neat

NEAT

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

H2020-ICT-05-2014
Project number: 644334

Deliverable D2.1

First Version of Low-Level Core Transport System

Editor(s): Naeem Khademi
Contributor(s): Zdravko Bozakov, Anna Brunstrom, Dragana Damjanovic, Kristian Riktor Evensen, Gorry Fairhurst, Karl-Johan Grinnemo, Tom Jones, Simone Mangiante, Giorgos Papastergiou, David Ros, Michael Tüxen, Michael Welzl

Work Package: 2 / Core Transport System
Revision: 1.0
Date: March 1, 2016
Deliverable type: R (Report)
Dissemination level: Public

Abstract

This document presents the first version of the low-level Core Transport System in NEAT, to be used for development of a reference implementation of the NEAT System. 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 basic Transport Services provided by the API, a set of low-level 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*. Some other components are needed to ensure connectivity using *Middlebox Traversal* techniques (e.g., TURN), discovery of path support for different transport protocols using *Happy Eyeballs* mechanisms, offering end-to end *Security* (e.g., (D)TLS over transport), gather *statistics* for the users or system administrators, and the ability to apply different *policies* in order to influence the decision-making process of the transport system. This document describes each of these building blocks and related design choices.

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

Contents

List of Abbreviations

AAA Authentication, Authorisation and Accounting

AAAA Authentication, Authorisation, Accounting and Auditing

API Application Programming Interface

CIB Characteristics Information Base

DCCP Datagram Congestion Control Protocol

DNS Domain Name System

DNSSEC Domain Name System Security Extensions

DSCP Differentiated Services Code Point

DTLS Datagram Transport Layer Security

ECN Explicit Congestion Notification

ENUM Electronic telephone number mapping

HTTP HyperText Transfer Protocol

IAB Internet Architecture Board

ICE Internet Connectivity Establishment

IETF Internet Engineering Task Force

IF Interface

IGD-PCP Internet Gateway Device – Port Control Protocol

IP Internet Protocol

IRTF Internet Research Task Force

KPI Kernel Programming Interface

LAN Local Area Network

LBE Less than Best Effort

MIF Multiple Interfaces

MPTCP Multipath Transmission Control Protocol

MTU Maximum Transmission Unit

NAT Network Address (and Port) Translation

NEAT New, Evolutive API and Transport-Layer Architecture

OS Operating System

PCP Port Control Protocol

PDU Protocol Data Unit

PI Policy Interface

PIB Policy Information Base

PM Policy Manager

POSIX the Portable Operating System Interface

PMTU Path MTU

QoS Quality of Service

RFC Request for Comments

RTT Round Trip Time

RTP Realtime Protocol

RTSP Realtime 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

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

TTL Time To Live

TURN Traversal Using Relays around NAT

UDP User Datagram Protocol

UPnP Universal Plug and Play

URI Uniform Resource Identifier

VPN Virtual Private Network

WAN Wide Area Network

1 Introduction

The NEAT System aims to change the transport layer interface in a way that Internet applications could 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 [14]. The NEAT *Core Transport System* plays a vital role in translating the 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 *Low-Level Core Transport System* the set of the most basic building blocks necessary to provide NEAT Transport Services described in 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 finally the ability to expose connection-level or system-wide statistics to an application.

This document reports on the first version of the low-level transport system being implemented in NEAT. The rest of this section provides a general discussion of low-level and high-level functions (§ 1.1), as well as a short overview of the NEAT architecture introduced in D1.1 [8] (§ 1.2). It also discusses the Transport Services provided by NEAT (§ 1.3), and briefly presents the basic building blocks, or *NEAT Components*, that comprise the low-level core transport system (§ 1.4).

Section 2 discusses each of the low-level 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 when necessary. Finally, Section 3 draws conclusions from this document and points at future work as part of Work Package 2.

1.1 Low-level and high-level functions

The NEAT System is a large and complex system comprising a plethora of components that need to interact with each other, which aims to provide a wide range of Transport Service Features in a flexible and evolvable way. To effectively handle the complexity of the NEAT System and facilitate the development of its reference implementation, a conceptual framework of two levels of abstraction has been considered for the design and development of its core transport system. In this context, the NEAT core transport system is composed of two different sets of building blocks: low-level components that comprise the *Low-Level Core Transport System* and high-level components that comprise the *High-Level Core Transport System*.

Low-level building blocks implement a set of fundamental functions (*low-level functions*) that are essential for the creation, utilisation and evolution of the NEAT System. These functions do not build upon any high-level functions and exhibit the lowest possible level of abstraction. High-level building blocks implement more advanced functions (*high-level functions*) that are normally built upon the low-level building blocks and present a higher level of abstraction; this means, realising a high-level transport functionality typically involves selecting and combining the proper low-level transport functionalities available on the system.

1.2 Overview of the NEAT Architecture

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

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

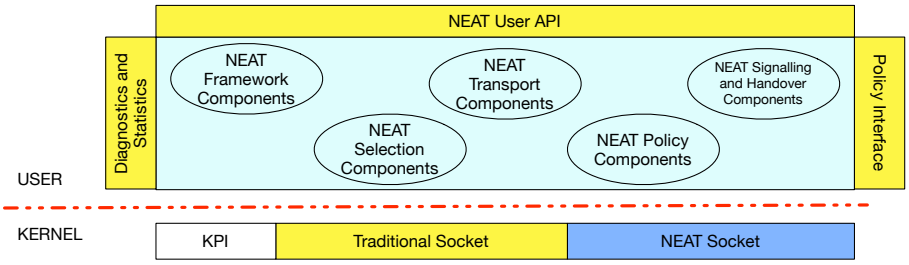

Figure 1: Component groups and interfaces used to realise the NEAT User Module.

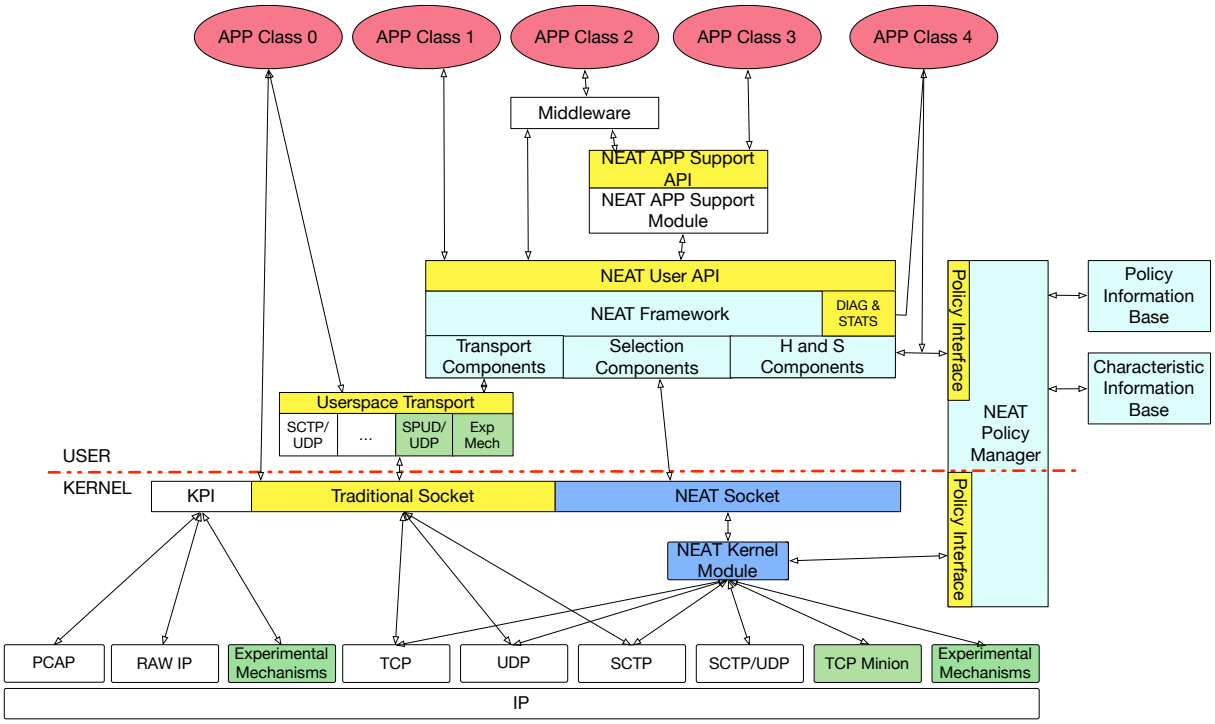

Figure 2: Components and interfaces to the NEAT System. The NEAT User Module is composed of all the blocks shown in light blue (NEAT Framework, NEAT Transport, NEAT Selection, NEAT Signalling and Handover, and Policy Components) and related APIs (NEAT User API, Policy Interface, Diagnostics and Statistics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

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 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 describes 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 mechanisms to probe/signal 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 also form a 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 OS 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 will allow 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.

1.3 Overview of the services provided by the NEAT API

Internet drafts from the IETF TAPS working group [7, 13], 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 clients.

Deliverable D1.2 [14] presents two sets of Transport Service Features: (a) Transport Service Features derived from draft-ietf-taps-transports-usage [13]; and (b) Transport Service Features derived from use cases in D1.1 [8]. Set (a) includes Transport Service Features that can be utilised using primitives and events derived from transport-protocol APIs. This includes TCP and SCTP in the current version and will be extended to cover more transport protocols in the future. 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) Transport Service Features that are associated with the information passed from the NEAT System to the application; and (2) Transport Service Features that are associated with the information passed from the application to the NEAT System.

The low-level core transport system is the set of components necessary to provide a “platform” to implement these Transport Service Features. We will summarise them in Section 1.4 and provide a more elaborate description in Section 2.

1.4 Overview of the low-level components required to provide the services

To offer the Transport Service Features presented in D1.2 [14], a set of low-level components are needed for the NEAT core transport system. These building blocks do not provide by themselves the

full range of Transport Service Features described in D1.2, but they are essential for the core operation of the NEAT System . This makes it possible for the NEAT developers to provide more Transport Service Features as the transport system evolves during the project. The list of core building blocks will be extended in D2.2 (expanding the low-level core transport system if needed, and including the high-level core transport system) and finalised in D2.3.

Based on the sets of NEAT Components defined in D1.1, the low-level 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 Endpoint*, 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 building blocks ensure connectivity using *Middlebox Traversal* mechanisms, by employing NAT traversal techniques such as TURN [11]. These also include the possibility to use DTLS over SCTP as well as TLS over TCP when *Security* is being requested.
- **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. Happy Eyeballs can be done between different transport protocols (e.g., SCTP/TCP) or IP versions (IPv4/IPv6).
- **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 uses 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 low-level components and their potential dependency on each other. As seen on the list above, low-level components are found in four out of the five component groupings in Figure 1 (Framework, Transport, Selection, and Policy); NEAT Signalling and Handover is part of the Extended Transport System under development in Work Package 3, and therefore out of the scope of this document.

2 Low-level Transport Functions

This section presents a detailed description of low-level transport functionalities required to realise the NEAT core transport system based on the components introduced in Section 1.4. Each component is described in detail and Transport Service Features they provide are specified when applicable[2]. Indicative examples of their operation are given where relevant, and relationships among these building 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. When appropriate, snippets of sample code in C are used for better presentation.

[2]The Transport Service Features mentioned in this deliverable are identified in Tables 1 and 2 of Deliverable 1.2 [14].

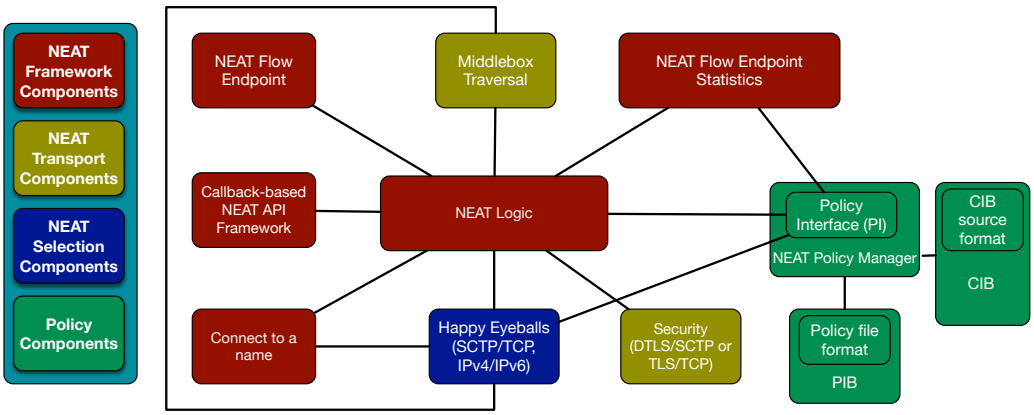

Figure 3: Low-level building blocks for the NEAT low-level core transport system. Each colour denotes a different component grouping.

2.1 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.

2.1.1 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 [4]. This is mainly used by the NEAT Logic to collect information about a NEAT Flow. 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 and keeps the information about the socket that is relevant during the NEAT Flow's lifetime (Listing 1). This includes:

- File descriptor for the underlying OS socket.
- Socket address family (e.g., AF_INET, AF_INET6).
- Socket type (e.g., SOCK_DGRAM, SOCK_STREAM).
- Protocol (e.g., IPPROTO_UDP, IPPROTO_TCP, IPPROTO_SCTP).

- Remote peer domain name.
- Remote socket address.
- Remote port.

```
1 struct neat_flow
2 {
3     int fd;
4     uint8_t family;
5     int sockType;
6     int sockProtocol;
7     const char *name;
8     const struct sockaddr *sockAddr;
9     const char *port;
```

Listing 1: NEAT Flow Endpoint structure.

It also contains information relevant to the NEAT System operations (e.g., DNS resolver outcome) as well as a backtracking log of NEAT System decisions (e.g., application requirements, connection attempts) and includes (Listing 2):

- Address resolver results, DNS results.
- *Requested properties* for the NEAT Flow: these are properties that an application has specified at the NEAT Flow creation time via the NEAT User API. The available properties in the current implementation are listed in Table 1 and the relevant code is in Listing 3.
- *Attempted properties:* the NEAT Policy Manager selects a subset of the application requirements and translates them into concrete socket configurations. The Happy Eyeballs component will probe these selected socket configurations. The subset of requirements attempted by Happy Eyeballs will be stored in the NEAT Flow Endpoint structure.
- *Used properties:* these are the properties from the list of the requested properties that apply to an established connection. They are present only if the underlying socket is successfully connected.

```
11     struct neat_resolver_results *resolver_results;
12     uint64_t propertyMask;
13     uint64_t propertyAttempt;
14     uint64_t propertyUsed;
```

Listing 2: NEAT Flow Endpoint structure (continued).

```
1 #define NEAT_PROPERTY_OPTIONAL_SECURITY          (1 << 0)
2 #define NEAT_PROPERTY_REQUIRED_SECURITY          (1 << 1)
3 #define NEAT_PROPERTY_MESSAGE                    (1 << 2) // stream is default
4 #define NEAT_PROPERTY_IPV4_REQUIRED              (1 << 3)
5 #define NEAT_PROPERTY_IPV4_BANNED                (1 << 4)
6 #define NEAT_PROPERTY_IPV6_REQUIRED              (1 << 5)
7 #define NEAT_PROPERTY_IPV6_BANNED                (1 << 6)
8 #define NEAT_PROPERTY_SCTP_REQUIRED              (1 << 7)
9 #define NEAT_PROPERTY_SCTP_BANNED                (1 << 8)
```

```
10 #define NEAT_PROPERTY_TCP_REQUIRED                (1 << 9)
11 #define NEAT_PROPERTY_TCP_BANNED                  (1 << 10)
12 #define NEAT_PROPERTY_UDP_REQUIRED                (1 << 11)
13 #define NEAT_PROPERTY_UDP_BANNED                  (1 << 12)
14 #define NEAT_PROPERTY_UDPLITE_REQUIRED            (1 << 13)
15 #define NEAT_PROPERTY_UDPLITE_BANNED              (1 << 14)
16 #define NEAT_PROPERTY_CONGESTION_CONTROL_REQUIRED (1 << 15)
17 #define NEAT_PROPERTY_CONGESTION_CONTROL_BANNED   (1 << 16)
18 #define NEAT_PROPERTY_RETRANSMISSIONS_REQUIRED    (1 << 17)
19 #define NEAT_PROPERTY_RETRANSMISSIONS_BANNED      (1 << 18)
```

Listing 3: NEAT Flow properties.

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 4). Their usage will be clarified in § 2.1.2.

```
15     struct neat_flow_operations *operations;
```

Listing 4: 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 5.

```
16     // NEAT base loop structure:
17     struct neat_ctx *ctx;
18     uv_poll_t handle;
19
20     // Functions for accessing the underlying socket:
21     neat_read_impl readfx;
22     neat_write_impl writefx;
23     neat_accept_impl acceptfx;
24     neat_connect_impl connectfx;
25     neat_close_impl closefx;
26     neat_listen_impl listenfx;
27
28     // NEAT internal flags:
29     int firstWritePending : 1;
30     int acceptPending : 1;
31     int isPolling : 1;
32     int ownedByCore : 1;
33     int everConnected : 1;
34     int isDraining : 1;
35
36     // Write buffer:
37     unsigned char *buffered;
38     ssize_t bufferedOffset;
39     ssize_t bufferedSize;
40     ssize_t bufferedAllocation;
41 }
```

Listing 5: NEAT Flow Endpoint structure (continued).

Table 1: Description of currently-implemented NEAT Flow properties.

Property[a]	**Description**
NEAT_PROPERTY_OPTIONAL_SECURITY	If this property is set, security is optional.
NEAT_PROPERTY_REQUIRED_SECURITY	If this property is set, security is required. If NEAT_PROPERTY_OPTIONAL_SECURITY is set as well it will be ignored.
NEAT_PROPERTY_MESSAGE	Requests a message-based protocol. A stream-based protocol will be used if this property is not set.
NEAT_PROPERTY_IPV4_REQUIRED	Only IPv4 addresses will be used. If an IPv4 address is not present, invoking neat_open will result in an error.
NEAT_PROPERTY_IPV4_BANNED	Do not use IPv4 addresses.
NEAT_PROPERTY_IPV6_REQUIRED	Only IPv6 addresses will be used. If an IPv6 address is not present, invoking neat_open will result in an error.
NEAT_PROPERTY_IPV6_BANNED	Do not use IPv6 addresses.
NEAT_PROPERTY_SCTP_REQUIRED	NEAT System will try only the SCTP protocol.
NEAT_PROPERTY_SCTP_BANNED	NEAT System will not try the SCTP protocol.
NEAT_PROPERTY_TCP_REQUIRED	NEAT System will try only the TCP protocol.
NEAT_PROPERTY_TCP_BANNED	NEAT System will not try the TCP protocol.
NEAT_PROPERTY_UDP_REQUIRED	NEAT System will try only the UDP protocol.
NEAT_PROPERTY_UDP_BANNED	NEAT System will not try the UDP protocol.
NEAT_PROPERTY_UDPLITE_REQUIRED	NEAT System will try only the UDP-Lite protocol.
NEAT_PROPERTY_UDPLITE_BANNED	NEAT System will not try the UDP-Lite protocol.
NEAT_PROPERTY_CONGESTION_CONTROL_REQUIRED	NEAT System will use a transport protocol that offers congestion control.
NEAT_PROPERTY_CONGESTION_CONTROL_BANNED	NEAT System will use a transport protocol that does not offer a congestion control feature.
NEAT_PROPERTY_RETRANSMISSIONS_REQUIRED	NEAT System will use a transport protocol that offers packet retransmission.
NEAT_PROPERTY_RETRANSMISSIONS_BANNED	NEAT System will use a transport protocol that does not offer a packet retransmission feature.

[a] If the same REQUESTED and BANNED (e.g. NEAT_PROPERTY_IPV4_REQUIRED and NEAT_PROPERTY_IPV4_BANNED) property are set, a call to neat_open will return an error. The same result will be seen if two or more of NEAT_PROPERTY_SCTP_REQUIRED, NEAT_PROPERTY_TCP_REQUIRED, NEAT_PROPERTY_UDP_REQUIRED and NEAT_PROPERTY_UDPLITE_REQUIRED are set.

In the current implementation, each NEAT Flow Endpoint structure corresponds to a single operating system socket and vice versa. When a foreseen extension of using SCTP multi-streaming is added as part of Work Package 3, a NEAT Flow Endpoint structure could correspond to a single SCTP stream and multiple NEAT Flows could communicate over a single SCTP socket. A final design decision on how to implement a server-side socket that listens on multiple ports and uses multiple transport/network protocols has not been made yet; depending on the outcome, the 1-to-1 mapping of a NEAT Flow and a socket could change in this case as well.

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.

Some examples of the operation: Listing 6 shows an example of how the NEAT Flow Endpoint structure is used. In this example the Happy Eyeballs will set the remote host address after the host address is resolved. For simplicity, the address selection process is omitted. The `neat_resolver` structure is an internal structure of the NEAT System and it contains two variables: a pointer to the NEAT Flow Endpoint structure and a pointer to the callback function. At line 18 the address parameter is set and at line 24 a callback function is called informing of a successful address resolution.

```
1  static void
2  he_resolve_cb(struct neat_resolver *resolver, struct neat_resolver_results *results,
       uint8_t code)
3  {
4      neat_flow *flow = (neat_flow *)resolver->userData1;
5      neat_he_callback_fx callback_fx;
6      callback_fx = (neat_he_callback_fx) (neat_flow *)resolver->userData2;
7
8      if (code != NEAT_RESOLVER_OK) {
9          callback_fx(resolver->nc, (neat_flow *)resolver->userData1, code,
10                     0, 0, 0, -1);
11         return;
12     }
13
14     assert (results->lh_first);
15     assert (!flow->resolver_results);
16
17     // In this example use the first address.
18     flow->family = results->lh_first->ai_family;
19     flow->sockType = results->lh_first->ai_socktype;
20     flow->sockProtocol = results->lh_first->ai_protocol;
21     flow->resolver_results = results;
22     flow->sockAddr = (struct sockaddr *) &(results->lh_first->dst_addr);
23
24     callback_fx(resolver->nc, (neat_flow *)resolver->userData1, NEAT_OK,
25                 flow->family, flow->sockType, flow->sockProtocol, -1);
26 }
```

Listing 6: Use of the NEAT Flow Endpoint structure.

Related building blocks:

- NEAT Logic (§ 2.1.3).

2.1.2 NEAT API Framework (callback)

The NEAT System implements a callback-based API. 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 [1] 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 7):

- Creating and destroying a NEAT Flow, i.e., a NEAT Flow Endpoint structure. (§ 2.1.1)
- Functions for getting and setting properties: using these functions, an application can set or change transport requirements (e.g., reliable transport, low latency, etc.) as well as change run-time parameters (e.g., flow priority).
- 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.
- 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.
- 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.
- Setting callback functions.

```
1 struct neat_flow *neat_new_flow(struct neat_ctx *ctx);
2 void neat_free_flow(struct neat_flow *flow);
3
4 neat_error_code neat_set_operations(struct neat_ctx *ctx, struct neat_flow *flow,
      struct neat_flow_operations *ops);
5 neat_error_code neat_open(struct neat_ctx *ctx, struct neat_flow *flow, const char *
      name, const char *port);
6 neat_error_code neat_read(struct neat_ctx *ctx, struct neat_flow *flow, unsigned
      char *buffer, uint32_t amt, uint32_t *actualAmt);
7 neat_error_code neat_write(struct neat_ctx *ctx, struct neat_flow *flow, const
      unsigned char *buffer, uint32_t amt);
8 neat_error_code neat_get_property(struct neat_ctx *ctx, struct neat_flow *flow,
      uint64_t *outMask);
9 neat_error_code neat_set_property(struct neat_ctx *ctx, struct neat_flow *flow,
      uint64_t inMask);
10 neat_error_code neat_accept(struct neat_ctx *ctx, struct neat_flow *flow, const char
       *name, const char *port);
```

Listing 7: 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 8.

```
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;
```

Listing 8: 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 requirement 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 requested by calling the `neat_open` function (corresponding to the OPEN primitive from D1.2 [14]). The function takes a host name and a port number as parameters. On the server side, the `neat_accept` function (corresponding to the ACCEPT primitive from D1.2 [14]) will be called with parameters: port number and local address the socket should be listening to.

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.

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.

Some examples of the operation: Listing 9 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 line 38. `neat_open` is called in line 45 and if it does not return an error the NEAT loop will be started (line 46). 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      // set callbacks
35      ops.on_connected = on_connected;
36      ops.on_error = on_error;
37
38      if (neat_set_operations(ctx, flow, &ops)) {
39          debug_error("neat_set_operations");
40          result = EXIT_FAILURE;
41          goto cleanup;
42      }
43
44      // wait for on_connected or on_error to be invoked
45      if (neat_open(ctx, flow, argv[argc - 2], argv[argc - 1]) == NEAT_OK) {
46          neat_start_event_loop(ctx, NEAT_RUN_DEFAULT);
47      } else {
48          debug_error("neat_open");
49          result = EXIT_FAILURE;
50          goto cleanup;
51      }
52
53      ...
54  }
```

Listing 9: NEAT API Framework example.

Related building blocks:

- NEAT Logic (§ 2.1.3)

2.1.3 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 [14]) to the primitives provided by each transport protocol.

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.

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 [14]), is used to open a NEAT Flow. The OPEN primitive does not specify any specific transport protocol. After initialising the NEAT Flow's address name and port, `neat_open` registers a Happy Eyeballs callback function (`open_he_callback`) as an argument of `neat_he_lookup` to return the outcome of the Happy eyeballs lookup as shown in Listing 10.

```
1  neat_error_code
2  neat_open(neat_ctx *mgr, neat_flow *flow, const char *name, const char *port)
3  {
4
5  ...
6
7      flow->name = strdup(name);
8      flow->port = strdup(port);
9  ...
10     return neat_he_lookup(mgr, flow, open_he_callback);
11 }
```

Listing 10: NEAT open function.

Related building blocks:

- NEAT Flow Endpoint Statistics (§ 2.1.5).
- Middlebox Traversal (§ 2.2.1).
- NEAT Flow Endpoint (§ 2.1.1).

- NEAT API Framework (callback) (§ 2.1.2).
- Connect to a name (§ 2.1.4).
- Happy Eyeballs (§ 2.3.1).
- Security (§ 2.2.2).
- NEAT Policy Manager (via Policy Interface) (§ 2.4.1).

2.1.4 Connect to a name

Connect to a name is the address resolver in the NEAT System. We initially only 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 during the lifetime of the NEAT project. An IP literal can also be provided, in which case no translation will be provided.

The resolver fully 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.

Interface/address tuples 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 § 2.1.1 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.

Connecting to a name is fairly straightforward, but there are at least two items that will be discussed and explored further as the design and implementation progress. The first is how the resolver can be used to optimise Happy Eyeballs. If a name is resolved to an AAAA record over an interface that has a IPv6 address, IPv6 will most likely work. Thus, assuming IPv6 is given priority, it will in most cases be redundant to perform IPv4/IPv6 Happy Eyeballs. Having an IPv4 fallback is also trivial.

The second point that will be explored further is which interfaces should be used when resolving names. For instance, should an interface that only returns internal addresses be *banned* (unless internal addresses are desirable)? Also, the policies could be used as input for filtering out interfaces.

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 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.

The current prototype of this building block does not perform any sorting of the resolver results, but this functionality will be added in a next version. As a minimum requirement the sorting described in RFC 3493 [9] should be implemented.

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 (§ 2.1.3), but several building blocks may depend on the offered functionality. One example is Happy Eyeballs (§ 2.3.1) that must be provided with a set of source/destination addresses to probe for IPv4/IPv6 and transport-protocol connectivity.

2.1.5 NEAT Flow Endpoint Statistics

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.

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.

Some examples of the operation: 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 (§ 2.1.3).
- Policy Manager (via Policy Interface) (§ 2.4.1).

2.2 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.

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).

2.2.1 Middlebox Traversal

The NEAT System needs to allow a transport to traverse a network path that may contain one or more middleboxes. This requires additional functions to be implemented in the system. Also, the NEAT System needs to enable peer-to-peer applications to work on all network types transparently. There are a number of network scenarios where NATs or middleboxes will refuse to pass certain types of network traffic, and inbound connections are difficult to establish through certain types of NAT. The initial work has explored options for integrating such mechanisms into the set of NEAT Transport Components.

A range of approaches are possible: methods that discover which transports can be used on a path (addressed by NEAT Selection components), proxy methods (e.g., signalling components that utilise a proxy in the network), encapsulation methods (e.g., transport components that allow one transport to be used over another), and methods to communicate with middleboxes (e.g. signalling components that enable explicit communication with a middlebox on the network path).

Two proxy methods were explored: SOCKS and TURN.

- *SOCKS:* A SOCKS [10] proxy was a common NAT traversal method to allow establishment of external connections through residential ISPs in the past, although less common now. More commonly now though, SOCKS is used as a means of tunnelling traffic through another application since support for SOCKS has been added to many applications. Examples include OpenSSH and Tor (TCP anonymous overlay network) clients, which both provide SOCKS interfaces to the tunnels they create.

- *TURN:* The TURN protocol [11] enables an endpoint to communicate with a peer endpoint, where one or both are behind a restrictive NAT. Communication is achieved using an explicitly chosen proxy to relay the transport protocol. TURN can utilise UDP-based encapsulation methods to support the middlebox traversal.

Current work in the NEAT project has investigated the code-base for integrating support for a SOCKS [10] proxy, or for adding support to use a TURN server. These are potential candidates for a next generation of the Happy Eyeballs component. Future work will design, implement, and integrate an appropriate UDP-based encapsulation method for the NEAT System.

The presence of middleboxes can create connectivity issues through two basic mechanisms:

- *Essential manipulation of packets:* An essential manipulation is something the middlebox was explicitly deployed to do.

- *Accidental manipulation:* A side effect of an essential manipulation, an effect of an implementation error in a middlebox, or an effect of a configuration or deployment error in a middlebox.

Accidental manipulations arise from a mismatch between the actual traffic on the network and the assumptions made by the designers of the middlebox about that traffic. These tend to persist in the network, given the long development and deployment cycles of networking equipment.

Middlebox Traversal will implement approaches the NEAT System will use to work around middleboxes which may corrupt or completely block traffic from a NEAT System through accidental manipulation, and will implement probing techniques used by the NEAT System to detect middleboxes.

There are two functions that this building block provides:

- Middlebox traversal using a substrate protocol.
- Ability to use in-network functionality through the use of a substrate protocol for middlebox cooperation.

Future work in Work Package 3 will explore how a NEAT Signalling and Handover component can build on the encapsulation method to enable the NEAT System to cooperate with middleboxes on a path where they have been detected by probing, and to explore whether such cooperation could be beneficial. This building block may utilise the NEAT Policy Manager to access path resources within the CIB.

To enable middlebox cooperation, we will collaborate with the Horizon 2020 MAMI Project[3]. MAMI will develop an architecture providing a shim layer that contains the Middlebox Cooperation Protocol (MCP), which allows transport and application protocols to selectively expose semantic information to middleboxes while maintaining protocol level details inside an encrypted encapsulation protocol. This shim layer once developed can be used as part of the NEAT System to provide signalling to middleboxes on path to enable traversal or to take advantage of beneficial in-network functionality.

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

Related building blocks:

- NEAT Logic (§ 2.1.3).
- Happy Eyeballs (§ 2.3.1).

2.2.2 Security

The Security component of the NEAT System offers end-to-end encrypted communication for the applications using the NEAT System. Where complete security with integrity, authentication and encryption is not possible, opportunistic security will be supplied offering data confidentiality and integrity.

The NEAT System will provide secure connections using the TLS [5] and DTLS [12] protocols. TLS will be used over TCP and DTLS for SCTP, UDP and UDPLite. More precisely, the possible combinations that NEAT is expected to provide access to are: TLS/TCP/IP, DTLS/UDP/IP, DTLS/UDPLite/IP, DTLS/SCTP/UDP/IP for a user-space SCTP stack, and DTLS/SCTP/IP for a kernel-level SCTP stack. Supporting an SCTP/DTLS/UDP/IP stack might be optionally considered. The OpenSSL library [2] that offers TLS and DTLS support will be used.

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

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

- 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.
- A secure connection is requested without a certificate verification: a certificate verification will not be performed.
- A secure connection is optional: if a secure connection cannot be established the NEAT System will not return an error, instead continue to use an insecure connection.
- 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 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) will be specified as a policy. Certificates and private key files will be specified as a NEAT Flow parameter.

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.

Security will be added in the NEAT architecture as an additional layer above the operating system socket. The NEAT Logic will call 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 will perform TLS or DTLS handshake and certificate verification depending on application requirements.

The NEAT System can be extended to use TCPINC [3] as soon as it is made available.

Provided Transport Service Feature(s):

- NEAT flow security.

Related building blocks:

- NEAT Logic (§ 2.1.3).

2.3 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. In the initial core transport system, Happy Eyeballs is the only Selection component.

Algorithm 1 The Happy Eyeballs Algorithm

```
1: function HAPPYEYEBALLSCOMPONENT(in listOfCandidates : list of transport solutions, out selectedConnection : transport
   connection)
2:     Require: listOfCandidates is sorted in priority order
3:     if length(listOfCandidates) = 0 then
4:         return NONE;
5:     else if length(listOfCandidates) = 1 then
6:         return doConnectionAttempt(listOfCandidates.first());
7:     else
8:         currentCandidate ← listOfCandidates.first();
9:         threadCount ← 0;
10:        repeat
11:            spawnConnectionAttempt(currentCandidate);
12:            threadCount ← threadCount + 1;
13:            nextCandidate ← listOfCandidates.nextCandidate(currentCandidate);
14:            currentPriority ← getPriority(currentCandidate);
15:            nextPriority ← getPriority(nextCandidate);
16:            if currentPriority > nextPriority then
17:                sleep(currentPriority,nextPriority);
18:            end if
19:            currentCandidate ← nextCandidate;
20:        until currentCandidate = listOfCandidates.last()
21:        spawnConnectionAttempt(currentCandidate);
22:        threadCount ← threadCount + 1;
23:        repeat
24:            selectedConnection ← waitFirstConnectionAttempt();
25:            if selectedConnection ≠ NONE then
26:                return selectedConnection
27:            else
28:                threadCount ← threadCount - 1;
29:            end if
30:        until threadCount = 0
31:    end if
32:    return NONE
33: end function

34: function DOCONNECTIONATTEMPT(in transportSolution : transport solution, out connection : transport connection)
35:    connection ← tryConnection(transportSolution);
36:    if connection ≠ NONE then
37:        policyManager.cacheResultConnectionAttempt(transportSolution, SUCCESS);
38:        return connection
39:    else
40:        policyManager.cacheResultConnectionAttempt(transportSolution, FAILURE);
41:        return NONE
42:    end if
43: end function
```

2.3.1 Happy Eyeballs

The Happy Eyeballs building block is part of the NEAT User Module and comprises one of the NEAT Selection components. In the first part of the NEAT selection process, 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, with a higher priority assigned to those that are considered most appropriate. The list is sorted in descending order on the basis of the priority.

The pseudo-code for the current version of Happy Eyeballs is presented in Algorithm 1. The component takes as input the list of candidate transport solutions created in the first part of the NEAT selection process. It concurrently tries out each of the candidate transport solutions in the list and finally returns a handle for the first successfully established connection.

It should be noted that the Happy Eyeballs algorithm is also able to handle the cases in which there

are no or only one candidate transport solution, i.e., cases when no "happy eyeballing" is actually needed. In the former case the component returns with no selected transport solution, i.e., it tells the caller that it failed to set up a NEAT Flow. In the latter case the outcome depends on whether or not the component is able to establish a NEAT Flow using the single transport solution on the candidate list.

As remarked in RFC 6555 [15], 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 configurable time after which they become invalid and have to be repeated. The impact and efficiency of the Happy Eyeballs mechanism is being explored as part of the evaluation of the core transport system, and the results will serve as input for future refinement of the algorithm.

Some examples of the operation: As an example of how Happy Eyeballs works, consider the scenario of a dual-stack IPv4/IPv6 client trying to setup a connection to a dual-stack IPv4/IPv6 server. Assume the client and server support both TCP and SCTP.

The Policy Manager puts together a list of candidate transport solutions. As an example, assume that the list contains TCP and SCTP over IPv6 (with equal priority) followed by TCP and SCTP over IPv4 (with equal priority), but the latter have a lower priority than their IPv6 counterparts. The NEAT Logic calls Happy Eyeballs, which traverses the list, and spawns a separate thread for each candidate transport solution. In our example, this means that two threads are created for TCP and SCTP over IPv4, and two threads for TCP and SCTP over IPv6. Since the IPv4 transport solutions have a lower priority than the IPv4 ones, the creation of the threads for the IPv4 transport solutions are delayed with respect to the IPv6 transport solution threads. The length of the delay depends on the size of the difference in priority. Within each thread, a connection attempt is made, and if the connection attempt is successful, the thread returns a connection handle. Otherwise, it signals a failure by returning an invalid connection handle. In both cases the outcome (connection attempt success or failure) is cached by the Policy Manager through the Policy Interface. Assume that all connection attempts succeed and therefore will be cached as successful connection attempts. In this case, since the IPv6 connection attempts were started earlier than IPv4 counterparts, one of these attempts will be selected by Happy Eyeballs.

Provided Transport Service Feature(s):

- NEAT selected transport protocol.

Related building blocks:

- NEAT Logic (§ 2.1.3).
- NEAT Policy Manager (via Policy Interface) (§ 2.4.1).
- Connect to a name (§ 2.1.4).

2.4 Policy Components

The NEAT Policy components allow the definition of policies which trigger the activation of additional constraints for a given connection based on the properties requested by applications as well as known

network characteristics. As a result applications will be able to, e.g., utilise different protocols depending on the chosen network interface.

The Policy components are comprised of the following building blocks described next:

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

For each new NEAT Flow 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 is 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 § 2.4.3). The PIB is 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.

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 as well as the 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.

2.4.1 NEAT Policy Manager

The NEAT Policy Manager compiles all stored policies into a single set of valid rules. Conflicting policies are resolved by prioritising their 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 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 (§ 2.1.5), together with statistics from CIB, to enable useful debugging.

The PM requires three types of inputs:

- *Application properties:* a list of `{key, value}` pairs describing the properties that a NEAT-enabled application desires for a newly opened NEAT Flow. The PM uses the approach described

in this section (see **Examples** below) to extract the most suitable candidates for the property requirements. Application properties are 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. Each policy is stored as a JSON file [6] with a `.policy` file extension (the policy file format is described in § 2.4.2).
- *CIB sources (characteristics):* they provide information about transport and interface characteristics in a predefined location in the file system. Each CIB source is responsible for generating and maintaining a CIB entry with a `.cib` extension in `/var/neat/cib/`. The CIB compiles this information into an internal parseable data structure (e.g., a database or graph).

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. The candidates should at least include a local interface and/or transport protocol parameters. 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.

Policy Interface (PI): The Policy Interface exposes a set of programming function calls that NEAT building blocks have to invoke to make requests to the NEAT Policy Manager. It is an internal interface enabling the communication between the Policy Manager and other NEAT components. It might also be exposed to external modules in future versions. The PI accepts JSON data as function arguments in order to achieve a decoupling from the rest of the NEAT System. In fact, the policy components described in this section may become optional or may even be run outside the host system using them in a future version of the NEAT System, e.g., for less powerful mobile devices. In the first implementation of the PI, only one call is provided to request the list of candidates after a user or an application asks for a Transport Service. As an example, assuming that a user/application makes a request to the NEAT System for a transport connection to the IP address `a.b.c.d` with `latency<150 ms`, the NEAT Logic passes the PI a JSON input like this:

```
1 {
2   // metadata like user ID, application ID, ...
3   "remote_address": "a.b.c.d"
4   "latency_lt":"150" //latency less than
5 }
```

Listing 11: Example of JSON data passed to the Policy Interface.

Some examples of the operation: An example of the proposed workflow of the Policy Manager 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 (e.g., `capacity>100 Mbps, bulk_download=True, delay_sensitive=False`) and the NEAT Logic itself (e.g., DNS lookup result).

2. The PM performs a CIB lookup and receives an initial set of transport option candidates which (mostly) 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 § 2.4.3 for more details.

3. 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 (preferred) properties. Further it may append mandatory (required) properties (e.g., "do not use 3G for bulk data"). See § 2.4.2 for more details.

4. PM prunes all candidates which do not satisfy the required properties. Next, the PM ranks the candidates based on the number of satisfied properties.

5. Finally the ranked candidate list is returned to the NEAT Logic. For each candidate the PM may also return a list specifying which properties were satisfied. This information may be used by the NEAT Logic to adjust the final 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 (§ 2.4.3).
- PIB (§ 2.4.2).

2.4.2 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.

NEAT policies are based on the following logic: `MATCH <set of provided properties>` → `OPTIONAL /REQUIRED <set of new properties to be 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 [6] and typically contains the following attributes:

- `name`: name of the policy.

- `description`: human-readable description of the policy.
- `priority`: integer for priority level of the policy. Higher number means higher priority.
- `match`: dictionary of properties matched against each CIB candidate in order to trigger the policy. A policy is activated only if all of these properties are matched. Each attribute specifies the key to lookup in CIB candidates and match against the provided value. A value can also be `ANY` or empty, meaning that only its presence is important, not its value.
- `optional`: dictionary of properties that the policy adds to the final candidate as optional. If a candidate already matches one or more of the optional properties from a policy, its rank within the final candidate list increases.
- `required`: dictionary of mandatory properties that the policy wants to enforce. If the candidate cannot satisfy any of these it is 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 name section is mandatory while the remaining sections can be omitted if empty. A policy may optionally include other attributes that will be identified in following versions. The JSON format keeps the sufficient level of freedom to extend and modify policy definition. Inside the PM each policy has a unique identifier to avoid conflicts with policies/applications sharing the same name.

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, Windows, Android, ...). 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.

In the first 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 read by the PIB and compiled into a PIB repository which is not directly exposed outside the NEAT Policy components. Read/write access to the PIB/CIB information by different entities is controlled using the existing OS user/access mechanisms. During operation the PM may cache the CIB/PIB information to optimise the performance. Optionally the internal state of the PM may be exposed through an entry in `procfs` when available in Unix-like OSes (or a similar tool in other OSes), or appended to the PM output JSON structure.

Some examples of the operation: A sample of the proposed policy file format is given below:

```
{
  "name":"policy A",
  "description":"bulk file transfer",
  "priority":"0",
  "match":{
    "is_wired_interface":true,
    "interface_speed_ge":1000
  },
  "optional":{
    "TCP_CC":"LBE"
  },
  "required":{
    "MTU":"9600"
  }
}
```

Listing 12: Policy file example.

Policies can be divided in three main types, based on the presence of certain attributes in their definition:

- *Enforcing policy*: a policy without the `match` attribute but containing `optional` and/or `required` attributes. It enforces some properties without checking any condition, e.g., “always use TCP”.
- *Filtering policy*: a policy without the `optional` and `required` attributes but containing the `match` attribute. It selects the best combination that satisfies the highest number of given conditions without adding any new properties, e.g. “prefer wired interfaces with MTU size greater than X”.
- *Full policy*: a policy containing the `match` attributes together with the `optional` and/or `required` attributes. It applies new properties to filtered candidates, e.g., “if an interface is wireless, use congestion control mechanism Y”.

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

Related building blocks:

- NEAT Policy Manager (§ 2.4.1).

2.4.3 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. A *CIB source* is defined as any module which provides an input for the CIB in the correct format accepted by the CIB.

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 will be provided by NEAT building blocks such as Happy Eyeballs (§ 2.3.1) which will store discovered transport protocols and parameters supported along paths in the CIB for future reuse. External CIB sources may be provided 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 information stored in the CIB whenever a new NEAT Flow is initiated.

CIB architecture: The CIB is compiled from multiple inputs generated by CIB sources, like various components of the NEAT System as well as external modules. In the first CIB implementation CIB entries are simple files stored in a predefined folder, which location depends on the OS used. Therefore, from the CIB perspective a CIB source equates to a file in that folder. Each CIB source generates a JSON file containing `{key, value}` pairs describing a set of attributes for a given resource. In the first 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 mechanism.

Three types of CIB sources have been initially defined:

- `local`: sources describing a local endpoint of a connection (e.g., interface information collected from the OS).
- `remote`: sources describing remote endpoints of a connection (e.g., destination host with IP and port).
- `connection`: attributes of an established connection such as transport protocol and associated options. This type of CIB may also be used to store information about paths, collected using active measurements or an external controller.

Each CIB entry contains a unique index (`idx`) which may be referenced by any another CIB entry. The following example illustrates the CIB entries associated (through their indices) with three NEAT Flows originating from the same local interface:

```
[local_X] <-> [connection_Y] <-> [remote_Z]
[local_X] <-> [connection_U] <-> [remote_Z]
[local_X] <-> [connection_V] <-> [remote_W]
```

The CIB parses all CIB entries to generate an internal list containing all active and archived connections, containing an arbitrary number of `{key, value}` pairs describing the characteristics of the

connections and the reference to all other related CIB entries. The CIB exposes an interface which the PM may use to obtain a list of connections or interfaces matching a set of properties requested by an application.

CIB lookup workflow: Local entries are continuously maintained—e.g., by a module monitoring the local OS capabilities. Whenever a new NEAT Flow is established, the responsible NEAT module creates a corresponding remote entry as well as a connection entry containing the negotiated attributes. These entries expire after a fixed time after 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.

We assume that a CIB lookup request contains a list of `{key, value}` pairs including the address of the NEAT destination. When the PM performs a lookup the CIB initially checks whether a matching remote CIB entry already exists in the repository. If this is the case the CIB returns the properties of the associated connection and the corresponding local CIB entry. If the remote destination is new the CIB may return a set of inferred properties supported by the local interfaces by evaluating previously established connections to other remote destinations.

The PM receives a ranked list of query results which best meet the application requirements. The list is evaluated by the PM to apply NEAT policies and generate the candidate list which will be returned to the NEAT Logic.

Some examples of the operation: Example of NEAT-provided CIB sources are:

- 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 characteristics such as metadata (e.g., signal strength, type), passive and active measurements.

The following listings show sample CIB entries and their relationship:

- A local interface named `en3`, whose characteristics are generated by `ethtool` from the OS (Listing 13).
- A remote interface known by a previous NEAT Flow, entry initiated by the Happy Eyeball component (Listing 14).
- An active or archived NEAT Flow involving the two endpoints (Listing 15).

```
1 {
2   "cib_source":"ethtool", // who generated the CIB file
3   "type":"local", // CIB source type
4   "idx":"319", // internal identifier
5   "interface":"en3", // properties of the resource
6   "is_wired_interface":"true",
```

```
  "MTU":"9600",
  "local_address":"10.10.3.1"
}
```

Listing 13: Local interface.

```
{
  "cib_source":"happy_eyeballs",
  "type":"remote",
  "idx":"234", // internal identifier
  "neat_flow_idx":["111FA"], // reference to the CIB identifier for the related NEAT
      Flow
  "address":"23:::23:12",
  "port":"8081"
}
```

Listing 14: Remote endpoint.

```
{
  "cib_source":"happy_eyeballs",
  "type":"neat_flow",
  "idx":"111FA", // internal identifier
  "timestamp":"13452350",
  "local_idx":"319", // reference to the local CIB source identifier
  "remote_idx":"234", // reference to the remote CIB source identifier
  "protocol":"TCP",
  "TCP_CC":"cubic" // TCP congestion control
}
```

Listing 15: Active or previous NEAT Flow.

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

Related building blocks:

- NEAT Policy Manager (§ 2.4.1).

3 Conclusions

This document has presented a first version of the low-level NEAT Core Transport System; the basic components necessary to provide NEAT Transport Services in the API described in Deliverable D1.2 [14]. This was developed as the first stage of the design and development efforts undertaken in Task 2.1. The building blocks ensure basic functionalities provided by four NEAT component groupings: NEAT Framework, NEAT Transport, NEAT Selection and NEAT Policy.

In Section 1 we discussed the notion of low-level and high-level functionalities provided by NEAT and provided an overview of the NEAT architecture and Transport Services provided by the API. We also introduced low-level components required to provide these Transport Services. In Section 2 we

discussed each of these low-level components in more detail and identified their internal dependencies, using examples of use or operation and C code-snippets.

A first implementation of some key building blocks discussed in this document has been finalised, while other building blocks continue to be developed by the consortium. The final outcome of these efforts will be reported in Deliverable D2.2.

Deliverable D2.2 will also report on the implementation of the high-level components developed in Task 2.2, which instantiate other functions beneath the API presented in Deliverable D1.2 [14]. D2.2 will describe how these high-level functions relate to and use the low-level components presented in this deliverable.

Key work in Work Package 2 during the second year of the project will explore:

- A set of Happy Eyeballs mechanisms and the resource requirements these place on the CPU and memory at the server side to confirm or modify our choice of selection algorithm.
- Per-message local drop precedence within SCTP streams.
- Local flow scheduling priority among a set of NEAT Flows, allowing prioritisation among different NEAT Flows that may share the same bottleneck or chosen by the NEAT Policy Manager.

At the conclusion of Tasks 2.1 and 2.2 NEAT will report on the evolution of the complete core transport system, describing any developments that extend the functions described in this deliverable.

References

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

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

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

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

[5] 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. [Online]. Available: http://www.ietf.org/rfc/rfc5246.txt

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

[7] G. Fairhurst, B. Trammell, and M. Kuehlewind, "Services provided by IETF transport protocols and congestion control mechanisms," Internet Draft draft-ietf-taps-transports, Dec. 2015, work in progress. [Online]. Available: https://tools.ietf.org/html/draft-ietf-taps-transports-08.txt

[8] 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/

[9] R. Gilligan, S. Thomson, J. Bound, J. McCann, and W. Stevens, "Basic Socket Interface Extensions for IPv6," RFC 3493 (Informational), Internet Engineering Task Force, Feb. 2003. [Online]. Available: http://www.ietf.org/rfc/rfc3493.txt

[10] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones, "SOCKS Protocol Version 5," RFC 1928 (Proposed Standard), Internet Engineering Task Force, Mar. 1996. [Online]. Available: http://www.ietf.org/rfc/rfc1928.txt

[11] R. Mahy, P. Matthews, and J. Rosenberg, "Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)," RFC 5766 (Proposed Standard), Internet Engineering Task Force, Apr. 2010. [Online]. Available: http://www.ietf.org/rfc/rfc5766.txt

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

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

[14] 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/

[15] 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 section 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.

Disclaimer