NEAT

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

H2020-ICT-05-2014
Project number: 644334

Deliverable D4.3

Validation and evaluation results

Editor(s): Zdravko Bozakov
Contributor(s): Anna Brunstrom, Dragana Damjanovic, Kristian Evensen, Gorry Fairhurst, Audun Fosselie Hansen, Fredrik Haugseth, David Hayes, Thomas Hirsch, Tom Jones, Naeem Khademi, Patrick McManus, Andreas Petlund, David Ros, Tomasz Rozensztrauch, Ricardo Santos, Daniel Stenberg, Michael Tüxen, Eric Vyncke, Hugo Wallenburg, Felix Weinrank, Michael Welzl

Work Package: 4 / Validation and evaluation
Revision: 1.0
Date: May 2, 2018
Deliverable type: R (Report)
Dissemination level: Public

Abstract

This document presents the main experiments carried out in WP4 for validation and evaluation of the NEAT System. Based on the test plan proposed in Deliverable D4.2 [11], the report provides a detailed overview of the test setups, equipment configurations, measurement methodologies and evaluations for each of the four industrial use cases developed in WP1. We demonstrate the feasibility of the developed NEAT approaches in realistic environments underlining the relevance of the designed solutions in the context of the industrial use cases, related to the partners' business needs. The experiments exercise key components of the core transport system designed in WP2, and highlight important research outcomes from WP3, related to the extended transport system and transport enhancements developed in the latter work package.

Furthermore, the document includes a discussion on the future of NEAT with an emphasis on NEAT's impact on scalability on a global scale as well as scalability aspects that relate to the end-host stack. Finally, the influence of the work carried out in NEAT on IETF standardisation efforts, in particular on a future standard transport API, is summarised.

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 **4**

1 Introduction **8**

2 Experimental results **8**
2.1 Celerway use case . . . 9
2.1.1 Test topology . . . 11
2.1.2 Test implementation . . . 11
2.1.3 Results . . . 13
2.1.4 Key findings and implications . . . 15
2.2 Cisco use case . . . 15
2.2.1 Test topology . . . 16
2.2.2 Test implementation . . . 21
2.2.3 Results . . . 22
2.2.4 Key findings and implications . . . 28
2.3 Mozilla use case . . . 29
2.3.1 Test topology . . . 29
2.3.2 Test implementation . . . 30
2.3.3 Results . . . 30
2.3.4 Key findings and implications . . . 36
2.4 EMC use case . . . 36
2.4.1 Test topology . . . 37
2.4.2 Test implementation . . . 37
2.4.3 Results . . . 38
2.4.4 Key findings and implications . . . 51

3 The future of NEAT **51**
3.1 Scalability . . . 52
3.1.1 Towards more aggressive protocols? . . . 52
3.1.2 Why is there a need for choice? . . . 53
3.1.3 Local scalability . . . 54
3.2 Evolution towards a standard API with implementation guidance . . . 60

4 Conclusions **62**

References **63**

A NEAT Terminology **67**

B SDN controller policies for handling elephant flows **70**

C How to build and test NEAT applications in MONROE **72**
C.1 Creating NEAT-enabled MONROE experiments . . . 72
C.2 MONROE metadata, Policy Manager and CIB . . . 75

List of abbreviations

AAA Authentication, Authorisation and Accounting

AAAA Authentication, Authorisation, Accounting and Auditing

API Application Programming Interface

BE Best Effort

BLEST Blocking Estimation-based MPTCP

CC Congestion Control

CCC Coupled Congestion Controller

CDG CAIA Delay Gradient

CIB Characteristics Information Base

CM Congestion Manager

DA-LBE Deadline Aware Less than Best Effort

DAPS Delay-Aware Packet Scheduling

DCCP Datagram Congestion Control Protocol

DNS Domain Name System

DNSSEC Domain Name System Security Extensions

DPI Deep Packet Inspection

DSCP Differentiated Services Code Point

DTLS Datagram Transport Layer Security

ECMP Equal Cost Multi-Path

EFCM Ensemble Flow Congestion Manager

ECN Explicit Congestion Notification

ENUM Electronic Telephone Number Mapping

E-TCP Ensemble-TCP

FEC Forward Error Correction

FLOWER Fuzzy Lower than Best Effort

FSE Flow State Exchange

FSN Fragments Sequence Number

GUE Generic UDP Encapsulation

H1 HTTP/1

H2 HTTP/2

HE Happy Eyeballs

HoLB Head of Line Blocking

HTTP HyperText Transfer Protocol

IAB Internet Architecture Board

ICE Internet Connectivity Establishment

ICMP Internet Control Message Protocol

IETF Internet Engineering Task Force

IF Interface

IGD-PCP Internet Gateway Device – Port Control Protocol

IoT Internet of Things

IP Internet Protocol

IRTF Internet Research Task Force

IW Initial Window

IW10 Initial Window of 10 segments

JSON JavaScript Object Notation

KPI Kernel Programming Interface

LAG Link Aggregation

LAN Local Area Network

LBE Less than Best Effort

LEDBAT Low Extra Delay Background Transport

LRF Lowest RTT First

MBB Mobile Broadband

MBC Model Based Control

MID Message Identifier

MIF Multiple Interfaces

MPTCP Multipath Transmission Control Protocol

MPT-BM Multipath Transport-Bufferbloat Mitigation

MTU Maximum Transmission Unit

NAT Network Address (and Port) Translation

NEAT New, Evolutive API and Transport-Layer Architecture

NIC Network Interface Card

NUM Network Utility Maximization

OF OpenFlow

OS Operating System

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

OVSDB Open vSwitch Database

PCP Port Control Protocol

PDU Protocol Data Unit

PHB Per-Hop Behaviour

PI Policy Interface

PIB Policy Information Base

PID Proportional-Integral-Differential

PLPMTUD Packetization Layer Path MTU Discovery

PLUS Path Layer UDP Substrate

PM Policy Manager

PMTU Path MTU

POSIX Portable Operating System Interface

PPID Payload Protocol Identifier

PRR Proportional Rate Reduction

PvD Provisioning Domain

QoS Quality of Service

QUIC Quick UDP Internet Connections

RACK Recent Acknowledgement

RFC Request for Comments

RSerPool Reliable Server Pooling

RTT Round Trip Time

RTP Real-time Protocol

RTSP Real-time Streaming Protocol

SCTP Stream Control Transmission Protocol

SCTP-CMT Stream Control Transmission Protocol – Concurrent Multipath Transport

SCTP-PF Stream Control Transmission Protocol – Potentially Failed

SCTP-PR Stream Control Transmission Protocol – Partial Reliability

SDN Software-Defined Networking

SDT Secure Datagram Transport

SIMD Single Instruction Multiple Data

SPUD Session Protocol for User Datagrams

SRTT Smoothed RTT

STTF Shortest Transfer Time First

SDP Session Description Protocol

SIP Session Initiation Protocol

SLA Service Level Agreement

SPUD Session Protocol for User Datagrams

STUN Simple Traversal of UDP through NATs

TCB Transmission Control Block

TCP Transmission Control Protocol

TCPINC TCP Increased Security

TLS Transport Layer Security

TSN Transmission Sequence Number

TTL Time to Live

TURN Traversal Using Relays around NAT

UDP User Datagram Protocol

UPnP Universal Plug and Play

URI Uniform Resource Identifier

VoIP Voice over IP

VM Virtual Machine

VPN Virtual Private Network

WAN Wide Area Network

WWAN Wireless Wide Area Network

1 Introduction

This document presents the main outcomes of WP4 activities carried out in months 24–38. The main scope of this report is on Task 4.3. The task focuses on the validation and evaluation of the experiments carried our as part of the industrial use cases developed and refined in WP1. These experimental activities build on previous WP4 work, i.e., porting selected existing applications to NEAT (reported in Deliverable D4.1 [12]), developing NEAT-enabled tools and outlining an initial plan for testing (first reported in D4.1 and updated in D4.2 [11]).

The motivation for the evaluations presented in this document is to demonstrate the feasibility of the NEAT concepts and software implementation in realistic environments with a clear mapping to use cases relevant to the core business of the involved industry partners. The industrial use cases and the related experiments highlight: (a) some key components of the core transport system developed in WP2 (e.g., Happy Eyeballs, basic Policy system), and (b) aspects from the research on the extended transport system and transport enhancements carried out as part of WP3. For the latter, the experiments demonstrate the use of NEAT's PvD integration, SDN datacenter integration through the extended NEAT Policy Manager in the datacenter, deadline-aware LBE file transfers over WANs as well as the use of NEAT proxies to optimise application performance in mobile broadband scenarios. Finally, the document provides a discussion on two important aspects related to the possible evolution and future of NEAT: scalability and broader architectural implications of NEAT results via new standards.

The rest of this document is structured as follows. In Section 2, we offer a detailed overview of the experimental work carried out as part of the selected industrial use-cases. For each use-case, a corresponding subsection provides a description of the test setup, equipment configurations, measurement methodology and an evaluation of the generated results. The document builds upon the test plan proposed in Deliverable D4.2 [11]. In Section 3 we discuss the evolution of NEAT with an emphasis on NEAT's impact on scalability on a global scale as well as scalability aspects that relate to the end-host stack. Moreover, we discuss the influence of the work carried out in NEAT on IETF standardisation efforts, in particular on a future standard transport API. Finally, Section 4 concludes the document.

2 Experimental results

This section provides a brief overview of the four industrial use cases and the test platforms used for their evaluation. In reference to the test plan proposed in D4.2 the subsequent sections summarize the test procedures and objectives for each use-case and provide key results and insights from the performed work.

In summary, four use cases were evaluated. The Celerway use case aims to demonstrate the benefits of using NEAT in a mobile broadband context, particularly when the mobile broadband network has NEAT support via Celerway proxies and routers. The Cisco use case uses the NEAT System to leverage discoverable network properties supplied by PvD to improve application QoS. The Mozilla use case looks at the use of NEAT's protocol selection mechanisms by a NEAT port of Firefox. The EMC use case evaluates the benefits of NEAT's SDN integration in the context of using MPTCP to improve flow completion times in datacenter environments. Furthermore, the benefits of deadline-aware, less-than-best effort (DA-LBE) mechanisms for WAN file transfers is evaluated.

The project partners are operating three testbeds to evaluate the NEAT library, NEAT mechanisms and industrial use cases defined in WP1. The testbeds offer a controlled experimentation platform where solutions can be deployed and tested in an environment that resembles real-world conditions. Table 1 provides a summary of the industrial use cases and the environments in which these were evaluated. Tests executed in these testbeds are augmented with results generated by experiments carried out on public Internet paths and/or lab setups, where appropriate. Applications ported to NEAT that were used for use-case testing were reported in Deliverable D4.2 [11], and are also referenced in the subsequent use case sections.

Table 1: Relevant test environments and applications/tools for testing the industrial use cases.

Use case	**Test environment***	**Most relevant applications and/or tools†**
Celerway	MONROE testbed	Multi-homed download manager PM diagnostics
Cisco	UoA Internet testbed	NEAT-streamer PM diagnostics
Mozilla	Lab setup	Firefox nghttp2
EMC	INFINITE testbed	Rsync PM diagnostics

† NEAT-streamer and the NEAT ports of Firefox and Rsync are presented in detail in Deliverable D4.1 [12].

* A detailed description of the testbeds is provided in Deliverable D4.2 [11].

2.1 Celerway use case

The Celerway use case is evaluated on the MONROE platform (detailed in deliverable D4.2 [11]) and aims to show the benefits of using NEAT in a mobile broadband (MBB) context, particularly when the mobile broadband network has NEAT support via Celerway proxies and routers.

The Celerway use case includes NEAT components as illustrated in Figure 1. This involves populating the PIB with relevant policies (step 0.1). Furthermore, the use case includes algorithms for collecting interface and networks statistics as metadata and active and passive measurements. This involves CIB sources that populate the CIB (step 0.2). Next, application requirements are learned via the NEAT User API (step 1) on a client device, or alternatively via signalling or by inferring application needs on network elements like router and proxy.

Next, the NEAT Framework (step 2) will through the NEAT Policy Interface (step 3) initiate the NEAT Policy Manager (PM) (step 4). The PM matches policies (PIB) and characteristics (CIB) with application needs (step 5) to make a selection of interface (step 6). This decision may be based on probing information and cached in the CIB. Next the NEAT transport component (step 7) connects to the host (step 8).

In order to test the Celerway use case, we have developed a set of CIBs and PIBs focusing particularly on mobile broadband, developed two applications using NEAT, and implemented NEAT in the H2020 MONROE platform as described in deliverable D4.2 [11].

NEAT in MONROE: A MONROE node runs the same software as a Celerway router, and it can connect to three mobile broadband networks, WiFi and Ethernet simultaneously. A MONROE node can

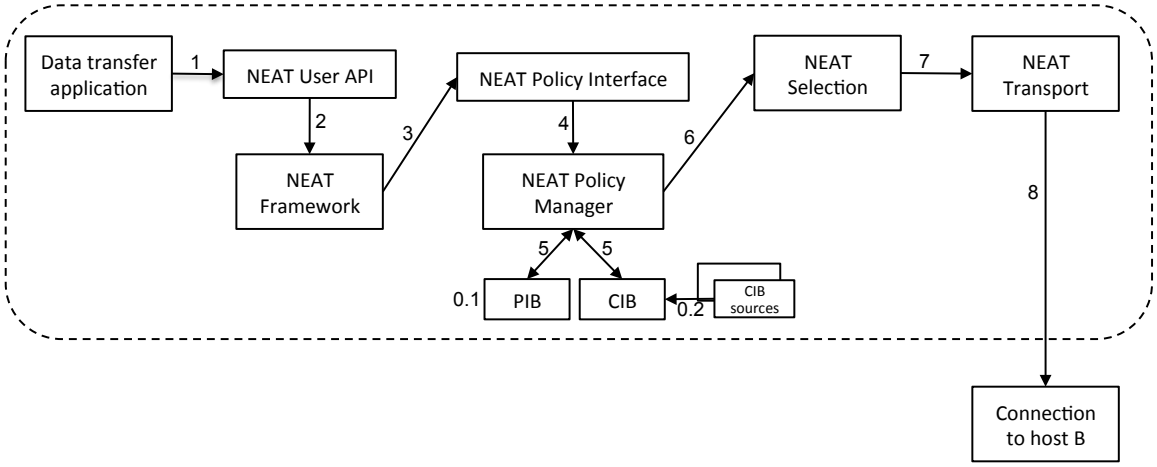


Figure 1: Interaction of NEAT Components in Celerway's specific use case.

act either as a client supporting NEAT-enabled applications, or as a proxy supporting non-NEAT applications. A detailed description of how to implement a NEAT application in MONROE is presented in Appendix C. Implementation of Celerway's use case includes the following elements:

- Deployment of the NEAT System on MONROE nodes. This also makes NEAT available to MONROE users so that they can plan and build experiments based on the NEAT architecture.
- Extension to MONROE's metadata exporting mechanism to export metadata to the NEAT PM (CIB). The Metadata exporter is one of the key components of the MONROE architecture. It collects information about available mobile networks and their properties and makes it available to other components. The Metadata exporter is designed to be easily extended in order to support new formats and new data recipients. In order to satisfy NEAT requirements, Celerway has built an extension that exports the data to the PM via a Unix domain. The PM then stores the data in a CIB.

  In addition, network quality estimates are exported. The NEAT PM is notified immediately upon every detected change in network properties and quality.
- Design and implementation of experiments to be run on a set of MONROE nodes. An *experiment* in MONROE terminology is an application that runs in an isolated environment (Docker container) and has access to selected network interfaces and related metadata. NEAT applications and non-NEAT applications can be scheduled on a selected set of MONROE nodes for an agreed period of time. Collected results are then available to the experimenter for analysis.

Celerway CIBs and PIBs: We have developed a **CIB source** and set of policies that collect and use information about network performance and metadata to make optimal interface selections. The CIB is populated with mobile broadband metadata for every available interface (table 2). The CIB is also populated with quality indications (0, 1 or 2, where 2 is the best network quality). This quality indicator is a result of a combination of passive and active measurements in Celerway routers.

Based on the CIB entries, we have specified policies in terms of scores that are stored as **PIB**. The interface with the highest total score — computed as the sum of the individual scores for each metadata value — is used by NEAT. Table 3 shows the scores for different metadata values and Celerway

Table 2: CIB mobile broadband properties.

Property	**Possible values**
Technology	mode, submode (LTE, 3G, 2G)
Signal Quality	For 2G, 3G: rssi, rscp, ecio
	For LTE: rssi, rsrp, rsrq
Cell location	lac, cid
Network operator	oper (MCCMNC code)
Device state	device_state, ipaddr

quality indicators for modems. For instance, a modem with a quality indicator 2 on 4G, frequency = 800, RSRP = -87, RSRQ = -4 would in total get a score of 64 (= 30 + 20 + 3 + 3 + 8).

Table 3: Scores for different modem metadata values and the Celerway quality indicator for modems.

Quality indicator	0	1	2
Score	0	5	30
Mode	2G	3G	4G
Score	1	10	20
Frequency	1800	800	2600
Score	1	3	7
LTE RSRP	-140 to -112	-111 to -84	-84 to 0
Score	0	3	8
LTE RSRQ	-20 to -11	-10 to -6	-5 to 0
Score	0	3	8
RSSI (3G and 2G)	-128 to -95	-94 to -70	-69 to 0
Score	0	5	15

2.1.1 Test topology

Figure 2 depicts the topology that served as the basis for the experiments in Test 2. The test setup was comprised of the key components listed in Table 4.

In addition, we have used a local lab setup to measure the overhead of using NEAT and the NEAT proxy (Test 1). This local setup consisted of one MONROE node as client and one MONROE node as server with files to download and a TCP-Ping responder. They were connected with a 1Gbit/s Ethernet cable.

2.1.2 Test implementation

Table 5 summarizes the experiments carried out as part of this use case. Specifically these tests were as follows:

- Test 1 used the Download manager described in D4.2 [11] and TCP-Ping (with and without NEAT) running in the local setup described above. In this case, both applications are NEAT applications

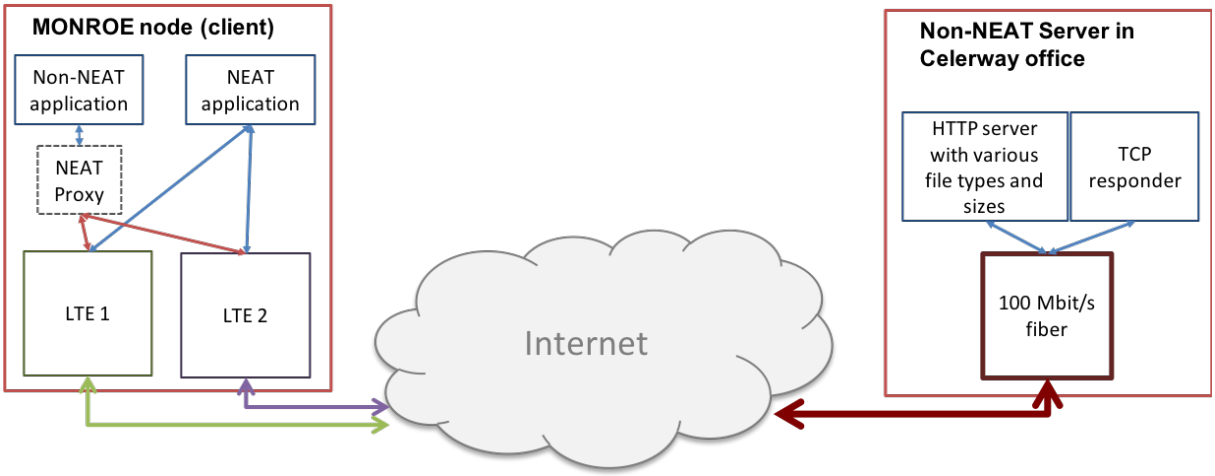


Figure 2: Test topology for the Celerway use case. Client nodes are actually deployed in the MONROE testbed, on-board high-speed trains.

Table 4: Components of Celerway's use case testing environment.

Component	Description
MONROE node as client	Described in detail in deliverable D4.2 [11].
Non-NEAT applications	Applications that are not using NEAT: Non-NEAT enabled Download manager and TCP-Ping
NEAT applications	Applications that are using NEAT: NEAT-enabled Download manager and TCP-Ping.
NEAT proxy	A proxy that fetches the non-NEAT traffic, infers needs and gives NEAT behaviour.
LTE 1	A cat6 Sierra Wireless MC7455 modem connected to operator 1 (different in different countries).
LTE 2	A cat6 Sierra Wireless MC7455 modem connected to operator 2 (different in different countries).
Non-NEAT server	An Intel NUC placed in Celerway's office with an HTTP server, TCP-Ping responder and 100 Mbit/s fiber link.

(i.e., Class-1 applications as described in D1.1 [17]) and non-NEAT applications (i.e., Class-0 applications as described in D1.1 [17]) using the proxy. They use CIBs and PIBs as described above. The main metric is overhead of using NEAT in terms of CPU and memory usage.

- Test 2 used the Download manager described in deliverable D4.2 [11] and TCP-Ping (with and without NEAT) running in the MONROE testbed described above. In this case, both applications are NEAT applications (i.e., Class-1 applications) and non-NEAT applications ((i.e., Class-0 applications)). They use CIBs and PIBs as described above. The main metric is quality in terms of throughput and RTT.

Table 5: Experiments for Celerway's use case testing.

Test ID	Summary
1	Evaluate the overhead of NEAT and NEAT proxy on system resources
2	Evaluate the impact of NEAT CIBs and PIBs on NEAT applications in multi-homed mobile scenarios

2.1.3 Results

Test 1 measured the overhead of using NEAT and the NEAT proxy in a local setup. We present the results from using the Download manager as it introduced the largest overhead. Table 6 shows that the policy manager adds some memory usage and that the proxy adds CPU usage. These numbers are not critical for normal devices, but for smaller embedded devices the code might need some optimisations.

Table 6: Overhead of using NEAT with CIBs, PIBs and proxy.

	Application (Process)	CPU (%)	Memory (MB)
Non-NEAT	Download Manager	13.8	1.9
NEAT	Download Manager	18.07	3.0
	Policy Manager	0.89	21.6
	CIB source	0.01	8.3
NEAT Proxy	Proxy	10.88	3.0
	Download Manager	11.89	0.8
	Policy Manager	1.48	21.4
	CIB source	0.01	7.8

Test 2 measured the potential of NEAT with regards to quality improvement in multi-homed devices. We ran multiple two-hour tests on mobile nodes with two active modems in MONROE. The nodes were installed on high-speed trains. For each test we ran a sequence of TCP-Ping (each second) for one minute and next fifteen seconds of download, both with NEAT and without NEAT. Measurements were run at different times (to capture rush hour vs. non rush hour and rural vs. urban). We also changed the policies to demonstrate the effect of those. Note that the policies were *not* fine-tuned for these experiments; the purpose of these tests is just to demonstrate the potential of NEAT to achieve better application performance.

Table 7 shows the average performance gains of using NEAT. *Average gain* means the percentage decrease in RTT and percentage increase in throughput, for TCP Ping and Download respectively, compared to the non-NEAT case. The experiments used some different policies just to get an indication of how that might affect results.

- **Policy 1** uses all values from Table 3. In addition, the default interface for use without NEAT is the interface that first gets an Internet connection on node boot.

- **Policy 2** uses all values from Table 3 except scoring Frequencies. In addition, the default interface for use without NEAT is the first registered interface after a network outage (not only node boot).
- **Policy 3** uses the same values and interface as Policy 2, but scores for signal quality were changed (they are shown in Table 8).

The results show that there is (as expected) variance in the results. However, they clearly demonstrate the potential of NEAT.

Table 7: NEAT performance gains. *No. of nodes* means the number of nodes that were available and ran the experiments. *Average gain* means the percentage decrease in RTT and percentage increase in throughput, for TCP Ping and Download respectively compared to the non-NEAT case.

Start time	Policy	No. of nodes	Average gain TCP-Ping	Average gain Downloads
2018-03-15 13:36	1	36	5.7	21.7
2018-03-16 15:38	1	28	11.5	38.7
2018-03-20 12:10	2	26	–10.4	4.1
2018-03-20 20:13	2	15	17.1	1.8
2018-03-21 20:13	2	17	18.9	8.0
2018-03-21 14:10	2	20	1.3	9.9
2018-03-20 15:01	2	26	20.6	4.6
2018-03-21 18:09	2	5	19.9	46.7
2018-03-22 15:00	2	21	15.1	–0.2
2018-03-26 13:51	3	24	–4.2	11.0
2018-03-26 15:54	3	21	–10.6	8.9
2018-03-26 19:10	3	16	11.8	7.7
2018-03-26 21:13	3	15	–11.4	–4.3
2018-03-27 07:11	3	21	23.3	–2.0
2018-03-27 09:14	3	16	23.2	3.7

Table 8: Scores for signal quality in Policy 3.

LTE RSRP	-140 to -112	-111 to -103	-102 to 0
Score	0	10	10
LTE RSRQ	-20 to -12	-11 to -8	-7 to 0
Score	0	5	10
RSSI (3G and 2G)	-128 to -95	-94 to -70	-69 to 0
Score	0	15	20

Further improvements The experiments and results presented above focus on interface selection for single path transfers, and did not attempt to find optimal policies for every situation. NEAT could also support multi-path transfers for instance with SCTP, which will be explored in more detail by

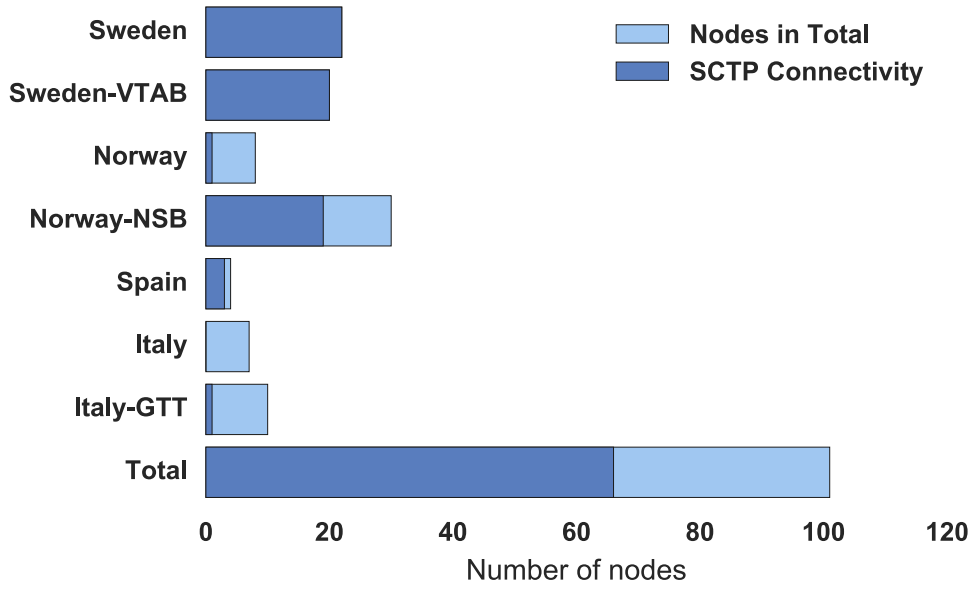


Figure 3: Support for the SCTP transport protocol in MONROE.

Celerway in the future. Figure 3 shows the support for SCTP in MONROE. The potential of using SCTP is well over 60% in the whole testbed, yet far from 100%. Thus, even if some mobile networks provide adequate SCTP support, without support from NEAT the application would have to settle with TCP at all times to secure connectivity.

2.1.4 Key findings and implications

The results presented above demonstrate the potential of using NEAT in multi-homed devices. The policies have not been fine-tuned, but they serve as examples on how to use NEAT. We also saw that NEAT adds some CPU and memory usage overhead that is not critical for normal devices, but for smaller embedded devices, the code might need some optimisations.

2.2 Cisco use case

The Cisco use case is evaluated on the UoA Internet Testbed and in a Cisco Virtual Testlab (§ 2.2.1) and aims to show the benefits of using NEAT in environments with network signalling. This requires network paths to offer services to match application requirements for multimedia traffic. For example, a two-way, live video connection can benefit from the ability to make decisions based on the properties of the local network environment via measurements and/or network signalling.

The NEAT System has been designed to enable straightforward integration of network signals and to allow the stack to act upon them when making selection decisions. Network signals provide one way for the network to directly inform the NEAT stack about local network characteristics.

Provisioning Domains (PvDs) Provisioning Domains (PvD) [9] describe a set of consistent information that identifies the characteristics of the service offered at a specific IP address or interface. This can describe both measurable (e.g., latency, capacity) and non measurable (e.g., price per byte, captive portal) properties as well as network configuration information such as name servers and captive portal locations. Signalling PvD information from the network to an endpoint enables the endpoint

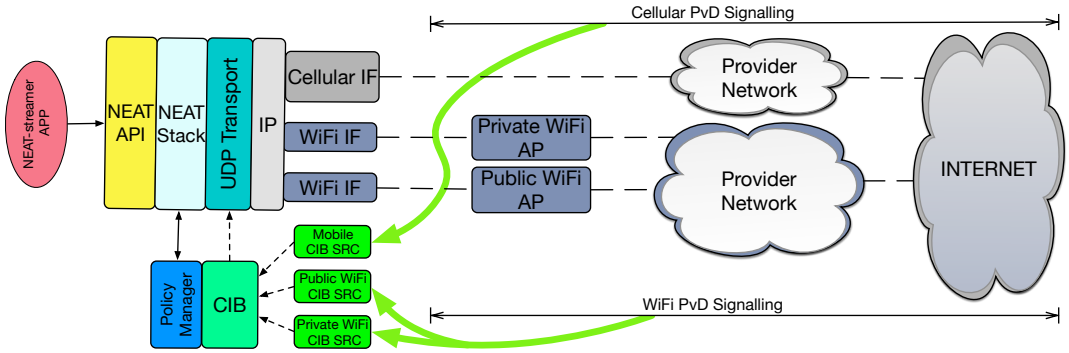


Figure 4: Example topology for the Cisco use case, showing multiple interfaces and their relation to the NEAT System. The set of available interfaces depends on the location within a deployment scenario.

to become aware of network characteristics. Once discovered, these characteristics can be used to influence or constrain selection decisions.

Multiple Provisioning Domains (MPvD) may be simultaneously supported on a single link creating a selection requirement for applications running on the host. This requires each application to become aware of each new function enabled and to date this has been difficult to deploy using existing protocol stacks. However, the approach taken in the NEAT System offers a significant change to overcome this barrier. In the NEAT Architecture, selection between multiple source addresses and interfaces can be performed by the stack on behalf of an application by evaluating requirements and preferences specified by the application. NEAT makes it practical for an application running in a MPvD environment to make the best use of the available source addresses, without each application needing to be updated with each new network function.

The test cases run one of two NEAT based applications to simulate workloads—either the NEAT-streamer multimedia test tool[1] or a HTTP test workload tool are used to evaluate how an application can select network interfaces based on properties. NEAT-streamer will attempt to establish an interactive video workload connection to a remote peer, reporting obtained throughput and latency.

Four workloads are used to evaluate how a NEAT-based application can select network interfaces based on properties. The user attempts to utilise all of the workloads, but success is constrained by the network capabilities available at each of the locations.

2.2.1 Test topology

The test topology (Figure 4) allows to evaluate a NEAT-enabled client connecting to different access networks with different configurations advertised on different interfaces.

A client uses the NEAT system (the left-most part of Figure 3). This has been updated to integrate PvD processing. PvD Router Advertisement (RA) messages were processed to access the signalling about each PvD and allowing this information to be used by the NEAT PM. The central part of the diagram represents the provider network(s), responsible for generating the RA PvD messages that advertise the capabilities of the provided networks.

The web server/remote host and remainder of the “Internet” is provided by the UoA testbed (the right-most part of Figure 3). Note although a web server, DNS server, routers etc. form a part of the testbed it was not necessary for these to implement NEAT to evaluate this use-case, since PvD is a

[1] See Deliverable D4.1 [12] for a more detailed presentation of NEAT-streamer.

Table 9: Components of Cisco's use case testing environment.

Component	Description
Application 1	The NEAT-streamer application (see [12]) has been written to evaluate multimedia workloads with this use case. The application should be able to take advantage of information signalled to the host about network configuration.
Application 2	A NEAT HTTP Client tool was used as the application to evaluate web style workloads. The application should be able to take advantage of information signalled to the host about network configuration.
Network	The network environment must be able to simulate several profiles of network configurations that have different network properties signalled. Both lab and live network environments are suitable places to evaluate the use of NEAT. The UoA Internet Testbed and the Cisco Virtual Testlab were used.

client-side feature. Multiple networks may be available to the NEAT System at any time. In NEAT, the decision about which interface to use is made at the time when a NEAT connection is setup. A PvD-enabled NEAT System can therefore take advantage of the signalled information to help inform network selection choices.

Evaluation testbeds The NEAT stack was run in the UoA Internet testbed and in a Virtual Machine based Cisco Virtual Testlab. Both were configured to emulate a set of different network environments simultaneously available, with configuration advertised over PvD.

The UoA Internet testbed for these use cases consisted of Raspberry Pi Single Board computers connected together via Ethernet using Multiple Provisioning Domains to support multi IPv6 source prefixes on a single network interface. The Raspberry Pi boards both ran a kernel that was updated to implement the PvD RA Option [35]. Use of the option could be enabled or disabled, depending on the test.

The client executes on a Raspberry Pi platform and uses a NEAT system updated to support PvD information. This includes adding a PvD daemon to accept RA messages from the provider network(s) and store this information within the NEAT CIB. Another Raspberry Pi is used in the provider network as the source of the PvD information. This generates PvD information for the RA messages and the same developed software could in the future be incorporated in the standard software of a provider router. When the provider router is not directly attached to the client, a proxy may be needed to relay the PvD information in the RA messages. This function was tested using a third Raspberry Pi placed between the NEAT client and the source of PvD information.

Cisco's Virtual Testlab[2] is a virtual machine image which uses network name spaces to implement the network test topology. Network name spaces give applications a per process view of a network configuration. This allows a single user account to launch processes in different network namespaces and for the applications to experience this as if they were running on machines with completely different configurations. The Cisco Virtual Testlab allowed development and evaluation to occur on a single machine instance.

Test environment The UoA Internet testbed provided a testing environment for this use case. This environment includes three main components listed in Table 9.

[2] https://github.com/IPv6-mPvD/pvd-dev

To understand how an application using the NEAT System can utilise PvD information we examine how a set of workloads each with specific requirements is used in four example locations. This is explored by considering the same set of locations in there different deployment scenarios:

1. **Deployment Scenario 1:** Single network interface. At each location, there is only one network interface available. PvD has not been enabled.
2. **Deployment Scenario 2:** Multiple network interfaces. At each location, two alternative network interfaces are available. PvD has not been enabled.
3. **Deployment Scenario 3:** Multiple networks and PvD enabled. At each location, two alternative network interfaces are available and PvD information is provided by each network.

Deployment scenario 3 can also be used to enable additional domains to be offered in PvD-enabled locations. The additional PvDs could, for instance, advertise various forms of "walled garden", such as a provisioning domain associated with the ISP's paid TV service that can be used to connect a set top box to an IPTV solution, or a local private network used to interconnect IoT devices for which there is no desired Internet connectivity, e.g., a house webcam. This is represented by the IoT workload in this use case.

Application workload Four application workloads are used to evaluate this use case, based on a single multimedia application and three different workloads using Application 2 (a NEAT HTTP Client).

1. **Audio Conference:**. NEAT-streamer is used to evaluate an audio conference workload, representing a conference call using a corporate network.
2. **Download:** An HTTP workload is used to download a large piece of multimedia content for later viewing.
3. **Web Browser:** A sequence of HTTP GET requests are made to browse web content via the Internet.
4. **IoT:** A sequence of RESTful web requests are made by an IoT application to update the application and exchange application data with a private web server. This server may be accessed from home ISPs, but using PvD's other providers (such as public access networks) may also enable this additional service.

Table 10: Application workloads.

	1	2	3	4
Description	Audio conference (Call to work)	Download (Download of a movie)	Web browser (View web content)	IoT (Update IoT application)
JSON request	Medium capacity Low latency	High capacity Low cost		Free cost
Legend:	Preferred	Mandatory		

Each application will signal preferences or requirements over capacity, latency and cost:

1. The capacity is the rate that can be sustained on a link, in megabits per second (Mbit/s).

2. The latency is the delay on a section of the link, in milliseconds (ms).

3. The cost is expressed in pence per megabyte (p/Mbyte).

NEAT will use the PvD information to match these requirements when performing interface selection. Table 10 shows the four applications we are going to look at in these scenarios and the requirements they will signal. Some of the parameters are optional.

Example usage scenarios To illustrate how the networks available at each location are used, we consider the workloads generated as a part of a typical employee's day, as the employee changes her location. Our typical employee uses a NEAT system that seeks to complete each of the four application workloads as soon as possible. The employee's workloads attempt to utilise the networks available at each of the four locations encountered over the course of a day.

1. At the start of the day, the employee goes to collect coffee at her local coffee shop. While drinking coffee, she decides to connect to the network to start her day's activities.

2. The employee sees the need to move to the office to conduct an audio conference call with the boss (limited only to the company network). This company network was not enabled to support the IoT device update, nor was it a suitable location for private web browsing.

3. At the end of the day, the employee calls-in for another drink at the coffee shop located in the foyer of the company. This coffee shop has access to some company networks as well as the general Internet.

4. Finally, having consumed her coffee, the employee heads home, where she is presented with yet another set of networks to choose from.

The full set of four scenarios was run through with three network configurations: a single network with no PvD, multiple networks with no PvD and multiple networks with PvD.

The NEAT System is able to choose between the offered networks using the provided PvD information. The NEAT System is expected to choose the appropriate available network to best satisfy the properties expressed by the four applications.

Tables 11,12 and 13 show each of the four applications and the *expected result* as our User progresses through the four locations. There are three possible results for each attempt to connect: success (green), fail (i.e., marginal progress on the connection, but unable to complete) (red) and not attempted (cyan). Each application continues to try to connect in each scenario until it succeeds, once it has succeeded it does not need to be tried again.

Table 11 shows the four application workloads being attempted when one network is available. In this case, a single WiFi interface is available with no PvD signalling. All four applications are attempted at the same time in each case. The limited capacity on the Free WiFi in the coffee shop is unable to handle the traffic generated by all of the applications. These public networks are only configured for public access, and do not support the IoT service.

Table 12 shows how the application workloads attempt to connect when there are multiple network interfaces. In this case, the workload first attempts to connect via WiFi and, if unsuccessful, attempts to use the mobile interface. PvD signalling has not been enabled and each network can provide only one service, and hence only the home ISP supports the IoT service.

Table 11: Deployment Scenario 1: Expected completion of application workloads with a single network interface (PvD has not been enabled).

		Coffee Shop	Work	Work Coffee Shop	Home
Audio Conference	WiFi	Fail	Success	Not Attempted	Not Attempted
Web Browser	WiFi	Fail	Success	Not Attempted	Not Attempted
Download	WiFi	Fail	Not Attempted	Fail	Success
IoT	WiFi	Fail	Fail	Fail	Success

Table 12: Deployment Scenario 2: Expected completion of application workloads with multiple network interfaces (PvD has not been enabled).

		Coffee Shop	Work	Work Coffee Shop	Home
Audio Conference	WiFi	Fail	Success	Not Attempted	Not Attempted
	Mobile	Fail	Fail	Fail	Fail
Web Browser	WiFi	Fail	Success	Not Attempted	Not Attempted
	Mobile	Fail	Not Attempted	Not Attempted	Not Attempted
Download	WiFi	Fail	Not Attempted	Fail	Success
	Mobile	Fail	Fail	Fail	Fail
IoT	WiFi	Fail	Fail	Fail	Success
	Mobile	Fail	Fail	Fail	Fail

Table 13 shows the attempts to connect when there are multiple network interfaces and there is PvD signalling information available on both the links. With PvD enabled the NEAT Stack is able to stop applications running when their requirements cannot be met (e.g., the Download is not run on the free Coffee Shop WiFi because it advertises less capacity than the application needs). Enabling PvD is expected to significantly reduce the number of attempts that will end in failure.

Table 13: Deployment Scenario 3: Expected completion of application workloads with multiple network interfaces and PvD Enabled.

		Coffee Shop	Work	Work Coffee Shop	Home
Audio Conference	WiFi	Not Attempted	Success	Not Attempted	Not Attempted
	Mobile	Not Attempted	Not Attempted	Not Attempted	Not Attempted
Web Browser	WiFi	Success	Not Attempted	Not Attempted	Not Attempted
	Mobile	Not Attempted	Not Attempted	Not Attempted	Not Attempted
Download	WiFi	Not Attempted	Not Attempted	Not Attempted	Success
	Mobile	Not Attempted	Not Attempted	Not Attempted	Not Attempted
IoT	WiFi	Not Attempted	Not Attempted	Not Attempted	Not Attempted
	Mobile	Success	Not Attempted	Not Attempted	Not Attempted

2.2.2 Test implementation

PvD in NEAT PvD support was integrated into the NEAT System by updating the NEAT Policy Manager to enable collection of the PvD information over HTTP and JSON, signalled using IPv6 Router Advertisements (RA). It also includes support to allow an application to use policies that can be based on the collected PvD information. Together this made it possible for an application to request a wide range of network characteristics (e.g., to use only networks that have no cost and to specify the network must be able to support specific features, such as low latency or support for a particular type of device).

The PvD information enters the NEAT Policy Manager and is stored as a Characteristic Information Base (CIB) source. A stand-alone interface was used that processes the PvD information received in RA messages and extracts the URL to contact the PvD server. From this interface PvD information can be retrieved in JSON format. The CIB source translates and deposits the JSON PvD information in the CIB, where it can then be utilised by the Policy Manager to make decisions. The Policy Manager also contains the appropriate machinery to ensure that expired information is removed from the CIB (i.e., control the way the information is cached).

An application can set properties to be used by the NEAT Policy Manager to match configured policies. When PvD is supported, a policy can depend upon the information about a PvD. For example, an application could request a mandatory property via the NEAT API, requiring that only prefixes with a PvD advertising this property would be selected — e.g., the company network could be required for a “conference application”. Whereas, when an application requests optional properties, NEAT would select the PvD based on a score system of how well each of the PvD interfaces satisfies the set of properties requested by the application. The result of the Policy Manager informs the selection components of the NEAT System and results in traffic being sent using the appropriate NEAT interface.

NEAT application platforms NEAT was evaluated on top of virtual infrastructure in Cisco's PvD testlab and was also evaluated on Raspberry Pi single board computers to evaluate the setup in a non-virtual environment.

Raspberry Pi single board computers were imaged with patched kernels to allow support for the PvD Router Advertisement Option. These tests were conducted on real hardware to get implementation experience in a real world network environment.

In both cases, NEAT applications were integrated with `pvdd`, a Provisioning Domains daemon that parses and handles Router Advertisements, developed by Cisco and released as open source[3]. The NEAT PM interacted with `pvdd` using the HTTP JSON API. This API demonstrated how PvD can be enabled for hosts which do not yet support the PvD RA option.

In each of the planned tests the NEAT stack was evaluated to validate the testbed and test topology. Validation ensured that the Characteristic Information Base was collecting information from `pvdd` and that the information was correct for the planned test.

Network configuration automation Tests were automated with a script in which the test topology was established and the set of test cases were run. The script was configured to run for each application in each deployment scenario and performed the following tasks:

- Configure network topology.
- Configure network characteristics.
- Configure `pvdd` to advertise characteristics.
- Launch PvD HTTP proxy.
- Launch an "other end" for the NEAT application to talk to.
- Run a NEAT application with JSON policy file.

To handle PvD RA processing in a single place, a host may run a daemon to process these messages and provide access to the signalled information. The `pvdd` daemon is used in this set of use cases. `pvdd` offers bindings to multiple languages, these use cases execute a nodejs application which proxies the PvD JSON over HTTPS.

2.2.3 Results

Validating that PvD information was being parsed and incorporated into the NEAT CIB was required to verify that the test environments were functioning. Listing 1 is an example of a CIB source generated from PvD signalled information.

Correct PvD properties appear in the NEAT CIB as device interface entries for the source address prefixed with the `pvd_` string to indicate that they have been signalled from the network. Once CIB entries have been picked up from the network they are then evaluated as if there are enhanced interface property sets.

```
{
    "description": "PvD CIB node for remote host 139.133.204.40",
    "expire": -1.0,
```

[3] https://github.com/IPv6-mPvD

```
"filename": "pvd_corporate_ext.erg.abdn.ac.uk.cib",
"link": false,
"priority": 0,
"properties": [
    [
        {
            "interface": {
                "precedence": 2,
                "value": "wlan0"
            },
            "local_interface": {
                "precedence": 2,
                "value": true
            }
        }
    ],
    [
        {
            "capacity":  {
                "precedence":  2,
                "value":  100000000
            },
            "cost":  {
                "precedence":  2,
                "value":  100
            },
            "domain_name": {
                "precedence": 2,
                "value": [
                    "pvd.erg.abdn.ac.uk",
                    "*"
                ]
            },
            "ip_version": {
                "precedence": 2,
                "value": 6
            },
            "latency": {
                "precedence": 2,
                "value": 10
            },
            "local_ip": {
                "precedence": 2,
                "value": "2001:2::2"
            },
```

```
                "transport": {
                    "precedence": 2,
                    "value": "TCP"
                }
            }
        ]
    ],
    "root": true,
    "uid": "pvd_corporate_ext.erg.abdn.ac.uk"
}
```


Listing 1: Generated CIB entry for a Corporate PvD from the UoA Testbed. This generated entry includes metadata for cost and capacity signalled from the network via PvD (shown in bold characters).

Scenario test results The tests considered the three deployment scenarios across each of the four locations using the set of application workloads. In each plot, a green box shows a successful connection and communication of the workload. A red box represents an application workload that failed (i.e., it connected to the server and started sending traffic, but did not complete the required workload). A white box shows a case that was not attempted (i.e., where PvD information avoided an attempt to connect and start a transfer that would not succeed).

The first deployment scenario has a single non-PvD enabled network interface. This provides a base line. The plot in Figure 5 shows application workload progress for each location.

The four applications attempt to run concurrently, even in locations where they have little or no chance of completion. Their attempts to connect can therefore interfere with each other, impeding each other's progress — and in some cases resulting in failure of connections that could have been successful, had they not been attempted in parallel. As an example, the download application workload creates a large volume of traffic when run on a network with limited capacity and prevents progress with any other concurrent application workload. A deployment scenario that presents multiple possible network interfaces compounds this interference between the different workloads. The burden of changing the network interface is shown by the long dependency chains through many networks while an application workload attempts continue to fail.

Figure 6 shows the effect of application workloads that attempt in turn to connect using the WiFi and 4G interfaces, while continuing to generate network traffic on networks that cannot pass this traffic.

Finally, the third deployment scenario shows the effect of using multiple networks that provide network signalling via PvD. This shows how this can enable the NEAT stack to make better selection decisions. With enough information the NEAT stack is able to not attempt connections on networks which are unable to fulfill application requirements.

In contrast to the previously presented deployment scenarios, Figure 7 shows that the use of PvD information effectively eliminates the impact of unproductive connection requests and avoids unproductive competing traffic from impacting the completion time of other application workloads. The result is faster completion of the set of application workloads.

Comparing the dependency plots for the three network configurations it becomes clear that incorporating PvD network signalling has benefits. Applications are able to complete much earlier in

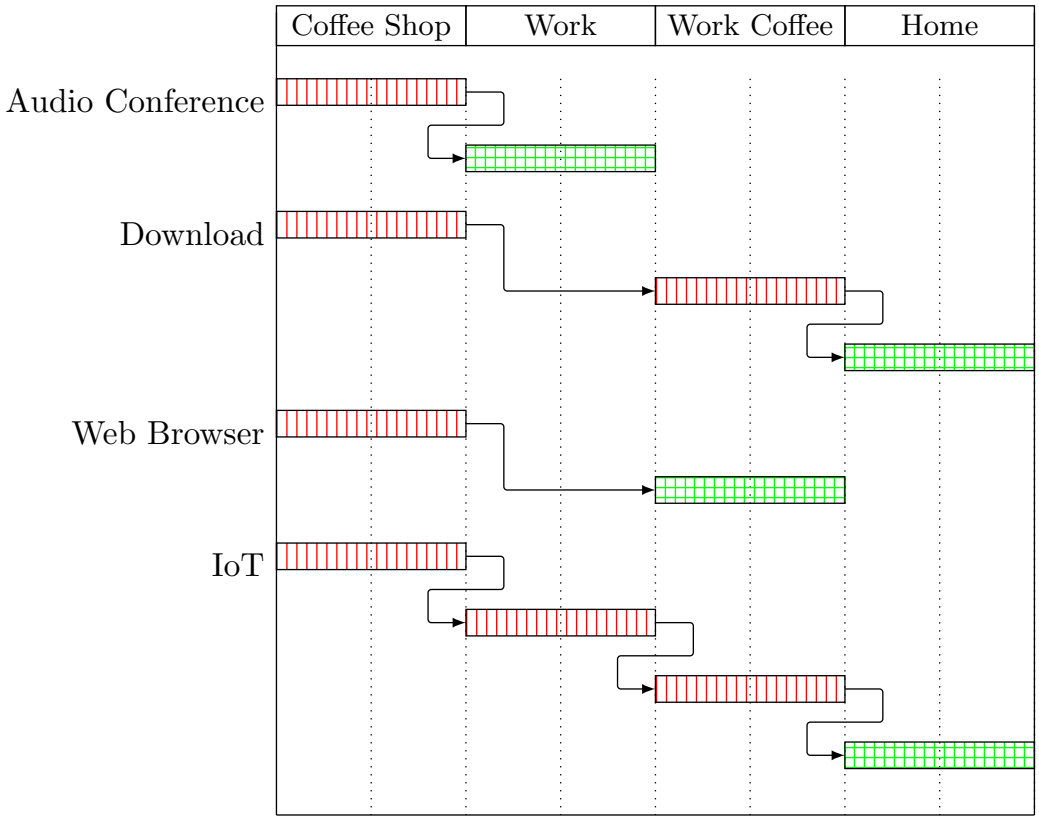


Figure 5: Dependency plot showing completion of the NEAT application workloads in each of the four locations for a deployment scenario with only a single network interface (PvD not enabled).

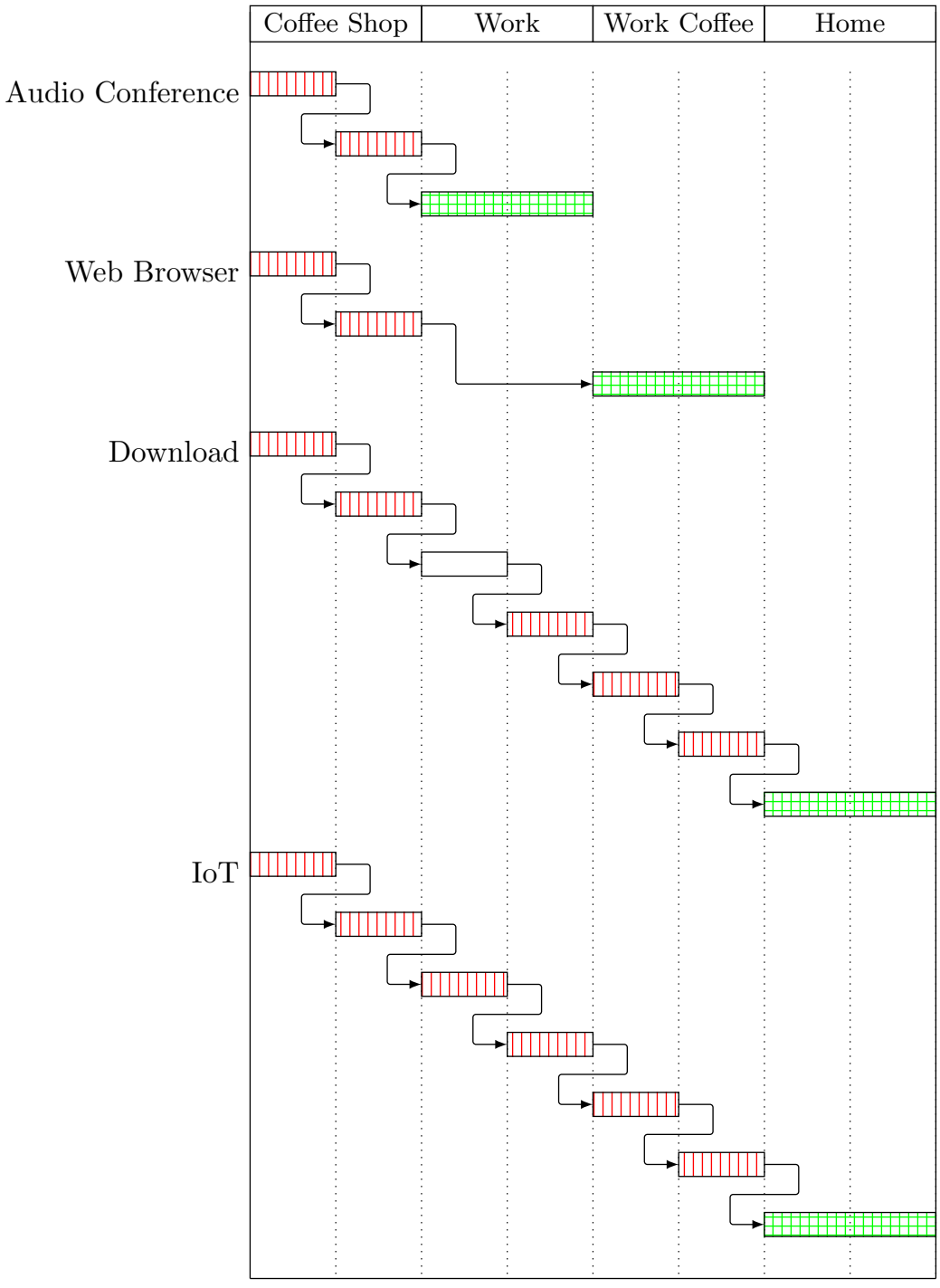


Figure 6: Dependency plot showing success of using each application in each of the four locations (PVD was not enabled). At each location the wifi and mobile network interfaces are shown respectively as the first and second columns under each of the locations.

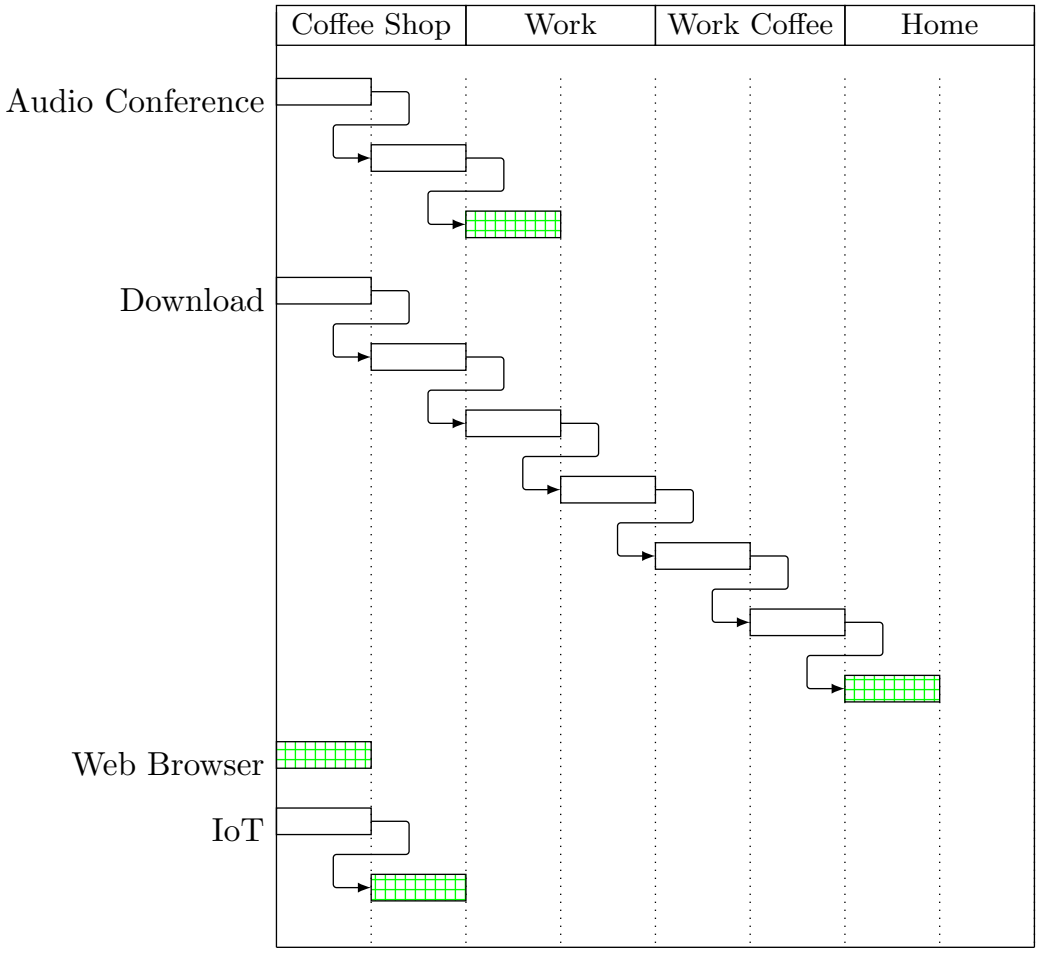


Figure 7: Dependency plot showing progress of the application workloads in each of the four locations for a deployment scenario with two network interfaces (PvD enabled).

the location when interference traffic is reduced. Failed attempts can be reduced or even removed on networks that cannot support the properties required by the application reducing the number of spurious bytes sent on the network. The Policy Manager HTTP interface allowed PvD support to be added without having to modify any NEAT based applications. The extensibility of the NEAT Architecture made it possible to integrate one signalling method, further methods could be trivially integrated through the same API.

2.2.4 Key findings and implications

This use case has explored the use of network layer signalling as a part of the interface/transport selection process in NEAT. Choosing an appropriate network interface to send data can have a very significant impact on the quality of experience for some applications (e.g., multimedia applications that desire a specific QoS).

The scenario added signalling between the network provider and the application stack to enable a more flexible choice of links/services. This can avoid needless probing/starting of network flows that are unable to sustain the required network service (e.g., have insufficient or no capacity to the required endpoint, would result in excessive cost, or are unable to meet the latency requirements of a specific application).

Implementation provided insight into the ways that network-layer signalling could be integrated in a NEAT System. In particular, the NEAT Architecture offers a coherent approach to handling network signals that enables caching of information (e.g., in the NEAT CIB), consistent logging and reporting of information (providing visibility of the input and outcomes of the policy framework), and the ability to manage how the information is used (based on policies provided by the system and/or applications) and validated (e.g., it is often important to not accept arbitrary inputs that can make a system vulnerable to Denial-of-Service attacks).

The use case utilised the new network-layer PvD mechanism to feed the NEAT Policy Manager with information about connected networks. The PvD mechanism was successfully integrated into the NEAT System. The components (e.g., pvdd) were demonstrated on multiple hardware platforms, although these tests only utilised components operating on the Pi platform. Furthermore, the developed software components were used in the UoA Internet Testbed to construct a demonstrator for the Cisco multimedia use case for NEAT. This provides a concrete example of how the PvD mechanism can be used within actual protocol stacks.

The structure of PvD information is extensible, and therefore once PvD signalling support is integrated, new policies could be developed that leverage any additional information provided. Since this information enters the CIB, any application may take advantage of the information learned about the network. Research into a range of other information types can thus build upon this work— for instance, signalling of the available DSCP mappings (to avoid inappropriately using a code point that is not locally supported), or the ability to utilise information about latency of the offered service (to help choose the appropriate interface for time-critical services).

The approach to integrate and cache PvD information in the CIB can also be used to collect and manage other information about the network path, enabling this to be used by transport protocols or selection mechanisms. For example, a transport system could provide a module that probes for and collects information about the maximum packet size that is supported by a path (i.e., Path MTU Discovery [18]). To do this requires integration of many different signals (probe success, ICMP messages,

interface parameters and information from explicit signalling). Like PvD information, the inputs need to be validated, decisions logged, and outcomes presented to the application in a consistent fashion.

The code components developed to support this use case within the NEAT Project continue to be used to promote and develop the specifications for PvD within the IETF INTAREA working group. While this standardisation work is directly applicable to the policy-based approach of NEAT, it can also be used with other systems that can utilise the information about the attached networks. The applicability therefore extends beyond the NEAT Project and is expected to become part of new products developed.

2.3 Mozilla use case

The Mozilla Firefox web browser is a sophisticated legacy application and runs on many hardware platforms. It natively implements many of the mechanisms offered by the NEAT library. Happy-Eyeballing between IPv4 and IPv6 and between different application layer protocols are just some of the overlap in functionality. Firefox using the NEAT library is a good feedback input for the NEAT System and Policy Manager as a whole, and serves also as an evaluation of the NEAT library and the NEAT API and their ability to support involved and complex applications such as Firefox.

The main expected result is to prove that a complex application, such as Firefox, can make use of NEAT's Happy-Eyeballs mechanisms and detect what network and transport protocols are supported along the path and available on the remote host (server). Such an existence proof gives credence to NEAT's ambition of enabling other applications to evolve the overall ecosystem through the introduction of new protocols and transports.

While doing this run-time detection the application performance, typically measured by page load times, should not be meaningfully degraded — Firefox is already a well-tuned application and an improved performance is not expected from NEAT per se. The aspect of page load being considered in these tests is connection setup overhead. Selection of different transport protocols would necessarily drive page load performance with their particular characteristics (e.g., multipath, or LEDBAT) after connection.

2.3.1 Test topology

The base topology used for experiments is shown in Figure 8. The three key components of this setup are described in Table 14.

Table 14: Components of Mozilla's use case testing environment.

Component	Description
Application	In this case the application is a Firefox distribution that uses the NEAT library (see Deliverable D4.1 [12]) on a host with an active Policy Manager.
Server	The web server must be able to serve content using IPv4 and IPv6. This will test the NEAT library's Happy-Eyeballs mechanism and additional components, such as the PM.
Network	It should be possible to control the test network and block certain protocols or add packet loss. For example, a scenario could include both a client and a server that support IPv6 and IPv4 protocols but the path is misconfigured and only allows IPv4 packets through.

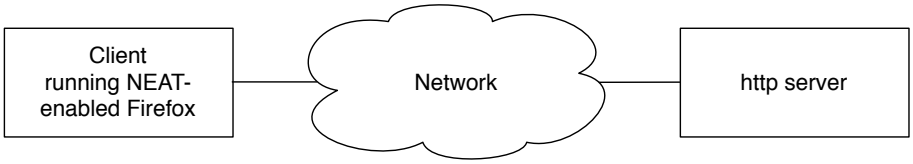


Figure 8: Base test topology for the Mozilla use case.

2.3.2 Test implementation

The experiments demonstrate how Firefox uses the NEAT System for IP version and transport protocol selection in order to favor more modern versions of each protocol. The NEAT port of Firefox disables its own native happy eyeballs support in favor of the NEAT connection management. Firefox uses NEAT to select the best protocol suite for use on the given network considering the capabilities of both the client and the server. This test will illustrate that an Internet application can rely on the NEAT System to select the best available protocol given the policies and circumstances it operates under.

The NEAT port of Firefox is using NEAT with preference to pick IPv6 over IPv4 and requests the use of SCTP if it is available.

The testing includes the scenarios listed in Table 15. Test 1 focuses on IP protocol selection, whereas Test 2 focuses on enabling use of SCTP transport on the client side. Test 2 also includes a test under lossy network conditions to validate the impact of happy eyeballs in the presence of lost packets.

The test network was the open Internet and the devices under test were separated, on average, by 50 ms of latency as measured by ICMP. Packet dropping for broken IPv6 emulation was implemented with Linux firewall rules where necessary on the client side.

The happy eyeballs configuration via the policy manager was written to give a 10 ms response advantage to SCTP (see Listing 2).

```
{
    "policy_type": "profile",
     [...]
    "properties":{
        "__he_delay": {
            "value": 10,
            "precedence": 1,
            "score": 0,
            "description": "special property defining HE delay in
               milliseconds"
       },
       [..]
    }
}
```

Listing 2: Policy Manager Happy Eyeballs Profile.

2.3.3 Results

The timings recorded in Test 1 were obtained from a libpcap packet capture made with tcpdump and analyzed with Wireshark.

Table 15: Experiments for Mozilla's use case testing.

Test ID	Summary
1a	The server supports both IPv4 and IPv6, but the network blocks IPv6
1b	The network supports both IPv4 and IPv6, but the server supports only IPv4
1c	Both the server and the network support IPv4 and IPv6
2	The server does not have SCTP available but the client should actively determine that under both normal and high loss conditions

Test 1a This test used an IPv6 capable server and a domain name that resolved to both v4 and v6 as well as a client that was enabled for both protocol versions. However, the network was configured to drop IPv6 TCP via a firewall rule.

Table 16: Test 1a — timeline of connection events at the client.

Timestamp (ms)	Event
0	Firefox, via NEAT, made DNS A and AAAA queries for server
42	Received the A record response
43	Received the AAAA response
58	TCP connection initiated to the IPv6 address
58	TCP connection initiated to the IPv4 address
103	IPv4 SYN-ACK. No further v6 packets were seen in the trace.

Repeating test 1a with the native Firefox created essentially the same result with the SYN-ACK being received at T=100ms.

Test 1b This test used an IPv6 capable server and a domain name that resolved to both v4 and v6, however, the server was only operating on the IPv4 address.

Table 17: Test 1b — timeline of connection events at the client.

Timestamp (ms)	Event
0	Firefox, via NEAT, made DNS A and AAAA queries for server
36	Receive the AAAA record response
36	Receive the A response
54	TCP connection initiated to the IPv4 address
55	TCP connection initiated to the IPv6 address
105	ICMPv6 Port Unreachable
106	IPv4 SYN-ACK. No further v6 packets were seen in the trace.

Repeating test 1b with the native Firefox created essentially the same result with the SYN-ACK being received at T=108ms.

Test 1c This test fully enabled IPv4 and IPv6 on both client and server.

Table 18: Test 1c — timeline of connection events at the client.

Timestamp (ms)	Event
0	Firefox, via NEAT, made DNS A and AAAA queries for server
41	Receive the AAAA record response
43	Receive the A response
61	TCP connection initiated to the IPv4 address
61	TCP connection initiated to the IPv6 address
98	IPv6 SYN-ACK Received
103	IPv4 SYN-ACK Received
110	IPv4 TCP FIN Transmitted. No further v4 seen in trace.

Repeating this test with the native Firefox created essentially the same result with the governing connection being completed at time T=105ms.

Test 2 This test utilized an SCTP and TCP enabled NEAT-port of Firefox that initiated communication with a server that was TCP only. The test used only IPv4 but was configured to give a preference to SCTP via the NEAT policy manager (Listing 3). This setup emulates the rollout period of a new protocol where applications will want to offer advanced functionality but need to be prepared to seamlessly fallback to traditional modes of operation.

This test was repeated 20 times in the baseline configuration. The test was then modified and rerun an additional 20 iterations with 10% packet loss being added to the network.

Test 2 measurements could be taken using the NEAT library logging and timestamp facilities instead of relying on the more laborious pcap and wireshark approach of Test 1. This is possible for this test because native Firefox, which obviously does not have NEAT logging, also has no SCTP functionality so cannot be used as a control for this test. Several of the NEAT log samples were tested against the PCAP data to confirm consistency.

```
{
    "uid":"reliable_transports",
    "description":"reliable transport protocols profile",
    "policy_type": "profile",
    "priority": 2,
    "replace_matched": true,
    "match":{
        "transport": {"value": "reliable"}
    },
    "properties":[
```

```
        [{"transport": { "value": "SCTP", "precedence": 0, "score": 0}},
         {"transport": { "value": "TCP", "precedence": 2, "score": 2}}
         ]
    ]
}
```

Listing 3: SCTP Favoring Profile.

A raw sample of NEAT library logging from one iteration is included in Listing 4. Note the preference for SCTP both in the candidate order and in the connection results at times 0.181953 and 0.191022.

```
[0.000448][DBG] neat_set_property
[0.000453][DBG] neat_set_property - {    "transport":        {
          "value": ["SCTP","TCP"],            "precedence": 1        }}
[...]
[0.001856][INF] Available src-addresses:
[0.001862][INF]   IPv6: 2604:6000:1513:4726:329f:40a8:a4f0:c325/64
          pref 604308 valid 604308
[0.001868][INF]   IPv6: 2604:6000:1513:4726:6534:3ba8:652:646e/64
          pref 0 valid 441040
[0.001873][INF]   IPv6: 2604:6000:1513:4726:e8d3:d851:8cb3:7758/64
          pref 8996 valid 527417
[0.001878][INF]   IPv6: ::1/128 pref 4294967295 valid 4294967295
[0.001883][INF]   IPv4: 192.168.16.138/24
[0.001887][INF]   IPv4: 127.0.0.1/8
[...]
[0.110229][DBG] on_pm_reply_post_resolve
[0.110330][DBG] Reply from PM was: [
  {

    "__he_delay": {
      "evaluated": false,
      "precedence": 1,
      "score": 0,
      "value": 10
    },

    "domain_name": {
      "evaluated": false,
      "precedence": 2,
      "score": 0.0,
      "value": "linode64.ducksong.com"
    },

    "transport": {
      "evaluated": false,
```

```
      "precedence": 1,
      "score": 0.0,
      "value": "SCTP"
    }
  },
[...]
[0.110399][DBG] HE Candidate  0:       lo [ 1]     SCTP/IPv4 <saddr 127.0.0
   .1>
                <dstaddr 192.155.95.102> port    80 priority 0
[0.110404][DBG] HE Candidate  1:       lo [ 1]      TCP/IPv4 <saddr 127.0.0
   .1>
                <dstaddr 192.155.95.102> port    80 priority 1
[0.110409][DBG] HE Candidate  2:     eno1 [ 2]     SCTP/IPv4 <saddr 192.168
   .16.138>
                <dstaddr 192.155.95.102> port    80 priority 2
[0.110414][DBG] HE Candidate  3:     eno1 [ 2]      TCP/IPv4 <saddr 192.168
   .16.138>
                <dstaddr 192.155.95.102> port    80 priority 3
[...]
[0.129643][INF] nt_connect: Bind fd 38 to 192.168.16.138
[0.129665][DBG] SCTP stream negotiation - offering : 123 in / 123 out
[...]
[0.139633][INF] nt_connect: Bind fd 43 to 192.168.16.138
[0.139672][DBG] on_he_connect_req: Connect successful for fd 43, ret = 0
[...]
[0.181936][DBG] HE Candidate connected:     eno1 [ 2]     SCTP/IPv4 <saddr
   192.168.16.138>
                <dstaddr 192.155.95.102> port    80 priority 2
[0.181953][DBG] he_connected_cb - Connection status: 111 - Connection
   refused
[...]
[0.191022][DBG] HE Candidate connected:     eno1 [ 2]      TCP/IPv4 <saddr
   192.168.16.138>
                <dstaddr 192.155.95.102> port    80 priority 3
[0.191032][DBG] he_connected_cb - Connection status: 0 - Success
[0.191037][DBG] First successful connect (flow->hefirstConnect)
[0.191043][DBG] send_result_connection_attempt_to_pm
```

Listing 4: Sample from a NEAT log.

A representative sample of a Test 2 iteration is included in Table 19 in order to illustrate the whole lifecycle of a connection.

Table 19: Test 2 — timeline of baseline connection events at the client.

Timestamp (ms)	Event
0	Firefox, via NEAT, made DNS A and AAAA queries for linode.ducksong.com (a host which only a A entry)
46	Receive the A response
58	SCTP session initiated
68	TCP connection initiated
108	SCTP session received an SCTP abort
123	IPv4 SYN-ACK Received

This result does meet the expectation of enabling seamless backwards TCP compatibility while giving the evolving SCTP protocol a negotiation advantage.

Test 2 repeatability The Test 2 scenario was repeated 20 times to assess the variabiltiy of the results and the appropriateness of the 10 ms advantage. The results (see Figure 9) contained low variation and in each case the SCTP abort was received prior to the TCP SYN-ACK. The dots on the plot represent the deterministic completion of the handshake regardless of whether it completed (i.e., TCP SYN-ACK received) or was aborted (i.e., SCTP ABORT received).

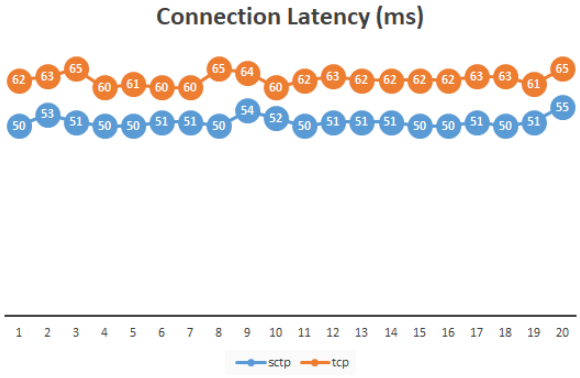


Figure 9: Happy Eyeballs with 0% Packet Loss

Finally, the same test was repeated using a lossy network. 10% packet loss was added to the network via a Linux firewall rule (Listing 5).

```
tc qdisc add dev eno1 root netem loss 10
```

Listing 5: Packet Loss Configuration.

The results of this test (see Figure 10) are consistent with the lossy configuration and the previous results. The baseline configuration results in all SCTP connection attempts being rejected which in turn requires a TCP connection for the test to complete. Because of that, TCP packet losses are represented on the graph as very large numbers, due to the Linux TCP kernel's 1-second SYN loss recovery algorithm. However, SCTP losses result in gaps on the graph as those connections are never deterministically resolved. The SCTP connections do not complete because the successful TCP connection

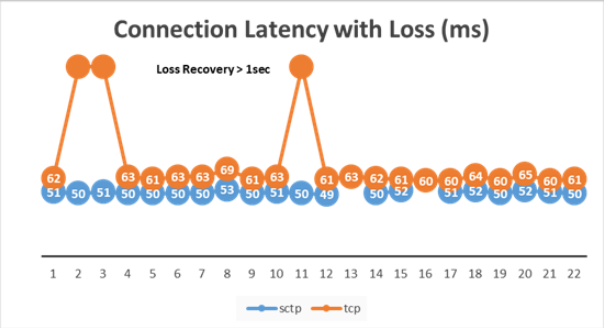


Figure 10: Happy Eyeballs with 10% Packet Loss

completes before the SCTP loss retry machinery finishes and the SCTP work is then abandoned as superfluous.

2.3.4 Key findings and implications

The key finding of this test is that NEAT Happy Eyeballs (HE) strategies are competitive with bespoke native HE implementations like those of Firefox. Due to the depth of the NEAT transport library and breadth of policy manager options Firefox was able to create SCTP association attempts to the web server without meaningfully penalizing its legacy TCP support in the tests described here. In this case TCP was used, due to lack of deployment of SCTP on the server, but SCTP was enabled as a path for evolution at minimal cost.

Utilising this functionality in a standard transport layer API promises to bring the protocol evolution that Happy Eyeballs enables to a wider range of applications through ease of use.

Mozilla has a particular interest in enabling QUIC and other UDP based protocols to thrive on the Internet. Doing so will require an ecosystem that supports HE techniques as workarounds for the long tail of firewalls and long forgotten configurations that can impair such deployment.

The tests described here are a reasonable proxy for evaluating mechanisms for widely deploying QUIC and other future transport innovations.

2.4 EMC use case

The EMC use case evaluates a datacentre scenario in which a logically centralised network controller is aware of application requirements and network conditions. The use case leverages the transport optimisations provided by the NEAT System interacting with an SDN controller/orchestrator that manages the datacentre network.

The primary goal of this use case was to improve the performance for large data transfers (so-called *elephant flows*) within a datacentre, using the NEAT System augmented by the knowledge of the underlying network with a minimal impact on the applications running over it. The optimisation aims to improve the performance of individual end-host applications as well as the overall network performance in terms of flow completion times (FCT). The goal was to exploit path diversity using MPTCP, in cases where this may have a positive impact on the FCT.

A second aspect evaluated in the context of large file transfers is the use of deadline-aware less-than-best-effort (DA-LBE) congestion control mechanisms that were designed as part of WP3. Here, a data replication scenario comprised of a client connected to a datacentre over a wide area network

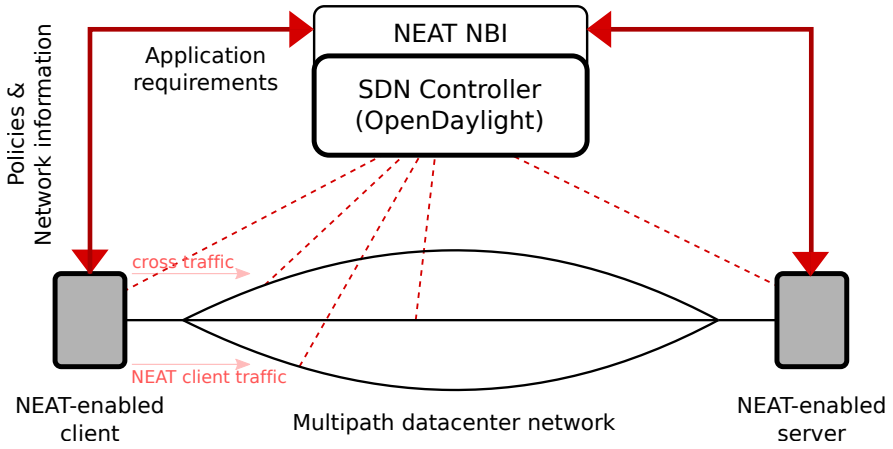


Figure 11: datacentre topology for the EMC use case (SDN experiments).

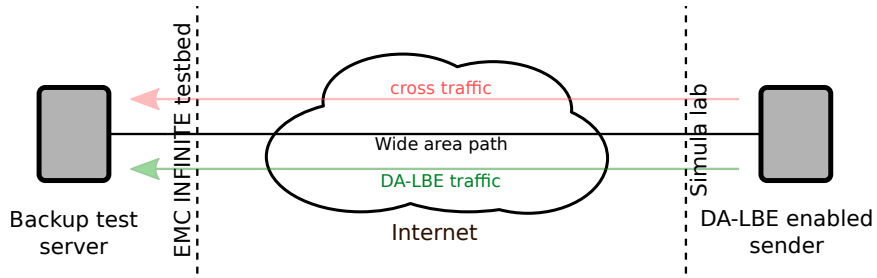


Figure 12: WAN topology for the EMC use case (DA-LBE experiments).

(WAN) was considered. The test demonstrated file transfers targeting a predefined completion time without adversely impacting concurrent network traffic.

2.4.1 Test topology

Figures 11 and 12 provide a high-level view of the two topologies which serve as the basis of the experiments, as defined in the test plan in deliverable D4.2 [11]. The components are described in Table 20. The topology in Figure 11 was used to evaluate an SDN datacentre scenario. The topology depicted in Figure 12 was used to evaluate a WAN cloud provider scenario using DA-LBE.

2.4.2 Test implementation

Several tests were constructed to demonstrate that an integration between NEAT and a SDN controlled network leads to improvements for both the application and the network, on the one hand, and the benefits brought by the use of DA-LBE, on the other hand. Table 21 summarizes the experiments defined in Deliverable D4.2 [11]. In the SDN experiments congestion was induced by replaying realistic cross-traffic generated using existing traffic generator tools [1, 40].

Table 20: Components of EMC's use case testing environment.

Component	Description
NEAT-enabled application	A client/server data synchronization application for transmitting large files across the network. A NEAT-enabled port of Rsync (`neat-rsync`) was selected as a representative open-source application. Denoted *NEAT-enabled client/server* in Figure 11.
Traffic generator	The traffic generators D-ITG and DCT2Gen were used to generate *cross traffic* with the desired characteristics.
Multipath datacentre network	SDN-enabled physical or virtualised topology within the EMC INFINITE testbed, comprised of three disjoint paths between a source and destination node, hosting the client and server of the application, respectively. The network is managed by a network controller supporting OpenFlow as the southbound protocol. The experimental network was used to simulate different conditions in a managed network, e.g., high/low congestion, high/low latency, heavy/light load.
Wide area path	A path traversing the public Internet between the INFINITE testbed and a node hosted at SRL.
SDN Controller	The OpenDaylight open-source SDN controller framework was used to manage the datacentre network, monitor its status and interact with the attached NEAT Systems on the hosts (relying on the work developed in WP3).

Table 21: Experiments for EMC's use case testing.

Test ID	Summary
1a	Large file transfer with legacy Rsync in both an empty and a congested network, in order to determine baseline performance
1b	Large file transfer from client to server in both an empty and a congested network with `neat-rsync`
2	Large file transfer between a `neat-rsync` client and server in a datacentre network with SDN-supported orchestration, and with empty and congested links
3	Transparent handling of elephant flows using controller-assigned differentiated services code point (DSCP) marking or MPTCP sub-flows mapped to disjoint network paths using `neat-rsync`
4a	Large DA-LBE file transfer between the DA-LBE enabled sender and the Backup test server over the WAN with other large TCP Cubic flows sharing the path
4b	Large file transfers between the DA-LBE enabled sender and the Backup test server over the WAN comparing the relative send rates of Cubic based DA-LBE and Vegas based DA-LBE with TCP Cubic

2.4.3 Results

The expected results for the above sequence of tests as put forward in Deliverable D4.2 can be summarised as: a) reduction of flow completion times, b) fine-grained control of how elephant flows are handled in datacentre networks, c) improved network utilisation by exploiting path diversity, and d) WAN file transfers with deadlines and negligible impact on concurrent traffic.

In the sequel we present our results beginning with an evaluation of data transfers in testbed and emulated SDN environments (Tests 1a, 1b, 2 and 3 in Table 21), followed by an analysis of the developed DA-LBA approaches in a wide area network (Tests 4a and 4b).

SDN integration and performance evaluation The SDN scenario considers a typical fat-tree datacentre topology in which a class of applications generates large flows with data sizes known to the respective NEAT system. We used the NEAT-enabled version of `rsync` (see D4.1 [12]) where for each opened flow the data size in bytes is passed to the NEAT API as a NEAT property `flow_size_bytes`. Using this information the aim was to allow the network to minimize the flow completion time for each individual client as well as to analyze the impact of the centralised approach on the network-wide flow completion times.

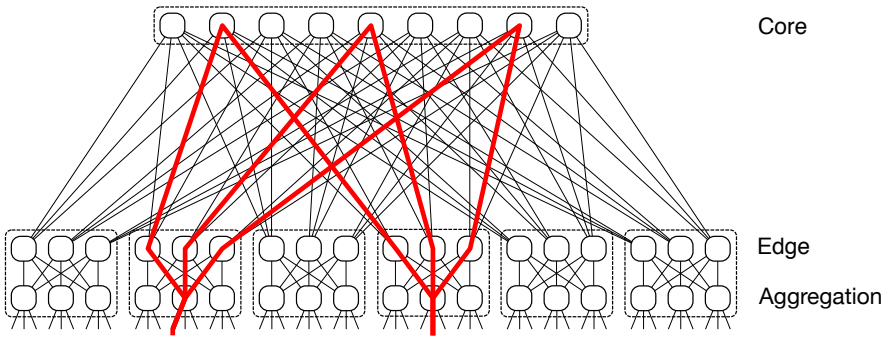


Figure 13: Fat-tree datacentre topology (K=6).

A key feature of fat-tree topologies is that they are rearrangeably non-blocking, meaning that for any communication pattern, there exists some set of paths that will saturate all the bandwidth available to the end hosts in the topology [8]. This implies that multiple paths may exist between any pair of source and destination nodes. A K-ary fat-tree topology is comprised of three layers: core, aggregation and edge, where K defines the number of pods in the topology, as well as the associated number of switches servers and links. A typical fat-tree topology with $K = 6$ is depicted in Fig. 13. The highlighted paths illustrate three node-disjoint paths between two end hosts.

Unfortunately, current routing algorithms rely on shortest path based approaches to identify the path taken by flows, which are not suitable when multiple equal cost paths are available. Allocating network resources on a per packet basis is problematic as the performance of most transport layer protocols used today suffers if packet reordering occurs. In addition, standard hash based ECMP approaches, which aim to distribute traffic across multiple equal cost paths, are also not ideal as hashing does not take into account the actual utilisation of the individual paths which may result in a mapping of competing flows to oversubscribed links. Furthermore, in practice it is not feasible to utilize a centralised controller to continuously rearrange all active end-to-end flows such that no oversubscription of the capacity occurs. This is due to the management overhead and timing requirements associated with managing dynamic and short-lived flows.

The motivation behind this work is to demonstrate that NEAT's SDN integration mechanism can be used to flexibly enable approaches for managing salient flows in a datacentre network. Specifically, our aim is to utilize an SDN controller to explicitly allocate the paths for elephant flows on demand, while handling smaller flows using static forwarding rules. By only processing a limited number of large flows we expect to minimize the controller load, as well as the number of forwarding entries that must be stored in the switches.

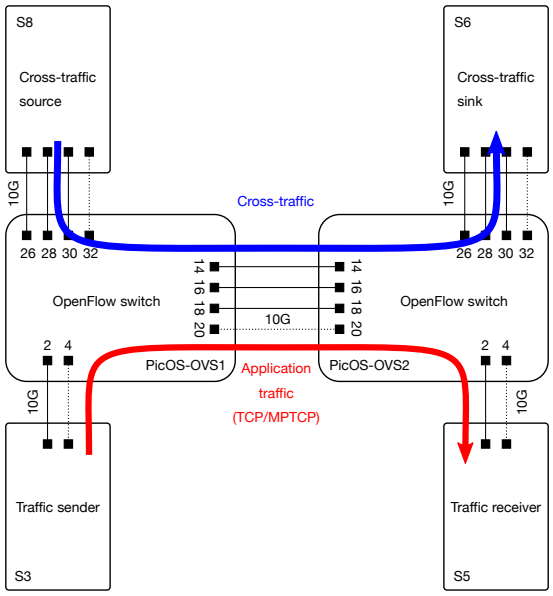


Figure 14: datacentre lab topology for the EMC use case.

Table 22: Lab setup for EMC use case.

Component	**Description**
Switches	Pica8 P-3920, PicOS Version 2.3.7
CPU	Intel Xeon E5-2609 @2.40GHz, 2 NUMA nodes
OS	Ubuntu 16.04.4 LTS, MPTCP kernel version 4.14.24.mptcp
NIC	Intel 82599ES 10-Gigabit

Our SDN experiments are divided in two parts. First we consider a subgraph of the fat-tree network topology and analyze the flow completion times for a single instance of the file transfer client under different traffic conditions using physical network infrastructure. Then, in the next section, a large scale datacentre topology is evaluated using an emulated setup.

The lab setup of the fat-tree subgraph implemented in EMC's Infinite testbed is depicted in Fig. 14. Table 22 provides details of the used components. The topology uses 10 Gbps fibre Ethernet links, where the client node *S3* and the server node *S5* are connected to SDN switches, using a single 10 Gbps edge link. Between the two switches, three 10 Gbps paths are configured, such that the flows from the client can be forwarded across three disjoint network paths (core links). The setup mirrors the paths highlighted in Figure 13. The forwarding rules on the SDN switches are installed by a controller running on a separate host. In addition, the servers *S8* and *S6* are used as a traffic generator source and sink, respectively, to inject cross traffic into the core links. The traffic generator is used to simulate different utilisation levels in the core network.

We evaluate the impact of SDN-supported orchestration, where a controller dynamically installs flowtable entries for each newly arriving TCP flow across all SDN switches in the network. First, we measure the additional latency introduced by the processing of the initial flow at the controller and

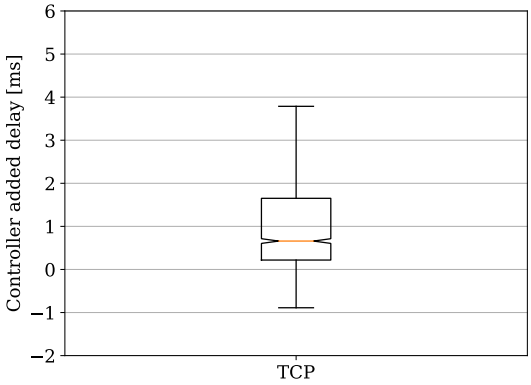


Figure 15: Controller impact on latency

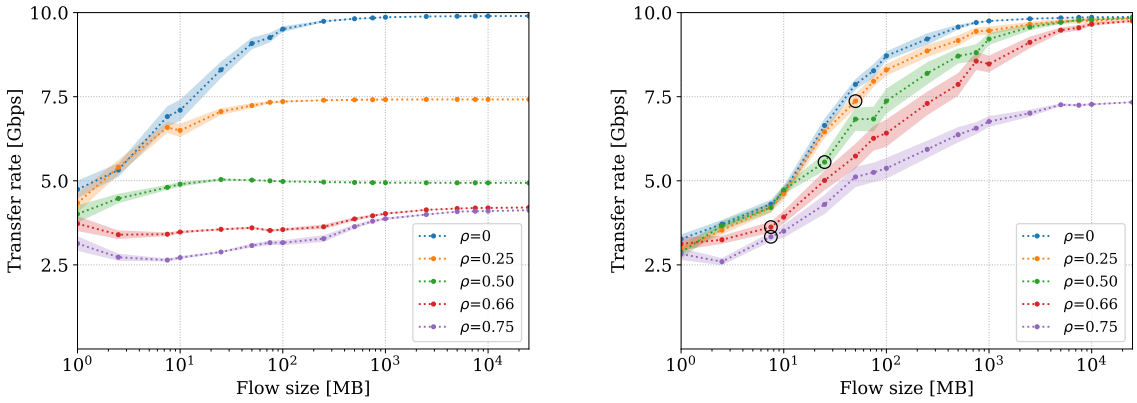
subsequent installation of the forwarding rules at the relevant switches. The following set of measurements map to test *2* in Table 21 for uncongested networks. We evaluate the controller induced delay for a single TCP flow. Figure 15 depicts the absolute increase in the FCT. Unsurprisingly, the mean FCT increases by approximately 0.6ms compared to a scenario where statically pre-configured forwarding entries are used. We note that for small flows such an increase yields a significant increase of the effective transfer rate.

Next, we consider the performance of the TCP and MPTCP protocols on congested and non congested links. We transfer flow sizes d_f ranging from 1 MB to 25GB from the client on *S3* to the server on *S5* and measure the FCT t_f at *S5*. From the FCT we calculate the effective transfer rate $r_f = d_f/t_f$ for each flow size. Each experiment is repeated 100 times.

Cross traffic is generated from node *S8* to *S6* to achieve a predefined utilisation ρ over each of the three core paths between the SDN switches. To this end, constant bitrate UDP flows are generated using the `pktgen` packet generator on each of the three interfaces in *S8*, where the utilisation on each interface is set to $\rho \in \{0, 0.25, 0.50, 0.66, 0.75\}$. On the client *S3* we use the *ndiffports* path manager for all MPTCP tests, which generates multiple sub-flows over a single interface for each file transfer. We set the number of sub-flows to 3. We use the *cubic* congestion control algorithm for TCP flows and the *OLIA* congestion control algorithm for MPTCP flows. In this scenario, we expect MPTCP to be able to exploit the network path diversity, as the controller allocates each sub-flow to a separate, disjoint path in the network core.

Figure 16 depicts the results (shaded areas represent 95% confidence intervals). As expected TCP-based transfers are limited by the available bandwidth on the individual core paths. For $\rho < 0.50$ the TCP traffic yields to the cross traffic. On the other hand, for MPTCP-based transfers the sender is able to utilize its full bandwidth (R=10 Gbps) for very large file sizes. We observe that the transfer rate is dependent on the utilisation of the cross traffic paths, where the full transfer rate is reached faster for less utilized cross traffic paths. As expected, for $\rho > 0.66$, i.e., when the available bandwidth of the three core links is less than R, the sender cannot reach its full transfer rate.

From the performed measurements we gain the following insights which impact the design of our SDN controller application, and the protocol selection using NEAT policies. Firstly, in an uncongested scenario, the current MPTCP setup using the *ndiffports* path manager consistently performs worse than the default TCP stack. While the authors of MPTCP note that this path manager has not been optimized for production use, for the evaluated single sub-flow scenario the performance difference may be attributed to the congestion control algorithm utilized by MPTCP. On the other hand, as net-



(a) TCP traffic

(b) MPTCP traffic

Figure 16: TCP/MPTCP throughput for different flow sizes on congested links.

work utilisation increases, MPTCP is able to achieve significantly better results than TCP by utilizing the bandwidth available across multiple core network paths. However, the exact flow size threshold T_f beyond which the benefits of MPTCP start to show effect is dependent on the network utilisation. Specifically, the threshold decreases as the utilisation increases. In Figure 16 for each level of utilisation ρ we mark the flow size thresholds beyond which MPTCP performs better than TCP with a circle.

To evaluate the cause for MPTCP's sub-optimal performance, we analyze the impact of the used congestion control algorithms on the FCT. We repeat the flow measurements in an uncongested network setup, where for each file transfer the client generates one connection for both TCP and MPTCP (single MPTCP sub-flow). Consequently, in this case a similar performance level is expected from both transport protocols. We evaluate the congestion control algorithms Cubic and New Reno for TCP, and Cubic, LIA and OLIA for MPTCP connections. Figure 17 depicts the results, where the shaded areas represent 95% confidence intervals. We also evaluate the baseline performance of the TCP and MPTCP protocols using statically configured forwarding entries in the SDN switches, corresponding to tests *1a* and *1b* in Table 21 for uncongested networks. The measurements show that TCP outperforms the MPTCP implementation regardless of the used congestion control algorithms, indicating that the observed performance difference is not caused by the used CC.

While further investigation is needed to identify the root cause for the unexpected performance of MPTCP, we argue that operators can expect to encounter similar idiosyncrasies whenever novel protocols are deployed in a production environment. Ultimately, facilitating such deployments is a key motivation behind NEAT. The exact protocol behaviour is likely to depend on their parametrization and the characteristics of the topology in which they are deployed. Thus, we propose an approach in which the SDN controller directly observes the performance of specific protocol configurations in order to select the optimal approach for a given environment. Specifically, the controller collects samples of the actual flow completion times using capabilities of the OpenFlow protocol. To this end, we make use of the OpenFlow `OFPFF_SEND_FLOW_REM` feature, which causes switches to notify the controller whenever a configured flow entry expires. By enabling this feature on all relevant flow entries and assigning an IDLE expiration timer for each, the controller can measure the duration for which a specific flow was active on the switch. This is achieved by analyzing `OFPT_FLOW_REMOVED` events received from the switches and subtracting the configured IDLE time from the flow duration included in the event packet. Our experiments indicate that the resulting FCT estimates provide a similar level of accuracy as the FCT estimates measured at the receiver side.

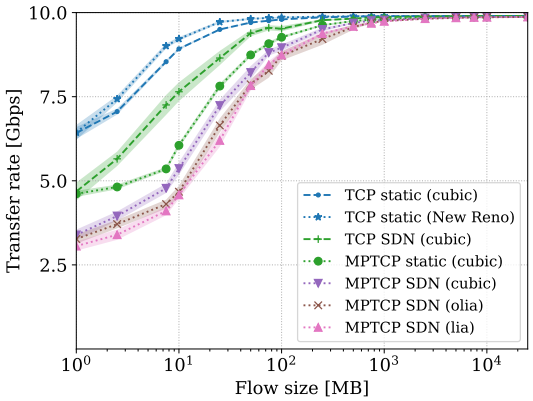


Figure 17: TCP and MPTCP data transfers with static and dynamically installed flow rules and various CC algorithms. No cross traffic.

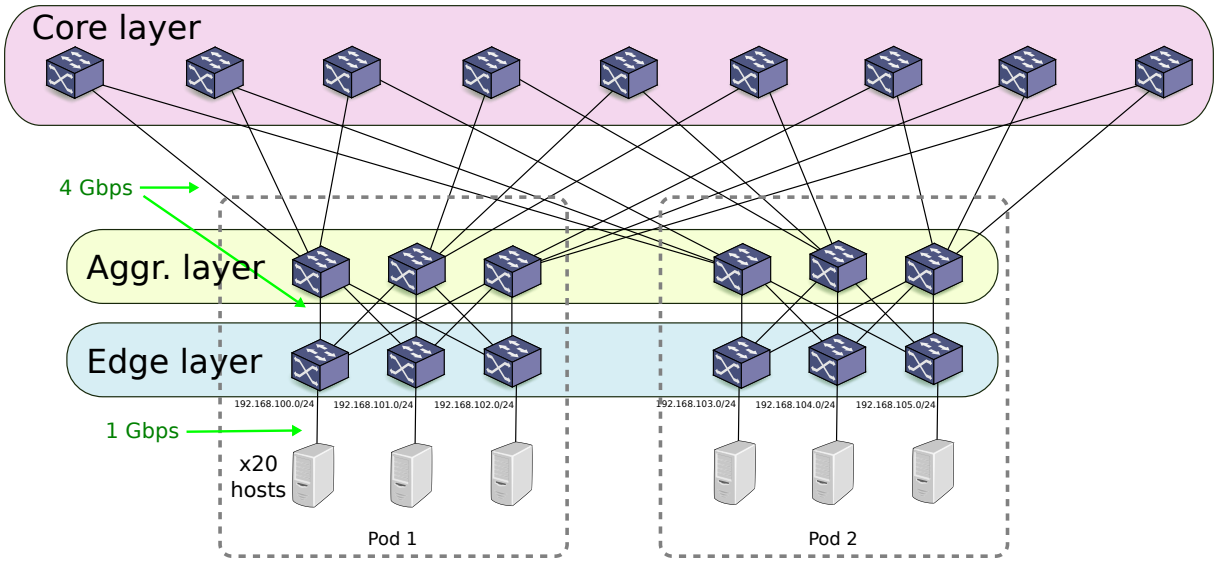
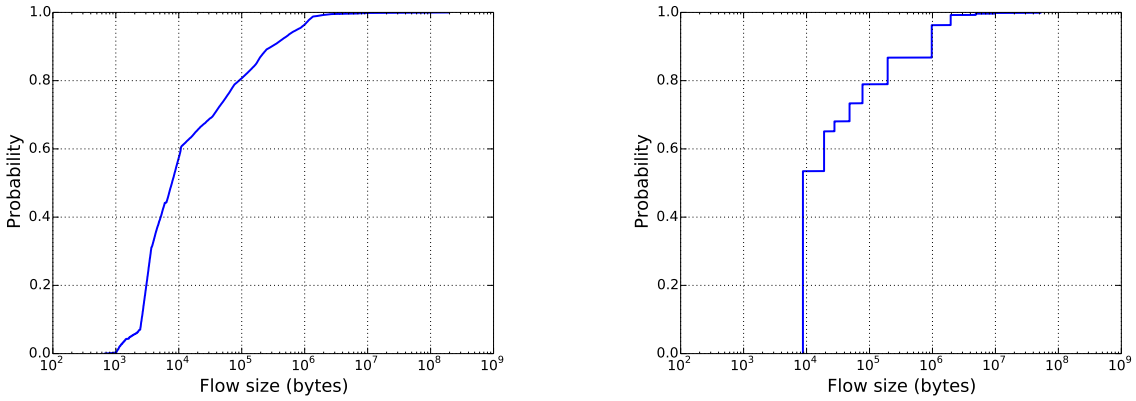


Figure 18: Emulated fat tree topology.

Large scale datacenter topology To assess the behavior of NEAT-enabled hosts with a larger scale of hosts and network nodes, a set of experiments were performed, through the emulation of a datacenter network topology. The main goal of the conducted experiments consisted in measuring the performance of the multiple concurrent file transfers within a Software-defined Networking (SDN) data center, comparing it to the usage of different transport protocols and with a NEAT-based approach. The following results map to Tests 2 and 3 in Table 21.

A fraction of a fat-tree topology with 2 pods was used for the evaluation, as pictured in Figure 18. Overall, the topology has 21 switch nodes (9 in the core layer, 6 in the aggregation layer, and 6 in the edge layer). Each edge switch has 20 hosts connected by one network interface, totaling 120 hosts in the topology. All the host to edge switch links were configured to have 1 Gbps of bandwidth capacity. The remaining links were limited to 4 Gbps of capacity. In addition, all the links had a fixed delay of 500 μs. The Common Open Research Emulator (CORE) [7] was used to emulate the topology. Every switch used Open vSwitch (OVS) [4] as an OpenFlow (OF) software switch (OF switch).

With this topology, it is possible to have 9 different shortest paths between hosts located in different pods, which includes 3 node-disjoint paths. For intra-pod traffic, it is possible to establish 3 different



(a) Flow size distribution

(b) Discretized flow size distribution

Figure 19: Flow size distributions used in emulations.

paths between the edge switches.

To specify the traffic flow scheduling, regarding their source and destination hosts, starting time and size, a traffic matrix file was created using the DCT2 Gen traffic generator [40]. The used traffic matrix contained 39302 flows, scheduled during approximately a minute, with sizes ranging between 667 bytes and 188 MB.

The cumulative distribution function (CDF) of the flow sizes is displayed in Figure 19a. The total amount of data in all flows combined is 10.51 GB. We emulated various levels of background load on all the inter-switch links by adjusting their capacity. A background load of 90% was for instance emulated by limiting the link capacity on all the inter-switch links to 400 Mbps. For some experiments, the flow size distribution was further discretized into twelve flow sizes as illustrated by the CDF in Figure 19b. This was done to allow more easy investigation of the performance at different flow sizes.

The SDN controller was built on OpenDaylight (ODL) [5] and featured basic network management capabilities, namely:

Address tracking The controller keeps track of where the different IP and MAC addresses were last seen in the network by storing the information about the OF switch ports where the addresses were seen. This information can then be looked up by the other controller components;

Path calculation A multiple path calculator is used by the SDN controller whenever it is required to compute multiple paths between different nodes. The controller allows the specification of different path calculation algorithms, though in the scope of these experiments, Yen's algorithm [41] was selected, using unitary weights for classifying each link, computing the shortest paths;

Flow scheduling When multiple paths are available between two different hosts, the SDN controller can apply different scheduling strategies to decide which of the multiple paths are available. For these experiments, the first path would be randomly picked, and the upcoming ones were selected by a Round-Robin scheduler;

Elephant flow processing The SDN controller can process elephant and mice flows differently. For example, a path with low delay can be assigned to mice flows, while elephant flows can be scheduled among multiple paths with high capacity. Different signalling methods can be used to distinguish these two types of flows. Specifically, during the conducted experiments, DSCP values

were used to mark elephant flows (this value was set to 1), while mice flows would have the default value (0). In addition, during the conducted experiments only elephant flow packets were sent to the controller. Therefore, all the computed packets were assigned to the elephant flows;

MPTCP Flow correlation By keeping track of the MPTCP negotiation header fields (exchanged keys and tokens) during the initial sub-flow TCP packets, the SDN controller can load-balance new MPTCP sub-flows from the same association to different paths, ensuring that those are as disjoint as possible;

Incoming packet handling Whenever a packet is sent from an OF switch to the controller, it enters a logic pipeline that finds its destination node, computes new paths (if necessary) and installs new flow rules, according to the scheduled path.

Due to computational restrictions, it is not feasible to send all the packets from new flows to the SDN controller in a data center network. With that, it is necessary to initially install generic forwarding rules instead of individual fine-grain matching rules. Equal-cost multi-path routing (ECMP) was used to configure incoming flows with multiple output port possibilities, whenever available. The output port of matching flow packets was then computed by a hash value containing some of its header fields (MAC addresses, IP addresses, TCP/UDP ports, etc.). Since there are multiple forwarding possibilities to reach out to the different subnets with upstream traffic, i.e. 3 per edge-layer and 3 per aggregation-layer switches, ECMP allows the different flows to be distributed among the different available paths.

The experiments were conducted in the emulated network with different transport protocol variants:

- **TCP on all flows:** All the flows use TCP and are routed according to installed ECMP forwarding rules;
- **NEAT-assisted:** TCP for mice flows, MPTCP for elephant flows. Two variants of this approach were tested:
  - **Local:** The SDN controller is not informed about the elephant flows. All the flows are routed with ECMP;
  - **SDN-aided:** The first packet of the elephant flows is pushed to the controller, which then decides where it should be forwarded.

For all the NEAT-assisted tests, the number of MPTCP sub-flows used per association was varied between 2 and 8. Different elephant flow detection thresholds were also used in the experiments: 10 KB, 100 KB, 1 MB, 5 MB and 10 MB.

All experiments were repeated over 25 iterations. Each emulated iteration was comprised of the following procedures:

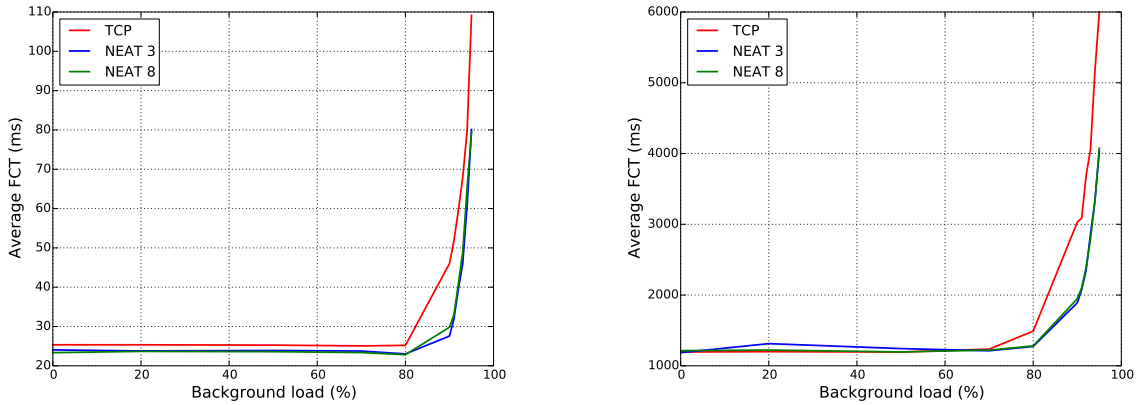
1. **Topology initialisation:** All the network nodes and links are created by CORE, which also initialises OVS in all the switch nodes. The SDN Controller IP address is provided in the topology configuration file and the switches use that address to connect to the SDN Controller.
2. **Identification of hosts:** The SDN controller initially does not know how many hosts there are, and where the hosts are located in the network. To populate the controller with this information, each host sends an ICMP message to an unused IP address located in the same subnet. Since the OF switches have no forwarding rules for the destination address, they send the packet to the SDN controller, which uses it to learn the host location.

3. **Installation of forwarding rules:** ODL installs the forwarding rules responsible for basic network connectivity in the OF switches. These rules include:
   - **Individual flow rules:** These rules are installed on the edge-layer switches and each have an IP address match corresponding to an end-host and, as action, they forward the matching packets to the port where the host is connected in OVS. Each edge-layer switch has 20 end-host IP forwarding rules, corresponding to all their connected hosts, respectively.
   - **Downstream forwarding rules:** These rules were installed in the core-layer and aggregation-layer switches and used a 24-bit prefix network mask as a match. The core-layer forwarding rules forwarded every subnet traffic to the corresponding aggregation switch, while the aggregation-layer forwarding rules forwarded the traffic to the edge-layer switch responsible for the matching subnet.
   - **Upstream ECMP forwarding rules:** Installation of OF group forwarding rules with ECMP, as previously described.
4. **Traffic generation:** This stage can be separated by different steps:
   (a) The traffic generator receiver application starts in all the hosts.
   (b) The traffic generator sender is launched on every host, providing the individual host number (from 1 to 120), the traffic matrix file location, and elephant flow strategy (not identifying elephant flows/identify with DSCP or without DSCP value packet flagging) as input parameters. When identifying elephant flows, the corresponding detection threshold is also provided.
   (c) Every sender application sorts their flows (the ones with the same source number as the host) by their starting time.
   (d) A signal is sent by the host machine, triggering the start of the flow scheduling on all senders.
   (e) For every completed flow, the receiver writes the respective FCT in a log file.
5. **Clean-up:** After all the flows are completed, the log files are saved, the topology is destroyed and the emulation host machine restarts. This guarantees the same clean state on every iteration.

With the individual FCT values of every flow from the different iterations, the average FCT values were calculated.

We first examine the impact of background load on the FCT for different protocol configurations. Background load values of 0%, 20%, 50%, 70%, 80%, 90%, 91%, 92%, 93%, 94% and 95% were evaluated. Figure 20 shows the FCT for TCP and for a NEAT-SDN aided approach with an elephant flow threshold of 10MB. For the number of subflows used by MPTCP, results for a setting of three and eight subflows are shown in the Figure.

Consistent with the results shown earlier in Figure 16, we can see that the use of MPTCP for elephant flows brings little benefit at low loads. Looking at the FCT for the largest flows (Figure 20b), a load of 80% is needed before any NEAT-SDN brings any visible benefits. At lower loads the single path between the end-host and the edge switch constitutes the bottleneck for elephant flows, eliminating any possible gains from using the multiple paths available between the switches. At higher loads, on the other hand, we can see a marked improvement when using NEAT-SDN as compared to TCP. As the bottleneck shifts to the links between switches, elephant flows can take advantage of the path diversity in the network.



(a) FCT for all flows

(b) FCT for large flows (> 20 MB)

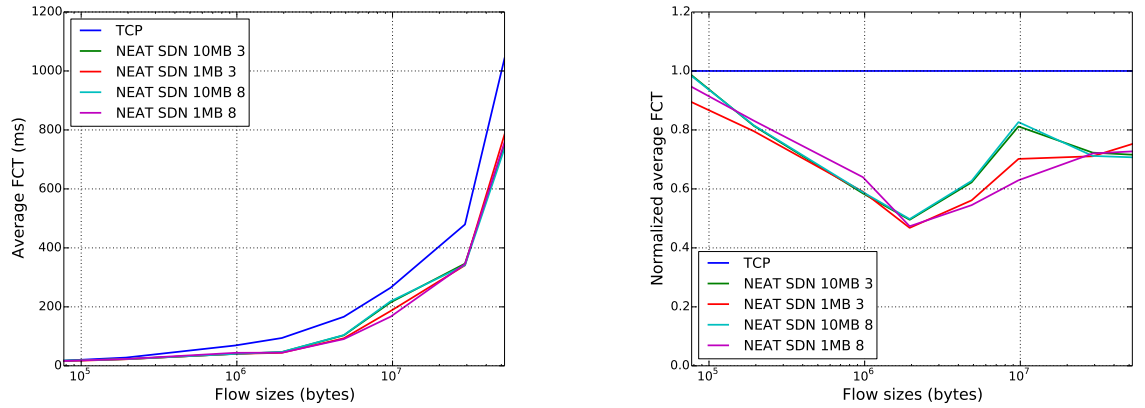
Figure 20: FCT at different background traffic loads.

Looking at the average FCT over all flow sizes (Figure 20a) there is a small gain also at lower loads. This gain comes from the medium sized flows benefiting when the large elephant flows use MPTCP, both from the less aggressive congestion control of MPTCP when competing for capacity on the host to edge switch link and from the load imposed on the network being more even. As the load increases, we can see a significant gain also when examining the average FCT over all flow sizes. Examination of the FCT of flows of varying sizes shows that this gain applies across flow sizes—flows of all sizes benefit when the elephant flows use MPTCP.

Next we examine the impact of the different protocol configurations on the FCT of different flow sizes in more detail, also using different elephant flow thresholds. Figure 21 shows the FCT of different flow sizes using elephant flow thresholds of 1 MB and 10 MB. To make it easier to examine the impact on different flow sizes we use a traffic matrix with a reduced number of flow sizes as shown in Figure 19b. The background load in the experiments is set to 90% and the same protocol configurations as in Figure 20 are used. Both the actual FCTs and the FCTs normalized with respect to the FCTs of TCP are shown in the Figure.

Looking at the results in Figure 21, we can see that, as expected, the actual gain in FCT from using NEAT-SDN is larger for larger flow sizes (Figure 21a). Looking at the relative FCTs (Figure 21b), we see that the trend for the relative performance gain at different flow sizes is less clear, although the smallest flow sizes still see the smallest gain. We see that the performance when using three or eight subflows for MPTCP is quite similar, suggesting that three subflows is mostly sufficient for reaping the benefits of the path diversity available in the network. For a threshold of 1 MB and for small flow sizes, the extra overhead generated by using eight subflows may even reduce performance. Looking at the impact of a 1 MB or a 10 MB threshold, we see that using a threshold of 1 MB provides some additional benefits at this high background load. Particularly, it is clear that using a threshold of 1 MB benefits flows that have a size between 1 MB and 10 MB.

Overall, the emulation results confirm the conclusions from the physical network infrastructure experiments. Using MPTCP for elephant flows can improve performance, both for the elephant flows themselves and for smaller flows, but the gain depends on the traffic load as well as the chosen elephant flow threshold. We saw performance gains from using a smaller threshold (1 MB) in our emulations, but a smaller threshold also increases the load on the controller. We were unable to obtain any stable results using even smaller thresholds (100 KB or 10 KB) for NEAT-SDN as the generated control



(a) FCT variation over different flow sizes

(b) Normalized FCT (TCP) over different flow sizes

Figure 21: FCT over different flow sizes and for different elephant flow thresholds at 90% background load.

traffic then became too excessive.

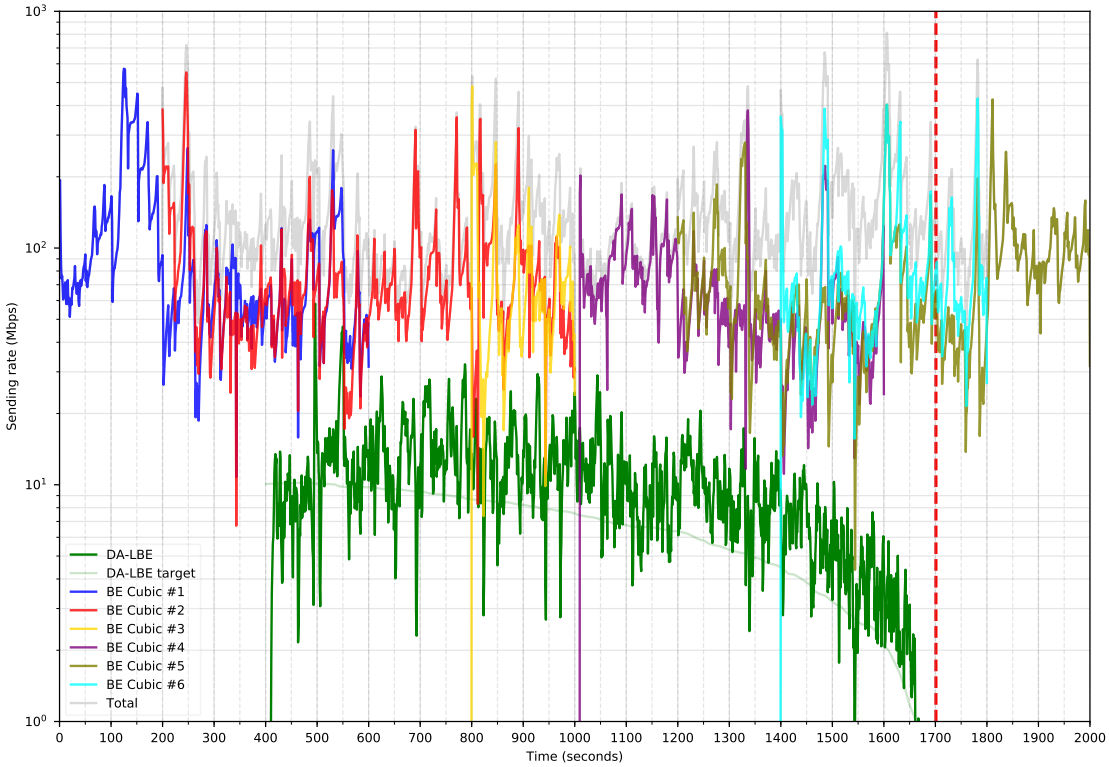
DA-LBE results in WAN The DA-LBE transport has a kernel part that manipulates the congestion price (ECN and delay) and a user-space part that calculates how the congestion price should be manipulated to keep the service LBE, but in a dynamic manner so that it can meet the given soft deadline. We use the framework outlined in [23] but, different to the example in that paper, for Vegas based DA-LBE we inflate the queuing delay since the α and β parameters cannot be changed dynamically on a per-flow basis in Linux.

We use the setup shown in Figure 12. The Internet path from the DA-LBE-enabled sender at Simula (Fornebu, Norway) to EMC (Cork, Ireland) traverses 17 hops. We also observe that the path has very high, fluctuating, and bursty cross traffic making it a good real network test for the DA-LBE mechanism. Figure 22 illustrates the operation of the Cubic and Vegas based DA-LBE mechanisms in a similar scenario to the simulations presented in [23]. The DA-LBE flow would need to send at about 10 Mbps to meet the deadline. We start and stop a number of end-to-end greedy TCP-Cubic flows so as to compare their behaviour with the DA-LBE flow. Their start/stop times are as follows (colours map to each of the plots in Figure 22):

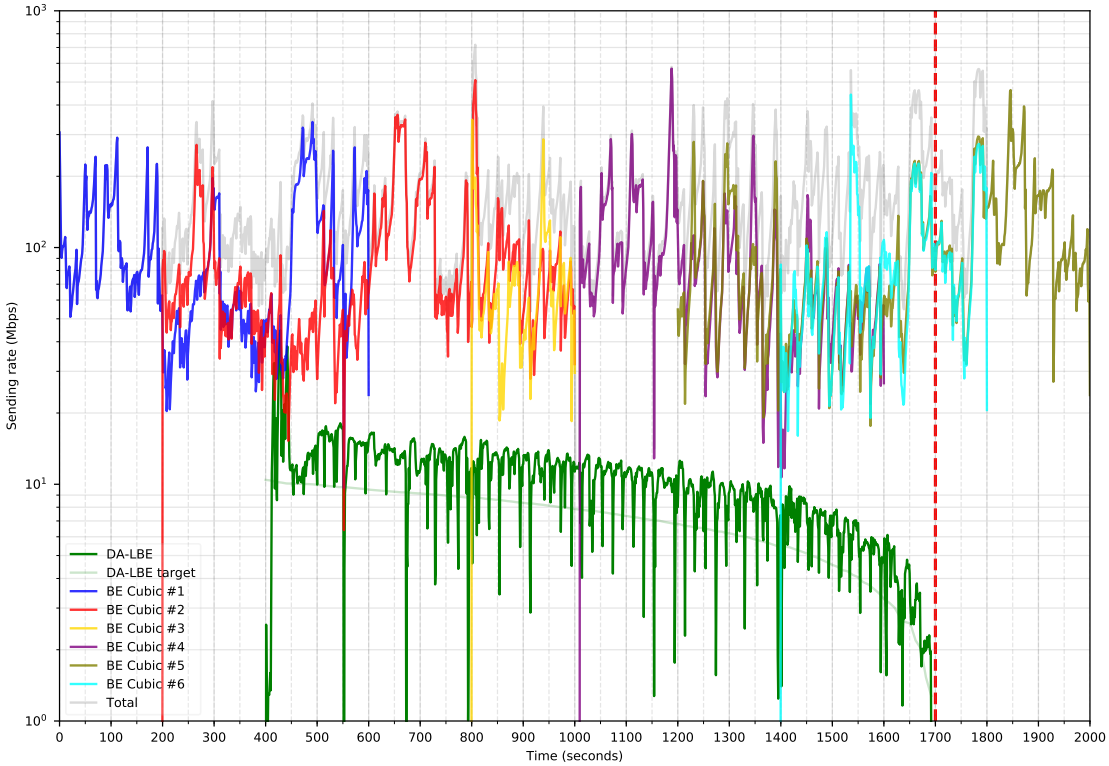
(i) $t = [0, 600]$ s (blue)
(ii) $t = [200, 1000]$ s (red)
(iii) $t = [800, 1000]$ s (yellow)
(iv) $t = [1100, 1600]$ s (magenta)
(v) $t = [1200, 2000]$ s (olive)
(vi) $t = [1400, 1800]$ s (cyan)

A light gray line shows the total traffic being sent between Simula and EMC-Cork. The DA-LBE flow (green) starts at $t = 400$ s with a deadline at $t = 1700$ s. From any point in time during the DA-LBE transmision, the light green (DA-LBE target) line shows the average rate the DA-LBE flow would now need to send at in order to finish sending its data at the deadline.

The Cubic based DA-LBE flow (green) in Figure 22a operates in an LBE manner, reacting appropriately to the traffic dynamics and completing before the deadline. The Vegas based DA-LBE flow (green) in Figure 22b generally operates in an even more LBE manner due to its more timely reac-



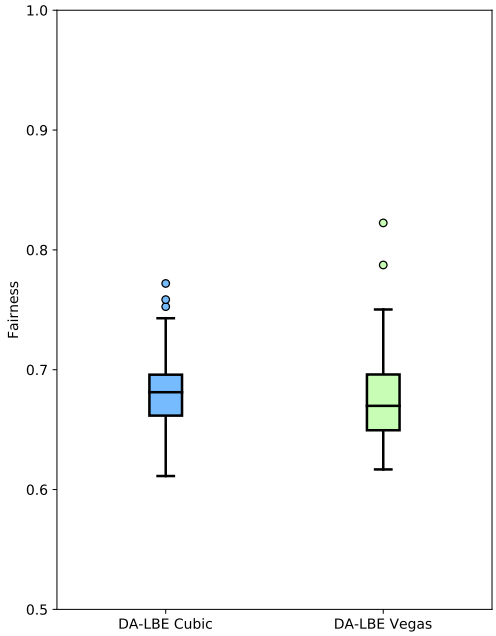
(a) Cubic based DA-LBE competing with Internet traffic and a number of "background" TCP Cubic flows.



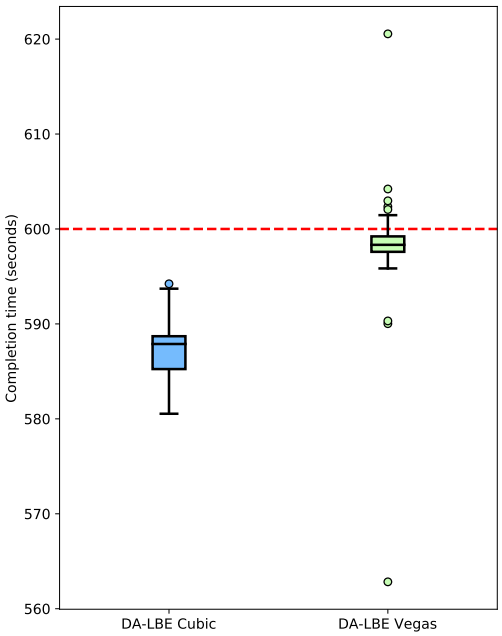
(b) Vegas based DA-LBE competing with Internet traffic and a number of "background" TCP Cubic flows ($\alpha = \beta = 16$, varying $\alpha = \beta$ did not seem to significantly change the behaviour)

Figure 22: Test 4a results (note logarithmic y-axis). DA-LBE data transfers over the WAN between Simula and EMC with competing TCP flows similar to the scenario in [23]. In addition to (uncontrolled) Internet traffic, various greedy "background" TCP Cubic flows start and stop through the scenario. The DA-LBE soft deadline is indicated by the vertical dashed red line at $t = 1700$ s.

tion to changes in queueing delay along the path, completing its transfer on the deadline. The Vegas estimate of base RTT is wrong (higher than it should be) in the initial period ($t = [400, 450]$ s). This causes the Vegas based DA-LBE to be more aggressive than is desirable for an LBE service in that initial period, a known problem with this type of delay-based congestion control (see [29]). Overall these real network results are similar to the simulation based results published in [23], giving confidence in the usefulness of the simulator as a prediction for the performance of the Linux implementation of DA-LBE across real networks.



(a) Jain's fairness index for DA-LBE data transfers compared to concurrent TCP Cubic data transfers. The range of possible values when comparing 2 flows is $[0.5, 1]$.



(b) Completion times for DA-LBE data transfers competing with the greedy TCP Cubic data transfers. The deadline of 600 s is shown with a dashed line.

Figure 23: Test 4b results. A DA-LBE data transfer competing with a standard greedy TCP Cubic transfer across the Internet from Simula to EMC. Box-and-whisker plots with boxes spanning the middle 50% and whiskers extending up to 1.5 times the inter-quartile range.

Test 4b measures the relative LBEness of the DA-LBE transfers with respect to a concurrent greedy standard TCP Cubic data transfer. The object is to send a 1.625 GB transfer within a 600 s deadline (the average data rate to achieve this is 21.67 Mbps). Each experiment ran for at least 10 minutes and was repeated 50 times for each DA-LBE mechanism. The experiments were run overnight on different nights, but starting at the same time of day. Unlike a simulation, the cross traffic is not within our control and cannot be measured. This means comparisons between the mechanisms are indicative, but less certain. Figure 23a shows a box-and-whisker plot of the Jain's fairness index, which is a measure of the relative share that TCP Cubic flow has with the concurrent DA-LBE flow. The Vegas based DA-LBE mechanism has a slightly lower fairness index, indicating that it yields more to the competing Cubic flow. This is also indicated in Figure 23b which compares completion times in the same experiment. Vegas-based DA-LBE yields more to the competing TCP Cubic (and cross traffic) flows, completing closer to the deadline.

2.4.4 Key findings and implications

In the context of SDN-enabled datacenters the observations and findings of the previous section highlight a key advantage of the developed SDN integration mechanisms for NEAT end-hosts. In real-world deployments different transport layer stacks can be expected to exhibit different performance levels depending on various – potentially unknown – factors. The MPTCP results above motivate the use of a controller that selects the threshold for elephant flows based on the observed performance of the individual protocols for a given network topology as well as the measured network utilisation. More generally, by deploying NEAT at the end-hosts the controller can utilize its global view of the network to distribute policies that govern the selection of the available transport layer stacks and define associated thresholds depending on the current network state. Examples of such policies were first outlined in D3.3, and updated versions are listed in Appendix B, Listings 6 and 7. Using NEAT policies the controller supplies selected clients with: i) an elephant flow threshold, ii) a flow handling approach. As a result, the controller can fine-tune the threshold for different network destinations based on the current utilisation of the relevant network segments. Further, the controller can specify the mechanism to use for elephant flows, e.g., setting a DSCP marking and/or enabling MPTCP as in the current scenario.

For WAN transfers the tests carried out in a real network demonstrate the efficacy of our implementation of the DA-LBE meta-congestion control framework for manipulating the performance of existing congestion control protocols to achieve an LBE service that adapts to an approaching deadline. The DA-LBE mechanism achieves this by inflating the particular network "price" appropriate for the underlying congestion control it utilises: for Vegas queueing delay, and for Cubic the probability of an explicit congestion notification (ECN). Thus the NEAT stack enables the use of LBE flows for non-critical file transfers while still allowing the application to specify deadlines by which the flow should complete.

3 The future of NEAT

The NEAT library is an open source code project that will hopefully be kept alive for a long time by project members as well as external parties. While we cannot know how future coding / "keeping-alive" of the software really will unfold, we can already now critically review possible future outcomes, both in terms of the potential use of the software itself and in terms of the broader architectural implications that the NEAT project has helped the community identify and understand.

For the software itself, we ask ourselves what would happen if the NEAT library becomes widely deployed. We answer questions such as: why would users have an incentive to use less aggressive transport protocols? Conversely, if all applications choose the most aggressive protocol, why is there a need for choice? How does "local" scalability of NEAT play out, in terms of CPU and memory usage? These concerns are addressed in the following, in Section 3.1.

Then we will look at broader implications of NEAT, in line with the ongoing work in the IETF Transport Services (TAPS) Working Group. For instance, not even considering the specific piece of software that the project has developed, there could be opportunities to use project outcomes in Android and Apple; indeed, the TAPS work on APIs is the first step towards a wider access to the approach, providing opportunities to ease the construction of new applications. NEAT has demonstrated how applications can be developed and then used even on a Raspberry Pi. NEAT was able to leverage advanced WebRTC features, such as ICE and selection of the most appropriate transport protocol, etc.

NEAT outcomes enable future opportunities for next generation networks. The NEAT architecture could allow future Internet applications to take advantage of advanced features to use the techniques and interfaces developed in NEAT. The consistent interface offered by NEAT could enable development of innovative support for new emerging multi-service network technologies and features. It could be used as a new basis to take best advantage of the diverse network capabilities provided by 5G networks (e.g., opportunistic access to very high capacity radio interfaces, ways to optimise how applications interface to specific radio resource capabilities through Network Slicing, etc.).

To better understand the implications of NEAT on the network stack and network application development in general, we discuss the most recent activities in the TAPS Working Group in Section 3.2.

3.1 Scalability

Scalability has multiple facets. If NEAT would be widely deployed, the following problems might arise:

1. Control traffic from Happy Eyeballs (HE) might overwhelm the Internet.
2. Too aggressively transmitted data traffic might overwhelm the Internet.
3. A NEAT client or server might quickly cease to operate as the number of outgoing or incoming requests increases.

We call the first two concerns "global" scalability and the latter "local" scalability. Regarding the first concern in the list, HE is already used in several ways by large-scale production systems: Google's Chrome browser and Mozilla's Firefox browser apply it for QUIC-vs-TCP, and Apple devices have been applying it for IPv6-vs-IPv4, using a sophisticated strategy in line with the description in [14]. In the following two subsections, we will consider concern number 2: protocol aggressiveness. Then, we will offer a short discussion of the third concern, which we call "local" scalability: overhead in terms of CPU and memory usage that is incurred by NEAT on a host.

3.1.1 Towards more aggressive protocols?

When protocols are used to reserve network capacity (e.g., using network signalling to request specific capacity, or to prioritise some traffic over other traffic), it must be ensured that users do not obtain more than they are allowed to consume within the traffic class they use. Additionally, mechanisms need to be in place to police the resources being consumed/reserved to ensure this use is acceptable to other flows that share the capacity. Such resource provisioning issues are *not* the primary focus of the NEAT project. In the following, we therefore assume that any special usage of NEAT (e.g., to provide capacity reservations) is handled by existing methods (such as the provisioning methods used in SDN, or through the use of pre-provisioned DiffServ classes).

The focus of NEAT is therefore on sharing within a capacity class, where the potential fairness problem that may arise from aggressive behavior in the network is in fact a *congestion control* concern much more than it is a protocol concern (see the SCTP vs. TCP example in the next subsection). RFC 6077 [31] defines congestion control as a "(typically distributed) algorithm to share network resources among competing traffic sources".

Such sharing should be efficient and fair, for some reasonable definition of fairness. Indeed, a single flow that does not implement any form of congestion control can completely push out any other flows that share a bottleneck with it; in the 1990's, this problem has led to concerns about a global congestion collapse—a "tragedy of the commons" as the result of every actor selfishly moving

towards more and more aggressive congestion control. This culminated in a proposal to mandate "TCP-friendliness" (which was then a commonly accepted notion in the IETF for a while): never sending more than a conforming TCP implementation under the same circumstances [20].

The notion of TCP-friendliness became questioned in the early 2000's with the advent of congestion control mechanisms that were not TCP-friendly, yet avoided congestion collapse by falling back to a TCP-compatible behavior when the packet loss ratio becomes high. In 2004, the "Binary Increase Control" (BIC) algorithm achieved unexpected fame via a mistake in a press release[4], and shortly after, the BIC congestion control algorithm became the default mechanism in Linux. The default congestion control choice in Linux was updated to "CUBIC" congestion control in 2006. Interestingly, this change was made although (or because?) CUBIC is less aggressive than BIC [26]. CUBIC backed off in response to congestion by multiplying the sender's congestion window (cwnd) by 0.8 at first (instead of the standard TCP's value of 0.5). This factor was later adjusted down to 0.7, leading to a less aggressive behaviour (see [27] for an in-depth study of the choice of back-off factor). Put together with the table of other Linux updates in [22], this forms ample evidence of a move towards less, not more aggressive congestion control, for CUBIC congestion control updates in the Linux kernel.

Meanwhile, the very idea of "TCP-friendliness" as a fairness notion has been questioned [13], and congestion control mechanisms that seek to minimize queuing delay while trying to compete with TCP have been proposed for production use [24, 25, 42]. Reducing latency has become a much more important goal than reducing the completion time of long transfers (i.e., being aggressive) for many common use cases, and "Lower-than-Best Effort" (LBE) mechanisms have been deployed that are extra-cautious (e.g., LEDBAT has been used for years in BitTorrent and for Apple's Operating System updates) [36]. Such mechanisms serve a user's own interest in minimal disruption of higher-priority flows on a shared bottleneck. For example, once a video application has a sufficient amount of data buffered, it might want to move to an LBE-like congestion control for more buffering in order to preserve its own ability to execute high priority actions (like changing videos in reaction to user interactions) without fighting bufferbloat. Also, certain data transfers are known by the application to be large but not urgent, making the transport system work better if these flows are sent in the background. This concerns Operating System update downloads and background file system synchronization applications such as DropBox, for example.

To summarize: despite a concern from approximately 15-20 years ago [20] that congestion control mechanisms would move towards more and more aggressive behavior, this has in fact not happened—the incentives do not appear such that more aggressive is strictly better. The primary goal of modern congestion controls is to either be more efficient in aggregate (and thus grow the pie of what is possible) or address severe shortcomings of legacy mechanisms (such as TCP Reno's inability to use all the available bandwidth for flows of typical durations). Protocols that simply shift bandwidth onto their own streams in a zero-sum (or worse) game are truly not in vogue at all.

3.1.2 Why is there a need for choice?

Some of the trade-offs that NEAT makes are entirely different from the trade-offs of more vs. less aggressive congestion control. For example, the SCTP protocol is generally slightly less efficient than TCP due to missing hardware support, but it may be faster for applications that can accept messages arriving out-of-order (because TCP does not support out-of-order message delivery, it produces head-of-line blocking delay for such applications). Yet, the congestion control used by both protocols is the

[4] http://www4.ncsu.edu/~rhee/export/bitcp/bicfaq.htm

same—hence, the trade-off of choosing SCTP vs. TCP is orthogonal to the trade-off of using more vs. less aggressive congestion control.

Applications, especially web applications, have already been working with different attributes of transports for many years. HTTP initially used one TCP flow per request (there can be 100 requests on a typical page) because of the convenient stream metaphor. However the interaction with the TCP transport was bad—a typical flow terminated before it ever got out of slow start and performance was dominated by the latency of handshakes and growing the congestion window—in effect congestion control was rarely applied and the effective throughput was often less than the available bandwidth. High latency problems are often addressed via parallelism and HTTP did this by using several TCP streams in parallel.

The advent of HTTP/2 changed the landscape by bringing many of those flows into a single TCP flow and congestion context with framing for the individual flows inside the TCP application data. The aggregation of that data has better congestion control properties in the sense that HTTP/2 is more akin to an elephant flow than the set of old HTTP mice flows it replaces. However it stood at a disadvantage in one way—the single flow's initial window is smaller than the aggregate set of initial windows it replaces.

Data at the time showed the median web flow to terminate with a congestion window of about 30 segments while a TCP flow was starting with an initial window of just 3. As a result the initial value was raised to 10 in conjunction with lowering the overall number of flows with a net positive impact to the overall environment.

The value 10 appears to be a good general choice today; it is likely to become too small again within some years. Whether 10 is a good or bad value depends on various factors—e.g., the potential usage of pacing, and typical connection speeds, which differ widely between the various countries of the world. Given that applications generally change faster than the operating system, they will always need to be able to configure the system for optimal performance, depending on their specific needs. NEAT gives applications a large number of knobs without requiring application programmers to re-invent the wheel for common mechanisms that the transport layer itself can and should take care of (e.g., performing "happy eyeballs" to discover which protocol is available along an end-to-end path).

3.1.3 Local scalability

We evaluated happy eyeballs on the server side between TCP and SCTP in an already published and reported paper [32]. Our results showed that, although HE increases CPU load as compared with a single TCP or SCTP connection establishment, the increase is in the order of 10% for 35 KiB Web objects, i.e., fairly typical Web objects, and is even smaller in those cases where HE takes place between TLS-encrypted connections. Moreover, our investigation in [32] showed that the caching of connection-request results substantially reduces the CPU load due to protocol racing, especially in comparison with the cost of TLS.

Complementing these earlier results, this section now elaborates on the resource utilization of a client-side NEAT system during connection establishment.[5] This has been done by comparing a NEAT application with an identical application implemented with libuv [2] or kqueue [30]. Since libuv builds upon kqueue and NEAT builds upon libuv, this comparison illustrates the direct overhead of using the NEAT library compared to a more low-level approach. Strictly speaking, libuv vs. NEAT shows

[5] Note that some related data, from tests in the MONROE testbed, are also reported in Table 6 in Section 2.1 (Celerway's use case). The overhead numbers in this table were also obtained at the client side, but these measurements were not limited to connection setup.

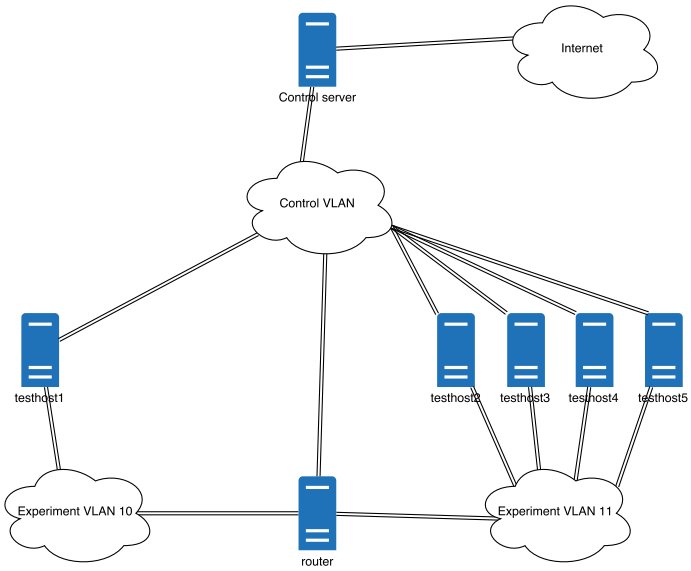


Figure 24: The TEACUP testbed at UiO.

the overhead of NEAT. However, including kqueue results enables us to understand how the resource overhead of a callback-based approach like libuv compares to a simple event handler like kqueue. This gives a better context to understand the resource overhead of NEAT compared to libuv. Also, these libraries impose an increased level of abstraction: libuv provides platform-independence, while NEAT extends upon this with protocol-independence. A comparison of the three enables us to see the overheads to provide these services.

We have considered CPU usage and memory usage of such applications as metrics for evaluating the resource overhead since these metrics directly affect the scalability of a NEAT-based application.

Figure 24 illustrates the experimental network testbed setup we have used for our experiments. We use TEACUP (TCP Experiment Automation Controlled Using Python) to control our experiments, which enables us to easily configure and automate the experiment runs [6]. TEACUP only supports running simple TCP experiments by default, and does not support running custom traffic generators like a NEAT application. We therefore extended the TEACUP code with the support for custom traffic generators and custom loggers. This enabled us to test the scalability and resource overhead of NEAT as the number of flows increases.

TEACUP is installed on the control server and is responsible for establishing SSH connections to the test hosts to start and stop traffic generators, traffic sinks and loggers in the experiments. It is also responsible for configuring the router with the correct shaping, scheduling, policing and dropping rules for the experiments. TEACUP is controlled by a configuration file which among other things controls which test hosts should be used as traffic generators and traffic sinks, and how parameters should be varied for each experiment run. Note that TEACUP only communicates with the test hosts over the control VLAN, and does not interfere with the experimental traffic sent on the experiment VLANs.

Table 23 presents the relevant hardware comprising the test hosts of the TEACUP testbed deployed at UiO. In our experiments we ran the server application on one test host and the client application on another. The router was configured to shape the bandwidth at 10 Mbit/s for both of the outbound

Hardware type	Model
Machine	HP Compaq 8100 Elite CMT
CPU	Intel Core i7-870 @ 2.93 GHz
RAM	4 x Samsung 4GB PC3-10600 DDR3-1333MHz
NICs	2 x Intel I210 Gigabit Network Connection

Table 23: The hardware components of the TEACUP testbed test hosts.

interfaces facing the experiment networks. Data was also delayed for 50 ms in both egress interfaces of the router using *netem* [3] to emulate a RTT of 100 ms. Socket buffers at the endpoints (test hosts) were set sufficiently large so that the router can become the bottleneck without any limitation of the data rate at the endpoints.

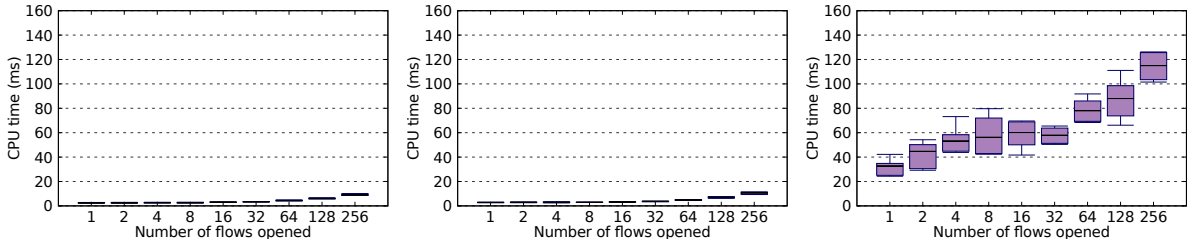
Unless noted otherwise, all the following graphs in this section show minimum, 10th percentile, median, 90th percentile, and maximum values to give an indication of the data distribution. All the results are measured on the client-side, and we evaluated CPU usage and memory usage for different numbers of flows to determine if the resource usage is linearly or exponentially increasing with the number of flows. We considered 1, 2, 4, 8, ..., 256 opened flows in our experiments, and all the experiments were run 10 times.

CPU usage Here we consider the accumulated CPU time spent when establishing connections using NEAT, libuv and kqueue. In the case of NEAT, we measure the CPU time spent from the point *neat_init_ctx* is issued until all flows have successfully connected and *on_connected* has been called for each of the flows[6]. In the case of libuv, we measure the CPU time spent from the point the *uv_loop_init* call is issued until all flows are writable and the associated callbacks have been called. In the case of kqueue, we measure the CPU time spent from the point *kqueue* is issued to create the kqueue until all the flows have connected and *kevent* has returned successfully with each of the flows marked writable. Alternatively we could start to measure the data right before we start the event loop, but then we would not measure the overhead of issuing *connect* for all flows in the cases of libuv and kqueue.

We consider the CPU time of all processes in the system that contribute to the global CPU usage and that are affected by the NEAT-, libuv- and kqueue-based applications. We have considered the *"kernel"* and *"rand_harvestq"* processes in addition to the application itself, and calculated the sum of the *user* and *system* CPU times for these processes in order to find the accumulated CPU usage of these processes at a given point in time. By calculating the difference of the accumulated CPU times for two specific points in time, we find the CPU time spent in an interval. This enables us to benchmark the CPU usage of a particular part of the code. We used the *clock_gettime* function to sample the current accumulated CPU time of a particular process.

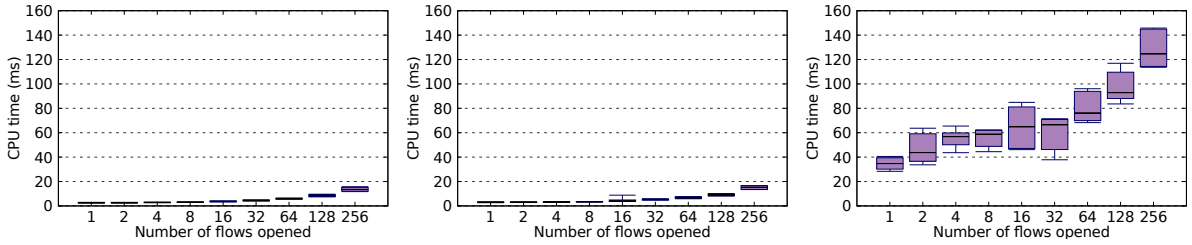
Figure 25 and Figure 26 show that the CPU usage of kqueue and libuv are almost identical when opening connections, and that the overhead of using NEAT is quite large by comparison. The reason why libuv and kqueue results are very similar is that libuv only provides a simple abstraction layer over the kqueue event loop. However, NEAT is an advanced library with many more features and components, which increases the CPU time.

[6]For details on the NEAT API calls, see [28, Appendix B].



(a) kqueue (b) libuv (c) NEAT

Figure 25: The CPU time spent when establishing connections using TCP at the client-side. The RTT of the link was 100 ms and the bandwidth of the link was shaped at 10 Mbit/s at the bottleneck router.



(a) kqueue (b) libuv (c) NEAT

Figure 26: The CPU time spent when establishing connections using SCTP at the client-side. The RTT of the link was 100 ms and the bandwidth of the link was shaped at 10 Mbit/s at the bottleneck router.

In order to understand the reasons behind higher CPU usage of the NEAT application, we performed CPU profiling using the `Callgrind` tool of `Valgrind`. This profiling tool enables us to check how many CPU cycles each function of the application is using, and also includes the number of CPU cycles imposed by all other functions called from within the function. It can also profile the CPU cycles of loaded shared libraries, which enables us to measure the CPU cycles for every function of the NEAT Library. Normally, the shared libraries are loaded on-demand during runtime, but to exclude the CPU cycles from this loading process from our results, we enabled the `LD_BIND_NOW` environment variable to load all shared libraries before application startup. It is important to note that `Callgrind` significantly slows down the execution of the application because different counters are increased during runtime. CPU time and CPU cycles are also different metrics, but the number of CPU cycles gives a good indication about the bottleneck in the code.

	kqueue	**libuv**	**NEAT**
`main`	1,122,359	1,343,781	63,804,570
`start_event_loop`	589,677	–	–
`neat_start_event_loop`	–	–	32,782,775
`uv_run`	–	702,909	32,782,728
`on_connected`	573,952	615,936	807,680

Table 24: Total number of CPU cycles executed by various functions in the kqueue, libuv and NEAT client applications. Here we show the total number of CPU cycles executed when opening 256 TCP flows for a specific test run. (Note that we have enclosed the kqueue event loop in a separate function `start_event_loop` to make it more comparable to NEAT and libuv.)

Function	CPU cycles
neat_open	22,889,216
neat_set_property	7,005,184
neat_new_flow	702,464
neat_init_ctx	260,664
neat_set_operations	39,936

Table 25: Total number of CPU cycles executed by some of the functions offered by the NEAT API outside the event loop. The results are based on opening 256 TCP flows on the client-side. This means that the total for e.g. neat_open is the total CPU cycles spent when calling neat_open 256 times.

Function	CPU cycles
he_connected_cb	17,823,232
nt_send_result_connection_attempt_to_pm	16,121,856
nt_resolver_literal_timeout_cb	11,641,344
open_resolve_cb	10,925,056
uvpollable_cb	1,251,584
nt_connect	689,920

Table 26: Total number of CPU cycles executed by some of the internal NEAT functions inside the NEAT event loop. The results are based on opening 256 TCP flows on the client-side. Some of the listed functions overlap in the sense that one function eventually calls the other, e.g., he_connected_cb calls nt_send_result_connection_attempt_to_pm.

As can be seen in Table 24, NEAT uses a lot more CPU cycles than kqueue and libuv both within the event loop and outside the event loop. By digging into the profile results, we can determine the parts of the code that are responsible for this high usage. Table 25 shows the CPU cycles for the NEAT functions called outside the event loop. It shows that the functions that are called 256 times in our example scenario have the largest overhead (neat_open, neat_set_property, neat_new_flow). When we dig into these functions, we find that almost all of the CPU usage is derived from calling various JSON operations from the libjansson library which is used by NEAT internally to handle NEAT properties. For instance, nt_json_send_once which is called from send_properties_to_pm in neat_open uses 8,683,264 CPU cycles to convert the JSON properties to a string for all of the flows, and 6,300,672 CPU cycles are spent in neat_set_property to convert the string representation of the NEAT properties to JSON objects.

Table 26 lists some of the internal NEAT functions with the largest CPU overhead. By analyzing the profile results, we divide the high-cost functions into *pollable* functions and *timer* functions. The overhead of *polling* occurs when the he_connected_cb internal callback is called, which is done whenever the Happy Eyeballs candidate has connected. Every time this happens, an attempt is made to push the results of the connections to the CIB of the Policy Manager by using a JSON string. As can be seen from the table, 16,121,856 CPU cycles are needed for this operation. The overhead of *timer* functions is related to address-to-name resolving and gathering of interface information for every source-destination pairs for all flows.

Compared to the "pure" connection setups of kqueue and libuv, NEAT carries out a quite large amount of extra work. If the goal were to optimise timing, some functionality could be removed in

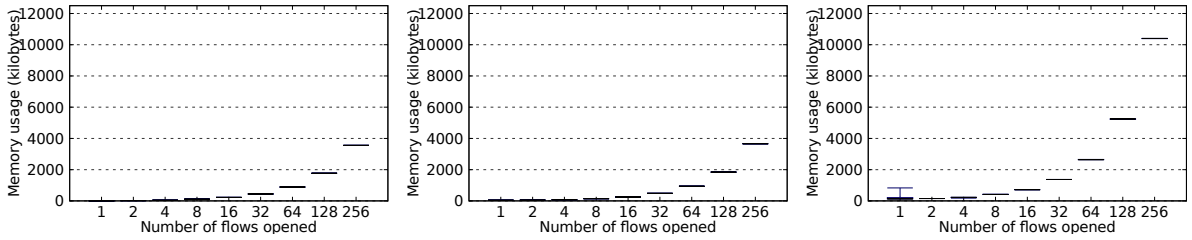
exchange for faster operation, and the code could be optimised in various ways (e.g., by caching some more data—the results of queries that are made repeatedly in these tests). In particular, we have found that the JSON operations and communication with the Policy Manager introduce a lot of overhead for NEAT. In an optimised version, JSON could be replaced with a binary representation such as CBOR [10], and the methods for communicating between the NEAT Framework and the Policy components could be improved with caching, shared-memory, etc. Either way, the NEAT library is a prototype, not production-ready code, and the primary goal was rich functionality rather than speed.

Memory usage Here we consider the memory impact of establishing connections in NEAT, libuv and kqueue. We sample memory data at the same places in the code as described in the above paragraph about CPU usage, and the graphs show how much the memory usage has increased during connection establishment. We measure the Resident Set Size (RSS) of the applications, which gives the total number of bytes that are currently in physical memory for the process. Since modern computers rarely need to move RAM memory into swap memory, the RSS gives a good indication of the actual memory usage of an application. To sample the RSS we issue the command *top -o rss -p <pid>* where *<pid>* is the process ID of the application. Similar to what was done for profiling CPU usage, the results are based on opening 256 TCP or SCTP flows on the client side.

Figures 27 and 28 show the memory impact of connection establishment for NEAT, libuv and kqueue using either TCP or SCTP. Like the CPU results, the kqueue and libuv results are for the most part identical, and NEAT introduces some overhead, but this overhead is proportionally not as large as in the CPU results. To better understand what part of the NEAT code introduces this memory overhead, we performed memory profiling using the `Massif` tool of `Valgrind`. This tool enables us to measure the current heap usage, stack usage, or total virtual memory page usage throughout the execution of the application. In our investigation we considered the heap usage of NEAT because we find that this gives a good indication of the memory usage and can be compared to the graphs showing the increase in RSS.

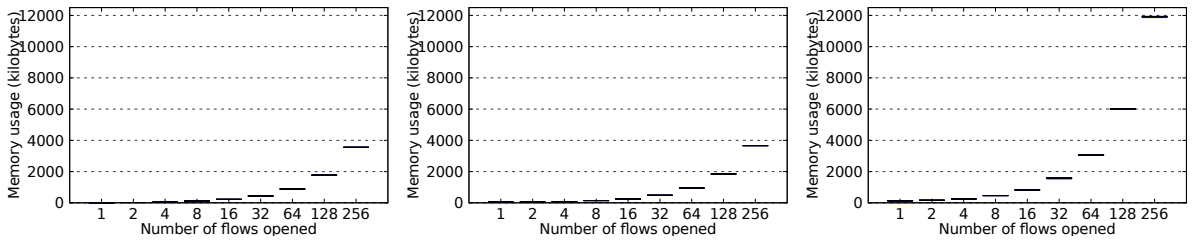
In libuv, we find that about 75% of the memory is allocated in `on_connected` while about 25% of the memory is allocated in `main`. The total heap usage is approximately 3.4 MiB. In NEAT, we find that about 29% of the memory is allocated in `on_connected`, 9% allocated in `main`, and that about 62% of the memory is allocated either within the NEAT event loop or within the functions of the NEAT API. 25% of the total memory consumption is related to allocating memory for the platform- and protocol-independent representation of network sockets for all Happy Eyeballs candidates internally in NEAT (*struct neat_pollable_socket*) in the `open_resolve_cb` function. Also, about 25% of the total memory consumption is related to allocating such internal socket representations when calling `neat_open` for all of the flows. The remaining memory usage seems to be related to various JSON allocations, allocation for the *neat_flow* structures, etc. The total memory usage was sampled to be about 8.9 MiB.

The memory overhead of NEAT does not seem to be very large considering that the library must keep information about various platform- and protocol-specific details that are not stored in the simple kqueue and libuv applications we have used throughout our comparisons. Also, as memory is becoming increasingly cheap, memory consumption is rarely the bottleneck of an application. Potential improvements to the memory usage in NEAT could be to divide the internal socket structures into several smaller structures that are related to specific protocols or specific platforms.



(a) kqueue (b) libuv (c) NEAT

Figure 27: The increase in application memory consumption when establishing connections using TCP at the client-side. The RTT of the link was 100 ms and the bandwidth of the link was shaped at 10 Mbit/s at the bottleneck router.



(a) kqueue (b) libuv (c) NEAT

Figure 28: The increase in application memory consumption when establishing connections using SCTP at the client-side. The RTT of the link was 100 ms and the bandwidth of the link was shaped at 10 Mbit/s at the bottleneck router.

3.2 Evolution towards a standard API with implementation guidance

NEAT has successfully shown that we can modularise the stack to avoid ossification so that there are no longer strong barriers to evolution. We have shown how the API can be raised, what is needed and what can now be introduced into this ecosystem to enable evolution of the way stacks operate and how a new design can be used to integrate with changes in the network.

Four components can optimise the matching of application requirements to network capabilities/characteristics of the service:

1. Application developers could know that they need relatively little capacity, but know performance benefits significantly from lower latency. A user may have a preference to lower cost for a video they wish to watch—for instance postponing a large download until a lower cost service is available. Maybe a user can access a corporate network for high-speed access, but only restricted to specific uses. NEAT brings a new API that allows applications/users to express their requirements or expectations to the endpoint protocol stack, so the stack becomes aware of how the application will use the network.

2. The second change is to allow the stack to choose between multiple mechanisms. The choice could include transport protocols, features of transport protocols, or the availability of the network to offer paths with lower latency, higher throughput, better resilience, etc.

3. No single solution can provide the best match between the network and the application. This requires policy to inform the selection and to decide between alternate ways of using the network.

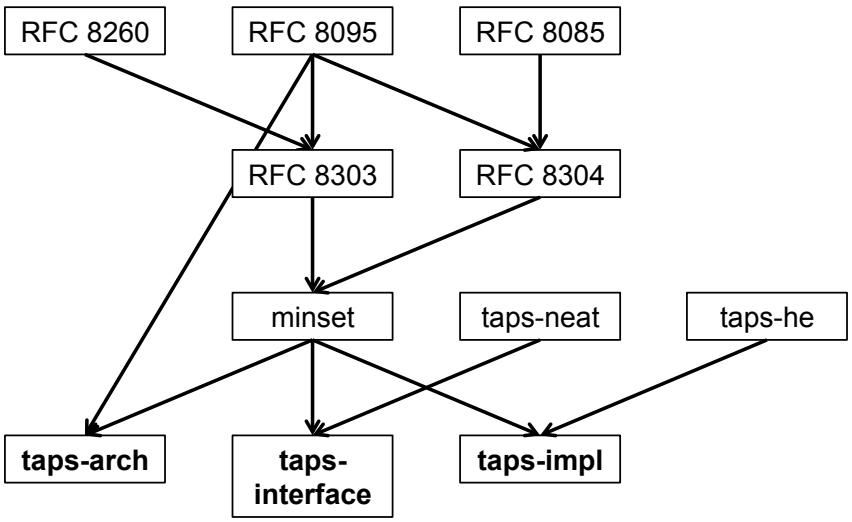


Figure 29: Dependencies of NEAT (co-)authored TAPS documents. Abbreviated names: minset = draft-ietf-taps-minset [38], taps-neat = draft-fairhurst-taps-neat [16], taps-he = draft-grinnemo-taps-he [21], taps-arch = draft-pauly-taps-arch [34], taps-interface = draft-trammell-taps-interface [37], taps-impl = draft-brunstrom-taps-impl [14].

4. The final component is how the stack interfaces to the network. Future networks will provide a much wider range of network services. Stacks can discover the capabilities of a range of network services that are currently available. In the future, some networks will provide specific features on demand—e.g., using SDN techniques, this can provide more than one different network service over a single interface.

NEAT has resulted in a shift in the view of people in the community. The initial perspective of the IETF was that the work was exploratory, and was interesting research. This started with a Charter for the IETF TAPS working group to publish a small number of RFCs. As the momentum grew, there was growing consensus that TAPS was an important area of work, and the number of Chartered RFCs grew. The outputs of the TAPS WG have now been embraced by multiple players and finally are resulting in Internet Proposed Standards that can be used by industry.

The most recent TAPS proposal for the interface that the Internet's transport layer should expose is described in draft-trammell-taps-interface [37], together with its companion documents which describe the underlying architecture [34] and give guidance on how to implement a transport system that offers this interface [14]. At the IETF-101 meeting in London, on March 2018, the TAPS Working Group expressed its clear consensus to adopt these three documents. NEAT has had a significant impact on these documents, both directly (by co-authoring them) and indirectly (by writing Internet-drafts that have by themselves influenced these three documents). Figure 29 gives an overview of the role that NEAT (co-)authored documents have played in the development of the three planned final documents of TAPS: RFCs 8095 [19], 8303 [39] and 8304 [15] constitute the first step in the charter of the TAPS Working Group, which was to analyse existing transport protocols. Following this first step, TAPS intends to define a minimal set of Transport Services that end systems should support; this is covered by the NEAT-authored "minset" document [38] and a security-related companion document [33] that is not written by NEAT participants and is therefore not shown in Figure 29.

When it comes to finally specifying the abstract interface and giving guidance on how to implement it, then this specification is—in line with the TAPS charter—naturally based on the "minset"

document. Care was taken to ensure that the interface in [37] can support all services defined in the "minset", and the implementation guidance in [14] refers to the mapping from services to protocols that the "minset" has identified for several functions. Moreover, the latter document explains how to implement "Happy Eyeballs" (called "racing" in [14]); as this explanation in [14] was developed, it was decided to abandon the "taps-he" document [21] and instead re-use its text in [14]. Similarly, the "taps-neat" document [16] was abandoned but had a very immediate influence on [37].

NEAT is currently the only known open source implementation of a TAPS-conforming transport system. While the interface that is described in [37] offers a slightly higher abstraction layer than the NEAT API, NEAT provides all the technical building blocks that are needed to implement a system that offers this more abstract interface. Thus, implementing a shim layer that maps the interface in [37] to NEAT should be extremely easy; to put "easy" into perspective, we judge the complexity and effort to be approximately equal to the amount of work that is usually done in the course of a master thesis.

4 Conclusions

In this report we presented key outcomes of WP4 with a focus on Task 4.3. Building upon previous WP4 work and the test plan proposed in Deliverable D4.2 [11] this deliverable has addressed the validation and evaluation of NEAT by means of experiments, carried out as part of the developed industrial use cases.

The document provides a detailed overview of the experimental work carried out, including: experimental setups and topologies, how the test plan was actually implemented, and a summary of the most important results and their significance. The presented results demonstrate the feasibility of the NEAT approach in realistic environments with a clear mapping to use cases relevant to the core business of the involved industry partners.

Finally, the report offers an analysis of the future evolution of NEAT with an emphasis on scalability issues — both in end-hosts running the NEAT stack and in regards to the impact that wide adoption of NEAT might have on the Internet. Moreover, we discussed the influence of the work carried out in the project on IETF standardisation efforts, in particular on the efforts by the TAPS working group to develop a future standard transport API.

References

[1] D-ITG (Distributed Internet Traffic Generator). [Online]. Available: http://traffic.comics.unina.it/software/ITG/

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

[3] NetEm - Network Emulator. [Online]. Available: http://man7.org/linux/man-pages/man8/tc-netem.8.html

[4] Open vSwitch. [Online]. Available: https://www.openvswitch.org/

[5] OpenDaylight. [Online]. Available: https://www.opendaylight.org/

[6] TCP Experiment Automation Controlled Using Python (TEACUP). [Online]. Available: http://caia.swin.edu.au/tools/teacup/

[7] J. Ahrenholz, "Comparison of core network emulation platforms," in *Military Communications Conference, 2010-MILCOM 2010*. IEEE, 2010, pp. 166–171.

[8] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, ser. SIGCOMM '08. New York, NY, USA: ACM, 2008, pp. 63–74. [Online]. Available: http://doi.acm.org/10.1145/1402958.1402967

[9] D. Anipko (Ed.), "Multiple Provisioning Domain Architecture," RFC 7556 (Informational), RFC Editor, Fremont, CA, USA, pp. 1–25, Jun. 2015. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7556.txt

[10] C. Bormann and P. Hoffman, "Concise Binary Object Representation (CBOR)," RFC 7049 (Proposed Standard), RFC Editor, Fremont, CA, USA, pp. 1–54, Oct. 2013. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7049.txt

[11] Z. Bozakov, A. Brunstrom, D. Damjanovic, G. Fairhurst, A. F. Hansen, T. Jones, N. Khademi, A. Petlund, D. Ros, T. Rozensztrauch, M. I. S. Bueno, D. Stenberg, M. Tüxen, and F. Weinrank, "Final version of NEAT-based tools," The NEAT Project (H2020-ICT-05-2014), Deliverable D4.2, Sep. 2017.

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

[13] B. Briscoe, "Flow rate fairness: Dismantling a religion," *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 2, pp. 63–74, Apr. 2007.

[14] A. Brunstrom (Ed.), T. Pauly (Ed.), T. Enghardt, K.-J. Grinnemo, T. Jones, P. Tiesel, C. Perkins, and M. Welzl, "Implementing interfaces to transport services," Internet Draft draft-brunstrom-taps-impl, work in progress, Feb. 2018. [Online]. Available: https://tools.ietf.org/html/draft-brunstrom-taps-impl

[15] G. Fairhurst and T. Jones, "Transport Features of the User Datagram Protocol (UDP) and Lightweight UDP (UDP-Lite)," RFC 8304 (Informational), RFC Editor, Fremont, CA, USA, pp. 1–20, Feb. 2018. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8304.txt

[16] G. Fairhurst, T. Jones, A. Brunstrom, and D. Ros, "The NEAT interface to transport services," Internet Draft draft-fairhurst-taps-neat, work in progress, Nov. 2017. [Online]. Available: https://tools.ietf.org/html/draft-fairhurst-taps-neat

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

[18] G. Fairhurst, T. Jones, M. Tuexen, and I. Ruengeler, "Packetization layer path mtu discovery for datagram transports," Working Draft, IETF Secretariat, Internet-Draft draft-ietf-tsvwg-datagram-plpmtud-01, March 2018, http://www.ietf.org/internet-drafts/draft-ietf-tsvwg-datagram-plpmtud-01.txt. [Online]. Available: http://www.ietf.org/internet-drafts/draft-ietf-tsvwg-datagram-plpmtud-01.txt

[19] G. Fairhurst (Ed.), B. Trammell (Ed.), and M. Kuehlewind (Ed.), "Services Provided by IETF Transport Protocols and Congestion Control Mechanisms," RFC 8095 (Informational), RFC Editor, Fremont, CA, USA, pp. 1–54, Mar. 2017. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8095.txt

[20] S. Floyd and K. Fall, "Promoting the use of end-to-end congestion control in the internet," *IEEE/ACM Transactions on Networking*, vol. 7, no. 4, pp. 458–472, Aug 1999.

[21] K.-J. Grinnemo, A. Brunstrom, P. Hurtig, N. Khademi, and Z. Bozakov, "Happy Eyeballs for transport selection," Internet Draft draft-grinnemo-taps-he, work in progress, Jun. 2017. [Online]. Available: https://tools.ietf.org/html/draft-grinnemo-taps-he

[22] S. Ha, I. Rhee, and L. Xu, "Cubic: A new tcp-friendly high-speed tcp variant," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, pp. 64–74, Jul. 2008. [Online]. Available: http://doi.acm.org/10.1145/1400097.1400105

[23] D. Hayes, D. Ros, A. Petlund, and I. Ahmed, "A framework for less than best effort congestion control with soft deadlines," in *Proceedings of IFIP Networking*, Stockholm, Jun. 2017, pp. 1–9. [Online]. Available: http://dl.ifip.org/db/conf/networking/networking2017/1570334752.pdf

[24] S. Holmer, H. Lundin, G. Carlucci, L. D. Cicco, and S. Mascolo, "A Google Congestion Control Algorithm for Real-Time Communication," Internet Engineering Task Force, Internet-Draft draft-ietf-rmcat-gcc-02, Jul. 2016, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-rmcat-gcc-02

[25] I. Johansson and Z. Sarker, "Self-Clocked Rate Adaptation for Multimedia," RFC 8298 (Experimental), RFC Editor, Fremont, CA, USA, pp. 1–36, Dec. 2017. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8298.txt

[26] D. D. Kashif Munir, Michael Welzl, "Linux beats windows! - or the worrying evolution of TCP in common operating systems," in *Proceedings of the International Workshop on Protocols for Fast*

Long-Distance Networks (PFLDnet '07). Marina Del Rey (Los Angeles), California, USA: ENS Lyon, February 2007, pp. 43–48. [Online]. Available: http://www.welzl.at/research/publications/pfldnet2007.pdf

[27] N. Khademi, G. Armitage, M. Welzl, S. Zander, G. Fairhurst, and D. Ros, "Alternative backoff: Achieving low latency and high throughput with ECN and AQM," in *IFIP Networking 2017 Conference and Workshops (Networking'2017)*, Stockholm, Sweden, Jun. 2017.

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

[29] D. J. Leith, R. N. Shorten, G. Mccullagh, L. Dunn, and F. Baker, "Making available base-RTT for use in congestion control applications," *IEEE Communications Letters*, vol. 12, no. 6, pp. 429–431, Jun. 2008.

[30] J. Lemon, "Kqueue – a generic and scalable event notification facility," in *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, 2001, pp. 141–153.

[31] D. Papadimitriou (Ed.), M. Welzl, M. Scharf, and B. Briscoe, "Open Research Issues in Internet Congestion Control," RFC 6077 (Informational), RFC Editor, Fremont, CA, USA, pp. 1–51, Feb. 2011. [Online]. Available: https://www.rfc-editor.org/rfc/rfc6077.txt

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

[33] T. Pauly, C. Perkins, K. Rose, and C. A. Wood, "A Survey of Transport Security Protocols," Internet Engineering Task Force, Internet-Draft draft-pauly-taps-transport-security-02, Mar. 2018, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-pauly-taps-transport-security-02

[34] T. Pauly (Ed.), B. Trammell (Ed.), A. Brunstrom, G. Fairhurst, C. Perkins, P. Tiesel, and C. Wood, "An architecture for transport services," Internet Draft draft-pauly-taps-arch, work in progress, Feb. 2018. [Online]. Available: https://tools.ietf.org/html/draft-pauly-taps-arch

[35] P. Pfister, E. Vyncke, T. Pauly, and D. Schinazi, "Discovering provisioning domain names and data," Working Draft, IETF Secretariat, Internet-Draft draft-ietf-intarea-provisioning-domains-01, February 2018, http://www.ietf.org/internet-drafts/draft-ietf-intarea-provisioning-domains-01.txt. [Online]. Available: http://www.ietf.org/internet-drafts/draft-ietf-intarea-provisioning-domains-01.txt

[36] D. Ros and M. Welzl, "Less-than-best-effort service: A survey of end-to-end approaches," *Communications Surveys Tutorials, IEEE*, vol. 15, no. 2, pp. 898–908, 2013.

[37] B. Trammell (Ed.), M. Welzl (Ed.), T. Enghardt, G. Fairhurst, M. Kuehlewind, C. Perkins, P. Tiesel, and C. Wood, "An abstract Application Layer Interface to transport services," Internet Draft draft-trammell-taps-interface, work in progress, Mar. 2018. [Online]. Available: https://tools.ietf.org/html/draft-trammell-taps-interface

[38] M. Welzl and S. Gjessing, "A minimal set of transport services for TAPS systems," Internet Draft draft-ietf-taps-minset, work in progress, Feb. 2018. [Online]. Available: https://tools.ietf.org/html/draft-ietf-taps-minset

[39] M. Welzl, M. Tuexen, and N. Khademi, "On the Usage of Transport Features Provided by IETF Transport Protocols," RFC 8303 (Informational), RFC Editor, Fremont, CA, USA, pp. 1–56, Feb. 2018. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8303.txt

[40] P. Wette and H. Karl, "DCT2Gen: A versatile TCP traffic generator for data centers," https://www-old.cs.uni-paderborn.de/fachgebiete/fachgebiet-rechnernetze/people/dr-philip-wette/dct2gen.html, 2014.

[41] J. Y. Yen, "Finding the k shortest loopless paths in a network," *management Science*, vol. 17, no. 11, pp. 712–716, 1971.

[42] X. Zhu, R. Pan, D. M. A. Ramalho, S. M. de la Cruz, P. Jones, J. Fu, and S. D'Aronco, "NADA: A Unified Congestion Control Scheme for Real-Time Media," Internet Engineering Task Force, Internet-Draft draft-ietf-rmcat-nada-06, Dec. 2017, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-rmcat-nada-06

A NEAT Terminology

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Transport Feature Short for Transport Service Feature.

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

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

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

B SDN controller policies for handling elephant flows

```
{
    "uid": "elephant_tag",
    "filename": "elephant_tag.policy",
    "policy_type": "policy",
    "match": {
        "flow_size_bytes": {
            "value": {
                "start": 10000000.0,
                "end": "Inf"
            }
        }
    },
    "priority": 2,
    "properties": [
            {   "elephant_flow": {
                    "precedence": 2,
                    "value": true
                }
            }
    ],
    "replace_matched": false,
    "time": 1493307808.030206
}
```

Listing 6: Controller generated NEAT policy for tagging elephant flows based on flow size.

```
{
    "uid": "elephant_handle",
    "filename": "elephant_handle.policy",
    "policy_type": "policy",
    "match": {
        "elephant_flow": { "value": true }
        }
    },
    "priority": 10,
    "properties": [
            {   "transport": {
                    "precedence": 2,
                    "value": "MPTCP"
                },
                "transport_cc": {
                    "precedence": 2,
                    "value": "olia"
                },
```

```
                "SO/IPPROTO_IP/IP_TOS": {
                    "precedence": 2,
                    "value": 40
                }
            }
    ],
    "replace_matched": false,
    "time": 1493307808.030206
}
```

Listing 7: NEAT policy for handling flows tagged as “elephant flows”.

C How to build and test NEAT applications in MONROE

In the following, we explain how to create a MONROE experiment that can be deployed and run on MONROE nodes that include the NEAT library, making it possible for the experiment to call NEAT API functions. We provide a practical step-by-step example on how to create, test and deploy a NEAT-enabled experiment. The code of our example is available in the neat-monroe git repository, at: https://github.com/NEAT-project/neat-monroe/tree/master/monroe-experiments/neat_test. We also describe the metadata information gathered by MONROE nodes for all available network connections and how this information is made available for experiments and for the NEAT Policy Manager (PM).

C.1 Creating NEAT-enabled MONROE experiments

First, you need to install Docker on your machine. The *MONROE Platform User Manual* recommends installing Docker via an installation script downloaded from the Docker webpage:

```
wget https://get.docker.com -O install.sh
chmod u+x install.sh
./install.sh
```

Test your installation, e.g., with Docker's hello-world example:

```
docker run hello-world
```

Next, download the MONROE base image for an experiment template. The MONROE toolkit for creating experiment images is available from MONROE's GitHub repository. Clone the project with the following command:

```
git clone https://github.com/MONROE-PROJECT/Experiments.git
```

Use the `template` folder located in the `experiments` folder as a base for your image. Copy the folder and save it under your experiment's name:

```
cd Experiments/experiments/
cp -r template neat_test
```

Rename dockerfile `template.docker` to match the experiment's folder name:

```
cd neat_test
mv template.docker neat_test.docker
```

Once your experiment has been prepared, you will need to upload the image to your dockerhub repository to make it available for MONROE certification and deployment. Edit the `push.sh` bash script to point Docker to your experiment's dockerhub repository by editing the corresponding line:

```
CONTAINERTAG=neatuser/neat
```

Next, you need to prepare your experiment binaries. Our example of experiment is built upon a simple HTTP client application (`neat_http_get`) that downloads a file from a specified URL using the NEAT User API. In our experiment we invoke the script `neat_http_get` periodically and we record and store the download time as a result. The source code of `neat_http_get` is available in the neat-monroe git repository: https://github.com/NEAT-project/neat-monroe/tree/master/neat-http-get. It uses `cmake` to build and package the application into a `.deb` file. Compilation of the tool itself is very straightforward, but we need two things to be considered beforehand. First, as already

mentioned, MONROE containers are based on Debian Jessie, so we need to cross compile the application against Debian Jessie. Second, we need the NEAT library to be installed on our development machine.

As a universal solution we can employ a temporary Docker container as a build environment:

```
cd neat-http-get
sudo docker pull monroe/base
sudo docker run -v ${PWD}:/mnt -ti monroe/base bash
```

Then (inside the container) prepare the build environment:

```
echo "deb http://ftp.debian.org/debian jessie-backports main" >> /etc/apt/sources.
    list
apt-get update
apt-get install -y -t jessie-backports git vim build-essential cmake
```

Build and install the NEAT library:

```
apt-get install -y -t jessie-backports libuv1-dev libldns-dev libmnl-dev libjansson-
    dev libsctp-dev libssl-dev
cd /root/
git clone https://github.com/NEAT-project/neat.git
cd neat/
mkdir build
cd build/
cmake ..
make
make install
```

Build and package `neat_http_get`:

```
cd /mnt/
mkdir build
cd build/
cmake ..
make
make package
```

and exit the container.

The resulting package (`neat-http-get_1.0.0_amd64.deb`) must be copied to the experiment's files directory:

```
cp build/neat-http-get_1.0.0_amd64.deb ../monroe-experiments/neat_test/files/
```

Our experiment script (`neat_experiment.sh`) looks as follows:

```
#!/bin/bash

# Run experiment

CMD="/usr/bin/neat_http_get -v 1 celerway.com"

while true; do

    DATE=`date +%Y%m%d-%H%M%S.%N`
```

```
    FNAME=/monroe/results/neat_test-${DATE}.txt
    TMP_FNAME=/tmp/neat_test-${DATE}.txt

    echo -n "/usr/bin/time -f 'TIME-SEC: %e' ${CMD} 1>/dev/null 2> ${FNAME} ..."
    /usr/bin/time -f 'TIME-SEC: %e' ${CMD} 1>/dev/null 2> ${TMP_FNAME}
    mv $TMP_FNAME $FNAME
    echo " DONE"

    sleep 15

done
```

It also must be placed in the `monroe-experiments/neat_test/files/` directory. Once the experiment binaries and scripts are ready, we can start creating the experiment's Docker image. Everything we want to install and/or configure in the image must be specified in the image dockerfile, `neat_test.docker` in our example. The interested reader is referred to the Docker documentation (https://docs.docker.com/engine/reference/builder/) for dockerfile syntax and supported commands. In order to support the NEAT library, the following sections need to be specified in the dockerfile.

Base our image on MONROE base container:

```
FROM monroe/base
```

Add yourself as a maintainer of the image:

```
MAINTAINER name@email.com
```

Presently, the MONROE base image is built on top of Debian Jessie. In order to support the NEAT library — which relies on some newer packages not available in the stable Jessie repository — we need to add the `jessie-backport` repository to `apt/sources.list` in our image:

```
RUN echo "deb http://ftp.debian.org/debian jessie-backports main" >> /etc/apt/
    sources.list
```

Install the necessary Debian packages required to build and run the NEAT library:

```
RUN apt-get update && apt-get install -y \
    git \
    time \
    build-essential \
    cmake \
    libuv1-dev \
    libldns-dev \
    libjansson-dev \
    libmnl-dev \
    libsctp-dev \
    libssl-dev \
    zlib1g-dev \
    libbz2-dev \
    libreadline-dev \
    libsqlite3-dev \
    llvm \
    libncurses5-dev \
```

```
    libncursesw5-dev \
    xz-utils \
    tk-dev \
    && apt-get clean
```

Install the Python version required by NEAT components (e.g., by the Policy Manager):

```
WORKDIR /opt/celerway
RUN wget https://www.python.org/ftp/python/3.5.2/Python-3.5.2.tgz
RUN tar xvf Python-3.5.2.tgz
RUN cd Python-3.5.2 && ./configure --enable-optimizations && make -j8 && make
    altinstall
```

Install Python packages required by the Policy Manager:

```
RUN pip3.5 install netifaces && pip3.5 install aiohttp
```

Download, build and install NEAT project itself:

```
WORKDIR /opt/celerway
RUN git clone https://github.com/NEAT-project/neat.git
WORKDIR /opt/celerway/neat/build
RUN cmake .. && cmake --build . && make install
```

And finally, copy experiment binaries, install them and create the experiment entry point:

```
COPY files/* /opt/celerway/
WORKDIR /opt/celerway
RUN dpkg -i neat-http-get_1.0.0_amd64.deb
ENTRYPOINT ["dumb-init", "--", "/bin/bash", "/opt/celerway/neat_experiment.sh"]
```

The experiment script `neat_experiment.sh` is launched when the container starts.

Now you are ready to test and deploy your experiment. The procedure for testing, approval, certification and deployment for NEAT-enabled experiments is exactly the same as for any other MONROE experiment and is described in detail in the *MONROE Platform User Manual* [7]. It is worth mentioning that the preliminary test of the image can be done locally on your own machine, e.g., by running the following commands:

```
sudo docker run -v /run/shm/myresults:/monroe/results neatuser/neat
```

And to access the container via bash console:

```
sudo docker ps   //->  to get [CONTAINER_ID]
sudo docker exec -i -t [CONTAINER_ID] bash
```

C.2 MONROE metadata, Policy Manager and CIB

The MONROE platform gathers metadata information about each network connection. It makes the metadata available to the experiments by means of ZMQ[8]. Celerway's `neat-metadata-exporter` is a CIB properties provider intended to run inside the experiment's container. It listens for messages coming from a MONROE node on the ZMQ socket, filters and translates the messages to the format expected by NEAT CIB database and forwards them via a Unix socket to the Policy Manager.

[7] https://github.com/MONROE-PROJECT/UserManual
[8] http://zeromq.org

Table 2 (in § 2.1) shows the metadata properties that are currently supported.

The description of the properties and their possible values can be found in the `data-exporter` README, at: https://github.com/NEAT-project/data-exporter/blob/master/README.md. In order to run the Policy Manager and `neat-metadata-exporter` in the experiment's container the following steps are required.

Add the following lines to the dockerfile:

```
RUN apt-get update && apt-get install -y \
  libzmq3-dev \
  libjsoncpp-dev \
  && apt-get clean
WORKDIR /opt/celerway
RUN git clone https://github.com/NEAT-project/neat-monroe.git
WORKDIR /opt/celerway/neat-monroe/metadata-exporter/src/build
RUN cmake .. && make && make install
```

Then, start the Policy Manager and `neat-metadata-exporter`. Both PM and `neat-metadata-exporter` daemons need to be running before the experiment starts. For our tutorial we can simply modify the `neat_experiment.sh` script to add the following lines at the beginning of the script:

```
# Start policy manager
mkdir -p /var/run/neat/cib/
mkdir -p /var/run/neat/pib/
python3.5 /opt/celerway/neat/policy/neatpmd --sock /var/run/neat/ --cib /var/run/
    neat/cib/ --pib /var/run/neat/pib/ &
# Start neat metadata exporter
neat-metadata-exporter --cib-socket /var/run/neat/neat_cib_socket &
```

For the sake of simplicity our example has not optimised the container size. To reduce the final image size all intermediate files should be stripped from the image. Additionally, each command in the dockerfile creates a file system layer that is then downloaded and applied sequentially when preparing the experiment container on the nodes. Therefore, the number of steps in the dockerfile should be kept to a minimum, by combining multiple instructions into a single docker command. Also, instead of installing `dev` packages and building software inside the container, we should install binaries or packages directly. Please refer to the *MONROE Platform User Manual* for additional tips for image optimisation.

Disclaimer