neat

NEAT

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

H2020-ICT-05-2014
Project number: 644334

Deliverable D4.2
Final version of NEAT-based tools

Editor(s):	Zdravko Bozakov
Contributor(s):	Anna Brunstrom, Dragana Damjanovic, Gorry Fairhurst, Audun Fosselie Hansen, Tom Jones, Naeem Khademi, Andreas Petlund, David Ros, Tomasz Rozensztrauch, María Isabel Sánchez Bueno, Daniel Stenberg, Michael Tüxen, Felix Weinrank

Work Package:	4 / Validation and evaluation
Revision:	1.0
Date:	August 31, 2017
Deliverable type:	R (Report)
Dissemination level:	Public

Abstract

This document presents the final versions of developed tools that are instrumental to testing and evaluating various aspects of the NEAT stack. We classify these tools into two categories: 1) *measurement and traffic generation* tools, used to carry out experiments that motivate design choices for the NEAT System, and to assess the behaviour of NEAT applications in different scenarios, and 2) *diagnostics* tools, used to verify correctness of the NEAT System prototype and to debug the behaviour of its components.

Furthermore, the document provides a test plan outlining the testing strategies and environments for the core NEAT library as well as the selected industrial use cases. To this end, we present our environment for performing automated testing of the core NEAT stack and lay out testbeds, test procedures and topologies for the selected industrial use cases.

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

Contents

List of abbreviations

AAA Authentication, Authorisation and Accounting

AAAA Authentication, Authorisation, Accounting and Auditing

API Application Programming Interface

BE Best Effort

BLEST Blocking Estimation-based MPTCP

CC Congestion Control

CCC Coupled Congestion Controller

CDG CAIA Delay Gradient

CIB Characteristics Information Base

CM Congestion Manager

DA-LBE Deadline Aware Less than Best Effort

DAPS Delay-Aware Packet Scheduling

DCCP Datagram Congestion Control Protocol

DNS Domain Name System

DNSSEC Domain Name System Security Extensions

DPI Deep Packet Inspection

DSCP Differentiated Services Code Point

DTLS Datagram Transport Layer Security

ECMP Equal Cost Multi-Path

EFCM Ensemble Flow Congestion Manager

ECN Explicit Congestion Notification

ENUM Electronic Telephone Number Mapping

E-TCP Ensemble-TCP

FEC Forward Error Correction

FLOWER Fuzzy Lower than Best Effort

FSE Flow State Exchange

FSN Fragments Sequence Number

GUE Generic UDP Encapsulation

H1 HTTP/1

H2 HTTP/2

HE Happy Eyeballs

HoLB Head of Line Blocking

HTTP HyperText Transfer Protocol

IAB Internet Architecture Board

ICE Internet Connectivity Establishment

ICMP Internet Control Message Protocol

IETF Internet Engineering Task Force

IF Interface

IGD-PCP Internet Gateway Device – Port Control Protocol

IoT Internet of Things

IP Internet Protocol

IRTF Internet Research Task Force

IW Initial Window

IW10 Initial Window of 10 segments

JSON JavaScript Object Notation

KPI Kernel Programming Interface

LAG Link Aggregation

LAN Local Area Network

LBE Less than Best Effort

LEDBAT Low Extra Delay Background Transport

LRF Lowest RTT First

MBC Model Based Control

MID Message Identifier

MIF Multiple Interfaces

MPTCP Multipath Transmission Control Protocol

MPT-BM Multipath Transport-Bufferbloat Mitigation

MTU Maximum Transmission Unit

NAT Network Address (and Port) Translation

NEAT New, Evolutive API and Transport-Layer Architecture

NIC Network Interface Card

NUM Network Utility Maximization

OF OpenFlow

OS Operating System

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

OVSDB Open vSwitch Database

PCP Port Control Protocol

PDU Protocol Data Unit

PHB Per-Hop Behaviour

PI Policy Interface

PIB Policy Information Base

PID Proportional-Integral-Differential

PLUS Path Layer UDP Substrate

PM Policy Manager

PMTU Path MTU

POSIX Portable Operating System Interface

PPID Payload Protocol Identifier

PRR Proportional Rate Reduction

PvD Provisioning Domain

QoS Quality of Service

QUIC Quick UDP Internet Connections

RACK Recent Acknowledgement

RFC Request for Comments

RTT Round Trip Time

RTP Real-time Protocol

RTSP Real-time Streaming Protocol

SCTP Stream Control Transmission Protocol

SCTP-CMT Stream Control Transmission Protocol – Concurrent Multipath Transport

SCTP-PF Stream Control Transmission Protocol – Potentially Failed

SCTP-PR Stream Control Transmission Protocol – Partial Reliability

SDN Software-Defined Networking

SDT Secure Datagram Transport

SIMD Single Instruction Multiple Data

SPUD Session Protocol for User Datagrams

SRTT Smoothed RTT

STTF Shortest Transfer Time First

SDP Session Description Protocol

SIP Session Initiation Protocol

SLA Service Level Agreement

SPUD Session Protocol for User Datagrams

STUN Simple Traversal of UDP through NATs

TCB Transmission Control Block

TCP Transmission Control Protocol

TCPINC TCP Increased Security

TLS Transport Layer Security

TSN Transmission Sequence Number

TTL Time to Live

TURN Traversal Using Relays around NAT

UDP User Datagram Protocol

UPnP Universal Plug and Play

URI Uniform Resource Identifier

VoIP Voice over IP

VM Virtual Machine

VPN Virtual Private Network

WAN Wide Area Network

WWAN Wireless Wide Area Network

1 Introduction

A goal of the NEAT project is to produce deployable, working code throughout the entire life cycle of the project. All partners have contributed to a common code base to develop the NEAT System and core library defined in WP1 and WP2, as well as extensions in WP3. Best practices have been followed for collaborative code development, including a strong focus on testing and validation. This approach has fostered high interaction between the partners and is expected to yield testable, high quality code.

Work Package 4 is comprised of three tasks that will allow the project to assess and demonstrate the benefits that the NEAT approach brings to networked applications:

- Porting key example applications selected in WP1 to the NEAT architecture and APIs.
- The development of test tools for measurements and performance analysis of the NEAT transport system.
- Experiments using the software developed in the two previous items to evaluate and validate the use cases defined in WP1, leveraging test facilities brought into the project by the partners.

This document reports on activities in WP4 carried out in months 13-30. The main scope of this report is on Task 4.2, focused on: (a) producing final versions of the developed test tools, (b) defining a test plan for validation and performance analysis. The document updates Sections 3 and 4 of Deliverable D4.1 [14].

The rest of this document is structured as follows. In Section 2, we describe the final versions of the auxiliary tools implemented or extended during the project to support different aspects of the development of NEAT. In Section 3 we present the strategy and environment for testing the NEAT software during its development cycle, as well as the test plan for carrying the experiments needed to validate the functionality and performance of the NEAT library in the context of the industry use cases. Finally, Section 4 concludes the document.

2 NEAT-based tools

2.1 Summary of tools

Table 1 summarises the diverse auxiliary software packages, either developed or adapted by the project, used both to test the NEAT stack and its functionalities, and to assist in the design and development process of the NEAT System. *Separate* means a tool is independent from the NEAT library but allows to test some specific aspect of interest to NEAT. *NEAT-based* refers to a software program that is actually built on top of the NEAT library, and that uses the NEAT User API. *Built-in* refers to a tool that has been integrated into the NEAT System. The table also indicates, where relevant, to what *Application Class* each tool belongs to, as defined in the NEAT architecture [17, Figure 11]. As a reminder, Class 0 refers to non-NEAT enabled applications, i.e., apps that do not use the NEAT User API (either directly or via some middleware), whereas Class-1 apps are directly built on top of this API. Class-4 apps make use of Policy and/or Diagnostics interfaces, but do not exchange data via the NEAT User API.

In the following we describe each of these tools and how they are used in NEAT.

Table 1: Software tools produced or modified by the project, for assisting with the design, development and testing of the NEAT System.

Tool category	Tool name	Relation to the NEAT stack	App class
Measurement and Traffic generation	phttpget	Separate	0
	thttpd	Separate	0
	NEAT web server	NEAT-based	1
	edgetrace	Separate	0
	fling	Separate	0
	PATHspider	Separate	0
	tneat	NEAT-based	1
	pReplay	Separate	0
	Nghttp2	NEAT-based	1
	Multi-homed download manager	NEAT-based	0 and 1†
Diagnostics	Buildbots	Separate	N/A
	Logging	Built-in	N/A
	Policy Manager Diagnostics	Built-in	N/A
	System-wide NEAT statistics (neatstat)	Separate	4

† There are in fact two versions of this software used, one NEAT-enabled (class 1), another non NEAT-enabled (class 0).

2.2 Measurement and traffic generation tools

This section presents several measurement and traffic generation tools used to evaluate areas that impact the performance and behaviour of the NEAT system. Such areas include the performance of transport protocols across Internet paths as well as the impact of middleboxes on the used transport protocols. The findings obtained from these tools influence the design of protocol selection mechanisms and associated default policies used by NEAT.

2.2.1 phttpget

URL	https://github.com/NEAT-project/HTTPOverSCTP/tree/multistream
Summary	Minimal HTTP client with pipelining support, extended by the project to support SCTP. Used to analyze the benefits and disadvantages of using multiple SCTP streams instead of multiple TCP connections.

Phttpget [6] is a minimal HTTP client with pipelining support. We utilised a modified version of the tool in combination with pReplay (§ 2.2.6) and thttpd (§ 2.2.2), to analyze the benefits and disadvantages of using multiple SCTP streams instead of multiple TCP connections, as reported in D3.1 [18, Section 3.1.1].

Phttpget is included in FreeBSD where it is used for the internal update mechanism and the portsnap tree[1]. The tool takes a remote server name and a list of one or more files as a command-line argument and downloads these files into the current directory. We chose phttpget because of its small

[1] https://www.freebsd.org/doc/handbook/ports-using.html

size of about 500 lines of code and its support for HTTP pipelining. Without HTTP pipelining, the client can only have a single request in flight and has to receive the server's response completely before sending the next request. HTTP pipelining allows the client to have multiple requests in flight without waiting for the server's response.

We extended phttpget with SCTP support, an interface to handle requests from external applications, and a statistics mechanism to track request and response handling in detail [4]; an example log is provided in Listing 1. In addition to reading a list of files given via command-line arguments, phttpget opens a named pipe where external applications can request files from the web server via phttpget. If phttpget uses a SCTP connection with multiple streams, every request is scheduled to a different stream and—in contrast to TCP pipelining—only one request per SCTP stream is in flight. The request-to-stream mapping allows the server to send its responses to the client without worrying about reordering. This solves the head-of-line blocking issue of TCP's pipelining where the responses have to arrive in the same order they have been sent by the client.

After phttpget receives the server's response, it reports detailed information about the request: http status code, payload- and header-size and the time span between request and response. If the response has been scheduled by an external application, phttpget sends this information via the named pipe to the application.

```
[    0.000002][PRG] Settings - timeout : 15
[    0.000165][PRG] Settings - protocol : SCTP
[    0.000253][PRG] Settings - pipelining : enabled
[    0.000344][PRG] Settings - max SCTP streams : 6
[    0.004580][PRG] interactive mode - reading from PIPE
[    0.047509][PRG] #1 - 404 - /pages/amazon.es/www.amazon.es/index.html - sctp sid: 0
[    0.049941][PRG] #2 - 200 - /missing/www.amazon.es/ajax/getntent.html - sctp sid: 0
[    0.050521][PRG] #3 - 200 - /missing/ecx.images-amazon.com/images/I/41H.jpg - sctp sid: 1

...

[    0.273344][PRG] #67 - 200 - /pages/amazon.co.uk/images/I/51wQhdhxVTL._SL135_.jpg - sctp sid: 0
[    0.275770][PRG] ###### STATS ######
[    0.275917][PRG]     requests      : 67
[    0.275996][PRG]     - # 200       : 41
[    0.276079][PRG]     - # 404       : 26
[    0.276157][PRG]     - # other     : 0
[    0.276235][PRG]     bytes header  : 16918
[    0.276313][PRG]     bytes payload : 531176
```

Listing 1: Log file from phttpget loading a sample website over SCTP.

2.2.2 thttpd

URL	https://github.com/nplab/thttpd/tree/multistream
Summary	Modified thttpd [9] web server with SCTP support. This includes the mapping of multiple http requests to distinguish SCTP streams over a single SCTP association.

The thttpd web server [9] is a small and well tested web server written in C which runs on multiple platforms. In addition to the extensions and changes reported in Deliverable 3.1 [18], we extended thttpd with HTTP pipelining and new SCTP features. Due to its support of TCP, SCTP and SCTP via UDP, thttpd is an important part of our automated testing environment (§ 2.3.1) and is used for several measurements like the comparison of HTTP pipelining via TCP and SCTP.

2.2.3 NEAT web server

URL	https://github.com/NEAT-project/neat/blob/master/examples/server_http.c
Summary	Simple NEAT-enabled web server supporting HTTP 1.1 and TLS.

In addition to the basic HTTP behavior of closing the transport connection after every finished response, this simple, NEAT-enabled web server also supports the Keep-Alive extension allowing to respond to several requests without closing the transport connection after every request, resulting in less overhead and a faster page load time.

The server supports all reliable transport protocols which are offered by the NEAT library and is part of the automated continuous integration system (§ 3.1), where it delivers static websites as well as large binary data blobs.

In addition to several tests where we compared the performance and benefits of the particular transport protocols for HTTP transfers, we also used this web server to analyze the benefits of transparent flow mapping for NEAT, described in Deliverable D3.2 [19].

2.2.4 Middlebox measurement tools

In today's Internet we see an increasing deployment of middleboxes. While middleboxes provide in-network functionality that is necessary to keep networks manageable and economically viable, any packet mangling—whether essential for the needed functionality or accidental as an unwanted side effect—makes it more and more difficult to deploy new protocols or extensions of existing protocols.

In this section we describe three tools, edgetrace, fling and PATHspider, developed to assess the behaviour and impact of middleboxes on transport protocols and to identify the deployment status of middleboxes in the Internet. The findings obtained using these tools directly benefit the decision-making process in NEAT. They enhance our understanding of path reachability in the following scenarios and help with developing necessary fallback mechanisms in NEAT:

- IPv4 vs. IPv6 with SCTP/UDP or native SCTP.
- QUIC (with both IPv6 and IPv4).
- Different DSCP values (more specifically with the two previous scenarios).
- ECN header bits.

edgetrace

URL	https://trace.erg.abdn.ac.uk/
Summary	Tool to measure DSCP mark survivability from multiple network edges to a single point in the core of the Internet.

Edgetrace is a tool to measure DSCP mark survivability from multiple network edges to a single point in the core of the Internet. Generating a diverse set of network paths requires edgetrace to be run by people from their own machines from their homes, local coffee shops and their places of work. Edgetrace is run by volunteers from their personal machines, this places tight restrictions on what the tool is able do while performing measurements.

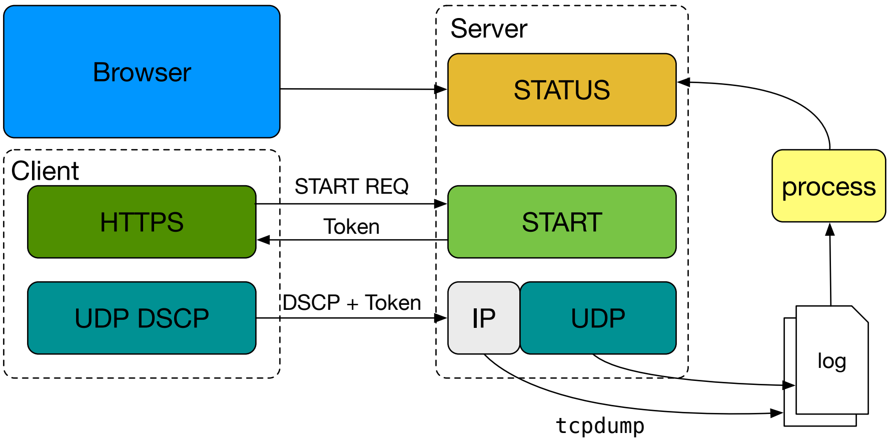


Figure 1: Edgetrace architecture.

There are two components to edgetrace, a very simple client and a server backend, both written in the Go programming language. The client was made available as a single self-contained binary to increase the chance of the tool being run.

The server performs two roles, first it generates sessions for the client to use and acts as an end point for UDP datagrams, second it performs a packet capture collecting IP fields not made available to the application in userspace. The server generates files that make it possible to match received datagrams to sessions, the capture log makes it possible to match sent DSCP values to received DSCP values. The session makes it possible to detect network black holes.

The client portion of edgetrace connects to the backend server over https and requests an object describing a session ID. This session creation mechanism allows the client to do connectivity assessments to the endpoint server. The client tool is expected to be run on networks where the captive portal might not have been passed yet, hence, edgetrace attempts to detect common network events when trying to access a network with a captive portal.

Edgetrace retrieves the session ID from the server over HTTPS, the session ID is used to detect unique runs of the tool. Each UDP packet edgetrace sends contains the session ID object with the addition of initial TTL, original DSCP mark, operating system and a free-form string passed by the user describing the network type. Edgetrace will then send out 10 packets for each DSCP mark to be tested, with packets passed out at a rate of 5 packets per second.

Edgetrace was used to perform an edge measurement campaign from March to August 2017. The measurement campaign was able to assess how DSCP marked traffic is treated from edge connections, 180 user-driven measurements were performed using the edgetrace tool. Data from the edgetrace experiment has been combined with other measurements and is being prepared for publication.

fling

URL http://fling-frontend.nntb.no

Summary When developing protocol extensions, it is often important to know what will happen to particular types of packets along a path (e.g. "If we add an option to this packet, is it more likely to be dropped? Will the option often be removed?"). Fling allows for experimentation of various protocols of interest. fling transmits packets from a pcap file, following a dialogue defined in a JSON file, between fling clients and servers.

"Flexible ping" (fling) is a tool developed in the NEAT project that aims to address the need to know exactly what middleboxes do. It combines the advantages of its prior work. Like TCPExposure [8], it can carry out a dialogue between a server and a client and identify what middleboxes have done to the packets belonging to the dialogue. Like Netalyzr [22], it allows to update the measurements with a server-side only change; no new software installation is required. Also, like Tracebox [16], it can detect which device along the path changed or dropped a packet. The fling tool does not need to be modified to carry out new measurements, and its flexibility allows for a variety of different experiments to be run.

Fling provides a capability to inject specific arbitrary packet formats between specific endpoints, allowing protocol interactions to be studied. Fling is best suited to specific tests evaluating whether new packet formats and new protocol features would be supported end-to-end. The preliminary results from fling measurements have motivated in NEAT the work on developing a application-level QoS fallback mechanism referred to as "Happy Apps" in Deliverable D2.3 [21], as we found that non-zero DSCP values may provoke consistent packet loss when traversing certain middleboxes.

A fling client is a static piece of software; it begins a test by pulling a test description from the server, which it then executes. A test description is comprised of a *pcap* file containing the test packets and a JSON file describing the test, specifically packet types, information about header fields, and the sending/receiving sequences of the dialogue. It never contains addresses: a fling client always only talks to a specified fling server configured through a command-line parameter. This allows to fully control the dialogue and collect measurement results at the server for research use; it also serves as a security measure, by ensuring that attackers cannot design tests that would turn fling clients into sources of traffic towards some other hosts in the network. To avoid getting in the way of normal Internet usage of fling users, the total maximum number of packets transmitted by fling is also statically configurable by the client. Appendix B provides an example of a fling JSON file.

We have prototyped fling in Python, based packet capturing and manipulation on Scapy [7], and prepared tests for 36 distinct protocols and protocol options. By preparing a test, we mean that we have generated all needed packet traces to test a protocol or an option. Our preliminary evaluation of fling included using traces we gathered from different vantage points with the participation of 34 users (providing 3384 tests in total). The evaluation results of these tests are presented in [12] (see Appendix D). The general goal of our preliminary evaluation was not to make general statements about the support of specific protocols in the Internet, but rather to examine fling's flexibility. The preliminary results in [12] show that fling can perform all of the tests provided by prior work while requiring no software update in order to perform new tests, making fling flexible to deploy. These results also include a variety of protocol tests over IP, with changes applied to the IP header (e.g., testing options or unknown protocol numbers) and the TCP header (e.g., testing options or using a wrong value in the Data Offset field).

We also conducted a larger-scale measurement study of various protocols and protocol options over both IPv4 and IPv6 networks. The study over IPv4 networks was done with 120 hosts from Ark[2], 50 hosts from Planetlab and 30 hosts from NorNet[3] using 35 fling servers whereas with IPv6 networks, 60 hosts from Ark and 15 hosts from NorNet were employed to use 22 fling servers. Interestingly, our findings come in contrast with many of the currently-held beliefs about which protocol(s) are likely to work across Internet paths. For example, we found that SCTP passes on virtually all measured paths. Our findings give fresh insights about a set of protocols and options that are safe to be used opportunistically. We believe that this will inform and have an impact on several ongoing debates at the IETF and beyond. Fling is available from http://fling-frontend.nntb.no for public use.

PATHspider

URL	http://pathspider.net
Summary	PATHspider is a framework for performing and analyzing easily customized A/B tests. PATHspider makes it easy to run large scale measurements on the Internet, with plugins available to support a large range of protocols. PATHspider has been developed by the MAMI EU project.

PATHspider [23] has been developed by the MAMI EU project[4] and is publicly available from https://pathspider.net. NEAT is using PATHspider to generate measurement traces. In contrast to fling, which has a client and a server component, PATHspider experiments are initiated by a client side only, enabling large scale measurements of arbitrary Internet destinations.

For the evolution of the protocol stack, it is important to know which network impairments exist and potentially need to be worked around. While classical network measurement tools are often focused on absolute performance values, PATHspider performs A/B testing between two different protocols or different protocol extensions to perform controlled experiments of protocol-dependent connectivity problems as well as differential treatment. PATHspider is a framework for performing and analyzing these measurements, while the actual A/B test can be easily customized.

PATHspider automates single-sided measurements of implemented protocols to very large number of target servers to evaluate whether the combination of paths and servers support particular concrete protocol features. PATHspider therefore is most suited to questions relating to whether features have been widely deployed in the general Internet. It is being used in NEAT to evaluate support for datagram network and transport options across the Internet.

The architecture of PATHspider is depicted in Figure 2. For each target hostname and/or address, with port numbers where appropriate, PATHspider enqueues a job, to be distributed amongst the worker threads when available. Each worker performs one connection with the “A” configuration and one connection with the “B” configuration. The “A” configuration will always be connected first and serves as the base line measurement, followed by the “B” configuration. This allows detection of hosts that do not respond rather than failing as a result of using a particular transport protocol or extension. These sockets remain open for a post-connection operation.

Packets are separately captured for analysis by the observer. First, the observer assigns each incoming packet to a flow based on the source and destination addresses, as well as the TCP, UDP or SCTP

[2] http://www.caida.org/projects/ark/.
[3] https://www.nntb.no/.
[4] https://mami-project.eu

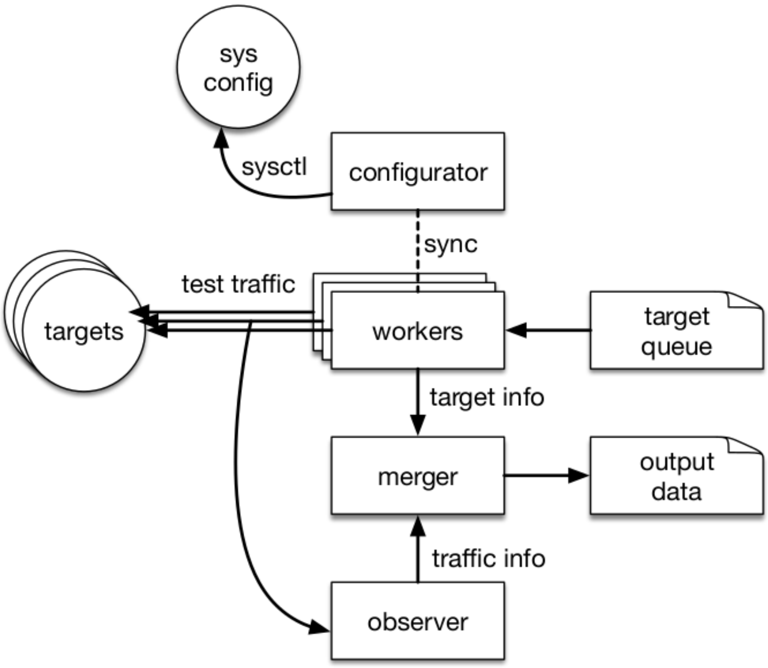


Figure 2: PATHspider architecture.

ports when available. The packet and its associated flow are then passed to a function chain. The functions in this chain may be simple functions, such as counting the number of packets or octets seen for a flow, or more complex functions, such as recording the state of flags within packets and analysis based on previously observed packets in the flow. For example, a function may record both an ECN negotiation attempt and whether the host successfully negotiated use of ECN.

A function may alert the observer that a flow should have completed and that the flow information can be matched with the corresponding job record and passed to the merger. The merger extracts the fields needed for a particular measurement campaign from the records produced by the worker and the observer.

Middlebox tool comparison Edgetrace is a specialised tool designed to measure a single case, UDP DSCP mark survivability from network edges. Due to the need to be run from the edge of the network, edgetrace has to be a paired down tool that is easy to run. Both fling and PATHspider are more flexible and general tools, but require much more user setup to run.

Fling and PATHspider are complementary tools, targeting different locations for tool endpoints (i.e., what paths are tested). PATHspider provides data about the support for existing transport mechanisms: ECN, UDP, DSCP, etc., in the general Internet, identifying in-network impairments that require the endpoints to choose alternate transport mechanisms. On the other hand, fling allows for testing of novel protocol mechanisms and whether these are supported in a specific context.

PATHspider has already been used to identify increased support for transparent passing of the ECN code points through the Internet, and support by web servers when requested (results made available via the MAMI project [25]). On the one hand, PATHspider experiments will continue to monitor support for ECN within the network. On the other hand, Fling can be used to determine if a specific end-to-end path can take advantage of this ECN support, i.e., can the method be used at endpoints, would the network appropriately handle advanced ECN (e.g., the proposed Accurate ECN feedback extension [15]). In this way, PATHspider provides statistical data, and fling detailed understanding.

Fling has been used to identify a pathological case where some places black-hole specific DSCP values. These findings were published in [11] (see Appendix E). Recent tests using a plug-in developed for PATHspider followed up with large-scale measurements from the core to the Alexa top 1 million web servers. In the path from core to server there was evidence that the same pathology could occur, but that more than 99.9% of tested paths and servers showed no black-holing. The conclusion is that this particular impairment likely only occurs in the access network or particular pieces of deployed equipment, and generally would not be normally expected. Further fling experiments may help pinpoint more examples of the pathological cases where this does occur. This data can enable NEAT to decide whether the anomaly is sufficiently important to justify the overhead of code that counters black-holing for DSCPs.

2.2.5 tneat

URL	https://github.com/NEAT-project/neat/blob/master/examples/tneat.c
Summary	Benchmarking and traffic generation tool testing a wide variety of NEAT features.

Tneat is a benchmark tool written against the NEAT library which focuses on bandwidth measurements and message handling for several flows. It has been specifically developed targeting the NEAT library and covers several test applications: traffic generation, flow measurements and library operation analysis. It either operates as a saturated data source or as a data sink.

The tneat sender transmits a predefined number of messages with a predefined size to the receiver and measures the time span between sending the first data and the callback event that all data has been sent for every flow. On the receiver side, the application measures the time span between receiving the first and the last data. When a run has finished, both sides report statistics: run time, average bandwidth and number of receive/send calls; the statistics are recorded for every flow. Besides the number of flows, message size, amount of messages and maximum run time, the user can specify a property which influences the behaviour of NEAT. By default tneat uses TCP and SCTP as transport stacks, both with the same priority. Tneat's use-cases are: analyzing throughput performance, determining internal bottlenecks and testing buffering mechanisms. Table 2 lists tneat's supported command-line options.

Tneat also supports a self-contained mode where it opens a server and a client simultaneously and communicates via loopback with itself. By covering the client- as well as the server-side in one application, tneat is optimal for automated continuous integration tests (§ 3.1).

2.2.6 pReplay

URL	https://github.com/NEAT-project/pReplay-public
Summary	HTTP request generator, simulating the typical request behaviour of a web browser loading a website. The tool supports TCP and SCTP.

PReplay is a HTTP request generator which walks through a dependency graph to simulate the HTTP-request behaviour of a web browser loading a website.

Table 2: tneat command-line options.

Parameter	Short description
`-l`	message length in bytes (sender)
`-L`	local run (server and client on localhost)
`-n`	number of messages to send (sender)
`-p`	listening / connecting port (receiver/sender)
`-P`	NEAT properties (receiver/sender)
`-T`	runtime limit (receiver/sender)
`-v`	log level (receiver/sender)

In addition to the features reported in Deliverable 3.1 [18], we improved the SCTP support by adding an interface to use phttpget as an alternative to libcurl. The user can choose between libcurl and phttpget via a command-line argument. When using phttpget, pReplay runs a phttpget instance in a separate thread and both applications communicate via named pipes. PReplay sends requests to phttpget and after the requests have been handled by phttpget, it reports some information back to pReplay. This includes the HTTP return code and the size of the responses header and payload.

PReplay is currently being utilized to analyse the impact of transport protocol features like multi-streaming on HTTP performance.

2.2.7 Nghttp2

URL	https://github.com/nplab/nghttp2/tree/weinrank/neat
Summary	Modified Nghttp2 [5] web server, using the NEAT library to deliver websites via HTTP2 over TCP and SCTP.

The Nghttp2 [5] library is a popular HTTP/2 implementation included in a number of software projects such as cURL and the Apache Web server to provide HTTP/2 functionality. We ported the web server and client distributed as part of the Nghttp2 project to make use of the NEAT library. The main motivation for this effort was the need for tools enabling the evaluation of the performance of HTTP/2 web traffic on top of the various transport protocol stacks provided by the NEAT System.

As the client and server applications were already event-loop driven, porting did not require major modifications to the core architecture of the applications. Using the NEAT User API allowed us even to reduce the lines of code significantly, by about 20% for each application. The two applications remain fully interoperable with regular TCP-based implementations, while being able to take advantage of NEAT functions.

2.2.8 Multi-homed TCP-based download manager

URL	https://www.neat-project.org/resources/#tools
Summary	Multi-homed TCP-based download manager allowing to demonstrate the throughput benefits of using NEAT's Policy Manager (PM).

Table 3: Celerway's 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

Applications running in a multi-homed environment have the possibility to choose an interface to use for their networking needs. The problem they face, though, is that usually they have no knowledge to make a reasonable choice. Typically, they use the default interface or rely on the user to choose one.

For the purposes of testing its use case, Celerway has developed a simple multi-homed, TCP-based download manager. This tool is a C++ application that uses the NEAT library[5], takes advantage of the NEAT PM and selects the interface that best suits its requirements. The tool is written with mobile broadband networks in mind and expects Celerway CIB properties to be co-installed on the host.

The interface selection is based on the properties requested by the user in command-line arguments. For instance, the user can request the tool to prefer LTE connections over WCDMA (3G) or GSM (2G) by providing the *mode* property (see Table 3 for the list of properties). The tool outputs average download speed and makes it possible to compare interfaces with various characteristics. The tool will be made available as an application in the NEAT implementation in MONROE as described in § 3.2.1.

2.3 Diagnostics tools

There are several mechanisms for diagnostics and runtime analysis of the NEAT stack, outlined below. In addition to the Buildbot system, which is used to identify compile time issues, the NEAT library provides logging facilities at both the level of individual NEAT applications and the Policy Manager, and the neatstat tool provides system-wide statistics. These mechanisms enable application developers and NEAT contributors to analyse choices made by NEAT throughout the lifetime of a NEAT connection.

2.3.1 Buildbots

To ensure a high code quality, we use the Buildbot Continuous Integration Framework [1]. The Buildbot framework is a Python-based application suite which runs on multiple platforms and architectures, this includes Linux, NetBSD, MacOS X and FreeBSD on Intel and ARM systems. The Buildbot Master is the central controller which schedules build tasks to the Build Slaves and displays the collected results on a website.

This automated Continuous Integration Framework gives the NEAT developers instant feedback about changes in the source code. Details about the test procedures and setup are given in Section 3.1.

[5] For the experiments in Celerway's use case testing (§ 3.3.3), a non-NEAT enabled version is also used, for comparison purposes.

Table 4: NEAT log levels.

Value	Short description
NEAT_LOG_OFF	Log system completely deactivated
NEAT_LOG_ERROR	Show NEAT internal errors
NEAT_LOG_WARNING	Same as previous one + warnings
NEAT_LOG_INFO	Same as previous one + informational messages
NEAT_LOG_DEBUG	Same as previous one + verbose debug messages

2.3.2 Logging

The NEAT library includes a logging mechanism to give users and developers a detailed insight about its internal operations. Table 4 shows the five log-levels of the NEAT library, from completely disabling the log messages to a verbose debug output. Log messages begin with the elapsed time since the NEAT library has been initialized, followed by the event level and the event message.

NEAT applications can switch the log-level at any time by calling the `neat_log_level` function[6], this is useful to debug only specific sections of a NEAT application like connection establishment or data transmission. The log messages displayed in the terminal have different colors, depending on their level, allowing the user to quickly distinguish informational messages from errors or warnings. By default, all log messages are written to *stderr*, the application can change this behavior and write all messages into a separate log-file by calling `neat_log_file`[7] at runtime.

In addition to controlling the log system at runtime, NEAT offers a compile-time option to enable or disable the log system.

```
[   0.000836][INF] Available src-addresses:
[   0.000846][INF]  IPv4: 127.0.0.1/0
[   0.000860][INF]  IPv6: fe80::1/0 pref 4294967295 valid 4294967295
[   0.000872][INF]  IPv6: ::1/0 pref 4294967295 valid 4294967295
[   0.000883][INF]  IPv6: 2a02:c6a0:4015:10::117/0 pref 4294967295 valid
4294967295
[   0.000940][INF]  IPv6: fe80::a00:27ff:fed9:53c8/0 pref 4294967295 valid
4294967295
[   0.000952][INF]  IPv4: 212.201.121.117/0
[   0.000963][DBG] neat_run_event_cb
[   0.000977][DBG] neat_new_flow
[   0.001059][DBG] neat_set_property
[   0.001136][DBG] neat_set_operations
[   0.001147][DBG] updatePollHandle
[   0.001158][DBG] neat_open
```

Listing 2: Log output of an example NEAT-enabled application.

2.3.3 Policy Manager Diagnostics

The PM, comprised of the PIB and CIB components, allows application developers and administrators to express intricate relationships between policies and system characteristics using a relatively simple domain-specific language encoded using JSON. Nevertheless, the interdependencies across user generated policy and CIB source entries may give rise to a considerable level of complexity or result in unintended behaviour, e.g., in the case of misconfigurations. To enable the debugging of such

[6] http://neat.readthedocs.io/en/latest/neat_log_level.html
[7] http://neat.readthedocs.io/en/latest/neat_log_file.html

Table 5: REST API endpoints for external access to the Policy Manager.

REST resource	HTTP method	Description
`/pib`	GET	lists all policies installed in the host
`/pib/{uid}`	GET/PUT	retrieve or upload a policy with a specific Unique Identifier (UID)
`/cib`	GET	lists all CIB nodes installed in the host
`/cib/{uid}`	GET/PUT	retrieve or upload a CIB node with a specific UID
`/cib/rows`	GET	retrieve all rows of the CIB repository

scenarios, the PM exposes interfaces for querying the internal state of the PIB and CIB repositories. Administrators can use the interface to insert or delete entries into the PM repositories. The interfaces are implemented as REST endpoints listening on port 45888. Administrators may then access the addresses listed in Table 5 using HTTP's GET/PUT semantics.

In addition to debugging, these interfaces provide external entities (such as SDN controllers) access to the PM.

Furthermore, the PM provides granular logging capabilities allowing users to trace which profile, CIB information and policy rules are used in the construction of candidates throughout the stages of the PM lookup process (described in Deliverable D2.3 [21]). The PM also performs syntax and consistency checks for externally loaded JSON inputs. Finally, the PM provides an interactive debug mode which can be used to inspect the state of the PIB, CIB and associated components. Listing 3 depicts an excerpt of the interactive PM output including some debugging information.

2.3.4 System-wide NEAT statistics (neatstat)

The NEAT Flow Endpoint Statistics [21] can provide a NEAT application with information about the NEAT instance created for the caller. A server, however, might have a large number of NEAT instances using protocol stacks from both kernel and userspace. Server operators and administrators need a way to collect global statistics across all NEAT instances on a machine in order to plan for scalability or to debug misbehaving applications. The *neatstat* tool provides such functionality.

In order to provide a platform-independent interface, neatstat receives data from the PM of core statistics collected from all the NEAT instances through CIB sources. The output of the tool is aggregate and/or per-instance statistics that can aid in understanding the system-wide dynamics of the NEAT instances. Examples of output are:

- Global number of open flows.
- Global bytes sent and received.
- Interface, protocol, and stack information for each flow.
- Remote endpoint(s) for each flow.
- Port numbers and state of the flow.

```
[INF]: Loading policy pib/example/test333.policy...
[DBG]: Using selector: KqueueSelector
[DBG]: Use Ctrl-\ to enter interactive debug mode.
Initializing REST server on 0.0.0.0:45888
Waiting for PM requests on /Users/bozakz/.neat/neat_pm_socket ...
Notifying controller at http://httpbin.org/post (repeat in 82s)
[DBG]: announce addr: 10.73.64.61:59234
^\
##################################### ENTERING INTERACTIVE DEBUG MODE #####################################

use Ctrl-D to exit debug mode
>>> pib.dump()
================================================ PIB START ================================================
  0. 23423     (remote_ip|10.1.23.45)  >>  [[capacity|10000.0-100000.0]]==[(transport|UDP), (transport|TCP)
       ]==[(MTU|9000,1500)]
  0. test2.pol (remote_ip|10.1.23.45)  >>  [[capacity|10000.0-100000.0]]==[(transport|UDP), (transport|TCP)
       ]==[(MTU|9000,1500)]
  0. test333   (remote_ip|10.1.23.45)  >>  [[capacity|10000.0-100000.0]]==[(transport|UDP), (transport|TCP)
       ]==[(MTU|9000,1500)]
  2. tcp options (is_wired|True)--(MTU|1500.0-9000.0)  >>  [(TCP_window_scale|True)]
  5. bulk_transfer (data_volume_gb|2.0-100000.0)--(remote_ip|10.1.23.45)  >>  [(transport|UDP), (transport|
       TCP,SCTP,DCTCP)]==[[capacity_gb|1.0-10.0]]==[(bulk_data|True)]==[(MTU|9000,1500)]
  5. google_domain (domain_name|www.google.com,www.google.de)  >>  [[interface|en7,en0]]
  5. tcp_opt   (remote_ip|10.1.23.45)  >>  [[capacity|10000.0-100000.0]]==[(transport|UDP), (transport|TCP)
       ]==[(MTU|9000,1500)]
  5. test.policy (remote_ip|10.1.23.45)  >>  [[capacity|10000.0-100000.0]]==[(transport|UDP), (transport|TCP
       )]==[(MTU|9000,1500)]
 10. sctp      (transport|SCTP)  >>  [[SCTP_DISABLE_FRAGMENTS|0]]==[[SO_SNDBUF|4096]]==[[transport|SCTP
      ].]==[[SCTP_NODELAY|False]]
100. mptcp_so (transport|MPTCP)  >>  [[SO_MPTCP_ENABLED|True]]
100. tcp_so    (transport|TCP)  >>  [[TCP_NODELAY|False]]
================================================= PIB END =================================================
>>> cib.dump()
================================================ CIB START ================================================
  0. [capacity|10000]--[interface|en0]--[local_ip|10.2.2.2]--(MTU|50.0-9000.0)--[remote_ip|10:54:1.23]--[
       remote_port|80]--[transport|TCP].--(utilization|0.63)
  1. [capacity|10000]--[interface|en0]--[local_ip|10.2.1.1]--(MTU|50.0-9000.0)--[remote_ip|10:54:1.23]--[
       remote_port|80]--[transport|TCP].--(utilization|0.63)
  2. [capacity|10000]--[interface|en0]--[local_ip|10.2.2.2]--(MTU|50.0-9000.0)--[remote_ip|10:54:1.23]--[
       remote_port|80]--[transport|TCP].--(utilization|0.63)
  3. [capacity|10000]--[interface|en0]--[local_ip|10.2.1.1]--(MTU|50.0-9000.0)--[remote_ip|10:54:1.23]--[
       remote_port|80]--[transport|TCP].--(utilization|0.63)
  4. (capacity|1000)--[remote_ip|10:54:1.23]--[remote_port|80]--[transport|TCP].--(utilization|0.63)
  5. [capacity|1000]--[interface|en7]--[local_ip|10.73.64.110]--(MTU|1500)
  6. [capacity|1000]--[interface|en1]--[local_ip|10.3.1.1]--(MTU|1500)
  7. [capacity|100]--[interface|en0]--(MTU|1500)
================================================= CIB END =================================================
>>>

###################################### EXITING INTERACTIVE DEBUG MODE #####################################
```

Listing 3: PM diagnostics output.

3 Test plan

This section describes a test plan targeting both the core library of the NEAT System and the complete industrial use cases. It is important to follow a clear strategy for testing the core library during the development cycle, since we aim to attract a community of NEAT users and developers. To achieve this goal, it is vital to provide a fully functioning and reliable framework.

The second focus of the test plan is the use cases defined in WP1 [17] that demonstrate the NEAT core library from WP2 [21], the extended transport system and protocol enhancements from WP3 [18, 19], and ported applications in a number of scenarios with differing requirements (see Deliverable D4.1 [14]). The use cases build upon the tested core library, they require a higher level test plan to generate measurement results and verify the expected benefits or performance improvements from NEAT.

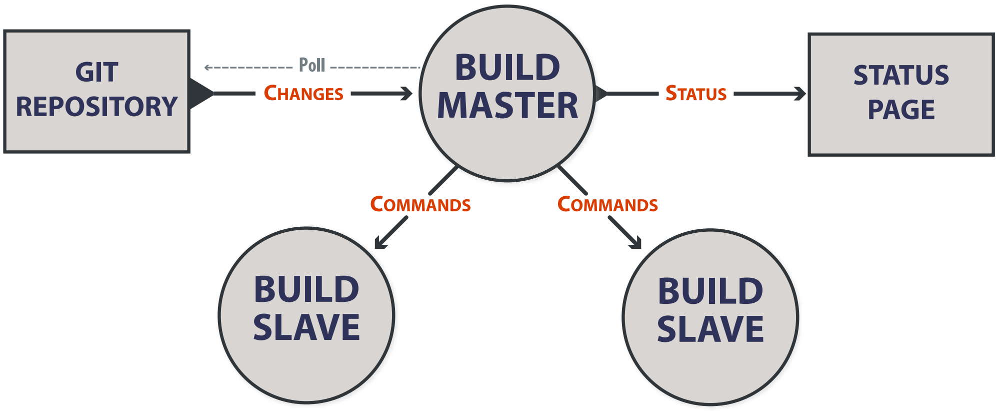


Figure 3: Buildbot system architecture overview (adapted from: http://docs.buildbot.net).

3.1 Core library testing

3.1.1 Test strategy

Automated testing is a core activity of any agile development methodology. It provides a quick feedback response to the developer teams about the health of the application. Automated tests need to be executed continuously, should be fast and test results should be consistent and reliable. In order to achieve these, most of the verifications happen as part of new features development. Quality should be considered from the beginning by ensuring that what is being developed works and that it has not broken any existing functionality.

NEAT aims to follow a fully automated software development processes. Automating the build and test procedures allows NEAT contributors to obtain immediate feedback on the impact of their work to the existing code base. Tests are run on multiple platforms, ensuring that code changes made on one platform do not cause failures on other platforms. Furthermore, the approach can highlight uses of platform-specific code which can be deterrent to the portability of the NEAT System. The automation stack also handles the time-consuming process of generating packaged software releases for multiple platforms, including signing builds and quality-assurance checks. Specifically, NEAT uses Buildbots (introduced in Section 2.3.1) to run the required automation efficiently. FHM has taken on the role of deploying, configuring and maintaining a Buildbot environment; this test activity has been going on since early in the project lifetime, and the test infrastructure has been continually extended and improved. As mutually agreed in plenary sessions on coding best practices, other partners contributing to the core library (i.e., adding a new functionality or patching existing ones) perform unit testing on their work before pushing it to the NEAT Github repository. For bug and test failures reporting we use the issue tracking service embedded in Github.

3.1.2 Test environment

The test environment for the NEAT System is based around a Buildbot architecture (Figure 3). Our Buildbot system consists of a single Buildbot Master (buildmaster) and several Buildbot Slaves (buildslave). The buildmaster controls the Buildbot slaves and hosts the status website. It runs on a dedicated FreeBSD system within a virtual machine. The Buildbot status pages are publicly accessible from the Internet, at: http://buildbot.nplab.de:28010.

Where possible, we run our buildslaves as virtual machines, this includes the FreeBSD, Linux and NetBSD buildslaves on X86 platforms. All our virtual machines run within VirtualBox on a dedicated FreeBSD rack-server hosted in the FHM laboratory. To achieve a good user experience for the contributing NEAT developers, it was important to provide the Buildbot test results as fast as possible. Therefore the virtual machines are hosted on a server with 16 cores, 64 gigabytes of memory and solid state disks. This powerful hardware setup allows us to run the complete tests in less than four minutes on all supported platforms, ensuring a fast feedback.

In addition to the virtual machines, we have dedicated buildslaves with MacOS X on an Apple MacMini and FreeBSD ARM on a Raspberry PI 2. All buildslaves have public IPv4 and IPv6 addresses to provide a good connectivity and ensure a maximum flexibility for the tests.

GitHub offers webhooks which are triggered by a variety of configurable events, this includes commits, issues or pull-requests. When a developer pushes code to the NEAT repository, GitHub fires a HTTP-POST request to our buildmaster with detailed information about the commit. This includes the author, branch name, unique commit id and the review URL. The HTTP-POST request triggers the Buildbot system test-suite.

3.1.3 Test procedures

Fetch and configure In the first step we test the reachability and consistency of our git repository (https://github.com/NEAT-project/neat). Our buildslaves fetch a fresh copy of the repository and start the CMake configuration procedure. The CMake configuration script checks if the platform is supported, all required dependencies are fulfilled and checks for optional libraries. For example if the platform doesn't have native SCTP support, CMake searches for the userland SCTP library and builds the project with it when available.

Compiling By default we compile the project with very strict compiler options to provide a high code quality. All available compiler warnings are activated and if the compiler triggers a warning, the compile process fails.

Test suite When the NEAT library has been successfully built, the buildslaves run a set of automated tests using simple example programs included in the repository. Our first set of tests focus on NEAT used by a client application which requests a HTML website from a web server. For this case we use the `client_http_get` example and connect to a thttpd web server via a combination of SCTP/TCP over IPv4/IPv6 connections. These tests cover the happy eyeballs and connection establishment functionality. In a second step, we test the server functionality of NEAT by using the included echo server. We establish connections from a client to the echo server via TCP, UDP and SCTP and check if the established connections reflect all the data correctly. In a third step, we execute a set of predefined PM unit tests which test the correctness of the PIB/CIB property logic, e.g., various property types, priorities and comparison operations.

When the functionality of the NEAT core functionality has successfully been tested, we repeat the same tests with the Valgrind framework [10] to detect memory leaks, uninitialized memory access and use after free errors.

In addition to run-time tests, we use the Clang Static Analyzer [2] to find bugs in the source code. If the static code analyzer finds an issue in the code, a HTML report is generated and automatically uploaded to the buildmaster status page.

Developers also have the ability to commit custom test scripts to the master branch (or a testing branch), enabling them to test specifics of their contributed functionalities. These tests may range from simple status checks to measurement-based performance tests evaluating the compliance to target metrics.

In addition to the aforementioned environment, we have started making use of the static code analysis service offered by Coverity[8]. As a result NEAT developers can be informed about critical code defects by accessing https://scan.coverity.com/projects/neat.

3.2 Consortium testbeds

The project partners are operating several testbeds used to evaluate the NEAT core library and industrial use cases defined in WP1. Each testbed offers a controlled experimentation platform where solutions can be deployed and tested in an environment that resembles real-world conditions. For the diverse set of tests to be delivered in WP4, experimental facilities have been identified for specific evaluations, listed in Table 6.

Table 6: Relevant test environments and applications/tools for testing the industrial use cases.

Use case	**Test environment**	**Most relevant applications and/or tools**†
Mozilla	Lab setup	Firefox thttpd (§ 2.2.2) nghttp2 (§ 2.2.7)
EMC	INFINITE testbed	Rsync PM diagnostics (§ 2.3.3)
Celerway	MONROE testbed	Multi-homed download manager (§ 2.2.8) PM diagnostics
Cisco	UoA Internet testbed	NEAT-streamer PM diagnostics

† NEAT-streamer and the NEAT ports of Firefox and Rsync are presented in detail in Deliverable D4.1 [14].

Table 6 provides a summary of the industrial use cases and the environments in which these will be evaluated. Tests executed in these testbeds will be augmented with results generated by experiments carried out on public Internet paths, where appropriate. In the following we detail the two largest testbeds, MONROE and INFINITE.

3.2.1 MONROE

The H2020 MONROE project (Measuring Mobile Broadband Networks in Europe) builds and maintains a testbed of several hundred multi-network nodes — up to five WANs, including three mobile broadband networks — that can be used to test new protocols and algorithms in realistic scenarios.

[8] http://www.coverity.com

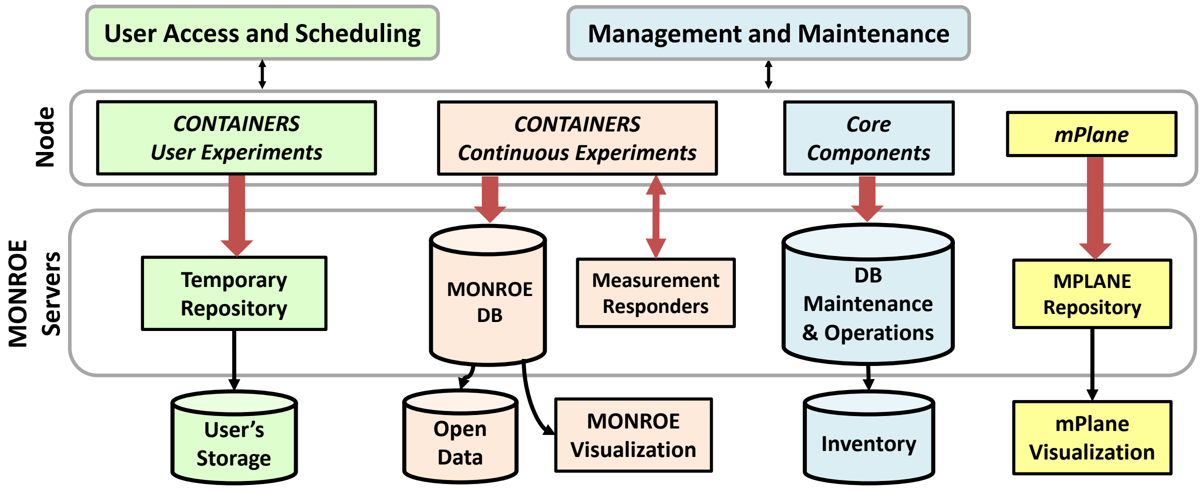


Figure 4: Design of the MONROE system.

The nodes are placed on buses, trains, trucks, and in offices and private homes. This testbed will help evaluating Celerway's industrial use case and work performed by SRL and KaU in WP3 on multipath scheduling. In addition, Celerway is building the NEAT library and some example (template) applications as MONROE *containers* in order to ease testing of NEAT for external users. Moreover, UoA has ported PATHspider and edgetrace to both run on MONROE, and these are now deployed experiments [24]. The following will describe the highlights of the MONROE architecture and node capabilities; for a full detailed overview, please visit https://www.monroe-project.eu. We also describe in detail how to build and deploy NEAT applications to be tested in MONROE in Appendix C.

MONROE overview The MONROE platform is composed of the following elements, which are also summarised in Figure 4:

- MONROE hardware nodes.
- Core software running in the nodes, including the management and scheduling systems.
- Base experiments running in the nodes.
- Synchronization modules that send data to the servers.
- Server-side software to collect data from the internal experiments and store them in databases.
- Server-side scheduling of experiments.
- Server-side scripts for data backup and database dump.
- Server-side software for node management and maintenance.
- Server-side experiment scheduling components.
- Web user interface.

It is the leftmost (green) components in Figure 4 that are relevant for NEAT's use of MONROE. Both default MONROE experiments and user experiments are executed inside Docker containers[9], which

[9] https://www.docker.com/what-container

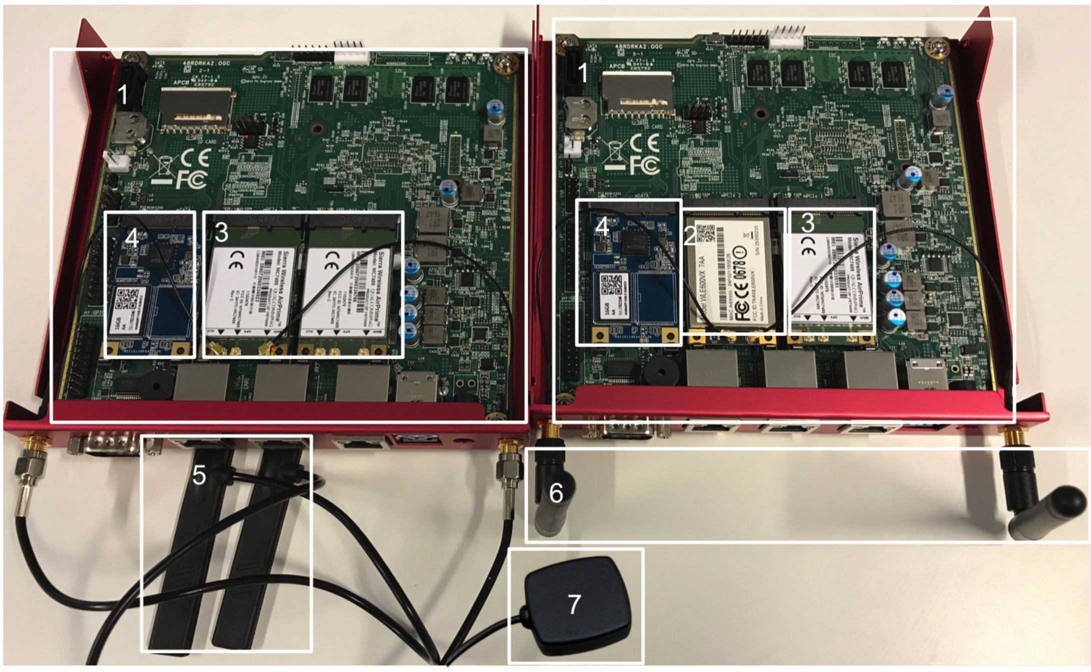


Figure 5: A MONROE node with all the main hardware components installed.

provide resource isolation from the host node. Docker containers are based on a layered file system, where a container can reuse layers shared with other containers. MONROE provides a default base image for the experiment containers. This base image provides a base operating system installation with default tools that are potentially useful for many experiments. When users base their own containers on the base image, they get automatic access to its underlying contents. This is a relevant consideration because it allows lightweight containers to provide just the contents that are unique for the particular experiment, significantly reducing the download and deployment time and traffic volume. As the base image resides permanently in the nodes, nodes need to transfer only the new layers created by the users for their experiments, saving data quota usage on the testbed's SIM subscriptions and hence, leaving more quota to run experiments.

Experiments running inside a container have access to the experimental network interfaces. They can read and write on their own file system, overlaid over that of the base MONROE image. They can use any software tool installed in the base image, or they can install their own tools and libraries in the overlaid file system.

MONROE nodes They consist of two APU2C4 with two SIM card slots each. We denote the two APUs *head* and *tail*, the former having two Sierra Wireless MC7455 CAT6 modems and the latter having one MC7455 CAT 6 modem and one dual band Compex WLE600VX WiFi mini PCIe card. The head and tail are connected with two Ethernet cables in order to support experimental scenarios over four networks from the head APU. Table 7 describes the core components of the node, which are shown in Figure 5 (numbers in the figure refer to block numbers in the table).

Appendix C describes in detail how to build an experiment with NEAT in MONROE.

Table 7: Hardware in a MONROE node.

Block no.	Component	Description
1	APU2C4 system board 4GB mounted in red metal enclosure with 3 LAN, USB 3.0 and 6 customised antenna holes[1]	• CPU: AMD Embedded G series GX-412TC, 1 GHz quad Jaguar core with 64 bit and AES-NI support, 32K data + 32K instruction cache per core, shared 2MB L2 cache. • DRAM: 4 GB DDR3-1333 DRAM. • 3 Gigabit Ethernet ports; 2 USB 3.0 ports. • 2 internal miniPCI express slots, with SIM slots. • Power: 12V DC, about 6 to 12W depending on CPU load. • Board size: 152.4 x 152.4mm. • The board also has an SD card reader, which is currently not used by MONROE.
2	Compex WLE600VX 802.11ac/b/g/n Dual-Band mPCIe module[2]	Dual band AC miniPCI express card that can be used in both access point and client mode.
3	Sierra Wireless MC7455 LTE mPCIe module[3]	miniPCI express card we use for MBB connection.
4	msata16g 16GB mSATA SSD module[4]	Used to store OS, MONROE SW, experiments, logs, and results. Can be extended to for instance 64GB if necessary.
5	External T-blade LTE Antenna[5]	We will use 2 LTE antennas that can be mounted on the red enclosure wall.
6	WiFi rubber swivel antenna 2.4/5.0GHz[6]	We attach to the node 3 WiFi dual band antennas.
7	Active/Passive GPS antenna[7]	We will use active and passive antennas (dependent of coverage in buses).
	Ethernet cables	Used to connect the APUs .

[1] http://www.pcengines.ch/apu2c4.htm
[2] http://www.pcengines.ch/wle600vx.htm
[3] http://techship.com/products/sierra-wireless-mc7455-lte-cat6/
[4] http://www.pcengines.ch/msata16g.htm
[5] https://techship.se/products/external-t-blade-lte-antenna/
[6] https://techship.se/products/wifi-rubber-swivel-antenna-24ghz-50ghz/
[7] https://techship.se/products/external-gps-antenna/

3.2.2 INFINITE

The INFINITE testbed[10], operated by EMC, provides a cloud infrastructure spanning three geographically dispersed datacenters in Cork, Ireland, interconnected via a dark fibre ring and combined with the Vodafone mobile M2M (machine-to-machine) network covering the island of Ireland.

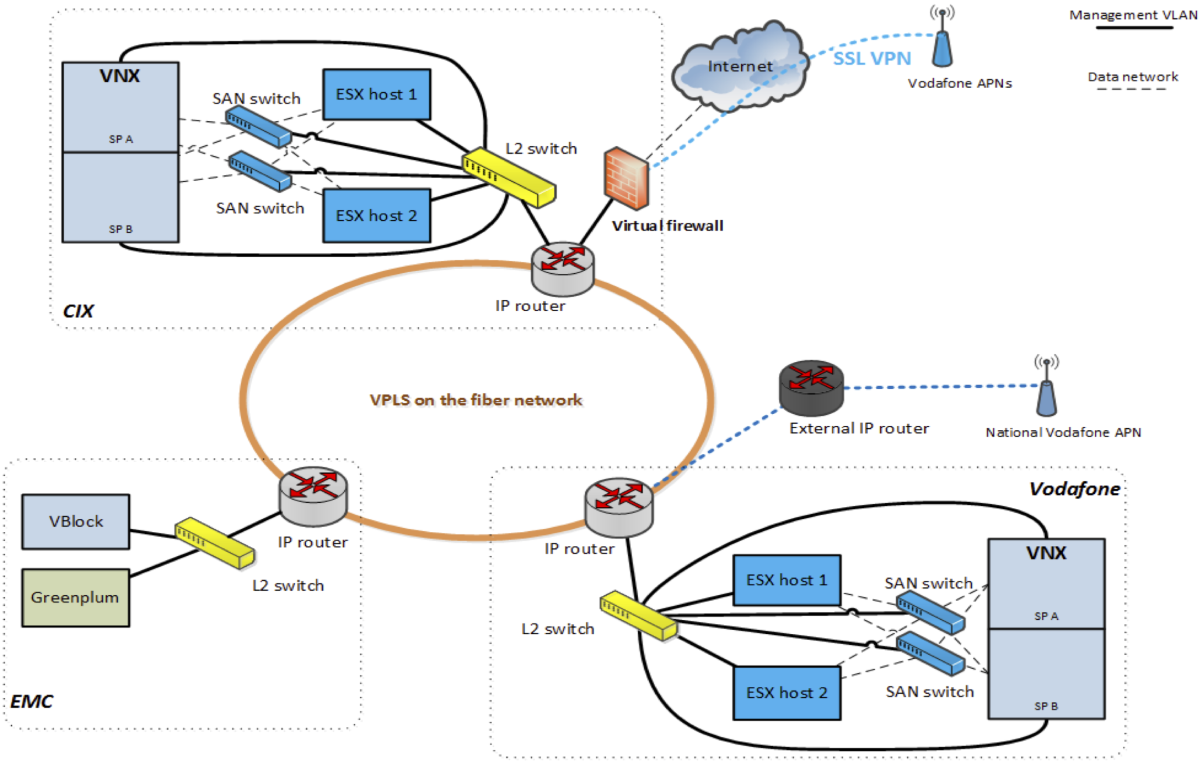


Figure 6: INFINITE Testbed topology.

INFINITE is comprised of a testbed environment with a large scale-out capability and enterprise technologies encompassing Cloud, Analytics, Security and Mobile M2M domains. The value provided by the testbed is derived from industrial use-cases that explore and clearly demonstrate the potential of innovative new solutions or products.

The goal of INFINITE is to make available a state-of-the-art secure virtual cloud and network infrastructure providing highly scalable bandwidth and capacity with analytics capabilities, spanning multiple domains (wireless, wireline, virtualized, physical) at scale.

The sites of the three INFINITE datacenters, depicted in Figure 6, are:

1. **Dell EMC datacenter:** it hosts Vblock converged infrastructure systems and GreenPlum analytics platforms.

2. **Vodafone datacenter:** it hosts compute, storage and networking resources.

3. **CIX datacenter:** it hosts compute, storage and networking resources. This datacenter serves as an independent location that offers wired Internet access to the testbed.

The logical architecture of the INFINITE Testbed is depicted in Figure 7. The testbed utilizes VMware technologies, such as vSphere, NSX, to provide isolated network slices distributed across the three datacenter locations for each experiment.

[10] http://www.iotinfinite.org

This testbed will be used for testing EMC's industrial use case as outlined in Section 3.3.4. This use case includes:

- An integration of NEAT with an SDN controller [13], developed by EMC and KaU, for intra-datacenter optimization. The SDN integration work was introduced in Deliverable D3.1 [18].
- A WAN testing of deadline-aware less than best-effort mechanisms [20] (DA-LBE), developed by SRL, in conjunction with NEAT for a cloud-based data transfer service. DA-LBE is described in detail in Deliverable D3.2 [19].

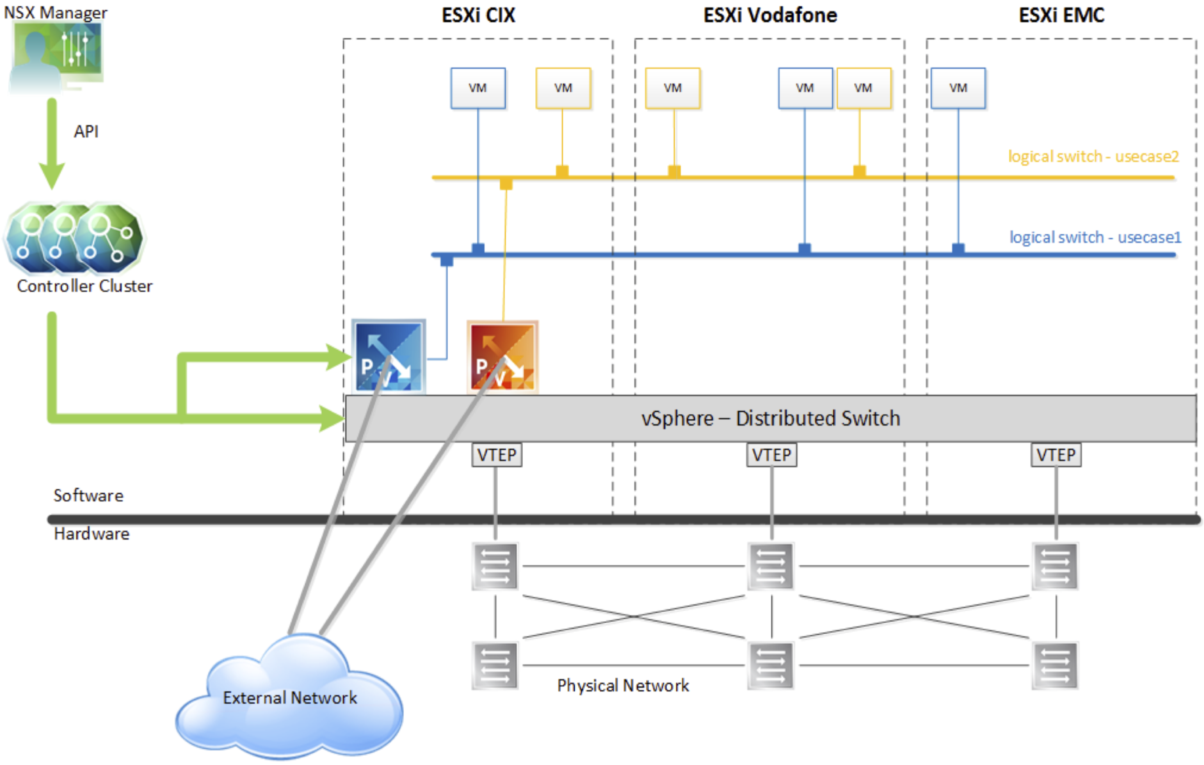


Figure 7: INFINITE Testbed logical architecture.

3.3 Use case testing

In the following we describe the use cases selected for testing and outline the testbeds in which these will be evaluated. Use case testing will allow not only to exercise basic functionalities of the NEAT core system, but also to demonstrate some of the benefits (described in Table 8) that WP3 mechanisms will bring to the selected use cases and applications.

Table 9, adapted from Deliverable D4.1 [14], shows the applications ported to NEAT that will be used for use-case testing. Similarly to the tools in Table 1, the Application Class as defined in the NEAT architecture is indicated; Class-3 apps make use of a NEAT Application Support module layered on top of the NEAT library, in this case a sockets shim developed in Task 3.3. Use case testing will be supported by a subset of the tools presented in Section 2.2. Specifically, thttpd and nghttp2 may be used in Mozilla's use case. Logging and Policy Manager Diagnostics will be utilised in all use cases.

Table 8: Expected benefits to industry use cases from transport protocol enhancements and extended transport system mechanisms, developed in WP3.

Use Case	**WP3 Mechanism**	**NEAT Contribution**
Mozilla	SCTP optimisations	Improved performance of SCTP and to make it a viable alternative to TCP; some optimisations needed to support flow priority
EMC	Deadline-aware LBE congestion control	Extended support for background traffic, including the traffic generated by data-backup and replication applications
Celerway, Mozilla	Multi-path scheduling	Support for latency-sensitive traffic, and efficient utilisation of available network resources
Mozilla	Coupled congestion control for single-path TCP	Improved performance of TCP, extended support for flow priority, and efficient use of available network resources
Mozilla	New transports for web browsing	Helps guide transport protocol selection and parametrization
Cisco, EMC	Extended policy system and transport selection	Enables rule based selection of transport protocols, interfaces and paths
Celerway, EMC	External controller integration	Support for externally generated policies and characteristics

Table 9: Applications ported to NEAT. *NEAT integration* indicates the migration path followed to port each application.

Application	**Test use case**	**NEAT integration**	**Key NEAT features used**	**App class**
Firefox	Mozilla	Hybrid solution using NEAT event loop in conjunction with Firefox event loop	Transport selection, policy	1
Rsync	EMC	Socket compatibility API	Policy, multipath	3
NEAT-streamer	Cisco	Full use of NEAT event loop and NEAT User API with Gstreamer	QoS, datagram service, policy	1

3.3.1 Mozilla use case

The Mozilla Firefox web browser is a sophisticated legacy application and runs on almost every conceivable hardware platform. It implements many mechanisms offered by the NEAT library. Happy-eyeballing between IPv4 and IPv6 and between different application layer protocols are just some of them. Firefox using the NEAT library is a good feedback platform for the NEAT System 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 an application such as Firefox can make use of NEAT's Happy-Eyeballing and use different available transport protocols. At the same time, the application performance, measured by page load times, should not be degraded — Firefox is already a well-tuned application and an improved performance is not expected from NEAT per se (though, of course, use of e.g. a multipath transport may boost throughput).

Table 10: Components of Mozilla's use case testing environment.

Component	**Description**
Application	In this case the application will be a Firefox distribution that uses the NEAT library (see Deliverable D4.1 [14]).
Server	The web server must be able to serve content using IPv4 and IPv6 and using different transport layer protocols, e.g., TCP, MPTCP, etc. It must be possible to configure the server to offer multiple IP and transport layer protocols as well as just a single one. The server host must have have two network interfaces to demonstrate capabilities of MPTCP. 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. For example, both a client and a server that support IPv6 and IPv4 protocols but the path is misconfigured and only allows IPv4 packets through.

Test topology

The base topology used for experiments is shown in Figure 8. The three key components of this setup are described in Table 10.

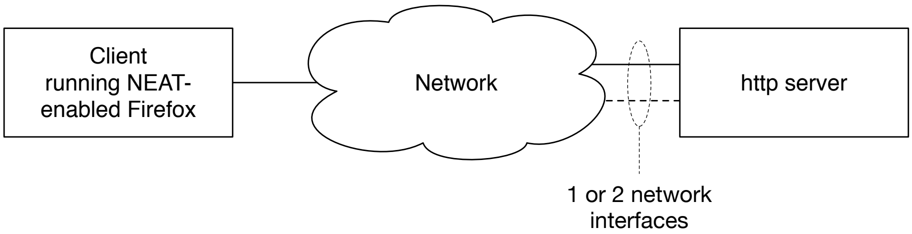


Figure 8: Base test topology for the Mozilla use case.

Planned tests

The experiments will demonstrate how Firefox uses the NEAT System for IP and transport protocol selection. Firefox will use NEAT to select the best protocol for the given network and capabilities of both the client and the server. This test will show that an Internet application can rely on the NEAT System to select the best available protocol.

Firefox is using NEAT with preference to pick IPv6 over IPv4 and request the use of MPTCP if it is available. The testing will include the scenarios listed in Table 11. In all these scenarios, the client supports both IPv4 and IPv6. Tests 1 focus on IP protocol selection, whereas Tests 2 focus on the use of multipath transport.

Expected Results

In each test Firefox should use the best available option according to the given application preference. The impact of NEAT's Happy-Eyeballs component should not influence the performance of Firefox.

Table 11: 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
2a	The server has one network interface; both client, network and server support IPv4 and IPv6
2b	The server has two network interfaces; both client, network and server support IPv4 and IPv6

Table 12: Components of Cisco's use case testing environment.

Component	Description
Application	It must be able to establish an interactive session between two hosts using traffic with suitable QoS marking. The NEAT-streamer application (see Deliverable D4.1 [14]) has been written to evaluate this use case. 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 QoS traffic with NEAT. The UoA Internet testbed will be used to evaluate lab environments for this use case.

The performance will be measured by page load times. Finally, use of the MPTCP protocol should show performance improvements compared to the standard TCP protocol.

3.3.2 Cisco use case

Cisco has an established line of business in providing remote video and audio conferencing software. Software implementing this use case must be able to offer services to the application that meet the requirements for low latency traffic, quality of service marking for that traffic and the ability to automatically discover properties about the local network environment via measurement and network signalling.

The testing environment for this use case includes two main components, listed in Table 12.

The goal of this use case is to improve application access to discoverable network properties (protocol support, cost, MTU) and to facilitate the deployment of low latency traffic. The NEAT System, informed by network properties discovered through PvD (Provisioning Domains) should be able to generate the traffic required to support a two-way, live video connection or signal that this is not possible on the current network.

The NEAT stack will be run in the UoA Internet testbed configured to simulate a set of different network environments simultaneously available, with configuration advertised over PvD:

1. Network without PvD.

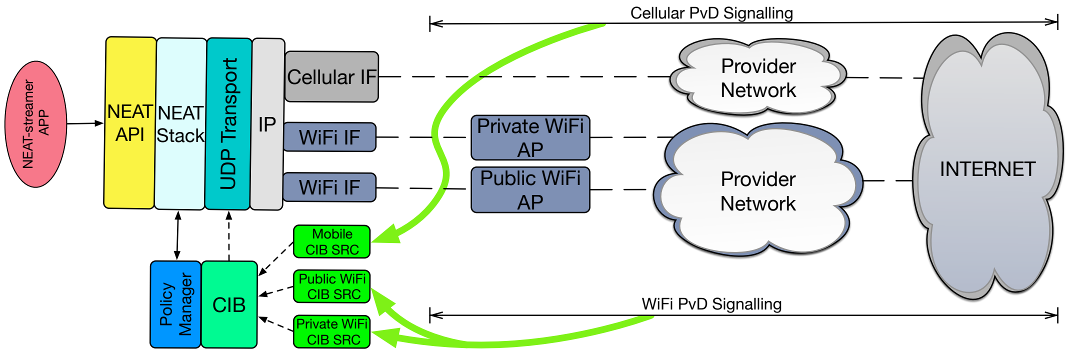


Figure 9: Wireless LAN topology for the Cisco use case.

Table 13: Experiments for Cisco's use case testing.

Test ID	Summary
1a	With a single, non signalling enabled network
2a	With dual networks signalling, non signalling
2b	Dual networks, signalling high capacity, signalling low capacity
2c	Dual networks, signalling high capacity + high cost, signalling low capacity + low cost; profile preferring cost
2d	Dual networks, signalling high capacity + high cost, signalling low capacity + low cost; profile preferring throughput
2e	Dual networks, signalling low latency + high cost, signalling low capacity + low cost; profile preferring low latency

2. Network Corporate PvD Policy.

3. Network Open Network PvD Policy.

The application will be expected to set up a bidirectional video feed in each case, exhibiting fallback between values if required. The NEAT stack will capture and use additional information from the network when it is available, i.e., PvD based signalling.

Test Topology

The test topology (Figure 9) will emulate a roaming NEAT-enabled client that can use access networks with different configurations advertised on different interfaces. Multiple networks may be available to the roaming host a single time, in this case the NEAT System will use signalled information to make a network selection choice.

Planned Tests

In all cases the same NEAT-streamer test will be run, NEAT-streamer will attempt to establish an interactive video workload connection to a remote peer, reporting obtained throughput and latency. This test will be run in several network configurations, shown in Table 13.

Expected Results

In each of the planned tests above, the NEAT stack is expected to produce a selection in line with the specified policy, availability of information from the network or to fall through to a default case.

3.3.3 Celerway use case

The Celerway use case includes NEAT components on multi-homed devices like smartphones and routers. Based on information about application needs and collection of network statistics, this use case involves optimal selection of interfaces and transport options. Application needs can be learned through the NEAT User API on client devices, or by inferring application types and needs on a router or proxy. In order to test the Celerway use case, we are developing a set of CIBs and PIBs focusing particularly on mobile broadband, developing an application using NEAT, and implementing NEAT in the H2020 MONROE platform as described in Section 3.2.1.

Celerway CIBs and PIBs: We are developing CIB sources and policies that collect and use information about network performance and metadata to make optimal interface and transport selections. An example CIB will be populated with mobile broadband metadata. Table 3 gives an overview of the properties that are supported.

NEAT in MONROE: Celerway will test and experiment with its use case by using the MONROE platform. 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 act either as a client supporting NEAT-enabled applications, or as a proxy supporting non-NEAT applications. Implementation of Celerway's use case includes the following elements:

- Deployment of the NEAT System on MONROE nodes. This will also make 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 a CIB via a Unix domain socket to the PM. The NEAT PM is notified immediately upon every detected change in network properties.
- 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.

Next, we present the test setup and topology, planned tests and experiments and expected results and outcome of the tests.

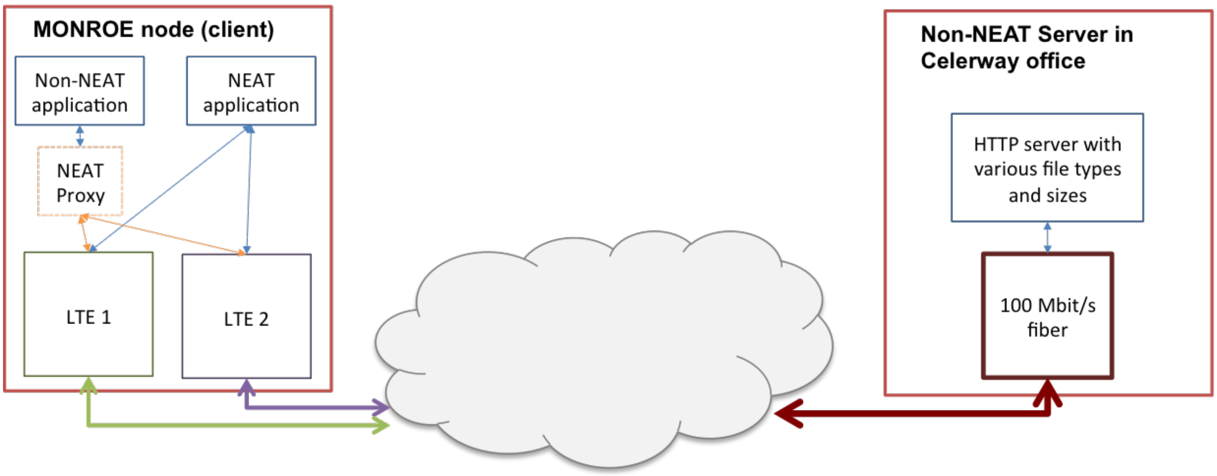


Figure 10: Test topology for the Celerway use case.

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

Component	**Description**
MONROE node as client	Described in detail in § 3.2.1.
Non-NEAT application	An application that is not using NEAT. A non-NEAT enabled Download manager will be the first example.
NEAT application	An application that is using NEAT. A NEAT-enabled Download manager will be the first example.
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 and 100 Mbit/s fiber link.

Test Topology

Figure 10 depicts the topology which serves as the basis of the experiments. The test setup will be comprised of the key components listed in Table 14.

Planned Tests

Table 15 summarizes the planned experiments that will be carried out as part of this use case. Specifically these tests will be as follows:

- Test 1 will use the Download manager described in § 2.2.8 running on a MONROE node using the topology described above. In this case, the Download manager will be a NEAT application (i.e., a Class-1 application). It will use CIBs containing metadata about LTE networks and bandwidth measurements to make optimal interface selection. The main metric will be throughput.

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

Test ID	Summary
1	Evaluate the impact of NEAT CIBs and PIBs on NEAT applications in multi-homed mobile scenarios
2	Evaluate the impact of NEAT CIBs and PIBs on non-NEAT applications using a NEAT proxy in multi-homed mobile scenarios

- Test 2 will use the Download manager described in § 2.2.8 running on a MONROE node using the topology described above. However, in this case, the Download manager will *not* be NEAT-enabled (i.e., a Class-0 application). Hence, it will go through the proxy that must infer the application needs. It will use CIBs containing metadata about LTE networks and bandwidth measurements to make optimal interface selection based on inferred application needs. The main metric will be throughput.

Expected Results

Through Test 1, we will demonstrate that NEAT can increase throughput and application quality by using the policy system with generated CIBs.

Through Test 2, we will demonstrate that NEAT can increase throughput and application quality also for non-NEAT applications by inferring the application needs.

3.3.4 EMC use case

The EMC use case aims to make a datacentre network aware of application requirements and network conditions. The use case expects to leverage the transport optimisations provided by the NEAT System interacting with a SDN controller/orchestrator which manages the datacenter network.

The primary goal of this use case is to improve the performance for large data transfers (also 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. Thus the optimisation of the whole network's performance will be considered in the evaluation. In addition the use case will test DA-LBE congestion control mechanisms incorporated into NEAT as part of WP3. This data replication scenario is comprised of a client connected to a datacentre over a wide area network. The test will aim to demonstrate the file transfers targeting a predefined completion time without adversely impacting concurrent network traffic.

Test Topology

Figures 11 and 12 depict the two topologies which serve as the basis of the experiments. The setup components are described in Table 16. The topology depicted in Figure 11 will be used to evaluate an SDN datacenter scenario. The topology depicted in Figure 12 will be used to evaluate a WAN cloud provider scenario using DA-LBE.

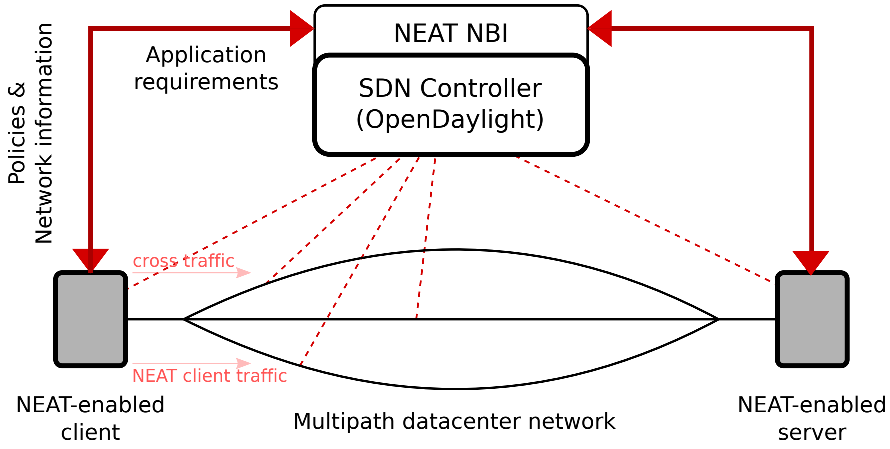


Figure 11: Datacenter topology for the EMC use case (SDN experiments).

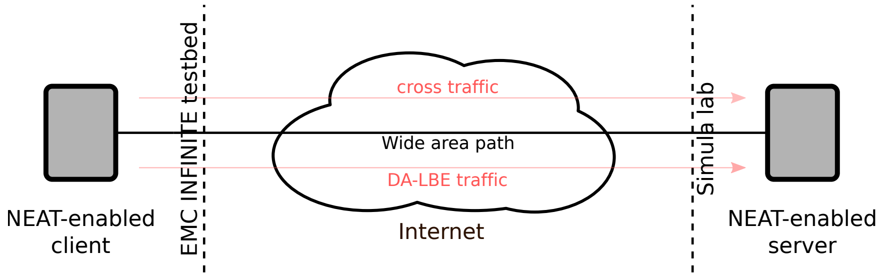


Figure 12: WAN topology for the EMC use case (DA-LBE experiments).

Planned Tests

Several scenarios will be tested 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 17 summarizes the planned experiments that will be carried out as part of this use case.

Congestion will be induced by replaying realistic cross-traffic generated using existing traffic traces and generator tools [3, 26].

Expected Results

The expected results for the above sequence of tests are:

- Faster transfer times due to the transport protocol optimizations provided by NEAT (Tests 1a and 1b).
- Improved isolation between elephant flows and on cross-traffic flows, resulting in less degradation in transfer speed and throughput in a congested network (Tests 2 and 3).
- Better network utilisation due to the seamless ability to exploit multiple physical paths in an intelligent way orchestrated through the SDN-NEAT integration (Test 3).

Table 16: 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`) has been selected as a representative open-source application. Denoted *NEAT-enabled client/server* in Figures 11 and 12.
Traffic generator	The traffic generators D-ITG and DCT2Gen will be used to generate *cross traffic* with the desired characteristics.
Multipath datacenter network	SDN-enabled physical or virtualized 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 will be used to simulate different conditions in a managed network, e.g., high/low congestion, high/low latency, heavy/light load.
Wide area path	An Internet path traversing the public Internet from the INFINITE testbed to a node hosted at SRL.
SDN Controller	The OpenDaylight open-source SDN controller framework will be used to manage the datacenter network, monitor its status and interact with the attached NEAT Systems on the hosts (relying on the work developed in WP3).

Table 17: 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 datacenter network with SDN-supported orchestration, and empty and congested links
3	Transparent handling of elephant flows using controller-assigned DSCP marking or MPTCP subflows mapped to disjoint network paths using `neat-rsync`
4a	Large file transfer between `neat-rsync` client and server over WAN path using TCP
4b	Large file transfer between `neat-rsync` client and server over WAN path using DA-LBE transport
4c	Large file transfer between `neat-rsync` client and server over WAN path with parallel DA-LBE and TCP transport

- Tests 4a, 4b and 4c will demonstrate that reference TCP flows, injected in parallel to the DA-LBE flow, will not be impacted significantly by DA-LBE file transfers.

4 Conclusions

In this deliverable we presented the final set of tools developed within the NEAT project in order to test performance and functionality of various components of the NEAT stack. In addition, we provided an overview of the testing environments which are used to test and evaluate the NEAT System and provided a test plan for evaluating the project's industrial use cases. The described experiments will demonstrate the benefits of the NEAT transport system. The experiments are carried out in Task 4.3 and their outcomes will be reported in Deliverable D4.3. Finally, the experiences and lessons learnt from these activities will provide additional feedback driving final code and API adjustments.

References

[1] Buildbot — The Continuous Integration Framework. [Online]. Available: https://www.buildbot.net/

[2] Clang Static Analyzer. [Online]. Available: https://clang-analyzer.llvm.org/

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

[4] NEAT's Pipelined HTTP GET utility. [Online]. Available: https://github.com/NEAT-project/HTTPOverSCTP

[5] nghttp2 — HTTP/2 C Library. [Online]. Available: https://www.nghttp2.org/

[6] Pipelined HTTP GET utility. [Online]. Available: http://www.daemonology.net/phttpget/

[7] Scapy. [Online]. Available: http://www.secdev.org/projects/scapy/

[8] tcpexposure — The middlebox measurement tool. [Online]. Available: https://github.com/micchie/tcpexposure/

[9] thttpd — Tiny/turbo/throttling HTTP server. [Online]. Available: http://acme.com/software/thttpd/

[10] Valgrind. [Online]. Available: http://valgrind.org/

[11] R. Barik, M. Welzl, and A. Elmokashfi, "How to say that you're special: Can we use bits in the IPv4 header?" in *Proceedings of the Applied Networking Research Workshop (ANRW)*, Berlin, Jul. 2016. [Online]. Available: https://irtf.org/anrw/2016/anrw16-final17.pdf

[12] R. Barik, M. Welzl, A. Elmokashfi, S. Gjessing, and S. Islam, "fling: A flexible ping for middlebox measurements," in *29th International Teletraffic Congress (ITC 29)*, Genoa, Italy, Sep. 2017.

[13] Z. Bozakov, S. Mangiante, C. Benet, A. Brunstrom, R. Santos, A. Kassler, and D. Buckley, "A NEAT framework for enhanced end-host integration in SDN environments," in *IEEE Conference on Network Function Virtualization and Software Defined Networks (IEEE NFV-SDN)*, Berlin, Nov. 2017, accepted for publication, to appear.

[14] 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.

[15] B. Briscoe, M. Kuehlewind, and R. Scheffenegger, "More Accurate ECN Feedback in TCP," Internet Engineering Task Force, Internet-Draft draft-ietf-tcpm-accurate-ecn, Oct. 2016, Work in Progress. [Online]. Available: https://tools.ietf.org/html/draft-ietf-tcpm-accurate-ecn

[16] G. Detal, B. Hesmans, O. Bonaventure, Y. Vanaubel, and B. Donnet, "Revealing Middlebox Interference with Tracebox," in *Proceedings of the 2013 Conference on Internet Measurement Conference (IMC)*, Barcelona, Spain, 2013, pp. 1–8. [Online]. Available: http://doi.acm.org/10.1145/2504730.2504757

[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] K.-J. Grinnemo, Z. Bozakov, A. Brunstrom, M. I. Bueno, D. Damjanovic, K. Evensen, G. Fairhurst, A. Hansen, D. Hayes, P. Hurtig, N. Khademi, S. Mangiante, M. Althaff, M. Rajiullah, D. Ros, I. Rüngeler, R. Santos, R. Secchi, T. C. Tangenes, M. Tüxen, F. Weinrank, and M. Welzl, "Initial Report on the Extended Transport System," NEAT Project (H2020-ICT-05-2014), Deliverable D3.1, Dec. 2016.

[19] K.-J. Grinnemo, Z. Bozakov, A. Brunstrom, D. Damjanovic, K. Evensen, G. Fairhurst, A. Hansen, D. Hayes, P. Hurtig, N. Khademi, S. Mangiante, D. Ros, I. Rüngeler, M. Tüxen, F. Weinrank, and M. Welzl, "Final Report on Transport Protocol Enhancements," NEAT Project (H2020-ICT-05-2014), Deliverable D3.2, Feb. 2017.

[20] 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. [Online]. Available: http://dl.ifip.org/db/conf/networking/networking2017/1570334752.pdf

[21] 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/

[22] C. Kreibich, N. Weaver, B. Nechaev, and V. Paxson, "Netalyzr: Illuminating the Edge Network," in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement (IMC)*, Melbourne, 2010, pp. 246–259. [Online]. Available: http://doi.acm.org/10.1145/1879141.1879173

[23] I. R. Learmonth, B. Trammell, M. Kuhlewind, and G. Fairhurst, "PATHspider: A tool for active measurement of path transparency," in *Proceedings of the Applied Networking Research Workshop (ANRW)*, Berlin, 2016, pp. 62–64. [Online]. Available: https://irtf.org/anrw/2016/anrw16-final13.pdf

[24] R. Secchi, A. Venne, and A. Custura, “Measurements concerning the DSCP for a LE PHB,” Presentation at the TSVWG meeting, 99th IETF, Jul. 2017. [Online]. Available: https://datatracker.ietf.org/meeting/99/materials/slides-99-tsvwg-sessb-31measurements-concerning-the-dscp-for-a-le-phb/

[25] B. Trammell, M. Kühlewind, P. De Vaere, I. R. Learmonth, and G. Fairhurst, “Tracking transport-layer evolution with PATHspider,” in *Proceedings of the Applied Networking Research Workshop (ANRW)*, Prague, 2017, pp. 20–26. [Online]. Available: https://irtf.org/anrw/2017/anrw17-final16.pdf

[26] 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.

A NEAT Terminology

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

B Example JSON file for a fling test

An example JSON file for fling is shown in Listing 4; both the client and server execute it after an HTTPS handshake has completed. In this example, the fling client sends a SYN packet and then waits for up to 2000 ms to receive a SYN/ACK packet which it stores upon reception. The server waits for up to 2000 ms to receive a SYN packet which it stores upon reception—entering state "S1"—and, either upon timer expiry or immediately after receiving the packet, enters state "S2". Then it immediately sends a SYN/ACK in response.

```
{"name":"TCP SYN/ACK test",
    "__index":{"0":"TCP SYN",
          "1":"TCP SYN/ACK"},
"packet_Info":[
{"name":"TCP SYN",
    "portFlip":[0,2,2],
    "ChksumType":"adler-32",
    "ChksumPos":[16,0],
    "ChksumLen":[2,0],
    "ChksumPseudoHDR":true},
{"name":"TCP SYN/ACK",
    "portFlip":[0,2,2],
    "ChksumType":"adler-32",
    "ChksumPos":[16,0],
    "ChksumLen":[2,0],
    "ChksumPseudoHDR":true}],
"client":{
"state_sequence":["S1"],
  "states":[{"state":"S1",
     "send":["TCP SYN"],
     "recv":["TCP SYN/ACK"],
     "delaySend":[0],
     "timeout":[2000]}]},
"server":{
  "state_sequence":["S1","S2"],
  "states":[{"state":"S1",
    "recv":["TCP SYN"],
    "timeout":[2000]},
  {"state":"S2",
    "send":["TCP SYN/ACK"],
    "delaySend":[0],
    "timeout":[2000]}]}}
```

Listing 4: JSON file for a simple TCP SYN-SYN/ACK dialogue test.

In a fling experiment description, every state contains the entries "send" or "recv" (or both). Whenever there is a recv entry, there must be a timeout, to specify how long fling will wait for reception of the specified packet. The last state must have a timeout anyway, even when there is no recv, because the timeout is also used to check if the transmitted packet made it to the other end. For each state, multiple packets can be specified to be sent or received, and each transmission can be accompanied by a “delaySend” value: the time that fling waits before sending a packet.

Listing 4 also shows the common header of a fling experiment description. It contains the name of the experiment as well as an index entry that maps the pcap file packet numbers to names in the description text. In this example, the pcap file contains a SYN packet, followed by a SYN/ACK packet.

C How to build and test NEAT applications in MONROE

In the following, we explain how to create a MONROE experiment that is able to 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. *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 build of 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*[11]. 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[12]. 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.

Table 3 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

[11] https://github.com/MONROE-PROJECT/UserManual

[12] http://zeromq.org

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.

D Paper: *fling: A Flexible Ping for Middlebox Measurements*

The following research paper [12] has been produced by project participants, and is accepted for publication at the 29th International Teletraffic Congress (ITC 29) to be held in Genoa, Italy in September 2017.

fling: A Flexible Ping for Middlebox Measurements

Runa Barik, Michael Welzl
Department of Informatics,
University of Oslo, Oslo, Norway
{runabk,michawe}@ifi.uio.no

Ahmed Elmokashfi
Simula Research Laboratory, Norway
ahmed@simula.no

Stein Gjessing, Safiqul Islam
Department of Informatics,
University of Oslo, Oslo, Norway
{steing,safiquli}@ifi.uio.no

***Abstract*—Middleboxes in private networks have been known to change packets in many ways, making it hard to design protocol extensions that work for the large majority of Internet users. Addressing the need to know what such middleboxes do, we introduce a tool called *fling* (“flexible ping”). *fling* can carry out (almost) any kind of protocol dialogue between a server and a client based on a simple specification in a json and a pcap file, and identify what middleboxes do to the packets of the dialogue. This fills a gap in the state of the art, where other tools that control both ends of a path are either limited in some form or have to be updated for every new test. We present results from small-scale tests that prove the flexibility of *fling*, which is a prerequisite for our next step: development of a large-scale measurement platform.**

***Index Terms*—Middlebox, Measurement, fling**

I. Introduction

When developing protocol extensions in the IETF, it is often important to know what will happen to particular types of packets along a path (“If we add an option to this packet, is it more likely to be dropped? Will the option often be removed?”). Such considerations have played a major role in the design of Multipath TCP (MPTCP) [40]. Middleboxes have also been shown to harm network measurements per se [13]. Our previous work [11] investigated the effects of middleboxes on certain fields of the IP header, and found that nonzero DSCP values may provoke consistent packet loss. This has affected the IETF rtcweb standard[1]. While this small measurement study already used a preliminary prototype of *fling*, this is the first time that we fully describe the tool itself.[2]

During the past decade, deployment of large scale measurement infrastructures has become popular both to inform policy makers and help users gain insights into their network performance (e.g., M-Lab, RIPE Atlas, BISmark, SamKnows, etc.). A good overview of existing measurement platforms is given in [9]. Most of them use customized variants of ping, traceroute, ntp, netstat, iperf, etc. to measure various network aspects, but they usually do not focus on analyzing the influence of middleboxes on traffic. Some can detect middleboxes to some extent: for example, Netalyzr [29] sends TCP and UDP packets to test port reachability and test for proxies, uses an ESP header over UDP packets to detect IPSec NATs, performs Path MTU Discovery and examines the effect of IP fragmentation.

A number of tools were designed to specifically measure the impact of middleboxes. For instance, TCPExposure [25] tests certain TCP options between clients and a dedicated server, and tracebox [17] combines middlebox testing with traceroute to deduce packet changes from the payload of ICMP Echo Reply messages. Generally, doing tests that require more than normal user privileges from people’s homes, which is where most problematic middleboxes are expected, is a difficult matter. It has been achieved via payment, e.g. to ship dedicated hardware [5], [44] or pay users to run a tool [32] (which creates a natural scalability limit), or by limiting the campaign to one-time tests, where testers are personally asked to run a tool that was created for a particular test. Such a one-time measurement campaign was done with TCPExposure [25], and the difficulty of repeating experiments is one of the lessons learnt according to the authors [24].

We introduce *fling*—our attempt to learn from these past success stories and combine them in a way that makes it easy to repeat two-sided middlebox measurements, even when they require administrator privileges. *fling* is an end-to-end active measurement tool that allows testing whether an arbitrary sequence of packets can be exchanged between a *fling* client and a *fling* server. These packets are defined in a PCAP file, which is accompanied with a JSON file that describes a dialogue. Tests are uploaded to the server and pulled by *fling* clients whenever they run, such that clients do not need to be updated whenever new tests are defined.

fling resembles ping in that it sends a number of packets from a *fling* client to a *fling* server and expects a few packets in return. Different from ping, these packets are not all equal – and *fling* executes a dialogue that is defined per test and can be much longer and more complex than ping’s exchange of two packets. The security implications of ping are well known because ping is simple and well understood. We expect the same to be the case for *fling*—at the same time, we tried to make it as flexible as possible. Naturally, there are limitations to this flexibility (100% flexibility can only be achieved by installing new code for each measurement, which we wanted to avoid). Like ping, *fling* is not intended and cannot be (reasonably) used for bandwidth measurements; it is meant for sending and receiving a handful of packets and seeing what happened to them.

[1] https://www.ietf.org/mail-archive/web/tsvwg/current/msg14431.html

[2] We also presented an earlier prototype at the IMC 2016 Works-in-Progress Session.

TCPExposure is very similar to *fling*: it supports a client-server dialogue of any type of packets and observes what happens to them along the path. However, with TCPExposure, for every new test, the code would have to be updated, and users would have to be asked to install it and run an experiment. We wanted to develop a static tool that would yield this flexibility without requiring to get in touch with users and ask them to install new code.

Netalyzr is attractive for users to run as it lets them learn about their own network connectivity. It managed to attract a large user base. We tried to learn from that lesson by offering the same type of feedback to users that download and run *fling* on their home machines; in our case, the effort to use it is higher (it is not embedded in the browser, because we need to use raw sockets) but the range of possible outputs is wider (because we use raw sockets). Also similar to Netalyzr, it is possible to update tests by changing a dedicated server only. Like tracebox, *fling* also identifies *where* on the path a packet change or drop occurred.

After an overview of related work in the next section, section III will introduce the design of *fling*, including a discussion of its inevitable limitations. We have put these limitations to the test with a small measurement study, which we report about in Section IV. Section V concludes the paper and discusses our next steps towards the development of a permanent *fling* measurement platform.

II. Related Work

The increasing popularity of middleboxes has motivated several efforts to characterize their deployment and assess their impact on data plane performance. Medina et al. [34], [35] actively probed a set of web servers using TBIT [39] to assess the interaction between middleboxes and transport protocols. Honda et al. [25] developed TCPExposure to test whether TCP options are supported. TraceBox [17] improved over TCPExposure by proposing a Traceroute-like approach to pinpoint routers that alter or discard TCP options. Cravan et al. [16] proposed TCP HICCUPS, a tool that reveals TCP header manipulation to both ends of a TCP connection. *PATHspider* [31] is a recent tool that allows for A/B testing of a baseline configuration against an experimental configuration.

Table I provides an overview of the measurement tools mentioned above, and shows how they compare to *fling*. The first two columns illustrate a limitation of prior work that *fling* addresses: while almost all of the tools use raw sockets, potentially allowing them to transfer *any* type of Internet packet, experiments so far have mostly been limited to TCP over IP: the TCP header's source port, initial sequence number, window and option fields were the focus of [16], [17], [25], the IP header's DSCP, ECN, flags, source address and option fields were considered in [11], [21], [34], [35], [37]–[39], [45], [46], while, to the best of our knowledge, the only recent middlebox measurement studies considering protocols such as UDP, SCTP and DCCP are [22], [33] and [19].

Other papers focused on investigating specific types of middleboxes such as web proxies [47], transparent HTTP proxies in cellular networks [48], firewalls and NATs policies in cellular networks [46], and carrier grade NATs [37]. Trammell et al. [45] have proposed correlating measurements from diverse vantage points to build a map of middlebox-induced path impairments in the Internet.

fling bears some resemblance to pcap replaying tools such as tcpreplay [6] and its variants; it differs in that *fling* is two-sided, describing a complete dialogue, with timeouts, behaviour that is triggered by the reception of packets, etc. We will now turn to a full description of *fling*'s design.

III. *fling* Design

Ping sends a number of specific packets (typically ICMP Echo Request) to a host and expects to get the same number of reply packets (typically ICMP Echo Reply). In essence, this is also what *fling* does, but it adds flexibility: *any* packet can be used instead of ICMP Echo Requests, and the dialogue can take any form, involving multiple packets (e.g. in case of TCP, using only single packets would limit *fling* tests to SYN-SYN/ACK tests).

```
{"name":"TCP SYN/ACK test",
"__index":{"0":"TCP SYN",
                "1":"TCP SYN/ACK"},
"packet_Info":[{
                "name":"TCP SYN",
                "swap":[0,2,2],
                "ChksumType":"adler-32",
                "ChksumPos":[16,0],
                "ChksumLen":[2,0],
                "ChksumPseudoHDR":true
                },{
                "name":"TCP SYN/ACK",
                "swap":[0,2,2],
                "ChksumType":"adler-32",
                "ChksumPos":[16,0],
                "ChksumLen":[2,0],
                "ChksumPseudoHDR":true}],
"client":{
        "state_sequence":["S1"],
        "states":[{
                "state":"S1",
                "send":["TCP SYN"],
                "recv":["TCP SYN/ACK"],
                "delaySend":[0],
                "timeout":[2000]
                }]},
"server":{
        "state_sequence":["S1","S2"],
        "states":[{
                "state":"S1",
                "recv":["TCP SYN"],
                "timeout":[2000]
                },{
                "state":"S2",
                "send":["TCP SYN/ACK"],
                "delaySend":[0],
                "timeout":[2000]
}]}}
```

Figure 1: json file for a simple TCP SYN-SYN/ACK dialogue test

A *fling* client is a static piece of software; it begins a test by pulling a test description (a pcap file containing the test packets and a json file describing the test) from the server, which it then executes. A test description specifies the packet types, some information about header fields, and the sending/receiving sequences of the dialogue. It never contains

Tool	Raw sockets	Test protocols other than TCP	Test update: need to change	Fully controlled client-server dialogue	Detect TCP connection splitters	tracebox-like location detection
fling	✓	✓	Server	✓	✓	✓
Netalyzr	✗	✓*	Server	✓	✓**	✓‡
TCPExposure	✓	✗	Both	✓	✓	✗
HICCUPS	✓	✗	Both	✓	✓	✗
Tracebox	✓	✓	Client	✗	✓	✓‡
PATHspider	✓	✓	Client	✗	✓	✓‡
TBit	✓	✗	Client	✗	✗	✗

Table I: Comparison of related tools. *ICMP,UDP; ‡One-sided only; ** only HTTP proxies.

addresses: a *fling* client always only talks to a specified (supported as a command-line option) *fling* server. This allows to fully control the dialogue and collect measurement results at the server for research use; it also serves as a security measure, by ensuring that attackers cannot design tests that would turn *fling* clients into sources of traffic towards some other hosts in the network. To avoid getting in the way of normal Internet usage of *fling* users, the total maximum number of packets transmitted by *fling* is also statically configurable by the client.

An example json file is shown in figure 1; both the client and server execute it after an HTTPS handshake (this is the control channel, which we will explain in the next section). Every *fling* test begins with at least one packet from the client. In our example, the client sends a SYN packet and then waits for up to 2000 ms to receive a SYN/ACK packet which it stores upon reception. Starting from the point where it answers the HTTP request (see Fig. 2), the server waits for up to 2000 ms to receive a SYN packet which it stores upon reception (state "S1"). Then, either upon timer expiry or immediately after receiving the packet, it enters state "S2" and sends a SYN/ACK in response.

In a *fling* experiment description, every state contains a timeout, which gives a limit for the duration of the state. Packets are logged until the state is over. The entries "send" or "recv" are used to transmit packets or expect the reception of packets, respectively. Receiving packets in accordance with "recv" terminates a state before the timeout. For each state, multiple packets can be specified to be sent or received, and each transmission can be accompanied by a "delaySend" value: the time that *fling* waits before sending a packet. When a state contains "send", the state's timeout begins after the last packet was sent.

Figure 1 also shows the common header of a *fling* experiment description. It contains the name of the experiment as well as an index entry that maps the pcap file packet numbers to names in the description text (in this example, the pcap file contains a SYN packet, followed by a SYN/ACK packet). The "packet_info" statement contains information about port swapping and checksums; we will explain this later.

A. The Fling Control Channel

fling is not meant to be shipped with a specific set of tests; rather, it obtains tests from a *fling* server. This query is made using HTTPS, initiated by the client (because we assume that many *fling* clients would operate behind a NAT). We call this HTTPS communication the Fling Control Channel (FCC) because it is used to transmit more control information—among them, a nonce that the client inserts into *fling* packets, at a position that can be defined per packet as part of the test description.

The nonce allows the server to identify *fling* packets. Because the idea of *fling* is to allow exchanging *any* packet type, we cannot rely on common methods to determine where a packet comes from. Consider, for example, two clients behind the same NAT, doing an ICMP test: because ICMP has no port numbers, we need our own identifier to tell these clients apart.

The FCC also lets us transport the client's received *fling* packets back to the server as HTTPS payload for further analysis, and it is used to control experiment initiation and termination. As shown in the Fig. 2, a session begins with the client sending an HTTPS GET request containing an empty "Hello" message. The server answers with a test number, the pcap and json files, a *salt* value for nonce generation and a dictionary that contains a random number for each *fling* packet. It also starts the first timer to wait for incoming *fling* packets. When the client gets the HTTP response, it starts the test by transmitting the first *fling* packet. When the test is finished, the client sends its results (all logged packets that it received) as HTTPS payload to the server, and the server responds with some further data about the test that is useful for the client to give feedback to the user.

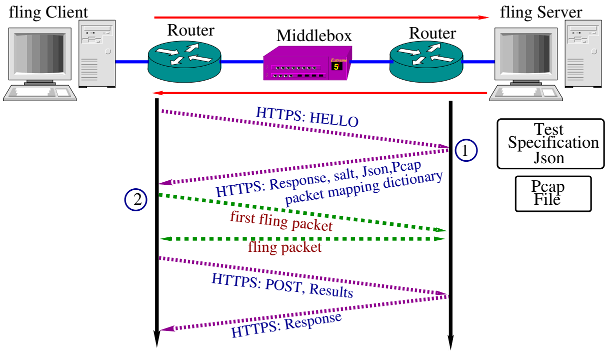


Figure 2: *fling* client and server interaction. The server starts the first timer at (1). At (2), the client begins by sending the first *fling* packet and starting its first timer.

B. Security and NATs

fling's nonce lets the server associate *fling* packets with the preceding HTTPS handshake. It also prevents a reflector attack, where an attacker would alter a *fling* client to provoke the server to send packets back to a spoofed source address. The nonce is the concatenation of the 8-bit *salt* value per experiment and a random number generated for each packet.

In the interest of flexibility, we allow freedom to choose the length of the nonce and to specify where in the packet it should be written. For the position of the nonce, an experiment designer could use any parts of a header that are likely to be immutable or—for data packets—use the payload. To handle cases where the nonce is in the payload and middleboxes insert header fields (e.g. options), the nonce offset can also be provided from the end of the packet (decided by optionally including a "fromBack" attribute in the json file). The default nonce length is 16 bit and the default position is the mostly unused Identification field in the IP header.

We chose a field of the IP header as a default position because we cannot make a default assumption about packet headers following IP; for *fling*, *anything* can be there. This also explains the need to specify details about a checksum ("packet_Info" in figure 1): if the nonce is written into a transport header or payload, this header's checksum will need to be recalculated. The value cannot be known beforehand because the nonce is calculated at run time.

fling allows a minimum nonce length of 8 bit (0 bit for the random numbers; such experiments can only contain one packet from each side). Because the nonce can be so small, the server also limits its responses to tests from the same IP address that was used on the HTTPS connection. Using a short nonce increases the chance for an attacker that is behind the same NAT as a regular *fling* client (or changes its source address to match an ongoing *fling* test) to inject wrong packets that the server would be forced to accept.

In a truly end-to-end Internet, the transport header should not matter to routers and *fling* should be able to do whatever it wants on top of IP. This notion is, however, already broken by NATs, which, in practice, often carry out NAPT (Network Address and Port Translation) [42]. When a client sends packets of a known transport protocol (e.g. TCP or UDP) from behind a NAT, the server cannot just apply a statically-defined transport header, but it needs to swap the NAT-written port numbers. Whether this functionality is desired or not depends on the packet format (e.g., ICMP does not have ports). Therefore, if "packet_Info" in the test description contains "swap: [*loc1*,*loc2*,*len*]" where *loc1* is the location of source port, *loc2* is the location of the destination port and *len* is the length of ports in bytes, the server sets the ports accordingly instead of taking them from the pcap file. This is another reason to carry out a checksum calculation. The attributes "swap" and "ChksumPos" are specified relative to the front of the transport header.

"swap" is in fact shorthand for two "copy" operations that copy fields from previously arrived packets. An example from our measurement campaign that highlights the need of such a generic "copy" operation is discussed in section IV-A.

C. Narrowing Down the Root Cause of Packet Drops

Given that *fling* is about reachability and tries to detect what middleboxes do to packets, we should be able to detect the rough location of middleboxes that change or drop packets. Moreover, if a *fling* packet is dropped on a path, we need to ensure that it happened due to the middlebox's behavior, not due to congestion. Hence, we rely on using additional packets that we call *anchor packets*. Our anchor packets are either of type ICMP Echo Request or TCP SYN, and we answer them with ICMP Echo Reply or TCP SYN/ACK packets, respectively. Since NAT boxes use the *id* and *seq* fields of the ICMP header to map ICMP reply to request packets, we also maintain the correct values for the *id* and *seq* fields of the request and reply packets [41]. Since both the client and server need to be able to associate anchor packets to their corresponding *fling* packets, we store the last 16 bits of the nonce value in the IP header's 16-bit Identification field of the anchor packet (if the nonce is smaller, the remaining bits are set to 0). Note that any kind of anchor packet can be defined as part of the test description itself, making the test description slightly more complex but allowing full flexibility for the placement of the nonce.

Anchor packets have strictly two purposes: 1) detect congestion, 2) trigger a tracebox-like test. They are *not* meant as a replacement for A/B-measurements, where type A packets would differ from type B packets in a particular way and it could be established that the difference between A and B caused an effect. Such measurements can easily be designed with *fling* by including both type A and B packets in a test description. A *fling* test could also easily define its own special packets to be sent in conjunction with all other measurement packets ("send" in the json syntax operates on arrays, making it easy to send multiple packets), as a way of implementing customized anchor packets. We hard-coded ICMP Echo Request/Reply and TCP SYN-SYN/ACK purely for convenience.

We send an anchor packet immediately ahead of every outgoing *fling* packet. The idea is that repeatedly losing both packets together gives an indication that the packets were dropped due to congestion (refer to Table II). More importantly, if the anchor packet repeatedly arrives but the *fling* packet does not, this is a strong indication of a middlebox-induced packet drop. Thus, if a *fling* packet is dropped we repeat the test up to three times. This number is configurable.

Fig. 3 shows how the client detects a packet drop using anchor packets. After knowing that a *fling* packet was dropped, we try to find the location of the drop by repeatedly sending the same *fling* packet with a growing TTL starting from 1. As soon as no ICMP TTL exceeded error packet arrives, we know the location of the packet drop. Because we send the results of these tests back to the server in the final HTTPS exchange, the server can also identify changes that may have happened to the *fling* packet given that responding routers are RFC1812-compliant [10]. This approach is similar to *tracebox* [17].

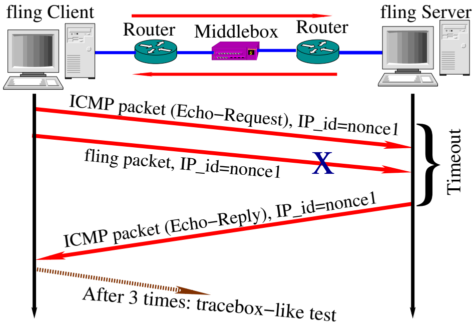

fling Client
Router
Middlebox
Router
fling Server
ICMP packet (Echo−Request), IP_id=nonce1
fling packet, IP_id=nonce1
X
Timeout
ICMP packet (Echo−Reply), IP_id=nonce1
After 3 times: tracebox−like test

Figure 3: Detection of *fling* packet drop and middlebox location

***fling* packet**	***anchor* packet**	**Interpretation**
PASSED	PASSED	SUCCESS
PASSED	DROPPED	SUCCESS
DROPPED	PASSED	Repeat 3×. Then, assume: MIDDLEBOX DROP; start tracebox-like test
DROPPED	DROPPED	Repeat 3×. Then, assume: CONGESTION; start tracebox-like test

Table II: Interpretation of arriving or dropped *fling* / *anchor* packets. CONGESTION: in one type of *fling* test, *anchor* packet passes and in another, it is dropped. SUCCESS: all *fling* packets passed; if packets have changed in transit, start tracebox-like test.

IV. Evaluation

We have prototyped *fling* in Python, based packet capturing and manipulation on Scapy [7], and prepared tests for 36 distinct protocols and protocol options. By preparing a test, we mean that we have generated all needed packet traces to test a protocol or an option. We then asked friends and colleagues to run *fling*. In total, 34 users ran *fling*, and each test was run against three servers simultaneously to avoid any server-related bias, giving us a total of 3384 tests. One of the three servers was hosted on Amazon EC2 cloud and the other two were hosted in Norway, where they were connected to the Internet via two different ISPs. The tests originated from nine countries with over half of them coming from Norway and Austria. Further, about two thirds of users stated that they ran the tests from their homes.

Next, we present our tests and the trial run results. Note that the goal of these runs is not to make a general statements about the support of specific protocols in the Internet, but rather to examine *fling*'s flexibility. We, however, plan to conduct large scale measurement campaigns in the future.

A. Test Design

To evaluate the flexibility of *fling*, we first analyzed middlebox measurements from previous work to try to understand whether we could replicate them. These measurements include:

- The ECN tests from TBIT [34], [35], [38], [39] and [30].
- The TBIT TCP options tests, and the IP options tests from TBIT and [21].
- The TCPExposure [25] tests: using *MP_CAPABLE*, *MP_DATA*, *MP_ACK*, the Timestamps option, and a TCP handhake followed by data and ACK packets (containing SACK) to check for sequence number changes.
- The HICCUPS [16] tests, where special numbers are inserted in the sequence number, IP Identification and receive window fields to convey integrity information.

We find that *fling* can replicate almost all of these tests, with only very minor limitations: because it relies on the nonce, *fling* cannot detect TCP splitters in the same way TCPExposure does; it does however detect them during its tracebox-like test phase. In some cases, a dynamic decision taken in a test must be replaced with separate static tests. For example, two static tests for SYN-SYN/ACK handshaking with or without ECN set-up are needed to replicate the TBIT ECN test; similarly, two tests are needed to conditionally answer a packet containing the *MP_DATA* option with *MP_ACK* or not, depending on whether *MP_DATA* passed through the path (this is done by TCPExposure). Finally, in HICCUPS [16], the values of the altered fields in the SYN/ACK packet are the result of a computation on the received SYN packet. Because it does not allow to define arbitrary calculations on header fields, *fling* cannot truly replicate the behavior of HICCUPS, but it can detect all the header changes that HICCUPS also detects.

Next, we decided to run a variety of protocols over IP, and do tests with changes applied to the IP header (e.g. testing options or unknown protocol numbers) and the TCP header (e.g. testing options or using a wrong value in the Data Offset field). The complete set of tests that we carried out is listed in Table III; these were pure *fling* tests, i.e. packets of the described type were transmitted between the clients and the server to see what would happen to them along the path. This table also shows how many tests succeeded. We only decided that a test failed when all three repetitions failed; if, in these three failures, anchoring ICMP messages were dropped but ICMP messages passed from the same client at least once in another test, we decided that this failure could have been due to congestion and removed the test from our set. There was only one such case.

Again, the static nature of *fling* highlighted a handful of limitations: RSVP requires to put IP addresses and port numbers in the RSVP header. At first, we could not do this; this prompted us to devise the aforementioned generic "copy" operation. Some protocols, however, require calculations, which *fling* cannot do: a Quick-Start [20] recipient (server) should generate entries in its response by computing the TTL difference as (IP_TTL–QS_TTL) mod 256, and retrieving the allowed rate request from the received IP option. Similarly, the AH *Integrity Check Value (ICV)* in the OSPF/AH test is wrong because *fling* changes the underlying IPv4 header but does not recalculate this value (section 3.3.3 of [28]).

Complete flexibility can only be attained by installing new code for each test, or allowing to specify an actual protocol (arbitrary operations on header fields based on prior received headers and local state). With *fling*, we intend to strike a balance in trying to be simple yet flexible. We conclude from our test design study that, despite being unable to do absolutely all tests, the number and diversity of tests that *fling* can do is indeed large: it was able to replicate the large majority of tests from existing work and allowed us to carry out truly "crazy" tests involving e.g. changes to the IP header that follow an obsolete specification, wrong field combinations in TCP, or transmit OSPF, HIP and DCCP packets end-to-end.

Next, we briefly evaluate the results of our tests. Our test set is small: at this stage, our intention is to test-drive *fling* before we roll it out on a larger scale. Thus, we do not try to derive broad statements about the Internet from our measurements, but we want to ensure that such statements *could* be derived in a larger-scale study.

B. *Results*

Table III shows the tests that we carried out. Unless otherwise noted, all protocols were used directly over IP, and Scapy defaults apply to header fields. IP and TCP header tests used Scapy-generated packets: a TCP SYN from client to server (src port 48001, dst port 443), and a TCP SYN/ACK in response. The "Success ratio" column shows the fraction of successful runs (i.e., all *fling* packets reached the other side within 3 trials) for each test.

As expected, no protocol has a 100% success rate except for UDP. Broadly speaking, our tests can be classified into four categories: IP header changes, IP protocols, TCP options, and transport protocols other than TCP. Packets with IP header changes were among the least likely to pass end-to-end, confirming earlier observations about IP options [21]. While an unknown TCP option worked in 50% of the cases, a wrong Data Offset value worked in only 5%. This indicates that the success of an unknown option does not mean that middleboxes do not look inside the TCP header; they *do* look, but they often allow unknown options to pass.

SCTP appears to enjoy a decent support with 2/3 of the tests succeeding; this Internet test confirms the local testbed result in [22]. We inspected all failed tests and found that in over 95% of cases a test failed to all three servers, which indicates that the packet drop happened close to the client. IP option tests succeeded only from one client to one server, which both were from the same autonomous system, indicating different blocking policies for intra and inter domain traffic.

As mentioned above, once a server or a client detects that a test has failed, it attempts to determine the packet drop location by repeatedly sending the dropped packet with a growing TTL starting from one. Figure 4 shows the packet drop location in terms of the number of hops. These plots show that most of the blocking either happens at the client's immediate gateway or two hops away. Further, there is a significant fraction of blocking that happens several hops away from the client on the forward path. This, however, is not the case on the reverse path. Since the tests are always client-initiated, the above observations hint that whenever a test succeeds on the forward path, it is likely to succeed on the reverse path unless the client's immediate gateway blocks incoming packets. Going forward, we plan to take a closer look at what causes the blocking several hops away from the client and add functionalities to *fling* that—depending on user consent—fingerprint home gateways.

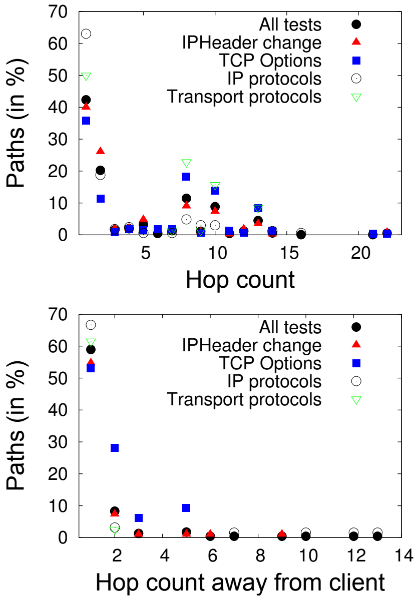


Figure 4: Packet drop location: forward path (top), and reverse path (bottom). The reverse path was determined from the server, but it is shown as the number of hops from the client.

Since *fling* servers and clients capture all exchanged packets, we can also investigate whether packet headers and options were modified. Inspecting packets from successful TCP tests, we find several cases of altered TCP MSS options and option removal. For example, the reserved TCP option 100 was removed in 14 experiments and the MPTCP option was removed in three experiments.

V. Conclusion and Next Steps

Our prototype tests have shown that *fling* is indeed very flexible, allowing for a wide range of middlebox tests. Our tool produces data that lets us better understand if, how, and where middleboxes influence packets of a certain type as a result of the dialogue that they witness.

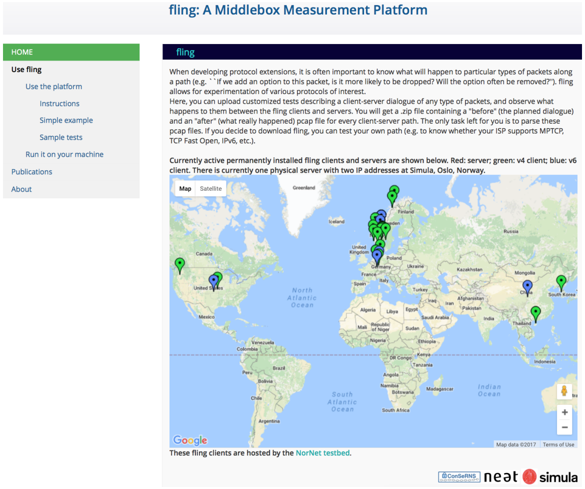
A. *Towards a permanent fling platform*

So far, we have described and used *fling* as a tool that a user downloads and runs once, yielding information about the network path between the user's host and our server.[3] For the "test-drive" measurement campaign described in this paper, we had to ask users to run the tool, similar to how TCPExposure tests were carried out [25]. The true goal of

[3] This "one-time" *fling* version is GPL licensed and available from the main *fling* webpage.

Test description	**Success ratio**
Routing and Addressing – **Initiation of a HIP session:** *I1* (a HIP Initiator Packet) is sent to the *fling* server, and *R1* (HIP Responder Packet) is sent to the *fling* client. On success, the client sends *I2* and in response, it receives *R2* (Section 5.3 of [36]). We ran the HIP package from [1] to collect HIP packets and observed HIP packets over UDP, which we extracted to run HIP over IP.	0.51
QoS – **Initiating an RSVP reservation:** The *fling* client sends a *Path* message to the server, and the server responds with a *Resv* message. We used two packets from the pcap file from [4].	0.0
Tunneling – **ICMPv6/IPv6:** ICMPv6 *Echo request* with non-existing IPv6 address pair is sent to the *fling* server which responds with *Echo reply*. We used 2 packets from the pcap file from [2].	0.5
ICMPv4/IP/GRE: We set *checksum present* = 0 in the GRE header. An ICMPv4 *Echo request* packet is put in an IP/GRE tunnel and sent to the server. The server responds with an ICMPv4 *Echo reply* packet over IP/GRE. We used 2 packets from the pcap file from [2].	0.66
Security – **ICMPv4/IP/ESP (Tunnel Mode):** The client sends an ESP packet containing an ICMPv4/IP *Echo request* packet as encrypted payload. The server responds with an ESP packet containing an ICMPv4/IP *Echo reply* as encrypted payload. We used two packets from the pcap file from [8], where it is explained how to decrypt the data stream. We did this to identify the packets.	0.649
AH (Transport Mode): The *fling* client sends an AH packet to the *fling* server, and the *fling* server answers with an AH packet. We used two packets from the pcap file from [2].	0.585
Transport – **SCTP association establishment:** The client sends an SCTP *INIT* packet with src port 48001 to the server on port 443, and the server answers with an *INIT_ACK* packet. Then, the client responds with *COOKIE_ECHO* and the server responds with *COOKIE_ACK*.	0.66
UDP: The client sends a UDP packet with src. port 48001 to the server on port 443, containing 4 bytes of data; the server responds with a similar UDP packet (but flipped ports).	1.0
UDP-Lite: The client sends a UDP-Lite packet with source port 32768 and destination port 1234, with *checksum coverage* = 0, containing 12 bytes of data; the server responds with a similar UDP-Lite packet (but flipped ports). We used two packets from the pcap file from [4].	0.511
Complete DCCP session: The client sends a DCCP *Request* packet with src port 32772 to port 5001 and the server answers with a *Response* packet. The client answers with an *Ack* and a *DataAck* packet containing 256 bytes of data. The server responds with an *Ack*. Finally, the client transmits a *Close* packet and the server responds with a *Reset* packet. We used the pcap file from [4], shortened the data transfer and adjusted the sequence numbers of the closing packets.	0.511
IPv4 options – **Quick-Start (QS) (QS request [20]):** The client sends a packet containing a QS *Request* option with IP TTL 64, *QS TTL* 90, IP option number 25, option length 8, *Function Value* 0, *Rate Request* 5 (1.28 Mbit/s), *QS Nonce* 2, and the *reserved* field (2 bits) set to zero. The server responds with a QS *Response* option as a TCP option in the SYN/ACK packet with *Rate Request* 5, *TTL Diff* 5, *QS Nonce* 2, and all *reserved* fields = 0.	0.053
Router Alert [26]: The same option was used on both packets, with *value* set to zero.	0.053
Security and Extended Security (historic) [27]: We made two tests: one for *Basic Security* where *Classification Level* is set to *Secret* and *SCI* and *NSA* bits set for *Protection Authority Flags*; and another for *Extended Security* (zero is set for security info and its code) along with *Basic Security* options. Both SYN and SYN/ACK packets carry these options.	0.011
Other IPv4 header changes – **DiffServ CodePoint (DSCP):** We tested some DSCP values from [18] which proposes to opportunistically set them for WebRTC: CS1 (8), AF42, EF PHB (46).	0.75
“Evil” bit [12]: The client and server exchange SYN-SYN/ACK packets with this bit set.	0.745
Unknown Protocol numbers (143, 200, 252, 253, 255): The client sends a TCP SYN and the server responds with a TCP SYN/ACK, but both use the same unknown protocol number.	0.51
TBIT test 2, IP Option X (option number 31): We carried this test out statically, i.e. without retrying three times in case of failure.	0.053
IP options tests from [21]: The client sends a TCP SYN with an IP option to the server on port 80 and the returns it on a TCP SYN/ACK. We generated the packets with *Scapy*.	0.06
TCP header changes – **TCP Fast Open (TFO) [14]:** We downloaded the pcap file from [3], removed two unnecessary packets (an intermediate GET and a final ACK) and edited the packets to test option kinds 34 (newly allocated) and 254 (experimental). For 254: The client sends a SYN packet with *magic number* 63881 and requests a *cookie* (kind 34 does not use a magic number, but requires padding). The server responds with a SYN/ACK packet containing the same *magic number* and a *cookie*. The client then sends an HTTP request inside a SYN/Fast Open packet with the same *magic number* and *cookie* value. The server responds with a SYN/ACK.	0.628
TCP mood (Test for TCP Happy packet (April 1 RFC [23])): The *fling* client sends a SYN packet with an option of kind 25 and value 14889 (“:)” in ascii) for *Happy* mood, and the server sends a SYN/ACK with a happy mood too.	0.755
TCP NoP (Test for NoP option / wrong Data Offset): This tests what happens if the Data Offset value in the TCP header is wrong, both for a SYN from the client and the corresponding SYN/ACK from the server. The IP *Length* field says that the packet is 42 bytes long and the TCP Data Offset value is 6, meaning 24 bytes. In reality, there are 20 bytes of regular TCP header, 2 bytes of NoP options, and the last 2 bytes do not exist in this packet.	0.053
TCP unknown option (Test for a large unknown TCP option using reserved option number 100): The client sends a SYN packet with option kind 100, a correct length field and 40 bytes option content; the server sends a SYN/ACK with the same option.	0.5
TBIT-based ECN test: The client sends a SYN packet with set ECN_ECHO and CWR flags to a web server on port 80, and the server sets the the ECN_ECHO flag in the SYN/ACK response. This handshake is (in TBIT, only in case of successful ECN negotiation) followed by an HTTP request with ECT and CE set in the IP header.	0.66
TCPExposure-based MPTCP tests: The client sends a SYN packet containing the *MP_CAPABLE* TCP option to the server; the server inserts the *MP_CAPABLE* option in its SYN/ACK response. This is followed by a TCP data packet from the client containing the *MP_DATA* option, which the server answers using an *MP_ACK* option in its ACK packet.	0.745

Table III: Tests with success ratios (how often packets passed through the network in both directions).



fling: A Middlebox Measurement Platform

HOME
Use fling
Use the platform
Instructions
Simple example
Sample tests
Run it on your machine
Publications
About

fling

When developing protocol extensions, it is often important to know what will happen to particular types of packets along a path (e.g. ``If we add an option to this packet, is it more likely to be dropped? Will the option often be removed?"). fling allows for experimentation of various protocols of interest.
Here, you can upload customized tests describing a client-server dialogue of any type of packets, and observe what happens to them between the fling clients and servers. You will get a .zip file containing a "before" (the planned dialogue) and an "after" (what really happened) pcap file for every client-server path. The only task left for you is to parse these pcap files. If you decide to download fling, you can test your own path (e.g. to know whether your ISP supports MPTCP, TCP Fast Open, IPv6, etc.).

Currently active permanently installed fling clients and servers are shown below. Red: server; green: v4 client; blue: v6 client. There is currently one physical server with two IP addresses at Simula, Oslo, Norway.

These fling clients are hosted by the NorNet testbed.

ConSeRNS neat simula

Figure 5: The front page of the *fling* platform

fling, however, is to be able to carry out tests without always having to personally interact with users—we made *fling* simple on purpose to increase the chance of convincing users to install it as a permanent piece of software or deploying it in other measurement platforms.

To this end, we have already begun to develop a *fling* platform that follows the described usage scenario. Figure 5 shows a screenshot of the front page.[4] The world map shows currently active "permanent *fling*" hosts, which are provided to us by the NorNet testbed.[5] NorNet nodes are multi-homed, meaning that a marker on the map translates into 2-3 different IP addresses, connected to different networks (around 4-6 for IPv6-capable hosts, which are shown in blue). These hosts are clients, configured to pull and run the current set of tests every hour. For now, there is only one server, in Oslo, Norway; however, we are planning to use many other nodes as servers as well (as part of the initial handshake with the Fling Control Channel, the main server can inform a "permanent *fling*" client about other servers that it should carry out its tests with).

After registering, users can run their own tests on the *fling* platform. To do this, a user designs a json test description and a pcap file containing the packets used in the dialogue. Instructions are provided on the website; the easiest way to create a test is by downloading and editing one of our (currently 44) examples, which also are the tests that *fling* already runs by default. Next, the user clicks "Use the platform", where (s)he can login using her/his credentials. This leads to a private space where users can upload a test, run it and obtain the result (typically after 1-2 hours). The output is provided as a zip file that contains client- and server-side pcap files for every path that was tested. We are currently developing an auto-generated summary text file to be included in the zip file, containing more information about the test.

B. *Scaling up*

To make our platform grow in scale, we see three major requirements that we need to satisfy:

1. Making it attractive to use: The website must make it clear that *fling* is useful and easy to handle. In addition to offering instructions and example tests, we have therefore created a page ("Simple example" in the menu) that lets users interactively test *fling* on the spot, from the browser. The page shows a diagram with a 3-way TCP handshake followed by a data packet sent over TCP; when a user clicks the IPv4 and TCP headers in the dialogue, the headers are shown with many editable fields. Next to the dialogue, a map shows the locations of the two special clients that we have assigned to participate in this interactive test; one of them is behind a NAT. After changing header fields at will, a user can press "Run the test", wait for a few seconds and obtain output that shows which packets were dropped and/or fields have changed. Our "one-time *fling*" client is also available for download. It is easy to run and produces an immediate output with interesting facts about the user's own Internet connection.

2. Making it attractive to install: The "permanent *fling*" client for end users is currently under development. It will be secure: by definition, *fling* initially only corresponds with our own trusted server, which is the only server that is allowed to redirect a client to other servers (users however need to accept that arbitrarily "strange" packets are transmitted between their client and *fling* servers; we will inform users about this as they download the permanent client). It will be possible to limit the total number packets that the client is willing to send and receive per hour (to limit overall load) and per second (to prevent *fling* tests from interfering with the user's other traffic). We have discussed more security aspects in section III. To make the tool itself appealing, we are considering different options on how the client should present itself (e.g. a user interface that provides information about the current connection's state and informs users about recent "special" tests). Additionally, as the platform grows, we may introduce a credit system like the RIPE Atlas [43] or Seattle [49] to only allow users to upload new tests if they have run the permanent client for some time.

3. Ensuring scalability: Our server in Oslo serves as a trusted entry point to the system; it is also the place where we collect measurement results. This makes our server the most critical element in the infrastructure. *fling* is lightweight; because all tests descriptions are pulled from the server, a simple server update suffices to limit the number of packets or the length of a *fling* measurement (and we can impose such limits on tests that users upload). We will also implement a measure to ensure that the communication with *fling* servers is spread out in time. Instead of requiring servers to remember clients and schedule their access time, we plan to implement a distributed collision avoidance strategy from sensor networks,

[4] This page is available at http://fling-frontend.nntb.no. Here, we will also make the source code of all *fling* components available, and we are planning to share anonymized datasets as well as a number of select permanent measurement results in order to identify long-term trends.

[5] https://www.nntb.no

where communication also happens at regular intervals but these intervals can differ between clients [15].

fling began with our wish to have more informed discussions in the Internet Engineering Task Force (IETF): we believe that it would be fantastic if theories on what middleboxes would do to certain protocols or header fields could spontaneously and convincingly be tested by uploading a test description to a platform, waiting for a few hours and getting a result. This would require a platform that is as flexible and easy to use as *fling* already is, but larger. Having covered the "flexibility and ease of use" requirement, our immediate next step is to increase the scale of the platform in accordance with the plans that we have laid out. The recent creation of the IRTF Research Group on Measurement and Analysis for Protocols (MAPRG) shows that the IETF is certainly interested in the more informed dialogue about middlebox behavior that we envision.

VI. Acknowledgements

This work was partially funded by the European Union's Horizon 2020 Research and Innovation Programme through A New, Evolutive API and Transport-Layer Architecture for the Internet (NEAT) project under Grant Agreement no. 644334. The views expressed are solely those of the authors.

References

[1] "http://openhip.sourceforge.net."
[2] "http://packetlife.net."
[3] "https://redmine.openinfosecfoundation.org."
[4] "https://wiki.wireshark.org."
[5] "https://www.samknows.com."
[6] "http://tcpreplay.appneta.com/."
[7] "http://www.secdev.org/projects/scapy."
[8] "http://www.spiceupyourknowledge.net/2012/11/decrypting-esp-packet-using-wireshark.html."
[9] V. Bajpai and J. Schönwälder, "A Survey on Internet Performance Measurement Platforms and Related Standardization Efforts," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 3, 2015.
[10] F. Baker, "Requirements for IP Version 4 Routers," RFC 1812 (Proposed Standard), Jun. 1995.
[11] R. Barik, M. Welzl, and A. Elmokashfi, "How to Say That You'Re Special: Can We Use Bits in the IPv4 Header?" in *ANRW '16*, 2016.
[12] S. Bellovin, "The Security Flag in the IPv4 Header," RFC 3514 (Informational), Internet Engineering Task Force, Apr. 2003.
[13] A. Botta and A. Pescap, "Monitoring and measuring wireless network performance in the presence of middleboxes," in *WONS 2011*, Jan 2011, pp. 146–149.
[14] Y. Cheng, J. Chu, S. Radhakrishnan, and A. Jain, "TCP Fast Open," RFC 7413 (Experimental), Internet Engineering Task Force, Dec. 2014.
[15] R. L. Cigno, M. Nardelli, and M. Welzl, "SESAM: A semi-synchronous, energy savvy, application-aware MAC," in *WONS 2009*, Feb 2009.
[16] R. Craven, R. Beverly, and M. Allman, "A Middlebox-cooperative TCP for a Non End-to-end Internet," in *SIGCOMM '14*, 2014, pp. 151–162.
[17] G. Detal, B. Hesmans, O. Bonaventure, Y. Vanaubel, and B. Donnet, "Revealing Middlebox Interference with Tracebox," in *IMC '13*. ACM, 2013.
[18] S. Dhesikan, D. Druta, P. Jones, and C. Jennings, "DSCP Packet Markings for WebRTC QoS," Internet Engineering Task Force, Internet-Draft draft-ietf-tsvwg-rtcweb-qos-18, Aug. 2016, Work in Progress.
[19] K. Edeline, M. Kühlewind, B. Trammell, E. Aben, and B. Donnet, "Using UDP for Internet Transport Evolution," ETH, Tech. Rep. ETH TIK Technical Report 366, Dec. 2016.
[20] S. Floyd, M. Allman, A. Jain, and P. Sarolahti, "Quick-Start for TCP and IP," RFC 4782 (Experimental), Internet Engineering Task Force, Jan. 2007. [Online]. Available: http://www.ietf.org/rfc/rfc4782.txt
[21] R. Fonseca, G. M. Porter, R. H. Katz, S. Shenker, I. Stoica, R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "IP options are not an option," University of California, Berkeley, Tech. Rep. UCB/EECS-2005-24, 2005.
[22] S. Hätönen, A. Nyrhinen, L. Eggert, S. Strowes, P. Sarolahti, and M. Kojo, "An Experimental Study of Home Gateway Characteristics," in *IMC '10*, 2010.
[23] R. Hay and W. Turkal, "TCP Option to Denote Packet Mood," RFC 5841 (Informational), Internet Engineering Task Force, Apr. 2010.
[24] M. Honda, "Lessons Learnt from Middlebox Measurement," in *Proceedings of IETF-93, Presentation to HOPS RG*, 2015.
[25] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda, "Is it still possible to extend TCP?" in *IMC '11*. ACM, 2011.
[26] D. Katz, "IP Router Alert Option," RFC 2113 (Proposed Standard), Internet Engineering Task Force, Feb. 1997.
[27] S. Kent, "U.S. Department of Defense Security Options for the Internet Protocol," RFC 1108 (Historic), IETF, Nov. 1991.
[28] ——, "IP Authentication Header," RFC 4302 (Proposed Standard), Internet Engineering Task Force, Dec. 2005.
[29] C. Kreibich, N. Weaver, B. Nechaev, and V. Paxson, "Netalyzr: Illuminating the Edge Network," in *IMC '10*, 2010, pp. 246–259.
[30] M. Kühlewind, S. Neuner, and B. Trammell, "On the State of ECN and TCP Options on the Internet," in *Proceedings of PAM'13*, 2013, pp. 135–144.
[31] I. R. Learmonth, B. Trammell, M. Kühlewind, and G. Fairhurst, "PATH-spider: A Tool for Active Measurement of Path Transparency," in *ANRW '16*, 2016.
[32] A. M. Mandalari, M. Bagnulo, and A. Lutu, "Informing Protocol Design Through Crowdsourcing: the Case of Pervasive Encryption." ACM SIGCOMM Workshop on Crowdsourcing and crowdsharing of Big (Internet) Data (C2B(I) D), Aug. 2015.
[33] S. McQuistin and C. S. Perkins, "Is Explicit Congestion Notification Usable with UDP?" in *IMC '15*. ACM, 2015, pp. 63–69.
[34] A. Medina, M. Allman, and S. Floyd, "Measuring Interactions Between Transport Protocols and Middleboxes," in *IMC '04*, 2004, pp. 336–341.
[35] ——, "Measuring the Evolution of Transport Protocols in the Internet," *SIGCOMM Comput. Commun. Rev.*, vol. 35, no. 2, pp. 37–52, Apr. 2005.
[36] R. Moskowitz, T. Heer, P. Jokela, and T. Henderson, "Host Identity Protocol Version 2 (HIPv2)," RFC 7401 (Proposed Standard), Internet Engineering Task Force, Apr. 2015.
[37] A. Müller, F. Wohlfart, and G. Carle, "Analysis and Topology-based Traversal of Cascaded Large Scale NATs," in *HotMiddlebox '13*, 2013.
[38] J. Padhye and S. Floyd, "Identifying the TCP Behavior of Web Servers," in *In ACM SIGCOMM*, 2000.
[39] J. Pahdye and S. Floyd, "On Inferring TCP Behavior," in *SIGCOMM '01*, 2001.
[40] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley, "How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP," in *USENIX NSDI'12*, 2012, pp. 29–29.
[41] P. Srisuresh, B. Ford, S. Sivakumar, and S. Guha, "NAT Behavioral Requirements for ICMP," RFC 5508 (Best Current Practice), Internet Engineering Task Force, Apr. 2009.
[42] P. Srisuresh and M. Holdrege, "IP Network Address Translator (NAT) Terminology and Considerations," RFC 2663 (Informational), Internet Engineering Task Force, Aug. 1999.
[43] R. N. Staff, "RIPE Atlas: A Global Internet Measurement Network," *Internet Protocol Journal*, vol. 18, no. 3, Sep. 2015.
[44] S. Sundaresan, S. Burnett, N. Feamster, and W. de Donato, "BISmark: A Testbed for Deploying Measurements and Applications in Broadband Access Networks," in *USENIX ATC 14)*, Jun. 2014, pp. 383–394.
[45] B. Trammell and M. Kühlewind, "Observing Internet Path Transparency to Support Protocol Engineering," in *Proceedings of IRTF/ISOC RAIM Workshop*, Oct 2015.
[46] Z. Wang, Z. Qian, Q. Xu, Z. Mao, and M. Zhang, "An Untold Story of Middleboxes in Cellular Networks," in *SIGCOMM '11*, 2011.
[47] N. Weaver, C. Kreibich, M. Dam, and V. Paxson, "Here Be Web Proxies," in *Proceedings of PAM '14*, 2014.
[48] X. Xu, Y. Jiang, T. Flach, E. Katz-Bassett, D. Choffnes, and R. Govindan, "Investigating Transparent Web Proxies in Cellular Networks," in *PAM '15*, 2015.
[49] Y. Zhuang, A. Rafetseder, and J. Cappos, "Experience with Seattle: A Community Platform for Research and Education," in *2013 Second GENI Research and Educational Experiment Workshop*, March 2013, pp. 37–44.

E Paper: *How to say that you're special: Can we use bits in the IPv4 header?*

The following research paper [11] has been produced by project participants.

How to say that you're special: Can we use bits in the IPv4 header?

Runa Barik, Michael Welzl
University of Oslo, Norway

Ahmed Elmokashfi
Simula Research Laboratory, Norway

ABSTRACT

The IP header should be the ideal part of a packet that an end system could use to ask the network for special treatment. Recently, there has been renewed interest in using bits of this header – e.g. the ECN and the DSCP fields. But can we really use these bits? Or should we try to use other bits? We contribute to the body of work that tries to answer these questions by reporting on IPv4 measurements regarding the DSCP field and the Evil bit. Our findings show unexpected treatment to packets that set either of these fields and also confirm recent results on IP Options and ECN.

CCS Concepts

•**Networks** → **Network measurement; Middle boxes / network appliances; Public Internet;**

Keywords

Middleboxes, Measurements, IPv4, DSCP, TCP

1. INTRODUCTION

The current Internet is full of middleboxes – devices that performing functions "other than the normal, standard functions of an IP router on the datagram path between a source host and destination host" [3]. For instance, a recent study [16] analyzing 57 enterprise networks revealed that they contain as many middleboxes as routers. The authors of [19] found that 82 out of 107 cellular networks have NAT devices. Measuring what these middleboxes do to packets has been a matter of much recent interest, and it is important, e.g. when designing protocol extensions in the IETF (e.g., [10] had an impact on the design of MPTCP).

However, in-band (per-packet) signaling from end systems to the network should ideally be done in the IP header – the part of the packet that any intermediate device, be it a middlebox or a regular router, *should* be able to analyze and modify. Recent IETF proposals utilize the bits of this header for such purposes – e.g. [5] defines how web browsers should directly set DiffServ Code Point (DSCP) in order to obtain a more suitable service for packets. Another example is the ECN field, which has been overloaded for various purposes (e.g. PCN [6] and ConEx [14]) – recently, it has been suggested to segregate traffic into two different queues depending on the value of this field [2]. The potential difficulty of using the IP header for signaling has also fueled work on other means for in-band signaling between end systems and network, e.g. SPUD [18].

Addressing this need, we present some measurement results that focus on IPv4 header fields, specifically the DSCP and the "Reserved" bit in the IP header – commonly, and in the following, called the "Evil bit" ([1], April 1). While directly setting the DSCP is now being proposed for WebRTC, the Evil bit may also become an opportunity for usage when we run out of available bits.

Our measurements, for which we asked private contacts to run a tool to communicate over raw sockets with our servers, point at some unexpected behavior regarding both DSCP (which can cause packet drops) and Evil bit (which works better end-to-end than the tested DSCP values). Our measurements also roughly confirm some previously published results regarding ECN and IP Options, and show a positive result regarding (mis-)use of Identification (ID) field as a side effect. We elaborate more on this in Sec. 4

2. TEST DESIGN

We implemented a tool based on *scapy* and Python *httplib*; the tool has both client and server side components. The tool executes a pre-specified exchange pattern between the client and the server. For each test, we prepared a packet trace and uploaded it to both the client and the server along with a description of how to exchange these packets. This flexibility allows testing different combinations of flags and options in the IP header.

In May 2016, we carried out a total of 1807 TCP SYN-SYN/ACK handshakes across 185 paths (IP address pairs), using various combinations of IP header flags and IP options. For some tests (e.g. ECN), the handshake was succeeded by an HTTP GET request, followed by an ACK. 35 people in 9 different countries Australia, Austria, Bangladesh, Germany, Norway, Spain, Sweden, Switzerland and United Kingdom installed and ran our scapy-based tool, which carried out several protocol dialogues over raw sockets with our 3 servers (15 hosts only communicated with 2 servers because the test was interrupted). Answering a query from our tool, about two thirds of the users stated that they ran the tests from their homes. One of our servers was based in Oregon (USA), the other two were based in Norway. We intend to do broader tests in the future, including differentiation between mobile and fixed networks.

To minimize the chance that congestion-based drops make us believe in a failure to communicate when using certain values in the IP header, we re-tried failed packet exchanges up to three times, and we sent an ICMP packet just ahead of every measurement packet. We only assumed a communication failure when the test failed three times and the ICMP packet succeeded.

ANRW '16, July 16 2016, Berlin, Germany

© 2016 ACM. ISBN 978-1-4503-4443-2/16/07…$15.00

DOI: http://dx.doi.org/10.1145/2959424.2959442

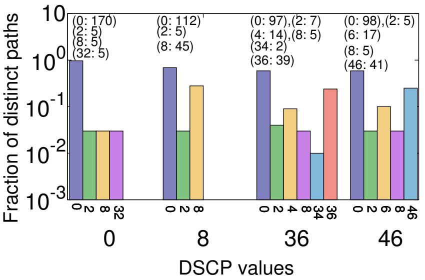


Figure 1: *DSCP value changes. x-axis: the lower (larger) number is the DSCP value that our senders used, the upper (smaller) number is the value that arrived at receivers. The brackets on the top show the absolute number of paths (IP address pairs) along which the change happened.*

3. RESULTS

Since our DSCP evaluation was motivated by the WebRTC QoS proposal [5], we used this Internet-draft to guide our tests and will discuss our results in its context. Considering table 1 in [5], we assumed the flow type "Interactive Video with or without Audio" with application priorities "Very low" (DSCP value CS1 (8)), "Low" (DF (0)) or "Medium" (AF42 (36)), and flow type "Audio" with application prioritiy "High" (EF PHB (46)).

Figure 1 shows the DSCP value changes that we saw. The most expected behavior is that DSCP values pass unchanged or are zeroed. Our results confirm this expectation, as the number of paths where packets were received with their original value (0, 8, 36 or 46) or set to 0 was always largest (and, irrespective of the input value, receiving DSCP=0 seems to be the most common behavior by far). We did, however, see consistent packet drops too: on 23 paths for DSCP value 8, 21 paths for 36, 19 for 46. This failure rate of approximately 10 - 13% is a reason to be concerned about WebRTC QoS; implementations should probably react to consistent failures with a fall-back to DSCP value 0.

On five paths, any DSCP value was changed to 8 – "Very low" in accordance with the table in [5]. Another value that occurred irrespective of the input value was 2, which is undefined [11]. Given the small number of paths, there may only have been a single or a handful of devices that produced these values. Much more interestingly, however, certain DSCP values appeared *only* when the sender applied a nonzero DSCP value. Marking packets as AF42 (36) provoked another undefined value (4) on 14 paths, but also AF41 (34), giving it a lower drop precedence and thereby potentially improving the service. Value 46 (EF PHB), on the other hand, was turned into 6 – yet another undefined value – on 17 paths.

Using the Evil bit provoked consistent packet loss on 11% of all 185 distinct paths – the same approximate range as the DSCP values. Among the successful tests, we observed that the Evil bit was zeroed on 4% of all paths (6 out of 164). This number is much lower than for the DSCP, which was zeroed in 62% of all cases (307 in the total 492 tests of distinct paths per DSCP value, for values 8, 36 and 46). This is perhaps expected, given that the Evil bit has so far been undefined, but it also means that it probably has a better chance to "survive" along a path than the tested DSCP values.

To better understand whether this zeroing and the DSCP value changes (to defined values, which are more interesting because they should also have a defined effect on packets) were done by the same devices, we examined the geographical location of source and destination IP addresses. Table 1 shows that, e.g., the AF42-AF41 change happened for two different source/destination IP addresses pairs between Switzerland and Oregon, USA, and nowhere else, indicating that there was probably only one device in Switzerland that made this change. Similarly, all the changes to CS1 (8) happened on paths to Austria, indicating that there might only have been a single device in Austria that made this change.

Table 1: *DSCP and Evil bit changes by source / destination countries*

DSCP Change {# of paths}	Src. Countries	Dst. Countries

DF (0) -> CS1 (8) {5}
AF42 (36) -> CS1 (8) {5}
EF -> CS1 (8) {5}
	Norway (ISP1);
	Norway (ISP2);
	Oregon, USA
		Austria

AF42 (36) -> AF41 (34) {2}	Switzerland	Oregon, USA

CS1 -> DF (0) {112}
AF42 (36) -> DF (0) {97}
EF (46) -> DF (0) {98}
	Many	Many

Evil bit cleared {3}	ISP1;ISP2; Oregon, USA	Switzerland
Evil bit cleared {3}	Switzerland	ISP1;ISP2; Oregon, USA

4. DISCUSSION AND CONCLUSION

Our measurements have shown some interesting behavior regarding the DSCP and the Evil bit. Perhaps the most important take-away is that using a nonzero DSCP value can provoke consistent packet drops, and hence opportunistically using them as suggested in [5] should come with a fall-back to DSCP 0 in case consistent packet loss is seen. It was also interesting to see that using a nonzero DSCP value can provoke different DSCP value changes than using DSCP 0, potentially leading to different behavior than expected, but also indicating that the DSCP value is indeed understood and reacted upon by the routers in the network.

As for the Evil bit, setting it caused approximately the same amount of consistent packet loss as with the various DSCP values that we tried, but the bit value seemed to have a much better chance to be correctly transmitted across a path. This can indicate that the Evil bit is a better option than the DSCP value for definitions of new behavior (e.g. the proposal in [20]).

There are several other bits and fields in the IP header that deserve a closer look. In particular, the ECN field has been the subject of many investigations (cf. [13] and references therein), and IP Options have also been investigated to some extent (cf. [9, 15]). Our measurements roughly confirm previous findings regarding IP Options: we repeated the tests from [9] but also added the Quick-Start Request [8] and Router Alert [12] IP options, and saw less than 6% of successful tests on distinct paths. For ECN, we repeated a test from [15], which involved sending an HTTP GET request that had the ECN field set to 11 after a successful TCP+ECN handshake. We ended up submitting 108 such GET requests, out of which 91 successfully reached the other side on 69 different paths, i.e. the ECN field being set to 11 caused a drop rate of around 16%.

Contradicting its "allowed" usage [17], our tool used ID field to enumerate and identify packets of a test (we needed this for other measurements that we carried out in the same campaign). This means that we would categorize both a change of ID field or a drop of the packet as a packet drop in our tests. However, only one out of our 35 total test sources was entirely unable to communicate except for HTTPS signaling, which either points at an extremely restrictive middlebox behavior or failure to forward packets with an ID field value other than 0. Unless routers or middleboxes react to this field differently depending on other fields of the packet, this indicates a very large success rate when trying to send a value in the ID field across the Internet (3599 packets on 185 distinct paths), confirming a finding in [7].

Next, we plan to extend our tool with functionality similar to tracebox [4] such that we can learn the IP addresses of devices that caused packet drops or header changes, and give a better indication of the number of distinct devices that caused a certain behavior.

5. ACKNOWLEDGEMENTS

This work was partially funded by the European Union's Horizon 2020 Research and Innovation Programme through A New, Evolutive API and Transport-Layer Architecture for the Internet (NEAT) project under Grant Agreement no. 644334. The views expressed are solely those of the authors.

6. REFERENCES

[1] S. Bellovin. The Security Flag in the IPv4 Header. RFC 3514 (Informational), Apr. 2003.

[2] B. Briscoe, K. D. Schepper, and I. J. Tsang. Identifying Modified Explicit Congestion Notification (ECN) Semantics for Ultra-Low Queuing Delay. Internet-Draft draft-briscoe-tsvwg-ecn-l4s-id-01, Internet Engineering Task Force, Mar. 2016. Work in Progress.

[3] B. Carpenter and S. Brim. Middleboxes: Taxonomy and Issues. RFC 3234 (Informational), Feb. 2002.

[4] G. Detal, B. Hesmans, O. Bonaventure, Y. Vanaubel, and B. Donnet. Revealing middlebox interference with tracebox. In *Proceedings of the 2013 Conference on Internet Measurement Conference*, IMC '13, pages 1–8, New York, NY, USA, 2013. ACM.

[5] S. Dhesikan, D. Druta, P. Jones, and C. Jennings. DSCP Packet Markings for WebRTC QoS. Internet-Draft draft-ietf-tsvwg-rtcweb-qos-17, Internet Engineering Task Force, May 2016. Work in Progress.

[6] P. Eardley. Pre-Congestion Notification (PCN) Architecture. RFC 5559 (Informational), June 2009.

[7] K. Edeline and B. Donnet. Towards a middlebox policy taxonomy: Path impairments. In *2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 402–407, April 2015.

[8] S. Floyd, M. Allman, A. Jain, and P. Sarolahti. Quick-Start for TCP and IP. RFC 4782 (Experimental), Jan. 2007.

[9] R. Fonseca, G. M. Porter, R. H. Katz, S. Shenker, and I. Stoica. IP options are not an option. Technical report, EECS Department, University of California, Berkeley, 2005.

[10] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. Is it still possible to extend TCP? In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, IMC '11, pages 181–194, New York, NY, USA, 2011. ACM.

[11] Differentiated Services Field Codepoints (DSCP). http://www.iana.org/assignments/dscp-registry.

[12] D. Katz. IP Router Alert Option. RFC 2113 (Proposed Standard), Feb. 1997. Updated by RFCs 5350, 6398.

[13] M. Kühlewind, S. Neuner, and B. Trammell. On the State of ECN and TCP Options on the Internet. In *Proceedings of the 14th International Conference on Passive and Active Measurement*, PAM'13, pages 135–144, Berlin, Heidelberg, 2013. Springer-Verlag.

[14] M. Mathis and B. Briscoe. Congestion Exposure (ConEx) Concepts, Abstract Mechanism, and Requirements. RFC 7713 (Informational), Dec. 2015.

[15] J. Pahdye and S. Floyd. On inferring TCP behavior. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 287–298, New York, NY, USA, 2001. ACM.

[16] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 13–24, New York, NY, USA, 2012. ACM.

[17] J. Touch. Updated Specification of the IPv4 ID Field. RFC 6864 (Proposed Standard), Feb. 2013.

[18] B. Trammell and M. Kühlewind. Requirements for the design of a Substrate Protocol for User Datagrams (SPUD). Internet-Draft draft-trammell-spud-req-04, Internet Engineering Task Force, May 2016. Work in Progress.

[19] Z. Wang, Z. Qian, Q. Xu, Z. Mao, and M. Zhang. An untold story of middleboxes in cellular networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 374–385, New York, NY, USA, 2011. ACM.

[20] J. You, M. Welzl, B. Trammell, M. KÂÿhlewind, and K. Smith. Latency Loss Tradeoff PHB Group. Internet-Draft draft-you-tsvwg-latency-loss-tradeoff-00, Internet Engineering Task Force, Mar. 2016. Work in Progress.

Disclaimer

The views expressed in this document are solely those of the author(s). The European Commission is not responsible for any use that may be made of the information it contains.

All information in this document is provided “as is”, and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.