC fails to traverse a middlebox, the web browser can fall back to using TCP.

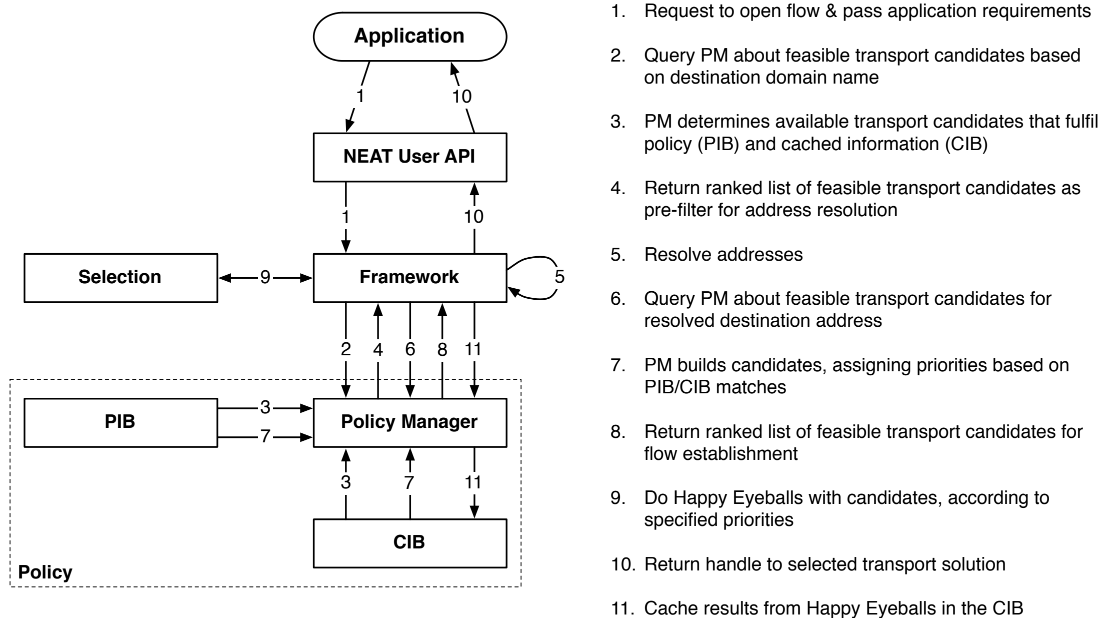


Fig. 2. Simplified workflow showing how NEAT components interact when opening a flow.

By moving a transport protocol from the operating system to the application (e.g., WebRTC and QUIC integrated in web browsers), the release cycle can be substantially shortened, the implementation becomes independent from the operating system, and the protocol can be tailored to the specific application. Using UDP encapsulation is the only option to not using TCP and be able to traverse middleboxes.

This enables a larger variety of transport protocols to co-exist and change over time, but does not help with the issue of per-application protocol stacks mentioned before. An application programmer has to add complexity to benefit from advanced features in their application; this requires utilizing different APIs, figuring out which protocols are supported by the remote end-points, selecting protocol mechanisms, and providing fallback mechanisms when these happen to not work across the current network path. None of these are specific to a particular transport protocol, but are related to the need for the programmer to work with a variety of transport protocols. This general problem could be addressed by defining a new transport system, as outlined next.

III. Re-enabling evolution: introducing NEAT

As discussed above, using transport services beyond TCP and UDP today puts a high burden on the application developer. The NEAT Library addresses this problem by providing application developers with one enhanced API that is transport-protocol independent, with the library providing support for selecting the best available transport option *at run-time* and handling fallback between transport protocols as needed. Running as a user-space library, NEAT can make use of transports running both in user space and in the kernel, all transparent to the applications. Protocols like SCTP are already supported over many paths, but they cannot be easily used by application programmers unless they are supported over *all* paths. NEAT changes this by placing functions such as selecting a transport and handling fallback below the API. NEAT allows such functions to evolve with the network, rather than be bound to specific applications.

Figure 1 provides a schematic view of the NEAT architecture. Applications employ the NEAT User API to access transport services. This API is located in the NEAT User Module, which is the core of NEAT and comprises components that together deliver services tailored to application requirements at run-time. The components in the module are grouped in five categories: Framework, Policy, Selection, Transport, and Signaling & Handover.

Framework components provide basic functionality required to use NEAT. They define the structure of the NEAT User API and implement core library mechanisms. Applications provide information about their requirements for a desired transport service via this API.

Policy components comprise the Policy Information Base (PIB), the Characteristics Information Base (CIB), and the Policy Manager (PM). The function of the PM is to generate a ranked list of connection candidates that fulfill the application requirements while taking system and network constraints into account and adhering to configured policies. All policy components operate on so-called NEAT Properties, which express requirements and characteristics throughout the NEAT System. Each property is a key-value tuple with additional metadata indicating the priority (mandatory or optional) and weight of the associated attribute.

Policies and profiles—stored in the PIB—extend and modify the property set associated with each connection candidate.

In addition, the CIB repository maintains information about available interfaces, supported protocols towards previously-accessed destination endpoints, network properties and current/previous connections between endpoints. The content of the CIB is continuously updated by local and external CIB sources.

Selection components choose an appropriate transport solution. The additional information provided by the NEAT User API enables the NEAT Library to move beyond the constraints of the traditional socket API, making the stack aware of what is actually *desired* or *required* by the application. On the basis of both the information provided by the NEAT User API and the PM, candidate transport solutions are identified. The candidate solutions are then tested by the Selection components, and the one deemed most appropriate is then used.

Transport components are responsible for providing functions to instantiate a transport service for a particular traffic flow. They provide a set of transport protocols and other necessary components to realize a transport service. While the choice of transport protocols is handled by the Selection components, the Transport components are responsible for configuring and managing the selected transport protocols.

Signaling & Handover components can provide advisory signaling to complement the functions of the Transport components. This could include communication with middleboxes, support for handover, failover and other mechanisms.

Figure 2 illustrates a simplified workflow, showing how the NEAT components interact when an application initiates a new flow. As follows from the above description, NEAT has an evolvable architecture that opens up for the introduction of new transport services and can enable interaction with network devices to improve such services. NEAT also enables the incremental introduction of new transport protocols, both in the kernel and in user space, as the API is independent from the underlying transport protocol.

IV. Benefits of NEAT

Next, we present four examples of key benefits of using the NEAT Library. First, NEAT provides an API that is simple to use. This allows existing applications to be easily ported to the NEAT Library, simplifying network communication and reducing code complexity.

NEAT also provides automatic fallback using a *Happy Eyeballs* (HE) mechanism. HE is a generic term for algorithms that test for end-to-end support of a protocol X simply by trying to *use* X, then falling back to a default choice Y known to work if X is found to not work (e.g., after a suitable timeout). This added functionality is lightweight and has negligible cost compared to other communication tasks. It allows applications to take advantage of the best available transport solution and in turn enables transport innovation (e.g., applications do not need to be recoded to use a new transport feature or protocol that becomes available).

NEAT not only facilitates evolution of the transport protocols and introduction of new transport mechanisms, it can also help enable innovation at the network layer. The higher-level of abstraction offered by the NEAT User API eases the path to utilizing Quality of Service (QoS) support for UDP-based applications, and could be used to access other network services should they become available (e.g., selection of the most cost-effective or secure path utilizing IPv6 provisioning-domain information). Applications and networks can also leverage the flexible control provided by the Policy components, for example to provide a generic interface for exchanging information between external SDN controllers and NEAT-enabled applications.

A. Porting applications to NEAT

The NEAT User API offers a uniform way to access networking functionality, independent from the underlying network protocol or operating system. Many common network programming tasks like address resolution, buffer management, encryption, connection establishment and handling are built into the NEAT Library and can be used by any application that uses NEAT.

Developers write applications using the asynchronous and non-blocking NEAT User API, implemented using the libuv [3] library which provides asynchronous I/O across multiple-platforms.

As shown in Listing 1, users can request the services that they expect from the network (e.g. low latency, reliable delivery, a specific TCP congestion control algorithm) by providing an optional set of properties to control the behavior of the library.

```c
static neat_error_code
on_connected(struct neat_flow_operations *ops)
{
// set callbacks to write and read data
ops->on_writable    = on_writable;
ops->on_all_written = on_all_written;
ops->on_readable    = on_readable;
neat_set_operations(ops->ctx, ops->flow, ops);
return NEAT_OK;
}

int
main(int argc, char *argv[])
{
// initialization of basic NEAT structures
struct neat_ctx *ctx;
struct neat_flow *flow;
struct neat_flow_operations ops;
ctx  = neat_init_ctx();
flow = neat_new_flow(ctx);
memset(&ops, 0, sizeof(ops));

// callback when connection is established
ops.on_connected = on_connected;
neat_set_operations(ctx, flow, &ops);

// optional user requirements in JSON format
static char *properties = "{\"transport\":[\"SCTP\", \"TCP\"]}";
neat_set_property(ctx, flow, properties);

// connect
if (neat_open(ctx, flow, "127.0.0.1", 5000, NULL, 0)) {
fprintf(stderr, "neat_open failed\n");
return EXIT_FAILURE;
}

// start libuv loop
neat_start_event_loop(ctx, NEAT_RUN_DEFAULT);

neat_free_ctx(ctx);

return EXIT_SUCCESS;
}
```

Listing 1. Code example from a simple client using the NEAT API.

The NEAT Library then uses a set of internal components to establish a connection over the network. To make an

appropriate selection, the Policy Manager maps user properties to policies and computes a set of candidate transports that can satisfy the request. NEAT also can utilize policy information directly set by the user, system administrator or developer.

Connections to a peer endpoint are made by creating a new flow, which is a bidirectional link between two endpoints similar to a socket in the traditional Berkeley Socket API but not strictly tied to an underlying transport protocol.

The NEAT API executes callbacks in the application when an event from the underlying transport happens, creating a more natural and less error-prone way of network programming than with the traditional socket API. The three most important callbacks in the NEAT API are `on_connected`, called once the flow has connected to a remote endpoint; `on_readable` and `on_writable`, called once data may be written to or read from the flow.

Our experience with NEAT shows a reduction of the code size by $\approx$ 20% for each application, as the library streamlines a number of connection establishment steps. For example, the single function call `neat_open` requests name resolution and all other functions required before communication can start, hiding complex boilerplate code. Ported applications remain fully interoperable with regular TCP/IP-based implementations, while being able to take advantage of NEAT functions. Besides, they can benefit from support for alternative transports, when available, relieving programmers from dealing with fallbacks between protocols. Finally, a traditional socket-based shim layer has been implemented on top of NEAT to allow legacy applications to make use of NEAT functionalities through policies without requiring direct porting to the NEAT API.

B. Happy Eyeballs: A Lightweight Transport Selection Mechanism

Selection components employ a HE mechanism to enable a source host to determine whether a transport protocol is supported along the current network path. This allows applications to benefit from advances in transports that may be only partially deployed in the Internet. The HE mechanism used by NEAT is similar to that introduced to facilitate IPv6 adoption [4], but works at the transport layer to select one of a set of connection-oriented transport solutions. The Selection components receive a ranked list of potential candidates generated by the PM, where a higher ranking indicates a better match with application and policy requirements. The HE mechanism then concurrently tries each transport solution from the list, delaying initiation of lower-priority transport solutions.

Figure 3 shows the HE mechanism in a scenario where the best transport to the destination is unknown and current policy dictates that the HE process is used to select between TCP and SCTP, but preferring SCTP. The initiation of the TCP connection is delayed for a time interval governed by policy, specifying a difference in priority between candidate protocols. If the SCTP connection does not complete within the time interval, a TCP connection is also started. The first transport to complete a connection is selected and becomes the transport

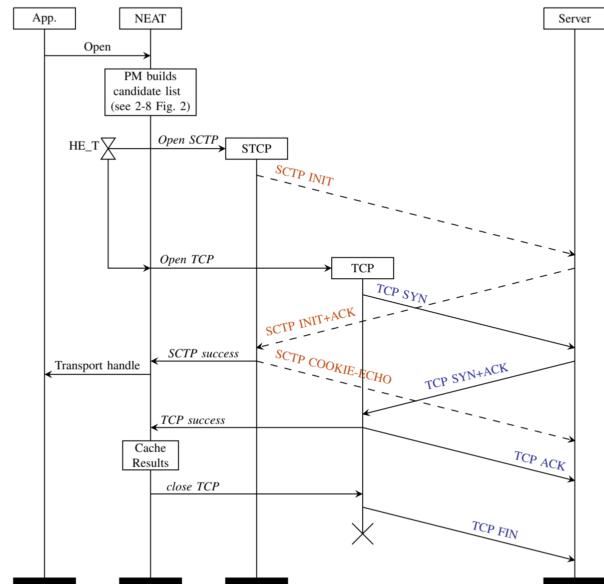


Fig. 3. Message Sequence Chart (MSC) illustrating the NEAT Happy Eyeballs (HE) transport selection process when selecting between TCP and SCTP, SCTP preferred.

of choice. Once connectivity is established, other methods are abandoned, and their connections closed.

To avoid wasting network resources by routinely attempting concurrent connections, HE instructs the Policy components to cache the outcome of each selection result in the CIB for a configurable amount of time. After expiry of the time, the selection is removed from the cache, re-enabling HE.

Consider the scenario in Figure 3. Attempting selection when there is no existing cache entry requires extra resources, potentially resulting in opening connections for each candidate transport protocol. In this example, SCTP completes first and the TCP connection is closed having sent no data. With typical web traffic and worst-case packet sizes, byte overhead is as small as $\approx$ 1%. For a cache hit rate of 80%, this reduces further to $\approx$ 0.2%. A detailed evaluation of the impact of HE in terms of memory and CPU utilization can be found in [5], where it is shown that CPU costs are relatively small (especially when considering the cost of TLS encryption), and that HE has only a minor impact on memory consumption.

C. Deployable QoS with NEAT

Network QoS is often used for traffic engineering, but few applications have managed to exploit this technology beyond a controlled network environment. One major obstacle is the lack of a consistent high-level API.

There have been attempts to add methods that directly associate QoS with IP traffic (e.g., [6], [7]), but they have seen little to no adoption. A key challenge is how to express the service requirements, while still enabling policy to influence choice and providing flexibility when the network is unable to directly satisfy the requirements.

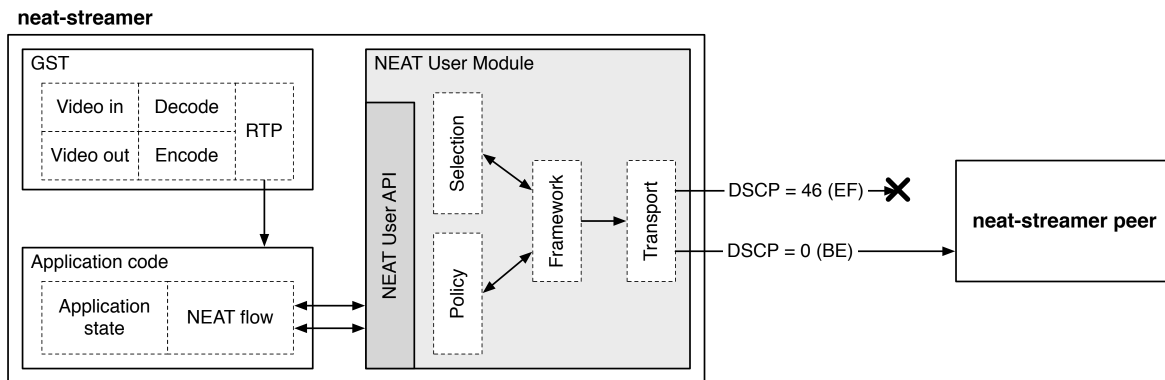


Fig. 4. Example of neat-streamer using QoS fallback with NEAT. The application sets up the media pipeline and uses NEAT to transfer data across the network, according to the requested service. The NEAT Library could try to send UDP datagrams with a DSCP set to High Priority Expedited Forwarding (EF). A timer triggers the NEAT Library to query the application status, which then reveals the application failed to use this DSCP, so NEAT can now try the next DSCP value, Default (BE). When the timer again triggers, the application reports success and this code point continues to be used.

The NEAT API can allow applications to specify QoS requirements. This can, for example, utilize policy information to drive an appropriate Differentiated Service Code Point (DSCP). The finally chosen DSCP can be based on both static policy and dynamic information collected from connections using NEAT.

The NEAT fallback mechanism can be used with any datagram services to enable the NEAT Library to select between a list of candidate datagram transports, network encapsulations and interfaces. This can assist an application to robustly find desirable connection parameters for any path by transparently falling back to alternative services when required (resembling, but different to the NEAT HE function for connection-oriented transports).

Neat-streamer [8] is a demo application that utilizes the NEAT Library for live streaming video over connectionless transports using the GStreamer (GST) media libraries. GST is a pipeline-based media system that supports a wide range of audio and video formats and other functions via a plugin system.

Figure 4 shows the interactions between NEAT and neat-streamer running on a network that drops traffic with certain DSCP values set.

Because neat-streamer uses NEAT, it can indicate the QoS treatment that it requires for each media flow, and the endpoint to which it wishes to stream. NEAT provides the required QoS marking and may determine which transport service to use (e.g., choosing between UDP-Lite, UDP, or use of Traversal Using Relays around NAT, TURN), and whether security functions are required.

NEAT also provides the protocol machinery to update the selected flow parameters should network connectivity problems be reported by the application. A timer triggers a callback function within the application to determine whether the application believes the network is delivering the service it requires (in many cases, only the application is aware of the performance reported by a remote datagram receiver). When an application reports failure it can allow NEAT to use the list of candidates, and potentially other information (e.g., held within the CIB) to search for alternate parameters.

D. SDN Integration

The ability of enabling external sources to query and augment the state of the Policy Manager is a key design choice of the NEAT architecture. As a consequence, NEAT-enabled end-hosts can be seamlessly integrated in centrally controlled environments, such as Software-Defined Networks (SDNs). In such environments, logically centralized controllers aim to maintain a global view of the network and optimize its utilization. To achieve this, controllers ideally require detailed and up-to-date knowledge of available resources, in addition to the requirements and characteristics of deployed applications. Today, controllers rely on time-consuming and error-prone heuristics to infer the association between applications, their requirements, and observed flows.

In this context, the benefit of the NEAT approach is three-fold. Firstly, NEAT applications may inform controllers directly about their particular requirements towards the network. In NEAT, such requirements are defined either explicitly by application developers, or through suitable system policies. This strategy can reduce the need for network controllers to guess how to treat individual flows. Secondly, through the Policy Manager CIB, NEAT enables controllers to supply applications with detailed information about the state of paths available to the host. In the absence of this feedback, metrics such as available bandwidth or latency may need to be inferred individually by each application through measurements. Finally, the controller gains the ability to deploy policies at the host level which influence the transport protocols, interfaces and associated parameters used in NEAT applications.

All mechanisms necessary for exchanging information between the controller and NEAT-enabled applications are implemented in Policy components. Specifically, the Policy Interface is exposed through a REST API, enabling external entities to push information to the PIB and CIB and query their contents. As a result, for each flow request created by a NEAT application, the Policy Manager will utilize the latest policies

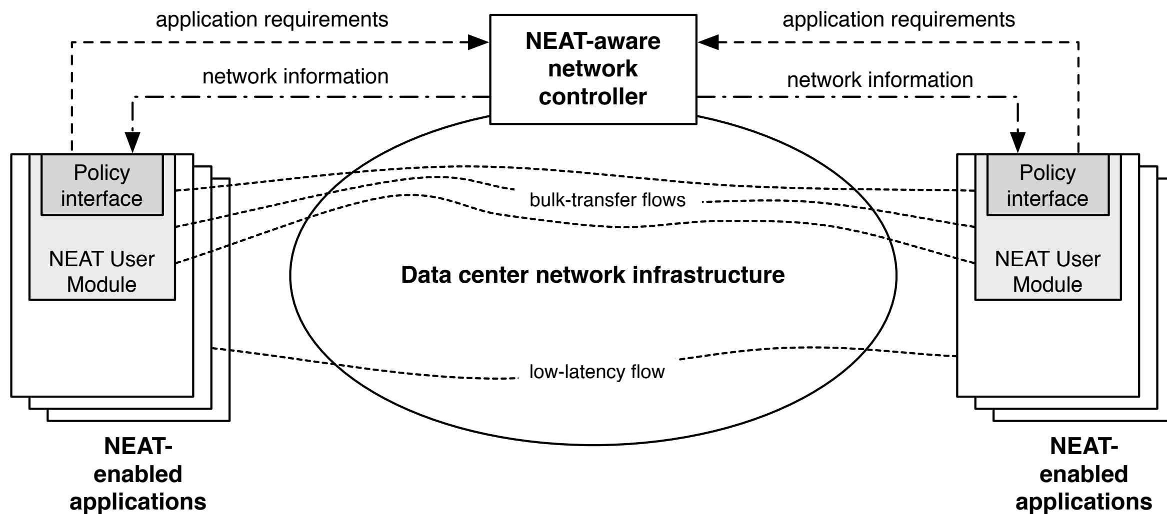


Fig. 5. SDN architecture in which a controller uses the NEAT Library to supply end-hosts with information about the available network resources, and to collect information about the application requirements.

and network attributes supplied by a controller to select the most suitable connection option. Similarly, the API allows the controller to query the CIB to identify the requirements associated with specific application flows or relevant policies configured in the PIB.

To demonstrate the feasibility of the aforementioned controller integration we have implemented a scenario comprised of NEAT-enabled hosts deployed in an OpenFlow SDN network. The aim of the scenario, depicted in Figure 5, is to enable a controller to steer the handling of bulk traffic flows. Each host runs a NEAT-enabled data replication application which provides the estimated flow size as part of the NEAT API call. We used the OpenDaylight framework to implement a controller which monitors the network utilization and calculates a data volume threshold above which flows are considered as bulk flows. We implemented a northbound API and periodically publish a policy to the NEAT end hosts. The policy is triggered when the flow size exceeds the threshold and forces the flows to be tagged with a predefined DSCP marking. As a result, flows affected by the policy are routed through a pre-provisioned network path.

V. STANDARDIZATION

Recognizing the need for the transport layer (socket) interface to become protocol-independent, the IETF chartered a working group called "Transport Services" (TAPS) in September 2014. A common approach in prior work was to start analysis based on the needs of applications. Instead, TAPS used a methodology that started from a survey of the services offered by available IETF transport protocols [9]. It is currently documenting the primitives and parameters used to access features of a subset of these protocols [10] to form a basis for the design of a protocol-independent API. NEAT developers have been actively contributing to this initiative based on experience of using the NEAT API, which shares many of the goals behind development of TAPS.

The working group is now shortening the list of transport features. Examples of features include "Specify ECN field" or "Choice between unordered (potentially faster) or ordered delivery of messages". A recent contribution by NEAT developers [11] recommends against exposing a transport feature in the API when either choosing or configuring it requires knowledge specific to the network path or the operating system, but not the application. A final step will eliminate features specific to a particular protocol that cannot reasonably be implemented using a different protocol—such features contradict the main purpose of TAPS, to be protocol-independent. At the end of this process, this will result in a subset of transport features that end systems supporting TAPS need to provide. NEAT implements all services specified in the current TAPS documents and may therefore be regarded as a prototype implementation of TAPS.

TAPS is also chartered to define experimental support mechanisms, for example to select and engage an appropriate protocol and discover the set of protocols available for a selected service between a given pair of endpoints, to allow the operating system to choose between protocols (e.g., HE and application-level feedback mechanisms). This approach of breaking the binding between applications and transport protocols is an important final step for TAPS.

VI. CONCLUSION

The service needs of today's Internet applications range well beyond the basic ones provided by TCP and UDP. Yet, the Internet's transport layer, as it presents itself to a developer via the socket API, has remained unchanged. This has led to per-application (and per-company) developments in user space, over UDP, such as QUIC for Google Chrome. While these new UDP-based transport protocols have recently

pushed the transport layer into the spotlight, they are also only silo solutions which do nothing to solve the architectural ossification problem: the socket API binds applications to protocols at design time—therefore, transport protocols cannot be replaced without changing applications.

In this paper we presented the NEAT Library, which lets application developers access features of transport protocols in a simple and uniform way. NEAT helps freeing developers from platform or protocol dependencies; they do not have to worry about the specifics of each protocol or operating system; they also do not need to worry about whether a protocol works on a given path. Underneath the NEAT User API, new protocols can seamlessly be inserted, automatically yielding benefits to the application on top. With NEAT's clear layer separation, the Internet's transport layer can finally evolve again.

At the time of writing, prototype code for all component types has been developed for several Unix-like OSs. Besides neat-streamer, the NEAT development team has ported example applications to NEAT for early testing, including the Nghttp2 [12] web server and client, several smaller applications like HTTP/HTTPS clients and performance measurement tools; also, a NEAT-supported Firefox implementation is currently under development by Mozilla. NEAT is an open-source project that welcomes contributions. Source code, documentation and implementation status can be found on GitHub [13].

Acknowledgment

The authors would like to thank the anonymous reviewers for their useful remarks.

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] L. Eggert and G. Fairhurst, "Unicast UDP Usage Guidelines for Application Designers," RFC 5405 (Best Current Practice), Internet Engineering Task Force, Nov. 2008, accessed on February 23, 2017. [Online]. Available: http://www.ietf.org/rfc/rfc5405.txt

[2] R. Stewart, "Stream Control Transmission Protocol," RFC 4960 (Proposed Standard), Internet Engineering Task Force, Sep. 2007, accessed on February 23, 2017. [Online]. Available: http://www.ietf.org/rfc/rfc4960.txt

[3] libuv — Cross-platform Asynchronous I/O. Accessed on February 23, 2017. [Online]. Available: https://libuv.org/

[4] D. Wing and A. Yourtchenko, "Happy Eyeballs: Success with Dual-Stack Hosts," RFC 6555 (Proposed Standard), Internet Engineering Task Force, Apr. 2012, accessed on February 23, 2017. [Online]. Available: http://www.ietf.org/rfc/rfc6555.txt

[5] G. Papastergiou, K.-J. Grinnemo, A. Brunstrom, D. Ros, M. Tüxen, N. Khademi, and P. Hurtig, "On the Cost of Using Happy Eyeballs for Transport Protocol Selection," in *Proceedings of the 2016 Applied Networking Research Workshop (ANRW)*. Berlin: ACM, Jul. 2016, pp. 45–51.

[6] H. Abbasi, C. Poellabauer, K. Schwan, G. Losik, and R. West, "A Quality-of-service Enhanced Socket API in GNU/Linux," in *4th Real-Time Linux Workshop*, 2002, accessed on February 23, 2017. [Online]. Available: https://www.osadl.org/fileadmin/events/rtlws-2002/proc/g08_abbasi.pdf

[7] P. Gomes Soares, 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.

[8] Neat-streamer Video Workload Tool. Accessed on February 23, 2017. [Online]. Available: https://github.com/uoaerg/neat-streamer

[9] G. Fairhurst, B. Trammell, and M. Kühlewind, "Services provided by IETF transport protocols and congestion control mechanisms," Internet Engineering Task Force, Internet-Draft draft-ietf-taps-transports-11, Sep. 2016, work in Progress. Accessed on February 23, 2017. [Online]. Available: https://tools.ietf.org/html/draft-ietf-taps-transports-11

[10] M. Welzl, M. Tüxen, and N. Khademi, "On the Usage of Transport Service Features Provided by IETF Transport Protocols," Internet Engineering Task Force, Internet-Draft draft-ietf-taps-transports-usage-01, Jul. 2016, work in Progress. Accessed on February 23, 2017. [Online]. Available: https://tools.ietf.org/html/draft-ietf-taps-transports-usage-01

[11] S. Gjessing and M. Welzl, "A Minimal Set of Transport Services for TAPS Systems," Internet Engineering Task Force, Internet-Draft draft-gjessing-taps-minset-03, Oct. 2016, work in Progress. Accessed on February 23, 2017. [Online]. Available: https://tools.ietf.org/html/draft-gjessing-taps-minset-03

[12] T. Tsujikawa. Nghttp2: HTTP/2 C Library. https://nghttp2.org/. Accessed on February 23, 2017.

[13] NEAT GitHub public repository. Accessed on February 23, 2017. [Online]. Available: https://github.com/NEAT-project/neat

D Paper: *On the Cost of Using Happy Eyeballs for Transport Protocol Selection*

The following research paper [22] has been produced by project participants.

On the Cost of Using Happy Eyeballs for Transport Protocol Selection

Giorgos Papastergiou[†], Karl-Johan Grinnemo[‡], Anna Brunstrom[‡], David Ros[†], Michael Tüxen[⋆], Naeem Khademi[*], Per Hurtig[‡]
[†]Simula Research Laboratory, [‡]Karlstad University, [⋆]Fachhochschule Münster, [*]University of Oslo
{gpapaste, dros}@simula.no, {karl-johan.grinnemo, anna.brunstrom, per.hurtig}@kau.se, tuexen@fh-muenster.de, naeemk@ifi.uio.no

ABSTRACT

Concerns have been raised in the past several years that introducing new transport protocols on the Internet has become increasingly difficult, not least because there is no agreed-upon way for a source end host to find out if a transport protocol is supported all the way to a destination peer. A solution to a similar problem—finding out support for IPv6—has been proposed and is currently being deployed: the *Happy Eyeballs* (HE) mechanism. HE has also been proposed as an efficient way for an application to select an appropriate transport protocol. Still, there are few, if any, performance evaluations of transport HE. This paper demonstrates that transport HE could indeed be a feasible solution to the transport support problem. The paper evaluates HE between TCP and SCTP using TLS encrypted and unencrypted traffic, and shows that although there is indeed a cost in terms of CPU load to introduce HE, the cost is relatively small, especially in comparison with the cost of using TLS encryption. Moreover, our results suggest that HE has a marginal impact on memory usage. Finally, by introducing caching of previous connection attempts, the additional cost of transport HE could be significantly reduced.

CCS Concepts

•Networks → Transport protocols; Network performance evaluation;

Keywords

Transport-protocol selection, Happy Eyeballs, TCP, SCTP, TLS, CPU load, memory usage.

1. INTRODUCTION

The deployment of new transport protocols on the Internet is not a trivial task. Several hurdles have to be cleared before a new transport can be used between an arbitrary pair of end hosts and see wide adoption. One of the main issues that has to be solved is, how can an end host *know* if a new protocol X is supported along the whole end-to-end path, including the remote host? In the absence of a priori knowledge or explicit signaling, the only way to know whether X works is to *try* it.

Testing a set of candidate protocols can be done serially—e.g., try first with the preferred choice X and, if the attempt fails after a suitable timeout, then fall back to a default alternative Y. Since a connection timeout can introduce a delay of up to tens of seconds, serializing attempts can incur a large latency penalty when the new protocol X is not supported, stalling the application until the subsequent connection trial succeeds.

The *Happy Eyeballs* (HE) mechanism was introduced as a means to facilitate IPv6 adoption [13]. Dual-stack client applications should be encouraged to try setting up connections over IPv6 first, and fall back to using IPv4 if IPv6 connection attempts fail. However, serializing tests for IPv6 and IPv4 connectivity can result in large connection latency. Happy Eyeballs for IPv6 minimizes the cost in delay by *parallelizing* attempts over IPv6 and IPv4.

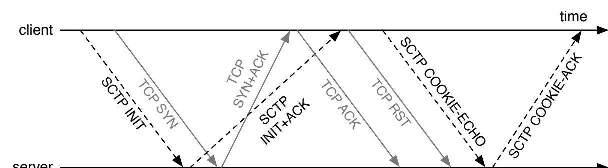


Figure 1: HE for selecting between SCTP and TCP, with SCTP being the preferred choice.

The basic idea behind Happy Eyeballs for IPv6 can be extended to discover support for transport protocols, and in particular to allow an application to use SCTP when it is available end-to-end, else revert to TCP when it is not [11,12]. Figure 1, adapted from [12], depicts how HE for transport selection may work. An end host simultaneously initiates a TCP connection and an SCTP association; it is assumed that the SYN+ACK arrives before the INIT-ACK. If SCTP (the preferred choice) is supported, the TCP connection is abandoned[1]. Similar to what is done in actual

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ANRW '16, July 16 2016, Berlin, Germany

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ISBN 978-1-4503-4443-2/16/07…$15.00
DOI: http://dx.doi.org/10.1145/2959424.2959437

[1]The figure assumes there is a mechanism to notify the application about the reception of the SCTP INIT-ACK, so the

implementations of HE for IPv6 [10], a small delay may be introduced to give an advantage to SCTP over TCP [14]; else, the host can just pick the protocol for which a response (TCP SYN+ACK, SCTP INIT-ACK) arrives first.

This paper focuses on the performance penalties introduced by HE for transports. In particular, we study the impact that a HE mechanism for selecting between TCP and SCTP may have on CPU load and memory usage at a destination end-host. Our results provide empirical arguments in favor of using such a mechanism for transport selection.

The rest of the paper is organized as follows. Section 2 provides some background and motivation for introducing a transport HE mechanism. As follows from this discussion, two major concerns about transport "happy-eyeballing" are higher CPU load and memory usage. In Section 3 we assess these concerns by experimentally evaluating HE between TCP and SCTP. Finally, Section 4 concludes the paper.

2. BACKGROUND

Some thirty years back, there were two Internet transport protocols to choose from: TCP and UDP. Since these two protocols basically represent the opposites in the services they provide—TCP provides a reliable, in-order, byte-stream oriented delivery service and UDP an unreliable, unordered message delivery—the selection between these two protocols from an application viewpoint was mostly straightforward. Furthermore, it could be expected that all hosts supported both TCP and UDP, and there were no middleboxes altering or blocking the traffic before it reached its final destination.

Today, the Internet looks rather different. The number of standard transport protocols and their options (and the different services they may provide) has increased, making the selection of a suitable transport less straightforward. New transports may allow to provide improved services to applications, but middleboxes such as firewalls, NATs and load balancers have become an integral part of the Internet, and there is a great diversity in how they are configured and deployed; it cannot be assumed that any transport or transport option can safely make it from sender to receiver.

As mentioned in Section 1, although HE was primarily introduced as a way to promote the use of IPv6, it has also been proposed as a way for an application to efficiently select transports [11, 12]. Wing and Yourtchenko [11] provide recommendations for HTTP clients on how to seamlessly migrate from TCP to SCTP without any adverse impact on the user experience. Moreover, they propose a way to combine an IPv6/IPv4 HE with a TCP/SCTP HE for a web browser running on a dual-stack machine [12]. Also worth mentioning in this context is the work carried out by the Transport Services (TAPS) working group of the IETF [8]. One of this working group's planned documents should "[...] explain how to select and engage an appropriate protocol and how to discover which protocols are available for the selected service between a given pair of end points [...]", something which will likely require HE between transport solutions.

Still, a HE transport-selection mechanism does raise questions about increased CPU usage and memory consumption.

---

application can then abort the TCP connection by sending a TCP Reset. Without such mechanism, the TCP connection can only be aborted after the full four-way handshake of SCTP is completed.

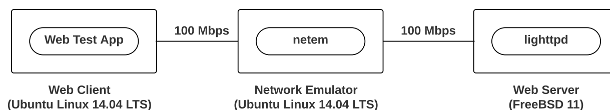


Figure 2: Experiment setup.

When HE is used, a single connection request from the application might result in several concurrent transport connection requests, i.e., not just one connection request at a time as is the case when HE is not used. Hence, the use of HE could result in an increase in both CPU and memory usage. Baker [3] provides recommendations on how to evaluate IPv4/IPv6 HE, however, metrics like CPU load and memory usage are not considered in [3]. Note that neither of these are key metrics for IP HE, whereas they are for transport HE—this is so because transport connection setup means creating state in end points. There have been a few discussions of the performance of HE for IPv6/IPv4 [1, 2, 4, 7] but, to the best of our knowledge, this paper is the first to focus on performance aspects of transport-layer HE, in terms of CPU load and memory usage.

A HE transport-selection mechanism also raises questions about increased use of network resources, a key issue for the scalability of HE. For instance, the aforementioned HE proposal by Wing et al. [12] transmits four packets for every application connection request. Still, as already pointed out by Wing et al. [11, 13, 14], HE network resource usage should be mitigated by the use of caching.

3. EVALUATION

This section evaluates HE between TCP and SCTP in terms of CPU load and memory usage. The section begins with a description of the experiment setup and the studied test scenarios. The remainder of the section presents and comments on the results from the execution of these scenarios.

3.1 Experiment Setup

In our experiment, we modeled a single wide-area network path to an upstream Web server. The laboratory network used in our experiment is shown in Figure 2. The three machines in the experiment were of type: Dell Optiplex 9020 with 3.60 GHz Intel Core i7-4790 (quad core) processors. The Web Client and Network Emulator machines ran Ubuntu Linux 14.04 LTS with kernel 3.13.0, and the Web Server machine ran FreeBSD 11 (revision r294499). All machines used the default network kernel settings, except those listed in Table 1. These changes assured that the testbed could properly support the connection rates considered in this work, and disabled all SCTP features that were not needed in the experiments. The Network Emulator machine used *netem* to emulate a propagation delay of 20 ms.

The Web Client hosted a custom-designed Web traffic generator in which two modified versions of the *httperf* [6] web traffic generator (one that supports TCP and one that supports SCTP) were combined to implement the studied test scenarios. HTTP/1.0 with the Keep-Alive option enabled was used in both httperf programs. The FreeBSD server hosted a *lighttpd* [9] server, modified to listen for both TCP and SCTP HTTP/1.0 unencrypted and TLS-encrypted requests. The lighttpd server was also modified

Table 1: Kernel Settings

Web Server	Settings
`net.inet.tcp.syncache.hashsize`	2048
`kern.ipc.somaxconn`	4096
`net.inet.sctp.pr_enable`	0
`net.inet.sctp.ecn_enable`	0
`net.inet.sctp.outgoing_streams`	1
`net.inet.sctp.incoming_streams`	1
`net.inet.sctp.asconf_enable`	0
`net.inet.sctp.auth_enable`	0
`net.inet.sctp.reconfig_enable`	0
`net.inet.sctp.nrsack_enable`	0
`net.inet.sctp.pktdrop_enable`	0
Web Client	**Settings**
`net.ipv4.ip_local_port_range`	10000 61000
`net.ipv4.tcp_tw_recycle`	1
Network Emulator	**Settings**
`net.ipv4.ip_forward`	1

Table 2: lighttpd Settings

Configuration Parameter	Settings
`server.network-backend`	writev
`server.event-handler`	kqueue
`server.max-fds`	4096
`server.max-connections`	2048
`server.max-worker`	7
`ssl.use-sslv2`	disable
`ssl.use-sslv3`	disable

Table 3: Malloc types and zones

Command	Malloc type / Zone
`vmstat -m`	filedesc, kqueue, ip6opt, ip6ndp, pcb, BPF, ifnet, ifaddr, ether_multi, lltable, routetbl, igmp, in_mfilter, in_multi, ip_moptions, sctp_map, sctp_stri, sctp_stro, sctp_a_it, sctp_atcl, sctp_atky, sctp_athm, sctp_vrf, sctp_ifa, sctp_ifn, sctp_timw, sctp_iter, sctp_socko, hostcache, in6_mfilter, in6_multi, ip6_moptions, mld, inpcbpolicy, ipsecpolicy
`vmstat -z`	KNOTE, socket, udp_inpcb, udpcb, tcp_inpcb, tcpcb, tcptw, syncache, hostcache, sackhole, tcpreass, sctp_ep, sctp_asoc, sctp_laddr, sctp_raddr, sctp_chunk, sctp_readq, sctp_stream_msg_out, sctp_asconf, sctp_asconf_ack, selfd
`netstat -m`	mbufs, mbuf clusters, 4k jumbo clusters, 9k jumbo clusters and 16k jumbo clusters

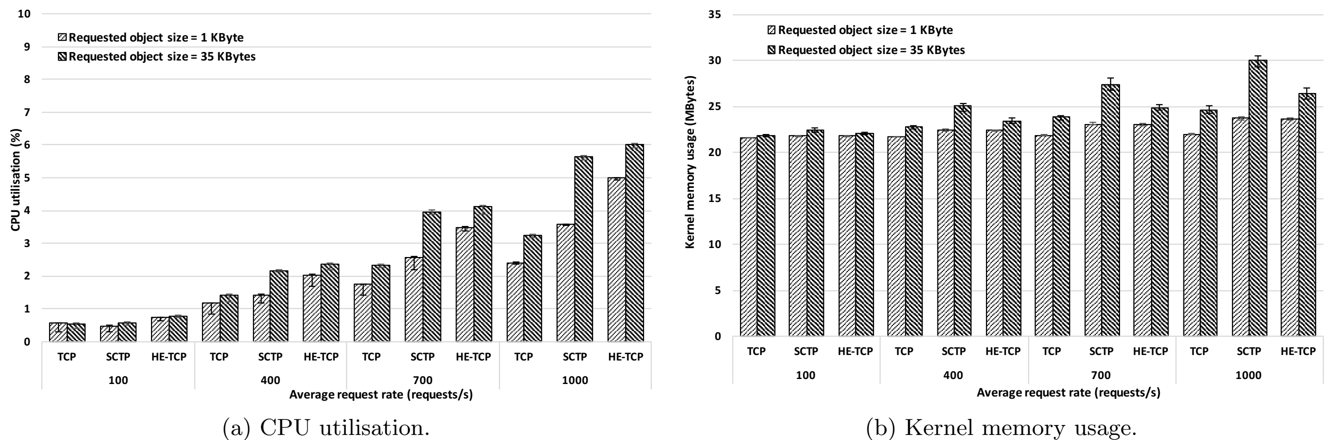
so that the Nagle algorithm was disabled on all listen sockets to assure that there were no additional delays in total connection time. The default configuration parameters of the lighttpd server were used, except those listed in Table 2, which assured that the lighttpd server could efficiently handle the HTTP request rates considered in the experiments and that TLS was always preferred. The *OpenSSL* library v1.0.1e was used for the TLS protocol. The preferred cipher suite was ECDHE-RSA-AES128-GCM-SHA256, while Intel's AES New Instructions (AES-NI) set for hardware accelerated AES operations was utilised [5]. The lighttpd server used the FreeBSD kernel SCTP implementation, and the Web traffic generator used the Linux kernel SCTP implementation.

An experiment run lasted for 600 s, during which the Web traffic generator generated exponentially distributed HTTP requests with a fixed average intensity, and with requested Web object sizes of 1 KiB and 35 KiB. In our experiment, we considered HTTP-request intensities ranging between 100 requests/s and 1000 requests/s. We measured:

- the total CPU load on the server,
- the CPU utilisation of every process that has a substantial contribution to the total CPU time,
- the total kernel memory used for networking.

Per-process CPU utilisation was sampled every 20 s and was calculated based on the accumulated CPU time given by *procstat -r*. The total CPU load on the server was measured by measuring the accumulated CPU time of the *idle* system process (i.e., the total time that the CPU was idle) and subtracting this time from the total available CPU time during the measured interval (i.e., 160 s for an 8 parallel thread CPU). Total kernel memory utilisation was also sampled every 20 s and was calculated based on the output of *vmstat -z*, *vmstat -m*, and *netstat -m*. Table 3 outlines the malloc types and zones that were used to calculate total network-related kernel memory utilisation.

Our experiment comprised three test cases. In the first case, we evaluated a naive HE mechanism that did not employ caching of the outcome of previous happy eyeball invocations and which always resulted in a TCP connection being set up. The rationale behind this case was to serve as a baseline for the remaining two cases. Next, in the second test case, we still considered the same naive HE mechanism as in the first case, however, this time we evaluated happy eyeballing between TLS-encrypted TCP and SCTP. The second test case aimed at providing an appreciation of how the increase in CPU load and memory usage due to happy eyeballing compares with that caused by the TLS encryption itself. Lastly, in the third test case, we evaluated an optimized HE mechanism that employed caching of the outcome of previous connection attempts, using TCP and SCTP both with and without TLS encryption. The purpose behind the third test case was to obtain an understanding of the extent to which HE CPU load and memory usage decrease with caching, and to get a feel for the overhead of HE with a more optimized implementation. In this test case, we considered three different outcomes of the HE mechanism: HE always results in a TCP connection being set up (HE-TCP); HE always results in an SCTP connection being set up (HE-SCTP); and, HE results in a TCP connection being set up half the time and an SCTP connection half the time (HE-50%).



(a) CPU utilisation.

(b) Kernel memory usage.

Figure 3: The results for the basic test case.

3.2 The Basic Test Case

The outcome of the basic test case is shown in Figure 3. Figure 3(a) compares the CPU load of HE between TCP and SCTP with that of single TCP and SCTP connection requests in the basic case. The figure shows how the CPU load varied as a function of the connection request rate and the size of the requested Web objects. The bar charts show the median values measured as described in Section 3.1, with error bars spanning the 10th and 90th percentiles. As follows, the CPU load of HE was quite substantial in the 1-KiB tests, roughly 40% higher CPU load than SCTP (i.e., the transport protocol considered in the study that consumed the most CPU load, in the tests with request rates 1000, 700, and 400 requests/s). However, as is evident from the 35-KiB tests, this was most likely an effect of the small amount of bytes transmitted in each Web response, and thus the small amount of bytes over which the CPU load was amortized: In the 35-KiB tests, the CPU load of HE was less than 10% higher than that of SCTP in the tests with request rates of 1000, 700, and 400 requests/s.

Figure 3(b) examines the kernel memory usage of HE compared with that of single TCP and SCTP connection requests. The bar charts show the median values with error bars spanning the 10th and 90th percentiles. Similar to Figure 3(a), the bar charts illustrate how the kernel memory varied with increasing connection request rates, and for different sizes of the requested Web object. We observe that HE had no or negligible impact on the kernel memory consumption – neither in the 1-KiB tests nor in the 35-KiB tests do we see a significant increase in kernel memory usage. In fact, the 35-KiB tests indicate that as the connection request rate increases, HE (at least in those cases where TCP wins) reduces the kernel memory usage as compared with SCTP.

3.3 Happy Eyeballing in the TLS Test Case

Figure 4 summarises the results from the TLS test case. Figure 4(a) is similar to Figure 3(a), but compares the CPU load in the case with TLS-encrypted connections. We observe that contrary to the basic case, the impact on CPU load of HE as compared with SCTP decreases significantly in the 1-KiB tests (less than 13% in all cases) and is not statistically significant in the 35-KiB tests (less than 4% in the tests with request rates 1000, 700, and 400). Similar observations also apply when compared with TCP, where the impact of HE on CPU load is significantly lower than that in the basic case. Again, the reason HE had less effect on CPU load in this scenario compared with the basic case, was an effect of the way the CPU load was amortized.

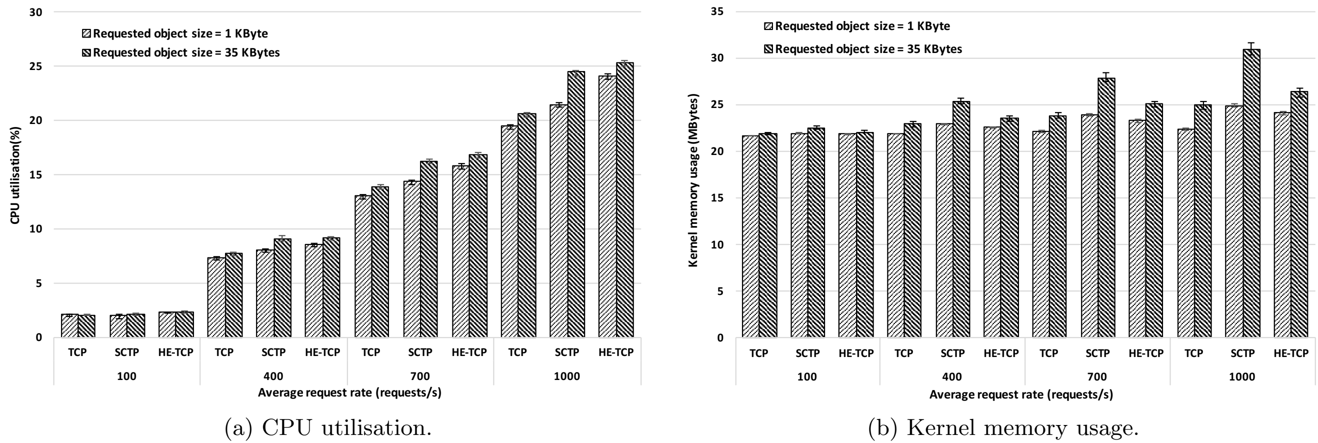
Figure 5 illustrates how the CPU was shared among the kernel (including the FreeBSD subsystem) and the lighttpd server when the HE mechanism is used in the 35-KiB tests; both the tests for the basic case, as well as those for the TLS case. We observe that since the CPU load inflicted by HE was almost the same in both test cases, the CPU load of TLS (as reflected in the increase on the user CPU time of the lighttpd processes) overshadowed that of HE. Thus, in sum, we draw the conclusion that although HE is done at the price of some extra CPU load, the price becomes marginal for larger Web object sizes, and becomes even less significant in those cases HE is done between encrypted connections.

As regards the kernel memory usage in the TLS case, it follows from Figure 4(b) that HE had a marginal impact on this factor in this test case as well: In all tests, the kernel memory usage of HE is slightly higher than that of TCP (less than 8% higher memory usage), and always less than the kernel memory usage of SCTP.

3.4 Happy Eyeballing with Cached Results

In the basic and TLS use cases, we evaluated a naive HE mechanism that always tried both TCP and SCTP. This is, however, a rather inefficient implementation of HE. A more efficient and, as we see it, more realistic implementation would cache the outcome of previous connection attempts. So, e.g., assume that we have a cache hit rate of 80%, then HE tries both TCP and SCTP in only 20% of the application connection requests; in the remaining 80% of the application connection requests, HE issues either a TCP or SCTP connection request depending on the content of the HE cache. A cache hit rate of 80% is actually not an unreasonable figure. In statistics we obtained from Mozilla, they observed a hit rate of $\approx$ 84% in the Firefox internal 'route' cache during a six-week observation period.

Figure 6 shows the median CPU load (with error bars spanning the 10th and 90th percentiles) of HE at different



(a) CPU utilisation. (b) Kernel memory usage.

Figure 4: The results for the TLS test case.

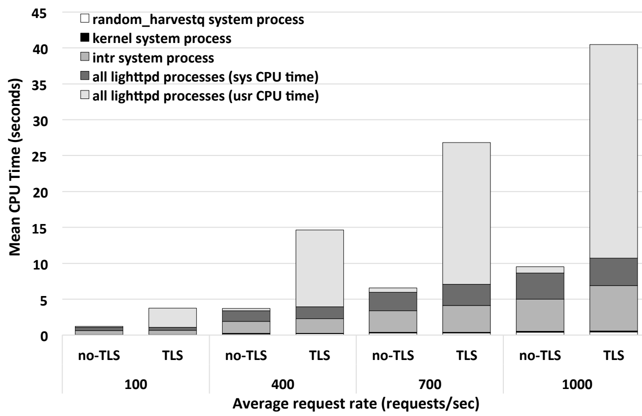


Figure 5: Breakdown of CPU utilisation for happy eyeballing between TCP and SCTP. Requested object size is 35 KiB.

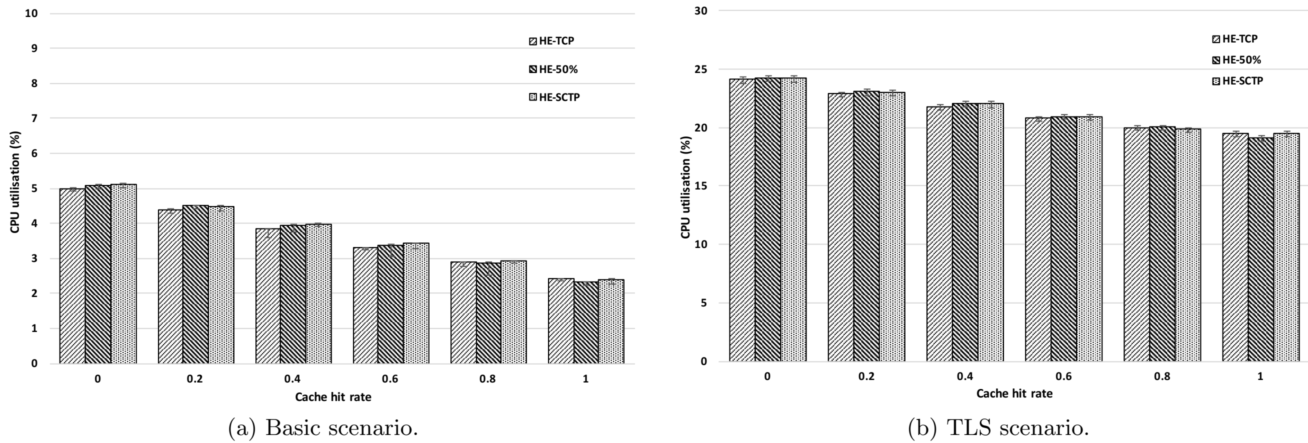
cache hit ratios between 0% (naive HE that always tries both TCP and SCTP) and 100% (single TCP flows) in the 1-KiB tests: Figure 6(a) shows the total CPU load when TLS is not used, and Figure 6(b) when TLS is used. Figure 7 shows the corresponding results for the 35-KiB tests. Since the CPU load increases with increasing connection request rates and is thus more pronounced at higher request rates, we only consider the tests with a request rate of 1000 requests/s in Figures 6 and 7. Still, it should be noted that similar results were obtained for the lower connection request rates. The HE tests considered three outcomes: HE always results in a TCP connection being setup (HE-TCP); HE always results in an SCTP connection being setup (HE-SCTP); and, HE results in a TCP connection being setup half the time and an SCTP connection half the time (HE-50%).

We observe that the CPU load of HE decreases linearly as the cache hit rate increases, and this decrease is higher, percentage-wise, when TLS is not used (in the 1-KiB tests about 43% reduction in the CPU load when the cache hit rate is 80% and TLS is not used, and 18% reduction for TLS encrypted connections and the same cache hit rate). Again, the reason caching had less effect on the decrease of the CPU load when TLS was used, was the effect of the way the CPU load was amortised. We further observe in Figure 6 that irrespective of whether TLS is being used or not, the difference in the CPU load imposed by HE-TCP, HE-50, and HE-SCTP is negligible for the 1-KiB tests. This implies that for small objects the additional cost of the http transaction is almost the same for both TCP and SCTP. For larger objects (e.g., 35 KiB), however, this cost is higher when SCTP is used, and hence significant differences between HE-TCP, HE-50, and HE-SCTP are observed at low cache hit rates in Figure 7.

We omit showing how the cache hit ratio influences the kernel memory consumption, since already the basic and TLS use cases suggest that kernel memory usage is not much of an issue for HE. Still, for completeness, we can mention that the cache hit ratio also had a positive impact on kernel memory usage. For instance, in the 1 KiB tests the kernel memory usage of HE at a cache hit rate of 80% was pretty much the same as for single TCP flows.

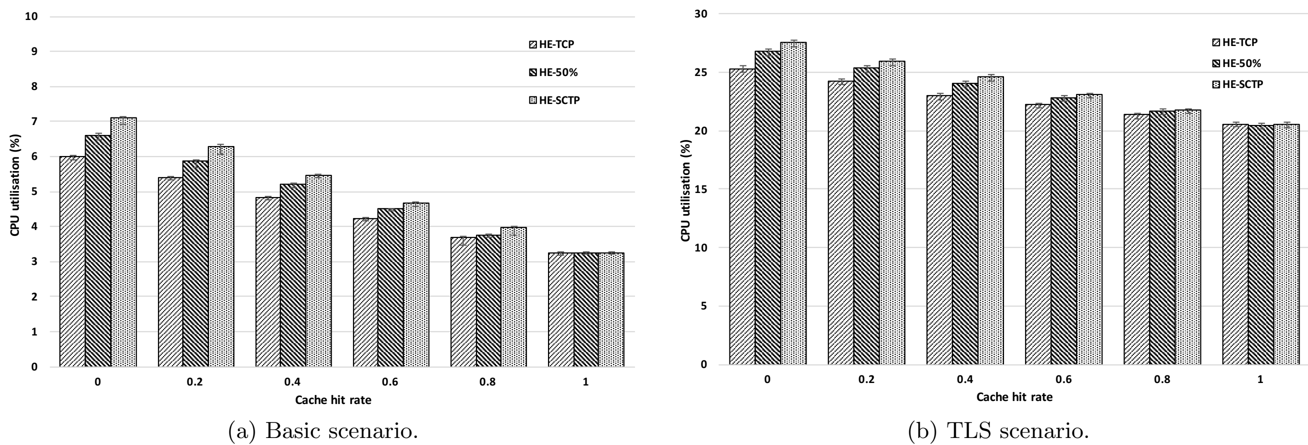
4. CONCLUSIONS

The Happy Eyeballs algorithm was originally proposed, and is currently being deployed, as a way of making a smooth transition from IPv4 to IPv6. However, the algorithm has also been proposed as a transport-selection mechanism. This paper evaluates happy eyeballing between TCP and SCTP, and shows that although HE increases CPU load as compared with a single TCP or SCTP connection establishment, the increase is in the order of 10% for 35 KiB Web objects, i.e., fairly typical Web objects, and is even smaller in those cases the happy eyeballing takes place between TLS-encrypted connections. Moreover, we show that the caching of connection-request results substantially reduces the HE CPU load, especially in comparison with the cost of TLS. As regards memory usage, our results suggest that HE has essentially the same memory footprint as single TCP/SCTP flows. The analysis in this paper shows that integrating a Happy Eyeballs mechanism into a library which provides a generic transport service is indeed a viable option to enable the use of advanced transport-protocol features whenever they are available. An example of such a library is the `neat`



(a) Basic scenario. (b) TLS scenario.

Figure 6: Impact of cache hit ratio on CPU utilisation. Requested object size is 1 KiB with a request rate of 1000 requests/s.



(a) Basic scenario. (b) TLS scenario.

Figure 7: Impact of cache hit ratio on CPU utilisation. Requested object size is 35 KiB with a request rate of 1000 requests/s.

library currently being developed by the NEAT project[2].

Future work includes evaluating the performance of HE when there are more than two transport solutions to be tried, e.g., TLS or DTLS encrypted traffic using IPv4 or IPv6 as the network layer, and TCP, native SCTP, or UDP-encapsulated SCTP as the transport layer, giving a total of six protocol candidates. Already, we note that the use of caching becomes even more important in these cases. Future work would also consider real-world experiments, where middlebox interference can be taken into account, as well as additional metrics to further examine the effects of transport happy eyeballing on both the network and destination end hosts, such as resource consumption on middleboxes, network load, and transaction times.

5. ACKNOWLEDGMENTS

The authors would like to thank Patrick McMcanus (Mozilla) for providing the Firefox cache-hit statistics.

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 authors.

[2]`https://github.com/NEAT-project/neat`

6. REFERENCES

[1] E. Aben. Hampering Eyeballs – Observations on Two "Happy Eyeballs" Implementations. RIPE NCC, Nov. 2011. `https://labs.ripe.net/Members/emileaben/hampered-eyeballs`.

[2] V. Bajpai and J. Schoenwaelder. Measuring the effects of happy eyeballs. Internet Draft draft-bajpai-happy, work in progress, July 2013. `https://tools.ietf.org/html/draft-bajpai-happy`.

[3] F. Baker. Testing Eyeball Happiness. RFC 6556 (Informational), Apr. 2012.

[4] O. Bonaventure. Happy eyeballs makes me unhappy..., Dec. 2013. `http://perso.uclouvain.be/olivier.bonaventure/blog/html/2013/12/03/happy.html`.

[5] S. Gueron. Intel® Advanced Encryption Standard (AES) New Instructions Set. *Intel Corporation*, 2012.

[6] httperf. The httperf page on SourceForge. `https://sourceforge.net/projects/httperf`.

[7] G. Huston. Bemused Eyeballs: Tailoring Dual Stack Applications for a CGN Environment. The ISP Column, May 2012. `http://www.potaroo.net/ispcol/2012-05/notquite.html`.

[8] IETF. Transport Services (taps) Working Group. `https://datatracker.ietf.org/wg/taps/charter/`.

[9] Lighttpd. Lighttpd – fly light. `https://www.lighttpd.net`.

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

[11] D. Wing and A. Yourtchenko. Happy Eyeballs: Trending Towards Success (IPv6 and SCTP). Internet Draft draft-wing-tsvwg-happy-eyeballs-sctp-02, work in progress, Oct. 2010. `https://tools.ietf.org/html/draft-wing-tsvwg-happy-eyeballs-sctp-02`.

[12] D. Wing and A. Yourtchenko. Improving User Experience with IPv6 and SCTP. *The Internet Protocol Journal*, 13(3), Sept. 2010. `http://www.cisco.com/c/en/us/about/press/internet-protocol-journal/back-issues/table-contents-49/133-he.html`.

[13] D. Wing and A. Yourtchenko. Happy Eyeballs: Success with Dual-Stack Hosts. RFC 6555 (Proposed Standard), Apr. 2012.

[14] D. Wing, A. Yourtchenko, and P. Natarajan. Happy eyeballs: Trending towards success (IPv6 and SCTP). Internet Draft draft-wing-http-new-tech, work in progress, Aug. 2010. `https://tools.ietf.org/html/draft-wing-http-new-tech-01`.

Disclaimer

The views expressed in this document are solely those of the author(s). The European Commission is not responsible for any use that may be made of the information it contains.

All information in this document is provided “as is”, and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.