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

H2020-ICT-05-2014
Project number: 644334

Deliverable D1.1
NEAT Architecture

Editor(s): Gorry Fairhurst, Tom Jones
Contributor(s): Zdravko Bozakov, Anna Brunstrom, Dragana Damjanovic, Toerless Eckert, Kristian Riktor Evensen, Karl-Johan Grinnemo, Audun Fosselie Hansen, Naeem Khademi, Simone Mangiante, Patrick McManus, Giorgos Papastergiou, David Ros, Michael Tüxen, Eric Vyncke, Michael Welzl

Work Package:	1 / Use Cases, System Architecture and APIs
Revision:	1.0
Date:	December 1, 2015
Deliverable type:	R (Report)
Dissemination level:	Public

Abstract

Ossification of the Internet transport-layer architecture is a significant barrier to innovation of the Internet. Such innovation is desirable for many reasons. Current applications often need to implement their own mechanisms to receive the transport service they need, but many do not have the breadth of adapting to all possible network characteristics. An updated transport architecture can do much to make the Internet more flexible and extensible. New ground-breaking services often require different or updated transport protocols, could benefit from better signalling between application and network, or desire a more flexible choice of which network path is used for which traffic.

This document therefore proposes a new transport architecture. Such architecture lowers the barrier to service innovation by proposing a "transport system", the *NEAT System*, that can leverage the rich set of available transport protocols. It paves the way for an architectural change of the Internet where new transport-layer services can seamlessly be integrated and quickly made available, minimising deployment difficulties, and allowing Internet innovators to take advantage of them wherever possible.

The document provides a survey of the state-of-the-art to identify the architectural obstacles to, and opportunities for, evolution of the transport layer. It also details a set of general requirements for a new transport architecture.

This new architecture is motivated by a set of use-cases, followed by a description of the NEAT architecture for a transport system, designed to permit applications to select appropriate transports based on their needs and the available transport services.

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

API Application Programming Interface

CIB Characteristics Information Base

DCCP Datagram Congestion Control Protocol

DSCP Differentiated Services Code Point

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

IRTF Internet Research Task Force

IF Interface

IP Internet Protocol

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

PMTU Path MTU

QoS Quality of Service

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
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TURN	Traversal Using Relays around NAT
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
VPN	Virtual Private Network
WAN	Wide Area Network

1 Introduction

The NEAT System seeks to change the transport layer interface that is exposed to Internet applications such that, instead of specifying a protocol, a service is specified. This seemingly simple change can have massive ramifications, because it will allow flexible usage of a range of technology underneath the new interface. To yield immediate benefits to applications, this project proposes a new architecture that tries to make the best possible use of the protocols/services that are available end-to-end along a given network path.

This document lays out the terminology used in the NEAT System, introduces the requirements for a new transport architecture and presents the NEAT Architecture.

Section 2 surveys the current state-of-the-art approaches to dealing with ossification of the Internet transport layer.

Section 3 sets out a use case from each of the corporate partners in the NEAT project. Each use case defines features that might be desirable in a new transport system and the set of features that would be required to implement the entire use case.

Section 4 identifies a set of general requirements for the new NEAT transport system, broader than those stemming from the corporate partners' use cases.

The NEAT Architecture is presented for the first time in Section 5. The application cases that the NEAT System will address are described here and the principles for design of the entire system are laid out. The NEAT Architecture is described in terms of the five main component groups: NEAT Framework, NEAT Selection, NEAT Policy, NEAT Transport and NEAT Signalling and Handover.

Section 6 furthers the description of the NEAT Architecture by walking through four example applications using the entire NEAT stack and interacting with the NEAT System.

Section 7 describes the testbeds available for evaluating the NEAT System. Candidate applications to be evaluated are discussed here and the potential for standardisation is highlighted.

Section 8 draws conclusions from the document and maps the work that will happen in other work packages in the NEAT Project, with the architecture design presented in this document as a basis.

The remainder of Section 1 describes the current Internet transport architecture and its failings. This lays out how the NEAT project intends to address these issues in a new NEAT Architecture. Here we also define some of the core NEAT Terminology required to understand the NEAT System.

1.1 Traditional transport-layer architecture, as embodied by the Socket API

The Socket API is one of the most pervading and long-lasting interfaces in distributed computing. It was developed by the Computer Systems Research Group at the University of California and was first released as part of the 4.2BSD version of UNIX in 1983 [38]. The Berkeley Socket API has since been standardised in a joint Open Group/IEEE/ISO standard for Portable Operating Systems Interface (POSIX) [31], and most modern operating systems are now POSIX compliant. This Socket API has been around for more than 30 years, and, as remarkable as it seems, has remained largely unchanged. The major changes have been addition of support for IPv6 [23, 45, 67], and the multi-homing capabilities of SCTP [69].

The Socket API constitutes the programming interface through which most applications communicate with the network stack. Conceptually, a socket is an abstraction of a communication endpoint through which an application may send and receive data in much the same way as to an ordinary file. On a Unix system, the distinction between a file and a socket is even further blurred, and socket

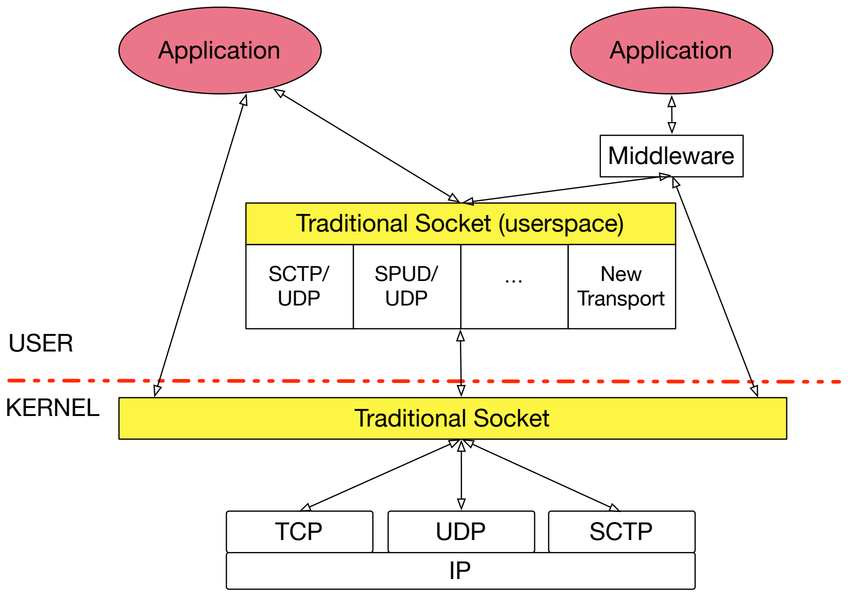

Figure 1: The Traditional Socket Interface, showing both kernel and user-space transport protocols.

descriptors are actually implemented as file descriptors.

The Socket API was from the start designed to be independent from the underlying protocol stack. This is visible in the way that a socket is created:

```
int socket(int domain, int type, int protocol).
```

The **domain** parameter determines the communication domain or protocol family of a socket. Examples of protocol families include: `AF_INET` for the IPv4 Internet domain; `AF_INET6` for the IPv6 Internet domain; and `AF_UNIX` for the local or Unix domain.

The **type** parameter determines the type of a socket, or, more specifically, the semantics for the *Transport Service*—e.g., whether the Transport Service should be connection-oriented, reliable and stream-oriented (`SOCK_STREAM`), or connectionless, unreliable and message-oriented (`SOCK_DGRAM`). Finally, the **protocol** parameter lets an application specify which transport protocol to use to provide the Transport Service specified by the **type** parameter.

Figure 1 shows the layers of the Socket API and the paths that code can call against it. This shows current applications that can either directly use the Socket API (shown in yellow) or can employ middleware that indirectly uses the Socket API on their behalf. The abstraction provided by the Socket API makes it possible to substitute different implementations of a transport protocol without applications (or middleware) needing to be aware of the change.

The figure shows the two most widely used protocols, TCP and UDP, both of which are shown implemented in the kernel of the Operating System (OS), together with the Stream Control Transmission Protocol (SCTP). SCTP can also be implemented in user-space, layered over UDP (SCTP/UDP), as may other transports, such as the experimental Session Protocol for User Datagrams (SPUD). When using a transport, an application does not need to know whether they are using a native (kernel) or user-space protocol.

Although the Socket API comprises a fairly large number of functions, there are less than a dozen core functions. For example, a simple connection-oriented, client-server application using TCP does not need to use more than the eight functions listed in Table 1. A server application generally executes

Table 1: Basic TCP Socket API functions.

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

the first four functions in the order given in the table, while a client application, after having created a socket, attempts to connect to the server; the `send()` and `recv` function calls may be called by both the client and the server, and both the client and the server close a connection.

An application can use options to further tune the properties of a socket. Options may affect a socket's general properties, or the behaviour of a specific protocol supported via the Socket API. Since being originally standardised, many extensions to the Socket API have been proposed, e.g., support for IPv6, multihoming, multipath, and multicast. However, the available set of options is platform-dependent.

There are essentially three ways to manipulate socket options: the functions `setsockopt()` and `getsockopt()` which allow a program access to the majority of the available socket options; the function `fcntl()`, which is primarily used with non-blocking and asynchronous I/O; and the function `ioctl()` that has traditionally been the way to access implementation-dependent socket attributes.

1.2 The NEAT approach

An important deficiency of the current Socket API is that transport (and network) decisions are often made either by an application at design time or deep within the network stack running on a host, where by default, only little information about the application's needs is available. For instance, the network stack has no idea of the size or communications style of a planned download, nor its transfer requirements. However, at least some of this information is often known by an application. An application though, has no understanding of the properties of the network capabilities (the set of available interfaces and what rates they support nor knowledge of which transports are available/usable), or understanding of the characteristics of a particular network path that could be used. This leads to application developers making decisions at design time—decisions which could have been best made at run-time using information that is available within the network stack.

The Internet was originally designed to be flexible, with transport functions operating end-to-end, and intermediate network devices only interacting with the transport in a way that supports end-to-end functions [62]. However, over time this model has evolved, with increased deployment of network devices that monitor, modify and sometimes replace transport protocols. Driven by business needs, these middleboxes (firewalls, NATs, load balancers, and a range of other devices) have become common on most Internet paths. However, since a middlebox interacts with the transport protocol, it implies that the middlebox must be able to interpret the transport protocol, and in many cases implement some part of the protocol. To deploy a new protocol (or protocol extension) it would often be required to change the middlebox configuration or design—and so the Internet has become rigid or inflexible, in a process often referred to as *ossification*.

Some applications have incorporated significant amounts of code to try to overcome the problems of ossification (e.g., the way web browsers use "happy eyeballs" techniques to select whether to use IPv4 or IPv6 [82], and Interactive Connectivity Establishment (ICE) [58]), but current methods are not sufficiently general to prevent continued ossification of the network stack.

To address these shortcomings, the NEAT project proposes a new transport architecture. This architecture seeks to replace the Socket API with a new API that will enable any application to express its communication preferences. This information can then be used by proactive policies to choose the appropriate transport, IP protocol version, to select an interface, or to tune network parameters. The API also allows network conditions and choices to be reported to the application.

Applications that use the new API can provide information about the requirements for a desired transport service and determine the properties of the offered transport service. It is this additional information that enables the NEAT System to move beyond the constraints of the traditional Socket API, since the stack then becomes aware of what is actually desired or required for each traffic flow. The additional information can be used to automatically identify which transport components (protocol and other transport mechanisms) could be used to realise the required transport service. This can drive the selection by the NEAT System of the best components to use and determine how these need to be configured. In making decisions, the NEAT System can utilise policy information provided at configuration, previously discovered path characteristics and probing techniques—a transport component will only be used when there is evidence of actual support by both the remote endpoint and the path used to reach it.

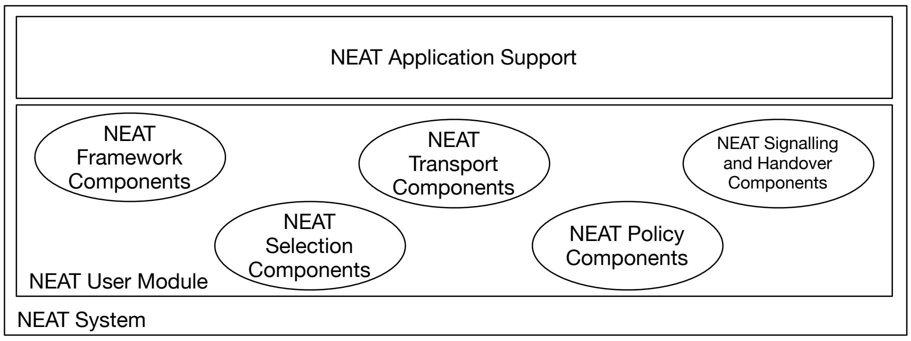
NEAT Application Support
NEAT Framework Components
NEAT Transport Components
NEAT Signalling and Handover Components
NEAT Selection Components
NEAT Policy Components
NEAT User Module
NEAT System

Figure 2: NEAT System, showing the relation between the different major blocks that compose the system.

1.3 NEAT terminology

This section defines terminology used to describe NEAT. These terms are introduced here, and used throughout the remainder of this document. Figure 2 illustrates the relationship between some of these terms.

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

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 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, as depicted in Figure 2: 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 The interface to allow querying of the NEAT Policy Manager.

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. Examples of end-to-end features include confidentiality, reliable delivery, ordered delivery, message versus stream orientation, etc. The desire to use a specific feature is indicated through the NEAT User API.

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.

2 State of the Art

The ossification problem of the Internet transport layer needs to be tackled to re-enable evolution within the transport layer. This is a multi-dimensional problem that requires the enhancement of multiple components of the end-to-end communication chain.

Several point solutions have been proposed in the literature, each aiming to address a specific facet of the overall problem. Yet, no single solution has emerged that is able to alone enable this evolution. This is what the NEAT System seeks to offer: a cohesive system that offers a comprehensive approach to fighting transport ossification.

This section reviews the current state of the art to identify existing work that could be leveraged by a NEAT System. The review is divided into five parts, each covering a different aspect of the problem space: 1) overcoming barriers imposed by middleboxes, 2) enhancing the API between the applications and the transport layer, 3) discovering and exploiting end-to-end capabilities, 4) interacting with the network for improved application experience, and 5) enabling user-space protocol stacks.

2.1 Overcoming barriers imposed by middleboxes

The most significant barrier that needs to be overcome for the deployment of a new transport architecture is the ossification of the network infrastructure caused by the introduction of a variety of middleboxes (from NATs to firewalls, accelerators, load balancers, and a range of portals and more exotic devices). Any new "native" transport (i.e., layered directly on IP) is doomed to fail to pass through most middleboxes until specific explicit support is added for that transport, and even new extensions to old transports (i.e., to TCP and UDP) are also vulnerable to potential middlebox interference. However, the case to support a protocol or a protocol extension that has not reached wide-scale use is almost non-existent. This mutual dependency is what causes the ossification.

Recent efforts to provide a richer set of transport services than those provided by TCP and UDP, within the constrained design space imposed by the ubiquitous deployment of middleboxes, span two research directions: a) extending TCP while guarding new extensions against potential middlebox interference, and b) building new application-specific transports atop UDP or TCP to ensure they transparently pass through existing middleboxes.

2.1.1 Extending TCP to offer a richer set of transport services

Although TCP was designed to be extensible, the ubiquitous deployment of middleboxes that interfere with TCP traffic (e.g., by stripping unknown TCP options, interpreting the semantics of TCP header fields, or modifying TCP header fields) has significantly restricted its extensibility. Several measurement studies have been conducted to investigate how existing middleboxes interact, either intentionally or unintentionally, with TCP extensions, how prevalent these interactions are, and to what extent they affect TCP's extensibility [11, 15, 30, 37, 41]. These empirical studies provide a first demarcation of the solution space and some first guidelines for designing middlebox-proof TCP extensions [30].

Multipath TCP (MPTCP) [21, 30, 51], Tcpcrypt [5, 6], and Gentle Aggression [19] are prominent examples of recent TCP extensions whose design was highly influenced by the need to account for known middlebox behavior (e.g., re-writing of sequence numbers). Several techniques (e.g., use of relative sequence numbers) are adopted to guard them against potential middlebox interference, while a fallback strategy to plain TCP is incorporated to handle cases where extended features fail. The latter is considered an important design goal to achieve widespread deployment.

Extending the TCP option space to increase the number and the extent of TCP extensions that can be simultaneously used by a TCP connection has also become an active research area that faces similar middlebox-related issues. Recent approaches that aim to tackle this problem are TCP Extended Data Offset (EDO) [73, 76], TCP SYN Extended Option Space (SYN-EOS) [74], and Inner Space [8]. However, these approaches are currently under development and further work is needed to evaluate their deployability.

Recent experience in the design of MPTCP inspired another possible dimension to the design space: TCP "camouflaging" [42]. It may be possible to deploy a new transport protocol to operate alongside TCP when it is disguised to look like TCP on the wire. In this context, the authors of [42] introduce the concept of Polyversal TCP (PVTCP). It remains to be seen whether the complexity of Polyversal TCP, or similar approaches, offers a feasible path to deployment.

The IAB Workshop on Stack Evolution in a Middlebox Internet [75], held in January 2015, identified a pressing need to understand deployed middleboxes and resulted in a new IRTF Proposed Research Group, Measurement and Analysis for Protocols[1] (MAP), that aims to address such measurement-related issues [33].

2.1.2 Using widely deployed transports as substrates

In today's Internet, UDP and TCP are more and more regarded as substrates over which new transport protocols can be built and quickly deployed. Typically, such transports are meant to be implemented as user-space libraries integrated into applications. They often have limited scope, and do not provide separable building blocks that support applications with different needs.

Characteristic examples of application-level transports are:

[1] Formerly called "How Ossified is the Protocol Stack" (HOPS).

- Google's Quick UDP Internet Connections (QUIC) protocol [12, 61], a UDP-based low-latency alternative to TCP/TLS for SPDY [25] and HTTP/2 [2].
- The WebSocket protocol [78] that facilitates message-oriented, bi-directional web communications over TCP.
- Adobe's Secure Real Time Media Flow Protocol (RTMFP) [72] that enables efficient peer-to-peer multimedia streaming over UDP.
- The widely used TLS [16] and DTLS [53] protocols that provide stream- and datagram-oriented security services over TCP and UDP, respectively.
- The uTorrent Transport Protocol (uTP) [44], a UDP-based alternative to the BitTorrent protocol designed to offer a less-than-best-effort service for peer-to-peer file sharing applications.

UDP encapsulation is also commonly used to enable middlebox traversal of native transports, and such methods have been standardised for SCTP [77] and DCCP [50]. A method for encapsulating TCP over UDP for coping with cases where only UDP is supported has been also proposed in [10]. Encapsulation can offer an additional benefit: it allows user-space implementations of native transports (e.g., *libusrsctp* implementation [48]) to be part of applications without requiring special privileges to access the IP layer. However, it can also increase capacity usage and processing overhead, and an encapsulated protocol cannot in principle interoperate with the native protocol. In addition, UDP-based encapsulations are by default classified by network boxes as UDP traffic rather than relating to a specific protocol, and hence may be subject to policies such as rate-limiting. More general encapsulation solutions, such as Generic UDP Tunnelling (GUT) [39] and Generic UDP Encapsulation (GUE) [27] have been proposed with the aim of enabling more consistent and transparent deployment. These solutions are, however, not entirely generic and protocol-specific adaptions may still be needed.

The authors of [40] approach the problem from a slightly different perspective and suggest, at a conceptual level, the reinterpretation of the semantics of TCP and UDP to support novel services. They propose the reinterpretation of UDP headers as transport identification headers and the relaxation of TCP semantics that are not visible on the wire (e.g., reliability, ordering, and flow control). Minion [34, 46], a recent effort that aims to, among others, relax the strict ordering constraints of TCP through kernel modifications and API enhancements, could contribute to this development.

A transport layer architecture that supports such innovations inside UDP and TCP can allow for faster and seamless deployment of new transport services, and of functions that do not rely on any special support from network devices.

2.2 Enhancing the API between the applications and the transport layer

The very success of TCP and UDP has led to the ossification of the transport API presented to the application, since these are now the only transports widely available. This is mainly reflected in the implementation of the widely used Socket API, which ties applications to a priori choices of transport protocol (either TCP or UDP), and hence forces applications to be coded to work with a specific transport. The inception of SCTP [68] was one of the reasons that the Socket API has undergone changes in more recent years to support new services such as multihoming and multistreaming. Still, the Socket API is very limited in terms of presenting services, instead of transport protocols, to applications.

To satisfy the needs of applications that require advanced services a large variety of middleware has emerged. This provides application developers with high-level APIs, often built on the Socket API and designed to tackle a specific problem. Middleware exists at different levels of abstraction. Examples of middleware can be very diverse, e.g., they include: ZeroMQ [32], a message-oriented middleware that adds support for advanced communication patterns (e.g., publish/subscribe) and enables applications to have control over policies and QoS features; Web Real-Time Communication (WebRTC) [3] that provides a complete framework for application developers to implement real-time audio/video communication and peer-to-peer file sharing between browser-based applications; and Common Object Request Broker (CORBA) [47], which enables object exchange between systems regardless of their deployment location and implementation language. Although middleware can be used to mitigate some ossification problems in the current Internet, this approach has limited scope and hence cannot completely solve ossification. For instance, although ZeroMQ has broad support for transport protocols, connection-oriented services are largely focussed on TCP.

A different approach to overcome ossification is to extend the Socket API. Over the years, several Socket API extensions have been proposed (Figure 3). Some only aim to remove perceived limitations and drawbacks with the standard Socket API (*basic extensions*), while others focus on ways to let an application express its service requirements to the transport layer (*high-level extensions*). Examples of basic extensions are msocket [14], which makes it possible for an application to explicitly select a protocol stack, and Sockets++ [7], a more elaborate extension introduced to support multimedia applications and which addresses a range of shortcomings, including support for multipoint connections and direct data forwarding to multiple streams.

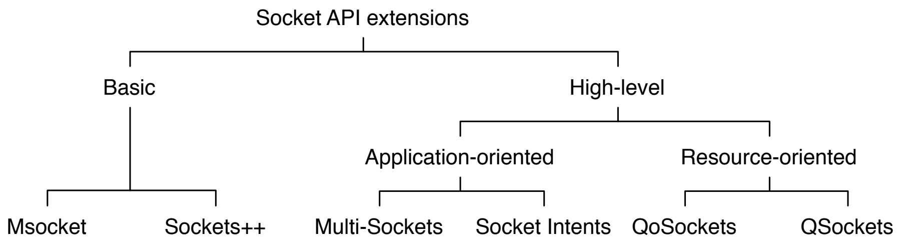

Figure 3: Extensions to the Socket API.

High-level extensions can be further divided into *application-oriented* and *resource-oriented* extensions. Application-oriented extensions are extensions that let an application express its QoS requirements in terms of application-dependent performance metrics, or that let an application express the characteristics of the traffic it will generate. Examples are multi-sockets [28] and its successor Socket Intents [64]. Socket Intents allows applications to supply qualitative information about the traffic they intend to send, enabling a policy-based selection of the most appropriate network interface and tuning of socket parameters. Resource-oriented extensions focus on system-wide, network-oriented performance metrics such as packet loss, reordering, bit rate, or end-to-end delay. Extensions that fall into this category are: QoSockets [24], which enables an application to negotiate QoS requirements with the transport layer, and the latter to signal violations of these requirements back to the application; and QSockets [1], which enables QoS-aware scheduling based on the application's QoS preferences and allows applications to obtain detailed QoS feedback from the transport layer.

A number of works have identified the need for a service-oriented API that can enable transparent protocol selection and hence loosen the coupling between applications and transports. Three steps towards resolving this problem are identified in [26] and further concretised in [54] based on a prototype implementation: a protocol-independent mechanism for describing a communication service by a combination of inherent properties (like reliability, security etc.) and qualitative properties (like transmission delay, flow setup delay, etc.); a service-oriented interface between applications and the transport layer where the applications describe the services they require; a broker performing dynamic protocol selection and optimisation at run-time, matching transport service offerings with application requirements. Similar concepts are also found in the “Adaptation Layer" proposed in [79] and the requirement-based socket API proposed in [65]. A strawman proposal for a service oriented transport API is also provided in [80]. This offers an adapted version of the Socket API, where a selected list of transport services can be accessed in a protocol-independent way through their service number.

A service-oriented API is key in enabling transport layer evolution, since it will allow for new transport services and functions to be added incrementally and transparently to the applications. A fundamental challenge for any new service-oriented interface to reach significant impact is the need for standardization of the interface and reaching community consensus on the interface features [79]. The recently started IETF work within the Transport Services working group (TAPS) [18] offers a unique opportunity where such proposals for API standardisation can be developed.

2.3 Discovering and exploiting end-to-end capabilities

Realising a flexible, extensible and deployable transport layer framework requires that endpoints know — that is, are able to discover—what protocols and protocol options are supported both along the whole end-to-end path, and by the remote endpoint(s).

Some application-layer proposals provide limited support for negotiating features, e.g., transport security for unicast, connection-oriented application sessions [57], or transport protocol, port and IP address for multimedia sessions [60]. A more generic approach is sketched in [22], which describes a negotiation protocol for end-points to exchange protocol-stack information and agree on a transport stack (i.e., transport and security protocols to be used, and their options) in an efficient manner. Achieving maximum efficiency, in terms of RTTs needed for negotiating, requires changes to the implementations of the transport protocols being negotiated.

In the absence of explicit end-to-end signalling or a negotiation protocol, the only way an end-host has to discover and (implicitly) agree on a particular choice of protocol(s) is to *try* different possibilities, then pick one that works and, generally, abort use of the others. Such “test-and-select” approach is often referred to as *happy eyeballs*, and has been proposed both for choosing between transports [83] and between versions of the IP protocol [82]. To the best of our knowledge only the latter has been implemented in real systems, coupled with address-selection algorithms such as [71] (e.g. [63]). A drawback of this kind of technique is the increased overhead in the number of packets sent, the server-side load, the amount of state created in middleboxes, and added latency for initiating a session; hence, it does not scale well with the number of possibilities to try. A component that caches results and allows for proper tuning of the testing process is considered fundamental for reducing this overhead [63, 83], especially when a wider variety of protocols is available.

While probing and discovery mechanisms are often implemented within the application, placing them below the transport API can allow applications to take advantage of the evolution of the transport stack, and also of shared information about the success/failure of previous attempts by this and other applications to use each known path.

2.4 Interacting with the network for improved application experience

While transports need to be designed to operate robustly over even the simplest of network paths, they can benefit from signalling to and from the network devices on the path. There is a wide range of proposals to support communication between end systems and the network: from protocols to control middlebox behaviour, either explicitly (e.g., PCP [81] and the Middlebox Communication framework (MIDCOM) [66]) or implicitly (e.g., STUN [59] and TURN [49]), to methods for signalling QoS requirements (e.g., differentiated services marking [4] and the Resource ReSerVation Protocol (RSVP) [84]) or flow metadata (e.g., SPUD [36]) from the endpoint to the network, and mechanisms for signalling hints from the network to the endpoints (e.g., Quick-Start [20] and ECN [52]).

However, it has been difficult to deploy applications that can utilise signalling methods at the network layer for various reasons. One issue is the heterogeneity of the Internet. While methods such as PCP [81] are available in some places to communicate port use to local middleboxes, in other places, other methods may be used, and new methods may emerge. An application can not rely on a single method being universally available. Another issue is that some paths have been known to block specific packet headers, or to set/erase particular fields—this has been one of the issues in enabling evolution to support ECN.

Placing optional network signalling below the transport API can allow applications to indicate how they would prefer to use network signalling, without requiring each application to be updated each time a new signalling protocol is introduced. Instead, the stack can determine the appropriate signalling mechanisms to use on a particular path to an endpoint. Freeing applications from choosing and supporting signalling protocols is expected to reduce the barriers to introducing new mechanisms in the network, and allow signalling messages to be exchanged with network devices as new protocols become supported across the network. This will also relieve applications from implementing common complicated functions and allow for more efficient implementations.

2.5 Enabling user-space protocol stacks

One of the major challenges of deploying new transport protocols, as well as evolving the existing ones, is the need for them to operate across multiple OS platforms. Modifying the kernel code is costly in terms of deployment and would often require an update of the OS. It is possible though to run a transport as a user-space library (e.g., user-space implementations of the TCP/IP stack and SCTP protocol already exist [35, 48]). This would enable easy introduction and evolution of new features and protocols that is not dependent on the speed of updating the operating systems.

However, the use of user-space transports presents an important challenge that needs to be addressed: network I/O operations that originate in user-space can incur higher latency compared to network I/O operations handled in the kernel. To address this problem, several solutions that aim to enable fast packet I/O in user-space have been developed. Multistack [29] is probably the most efficient and complete solution available today, and it is able to provide commodity operating systems with support for dedicated user-level network stacks. It can concurrently host a large number of independent stacks and can fall back to the kernel stack if necessary. Other libraries that can help achieve fast packet I/O in Linux user-space are Data Plane Development Kit (DPDK) [13] and PACKET_MMAP [9].

Another technique for running transport protocols in user-space is to run the entire kernel (instead of only the transport) in user-space. User-Mode Linux (UML) [17] runs the kernel as a user-space process and permits experimenting with new transport protocols implemented in different Linux kernels without interfering with the host Linux setup. A similar approach is followed by LibOS [70], which runs the kernel as a library that can be called by an application. LibOS has been used by NUSE [43] to provide a Linux network stack for user-space applications.

Such new techniques can enable portable user-space implementations of new protocol methods, enabling evolution of protocols/mechanisms or as a basis to stimulate initial deployment followed by later implementation in the OS kernel. An architecture that allows either approach offers the flexibility needed to support transport evolution.

3 Use Cases and Their Requirements

This section describes the use case and requirements from each of the commercial partners in NEAT. Each use case description focuses primarily on the aspects that are seen by the partner as specifically distinguishing and important. They therefore do not cover what is considered to be common, or well-understood, underlying requirements—such as, for example, the need of supporting reliable or unreliable delivery. These general requirements are detailed elsewhere in the document.

The use cases were not developed with the intention to define a necessary or sufficient scope for the NEAT architecture, but rather to provide a set of illustrations of the motivation for the work and to set provide the background that will guide the design of the architecture and all subsequent development work. In particular the use-cases have helped develop understanding of the flexibility needed in the system so that it can continue to allow evolution of the transport system offered by NEAT.

3.1 Mozilla Use Case

3.1.1 Description

The Mozilla Firefox web browser runs on almost every conceivable hardware platform, from phones in the 20 € price range to high-end gaming workstations. It also utilises equally diverse access networks, from hardwired fibre to GPRS cellular. However, the goal of the browser does not change across these uses — it presents one unified World Wide Web to everyone.

The primary transport for the web remains HTTP/1 over TCP because it is the least common denominator in all environments. Using a single older transport is not a good match for many new clients, especially the mobile clients being added to the Internet from new geographies.

These mobile clients often are connected over high latency and low bandwidth connections, but even then, can experience tremendous variability over short time periods compared to traditional access mechanisms. TCP has well-documented performance bottlenecks in these scenarios. Additionally, the way HTTP/1 utilises large numbers of short-lived TCP sessions exacerbates performance problems arising from TCP because the often-assumed independence of TCP flows does not hold true.

The development of HTTP/2 [2] has overcome some of the issues associated with HTTP/1. It enables a better pipeline of multiple request-response transactions over a single TCP connection, which eliminates use of multiple short-lived uncoordinated TCP flows. But at the same time, in some environments, TCP remains a significant bottleneck and the use of a single TCP connection introduces an additional issue in the form of head-of-line blocking after packet loss.

Alternative transport functions such as the multi-streaming properties of SCTP, and the new breed of experimental transports tunnelled over UDP (e.g., QUIC) promise to improve performance with some of these bottlenecks. Unfortunately, an environment as diverse as the World Wide Web will never be able to have a "flag day" where such new transports can be reliably deployed. Instead, we need to deal with a world where some peers are updated and some are not, and some networks allow the traversal of new protocols and some do not, and so on. Experience shows that this transition has progressed slowly, with much of the world still using HTTP and TCP.

User security and privacy are fundamental aspects of the Firefox browser. End-to-end encryption of communication is an important part of it. Where the complete security with integrity, authentication and encryption is not present, at least opportunistic security could offer to users data confidentiality and integrity. Furthermore, TCPINC[2] is an IETF effort that seeks to provide opportunistic data encryption and integrity protection for TCP connections.

The are key issues to be resolved: The lack of transport security, the induced congestion of uncoordinated flows, the frequency of timer-based retry, and the under-utilisation of available bandwidth are high priorities to enable optimal working of Firefox in mobile environments.

Transition to new protocols and mechanisms is always a challenge. Discovery of new capabilities needs to be done dynamically host-by-host, path-by-path, and feature-by-feature to arrive at the best set of capabilities for any combination of client, path, and server that may happen to be used. In contrast to current approaches, this should be done at runtime (rather than when web clients are configured) so that clients are able to determine the presence of key transport features that are only available some of the time and to further enable new better-suited transport protocols to be used as they become supported, such as SCTP or UDP-based transports.

A new transport system is able to define a new interface between the applications (web client) and the transport layer, where applications can specify services instead of being designed to use only specific transport protocols. The NEAT project provides an opportunity to match the service, protocol and capability of end hosts and the network path for Firefox. Examples of features that are expected to improve the browser experience are: multi-streaming, methods to determine IPv6 reachability, detecting availability of support for SCTP, enabling TCP in user-space with shared congestion control, and support for opportunistic security. An important feature of any new system is that it it needs to support evolution of the transport system to enable support for integration of other, as yet unspecified, transport protocols.

3.1.2 Requirements derived from the Mozilla use case

The following requirements have been derived for the Mozilla use case.

- **Network selection:**
  - The system should support the seamless use of IPv6 and IPv4.
- **Transport selection:**
  - The system should support new transports (e.g., SCTP, user-space TCP and as-yet-unspecified transports) allowing these to be used wherever these are deployed and the network path supports them.
  - The system should provide an API that can be used with multi-streaming.
  - The system should support feature discovery and to robustly select a set of transport features that can be used over a given path.

[2] TCP Increased Security working group, https://datatracker.ietf.org/wg/tcpinc/charter/.

  - The system should support one-sided and two-sided deployment.

- **Transport security:**
  - An application should be able to indicate its security requirements to the transport system.
  - The system should offer end-to-end transport security including data encryption and integrity.
  - The system should enable the use of opportunistic security.
  - It should be possible for an application to authenticate the end host with which it is communicating.

- **Flow coordination:**
  - An application should be able to indicate its expectations to the transport system to let the transport system choose appropriate flow coordination mechanisms.
  - It should be possible to coordinate congestion control of an application by having common congestion management.
  - It is desirable to indicate the relative priority, and for the system to use this to prioritise flows.

3.2 Cisco Use Case

3.2.1 Description

Cisco's main business is network infrastructure, hardware, software and control products and the wide range of resulting network design, solutions and managed services in Service Provider, Enterprise and IoT networks. These networks support a wide range of intelligent service differentiations from a variety of QoS mechanisms (intelligent queuing, policing, shaping, classification, ...) to multi-pathing, advanced security (firewalls, support for Virtual Private Networks (VPNs), user-groups, ...), address translation, IPv6 support, transport acceleration/caching and so on. Mutual awareness between the intelligent network and applications is the key requirement to offer the best services to applications and to provide the best methods for network operators to apply desired policies in secure, easily managed and scalable ways—especially in controlled networks where the majority of such advanced services are deployed.

Most network services and policies are intended to be applied on a per-application or transport-layer connection level, therefore, the layers of the applications stack that primarily need to interact with these network services are the IP, transport and session layers.

Cisco is only in the business of developing full application stacks in very specialised market segments such as voice and video over IP, so the majority of applications running across Cisco networks are using third-party network/transport stacks. Therefore Cisco is interested in evolving and proliferating freely available third-party stacks as planned in the NEAT project to enhance integrated applications with the advanced network services.

Cisco sees IPv6 as a key tool to improve services offered by the network because of its architectural improvements over IPv4, and because IPv6 will become the most widely used alternative due to IPv4 address exhaustion and the known limitations of relying on NAT work-arounds. Therefore, Cisco's development of new/advanced network functionalities focuses on IPv6 and the transport stack is the natural counterpart in the endpoints to leverage such improvements.

Support for dual-stack and IPv6 operation are both important. This usually translates into having IPv6-only capable data and control planes. In other words: no IPv4-only functionality, but if desirable/easier then having IPv6-only options. A new transport system needs to support new features introduced by IPv6, such as allowing different address prefixes, to leverage different (source) IPv6 addresses so as to impact the policy used by the transport stack. Information about the semantics of paths/addresses such as the provisioning domain signalling may help support multiple interfaces (e.g., interface selection for wired/wireless to minimise delay).

A key aspect relevant for a new transport system is an ability to utilise the better framework offered by IPv6 for addressing and header options. IPv6 addressing makes it easier to use multiple addresses and to express additional network service semantics through those different addresses, for example different security or quality-of-service options as well as selecting a provider or other type of multihoming to provide alternative path selections (e.g., within a datacentre). The more flexible header options framework in IPv6 may also be used to include additional information.

Home-to-home communication is important for many applications beyond the web. Protocols such as STUN, TURN and ICE have been developed (and are still being improved/expanded upon) to support the creation and selection of the best media shortcuts in the presence of NATs, firewalls and multiple addresses (IPv4/IPv6 as well as multiple addresses within each address family). Cisco thinks that the ability to easily create peer-to-peer transport shortcuts in the face of NAT, firewall, multi-address and multi-protocol support in the NEAT System can also help make this new stack more attractive to other applications beside real-time media. This is valuable for datagram and reliable transport services.

Improved path/endpoint selection information is crucial for efficient establishment of peer-to-peer connections, such as for RTCweb collaboration. This is especially important when each endpoint has multiple addresses due to inside/outside NAT, to optimise total amount of signalling before a peer-to-peer session can be established.

Cisco also has a strong presence in network and middlebox layer security solutions. Wide ranges of those solutions rely today on deep analysis of application data. This approach is expected to undergo significant change with more end-to-end encryption. Likewise, operators still expect that these middleboxes continue to help keep malware out of the operator domain and lock confidential data inside the domain. To that end, intelligent signalling between applications and the network can help to establish trust relationships that either permit to make the right policy decisions without deep packet analysis, or to establish the rights of a middlebox to do inspection without unexpectedly breaking end-to-end trust relationships.

DSCP is the traditional method used to signal QoS to a network and this should be supported, but easier and more flexible options would be valuable. With DSCP it is difficult to figure out how to "just" request low-latency or scavenger service for example. DSCP is also not meant to allow distinguishing flows with the same so-called "per-hop-behaviour", but different policy expectations. Example: different instances of the same application, with one instance being more "important" than the other.

Current designs of transport protocols have focussed on the end-to-end principle of the TCP/IP stack without consideration for signalling between the endpoints and the network (except for the DSCP and ECN fields in the IP header).

A variety of ways have been proposed to improve the network/transport interaction. A change to the interface between the application and the network, can enable development and deployment of new methods for signaling the requirements/expectations of applications to devices in the network. This signalling should allow the network to evolve. The design ought to allow the transport system to be used without requiring special privileges to access the signalling information.

Metadata signalling can be introduced to help network devices select appropriate policies. Traditional middlebox policies have often been based on signature detection-based application classification. This fails in the presence of encryption and standardised transport session layers without static per-application transport port allocation (e.g., most real-time applications).

A variety of methods are being suggested for signalling STUN-DISCUSS, PCP or similar approaches. SPUD is an effort in the IETF to create a (UDP based) layer inside the transport stack to support such signalling and achieve the above goals specifically for security focussed middleboxes and secure applications. Other approaches such as embedding the signalling into pre-existing transport protocols, e.g., SCTP is equally an option.

Expressing typical QoS parameters such as delay, throughput and loss parameters is not sufficient. Applications and networks may have policy requirements not typically covered by existing QoS definitions. Using an application identification, and letting the network (e.g., operator policy) determine what the actual requirements for this application are is the current best way. Allowing applications to explicity indicate specific non-QoS service requirements is more explicitly controlled by the application.

The information provided to the network could evolve to also include non-QoS service parameters (e.g., high value or cheapest-network option), security/privacy (e.g., pass traffic only via “friendly” geographies) or auditability (as required for traffic in regulated application environments such as banking).

Signalling an authenticated application classification provides a potential way to allow middleboxes to overcome constraints (e.g., work in SPUD at the IETF to determine the best minimum set of information for middleboxes to permit such traffic).

A new transport system could evolve to utilise network signalling to improve the design and operation of the transport protocol mechanisms. Traditional TCP congestion control has been based on loss, which results in delay being experienced by an application suffering congestion. Delay-based methods are typically used to provide low-latency congestion control. A network able to separate traffic flows so only delay-sensitive flows compete with each other, could signal this fact so that the applications can confidently do delay-based congestion control without being pushed aside by loss-based traffic. Hints can also provided by the network to assist in making end-to-end congestion control decisions. A transport system ought therefore to allow development of signalling from the network to the application about free capacity, so that the application can increase bandwidth without the fear of congestion loss or higher delay. This may not be per-flow/end-to-end but just for the users’ access network.

Cisco is also in the business of collaboration solutions: application software (e.g., Jabber), dedicated collaboration endpoints for users and integrators (e.g., Telepresence systems), services (e.g., Webex, Spark), and infrastructure components (e.g., conference bridges, H26x/SIP call managers). For these collaboration solutions, critical features expected in a future transport stack include:

- **Priority-Aware Datagram Transport:** media traffic (audio/video) is best built on a transport layer providing a datagram service (e.g., UDP) because the applications desire low latency, but also can benefit from the ability to send bursts of traffic at a high rate—with transmission tightly coupled to the control loop of the media (e.g., a video codec), rather than a preferred target rate specified by the transport stack.

  Enhancements to datagram transports, such as indications of per-packet drop priority to the network, can be valuable to maximise performance. If there is temporary congestion in a router queue carrying a video flow, it could for example preferentially drop video P/B frames but not I frames. Explicit Congestion Notification (ECN) is also potentially valuable to improve real-time congestion control under transient congestion.

- **Media Shortcuts:** media traffic requires high capacity and low delay. One way to enable home-to-home connectivity is to loop the connection through a remote cloud location using an unoptimised client/server application design. However, the best (lowest latency) path is achieved by passing packets along the shortest path through the network, e.g., directly between endpoints at each home.
- **Monitoring:** Because of a need for low-latency on media shortcuts with potentially advanced network interactions (ECN, advanced packet dropping), the ability to monitor actual traffic flows becomes important for operations/diagnostics/SLA-assessment/isolation. Monitoring includes aspects such as tracing paths, and measuring at individual hops along the path quality metrics such as loss, jitter, delay or reordering. It is therefore important to support transport options in which the packet elements to derive such metrics (timestamps, sequence numbers) are encoded in a way that they are accessible to the network (unencrypted or encrypted in a manner that a trusted network can access these attributes). Cisco prefers transport stacks that include parameters to allow hop-by-hop QoS validation in the network (e.g., RTP headers in the clear). This may, for example, use an explicit option chosen by the application (so there is no unexpected exposure of QoS behaviour by applications not desiring to be measured).

As a result of the application and middlebox market spaces in which Cisco operates, Cisco has research and development with many aspects of the solution space as embodied in, for example, Cisco Medianet and Telepresence technologies as well as non-productised experimentation.

3.2.2 Requirements derived from the Cisco use case

The following requirements have been determined for the Cisco use case, and indicate both a need for flexibility in the architecture to enable evolution, and requirements for the transport system including components responsible for signalling.

- **Network requirements:**
  - **A new transport system needs to provide dual-stack support (IPv4 and IPv6) as well as IPv6-only operation**. This needs to be designed to allow exploitation of the unique benefits of IPv6 (e.g., IPv6 prefixes and extension headers).
  - **The transport system be designed to allow direct home-to-home communication** using methods such as STUN, TURN, and ICE. Improved selection information is crucial for efficient establishment of peer-to-peer connections, such as for RTCweb collaboration.
  - **Monitoring/Tracing: The transport system should provide the ability to monitor and trace the performance** over the network path being used.
- **QoS / Low latency:**
  - **A new transport system should allow an application to select transports optimised for low latency** (e.g., support for ECN) using both datagram and (partially) reliable transports.

In reliable transport, further low-latency benefits could be achieved via limited retransmissions or FEC.

  - **The interface to a new transport system should permit applications to indicate a range of QoS-related properties requirements to the transport system.** This should support use of DSCPs, but also enable evolution to easier and more flexible QoS identification.

- **Transport System to network signalling (Flow Metadata):**
  - **Per-message priority:** The transport system should allow per-packet signalling of packet properties to the network (e.g., indications of priority in conjunction with intelligent packet discard by the network, with ECN style or multi-level marking to prefer I frames over P over B media frames).
  - **The transport system should permit applications to signal QoS-related properties to the network.** Methods should be extensible and allow continued evolution of signalling. Such signalling can assist middlebox policy selection and could include advanced methods such as per-flow advisory admission control.
  - **The transport system should permit applications to signal additional non-QoS information,** including identification of the type of application/media transported and explicit non-QoS service requirements that can help future network devices make informed decisions.
  - **Transport System-to-network signalling should not requiring special privileges:** Applications should be allowed to evolve and ought to be able to use the transport system without requiring special privileges.
- **Network to application signalling:**
  - **Capacity hints:** A transport system can utilise signalling from the network to the application (e.g., notifying free capacity) that may be input to congestion control decisions at the transport layer.
  - **Robust signalling:** The transport selection process must solely be based on what works best (e.g., using STUN, PCP, SPUD and other methods).
- **Naming:** The transport API should allow use of session-layer names and avoid use of network-layer addresses. Dynamic naming should be possible (e.g., application assigning names for themselves) and be available across the network (e.g., via DNS/DNS-SD). Naming should be designed to support peer-to-peer applications in an application independent fashion.

3.3 Celerway Use Case

3.3.1 Description

Celerway is an SME that develops software for network devices, such as routers and smartphones, to enable faster and more reliable Internet connectivity. This software provides the following features:

- Active and passive assessment of the performance of available networks.
- Selection of the best network for certain services.
- Selection and manipulation of transport protocols to fit the selected network and optimise throughput.

- Hand over applications to a better network path if the current path does not perform optimally or is disconnected.
- Simultaneous use of multiple networks when available.

The software is provided to consumer **WiFi routers**, transforming them into multi-network routers and enabling the features above. In addition, Celerway is developing **Proxy software** that enables transport splitting. A proxy enables use of proprietary transport options between the router and proxy, and between client devices and the proxy to enhance performance for the features listed above. Celerway is also developing software with these features for user **client devices** (e.g., smartphones). However, in contrast to developing a router, Celerway does not have access to all functions offered by a client device vendor.

For Celerway to optimise the performance of the features listed above, the following enhanced features would be needed:

- **Information about applications:** for instance the type and name, and requirements an application has for performance and treatment. An elaborated list of such information is given in § 3.3.2.
- **Network diagnostics:** to supply interface and path characteristics.
- **Transport support:** with statistics about supported transport options.
- **Network selection:** a process that can select network(s) suited to given application requirements.
- **Transport selection:** a process that selects suitable transport options given application requirements, network diagnostics/selection, and transport support over a selected network.
- **Transport optimization:** to improve inefficient transport protocols and options in mobile scenarios, and particularly in cases where network aggregation is needed to meet application requirements.
- **Signalling:** of information and selections between software on client, router and proxy. Although the application runs on client, knowledge about its requirements on router and proxy would enable more optimized network and transport selections. This will allow IPv6 and endpoint prefix selection to be utilised in implicit signalling to router.

Celerway seeks to support selected features in both a router and a proxy, and to also explore client endpoint implementations. The following scenarios illustrate how the enhanced features could be used with the Celerway products.

A client endpoint that only supports the traditional transport architecture In this case, the router could potentially infer application type and needs, and deploy relevant features of the NEAT System.

In a scenario where *both* a Celerway router and proxy are deployed (Figure 4), the following features could be used:

- **Network selection** between router and proxy.
- **Transport selection** between router and proxy.

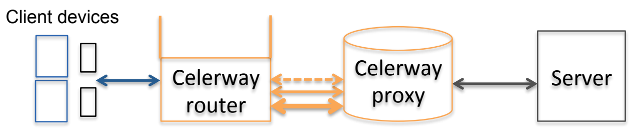

Figure 4: Communication via a path including a Celerway Proxy and Router.

- **Transport optimization** between router and proxy (since in this case Celerway controls two end points, new transport protocols and modifications can be used between the two).

In a scenario where *only* a Celerway router is deployed (Figure 5), a **Network selection** feature could be used.

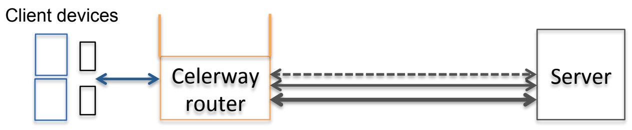

Figure 5: Communication via a path including a Celerway Router.

A client endpoint that supports a new transport architecture and uses a Celerway router In a scenario where *both* a Celerway router and proxy are deployed (Figure 4), *and* the client supports the new transport system, the following features could be used:

- **Signalling the router-to-proxy transport options to a client** to inform client endpoint transport selection.
- **Network selection** between router and proxy, where information includes signalling of application information and requirements from a client endpoint.
- **Transport selection** between router and proxy.
- **Transport optimization** between router and proxy (since in this case Celerway controls two end points, new transport protocols and modifications can be used between the two).

In a scenario where the client supports the new transport architecture but *only* a Celerway router is deployed (Figure 5), the following features could be used:

- **Signalling application information and requirements** from client to router for network selection between router and server. Implicit signalling with IPv6 prefixes could be explored.
- **Signalling transport selection** from the client to the router.
- **Network selection**.

A client endpoint supporting a new transport architecture, but that does *not* use a Celerway router In a scenario where *only* a Celerway *proxy* is deployed (Figure 6), the following NEAT features could be used:

- **Network selection** between client and proxy.
- **Transport selection** between client and proxy.

- **Transport optimization** between client and proxy (e.g., support for new transport protocols or transport modifications).

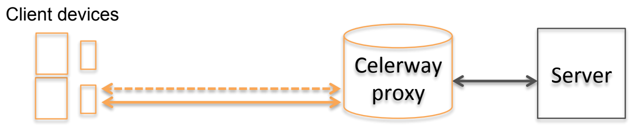

Figure 6: Communication with an updated endpoint via the Celerway Proxy.

Celerway is not aware of any efficient solution supporting all these components and their interactions and may develop its own proprietary user-space solutions on router and proxy. However, it is expected that there can be substantial improvements in performance when compatible support is introduced in the clients and servers.

Celerway's motivations for pursuing this approach are expected to result in significant performance improvements compared to current network selection, transport selection and optimisation. At present these are based on limited information and hence provide a more static selection than desired. Traffic is distributed over different networks optimising overall performance, however, more could be done to optimise the performance of individual applications. Transport selection and optimisations can only be deployed between router and proxy, since a network device does not know the methods supported end-to-end. Furthermore, transport selection and optimisation only consider network performance. Decisions about aggregation and handover are currently based on the overall network performance, and not on the needs of a specific application.

Information about the applications, their requirements and characteristics in combination with knowledge of the path transport is expected to be allow considerably improvements in the selection and decision process. A signalling method for this information to the router and/or the proxy would be needed to gain the full potential. Knowing the characteristics of applications will also improve the accuracy of passive network measurements (diagnostics). With this information available, Celerway software is expected to be able to improve the performance and QoS of applications in addition to improving the overall network utilization.

3.3.2 Requirements derived from the Celerway use case

The following requirements and information from applications have been determined to be useful for the Celerway use case and its network selection, transport selection and optimisation processes.

An application should be able to indicate to a new transport system:

- **Data delivery requirements:** whether it needs real-time or bulk delivery (in contrast to a real-time application, bulk delivery could for instance tolerate a network with higher latency, network aggregation, and buffering).
- **Capacity requirements/profile:** to indicate its capacity requirements (e.g., high capacity; constant or bursty use of capacity).
- **Reliability:** to indicate a requirement for continuous, stable Internet connectivity (e.g., whether LTE should be preferred over a WiFi hotspot, or whether seamless handover should be enabled).
- **Network security:** requirement for security (e.g, impacting network selection by avoiding open WiFi or in the choice of the transport service).

- **Delay budget from the application:** to indicate a requirement or desire for the network/transport to buffer data (e.g., informing a router whether it is desirable to buffer packets to increase throughput/robustness).
- **Metadata about the application:** to indicate its name and type (e.g., this could be useful in cases where requirements are not completely specified).
- **Desire for seamless handover/mobility:** to inform decisions about whether the selected transport service should seek to enable handover and mobility functions.
- **Disable multipath support:** to indicate whether it is able to utilise multiple available networks itself, in which case the transport system should not enable techniques for that (e.g., MPTCP).
- **Prioritization of the requirements:** to indicate priorities that help inform network and transport selection in the case where all the requirements cannot be met.

In addition, metadata about the client device (from the OS) could be useful input to the selection and decision processes. For instance, for a desktop PC we could select a different network and transport protocol than for a mobile smartphone. In addition, for mobile devices low battery level could result in a different network selection.

3.4 EMC Use Case

3.4.1 Description

There has been significant growth in the volume of traffic within and between datacentres. This is accompanied by a growing desire to move large datasets. However, the limitations of existing infrastructures and technologies lead to inefficient, slow and costly data transfers. It is therefore essential to have a more adaptable and dynamic approach to network resource utilisation in an end-to-end context.

The data generated by mobile, cloud, big data and social media traffic will transform the way the IT industry works, connecting billions of users and millions of applications. These datasets will dwarf traditional, structured datasets and a central challenge will involve the transfer and analysis of such large datasets. The transformative impact of this approach on business models will be realised in the many domains that suffer from present-day restricted infrastructures. Applications are becoming more service-aware and will be able to specify their requirements in terms of resources and QoS parameters. The requirement therefore is for more dynamic, flexible and automated provisioning engines coupled with an end-to-end approach to orchestration and management of network resources based on user, service or application requirements. Software defined networking (SDN) techniques can enhance the capabilities of the network to provide solutions to efficiently manage large end-to-end application workflows. Optimisation for network resilience and performance can be achieved by implementing several actions, possibly combined together, like:

- Provisioning of multipath or parallel/disjoint paths.
- Centralised control with end-to-end and application requirement awareness.
- Transport protocol optimisation determined by the actual network (path) status.

A network performance monitoring solution needs to be in place to support these actions. This needs to measure, report and assure the global status of the network service. EMC is interested in a transport service abstraction that can take into account application requirements and network conditions. EMC expects to leverage the transport optimisations provided by such a transport system in conjunction with performance feedback from a SDN controller/orchestrator managing a datacentre network. Such an approach is expected to improve data transfer performance within and across datacentres, using a next-generation smart transport system augmented by the knowledge of the underlying network either seamlessly or with a minimal impact on the applications running over the network. The NEAT EMC use case therefore focuses on applications designed to transfer large data sets. Such applications can gain important benefits if they can map their requirements to the network through a new transport system. Examples of target workloads are:

- **VM/container migration:** A virtual machine (configuration metadata + disks) needs to be migrated from one host to another over a network. The request may come from a user administrating the hypervisor or automatically from the hypervisor itself, for example if the host fails. The priority is to perform the job immediately and as fast as possible, to minimize the downtime of the affected VM. Initial network requirements focus on large capacity and low latency. During the transfer, if statistics from the network show the process is getting slower, a network controller can switch to use an available better path.
- **Data Backup/replication:** A storage system periodically performs data backup and replication from one node to another over a network. This task does not need to start immediately and can also be paused and resumed, but it must be finished by a deadline to ensure the correctness of replicated data. The initial network requirements focus more on the reliability of the network than on required capacity and speed. Incremental backups do not involve a large amount of data and the main objective is to avoid their retransmission. A network controller can decide to start, delay or partition the job if it discovers a later time window with less traffic.
- **Disaster recovery and failover:** When a site or a node suddenly fails, a storage replication can activate a new node and this will likely require data to be moved to synchronise the new configuration. This task has to transfer a large volume of data, and thus performance can be improved by using multiple parallel network paths between the two endpoints. This suggests a need to look for the largest available network path capacity, that possibly needs to be retained for a long period of time.
- **Big data real-time transfer:** Big data analytics applications often need to move a large volume of data within their processing pipeline. If they analyze real-time datasets or streams, transfers from one processing node to another must be fast, highly efficient and optimised. Network requirements are similar to the disaster recovery workloads, with a stronger constraint on the lowest possible latency.

These examples illustrate a varying range of user network requirements: from real time and non-real time/scheduled traffic, to high path availability, and a desirability to support multipath connections.

The path between application endpoints is not a simple point-to-point link, but rather an entire network (LAN or WAN), which might itself be virtualised with the physical network unknown to the applications (or endpoints). The applications consider the network path as a single pipe, but the underlying network may have many hops and potential bottlenecks. This network will be managed by a network controller/orchestrator that is aware of the entire network state providing, for example, multiple parallel paths with different network features and conditions. The endpoints need to perform continuous monitoring of the resources being used. The network controller will also need to monitor the network status to inform optimisations in the transport service offered to applications.

The new transport system needs to be able to work together with the network controller and updated endpoints, as shown in Figure 7. According to the type and request of the application and the status of the network, the network controller will compute the best setup to complete a data transfer. The main goal is to feed information about the network to the transport system to inform selection of the best current possible transport service provided for an application.

Integrating a network controller/orchestrator with the new transport system is expected to enable it to meet the QoS targets and capacity requirements for applications, allowing it to (re)schedule flows in more efficient time windows and to respond to new load/jobs. This integration may be passive or active. In the passive mode, network controllers/orchestrators and other hardware or software equipment provide feedback about the network to the transport system by informing it about network characteristics (e.g., the latest measurements reflecting the status of the network). Therefore it is expected that a third-party network device can influence the policies used for selecting the network and transport protocol. This represents an extension to provide more dynamic features, allowing the characteristics of a network path to be updated by a trusted entity that seeks to customise the default behaviour of a new transport endpoint.

In the active mode, the transport system may use functions from trusted and registered third-party network devices to implement or improve the selected end-to-end transport service, the network controller for example sets up the routing and configures the network paths.

The traffic in this use case considers an example application that move large datasets. In the new approach, such applications should never directly deal with the underlying network. They should instead present high-level requirements and then delegate to the transport system to select protocols and mechanisms to obtain the best performance in their running context. The EMC use case seeks to inform the decisions made to allow the system to tune transport protocol parameters without requiring interaction with an application.

A second goal is to develop an appropriate set of mechanisms to enable integration between the new transport system and a network controller/orchestrator. This requires the system to offer statistics on actual performance and to enable policies used at endpoints to be updated based on changes in the network. This also presents requirements for diagnostics and tools for testing and monitoring the interaction between external sources and the new transport system.

3.4.2 Requirements derived from the EMC use case

The EMC use case expects the following information to be exchanged between applications, the new transport architecture and the entities managing the network (controller/orchestrator).

- An application should be able to indicate to a new transport system:
  - **Application type/name**: metadata useful for tracking and diagnostics.
  - **Capacity requested by the application**: a hint about how much capacity the application needs, rather than a capacity guarantee that the network will provide to the application.
  - **Latency expected by the application**: a hint about how much latency sensitive the application is, without any latency guarantee provided to the application by the network.

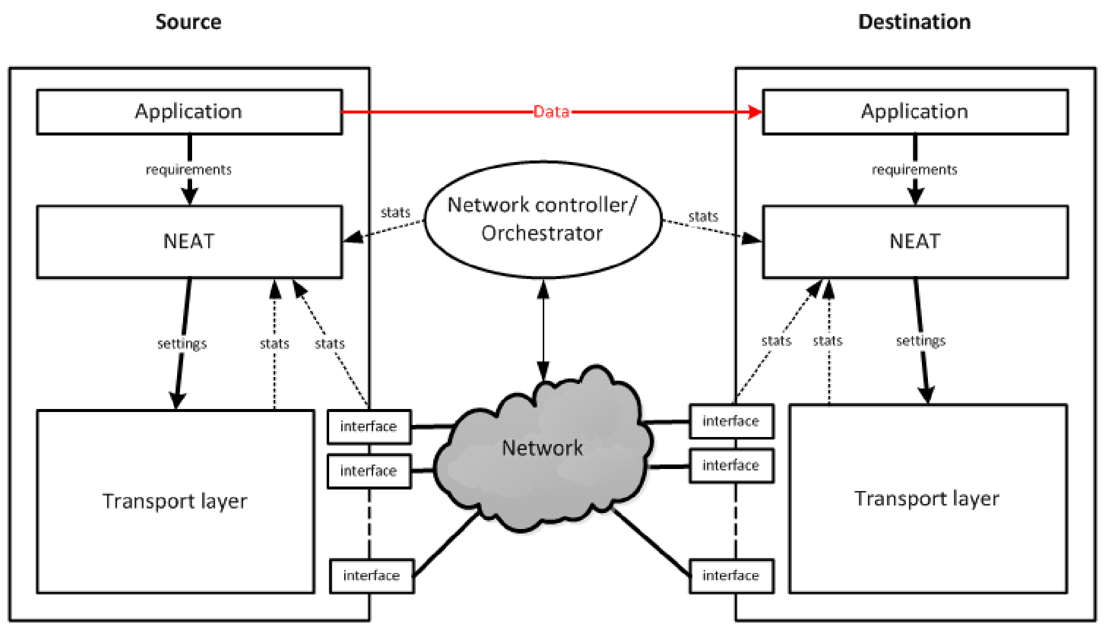

Figure 7: Architecture of the EMC use case showing the way in which a NEAT System could be integrated in a Software Defined Network (SDN).

- **Security needs from the application:** in terms of end-to-end encryption and confidentiality.
- **Disable dynamic enhancement:** applications should be able to tell a new transport system that they do not want it to make modifications to flows that have been set up to react to network status. For example, they might not allow the system to change transport protocol parameters within a session because they already manage network changes using internal algorithms or the application logic would break after particular changes (e.g., to not allow a handover).

• The network (through a SDN controller) should be able to provide information to a new transport system:

- **Latency, packet loss, jitter, throughput and other network measurements**: to inform the transport system and about the end-to-end, up-to-date network characteristics and as an input to inform transport selection.

• The transport system should be able to provide information to an application:

- **Method to determine the transport selected and transport parameters**. This could provide a SDN network controller with visibility to discover the result of choices made and may allow it to verify the configuration is appropriate.
- **Statistics from the network**: to provide details about the network status to specific diagnostic applications.

4 Requirements for an evolvable transport layer

The design of any evolvable transport-layer architecture has to consider a series of general requirements, beyond those imposed by specific use cases (§ 3). Such generic design requirements can be summarised in five categories:

1. Deployability.
2. Extensibility.
3. API flexibility.
4. Guided parametrisation.
5. Scalability.

We elaborate below on these five generic requirements that will realise a system aligned with the major objectives of the NEAT project.

A further goal, that cuts across several of these generic requirements, is the ability to operate in a *best-effort* manner—i.e., making the best use of available features, without any hard guarantees, but providing a fall-back to a solution that can work across the desired network path.

Figure 8 shows the four possible combinations of having or using a path with NEAT-enabled devices and having or not having a NEAT-enabled host as a the receiving endpoint—from the best case (arrow A) to the worst (arrow D). All combinations require at least a change to the stack at one end point. A proxy for the new transport-layer architecture could also provide these functions on behalf of an endpoint that is unable to support the new architecture. For clarity, such proxies are not shown in this figure and a sender supported by such a proxy would be regarded as "NEAT-enabled" in the context of Figure 8.

While adhering to the requirements listed above, the NEAT System must not degrade transport functions that are currently available. For instance, security and mobility must at least work as well with the new architecture as they do with a system that does not support this architecture. However, the NEAT System is expected to provide opportunities to improve both security and mobility (examples are elaborated in Section 6).

4.1 Deployability

One of the main goals of NEAT is to create a transport-layer architecture that can be deployed in the Internet with as little disruption as possible. This requires avoiding a complete overhaul of the Internet architecture, network devices or endpoints (hosts), which rules out any clean-slate approach. Moreover, the design of the NEAT System should consider one-sided, incremental deployment (possibly at the cost of providing less benefit), but enabling a sending endpoint to start using the NEAT System before the destination endpoint and/or middleboxes have been updated. Further, the design should enable fast adoption of the NEAT System.

Deployability, as outlined above, thus translates into the following specific requirements:

Application focus: The ability to evolve the architecture requires that the NEAT System can be used on existing host operating systems. It must be installable, usable and upgradable for users without specific privileges. This allows evolution of the NEAT System to be decoupled from the evolution of the operating systems (OS) and enables it to be supportable on deployed platforms.

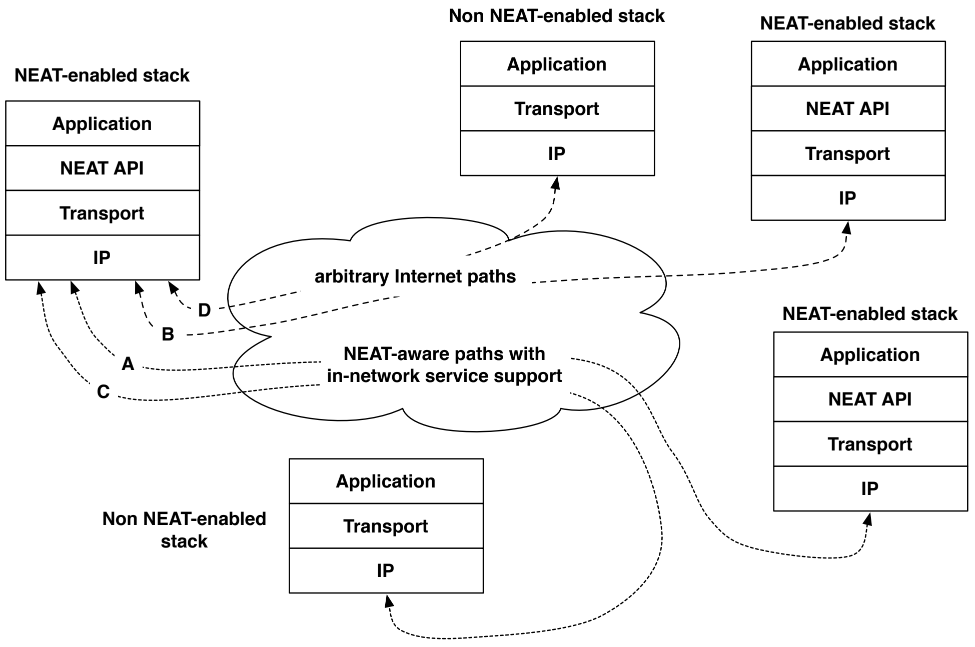

Figure 8: Possible NEAT deployment scenarios: (A) a direct connection between NEAT-enabled endpoints over a path that understands NEAT; (B) NEAT-enabled endpoints over a path that does not understand NEAT; (C) a connection to a non-NEAT-enabled endpoint over a path that would understand NEAT; (D) a connection to a non-NEAT-enabled endpoint over a path that does not understand NEAT.

Host operating system feature tolerance: The NEAT System should not only make use of protocols and features already available on the host OS, but should also provide support for any additional protocols and features as an integral part of it (e.g., user-level transport stacks should be supported to enable easy deployment of new transport protocols and/or transport protocol components).

Peer feature tolerance: The design must not require all communicating parties to use the same version of the NEAT System. This robustness principle also extends to the components (protocols and mechanisms) used to realise a Transport Service.

Network feature tolerance: The ability to use the NEAT System must not require the network to support specific features (e.g., quality of service or middlebox interaction mechanisms).

An endpoint using the NEAT System could be able to further improve performance and/or interoperability when it is able to communicate with NEAT-enabled network devices along a path. Any signalling must also be provided in an *best-effort* way to ensure that a lack of signalling does not hinder the Transport Service. Examples include: signalling to facilitate connectivity (e.g., relaying port and protocol usage to a NAT or firewall), signalling flow priority, or signalling a variety of other information (e.g., SPUD).

4.2 Extensibility

A key goal of NEAT is to fight ossification of the Internet's transport layer. Therefore, the NEAT System must be able to support seamless, *independent* evolution of the different components of the end-to-end communications chain:

Support for evolution of the transport system: The NEAT System must: (a) provide a way for future addition of components (protocols and features); (b) allow new (possibly experimental) components to be used as support for them is introduced into network devices, without necessarily updating the application; (c) allow moving the implementation of components into the host operating systems (e.g., the OS kernel) and vice versa during evolution of a component.

Support for operating system evolution: The interface between the NEAT System and the OS must be able to use standard methods. These methods may change over time to improve the service provided by the NEAT System. However, this must not require changes to applications using the NEAT System.

Support for network evolution: The NEAT System must allow evolution of methods used to interact with network devices. The NEAT System must permit integration of new mechanisms and allow applications using the system to benefit from new mechanisms without requiring modification of the applications.

4.3 API flexibility

The NEAT System must support different levels of abstraction to access the services provided; it should be possible to decouple the offered Transport Services from the specific choice of, e.g., transport protocol and options chosen to implement a service. The design should also permit future-proof use and offer a simple pathway for porting existing applications. This means that the API—i.e., the only way applications interact with the System—must be flexible, in the sense of the following requirements:

Backwards compatibility: The NEAT System should permit evolution of the system without affecting programs using it, i.e., the API must provide backwards compatibility to different versions of the NEAT System.

Support of high level configuration: The NEAT System must also allow for configuration using a much higher level of abstraction than that offered by the Socket API:

- It should provide mechanisms to describe the needs of an application in a generic way. This requires definition of a set of API parameters to characterise the required Transport Service. Possible needs include message-orientation, preservation of message order, reliability, low latency, mobility support, relative priorities and security features.
- The application can assume that the selected Transport Service Instantiation satisfies the API request (or an error indication is returned), but should not implicitly assume additional ones. This provides an optimal way for the NEAT System to make any further decisions necessary to establish the communication with the peer endpoint.
- It should allow evolution. This should allow new additional choices that offer a more suitable service to be selected without the need to change an application.

Support of low level configuration: The NEAT System should continue to allow the detailed configuration provided by the classical Socket API. The Socket API requires that applications specify the network and transport protocols and select the protocol-specific parameters when values different from the defaults are needed. This selection specification should also be possible in the NEAT System.

Comprehensibility: Although it automates the selection of components, the API must make low-level information available to the application to understand why particular components where selected and configured.

4.4 Guided parametrisation

A NEAT System must provide *guided parametrisation*. Current transport and network stacks are explicitly parametrised. For instance, using the Socket API, an application program may choose IPv4 or IPv6 and select DCCP, SCTP, TCP, UDP-Lite or UDP, and several parameters can be specified by explicit socket or protocol-level socket options.

This is different to the approach being developed in NEAT, where the NEAT System must consider both application requirements and features that may need to be supported along the entire end-to-end path, to create an appropriate Transport Service Instantiation that makes the best of available features. Such *guided parametrisation* corresponds to the following requirements:

Derivation of parameters: The NEAT System must map the high-level requirements provided by an application to the low-level parameters actually being used. This mapping performed is guided by both the requirements provided by the application, and the appropriate policy. The result is the selection of the transport protocol stack, the network protocol, the interfaces used, and the parametrisation of each component.

Dependency on local tools: If possible, tools/techniques provided by the OS should be used when making selection decisions.

Dependency on the peer endpoint: The NEAT System needs to take into account the components (e.g., protocols) supported by the local and remote endpoints. This requires discovery of the set of transport protocols (and features) available at a particular endpoint address.

Dependency on the network path: The selection of the appropriate path needs to consider the current network state. The local endpoint needs to verify that components supported by the peer endpoint are actually supported by the current network path to the endpoint.

Also, the selected protocol and path need to be robust to any middleboxes located on the network path. This may include discovering middleboxes on the path and identifying suitable ways to interact with them, as well as optional mechanisms to detect the support of network features, such as mechanisms to provide quality of service, or flow priority.

Dependency on time: The NEAT System must be able to optionally support applications that wish that the System responds to changes in network state. An application may decide that the NEAT System should be allowed to select an alternate path to improve the performance of the ongoing communication (e.g., by using MPTCP, SCTP, SCTP-CMT, or other methods).

Application protocol agnostic: Mechanisms to discover the path and supported components must not require any change to, or specific support by, the application protocols.

4.5 Scalability

Any implementation will be limited by its own design choices. However the *architecture* of the NEAT System must itself be scalable along several dimensions:

Size of feature set: It should support a variety of combinations of protocols, configuration options and network interactions. The process for selecting among these combinations must be able to accommodate a large number of possibilities while providing an acceptable communication setup time.

Traffic volume: It must support communication that requires a path with high capacity or low latency (or both). This will dictate a need to minimise the impact on processor load.

Number of peers: The architecture must permit a large number of NEAT Flows to be supported efficiently, although the number of NEAT Flows that need to be simultaneously supported depends on the use case.

5 Architecture Design

The NEAT System is a layered architecture that provides a flexible and evolvable transport system. The applications and middleware served by the NEAT System utilise a new NEAT User API that abstracts network transport. The NEAT System can provide Transport Services in a way that allows the best transport protocol to be used by an application without the application having to handle selection from application code.

5.1 Overview of the NEAT Architecture

Figure 9 presents an abstract, high-level overview of the NEAT Architecture. This diagram depicts both applications and middleware making use of the NEAT System (solid arrows), and traditional "non-NEAT-enabled" applications and middleware (dotted arrows) that use e.g. the Socket API.

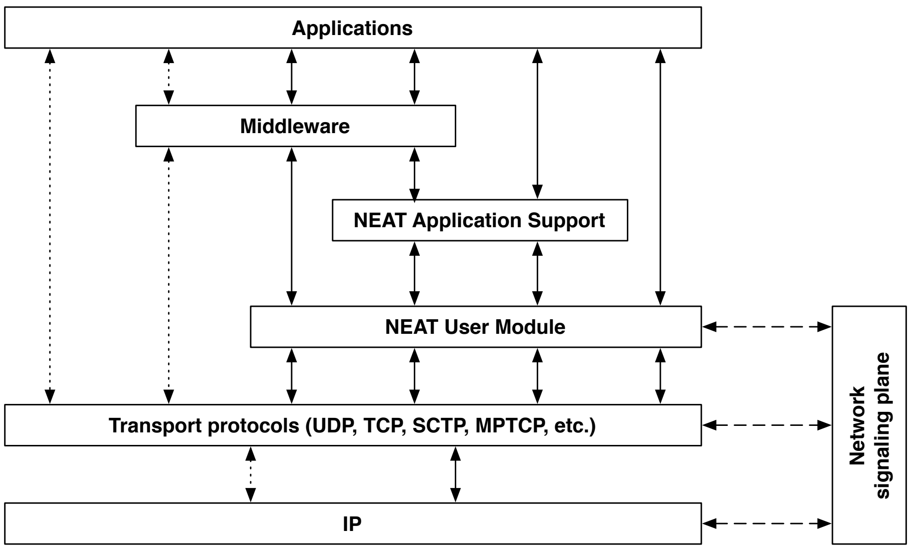

Figure 9: NEAT Architecture: A high-level overview.

The main part of the NEAT System is the NEAT User Module. 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.

The NEAT Application Support Module encompasses helper libraries and functions that applications could use to access a more abstract transport system. This uses the services of the underlying NEAT User Module.

Figure 10 provides a more detailed overview of the components of the NEAT System and its interfaces.

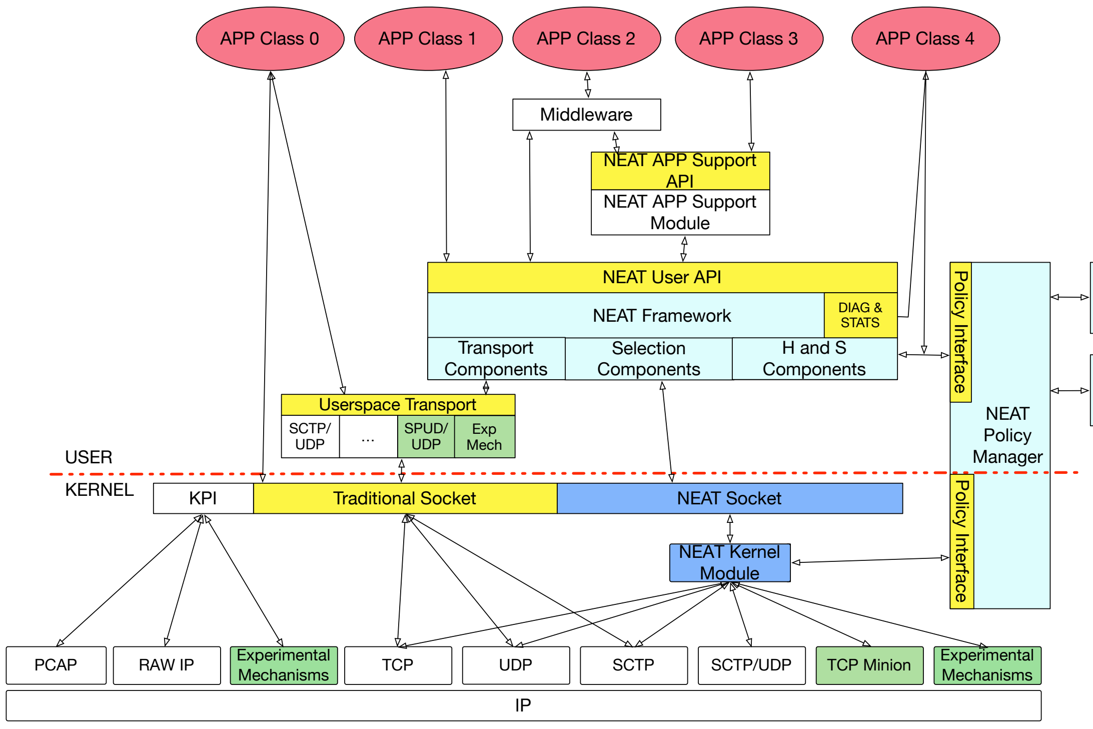

Figure 10: Components and interfaces to the NEAT System. The NEAT User Module is composed of the all blocks shown in light blue (NEAT Framework, Transport, Selection, Signalling & Handover, and Policy Components) and related APIs (NEAT User API, Policy Interface, Diagnostics and Statistics Interface).

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 core functions of the NEAT User Module.

A group of components are responsible for 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 in the kernel of the platform OS. Other transport protocols are provided in user space, but may optionally also be provided in 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 NEAT Signalling and Handover components. The NEAT User Module and its components are further detailed in Section 5.4.

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 10 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.

5.2 Applications and the NEAT System

Applications using the NEAT System can be classified into one of a set of classes with respect to the API that they use to access the network:

- **Application Class 0:** Applications using the traditional socket API. This is the default behaviour before the NEAT System is introduced. These applications are included here to contrast with the other application classes.
- **Application Class 1:** Applications that use the NEAT User API directly. The long-term vision is that this becomes the default for Internet applications.
- **Application Class 2:** An application that uses middleware can indirectly access the Socket API. Once the middleware has been updated to support the NEAT User API, this class of application uses the NEAT System. This class of application might not be aware of the NEAT System at any level.
- **Application Class 3:** This class of application can use the NEAT Application Support API, or could directly use the functions and example code provided for NEAT applications support. A Class-0 Application may be rewritten using the help provided by the NEAT Application Support module to enable it use the NEAT System.
- **Application Class 4:** NEAT provides a diagnostics interface, allowing applications to gather statistics about the NEAT System and NEAT Flows. This interface can also provide access to debugging information. A Class-4 application is specifically developed to take advantage of this interface in addition to the NEAT User API. An example of a Class-4 application is one designed to monitor and control the network behaviour of other applications, for instance to interact with an SDN Controller when operating in a SDN environment.

5.3 Principles for design

This section elaborates on the design principles of the NEAT architecture from different design and implementation aspects.

5.3.1 NEAT's core in user space

The NEAT architecture separates functions into operating system (i.e., kernel) and user-space components.

It is expected that a key success factor for the deployment of NEAT will be the ability to operate across multiple OS platforms (e.g., Linux, FreeBSD, Mac OS X and Windows). Therefore all core components needed in NEAT (including the NEAT User Module) are being implemented in user space. This approach promotes *portability* and *deployability* across multiple OS and hardware platforms.

The choice of a user space implementation eases the introduction of new (or existing) transport protocols in NEAT. User-space implementations of such transport protocols tend to be more portable and can potentially run on multiple platforms. This approach can enable easy introduction and testing of new features. This has the potential to make many *failed* Internet standards deployable, boosting innovation that goes well beyond what this project will develop, resulting in a possible impact on a range of protocols at/below the transport layer.

However, the use of user-space transports presents a range of challenges. One major challenge is that network I/O operations that originate in user space can incur higher latency compared to network I/O operations handled in the kernel. However, solutions such as, or similar to, [9, 13, 55, 56] can provide fast packet processing (mostly available in Linux user space via the KPI) and can mitigate this drawback. For instance, MultiStack [29] provides commodity operating systems with support for dedicated user-level network stacks [55, 56]. It can concurrently host a large number of independent stacks, and can fall back to the kernel if necessary. It provides high-speed packet I/O at rates up to 10 Gb/s [29].

5.3.2 Optional kernel support for NEAT and the NEAT Kernel Module

NEAT allows some Components to be implemented in the OS kernel. The division between the kernel and user space is represented in Figure 10 by a horizontal dashed line.

Some protocols are already widely available as kernel implementations and are commonly supported by various Operating Systems (such as UDP or TCP). These may be accessed by the NEAT User Module via the Socket API.

A NEAT System *may* choose to introduce additional NEAT Components in the NEAT Kernel Module (accessed via the associated NEAT Socket Interface). There are various reasons to implement extensions in the kernel. Four major cases where NEAT kernel support *may* be desirable are: (1) whenever it is required to perform functions across multiple processes, and not per-process; (2) for performance enhancement, e.g., making use of hardware acceleration etc. where the solutions mentioned in § 5.3.1 may not be adequate; (3) whenever access rights prohibit sending/receiving data, e.g., for a new usage of TCP; (4) to access new or experimental functionality (such as transmission/reception of certain network signalling protocol packets).

As depicted in Figure 10, the Policy Manager *may* also introduce Kernel Components within the control plane (see § 5.4.3).

NEAT Kernel Module support is optional in the NEAT System. The implementation of kernel components will vary by OS (e.g., Linux, FreeBSD, Android and Windows). Changes to these modules will typically imply a kernel change, to update the interface, and also to create NEAT entities. These enhancements are not core parts of the NEAT System, and therefore to ensure portability it is expected that all components that are not widely available will also be implemented in user space.

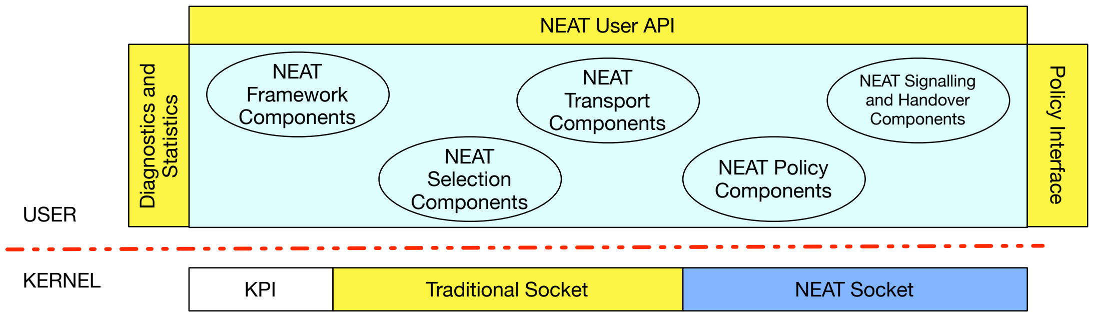
NEAT User API
Diagnostics and Statistics
NEAT Framework Components
NEAT Transport Components
NEAT Signalling and Handover Components
NEAT Selection Components
NEAT Policy Components
Policy Interface
USER
KERNEL
KPI
Traditional Socket
NEAT Socket

Figure 11: The groups of components and external interfaces used to realise the NEAT User Module.

Table 2: The five groupings of components provided by the NEAT User Module.

No.	Name	Description
1	NEAT Framework	Things relating to the system: API/Statistics/Diagnostics
2	NEAT Selection	Choice of Transport Service
3	NEAT Policy	Default, application, path information
4	NEAT Transport	Components to configure and manage transport protocols
5	NEAT Signalling and Handover	Components that support on-going usage of the service

5.4 The NEAT User Module

The NEAT User Module takes care of all aspects required to create a Transport Service Instantiation for a NEAT Flow. The module is implemented as a set of NEAT Components.

Figure 11 and Table 2 present the five main groupings of NEAT components forming the NEAT User Module. The three external interfaces (Figure 11) presented by the NEAT User Module are: the NEAT User API (an interface to the upper layer protocol/application); the Diagnostics and Statistics interface (an interface that supports measurement, tracing and debugging) and the Policy Interface (the interface to the policy manager).

The NEAT Framework components (1) comprise the functionality required to use the NEAT System. This defines the structure of the NEAT User API and the interface into the NEAT Logic that implements the basic mechanisms required by NEAT.

NEAT-based applications use the NEAT User API to provide information about the requirements for a desired Transport Service and to determine the properties of the offered Transport Service. It is this additional information that enables the NEAT System to move beyond the constraints of the traditional Socket API, since the transport system then becomes aware of what is actually desired or required for each NEAT Flow.

The NEAT Framework also includes components for diagnostic, debugging and measurement interfaces. These allow access to statistics and information similar to the traditional Socket API, but related to a NEAT endpoint. The additional information reported by NEAT can be used to identify which NEAT Components are in use, and how these have been configured.

A set of components provide NEAT Selection (2) and NEAT Policy (3). Together these are core parts of the NEAT System that determine the composition of the protocols / mechanisms that will provide the service to the application.

NEAT Selection components are responsible for choosing an appropriate transport endpoint and set of protocols / mechanisms. This utilises the information passed through the NEAT User API, and combines this with inputs from the NEAT Policy Manager, accessed via the user-space Policy Interface. This defines sets of default policies for the system or policies for individual applications (when specified). Together this is used to derive a set of candidate transport services.

After identification of candidate services, the NEAT Selection components will test the suitability of the candidates utilising information known about the path (also made available via the Policy Interface) and by attempting to make endpoint connections. Parts of this algorithm may be performed in parallel to avoid unnecessary delay, using a "Happy Eyeballs" mechanism. If no viable Transport Service can be found, the NEAT connection fails.

The Selection and Policy components enable NEAT to make an appropriate decision based on application requirements and what is made available for a network endpoint. Making these decisions at run time within the NEAT User Module, rather than at design time for an application, ensures an appropriate choice is made, provides opportunities to take into consideration multiple (possibly conflicting) constraints—and avoids each application to code for the possibility that a path does not support a particular mechanism or combination of mechanisms.

Another set of NEAT components are responsible for providing the functions required to configure and manage the NEAT Transport Service for a particular NEAT Flow. Within this set, NEAT Transport components (4) manage a set of endpoint transport protocols and other mechanisms needed to realise the Transport Service. The components provide functionality to configure protocols or mechanisms such as SCTP/UDP, TCP, LBE congestion control, or TLS and DTLS.

NEAT Signalling and Handover components (5) provide extended functions that complement the functions of the NEAT Transport Components. These functions provide advisory signalling to the network, communication with middleboxes, support for fail-over or hand-over, and other mechanisms that are used during the progress of a NEAT Flow.

The abstraction provided by the NEAT User API enables evolution of the transport system. The Selection and Policy components allow an application to be coded independently of the transport protocols currently available (e.g., an application does not even need to know if SCTP is implemented on a current platform, but can use SCTP when it is). One key advantage is that new transport protocols introduced at the NEAT endpoints or new signalling mechanisms that communicate with the network can be used as soon as the network path enables their usage, rather than relying on specific support in individual applications.

While the full details of NEAT Components are not specified in this document, and will be defined in Work Packages 2 and 3 of the NEAT project, we provide some further information on these components below.

Core and Extended NEAT Components NEAT Components can be categorised as being *core* or *extended* components:

- **Core components** are mandatory for the operation of the NEAT System.
- **Extended components** are desirable, but not essential. They provide additional flexibility, further enhance performance or offer additional functionality.

Core components of the architecture include the NEAT Framework components, a Policy Manager with a basic policy system, methods for transport selection and transport protocol components. These components are required to realise a (core) NEAT System.

The extended components are considered a part of an extended NEAT System. These include more advanced policy functions and the inclusion of external CIBs. An extended NEAT System also

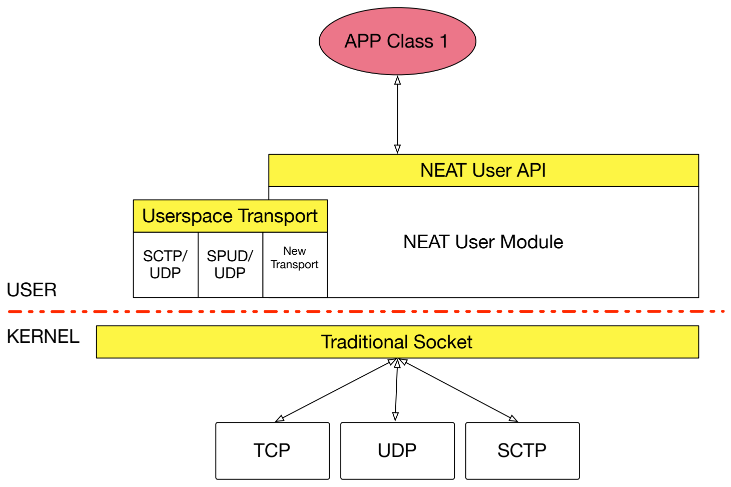

Figure 12: Using the NEAT System with Class-1 Applications.

introduces the possibility of implementing other NEAT components in the OS kernel. All signalling is considered a part of the extended NEAT System, in particular the use of signalling between NEAT applications / end points and network elements. Support for mobility and intelligent use of multiple interfaces is also a part of the extended NEAT System. Example mechanisms include the use of redundancy for applications in need of robust transmission or intelligent path scheduling algorithms for mobile applications.

NEAT allows new transport protocols to be added in an extended NEAT System, such as QUIC, or new transport protocol components, such as LBE or coupled congestion control. To support applications that run on nodes that do not support the NEAT User API or to offer extended functionality, an extended NEAT System may also implement a proxy solution and other NEAT Application Support functions.

The set of extended components selected for implementation within the project will be described in Work Package 3 deliverables, whereas the core components will be reported in Work Package 2 deliverables.

5.4.1 NEAT Framework Components

In the following, various components of the NEAT Framework are briefly described.

NEAT User API Applications access the NEAT System using the NEAT User API, as illustrated in Figure 12. This allows a NEAT Flow to be bound to a transport determined by the NEAT User Module. This is intended as the default behaviour for applications using the API.

The NEAT User API is the interface through which an application requests a Transport Service from the NEAT System. The details of the API will be specified in Deliverable D1.2.

The API allows applications to send hints to inform selection of the desired Transport Service (type of application, expectations for path, transport protocol features and additional constraints to override default policies). The NEAT User API and its associated statistics interfaces also allows a NEAT Flow to be informed of the properties of the NEAT Transport Service Instantiation (e.g.,current policy, actual transport protocol used, usage statistics relating to the NEAT Flow, etc.).

This API shall also allow an application to connect without specifying the actual transport protocol to be used, the network-layer (destination) address to which the NEAT Flow is to be bound, or the interface (source) address to be used. The NEAT System will perform this selection as a part of creating the NEAT Transport Service Instance. Importantly, it will enable a move to specifying a service instead of a particular protocol. Such a service could then be realised by any means, including an upcoming or future transport protocol.

In some cases there are benefits for an application to specify via the API that it can only use one particular transport (e.g., TCP on a specific port). In this case, the NEAT System could still provide benefit using the policy information—for instance for interface selection or DSCP or other signalling that could be even dictated by a policy for the application.

Event-driven (callback) API The NEAT Architecture assumes *callback-based functions* provide the NEAT User API. This decision may have implications, in terms of pros and cons. However, we believe the benefits of an event-driven approach outweigh any potential disadvantages:

- Some application developers may not be familiar with writing callback-based functions, and may then need to learn this new method or use an abstraction library that masks this approach from the developer. However, callback is becoming the new norm (e.g., Javascript, network code in iOS or Android, ...), and most modern server-side applications already use a callback-based approach anyway.
- A callback-based approach is faster (in terms of latency) and also more scalable in terms of the number of concurrent flows that may be supported with a given workload. To assist application developers unfamiliar with the callback approach, simple applications can be provided by NEAT developers as sample code that demonstrate how to use the callback-based NEAT User API. In addition, developers who have already implemented existing libraries using callback-based approach over sockets may not need to completely rewrite their code to make NEAT callbacks happen.

A full discussion of event-driven versus traditional Socket API approaches will be provided in the companion document (Deliverable D1.2) that presents in detail the NEAT User API.

Diagnostics and Statistics Interface The components for diagnostic, debugging and measurement interfaces allow access to statistics and information similar to the traditional Socket API, but related to a NEAT endpoint. The additional information reported by NEAT can be used to identify which NEAT Components are in use, and how these have been configured. Application-level debugging and measurement can be offered by directly writing against the NEAT User API, and utilising the associated statistics interfaces.

Tools (Class 4 applications) can offer support for NEAT debugging. Such tools are essential in supporting deployment and in ensuring scalability of the system.

Class 4 Applications may also access NEAT policy via the Policy Interface, and combine this with statistics collected from the NEAT User Module to build a picture of the service being offered via NEAT. One application of this type could be to monitor the operation of a SDN node, and to communicate the network information to a SDN control node.

5.4.2 NEAT Selection Components

The NEAT Selection components provide high-level 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 service.

In the first part of the selection, information passed through the NEAT User API is combined with inputs from the NEAT Policy Manager (accessed via the Policy Interface). Policies can be pre-configured or based on the experience on using specific paths or endpoints. Together the policy information and requirements are used to derive a set of one or more candidate transport endpoints and a set of candidate protocols and mechanisms.

After identification of candidate services, the next step is to test the suitability of the candidate(s) utilising information known about the path (also made available via the Policy Interface) and by attempting to make endpoint connections.

When more than one set of candidates are available that each could satisfy the requirements, the NEAT Selection components need to make a selection between the possible candidates. This may employ a method that attempts parallel connections across the selected network path to determine which candidate(s) can achieve end-to-end connectivity. Parallel probing avoids unnecessary delay and is also known as “Happy Eyeballs”. NEAT *may* also use other probing methods, including the possibility of end-to-end signalling between NEAT Systems to help identify the set of supported services at a remote NEAT endpoint.

In some cases, a single transport could be available in multiple encapsulations. For example, the SCTP protocol could be available both natively (directly over IP) or via a UDP over IP encapsulation. The NEAT User Module can therefore make the choice of which encapsulation to use.

The complete process includes choice of the interfaces used, choice of the network protocol, selection of the transport protocol, and the parametrization of each layer. If no viable transport service can be provided, the NEAT Selection components will return an error.

5.4.3 NEAT Policy Components

The Policy Manager is invoked whenever a new NEAT Flow is created. Depending on the requirements of the application and the configured policies, the Policy Module will identify an appropriate candidate network interface, transport protocol and transport protocol parameters. The characteristics of the local and peer system as well as the network state will be taken into account when this decision is made.

An example is access to path information to an endpoint that can help select whether this path is chosen for a later connection towards the same/similar endpoint (e.g., to minimise cost or latency, or to maximise throughput). NEAT components can share information via the CIB. This allows one NEAT component to benefit from experience of other NEAT components.

The Policy Manager has four main components:

- **Policy Information Base (PIB):** A set of policies will define the way different requirements, end systems and environment properties influence the interface, protocol and configuration selection. The PIB is the repository where these policies are stored.
- **Characteristics Information Base (CIB):** A repository that contains cached values of dynamic and static characteristics observed for the network, local and peer endpoint (interfaces, protocols, path properties) collected from different CIB sources.

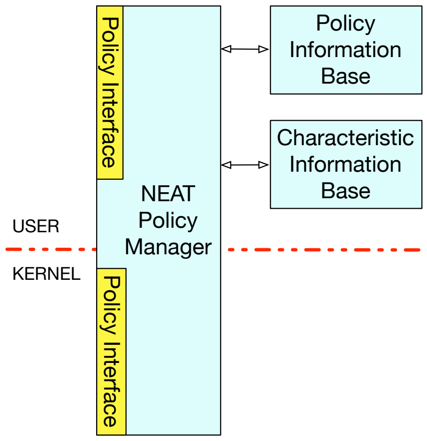

Figure 13: The NEAT Policy Manager.

- **Policy Manager (PM):** An entity that maps application requirements and CIB data to a matching policy in the PIB to drive the selection of one or more candidate NEAT transport services.
- **Policy Interface (PI):** An interface to allow communication with other parts of the NEAT System.

In the simplest form, the policy information could be a shared file (describing a static PIB), but it could also be a shared data structure, a daemon process, a multicast message bus, etc. There are multiple possibilities each with merits (some better suited to user-space implementation)—but all are hidden from the application by the Policy Interface.

The key parts of the Policy Manager are implemented in user space as components of the NEAT User Module. The architecture also permits a NEAT System to implement some components in the kernel, for example to interact with kernel transport protocols, signalling or interface functions. Applications are not expected to directly interact with these Kernel Components, and all communication between applications and the PM is expected to be via the user-space Policy Interface.

Tho following work flow illustrates the operation of the Policy Manager:

1. The PM receives indications of requirements from the NEAT User API (e.g., high capacity, bulk download, not delay sensitive).
2. The PM polls the CIB for relevant information (e.g., available interface types, path characteristics, etc.).
3. The PM extracts the active rule set for the current application from the PIB.
4. Using the extracted policies, CIB data and application requirements the PM selects a candidate configuration comprised of a set of feasible options (e.g., use interface 1 with TCP Cubic, with Nagle disabled).
5. The selected configuration is returned to the NEAT System.

Policy Information Base The following Policy Information Base repositories are distinguished:

- **NEAT System-global:** This repository contains the default set of rules provided by the NEAT System.

- **External system-global:** This repository contains policies provided by operating system providers, device vendors, etc.
- **Application-local:** The Policy Interface allows application developers to set application-specific policies.

The Policy Manager combines policies in the three PIBs repositories into a single set of valid rules. Conflicting policies will be resolved by prioritising the Application local policies over the global policies and the External system policies over the NEAT System policies. System global policies are combined during initialisation of the NEAT System and serve as the default. Application local policies override the default policies only for a NEAT application instance. For each application, the set of valid rules is static and will not change at runtime.

Policy updates are possible during an update of the NEAT System, operating system or the application by importing the policy information to the PIB from a configuration file.

Characteristics Information Base The Characteristics Information Base (CIB) collects information about the characteristics of the local and remote endpoints together with information about the network path. The following CIB categories are distinguished:

- **NEAT default CIB Information:** Network and transport statistics provided by the standard operating system tools as well as information provided by other NEAT User Modules are collected in this CIB.
- **NEAT extended CIB Information:** Information provided by extended NEAT User Modules. This has the same trust level as the default CIB entries and could be stored in the same repository.
- **External CIB Information:** Information provided by external entities.

Some information concerns the local endpoint (e.g., implemented set of transports, support for IPv4/IPv6 and information about hardware interfaces). Information about local configured system properties is static (a result of configuration). Much CIB information is learned from management information of active NEAT Flows and needs to be collected by the PM. This could be via the user-space Policy Interface—e.g., retrieved via snap or iostat calls. It could also be provided at the kernel level (e.g., via access to shared data structures).

All learned information is ephemeral, that is, it is time dependent. Information collected at one time may not be correct or appropriate at a later time. This requires the information to be managed, and is to be regarded as starting point for tuning a transport component or for probing the network to discover actual availability. The information may be subdivided as follows:

- Local information about local interfaces (e.g., available interfaces, MTU, interface properties).
- Local interface characteristics provided by an OS (e.g., interface rate, signal strength, network type, passive and active measurements).
- Local information about the network layer (e.g., support for IPv4/IPv6, tunnel support, security functions).
- Local information about the availability of signalling mechanisms (e.g., support for PCP, SPUD).
- Local availability of transport protocols.

- Path characteristics information derived from various passive and active measurement techniques (e.g., RTT, PMTU, information about middlebox policies, etc.).
- Path information about protocols supported over a particular path. This information may be obtained from previous sessions and the results of happy-eyeballs mechanisms (e.g., transport protocol, IP version support, etc.).
- Peer information about the network layer (e.g., support for IPv4/IPv6, tunnel support).
- Peer availability of transport protocols (transport protocols available at a remote endpoint(s)).
- System information and metadata (provided by the OS, e.g., socket statistics, CPU usage, memory usage, battery drain, etc.).

All CIB information is regarded as system global. However, applications should be able to indicate which CIB instances and CIB sources they trust, for instance by setting an application-specific policy. The default behaviour could be that only the NEAT system and the NEAT extended CIBs are to be trusted.

External contributions (CIB sources) could come from device and OS vendors or other applications for measurements, statistics, metadata collection and improvement of transport protocol selection.

The PM uses a pull mechanism to access information from the CIB when requested by the Policy Interface. As an extended version, the PM could offer a service that improves the transport characteristics during a life-time of a connection, e.g., by switching to a better network interface and another transport protocol if this becomes available, etc. For such a use-case, the CIB could also push updates to the PM when CIB values reach a predefined threshold that could affect policy decisions.

5.4.4 NEAT Transport Components

The NEAT User API offers applications seamless access to the 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 System is expected to provide access to currently standardised transport protocols including:

- TCP/IP (Native).
- MPTCP/IP (Native).
- UDP/IP (Native).
- SCTP/IP (Native).
- SCTP/UDP/IP (Encapsulated).
- TLS/TCP/IP (Native).
- SCTP/DTLS/UDP/IP (Encapsulated).

Many of these transport protocols can provide more than one service and the NEAT Transport Components are responsible for configuring the protocols in the correct way. 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).

5.4.5 NEAT Signalling and Handover Components

A best-effort IP network does not rely on any signalling to network-layer devices. Signalling is an optional extension to a NEAT System. Various interactions may be considered:

- Application Signalling: End-to-End Control/Management (using a protocol such as RTSP or SIP).
- NEAT Transport Signalling: NEAT to NEAT Capability discovery (e.g., discovering the set of supported remote transports to populate the CIB).
- NEAT Transport Signalling: NEAT to NEAT Control/Management (e.g., to enable interaction with the remote peer).
- Network Path Signalling: End to Path Intention (one-sided indication of something about a flow to network devices, e.g., DSCP, PCP, message flow priority, interaction with middleboxes).
- Network Path Signalling: Path to End Capability (allowing the network to inform endpoints of available resources e.g. , capacity hints).

NEAT Transport Signalling can be utilised in the architecture when both candidate endpoints support the NEAT System. If supported, this could exchange capability information between the endpoints and thereby provide inputs to the Policy Manager concerning what capabilities can be provided by peers.

NEAT Transport Signalling could also be utilised when a NEAT System is able to communicate with a NEAT-enabled middlebox within the network, or to control the communication over the end-to-end path between NEAT peers.

Application Signalling logically occurs above the NEAT System, controlled by the application. However, this may interact with NEAT Components, since it often includes description (e.g., in SDP) of transport and network protocol parameters.

Network Path Signalling is used to communicate with network elements present on the path between two NEAT endpoints. This could be as simple as introducing a particular differentiated services code point, or as complex as using a purposely designed network signalling protocol.

By abstracting the interface to these modules from the application to the transport system, this is expected to increase the extensibility and ease deployment of new methods.

5.5 NEAT Application Support Module

The NEAT Application Support Module provides advanced functions (or abstract methods) that create an alternate way to access the NEAT System.

For instance, the NEAT Application Support module may take the form of example code that some applications may directly incorporate and adapt into the code for the application. Application Support functions could be encoded into primitives for communication that form an application-specific API, above the NEAT Application Support API.

We envision the possible need for a shim layer that resembles the NEAT Socket API, to ease migration of current applications designed for the traditional Socket API (i.e., Class-0 Applications) to the new architecture. Applications using the shim could be unaware of the transport mechanism used and instead operate using a traditional Socket API. If provided, this would form a component of the NEAT Application Support Module.

Flexibility is needed. Different classes of upper layer protocols may require different functions, and this could motivate creation of multiple sets of example code (a bulk sending application such as netperf, an interactive example such as a response time test, etc).

6 Examples using the NEAT System

This section provides a set of examples that illustrate the ways that applications can use the NEAT User API to provide information that can then be used by proactive policies to choose the appropriate transport, IP protocol version, select an interface, tune network parameters or report network conditions.

6.1 A mobile application example

This example shows how a mobile application can be enabled to move beyond the constraints of the traditional Socket API, responding to changes in network conditions.

A mobile application built to use the NEAT System can benefit from network mobility when the system supports multiple interfaces. NEAT provides help in selection of an appropriate interface, by enabling the application to express its requirements for the network service (e.g., minimise latency, optimise for throughput, optimise for reliability), and allowing the NEAT System to make decisions about which interface to choose at run time, rather than compiling the decision-making logic into the application. This allows for the policies to be updated (by changing entries in the PIB), or to be dependent on the current network conditions (reflected in the CIB).

The selection of the Transport Service within the NEAT User Module can include functional requirements (such as the need for message reliability, ordering, integrity protection) and select the best path to satisfy multiple constraints.

The NEAT System can also help make an application more robust to failure or changes in network conditions. It can select transports that provide path-failover or multi-path communication (e.g., MPTCP, SCTP, SCTP-CMT). This allows a NEAT Flow to benefit without requiring specific application code to implement these functions. When the network provides support for signalling (e.g. to ease handover or notify a path failure), this can also be used by the NEAT Flow without needing to rewrite the application.

6.2 An example of improved application security

NEAT can help enable more predictable and available network security. Security choices are often linked to the protocol choice, but dependent on what is supported at the local and remote endpoints. By providing a richer API, the NEAT System enables better informed decisions within the transport

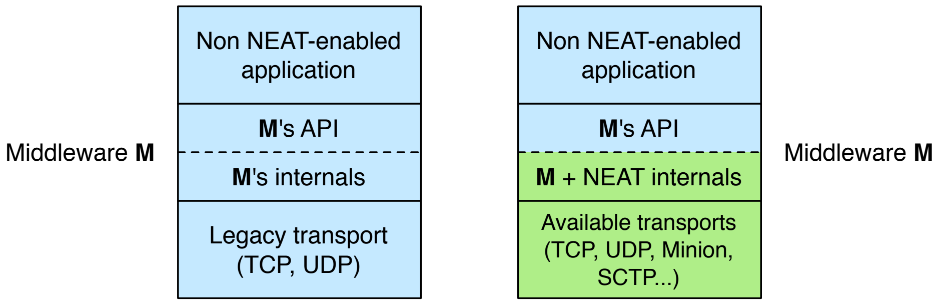
Middleware **M**
Non NEAT-enabled application
M's API
M's internals
Legacy transport (TCP, UDP)
Non NEAT-enabled application
M's API
M + NEAT internals
Available transports (TCP, UDP, Minion, SCTP...)
Middleware **M**

Figure 14: NEAT Application Support: Offering different abstractions of the API.

system (signalling whether security functions are desirable, required, etc). This can improve robustness of the network service and selection of mechanisms.

An application that uses the NEAT User API can benefit from the evolution of the transport system. It can take advantage of lower-layer components that provide security services. For instance an SCTP session can utilise DTLS by choosing to use a transport system that supports SCTP/DTLS/UDP when a UDP encapsulation is needed to communicate with an endpoint; when native SCTP becomes available, the NEAT System could choose to replace this with DTLS/SCTP.

In some cases, multiple methods may need to be tried in parallel to determine what security choices are actually available at the remote endpoint and whether the current path supports these choices. Moving such decisions to the transport system can enable the policy used to select transport components to become explicit, and can enable decisions to be audited (e.g., via the NEAT Flow endpoint statistics and diagnostics). Knowledge of how decisions are reached and reproducible selection can help reduce the chance of unexpected choices (e.g., avoiding a down-grade attack).

6.3 An example of an application using middleware

An application that uses a middleware library to access the network can take immediate and effective use of NEAT functions by using a middleware library that has been updated to work with the NEAT User API. In most cases we do not envisage such an application needs to be redesigned, nor necessarily recompiled since its interface to the middleware library is expected to remain unchanged.

Figure 14 shows an application using middleware (i.e, Class-2 Application) over the classic Socket interface (left) and after changing the middleware to use NEAT (right). Nothing changes in the application and the middleware API that the Application uses.

Some applications or middleware (Class 3) can benefit from a more abstract or different API provided by the NEAT Applications Support module. This provides an alternate way to access the NEAT User API. Functions or examples of the NEAT Applications Support module provide more abstraction and can ease programming for application developers unfamiliar with the NEAT User API.

By utilising the NEAT User API within the middleware, the application can indirectly and immediately benefit from using the NEAT System. This allows the middleware to express information about how it plans to use the network, and therefore allows the NEAT User Module to choose suitable transport parameters reflecting the network path it expects to experience (based on configured information in the PIB, and collected information in the CIB).

The application has the further opportunity to take advantage of new NEAT components as they become available. Some of the opportunities are one-sided, such as ability to enable differentiated services, and some are two-sided, where the remote endpoint also has to support the component, such as choosing to using SCTP-PR in place of TCP for message transmission. By enabling NEAT in the middleware library, any application that runs using this middleware over the NEAT System will start to gain immediate advantage.

6.4 An example of NEAT with a proxy

A NEAT proxy may be designed to provide a transition mechanism to enable immediate access to NEAT features in equipment that has not yet been updated to enable NEAT support. Client systems and applications could gain some of the key benefits of the NEAT System by communicating via the NEAT proxy, enabling NEAT features over the network path between the proxy and the intended server.

A NEAT proxy is a network device that implements two network interfaces: one facing the client and one facing the server. The first interface is a traditional network interface and the second interface uses the NEAT System. Connections and data are relayed between the two interfaces. A configured policy would allow the NEAT System to infer application requirements and use these to inform the configuration of the NEAT transport system via the NEAT User API.

Clients that are running on a NEAT-enabled system can also use a proxy to assist in optimise performance between the client and the proxy. Performance benefits are expected for applications using a network path with challenging characteristics (such as utilising a wireless/mobile link). In such a usage, NEAT selection decisions may take advantage of active and passive measurements to create entries in the characteristics information base (CIB) to reflect the current network path properties experienced by the proxy. This will help inform selection of appropriate transport components within the NEAT System (e.g., choosing a transport protocol that improves resilience to link failures or robustness to packet loss, or one that can particularly benefit an application in a mobile environment). It can also enable evolution of the transport system by addition of experimental mechanisms and extended functionality not supported at the remote server (e.g., new transport components and protocols tailored to a particular use).

7 Realising the NEAT Architecture

We expect the success of the new architecture to largely depend on our ability to show significant benefits for applications. This, in turn, depends on the availability of the transport protocols that the NEAT User API may use (i.e., the core transport code must already exist and be available). To assist in demonstrating the architecture, the project will provide new code that implements this architecture and can be used for trials and experimentation in testbeds.

This section outlines the testbeds that can be used, the applications that may be evaluated in the testbeds and the opportunities for standardisation of the proposed architecture. The success of standardisation is expected to be in part dictated by the ability of the testbeds to show the true advantages of the new approach.

7.1 Evaluation in testbeds

Trials are planned to evaluate the design and performance of the NEAT System, both over the public Internet and in testbeds brought into the project by its partners.

A testbed offers a controlled experimentation platform where solutions can be deployed and tested in an environment that resembles real-world conditions. Testbeds explore untested technologies or existing technologies working together in an untested manner. Testbeds generate requirements and priorities for standards organizations, and culminate in new (potentially disruptive) products and services. The NEAT use cases present a wide range of variety and scale. This section has identified the need for a right environment to test such large and complex use cases using several testbeds shared among partners.

Experiments to demonstrate the capabilities of the NEAT System will be performed in WP4, leveraging some of the following facilities:

- The H2020 MONROE project[3] (Measuring Mobile Broadband Networks in Europe) builds and maintains a testbed of several hundred multi-network routers that can be used to test new protocols and algorithms in realistic scenarios (SRL, Celerway, KaU). There will be available routers in Norway, Sweden, Italy and Spain, both in urban and rural areas, in stationary locations and on busses and trains.
- Cisco's Advanced Scenario Testing (AST) lab for interactive applications.
- An Internet testbed connected to the JANET academic network (UoA) supporting full routed and bridged multicast and unicast services for IPv4 and IPv6, utilising HP, Cisco and Linux-based equipment.
- The INFINITE testbed (EMC): a cloud and analytic infrastructure spanning three geographically diverse Data Centres in Cork, Ireland, interconnected via a full-mesh dark fibre network and combined with the Vodafone mobile M2M (Machine to Machine) network in Ireland. It is a horizontal testbed, operating as a service, providing highly scalable bandwidth and capacity that would facilitate an extensive scope of advanced development and use case scenario testing.

Work Package 4 of the NEAT Project will select and define the plan for experiment design, implementation and validation in the testbeds to be used in the project.

7.2 Identification of applications to be evaluated

The use cases (Section 3) were used to inform the requirements for designing the architecture of the NEAT System (Section 5). This allowed the project to identify a set of key application types that can be used to assess and demonstrate the benefits the NEAT System can bring to networked applications. The most appropriate testing environments for their evaluation are also identified. The selection of the actual applications to be evaluated will be made in WP4. Selected applications will be ported to use the most appropriate NEAT API (instead of the Socket API) and evaluated according to the test plan that will be developed in WP4. The following application types have been identified as possible candidates:

- **Mobile browser application (Mozilla's Firefox for Android):** This application will be used to evaluate and demonstrate the ability of the NEAT System to deliver a faster, more secure and unified Web experience to mobile users with diverse characteristics. It will mostly leverage the core functionalities of the NEAT System, such as discovery of end-to-end capabilities and transparent transport and network selection. Both one-sided and two-sided deployment of the NEAT System can be explored. For the latter, a NEAT-enabled web server will also be needed. Mozilla's Firefox for Android pre-release channel can offer a large and geographically wide base of users

[3]https://www.monroe-project.eu

for testing one-sided deployment, while a more controlled environment will be needed for testing two-sided deployment (e.g., MONROE platform).

- **Video/audio applications using NEAT:** Streaming and peer-to-peer conference applications will be considered to evaluate and demonstrate the ability of the NEAT System to provide better-quality and more robust operation over heterogeneous Internet paths. Available testbeds include mobile nodes with multiple interfaces (e.g., WiFi, 3G/4G, etc.), as provided in the MONROE platform available at SRL, Celerway and KaU, and the wired testbed available at UoA. These enable evaluation of the expected benefits from using extended NEAT components to support applications (e.g., network signalling, firewall/NAT traversal mechanisms and QoS-aware queuing and scheduling).
- **Data centers storage and workload transferring application suite:** This application suite will be used to evaluate and demonstrate the ability of the NEAT System to efficiently support different types of transfers between data centers and over virtualized network infrastructures, as described in § 3.4. It will leverage NEAT's ability to be extended to incorporate interactions with third party entities (such as an SDN controller) that can further improve the selection decisions made by a NEAT System. EMC's INFINITE testbed is the most appropriate testbed for evaluating this application suite.

Task 4.2 of the NEAT Project will select traffic and tests to be performed in the project.

7.3 Standardisation of the NEAT Architecture

The NEAT project is defining and implementing an architecture for use in the general Internet. As such, standardisation of the NEAT approach can be an important outcome to promote deployment and adoption of the developed architecture.

The work on the NEAT architecture relates to one of the current standardisation efforts of the IETF, where the Transport Services (TAPS) Working Group[4] is developing a series of documents that specify a new network programming interface oriented on a service model. It seeks to eliminate the binding between applications and a transport protocol, thereby enabling greater flexibility under the transport API and relieving application programmers of the load of having to implement their own transport protocol over UDP.

The TAPS standardisation effort is expected to focus on the API of the architecture—where results from NEAT should naturally flow into TAPS. The Working Group also provides a point of coordination with IETF initiatives relating to several key protocols within the architecture. NEAT therefore has a work package (WP5) that plans to continue engagement with the TAPS WG and other relevant IETF working groups to ensure the best possible prospect of standardisation.

8 Conclusions

The NEAT System defines a new architecture that changes the transport layer interface exposed to Internet applications. By allowing applications to provide information that can allow the stack to select the properties of the required service, this enables the stack to automatically choose an appropriate protocol. This seemingly simple change in a NEAT System can have massive ramifications, because

[4] http://datatracker.ietf.org/wg/taps/charter/

it allows flexible usage of a range of protocol components underneath the new user interface. This can enable the best possible use of the protocols/services that are available end-to-end along a given network path.

This document outlines the architecture of the NEAT System. A companion document (Deliverable D1.2) will explain how the NEAT User API is constructed and further documents will describe in detail the operation of the system, benefits that can be realised using the NEAT System and the opportunities it presents for enabling continued evolution. A key goal of the NEAT Project was to define a base architecture that is extensible. Future documents produced by the project will present the design choices and motivations for implementing representative functions within the architecture, chosen as a subset of the complete set of possible functions that the NEAT Architecture has been designed to support. This selection will be consistent with both the project goals specified in the Description of Work and the chosen main use cases.

Section 1 provides an introduction to the problem space and the NEAT approach, defining new terminology and the idea of modules and components in the architecture. This starts by examining the Berkeley Socket API, the de facto standard for applications communicating over the Internet, which was designed in the early/mid 1980's, and identifies the strengths and weaknesses of this design.

Section 2 provides the background and necessary theoretic and foundational work needed for the design of the NEAT System. This motivates the need for the architectural change to allow flexible usage of a range of technology beneath a new transport interface.

Section 3 defines a set of use cases that helped formulate the new approach and to identify where this new API may be expected to show benefit. These use cases provide one important input to establish a set of requirements for the new architecture ensuring that the set of features are appropriate to a wide range of applications.

Section 4 provides the requirements in terms of deployability, extensibility, flexibility, parameterisation and scalability. This sets the requirements for the new architecture and provide insight into the desired features of the NEAT System.

Section 5 defines the architecture. This describes the components of the system, relating these to implementation within a system, discussing how these components can be divided between user land and kernel implementations on platforms where there has been this traditional differentiation. It identifies two core modules, the User Module and the Application Support Module. The former is subdivided into five types of components, that provide a framework, selection and policy, and transport and signalling.

Section 6 provides specific examples of the way in which this NEAT System may be used by a range of applications. In the long term, we envisage the new approach will change the API used by all applications.

Section 7 discusses the approach to be taken to implementation of the NEAT System, and how the architecture can be used to help understand the utility of the new approach and allow exploration of whether these approaches can offer benefit to a specific application and/or the network.

References

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

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

[3] A. Bergkvist, D. Burnett, C. Jennings, and A. Narayanan, "WebRTC 1.0: Real-time communication between browsers," W3C Working Draft, Feb. 2015. [Online]. Available: http://www.w3.org/TR/webrtc/

[4] Y. Bernet, P. Ford, R. Yavatkar, F. Baker, L. Zhang, M. Speer, R. Braden, B. Davie, J. Wroclawski, and E. Felstaine, "A Framework for Integrated Services Operation over Diffserv Networks," RFC 2998 (Informational), Internet Engineering Task Force, Nov. 2000. [Online]. Available: http://www.ietf.org/rfc/rfc2998.txt

[5] A. Bittau, D. Boneh, M. Hamburg, M. Handley, D. Mazieres, and Q. Slack, "Cryptographic protection of TCP streams (tcpcrypt)," Internet Draft draft-bittau-tcp-crypt, Feb. 2014, work in progress. [Online]. Available: https://tools.ietf.org/html/draft-bittau-tcp-crypt-04

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

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

[8] B. Briscoe, "Inner Space for TCP options," Internet Draft draft-briscoe-tcpm-inner-space, Oct. 2014, work in progress. [Online]. Available: https://tools.ietf.org/html/draft-briscoe-tcpm-inner-space

[9] U. A. Camaró and J. Baudy. PACKET-MMAP. https://www.kernel.org/doc/Documentation/networking/packet_mmap.txt.

[10] S. Cheshire, J. Graessley, and R. McGuire, "Encapsulation of TCP and other transport protocols over UDP," Internet Draft draft-cheshire-tcp-over-udp, Jul. 2013, work in progress. [Online]. Available: https://tools.ietf.org/html/draft-cheshire-tcp-over-udp-00

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

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

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

[14] R. Davoli and M. Goldweber, "Msocket: Multiple stack support for the Berkeley socket API," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, ser. SAC '12. New York, NY, USA: ACM, 2012, pp. 588–593. [Online]. Available: http://doi.acm.org/10.1145/2245276.2245390

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

[16] 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

[17] J. Dike, *User mode Linux*. Prentice Hall, 2006. [Online]. Available: http://ptgmedia.pearsoncmg.com/images/9780131865051/downloads/013865056_Dike_book.pdf

[18] G. Fairhurst, B. Trammell, and M. Kuehlewind, "Services provided by IETF transport protocols and congestion control mechanisms," Internet Draft draft-ietf-taps-transports, Feb. 2015, work in progress. [Online]. Available: https://tools.ietf.org/html/draft-ietf-taps-transports-03.txt

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

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

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

[22] B. Ford and J. Iyengar, "Efficient cross-layer negotiation," in *Eighth ACM Workshop on Hot Topics in Networks (HotNets-VIII)*, New York City, Oct. 2009. [Online]. Available: http://conferences.sigcomm.org/hotnets/2009/papers/hotnets2009-final123.pdf

[23] 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

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

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

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

[27] T. Herbert, L. Yong, and O. Zia, "Generic UDP encapsulation," Internet Draft draft-ietf-nvo3-gue, Apr. 2015, work in progress. [Online]. Available: https://tools.ietf.org/html/draft-ietf-nvo3-gue-00

[28] B. Higgins, A. Reda, T. Alperovich, J. Flinn, T. Giuli, B. Noble, and D. Watson, "Intentional networking: Opportunistic exploitation of mobile network diversity," in *Proceedings of ACM MOBICOM*, 2010, pp. 73–84.

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

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

[31] IEEE/The Open Group, "The Single UNIX Specification," 2004. [Online]. Available: http://www.unix.org/what_is_unix/single_unix_specification.html

[32] "ZeroMQ," iMatix. [Online]. Available: http://www.zeromq.org/

[33] IRTF, "How Ossified is the Protocol Stack? (HOPS) Proposed Research Group Charter," https://datatracker.ietf.org/doc/charter-irtf-hopsrg/.

[34] J. Iyengar, S. Cheshire, and J. Graessley, "Minion - wire protocol," Internet Draft draft-iyengar-minion-protocol, Oct. 2013, work in progress. [Online]. Available: https://tools.ietf.org/html/draft-iyengar-minion-protocol-02

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

[36] M. Kuehlewind and B. Trammell, "SPUD use cases," Internet Draft draft-kuehlewind-spud-use-cases, Jul. 2015, work in progress. [Online]. Available: https://tools.ietf.org/html/draft-kuehlewind-spud-use-cases-00

[37] A. Langley, "Probing the viability of TCP extensions," 2008. [Online]. Available: http://www.imperialviolet.org/binary/ecntest.pdf

[38] S. Leffler, W. Joy, and R. Fabry, "4.2BSD networking implementation notes," Computer Systems Research Group, University of California, Berkeley, CA., U.S., Tech. Rep., Jul. 1983.

[39] J. Manner, N. Varis, and B. Briscoe, "Generic UDP Tunnelling (GUT)," Internet Draft draft-manner-tsvwg-gut, Jul. 2010, work in progress. [Online]. Available: https://tools.ietf.org/html/draft-manner-tsvwg-gut-02

[40] S. McQuistin and C. Perkins, "Reinterpreting the transport protocol stack to embrace ossification," in *IAB Workshop on Stack Evolution in a Middlebox Internet (SEMI)*, Zürich, Jan. 2015. [Online]. Available: https://www.iab.org/wp-content/IAB-uploads/2014/12/semi2015_perkins.pdf

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

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

[43] "Network stack in USerspacE (NUSE)." [Online]. Available: https://github.com/libos-nuse/linux-libos-tools

[44] A. Norberg, "uTorrent transport protocol," BitTorrent, Jan. 2009. [Online]. Available: http://www.bittorrent.org/beps/bep_0029.html

[45] E. Nordmark, S. Chakrabarti, and J. Laganier, "IPv6 Socket API for Source Address Selection," RFC 5014 (Informational), Internet Engineering Task Force, Sep. 2007. [Online]. Available: http://www.ietf.org/rfc/rfc5014.txt

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

[47] "CORBA," Object Management Group. [Online]. Available: http://www.corba.org/

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

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

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

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

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

[53] 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

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

[55] L. Rizzo, "Netmap," http://info.iet.unipi.it/~luigi/netmap/.

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

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

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

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

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

[61] J. Roskind, “QUIC: Multiplexed stream transport over UDP,” Google working design document, 2013. [Online]. Available: https://docs.google.com/document/d/1jdKEQMlM7ThDMDalFYFR_9-Yw91PhoBmkAPQcCicX3s/pub

[62] J. H. Saltzer, D. P. Reed, and D. D. Clark, “End-to-end arguments in system design,” *ACM Transactions in Computer Systems*, vol. 2, no. 4, pp. 277–288, Nov. 1984.

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

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

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

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

[67] W. Stevens, M. Thomas, E. Nordmark, and T. Jinmei, “Advanced Sockets Application Program Interface (API) for IPv6,” RFC 3542 (Informational), Internet Engineering Task Force, May 2003. [Online]. Available: http://www.ietf.org/rfc/rfc3542.txt

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

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

[70] H. Tazaki, R. Nakamura, and Y. Sekiya, “Library operating system with mainline Linux network stack,” in *Proceedings of netdev 0.1*, Ottawa, Feb. 2015. [Online]. Available: http://people.netfilter.org/pablo/netdev0.1/papers/Library-Operating-System-with-Mainline-Linux-Network-Stack.pdf

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

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

[73] J. Touch and W. Eddy, “TCP extended data offset option,” Internet Draft draft-ietf-tcpm-tcp-edo, Apr. 2015, work in progress. [Online]. Available: https://tools.ietf.org/html/draft-ietf-tcpm-tcp-edo-03

[74] J. Touch and T. Faber, “TCP SYN extended option space using an out-of-band segment,” Internet Draft draft-touch-tcpm-tcp-syn-ext-opt, Apr. 2015, work in progress. [Online]. Available: https://tools.ietf.org/html/draft-touch-tcpm-tcp-syn-ext-opt-02

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

[76] H. Trieu, J. Touch, and T. Faber, “Implementation of the TCP extended data offset option,” USC/ISI, Technical Report ISI-TR-696, Mar. 2015. [Online]. Available: ftp://ftp.isi.edu/isi-pubs/tr-696.pdf

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

[78] W3C. The WebSocket API (draft). [Online]. Available: http://dev.w3.org/html5/websockets/

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

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

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

[82] 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

[83] D. Wing, A. Yourtchenko, and P. Natarajan, “Happy eyeballs: Trending towards success (IPv6 and SCTP),” Internet Draft draft-wing-http-new-tech, Aug. 2010, work in progress. [Online]. Available: https://tools.ietf.org/html/draft-wing-http-new-tech-01

[84] J. Wroclawski, “The Use of RSVP with IETF Integrated Services,” RFC 2210 (Proposed Standard), Internet Engineering Task Force, Sep. 1997. [Online]. Available: http://www.ietf.org/rfc/rfc2210.txt