Performance Evaluation of NEAT Internet Transport Layer API and Library

Fredrik Haugseth

Thesis submitted for the degree of
Master in Informatics: Programming and Networks
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2018

Performance Evaluation of NEAT Internet Transport Layer API and Library

Fredrik Haugseth

Performance Evaluation of NEAT Internet Transport Layer API and Library

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

For many years, the only available transport protocols were TCP and UDP. More recently, innovative transport protocols like SCTP have been developed, and they can improve application performance. However, they are not widely used in the Internet partly due to the inflexible *BSD sockets API* that most network applications are programmed with. In particular, the API requires that the application specifies which transport protocol to associate with every network socket. Many middleboxes in the Internet support only TCP and UDP, which often leads application developers to use these protocols. If a protocol like SCTP is used, it is the responsibility of the programmer to devise a fallback method (e.g. use TCP instead) if it is not supported in the network path.

NEAT is a new transport layer networking library that provides a flexible platform- and protocol-independent API where the application can specify *transport services* instead of specific transport protocols. The *transport services* are transparently mapped to specific transport protocols internally in NEAT, which enables an applications to leverage new protocols without the need to re-code the application.

NEAT implements a *Happy Eyeballs* mechanism that probes the network for the support of transport protocols. Therefore, NEAT enables applications to easily leverage the best available protocols that match the specified *transport services*, and that are supported in the network path.

We find that NEAT introduces quite a lot of CPU overhead compared to using other state-of-the-art networking APIs. However, based on our analysis, a lot of this CPU overhead can be mitigated through optimizations in the code. We also find that NEAT introduces some memory overhead, but this overhead seems to be small and insignificant; especially in modern systems with abundant memory.

Contents

List of Figures

List of Tables

Preface

Acknowledgements

I would like to express my sincere gratitude and appreciation to my supervisor Dr. Naeem Khademi for providing me with invaluable feedback and guidance on research work and thesis writing. I want to thank you for your patience and for the long hours of discussions that has developed me both personally and professionally.

Further, I would like to thank Prof. Michael Welzl for his guidance, and for giving me pointers to relevant research works. Thanks to Prof. Stein Gjessing and Dr. Safiqul Islam for showing interest in my work and helping me in the writing process.

Thanks to the members of the Networks and Distributed Systems (ND) group at the Department of Informatics for making the group such a great place for collaboration and knowledge sharing. In particular, I would like to thank Marcel Marek for helping me with technicalities related to testbed and experiment setup, and Kristian A. Hiorth for helping me with troubleshooting the FreeBSD kernel.

Thanks to Prof. Michael Tüxen and Felix Weinrank from Münster University of Applied Sciences, Germany, for providing information about the SCTP protocol and the internal workings of the NEAT library.

A great thanks to family and friends who have supported me throughout the project. I would like to give out a special thanks to my parents for their endless support and for always believing in me no matter what.

I dedicate this thesis to my dear Juliane who has constantly supported and encouraged me throughout the thesis work. I could not have done this without you!

Chapter 1

Introduction

NEAT (A New, Evolutive API and Transport-Layer Architecture for the Internet) is a new, open-source, transport layer networking *Application Programming Interface* (API) and library that is designed to change the way network applications interact with the network [64]. In particular, the API is platform- and protocol-agnostic, meaning that the application does not specify which transport protocols or operating system mechanisms to use when communicating with other machines over the network. Instead, these details are handled internally by NEAT, and NEAT can therefore offer a cross-platform, uniform API for all operating systems. Currently, the reference implementation of NEAT [64] can be run on FreeBSD, Linux, OS X, and NetBSD.

In order for NEAT to select transport protocols and options internally, it requires the application to specify which network properties are required and desired when communicating over the network. It also selects protocols based on information about the current network and host characteristics. This information is maintained and updated internally by NEAT. Based on these inputs, the NEAT library can leverage the best available transport protocols and features that are available in the system[1], and that match the requirements of the application. In this way, developers can leverage novel transport protocols and advanced network services in their applications without having to re-code or re-design the application.

NEAT and similar libraries [87, 88] have recently been developed, and the goal is that they replace the existing *BSD sockets API* [83] that most of existing network applications are developed with. The reason why a new networking API is needed is that the BSD sockets API is too inflexible. When programming network applications with the BSD sockets API, most developers use either the *Transmission Control Protocol* (TCP) [RFC793] or the *User Datagram Protocol* (UDP) [RFC768] because these protocols are safe alternatives that will most likely work in the Internet. There exists other transport protocols like the *Stream Control Transmission Protocol* (SCTP) [RFC4960] that can offer other network services, but such protocols are not as widely supported in the Internet. If developers want

[1]Since NEAT is a user-space library, it can leverage both user-space and kernel-space protocols and libraries.

to leverage protocols like SCTP in their network applications, they will need to implement a fallback mechanism and use e.g. TCP in case another protocol is not supported. This introduces unnecessary complexity to the application logic, that can be handled by a more flexible library like NEAT. The hope is that libraries like NEAT will change the global traffic patterns if ubiquitously deployed, which can reenable the innovation and evolution of the Internet transport layer [71] and make other transport protocols than TCP and UDP more available to developers.

This thesis investigates whether NEAT-like systems can be widely deployed in the Internet, and how it will perform on a local machine with regards to resource utilization. An important aspect to global deployability of such a system is how well it can scale when the system is under heavy load. The scalability of the system is determined by how well it can handle an increasing load of any kind, for instance an increasing number of incoming and outgoing requests. This thesis considers multiple scenarios that puts the NEAT library under various load, and it analyses how this load affects the scalability of the library with regards to resource utilization.

1.1 Problem statement

The performance and scalability of NEAT-like systems is not well understood. A major concern is whether such systems can be deployed in the Internet at global scale. We argue that in order to address this concern, the following concerns need to be addressed:

1. The wide-spread deployment of NEAT-like systems may introduce more network traffic than the network can handle, which can lead to congestion collapse, lower throughput, or unfair sharing of network resources.

2. The resource usage on a local machine running a NEAT-like system may be too high to meet the requirements of the end-user.

These concerns cover both a global (1), Internet-wide concern, and a local (2), resource usage concern. We expect that if these concerns can be addressed, we can conclude that NEAT-like systems *can* be deployed in the Internet. There are also other factors to the deployability of NEAT-like systems. For instance factors such as deployment strategy/process, and the prospects of future deployment of such a system. However, we do not consider these additional topics in this thesis. Below we address both of the concerns listed above:

1.1.1 Global concern of using NEAT

NEAT implements a *Happy Eyeballs*[2] (HE) mechanism that initiates several connection requests simultaneously with different transport protocols. This is done to probe the end-to-end network path including the remote

[2]See Section 2.2.4 for a detailed description of the *Happy Eyeballs* mechanism in NEAT.

end-host for the support of transport protocols without introducing any significant delay compared to normal connection establishment using a single protocol. This mechanism of NEAT introduces additional traffic to the network, but the results of connection attempts can be cached so that subsequent connection attempts can skip unsupportive transport protocols [70]. When there are many available transport protocols to choose from, the HE mechanism can lead to a burst of connection requests that can congest the network. However, this can be mitigated by adding a short delay to every connection request so that they are spread out over time [99].

HE is already widely deployed in the Internet. For example, web browsers like Chrome and Firefox use HE to probe the end-to-end support for *Quick UDP Internet Connections* (QUIC) [37], and they fall back to TCP if end-to-end support is missing for QUIC [11]. HE is also widely used to facilitate IPv6 [RFC8200] adoption in the Internet [RFC6555]. The already ubiquitous deployment of HE in the Internet testifies that the extra network traffic introduced by NEAT is acceptable.

Another concern is whether the use of HE in NEAT will lead to more aggressive transmission of data [11]. This concern is addressed by noting that the aggressiveness of a sender is related to the congestion control algorithm and not by the transport protocol [11]. Also, modern congestion control algorithms like *CUBIC* [26] are less aggressive than previous congestion control algorithms, by decreasing the CWND *backoff factor* to facilitate low-latency data transfer [42]. These global trends testifies that the extra network traffic introduced by NEAT-like systems can traverse the Internet without disrupting the services that are already provided by the network, like fair bandwidth-sharing. We therefore choose to exclude the global concern in this thesis.

1.1.2 Local concern of using NEAT

All computer systems have a limited set of resources related to computing power and memory, and the number of available resources depends on the type of system, e.g. embedded, mobile, desktop, and high-load servers. This thesis evaluates the local resource usage of the NEAT library, and investigates on the local resource overhead of using NEAT compared to other state-of-the-art networking APIs (*RQ*, see Section 1.2). We argue that this investigation enables us to conclude whether the local resource usage concern mentioned above can be addressed. By comparing the resource usage of NEAT with the resource usage of the other APIs, we can quantify the resource overhead.

1.2 Research questions

In this thesis we elaborate on the following research question:

RQ. What is the local resource overhead of using NEAT compared to other state-of-the-art APIs?

1.3 Contributions

During the work on this master thesis, we have done the following:

1. Evaluated the performance and scalability of the NEAT library [64] compared to other state-of-the-art networking APIs under various network scenarios.
2. Contributed to the NEAT library [64] with bug fixes and extensions.
3. Made a *test suite* [63] for evaluating the performance of NEAT. The test suite also includes scripts for parsing results and producing graphs.

1.4 Research methodology

In order to evaluate the resource utilization of the NEAT library, we run our experiments in a physical testbed setup consisting of several machines. We run our experiments in a typical client-server fashion where we run the server and client applications on different machines, and connect the machines by a router on which we emulate various network conditions. In this way, we can evaluate NEAT on real hardware, but in a controlled environment to get concise results.

1.5 Thesis structure

The remainder of this thesis is organized as follows. Chapter 2 provides background on relevant concepts and related work. The BSD sockets API is introduced with a history of its evolution and features, and how it leads to the ossification of the Internet transport layer. Then, NEAT is introduced, describing how NEAT enables innovation and evolution of the Internet transport layer. Chapter 3 describes how the research question of this thesis is answered by collecting and analyzing the relevant data. Chapter 4 describes how to set up the experiments performed in this thesis and which experiment scenarios that are considered. Chapter 5 presents the results for the evaluation experiments of NEAT, comparing NEAT to other networking APIs. Then in Chapter 6 the results presented in Chapter 5 are discussed and compared. Finally, Chapter 7 wraps up the thesis, answering *RQ* and lists future work.

Chapter 2

Background

This chapter begins with an overview of the *BSD sockets API* [83], describing how it formed the Internet as we know it, and pointing out its limitations. Some limitations is that it is hard to implement modern network services that require the use of other transport protocols than TCP [RFC793] and UDP [RFC768], because the API exposes protocol-specific details and puts responsibility on the application developer to integrate other protocols and implement fallback mechanisms. The work in [71] describes how this has lead to the *ossification* of the Internet transport layer, i.e. that it has become hard to facilitate innovation of new transport protocols and deploying them in the Internet.

Following this the NEAT transport layer API and library [64] is presented. It is described how NEAT addresses the limitations of the *BSD sockets API*, and provides a platform- and protocol-independent API. Also the components of NEAT is presented, describing how they help in *de-ossifying* the Internet transport layer, and how they enable novel transport protocols and features to be easily accessible by applications.

Then, a list of other libraries and APIs that can potentially *de-ossify* the Internet transport layer is presented. We argue that NEAT is the most promising solution that can pave the way for establishing standards on how NEAT-like transport systems should be implemented in the Internet.

Finally, a summary of related work related to resource usage evaluation is given.

2.1 The BSD sockets API

The *Berkeley sockets API* (also known simply as the *sockets API* or the *BSD sockets API*) was developed by the Computer Systems Research Group at the University of California at Berkeley, and was first implemented in the 4.1cBSD operating system in 1982 [67]. Later, the API has evolved into a POSIX standard for developing network applications in UNIX systems [83]. All major operating systems implements the concept of *network sockets*, which are accessible through a BSD or POSIX like sockets API [6, 58]. These *sockets* are used to communicate with other machines over a network, and are often implemented as *socket descriptors* that are treated like regular files

in the operating system. In particular, reading and writing to a socket descriptor is handled the same way as reading and writing to a regular file. This simple, familiar, and high-level approach to data communication lead to the success and wide-spread adoption of the BSD sockets API.

The standard mandates that the application must specify the transport layer protocol to use for data transmission [33, 34]. To begin with, only two such transport protocols were available, namely the *Transmission Control Protocol* (TCP) [RFC793], which offers a stream-oriented, reliable and ordered delivery service, and the *User Datagram Protocol* (UDP) [RFC768], which offers a message-based, unreliable, unordered delivery service. Even though the Internet has evolved tremendously since the standard was proposed, the vast majority of network applications today still depend on either TCP or UDP [4].

Over the years, the use cases and context of the TCP and UDP protocols have changed from the original design philosophy of the Internet [14]. Originally, the few Internet home users in existence were typically connected to a Local Area Network (LAN) through a single network interface, and there were only a single network interface to access the Internet [67]. In the modern Internet, networks have much larger Bandwidth-Delay Products (BDP) due to faster networking equipment, and the Internet has grown to become a world-wide web of inter-connected devices. Several extensions to TCP have been added over the years to tackle these challenges [RFC7323]. TCP has historically been used for file transfer, web browsing and video streaming, while UDP has been used for service discovery and interactive media [44]. However, there are many applications that require more specialized services than what TCP and UDP can offer (see Table 2.1 for a list of such services [39] and examples of applications). For instance, modern end-hosts are often connected to several network interfaces. A laptop may be connected to the Internet via Ethernet, Wi-Fi and mobile network. To improve the availability and quality of a network connection, the applications on the laptop can be configured to use several network interfaces simultaneously which enables data to be sent over multiple paths in the network. This mechanism is called *multihoming* and is not available when using TCP and UDP.

Several new transport layer protocols have been developed since TCP and UDP were proposed, e.g. *Stream Control Transmission Protocol* (SCTP) [RFC4960], *Datagram Congestion Control Protocol* (DCCP) [RFC4340], and *The Lightweight User Datagram Protocol* (UDP-Lite) [RFC3828]. They offer services beyond what TCP and UDP can offer, for example SCTP can mitigate the *Head-of-Line Blocking* (HoLB) problem that is prevalent when using TCP [23, 79, 81]. This problem occurs when a packet is lost, in which case all subsequent packets are not delivered to the remote peer until the lost packet has been retransmitted and delivered successfully. The reason why HoLB occurs in TCP is because TCP offers both a reliable and ordered delivery service which means that all packets will need to be delivered in the exact sequence as they are sent. This can be a problem for many applications. For instance, if a web browser requests several web objects from the same web server over a single TCP connection, and the first web

Service	Application
`Partial reliability`	*Real-time* applications where the application data can expire and lose the usefulness due to later events (time passing, newer messages, etc) [54]. An example is a sensor that samples various data and sends this data over the network for processing. The sensor may only be interested in sending the newest sampled data and not outdated data.
`Partial error detection`	Applications that can handle partially corrupted data delivery from lossy links. For example, voice codecs like *Adaptive Multi-Rate* (AMR) [RFC3267] can cope better with errors in the payload than loss of entire packets [RFC3828].
`Multistreaming`	Applications that can partition the application data into independent parts. For example, when a web browser requests multiple web objects from a web server, each of these web objects can be sent on different streams [RFC3286]
`Multihoming`	Applications that must stay connected to remote peers even when a *proper subset* of the connecting network links go down [RFC3286]. An example is a video chat application that can fallback to use mobile network if a Wi-Fi network goes down.

Table 2.1: Network services that are not supported by TCP and UDP, and examples of applications that can benefit from these services.

object sent from the server is lost, none of the subsequent web objects will be delivered to the web browser application until the lost web object has been retransmitted and successfully delivered. If the *Round-Trip Time* (RTT) of the connection is large, this can lead to a significant delay to the delivery of web content because it can take some time for the lost web object to be retransmitted. When using SCTP, the web objects can be sent on different *SCTP streams*, and if data is lost within a certain stream, it will not affect the data sent and delivered on the other streams.

Although the transport protocols beyond TCP and UDP can offer many improvements over TCP and UDP, they contribute to a small portion of the total Internet traffic today [71]. There are primarily two reasons why they are not more frequently used as transport solutions [27]:

1. **Inflexibility of the BSD sockets API:** The BSD sockets API requires that the application developer specifies which transport protocol should be used for a specific network socket, and it is also the responsibility of the application developer to set protocol-specific options for each socket. This means that applications will need to be re-coded if new transport protocols or transport protocol features are to be leveraged, which might not be worth it from a business perspective. Also, it is not guaranteed that a new transport protocol

is supported end-to-end in a network path, that is, supported both by the end-hosts and by middleboxes on the path. There can be *Network Address Translation* (NAT) middleboxes in the path that does not support the protocol [28]. If an application attempts to use a transport protocol for communication (e.g. SCTP), and it is not supported end-to-end in the network, it is the application developer's responsibility to devise a fallback method (e.g. by using TCP or UDP instead), which adds more complexity to the application logic. The configuration of the protocol-specific options may also depend on the network environment the application will run in, which further increases the complexity of integrating new transport protocols into applications.

2. **Deployment vicious circle:** Middleboxes in a network may need to be reconfigured or upgraded in order to support new transport layer protocols and extensions. Since application developers cannot rely on new transport protocols to work over many network paths in the Internet, they often tend to use the safe option of using either TCP or UDP since these protocols have been supported in the Internet from the start. The middlebox vendors and maintainers hesitate to invest money in upgrading the networking equipment to support new protocols because they know that few applications use these new protocols. Also, other parts of the network may not support the protocols yet, meaning they may not be supported end-to-end in the network even though some elements in the network path support them.

Both the complexity of introducing new transport layer protocols into applications using the BSD sockets API, and the issues with deploying these protocols in the Internet, has lead to the *ossification*[1] of the Internet transport layer [71]. This has made it hard to realize innovation and evolution in the Internet transport layer, and is the reason why TCP and UDP are still so widely used today. This is a problem because TCP and UDP do not offer the services required by many applications. In addition, innovative transport protocols have been shown to improve application performance [59, 60]. As modern society is increasingly depending on technology and global communication over the Internet, it is important to deploy high performance transport systems that can meet modern requirements.

[1]The *Internet transport layer* is a broad and abstract term that incorporates both the term *transport layer*, that is about *end-to-end* communication over a network, and the broad term *Internet*, that describes the global network of inter-connected devices with different software and hardware technologies. *Internet transport layer* encompasses the concepts of both these terms, to describe the end-to-end communication methods and technologies that are widely deployed at global scale. We argue that these methods and technologies have converged to specific standards, and that it has become hard to change them. We use the term *ossification* to describe this phenonmenon.

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

Table 2.2: The core functions of the BSD sockets API

2.1.1 Overview of the API

The BSD sockets API enables the programmer to easily access network services through a uniform API that is designed to be independent from the underlying protocol stack. Even though the application developer will need to specify the transport protocol to associate with each network socket, the same API functions are used for most transport protocols.

Table 2.2 lists the core functions of the BSD sockets API that are required to create and release sockets, handle connection requests, and transfer data. In addition to these core functions, there are several other functions in the API that are used to tune various options and to handle Domain Name System (DNS) [RFC1035] requests. For example, `setsockopt` is used to set the majority of socket options, `fcntl` is primarily used to tune non-blocking sockets and asynchronous I/O (see Section 2.2.3), and `ioctl` is often used to access implementation-dependent options and attributes [71]. For a description and reference of BSD sockets API functions relevant for this thesis, see Appendix C.

2.1.2 Inflexibility of the API

This section highlights why the BSD sockets API is too inflexible to enable applications to easily leverage other transport protocols than TCP and UDP, which has lead to the *ossification* of the Internet transport layer as described above.

Exposure of protocol-specific details

When creating a new network socket with the BSD sockets API, the application must specify which transport protocol to associate with it. Listing 2.1 shows the protocol-specific details that the application must specify in order to create a socket. `AF_INET` specifies that the socket should be used in an IPv4 network, and `IPPROTO_TCP` specifies that the transport protocol TCP should be used for end-to-end communication.

Listing 2.1: Code example showing the protocol-specific details exposed in BSD sockets API function `socket`.

```
1 some_socket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
```

Modifiying the transport protocol specified in the BSD sockets API function `socket` is not necessarily enough to leverage other network services than those provided by TCP and UDP. Protocol-specific details are also exposed when setting socket options and sending/receiving data. Listing 2.2 shows how the TCP specific option `TCP_NODELAY` can be enabled, which disables the *Nagle* algorithm [RFC896].

Listing 2.2: Code example showing the protocol-specific details exposed when setting socket options with the BSD sockets API.

```
1 int flag = 1;
2 setsockopt(some_socket, IPPROTO_TCP, TCP_NODELAY,
3                         (char *)&flag, sizeof(int));
```

The BSD sockets API functions listed in Table 2.2 constitute a simple API where the user can connect ro remote peers and send/receive data. However, the API is not *expressive* enough to offer all kinds of specialized services that are provided by different protocols. The BSD sockets API functions `sendmsg` and `recvmsg` enable the application to respectively send and receive *ancillary* data that can contain protocol-specific data needed to provide more advanced network services. For example, SCTP uses the ancillary data to provide multistreaming and multihoming services. The set of ancillary data that can be sent/received for a specific protocol may be complex, which demands that the application developer have deep knowledge about the protocol and its implementation on different operating systems. Additionally, handling of the ancillary data introduces extra application logic complexity. Listing 2.3 gives an example of the protocol-specific details exposed with ancillary data.

Listing 2.3: Code example showing the protocol-specific details exposed when handling ancillary data with the BSD sockets API.

```
1  recvmsg(some_socket, &msghdr, 0);
2
3  /* Iterate the ancillary data (if present). */
4  for (cmsg = CMSG_FIRSTHDR(&msghdr); cmsg != NULL;
5       cmsg = CMSG_NXTHDR(&msghdr, cmsg)) {
6      if (cmsg->cmsg_len == 0) {
7          /* Handle error */
8      }
9
10     if (cmsg->cmsg_level == IPPROTO_SCTP) {
11         if (cmsg->cmsg_type == SCTP_RCVINFO) {
12             rcvinfo = (struct sctp_rcvinfo *)CMSG_DATA(cmsg);
13             /* Determine the stream the message were
14                received on. */
15             stream_id = rcvinfo->rcv_sid;
16         }
17         ... /* Handle other SCTP ancillary data. */
18     }
19     ... /* Handle ancillary data from other protocols. */
```

No fallback mechanism

If a protocol or protocol configuration is not supported end-to-end in the network, the application will need to fallback to another protocol or configuration until end-to-end communication can be established. However, this fallback functionality is not provided with the BSD sockets API.

API changes needed to integrate new functionality

The BSD socket API did not originally support multihoming, but it has later been extended to support it [RFC6458]. The API also needed to be extended in order to support IPv6 [RFC3493]. For example, to create a IPv6 socket instead of IPv4 socket, `AF_INET6` must be specified instead of `AF_INET` in the BSD sockets API function `socket`. In general, the API needs to be changed whenever new functionality is added. This is to preserve backwards compatibility with legacy software. The consequence of extending the API is that existing applications will need to be re-coded in order to leverage new functionality.

2.2 The NEAT Internet Transport Layer API and Library

NEAT (A New, Evolutive API and Transport-Layer Architecture for the Internet) is a new, open-source, user-space software library [64] which is implemented in accordance to the ongoing standardization efforts at the Transport Services (TAPS) Working Group [35] of the Internet Engineering Task Force (IETF). The NEAT library is developed by the NEAT Project [61]. The goal of NEAT is to re-enable the evolution of the Internet transport layer by offering a protocol- and platform-independent programming interface to the application layer.

Instead of requiring the application developer to specify transport protocols and options like with the BSD sockets API, the NEAT API is *protocol agnostic*, and requires that the application developer specifies *transport services* for each *NEAT flow*. NEAT flows can be viewed as communication endpoints, and are either mapped one-to-one to network sockets or to SCTP streams if SCTP is used[2], Each *transport service* [94] consists of a set of *transport features*, which are defined as end-to-end features that the transport layer provides to an application. These features includes security, reliable delivery, ordered delivery, message or stream orientation, etc. The combination of these transport features provides a complete service to an application. In the NEAT terminology[3], these

[2]Section 2.2.4 provides more details about NEAT flows and the architecture of NEAT.
[3]See Appendix A in [43] for a general overview of the NEAT terminology.

transport services are called *NEAT properties*, and the NEAT library will choose the best available transport protocols and options for an application based on the NEAT properties specified through the NEAT API. This way, the NEAT library offers a protocol-agnostic API for transport protocol selection.

This section provides an overview of the NEAT API and library, describing how the components of NEAT enables applications to access advanced and innovative network services that can lead to de-ossifying the Internet transport layer [71]. The rest of this section is organized as follows. Section 2.2.1 elaborates on the complexity of implementing transport protocols in operating systems, and that many protocols are instead implemented in user-space or encapsulated in UDP. It also describes how UDP encapsulation can improve the chances of NAT traversal. Since NEAT is a user-space library it can leverage both user-space and kernel-space protocols, libraries, and mechanisms. Section 2.2.2 describes different methods for determining which transport protocols are supported between the local endpoint and the remote peer. The Happy Eyeballs mechanism of NEAT is introduced. Section 2.2.3 describes different event-handling mechanisms, their benefits and limitations, and explains why NEAT uses the callback-based approach offered by libuv [50]. Section 2.2.4 presents an overview of the NEAT architecture summarizing all components and how they interact.

2.2.1 Leveraging user-space libraries and UDP encapsulation

It is important to note that NEAT builds upon the BSD sockets API, and that the same services can be provided both through using NEAT and the BSD sockets API directly. There are however many benefits to using NEAT instead of directly accessing the socket layer.

NEAT is a user-space library, and therefore has access to both user-space and kernel-space libraries and mechanisms on a particular platform. On the other hand, the BSD sockets API is part of the operating system and communicates directly with the socket layer of the kernel. Updates to the BSD sockets API will therefore follow the release cycles of the operating system which may be very long especially for stable releases. Consequently, new transport protocols and options may not be available in the operating system for a long time. Additionally, the new transport protocols will need to be integrated into different operating systems that have different implementation requirements and behaviours. Many transport protocols are therefore implemented in user-space so that they can be decoupled from the operating system details, and be provided more timely updates.

There are several transport protocols implemented in user-space including SCTP [1, 72], Google's QUIC (Quick UDP Internet Connections) [37], and WebRTC (Web Real-Time Communication)[4] [9]. However, one

[4]WebRTC is not a stand-alone transport protocol but is a library that leverages other protocols to achieve peer-to-peer communication.

problem with transport protocols implemented in user-space is that every application will run a separate network stack which can lead to increased memory usage, sub-optimal performance and errors. Another problem with user-space library implementations is that the application developer will need to interact with a variety of different APIs which can increase the application logic complexity.

The NEAT library can be updated to add the support for new user-space libraries when they become available. For all the user-space libraries that are available on the operating systems that NEAT supports, the NEAT library can leverage the different APIs and libraries internally if they match the requirements specified by the user. This enables the NEAT application to leverage the services provided by the user-space libraries without accessing the APIs directly.

A common method for implementing user-space transport protocols is to use UDP as a substrate protocol and implement new features on top of it [16]. The primary reason why UDP is used as the underlying transport is to improve the chances that the encapsulated packets belonging to that transport will be able to traverse NAT middleboxes [13, 28, 85, RFC6951]. The reason why UDP has a better chance to to traverse NAT middleboxes compared to newer protocols is that UDP has been used in the Internet from the start. UDP is also a very minimal protocol that only supports port numbers and a checksum, which makes it a good candidate for further extensions. Since UDP is so simple and minimalistic, all applications that use UDP will need to implement the same core set of functionality, e.g. congestion control to not cause congestion collapse in the network [RFC8085]. UDP encapsulation can help innovative transport protocols to be deployed in the Internet, but it is not a problem-free solution. In particular, the extra UDP layer poses some overhead on the systems that handles the encapsulated packets. Also, NAT gateways typically use shorter timeouts for UDP port mappings than e.g. TCP port mappings, so it is more desirable to use a native transport for long-lived connections [13].

2.2.2 Transport protocol selection using Happy Eyeballs

The BSD sockets API does not provide any mechanisms for determining which transport protocols or transport protocol extensions are supported both on a network path and at the remote endpoint(s). Also, if more than one protocol is found to be supported end-to-end, the BSD sockets API does not offer any negotiation mechanisms between the endpoints to use the best available protocol. If a protocol is used but it fails to traverse the network path e.g. due to an unsupportive NAT middlebox, the application developer must devise a fallback method and attempt the use of another protocol until end-to-end communication is established. Sometimes the packets are silently dropped by middleboxes, and the application may not be notified that a protocol is not supported. This all puts responsibility on the application developer to add support for new transport protocols.

There exists multiple approaches to discover the support for transport

protocols and extensions between two endpoints. [20] proposes a negotiation mechanism where remote endpoints can list the available transport protocols and negotiate on the best fitting one. [100] describes how the Uniform Resource Identifier (URI) format can be extended to include information about transport protocols. For instance "http" would suggest to use TCP as transport while "http-sctp" would suggest to use SCTP. Another approach is to use the Session Initiation Protocol (SIP) [RFC3261] and have the SIP clients and proxies select a transport protocol based on the transport protocols returned in the DNS SRV records [RFC2782, RFC3263]. However, all these approaches only determine whether a transport protocol is supported by both endpoints, but does not determine if the protocol is supported by NAT middleboxes, load balancers, firewalls, etc. in the network.

NEAT uses a mechanism called *Happy Eyeballs* for transport selection that discovers if the protocols are supported along the entire network path and by the endpoints [25]. The Happy Eyeballs mechanism was first introduced to facilitate IPv6 adoption in the Internet [RFC6555], but the same technology has been found to also facilitate transport selection [97, 98]. It works by simultaneously initiating different transport protocols when wanting to connect to the remote peer. If one of the protocols fails to traverse the entire network path due to e.g. an unsupportive NAT middlebox, the others may succeed and successfully establish connections. The idea is that one can probe the network for the support of a desired protocol and fallback to another if the former is not supported (e.g. fallback to TCP if SCTP is not supported) without introducing any significant connection establishment delay overhead. If it is desired that one protocol is used over another (e.g. SCTP over TCP), the initiation of the connection establishment for the less desired protocol can be delayed by a short amount of time to give the most desired protocol a head-start [99]. Although the simultaneous initiation of several protocols can produce some delay, and adds system and network load, this can be mitigated by caching connection data when opening many flows [97, RFC6555, 99].

The transport layer Happy Eyeballs mechanism used by NEAT combined with its protocol- and platform-independent user API is what enables the use of new transport protocols, paving the way for more innovation. The Happy Eyeballs mechanism gives innovative protocols and extensions that can improve application performance a chance by trying to establish connections with them first. This leads to a change to the traffic travelling on the wire and through middleboxes. The new traffic patterns can potentially lead middlebox vendors and maintainers to upgrade and reconfigure the equipment to support the traffic [44]. In addition, the protocol agnostic API enables new features to be added seamlessly.

An implementation of Happy Eyeballs has been made by Apple [82] to facilitate IPv6 adoption. Also, popular web browsers like Firefox and Chrome use Happy Eyeballs to discover the end-to-end support for the QUIC transport protocol and fall back to TCP if it is not supported [11]. The global deployment of Happy Eyeballs indicates that it is a suitable mechanism for transport selection. The same issues with IPv6

adoption are also present with transport protocol deployment (lack of end-to-end support). Although most of todays Internet supports IPv6, the connectivity for IPv6 is still worse than IPv4 [32], and there does not exist any mechanisms to determine the end-to-end support of an Internet protocol without testing it in the network. [2] show that the Happy Eyeballs implementation in Chrome[5] introduces less connection establishment delay compared to other implementations. [8] evaluates how the delay introduced between IPv6 and IPv4 candidates affects the connection establishment time. [70] evaluates the local resource cost of transport protocol selection with the Happy Eyeballs mechanism on the server-side, and shows that the increase in resource usage is proportionally not so large compared to initiating connections with a single protocol, especially when caching of connection results is enabled (see Section 2.4 for more information on this study).

2.2.3 Event-handling

NEAT is a complex library that needs to handle a variety of *events* like reading and writing to network sockets, polling network sockets for readability or writability, handle timeouts and signals, and facilitate communicate between NEAT components. There are different APIs that can handle such events with different benefits and limitations. Also, some of the event-handling APIs are only supported on specific operating systems.

This section describes different approaches to event-handling. First it describes APIs available in most operating systems due to POSIX standardization. Second it describes APIs that are available in specific operating systems. Third it describes different callback-based APIs that calls user-specified functions when events occur. Fourth it describes the difference between asynchronous I/O and non-blocking I/O. Finally, it elaborates on the event-handling mechanisms used in NEAT.

POSIX APIs

select [80] and *poll* [75] are event-handling mechanisms that are standardized in POSIX [33, 34] and available in most operating systems. They are used by applications to monitor file descriptors for events, waiting until an event occurs. Frequently, they are used to monitor socket descriptors to determine when data is writable or readable.

select is the oldest and most inefficient mechanism. An example of an event loop implemented with *select* is given in Listing 2.4. *select* performs poorly when monitoring a lot of socket descriptors. It was implemented in a time before the global-scale Internet of today, and was not designed to scale well with thousands of connections in multi-threaded environments. It has the following limitations:

[5] Chrome first attempts connecting to one address family (IPv6 or IPv4) and falls back to the other after 300 ms if no response is received.

Listing 2.4: Code example of an event loop using select

```
for (;;) {
    /* Need to re-copy socket descriptors for every iteration
       because select modifies the fd_set passed as argument */
    read_fd_set = active_read_fd_set;

    /* Block until event is available */
    if (select(FD_SETSIZE, &read_fd_set, NULL, NULL, NULL) < 0) {
        /* Handle error */
    }

    /* Iterate through all descriptors to see if an event is
       available for any of them */
    for (i = 0; i < FD_SETSIZE; ++i) {
        /* If an event is available */
        if (FD_ISSET(i, &read_fd_set)) {
            if (i == server_socket) {
                /* Accept new connection */
            } else {
                /* Read data from client */
            }
        }
    }
}
```

- Supports at maximum 1024 simultaneous connections. This is because the `fd_set` structures passed as arguments have a limitation on the number of bits that can be set. The maximum number of simultaneous connections allowed is determined by the `FD_SETSIZE` macro.
- Modifies the `fd_set` structures passed as arguments. This means that the user must re-copy the socket descriptors that should be monitored every iteration of the event loop.
- To determine which socket descriptor an event is available for, the user must iterate through all the monitored socket descriptors and call `FD_ISSET` to check if an event is available.
- Does not support multi-threaded environments where another thread modifies the `fd_set` given as argument while *select* is blocking. This will lead to unspecified behaviour.
- Can only determine if a socket descriptor is closed by the remote peer by trying to read from the socket. The read will return 0.
- Requires that the user must calculate the largest socket descriptor that is monitored. Alternatively, the macro `FD_SETSIZE` can be used (which is the upper bound).

poll is a more modern event-handling mechanism and mitigates many of the issues with *select*. An example of an event loop implemented with *poll* is given in Listing 2.5. *poll* mitigates the following issues of *select*:

Listing 2.5: Code example of an event loop using poll

```
for (;;) {
    /* Block until event is available */
    if (poll(pollfds, number_of_fds, NULL) < 0) {
        /* Handle error */
    }

    /* Iterate through all descriptors to see if an event is
       available for any of them */
    for (i = 0; i < number_of_fds; ++i) {
        /* If no event, try next */
        if (pollfds[i].revents == 0) {
            continue;
        }

        if (pollfds[i].fd == server_socket) {
            /* Accept new connection */
        } else {
            if (pollfds[i].revents == POLLIN) {
                /* Read data from client */
            }
        }
    }
}
```

- There is no limit on the number of simultaneous connections.
- Does not modify the `pollfd` structures passed as arguments. This means that the user does not need to re-copy the structures for every iteration of the event loop.
- Can determine if the remote peer closed the connection without having to read from the socket descriptor by setting the `POLLHUP` flag in the `revents` field of the `pollfd` structures.
- The user does not need to specify the maximum socket descriptor that will be monitored.

However, *poll* still has the same multi-threading issues as *select*. Also, the user must still iterate through all of the `pollfd` structures to determine which socket descriptor an event is available for. This might not be a problem except for the cases when the application needs to handle thousands of connections. The choice of event handling mechanism used at the client-side is usually not significant except for *peer-to-peer* applications with thousands of connections.

Platform-specific APIs

Every operating system typically include a high-performance event-handling API that depends on implementation details of the operating system which are not standardized by POSIX. Table 2.3 lists the high-performance event-handling APIs on all major operating systems.

Operating system	Event-handling API
Linux	*epoll* [17]
FreeBSD *NetBSD* *OpenBSD* *DragonflyBSD* *macOS*	*kqueue* [45]
Microsoft Windows	*I/O Completion Ports* (IOCP) [36]
Solaris	*Event Ports* [18]

Table 2.3: The high-performance event-handling APIs of different operating systems.

These high-performance APIs are designed to support server-side applications that can handle tens of thousands of connections in a multi-threaded fashion. Listing 2.6 gives an example of an event loop implemented with *kqueue*. Using these platform-specific APIs are not always better than using *poll*, especially when the connections are short-lived or when the events that are monitored for each socket descriptor are modified rapidly. This can be the case in the web, where a server may need to accept thousands of new short-lived connections every second. In this case, *epoll* requires that a system call is made for every socket descriptor when modifying the monitored events for that descriptor, which can introduce a high amount of resource usage. *kqueue* can modify the monitored events for multiple socket descriptors in a single system call which is more efficient than *epoll*.

Callback-based APIs

Several cross-platform event-handling libraries have been developed, like *libevent* [49], *libev* [48], and *libuv* [51]. These libraries offer callback-based APIs that enable the applications to set user-specified *callback functions*. These callback functions are set for specific kinds of events and called whenever the events occur, for instance whenever a network socket is writable or readable, or when a signal is received. These libraries are designed based on the concept of an event loop that loops continuously monitoring events, and calling callbacks whenever events occur. The event loop is an abstraction of the platform-specific event-handling API that is available in the currently running operating system. For example, if *libuv* is run in FreeBSD, it will use *kqueue* internally to handle events.

libevent was developed first, and it was developed to replace the event loop in event-driven network servers. However, due to several limitations and security vulnerabilities [96], the library *libev* was developed to replace *libevent*. *libev* is a stripped-down implementation of *libevent* that is more resource efficient [96]. However, a major drawback of *libev* is that it does not support event-handling with *IOCP* in Microsoft Windows.

Listing 2.6: Code example of an event loop using kqueue

```
for (;;) {
    /* Block until event is available */
    if ((nev = kevent(kq, NULL, 0, evlist, evlist_length, NULL)) == -1) {
        /* Handle error */
    }

    /* Iterate through the events */
    for (int i = 0; i < nev; i++) {
        if (evlist[i].flags & EV_ERROR) {
            /* Handle error */
        }

        if (evlist[i].ident == server_socket) {
            /* Accept new connection */
        } else {
            if (evlist[i].filter == EVFILT_READ) {
                /* Read data from client */
            }
        }
    }
}
```

libuv (Unicorn Velociraptor Library) [51] was originally developed on top of *libev* with extensions to support *IOCP* for Microsoft Windows. However, in later versions[6] it does not depend on *libev* and is a stand-alone state-of-the-art library for cross-platform event-handling supporting all major operating systems [3]. *libuv* was primarily developed for use by Node.js [68], but is also used by other projects [40, 55, 64]. Listing 2.7 gives an example of how callbacks can be set in a server application using libuv.

Asynchronous vs non-blocking I/O

It is important to understand the differences between *asynchronous I/O* and *non-blocking I7O*.

Non-blocking I/O If the user creates a socket it is put in *blocking* mode by default. This means that all the system calls that are performed on the socket will block (the process will sleep) until the system call can complete successfully. For instance `accept` will block until a new incoming connection can be processed, and `recv` will block until there are data to be read. Intuitively this seems sub-optimal since the application can do other useful work instead of waiting for I/O. This is why a socket can optionally be put into *non-blocking* mode. In this mode, all system calls will return immediately even if there are data to be processed or not. If the system call would block if the socket was set in *blocking* mode, the system call will instead return with `errno` set to either `EWOULDBLOCK` or `EAGAIN` if the socket

[6] In version `node-v0.9.0` of *libuv* the *libev* dependency was removed.

Listing 2.7: Code example of the callback-based API in libuv

```
1  uv_loop_t *loop = calloc(1, sizeof (uv_loop_t));
2  struct server_ctx *ctx = calloc(1, sizeof (*ctx));
3  ctx->handle = calloc(1, sizeof (*ctx->handle));
4  ctx->server_socket = server_socket;
5
6  /* Initialize the event loop */
7  if (uv_loop_init(loop) != 0) {
8      /* Handle error */
9  }
10
11 /* Initialize a handle to a event watcher for a server socket */
12 if (uv_poll_init(loop, ctx->handle, ctx->server_socket) < 0) {
13     /* Handle error */
14 }
15
16 /* Register the callback that will be called when receiving incoming
17    connections. In this case, call on_connected */
18 if (uv_poll_start(ctx->handle, UV_READABLE, on_connected) < 0) {
19     /* Handle error */
20 }
21
22 /* Set the pointer to the user data that can be associated with each
23    handle. This enables us to easily access data that are related to
24    a specific socket once a callback is issued */
25 ctx->handle->data = ctx;
26
27 /* Start the event loop */
28 uv_run(loop, UV_RUN_DEFAULT);
```

is put in *non-blocking* mode. The application can then attempt to issue the system call later to see if new data are available.

Asynchronous I/O When using an event-handling API the I/O events are handled in an *asynchronous* manner, and if no event-handling API is used the events are handled in a *synchronous* manner. If no event-handler is used, the only way to determine if events are available is to continuously loop the set of socket descriptors checking for events. On the other hand, when event-handling APIs like *select*, *poll*, and *kqueue* are leveraged, the application can be notified about events *asynchronously*. The event-handling APIs do not require the monitored socket descriptors to put in *non-blocking* mode, because the event loop is not issuing any I/O system calls on the sockets.

Non-blocking sockets are often used together with *asynchronous I/O* to improve performance [10]. For instance if a socket descriptor is marked readable by *select* and the socket is in *blocking* mode, the BSD sockets API function `recv` can potentially block if it is called more than once for every event loop iteration. If the application receive buffer is small, the event loop will need to iterate many times in order to read a lot of data. This will increase the CPU usage and delay of the application because *select* is called unnecessary many times. If the sockets are put in *non-blocking* mode, `recv`

can be called arbitrarily many times in every event loop iteration until all data has been received.

Asynchronous event-handling in NEAT using libuv

The NEAT library uses libuv internally for all event-handling. This enables NEAT to leverage the best event-handling APIs available in every operating system by accessing a uniform, cross-platform API[7]. NEAT offers a callback-based API, and the callbacks are administred internally by libuv. NEAT uses a callback-based approach so that the details of the NEAT event loop can be hidden from the application. When callbacks are set in NEAT, they are not called directly from libuv once the associated events occur. Instead, libuv calls internal callback functions in NEAT that handles internal logic, and from these internal callback functions the user-specified callback functions are called.

When libuv is used as the event-handling API, the sockets are automatically put in *non-blocking* mode, and the events are handled *asynchronously*. This enables NEAT to be notified by libuv when events occur, and enables NEAT to perform I/O operations efficiently.

A drawback of using libuv in NEAT is that libuv will always use the platform-specific event-handling APIs even though these APIs are not always the best options for handling events efficiently as described earlier. For instance, in Linux *epoll* is always used although using *poll* can be more efficient in some scenarios (e.g. for short-lived connections). NEAT uses libuv to be as portable as possible, and to minimize the number of bugs related to event-handling. The NEAT library is still prototype software, and in a later version, the internal event-handling in NEAT can be optimized based on the callbacks that are set and the NEAT properties.

2.2.4 The NEAT architecture

The *NEAT architecture* consists of several independent categories of components that are responsible for specific tasks. The *transport services* provided by these components are accessed through the *NEAT User API* (NEAT API). This is the interface the user interacts with to access the services of the *NEAT system*. All the components of NEAT including the NEAT API are part of the *NEAT User Module*. The *NEAT API* enables the developer to specify the transport services at runtime, and the *NEAT system* will dynamically handle these requests based on cached information, current network configurations and local policies. The components of NEAT constitute a *Happy Eyeballs* mechanism that is used for transport selection.

Figure 2.1 illustrates the context in which the *NEAT library* operates, and depicts the different component categories that constitute the *NEAT architecture*. There are five component categories: *Framework*, *Policy*, *Selection*, *Transport*, and *Signaling & Handover* [44]. A detailed description of the *NEAT architecture* and the different components can be found in

[7]Currently NEAT is supported on FreeBSD, Linux, OS X and NetBSD.

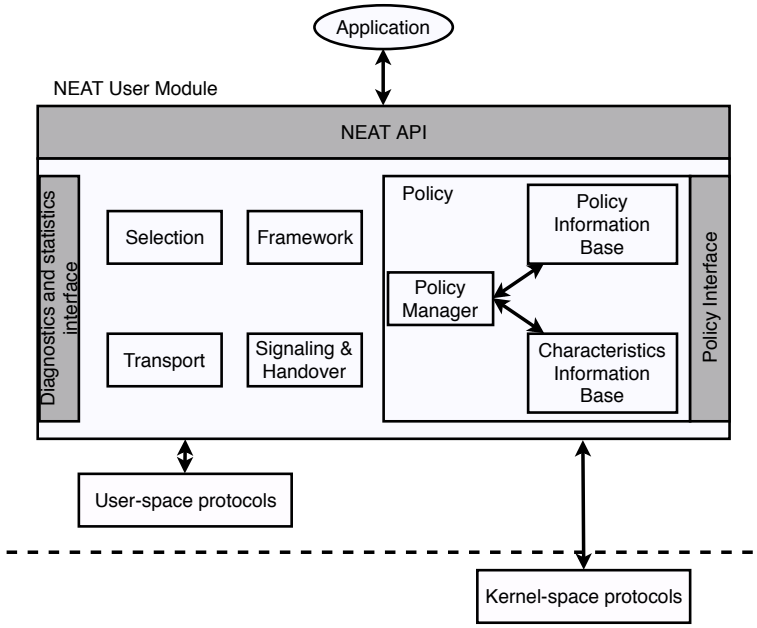


Figure 2.1: The architecture of NEAT. This figure is inspired by Fig. 1 in [44].

[43]. The following is an overview of the most important functionality and mechanisms. Note that this is the *intended* architecture of NEAT, and that the functionality of the actual reference implementation [64] may not fully conform to this specification. The reference implementation is a prototype software, although it implements the main ideas from the architectural specification.

As illustrated in Figure 2.1, the *NEAT User Module* is located in user-space, and has direct access to both kernel mechanisms and other user-space mechanisms. The workings of the *NEAT components* are hidden from the application, and the application has a single interface to deal with.

Framework The *Framework* components constitute the core functionality of the *NEAT library*, and are the minimum requirement to implement the *NEAT system*. These components are responsible for binding the other components together to form a coherent system. The *Framework* components are:

- The *NEAT API* that replaces the *BSD sockets API* to request network services in a platform- and protocol-independent fashion. The NEAT API is callback-based and provides asynchronous I/O operations.
- The concept of a *NEAT Flow Endpoint* or simply *NEAT flow*. The sockets instantiated by the *BSD sockets API* is replaced with this platform- and protocol-independent data structure. It represents an endpoint that maps to a single socket or to a single stream if multiplexed data delivery is used [RFC4960]. Like the Transmission Control Block (TCB) [RFC675] contains information for a TCP socket,

the *NEAT flow* aggregates the data that are needed to model an endpoint, including data like the associated socket descriptor, remote address, remote port, congestion control algorithm, *Happy Eyeballs* candidate list, etc.

- The ability to *connect to a name*. If the application specifies a specific IP address to connect to, this component is not needed, but if it specifies a remote domain name, the name will need to be resolved through a DNS lookup. The DNS lookup in NEAT extends upon the POSIX DNS lookup function `getaddrinfo` by enabling DNS lookup on all interfaces, and can lookup more than a specific transport protocol.

- The ability to retrieve statistics from the *NEAT system*. A user should be able to retrieve the current system state including information about which flows are open, the transport protocol used for each flow, the transport protocol options that are enabled/disabled, etc. Also, general flow statistics like the number of bytes sent/read, the number of messages sent/read, etc. should be available.

Policy The *Policy* components are responsible for generating a ranked list of transport solution candidates based on the *NEAT properties* specified through the *NEAT API*. These candidates are used when performing *Happy Eyeballs* for transport selection. The *Policy* components comprise three entities: the *Policy Manager* (PM), the *Characteristics Information Base* (CIB), and the *Policy Information Base* (PIB). These components interact with each other to map the specified application properties to specific transport protocols and options.

The *Policy Manager* is the core component of the *Policy* components, and it manages both the *CIB* and the *PIB*. The *NEAT system* communicates with the *PM* through the *Policy Interface*. In the reference implementation of the *NEAT library*, the *PM* is implemented as a separate Python daemon that runs independently from the NEAT application, and the *Policy Interface* of the *PM* is implemented as a *UNIX domain socket*. The NEAT application sends the *NEAT properties* through this interface, and the *PM* responds with a candidate list back to the NEAT application. The NEAT application specifies the *NEAT properties* in a *JavaScript Object Notation* (JSON) format [15] because JSON is a format that is suitable for storing key-value information, is easy to parse, and is human-readable. When the application opens a new *NEAT flow*, the *NEAT properties* specified for that flow are handled by the *PM* given that it is running[8].

In order for the *PM* to build the candidate list, it takes as input the *NEAT properties* specified through the *NEAT API*, and information from the *CIB* and *PIB*. The *CIB* and *PIB* are repositories that store various information and data. The *CIB* stores information about available interfaces, supported

[8]NEAT can open flows when the *Policy Manager* is not running. The NEAT library implements a simple *Policy Manager* function internally in the library that can translate simple NEAT properties to transport solutions. However, for more advanced NEAT properties and to leverage the *CIB* and the *PIB*, the *Policy Manager* must be running.

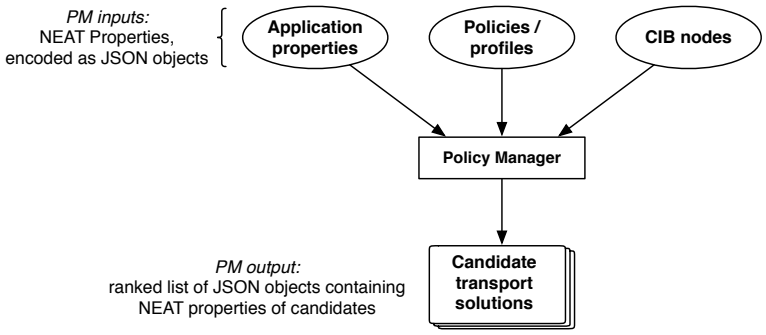


Figure 2.2: How data flow between the Policy components and the NEAT system. This figure is taken from [43].

transport protocols, network configuration and characteristics, current connections, cached data from previous connections, etc. In other words, it stores information about the characteristics of hosts, networks and systems, and this data is continuously being updated. The *PIB* stores *policies* and *profiles* that map the high-level properties requested by the application to actual transport protocols and options. This mapping constitute the *semantics* of the *transport services* specified via the *NEAT properties*. Unlike the data stored in the *CIB* that is continuously updated, the data in the *PIB* remain static during runtime. Figure 2.2 [43] illustrates how the *PM* retrieves information from several sources to produce the candidate list.

The difference between the *policies* and *profiles* stored in the *PIB* is that the *profiles* are applied before the *CIB lookup* while *policies* are applied afterwards. The motivation behind using *profiles* is that high-level *NEAT properties* can be mapped to specific host-specific properties related to the *CIB*. For instance, a *low delay NEAT property* can be mapped to a specific link medium (e.g. use Ethernet and not Wireless for low latency). When the *CIB lookup* occurs, the *NEAT properties* are updated to form a preliminary candidate list. Based on the *profiles* we might e.g. want to send data on a specific link medium, on a specific interface, on an interface with a specific MTU or bandwidth, etc. Finally, the *policies* update each of the preliminary candidates to form a complete transport solution candidate list that is returned back to the *NEAT framework*. Figure 2.3 [43] illustrates the workflow of the *PM*.

Selection The *Selection* components are responsible for selecting an appropriate *transport solution* that will work end-to-end and enable the application to communicate with the remote peer. A *transport solution* is the complete description of a transport candidate including transport protocol, its options, and other configurations that may or may not be supported end-to-end in a network. The *PM* returns a list of such *transport solutions* that must be handled by the *Selection* components. *Happy Eyeballs* is one of the components, that attempts to establish communication with the remote peer based on the *ranking* of the transport solution candidates. The *PM* ranks candidates with different priorities based on the user-

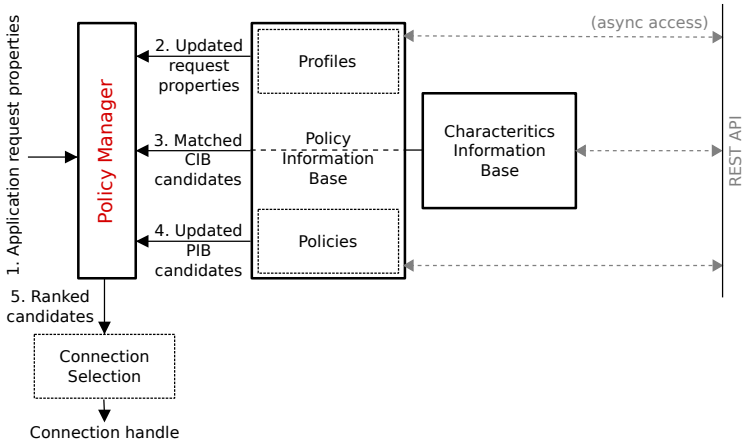


Figure 2.3: The order in which the Policy Manager receives data from its sources. This figure is taken from [43].

specifed properties and cached data. Candidates with lower priorities have higher precedence than candidates with higher priorities. The candidates are scheduled with a certain delay between each candidate to not cause bursty network behaviour and to let transport solutions a better chance at establishing communication before doing a fallback. Based on the priority value of a candidate and the delay introduced between two candidates, the total time a candidate needs to be delayed during *Happy Eyeballs* can be calculated by multiplying its priority value with the delay value. This means that the first candidate with a priority value of 0 will be attempted immediately while all other candidates are delayed appropriately. In the reference implementation, the delay introduced between candidates is set to a static value of 10 ms. The *Happy Eyeballs* candidates are scheduled in an *asynchronous* manner by using timers, offered in the underlying *libuv* event loop.

It is important to note that the *transport solutions* are not limited to transport layer details. Basically any network feature that may have the potential for being blocked in the network can be probed using *Happy Eyeballs*. An example of such features is the *Quality of Service* (QoS) marking of the packets. QoS marking enables the programmer to specify desired requirements for the sent network packets (e.g. that they should have higher priority compared to other packets, or that they should have low latency). In the BSD sockets API, the packets can be marked with QoS codes through the `setsockopt` system call. This modifies the *Differentiated Services Code Point* (DSCP) part of the *Type of Service* (ToS) field in case of IPv4, and *Traffic Class* field in the case of IPv6. The NEAT API enables the application to specify high-level abstract QoS types that can be mapped to several specific QoS codes internally by NEAT.

There are different network features that can be included in transport solutions, but where the transport protocol does not have the capability to signal the application about a successful or failed transport selection process. Examples of this is connectionless transport protocols like UDP

and UDP-Lite or QoS marking. In these cases, the application has full responsibility to define the semantics of the network service, and it needs to handle varying network conditions. In order to integrate these features with the transport selection process, another *Selection* component called *Happy Apps* is included. This component is only activated when the transport protocol cannot handle the selection signaling as described above. With *Happy Apps* enabled, an application-defined callback will be issued after some time, where the application can specify how 'happy' it is (e.g. the application can specify that it is happy beause it received response from the remote peer). If the application signals that it is 'unhappy' with the current condition, the *NEAT system* can fallback to another transport solution. This mechanism can be used to probe different QoS codes in the network.

Use of the *Happy Eyeballs* mechanism allows for protocols such as SCTP to be tried on the wire on the client-side, hence giving more incentive for the wider adoption of it in the Internet. However, SCTP uses a four-way handshake (4WHS) for connection establishment while TCP uses a three-way handshake (3WHS). This means that it takes 1 Round-Trip Time (RTT) before a TCP client can send data, while for SCTP it takes 2 RTTs. This means that if SCTP and TCP is initiated concurrently using *Happy Eyeballs*, TCP will probably win. Even if SCTP is given a head-start, and a delay is introduced before attempting to connect with TCP, TCP will in many cases still win over SCTP. This is a major drawback in the *Happy Eyeballs* mechanism. See Chapter 7 for a discussion on the possible solutions to this problem. More generally, *Happy Eyeballs* only works properly if the different *transport solutions* uses the same number of RTTs to connect.

For every *Happy Eyeballs* candidate that are probed in the network, the result is stored in the *CIB* for later reference. The result may include that the connection was successful or failed. This caching of connection establishment results mitigates the problem of introducing extra network traffic because if subsequent connection establishment requests are made to the same remote peer, NEAT can lookup the *CIB* and determine which *transport solutions* are supported end-to-end. The state of the network may of course change over time, and therefore, the cached data in the *CIB* will expire after some time. In order to get a clear picture about the transport protocols that are supported in the network, *transport solution* candidates may be given a chance to successfully connect before they are aborted.

Transport The *Transport* components are responsible for *configuring* and *managing* the transport protocols that are selected with the *Selection* components. This includes handling the transport protocols and features in such a way that a single, uniform API can be exposed to the user. For example, even though UDP is a connectionless protocol, the *Transport* components implement a *virtual accept* mechanism where the application can *accept* UDP 'connections' by calling `neat_accept` (see Appendix C for a description of NEAT API functions). Since the application in many cases do not specify the specific transport protocol to use, the application cannot

set protocol specific options, like the *Nagle* algorithm for TCP [RFC896]. Instead, these options are set automatically by NEAT based on the given properties.

The *NEAT Library* currently supports the following transport protocols: TCP, UDP, kernel-space SCTP, user-space SCTP, Multipath TCP (MPTCP) [RFC6824], UDP-Lite, SCTP over UDP in kernel-space, SCTP over UDP in user-space, and WebRTC (not a transport protocol, but an aggregation of protocols that can offer a flexible transport service). The *Transport* components need to *configure* these transport protocols to operate as efficiently as possible based on the user-specified properties. If the use of a specific transport protocol is forced through the properties, and services that are not supported by this protocol are also specified, these services are simply ignored by the *PM*. An example of a service that is not supported by all the protocols is *multihoming*, that enables a peer to communicate with the remote peer over several network interfaces. When this service is specified, the *Transport* components are responsible for binding to all of the specified addresses, for instance with `sctp_bindx`.

NEAT supports peer-to-peer communication by using the *data channels* of the *WebRTC* library. By leveraging protocols like *Session Traversal Utilities for NAT* (STUN) [RFC5389], and *Traversal Using Relays around NAT* (TURN) [RFC5766], WebRTC can improve the chances to traverse middleboxes.

A feature provided by some *Transport* components is the ability to *prioritize* data sent on different *NEAT flows* in a *flow group*. This means that the bandwidth capacity can be shared among the *NEAT flows* based on their priority values. If only TCP flows are opened, the flows can be coupled with the *Coupled Congestion Control* (CCC) mechanism described in [95]. This mechanism is currently only implemented in FreeBSD [21]. In this case, the Congestion Window (CWND) of the flows are limited by the priority values of the flows. If only SCTP flows are opened, either a new SCTP associatioon is established for every NEAT flow, or the NEAT flows can be transparently mapped to different SCTP streams in a single SCTP association using the *Transparent Flow Mapping* mechanism in NEAT [24, 92]. If Transparent Flow Mapping is used, the data sent on the different SCTP streams can be scheduled as defined in [RFC8260][9]. If a combination of TCP and SCTP flows are opened, the priority values are simply ignored because there does not exist any mechanisms for coupling TCP and SCTP connections.

Another feature that can be specified through the *NEAT properties* is *security*. Based on the transport protocol in use, this enables *encryption* and *decryption* of user data to facilitate *confidentiality*, *integrity*, and *availability* concerns related to data transfers, and enables the end-hosts to *authenticate*. NEAT uses the *Transport Layer Security* (TLS) [RFC5246] and *Datagram Transport Layer Security* (DTLS) [RFC6347] protocols implemented in the *OpenSSL* library [69] to provide this service. *TLS* is used with TCP, and *DTLS* is used with UDP and SCTP.

[9]The SCTP scheduling algorithms defined in [RFC8260] are not implemented in any major operating system yet.

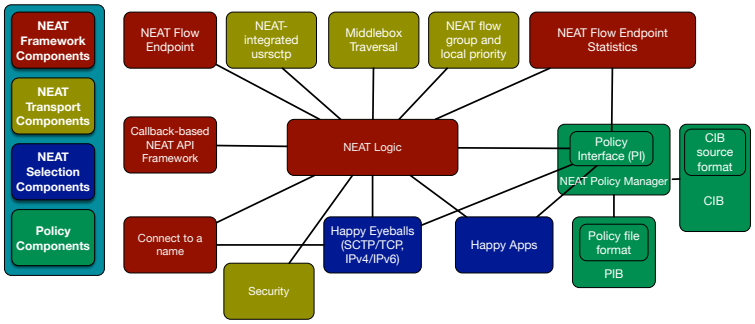


Figure 2.4: The components of NEAT and their interactions. This figure is taken from [43].

Signaling & Handover *Signaling & Handover* components are optional extensions to the *NEAT system* to complement the functions of the *Transport* components. They can send signals to remote devices and peers independent of user data, for example it can be used in *Software-Defined Networking* (SDN) [46].

Figure 2.4 [43] gives an overview of all the components in the different component categories, and illustrates how they interact.

2.2.5 Overview of the API

This section gives a brief overview of the NEAT API for the functions and callbacks that are considered in this thesis. For a description of how programming can be done in NEAT and a more in-depth description of the core functions of the API, see Appendix C.2. For a detailed reference to the NEAT API, see Appendix B in [43].

Table 2.4 lists the core functions of the NEAT API and gives a short description of their functionality. Table 2.5 lists the core callbacks of the NEAT API and describes in which scenarios they are called.

2.3 Existing approaches to de-ossify the Internet transport layer and why NEAT is needed

The *TAPS* working group is working on defining a *TAPS Transport System* for leveraging innovative transport protocols and features through a platform- and protocol-independent API where the user can specify *Transport Services*. The working group is primarily working on three documents to define this system [5]:

- Defining the *architecture* of the system.
- Defining the *abstract API*.
- Defining guidelines on how the system can be *implemented*.

Function	Description
`neat_init_ctx`	Creates a new *NEAT context* (one per event loop).
`neat_new_flow`	Creates a new *NEAT Flow Endpoint*.
`neat_set_property`	Sets the user-specified *JSON properties* for a specified *NEAT Flow*.
`neat_set_operations`	Sets the user-specified *callbacks* for a specified *NEAT Flow*.
`neat_start_event_loop`	Starts the *NEAT event loop* running *libuv* internally.
`neat_get_event_loop`	Returns the *libuv* handle used internally in the *NEAT event loop*.
`neat_open`	Connects to the specified remote peer using the specified *NEAT Flow*.
`neat_accept`	Accepts connection requests using the specified *NEAT Flow*.
`neat_read`	Reads data from the specified *NEAT Flow*.
`neat_write`	Writes data to the specified *NEAT Flow*.
`neat_close`	Closes the specified *NEAT Flow*.
`neat_stop_event_loop`	Stops the *NEAT event loop*.
`neat_free_ctx`	Releases all the resources associated with the specified *NEAT context*.

Table 2.4: The core functions of the NEAT API

Callback	Description
`on_connected`	The *NEAT Flow* successfully connects or a remote peer has connected.
`on_error`	An error has occured.
`on_readable`	The *NEAT Flow* is readable.
`on_writable`	The *NEAT Flow* is writable.
`on_all_written`	All the data that is buffered in the *NEAT Flow* is successfully sent.
`on_aborted`	The *NEAT Flow* is aborted.
`on_close`	The remote peer closes the connection or the application closes the connection associated with the *NEAT Flow*.

Table 2.5: The core set of callback functions that can be set through the NEAT API

The *TAPS* group has done work on three concurrent projects: *NEAT* [61], *Post Sockets* [88], and *Socket Intents* [87]. The participants of these projects collaborate to define the coherent concept of the *TAPS Transport System*. *NEAT* and *Socket Intents* provide actual implementation code [7, 64], while *Post Sockets* only provides an abstract API specification [88]. However, one of the authors of *Post Sockets*, Brian Trammell, has later collaborated with *Apple* which has implemented code. However, at the moment, this code is not open-source.

Out of the three *TAPS* projects, *NEAT* has the most explicit implementation approach, supporting a variety of different protocols and features. *Socket Intents* is mainly developed to facilitate *path selection*, by leveraging a similar mechanism to the NEAT *Policy Manager*. *Post Sockets* is quite similar to *NEAT*, but it requires that all data are sent as messages.

The work done by the *TAPS* working group is probably the most promising research for building a specification of a *Transport Service System* because it is part of the *IETF* that proposes Internet standards that should be followed by the wide Internet community. However, other related work has been conducted. [29] discusses the inflexibility of the *BSD sockets API*, and proposes *Dynamic Application Oriented Network Services* (DANCE) which is a *model* where the user can specify *transport services* through a new API [77]. Work done in another master thesis proposes a protocol-independent API [39, 93], and a paper related to a Ph.D thesis does the same [56]. [44, 71] provides a survey of related work.

2.4 Related studies concerning resource usage

The impact of *Happy Eyeballs* between SCTP and TCP on the server-side has already been evaluated in [70]. The paper considers CPU utilization, and memory usage attributed by network data structures in the kernel. RFC6556 [RFC6556] describes how the performance of a HE algorithm can be tested in the case of a dual-stack host that supports both IPv4 and IPv6 address families, and wants to use HE to quickly determine which addresses are available end-to-end in the network. However, RFC6556 only considers the *timeliness* of the algorithm, meaning that it only tests whether a specific HE implementation can establish connections within some time frame. On the other hand, [70] considers HE for transport protocol selection, and describes how the HE mechanism impacts the resource usage on a local machine with regards to CPU and memory, and describes how these metrics can be measured. The HE mechanism that is implemented and evaluated in the paper is not part of the NEAT library; therefore, the paper evaluates only a part of what needs to be evaluated in order to evaluate the performance of the entire NEAT library.

The paper examines how much resource overhead is induced by receiving extra connection requests from the client as a result of HE. The paper shows that HE increases the *CPU load* as compared with a single TCP or SCTP connection establishment[10], and that HE has a negligible memory

[10]The increase in CPU load on the server was reported to be in the order of 10% on

usage overhead compared to single TCP/SCTP flows [70]. The paper also shows that when caching of connection-request results was enabled in the experiments, the CPU load was substantially reduced because there was no need to attempt the initation of all protocols for every connection attempt [70].

Work has also been done to measure the CPU and memory impact of the NEAT library in a mobile broadband network scenario [11]. The study reports that when downloading files from a remote server, the CPU utilization increases with on average 4.27% (from 13.8% to 18.07%) when using NEAT compared to not using NEAT, and that the memory usage increases with on average 1.1 MB (from 1.9 MB to 3.0 MB). The CPU impact of connection establishment was included in these results[11].

average when responding with web objects of 35 KiB [70].

[11] Some of this information is not available in [11]. Instead we contacted the authors so that they could clarify the methodology further.

Chapter 3

Research Methodology

This chapter elaborates on the data collection and analysis methods used to answer the research questions in Section 1.2. Chapter 4 gives a detailed description of how these methods are applied in the experiments.

3.1 Local resource usage: NEAT vs other APIs

This section elaborates on the methods used to address *RQ*. It first provides an overview of how related work is used as a foundation to our own research, and it describes how our work complements the results, methods and limitations of the related work. It elaborates on the motivation behind comparing NEAT to other state-of-the-art networking APIs to get comparable results. Then it explains the performance metrics considered in this thesis, and elaborates on why we have considered these metrics. Finally, a technical description of how we sample the relevant metric data is given, and which techinques we have used to analyze the sampled data.

We base the research methodology in this thesis on real-life experiments in a controlled physical network setup, where we run our experiments based on multiple scenarios in order to sample the relevant data.

3.1.1 How our work builds on existing research

Section 2.4 lists research that has been conducted to evaluate the resource usage of the Happy Eyeballs mechanism and NEAT. However, this research does not cover all aspects of the performance evaluation of NEAT.

The evaluation of HE [70] was realized by implementing a HE mechanism in *httperf* [31], and extending the HTTP server *lighttpd* [52] to listen for both TCP and SCTP. That is, the paper does not consider the performance of the *NEAT library*; it only considers the impact of HE. Also, only the server-side performance is considered, which ignores the resource utilization of the host performing HE on the client-side. Finally, the paper only considers the memory usage of kernel data structures like *mbufs* used to store packets in the kernel. The paper does not consider the memory usage of the server application, e.g. stack usage, heap usage, or virtual memory page usage.

The mobile broadband network evaluation of NEAT [11] is more closely related to the work done in this thesis compared to the evaluation in [70] since it measures the impact of the entire NEAT library and not just a part of it. However, the study considers only a single scenario where a NEAT client downloads data from a remote server. It does not consider other scenarios like different numbers of incoming/outgoing requests. Also, it does not differentiate between the evaluation of connection establishment and data transfer, but instead includes the measured resource usage for both these cases into a single average value. Finally, it does not show the distribution of the sampled data.

3.1.2 Comparing NEAT to other APIs to quantify the resource overhead of NEAT

In this subsection we elaborate on how we can quanitfy the *direct resource overhead* of using the NEAT library compared to using other networking APIs. By *direct resource overhead* we mean the difference in resource usage between NEAT and other APIs. In other words, how much the resource usage increases by using NEAT compared to another API. To be able to measure this overhead we consider identical applications implemented with each of the APIs, that is, the only thing that differs between these applications is how they access the network services through the networking APIs; the semantics of the applications is the same.

We decided to compare the resource utilization of a NEAT application with an identical application programmed with libuv, since NEAT uses libuv internally to handle asynchronous I/O. We also decided to measure the resource utilization of an identical kqueue application, since libuv uses kqueue internally to handle asynchronous I/O[1]. For an explanation about why we decided to use kqueue, see Section 3.1.3.

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

3.1.3 Choice of operating system

Our experiments could potentially run on either FreeBSD or Linux since both are open-source well-documented OSes with active user communities and are supported by NEAT. We decided to use FreeBSD instead of Linux because FreeBSD's SCTP implementation is more compatible with the IETF standards in comparison with Linux [89].

[1] At least when running FreeBSD 4.1 and above.

Metric	Description
CPU time	The time spent by the CPU to execute instructions. This can be measured at a per-process level where the time can be divided into the following categories: • **User time**: The time spent by the CPU executing instructions for the process in user-space. • **System time**: The time spent by the CPU executing instructions in the kernel on behalf of the process.
CPU instruction count	The number of CPU instructions executed. This can be measured at a per-process level, giving the number of CPU instructions executed by every function that is called by the process.
Resident Set Size (RSS)	The number of bytes that are currently resident in the *Random-access memory* (RAM) for a specific process.
Heap memory usage	The number of bytes that has been dynamically allocated. This can be measured at a per-process level.
Real-time delay	The wall-clock time spent during a computation. This includes CPU idle time that would not otherwise be accounted for in the *CPU time*.

Table 3.1: The performance metrics considered in this thesis.

We consider kqueue for our experiments rather than epoll because kqueue is generally deemed as a more efficient state-of-the-art API [78] which can provide a better baseline for our evaluations. Since kqueue is not available on Linux, we decided to use FreeBSD.

3.1.4 Performance metrics

Due to the numeric nature of resource usage (and hence resource overhead), quantitative research approach for data collection and analysis is applied. In this thesis we consider the performance metrics listed in Table 3.1. We build our research based on the methods used in [70], and complement these methods with additional considerations to target the limitations discussed in section 2.4. In particular, we aim to measure the impact of the NEAT library; not only the impact of the HE mechanism. Also, we aim to find the per-process resource usage instead of the global system resource usage so that we can pinpoint the bottlenecks and overheads of using NEAT.

We choose to collect CPU and physical memory data because the CPU and physical memory are key resources on every computing system, and they are limited resources that are shared among all processes on the system. In order for NEAT to be deployable in end-hosts, it must share these resources fairly with other processes, but also use the resources in an

efficient way so that the end-user will experience the library as responsive and fast. We also choose to collect real-time application delay to quantify the responsiveness of the library.

3.1.5 Data sampling method

This subsection provides technical details about how the different performance metrics listed in Table 3.1 are sampled, and it elaborates on our considerations and thoughts when choosing these methods.

[53, 73] provide information about tools and methodologies to sample performance data in FreeBSD and Linux. These sources describe different observation tools that can monitor the global system and per-process resource usage associated with different parts of the operating system and different hardware. The information in these sources are used as a base for the methodology in this thesis.

CPU time

Operating systems like Linux and FreeBSD offer a large set of performnace observability tools that can provide information about global and per-process CPU usage. Some examples of such tools are *top*, *ps*, *mpstat*, *sar*, and *iostat*. These tools report either the percentage of time, or the number of *CPU ticks* ('jiffies'), that the CPU has executed for the different *types of tasks*. The types of tasks are not the same for all operating systems, but they can usually be divided into the following categories:

- **user**: The CPU is running in user-space executing application code.
- **system**: The CPU is running in the kernel, for instance executing system calls, and running device drivers and kernel modules.
- **interrupt**: The CPU is servicing interrupt requests.
- **idle**: The CPU is idle and is not doing any useful work.

Based on the operating system and configuration, the timer hardware can be programmed to interrupt the kernel a specific number of times every second [30]. The time between two such interrupts is defined as a *CPU tick*. Note that CPU ticks are not the same as *CPU clock cycles*: modern CPUs can execute billions of instructions every second, but many operating systems require that the kernel operates at a smaller clock rate. The reason is to for instance be able to schedule processes and perform context switching at a fixed rate, but not so quick that the machine is using too much power, or that the operating system will need to spend too much CPU time for housekeeping instead of executing more useful application code [30].

The tools mentioned above provide millisecond-precision CPU time measurements and therefore are not suitable for precisely measuring the impact of short-lived operations like opening a NEAT flow. Also since these tools run independently of the application of interest, it makes it hard to accurately synchronize the sampling of relevant data. This is mitigated

Listing 3.1: Code example showing how to sample CPU time data for the calling process.

```
1 #include <time.h>
2
3 struct timespec ts;
4
5 clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &ts);
```

by using the C function `system` to execute shell-commands from within the application. Therefore, the current CPU usage of an application at specific points (in the code) could be sampled by calling `system`.

procstat tool of FreeBSD can report CPU time of a specific process at microsecond granularity. This is the best precision we have been able to achieve for any of the tools that we have examined. The CPU time data reported by *procstat* is also not restricted by the *CPU tick* unit, which makes the data more *accurate* in addition to more *precise*. *procstat* is used in [70] to measure per-process CPU utilization.

The C function `clock_gettime` reports the per-process CPU time with similar precision as *procstat* (microsecond-precision). This function has the benefit that we do not need to call `system` to run a command-line tool in the shell. An example of how to sample the current CPU time of the calling process can be found in Listing 3.1. While *procstat* reports both the accumulated *user* and *system* times of the process (that is, how much time the process has executed in user-space and kernel-space), `clock_gettime` reports the sum of these values[2].

CPU instruction count

The CPU instruction count metric is included as an additional CPU usage metric because it enables performing more fine-grained analysis on the collected CPU time data. While CPU time data can quantify the CPU overhead of NEAT operations, it is also desirable to *profile* the NEAT code during runtime to determine which NEAT functions account for the most overhead. The CPU instruction count metric is a different metric than CPU time, but it gives a good indication of the CPU-intensive bottlenecks in the code. By profiling the code, we can determine how many times each function is called, and how many CPU instructions are executed for every function[3].

The *Valgrind tool suite* [90] includes different tools for profiling applications. One of these tools is *Callgrind* [12] that can count the number of CPU instructions executed by every function of a given application. It is also capable of collecting this data for loaded shared libraries, which enables it

[2] `clock_gettime` can be given different *clock IDs* to get the current time values for different clocks in the operating system. The clock ID `CLOCK_PROCESS_CPUTIME_ID` reports the sum of the calling process's user and system time.

[3] The number of CPU instructions executed by a function also includes the CPU instructions executed by functions called from that function.

Listing 3.2: Example of running Callgrind to profile the CPU usage of an application on the command-line. The $ symbol is the command prompt.

```
1 $ LD_BIND_NOW=y valgrind --tool=callgrind ./neat_application
```

to profile the CPU instructions executed by NEAT library functions when NEAT is loaded as a shared library into an application. *Callgrind* requires that the application is compiled with debugging information included[4].

Normally when an application is executed in FreeBSD, the dynamic linker will load the required shared libraries on-demand when they are needed in the code. However, this loading process introduces some additional CPU usage. Our goal with profiling the applications is not to calculate precise CPU usage (this is instead done by sampling the CPU time). Instead, we want to understand which application functions have the biggest impact on CPU usage. In order to remove the CPU usage of the dynamic linking process from the profiling results, the environment variable `LD_BIND_NOW` is enabled before *Callgrind* starts (see Listing 3.2). This causes all shared libraries to be loaded *before* the application starts.

Note that *Callgrind* significantly slows down the execution of an application because different counters are increased during runtime. This is another reason why *Callgrind* is not ideal for quantifying the actual CPU usage of an application.

An example of how we run *Callgrind* on the command-line is given in Listing 3.2. When the application terminates, the *Callgrind* tool will generate a log file containing the sampled data.

Resident Set Size

The memory used by an application at runtime (the virtual address space) can be categorized as follows:

- **Text segment**: Contains executable instructions to run the application (the compiled code).
- **Data segment**: Contains the global variables.
- **Stack**: Contains stack frames. A new stack frame is created for every function call, and removed when a function returns. The stack frames contain local variables of the function, function arguments, return address, etc.
- **Heap**: Contains the memory that is dynamically allocated during runtime.

Modern operating systems generally manage the available memory resources (physical memory and secondary storage) as *virtual memory* to

[4]On FreeBSD we compiled the applications with the `clang` compiler. We had to specify the `-g` compiler flag to compile with debugging information.

simplify memory management for applications. This memory is divided into *pages* of a fixed size. Therefore the memory consumption of an application is measured in the number of pages it uses.

Some tools that measure the memory usage of a process are *ps*, *pmap*, and *smem*. These tools report the total number of pages that are allocated for the process. However, this number can be interpreted in the following ways:

- **Resident Set Size (RSS)**: The memory that is resident in physical memory (RAM) including memory used by shared libraries. *RSS* can over-estimate the memory usage of a process because it includes the total memory usage of shared libraries although this memory can be shared among multiple processes.
- **Virtual Memory Size (VSZ)**: The memory that are accessible by a process in its address space. This includes the memory reported by *RSS*, but also includes memory that are swapped to disk, that are allocated but uninitialized, etc.
- **Unique Set Size (USS)**: The memory in RAM that is unique for the process. This excludes any memory that can be shared with other processes, like shared libraries.
- **Proportional Set Size (PSS)**: The same as *USS*, but includes the proportion of memory used by shared libraries. For example, if five processes depend on the same shared library that consumes 50 pages, the shared memory usage reported with *PSS* will be 10 pages.

In this thesis we consider the *RSS* of the application because it describes the total memory impact of the application. *VSZ* is not suitable for reporting the real memory usage because some of the memory it reports might not even be present in physical memory. *USS* might not include memory usage of shared libraries if those libraries are used by more than one process. In this case, the total memory usage of the application will not be correctly reported. *PSS* gives a good estimation of total system memory usage by processes, but it does not report the total memory usage of a single application.

We sample the *RSS* of the application by using the *ps* command-line tool. In order to synchronize the sampling of memory data at specific points in the application code, we run *ps* from inside the application by using the C function `system`. An example of how we sample *RSS* is given in Listing 3.3.

Heap memory usage

The *Resident Set Size* of the process gives an accurate value for the total physical memory usage. However, it does not imply what parts of the code is causing the memory usage. This can be determined by *profiling* the application at runtime, similar to how the *CPU instruction count* metric is sampled.

Listing 3.3: Code example of sampling RSS from within the application.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

char command_string[200];

/* Get the PID of this process. */
long int my_pid = (long int)getpid();

/* Build the string that will be executed. */
snprintf(command_string, sizeof (command_string),
         "ps -o rss -p %ld >> rss.log &", my_pid);

/* Execute it. */
system(command_string);
```

Listing 3.4: Example of running Massif to profile the heap memory usage of an application on the command-line. The $ symbol is the command prompt.

```
$ LD_BIND_NOW=y valgrind --tool=massif ./neat_application
```

The *Valgrind tool suite* [90] includes a tool called *Massif* [57] that logs the current heap usage, stack usage, or total virtual memory page usage throughout the execution of the application. *Massif* keeps track of the amount of dynamic memory that is allocated at every point in the code that allocates memory. This enables summarizing the memory usage statistics in such a way that the total memory usage allocated by every application function can be determined. *Massif* can also collect the memory usage for shared libraries that are loaded into the application.

Like *Callgrind*, *Massif* requires that the application is compiled with debugging information. We also decide to enable the `LD_BIND_NOW` environment variable when using *Massif*, to run the application the same way as when using *Callgrind*.

We consider the heap usage statistics collected by *Massif*. Heap memory usage is a subset of the *Resident Set Size* memory, and is therefore not a precise representation of the total memory usage. However, the heap memory usage constitutes a large portion of the total memory usage of the application, and gives a good indication to find the most memory-consuming parts of the code. An example of how we run *Massif* on the command-line can be found in Listing 3.4.

Real-time delay

Operating systems maintain different system-wide and per-process clocks. The system-wide clocks include clocks that represent the current wall-clock

Listing 3.5: Code example showing how to sample current time data and calculate the real-time delay of a code part.

```
#include <time.h>

struct timespec before;
struct timespec after;
struct timespec real_time_delay;

/* Sample current time before code part. */
clock_gettime(CLOCK_MONOTONIC_PRECISE, &before);

/* Some code part to time. */
...

/* Sample current time after code part. */
clock_gettime(CLOCK_MONOTONIC_PRECISE, &after);

/* Calculate the real-time delay. */
if ((after.tv_nsec - before.tv_nsec) < 0) {
    real_time_delay.tv_sec = after.tv_sec - before.tv_sec - 1;
    real_time_delay.tv_nsec = after.tv_nsec - before.tv_nsec + 1000000000;
} else {
    real_time_delay.tv_sec = after.tv_sec - before.tv_sec;
    real_time_delay.tv_nsec = after.tv_nsec - before.tv_nsec;
}
```

time that are affected by the current time of day value of the system. These clocks may be modified by protocols like the *Network Time Protocol* (NTP) [RFC5905]. There are also *monotonic* clocks that increase in SI seconds, that cannot be modified, and are running from an unspecified point in time.

To measure the real-time delay of a code part, the current time must be sampled both before and after running the code part. The real-time delay is the difference of these values. Since we are interested in the time difference, there is no need to sample the current-time-of-day clock. We choose to sample the *monotonic* clock because it cannot be modified while experiments are running, and it provides nanosecond-precision time samples. The *monotonic* clock can be sampled with the C function `clock_gettime` with the *clock ID* `CLOCK_MONOTONIC_PRECISE`[5].

3.1.6 Data analysis method

All of the data sampled are put into different log files for later parsing and manipulation. When using the C function `system` to execute command-line tools from the application, either the tool implicitly creates a log file as a result, or the output of the tool is redirected to a log file. The data that are sampled with the C function `clock_gettime` are initially stored inside the

[5]FreeBSD and Linux have different names for clock IDs. Both systems provide different precision for the different clocks. For example, the *monotonic* clock can be sampled at maximum precision, or at the precision of a CPU tick. In this thesis, we consider the most precise clock.

application, but later written to a log file. The log files are named based on the experiment scenario and parameters, which makes it easier to identify them later. We use shell scripts to extract the relevant data from the log files, use R [76] to calculate the statistical data, and use gnuplot [22] to visualize the data.

3.1.7 NEAT evaluation test suite

We have made a *test suite* to evaluate the performance of NEAT [65]. The goal of this test suite is to measure the *resource overhead* of the NEAT library compared to libuv and kqueue.

The test suite includes source code in C for client and server applications programmed using NEAT, libuv, and kqueue. It also includes shell scripts to manage the resource usage data that are sampled during experiment runs, and data visualization shell scripts. The test suite makes it possible for anyone to reproduce the results demonstrated in this thesis.

Chapter 4

Experimental setup

This chapter describes the experimental setup that is used to perform the experiments considered in this thesis work. First it describes the physical testbed setup that is used to run the experiments. Then, TEACUP (TCP Experiment Automation Controlled Using Python) is introduced, which is used to administer the experiment runs on the testbed. Then, the experiment scenarios considered in this thesis is presented. Finally, the different configuration settings that are set prior to running the experiments are presented.

4.1 Testbed topology

This section provides a detailed description of the testbed topology we have used to run the experiments, describing the relevant hardware and software components and how they interact. The experiments were run in the TEACUP testbed setup in the CPS lab at the Department of Informatics, University of Oslo. For a complete overview of this testbed setup, how to use TEACUP, and how we extended TEACUP to meet our requirements, see Appendix A.

4.1.1 Overview

Figure 4.1 gives an overview of the experimental network testbed setup we have used to run our experiments. The testbed consists of two experimental hosts that are used to run server and client applications respectively. These two experimental hosts are connected to a router which routes the packets sent from one experimental host to the other. By having a router on the network path between the experimental hosts, we are able to emulate various network conditions by configuring the router.

The experimental hosts and the router are connected to a control server that administers all the experiments. The control links connecting the experimental hosts and router to the control server are used by the control server to control the experiments and to retrieve different log files from the experiments. The experimental links connecting the experimental hosts to the router are used to send experimental data and packets. The

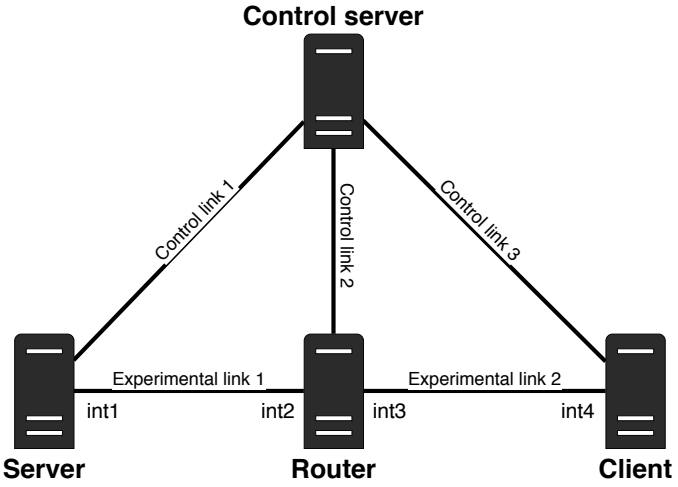


Figure 4.1: The experimental network testbed setup

Hardware type	Model
Machine	HP Compaq 8100 Elite CMT
CPU	Intel Core i7-870 @ 2.93 GHz
RAM	4 x Samsung 4GB PC3-10600 DDR3-1333MHz
Experimental interface NIC	Intel I210 Gigabit Network Connection
Control interface NIC	Intel 82578DM Gigabit Network Connection

Table 4.1: The hardware components of the experimental hosts.

experimental hosts and router are configured so that none of the control packets sent on the control links can interfere with the experimental traffic on the experimental links.

4.1.2 Hardware

The hardware setup of the experimental hosts is listed in Table 4.1. The hardware setup of the router is listed in Table 4.2. Note that the hardware components of the experimental hosts and the router is the same except that the router consists of an additional NIC to connect the experimental hosts. Also note that that the different machines of the testbed are connected by a *Cisco Catalyst 2950* switch. See Appendix A for a complete description of the testbed.

4.1.3 Software

The experimental hosts run FreeBSD 11.0-RELEASE-p1 and applications are compiled with clang version 3.8.0. The router runs Ubuntu 16.04.2 with

Hardware type	Model
Machine	HP Compaq 8100 Elite CMT
CPU	Intel Core i7-870 @ 2.93 GHz
RAM	4 x Samsung 4GB PC3-10600 DDR3-1333MHz
Experimental interface NICs	2 x Intel I210 Gigabit Network Connection
Control interface NIC	Intel 82578DM Gigabit Network Connection

Table 4.2: The hardware components of the router.

Linux kernel 4.6.0[1]. The control server runs Ubuntu 16.04.1 with Linux kernel 4.8.0-36-generic.

4.2 Controlling experiments with TEACUP

TEACUP (TCP Experiment Automation Controlled Using Python) is a tool that simplifies the process of running TCP experiments in a testbed scenario by automating the experiment runs and providing a config file where the experiment parameters can be defined [47, 86]. The experiments performed in this thesis were controlled using TEACUP. TEACUP only supports running TCP experiments by default, which means that it cannot automate all kinds of experiments. To enable TEACUP to run any kind of application on the experimental hosts so that any kinds of traffic generators or traffic sinks can be used, we extended the TEACUP code with the support for custom traffic generators and custom loggers. For more information on this extension, see Section A.2. This extension enables running NEAT, libuv, and kqueue applications in different scenarios and to sample the resource usage of these applications.

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

For detailed information about the TEACUP configuration files we used for the experiments, see Appendix A.

[1]The Linux kernel of the router is patched with Web10G [91], but this should not affect our experiments as we are only concerned with the performance of the experimental hosts.

4.3 Experiment scenarios

This section describes the different experiment scenarios considered in this thesis to answer *RQ*.

In all the experiments, the router was configured to shape the bandwidth at 10 Mbit/s for the outbound interfaces *int2* and *int3* (see Figure 4.1). Also, the router was configured to delay packets for 50 ms on the same interfaces using *netem* [66]. This router setup emulates a network path with a RTT of 100 ms and 10 Mbit/s bandwidth, which is used to have more control over the behaviour of the network, and to emulate the network conditions when accessing services in the Internet. The socket buffers in the experimental hosts were set sufficiently large so that the router can become the bottleneck without limiting the data rate in the experimental hosts.

4.3.1 Connection establishment

Key functionality in NEAT includes translating NEAT properties specified by the user into the best available transport solutions, and then perform transport selection using the Happy Eyeballs mechanism. To evaluate the performance of this functionality, we decide to measure the resource usage of the connection establishment process in NEAT and compare these results to similar experiments performed with libuv and kqueue.

Overview

Table 4.3 specifies how we define the *connection establishment period* of NEAT, libuv and kqueue. Sampling of CPU time, Resident Set Size, and current time is performed at the beginning and end of the connection establishment period to be able to calculate the total CPU usage, memory usage, and delay of connection establishment.

In the definition of the connection establishment period we choose to specify the start of the period from the point the different libraries and APIs are *initialized*. This is done because the different APIs perform different code at different places in the API functions, and to get comparable results it is important to include all similar code parts between the APIs in the resource usage measurements. For example, alternatively we could start to sample data from the start of the event loop instead of including initialization code, but then we would exclude the resource impact of issuing the BSD sockets API function `connect` for libuv and kqueue which is called before the event loop starts for all flows. In comparison, the NEAT function `neat_open` does not call `connect` internally, but schedules this call until the NEAT event loop has started.

We do not run the Policy Manager of NEAT in any of the connection establishment experiments. Instead, we use the internal simple Policy Manager provided by the NEAT library that can translate simple NEAT properties to transport solutions. In the experiments, the NEAT properties passed to the NEAT applications specifies that certain transport protocols

API	Definition of connection establishment period
NEAT	From the point the NEAT function *neat_init_ctx* is issued until all NEAT flows considered in the experiment have successfully connected and the NEAT callback *on_connected* has been called for each of the NEAT flows.
libuv	From the point the libuv function *uv_loop_init* is issued until all flows considered in the experiment are writable or readable and the associated callbacks that signals that the connection is established have been called.
kqueue	From the point the kqueue function *kqueue* is issued to create the kqueue until all the flows considered in the experiment have connected and the kqueue function *kevent* has returned successfully with each of the flows marked writable or readable.

Table 4.3: Definition of connection establishment period for the different APIs considered in this thesis.

are *required*. This allows us to control which transport protocols are used in experiments.

In the experiments we consider the time it takes for different flows to be established in NEAT, libuv and kqueue. This delay is measured in SI milliseconds. We define the per-flow connection delay in NEAT as the elapsed time from when the NEAT function *neat_open* is issued for a flow until the *on_connected* callback is called for that same flow. For libuv and kqueue, we define the per-flow connection delay as the elapsed time from when the *connect* is issued for a socket until that same socket is writable. In libuv we consider the socket to be writable when the associated callback for that event is called. For kqueue we consider the socket writable when the *kevent* function notifies the application that the socket is writable.

For more information on the TEACUP configuration files we used to run connection establishment experiments, see Appendix A. For references to the source code for the NEAT, libuv, and kqueue applications that were considered to measure connection establishment resource usage, see Appendix B.

Experiment parameters

This section describes the different experiment factors and parameters are considered for the evaluation of the resource usage during connection establishment in NEAT, libuv, and kqueue.

Scenario The experiment scenario is a client application connecting to a server application. The resource usage data is sampled both on the client-side and the server-side. The router emululates a RTT of 100 ms and bandidth of 10 Mbps.

APIs The connection establishment experiments are performed with NEAT, libuv, and kqueue applications. In each experiment, the same API is used both on the server-side and client-side.

Number of flows We have considered opening different numbers of flows to put the APIs under different load in the experiments. We have considered 1, 2, 4, ..., 256 flows. This large range enables investigating how the APIs scale to an increased load. Note that when Happy Eyeballs between TCP and SCTP is used in the NEAT experiments, the number of connection requests initiated is not equal to the number of flows opened.

Transport protocols We have considered establishment of both TCP and SCTP connections. In every experiment except the experiments using Happy Eyeballs, the client and server are initiating/listening to the same transport protocol.

Happy Eyeballs scenarios In some of the NEAT experiments, Happy Eyeballs is performed between TCP and SCTP on the client-side. We have considered the following Happy Eyeballs scenarios:

1. Server only listens to TCP.
2. Server only listens to SCTP.
3. Server listens to both TCP and SCTP. The default Happy Eyeballs delay of 10 ms is used[2]. This causes TCP to always win over SCTP.
4. Server listens to both TCP and SCTP. The Happy Eyeballs delay is set to 260 ms to ensure that SCTP always wins over TCP.

Number of experiment runs All the experiments were run 10 times to get a good overview of the distribution of the data[3].

Performance metrics All the performance metrics listed in Table 3.1 are considered.

[2]When NEAT performs Happy Eyeballs with several candidates, it adds a static delay of 10 ms between initiating the candidates to not cause bursty behaviour and to give the best transport solutions a head-start to establish connections first. We have conntributed to the NEAT library by adding an option where the Happy Eyeballs delay can be set by the application.

[3]The profiling experiments using the Valgrind tools *Callgrind* and *Massif* were run independently of the other experiments, and were only run for the experiments that we wanted to analyze in greater detail. For example, we only ran profiling experiments in the cases of 1 flow opened and 256 flows opened. Also, these experiments were run only once to give an indication of the code parts with CPU or memory bottlenecks.

	Definition of data transfer period
Client-side (sender)	From the point right before the first data is sent until all data has been sent.
Server-side (receiver)	From the point right before the first data is read until all data has been read.

Table 4.4: Definition of data transfer period.

4.3.2 Data transfer

Although the novelty in NEAT is attributed to the platform- and protocol-independent selection of transport solutions based on the specified NEAT properties, it is also important to evaluate NEAT during data transfer. The NEAT API functions that send and receive data is also platform- and protocol-independent, and it is desired to understand how much resource overhead such an API introduces. To evaluate this functionality, we decide to measure the resource usage of sending different data object sizes in NEAT and compare these results to similar experiments performed with libuv.

Overview

Table 4.4 specifies how we define the *data transfer period* based on whether the application is receiving or sending data. Sampling of performance metrics is performed at the beginning and end of the data transfer period to be able to calculate the total resource usage of the data transfer.

We have developed both HTTP client and HTTP server applications running NEAT, libuv, and kqueue. These applications are used to evaluate the resource usage of data transfer. The HTTP applications work the following way. First the HTTP client connects to the HTTP server using either TCP or SCTP. When the connections are established the HTTP client sends HTTP POST [RFC7231] requests to the HTTP server with enclosed user data of arbitrary length in the body of the request messages. When the HTTP server parses the HTTP requests, it can determine the length of the user data contained within and the number of bytes that remains to be received. The HTTP applications use the *picohttpparser* [74] library to parse the HTTP requests.

For more information on the TEACUP configuration files we used to run data transfer experiments, see Appendix A. For references to the source code for the NEAT, libuv, and kqueue applications that implements the data transfer functionality, see Appendix B.

Experiment parameters

This section describes the different experiment factors and parameters we have considered for the evaluation of the resource usage during data

transfer in NEAT and libuv.

Scenario The experiment scenario is a client application connecting to a server application and then sending a specific number of bytes. The data transfer resource usage is sampled both on the receiving side (server-side), and on the sending side (client-side).

APIs Due to time limitations we only performed data transfer experiments with NEAT and libuv applications. In each experiment, the same API is used both on the server-side and client-side.

Number of flows We have considered opening different numbers of flows to put the APIs under different load in the experiments. We have considered 1, 2, 4, and 8 flows to determine how the increase in the number of flows affects the resource usage.

Data object sizes We have considered sending data objects of 1 kB, 10 kB, 100 kB, 1000 kB, and 10000 kB. Note that the data objects are sent on every flow in the experiment.

Transport protocols We have considered data transfer with both TCP and SCTP connections.

Number of experiment runs All the experiments were run 10 times to get a good overview of the distribution of the data.

Performance metrics All the performance metrics listed in Table 3.1 are considered except the CPU instruction count and heap memory usage.

We did not profile the applications during data transfer with the Valgrind tools *Callgrind* and *Massif* because we found the overhead of NEAT to be small, and did not need to determine the bottleneck in the code.

4.4 Configurations

This section describes the different configurations and options that are set prior to every experiment run. It describes the general configurations that apply to both the experimental hosts, and configurations that are specific to either the server-side or the client-side.

4.4.1 General configurations for the experimental hosts

The following configurations are made for all the experimental hosts.

`net.inet.tcp.recvbuf_auto=0`

This option disables the auto-tuning of the data receiver's *receive window* used in the *flow control* mechanism. Instead we set a statically large receive window size so that the bottleneck in the network always occurs in the router.

`net.inet.tcp.sendbuf_auto=0`

Same as for receiver window auto-tuning, this disables the auto-tuning of the sender's window, but this relates to the *congestion window*. Instead, we set a statically large sending window size so the bottleneck in the network always occurs in the router.

`net.inet.tcp.recvspace=300000`

Set a statically large receive window size. Since the BDP of the network path between the experimental hosts is $100Mbps/s * 100ms = 125000bytes$, we set a receive window that is more than double this size so that the bandwidth is not limited by the receiver.

`net.inet.tcp.sendspace=300000`

Set a statically large sending window size.

`net.inet.sctp.recvspace=300000`

Also set a statically large receive window for SCTP. Auto-tuning of receive window is not supported for SCTP.

`net.inet.sctp.sendspace=300000`

Also set a statically large sending window size for SCTP. Auto-tuning of congestion window is not supported for SCTP.

`net.inet.tcp.sendbuf_max=2097152 (set by TEACUP)`

This option is set by TEACUP, but should not be relevant since we disable window auto-scaling.

`net.inet.tcp.recvbuf_max=2097152 (set by TEACUP)`

This option is set by TEACUP, but should not be relevant since we disable window auto-scaling.

`net.inet.tcp.msl=5000`

Set the *Maximum Segment Lifetime* (MSL) of TCP segments to a small value (in milliseconds). This value is used when calculating the *TIME_WAIT* interval of TCP connections [RFC793]. The formula to determine the *TIME_WAIT* value is $2 * MSL$. By lowering the *TIME_WAIT* interval to 10 seconds, we are guaranteed that no TCP connections are lingering in *TIME_WAIT* state between experiments.

`net.inet.tcp.tso=0 (set by TEACUP)`

Disables the *TCP Segmentation Offload* (TSO) mechanism that enables large packets to be sent through the network stack without being fragmented. Instead, the *Network Interface Controller* (NIC) is responsible for fragmenting the packets before transmission can begin. This option can enable higher throughput for Gigabit network speeds, but can also make the transfer bursty [41]. Since our goal is to emulate a certain bandwidth and a certain delay in our network, TEACUP disables this option to get as stable network conditions as possible.

`net.inet.tcp.hostcache.expire=1 (set by TEACUP)`

Sets the timeout in seconds before a *host cache* entry is marked as *expired*. The *host cache* stores *entries* with information about the path between the local peer and the remote peer, like RTT, *Path Maximum Transmission Unit* (PMTU), and *Slow-Start Threshold* (ssthresh) [84]. TEACUP sets *host cache* options in such a way that the entries are *purged* (removed) between every experiment run so that subsequent experiments can start from a clean state.

`net.inet.tcp.hostcache.prune=5 (set by TEACUP)`

Remove expired *host cache* entries every 5 seconds. This ensures that no *host cache* data remain between experiment runs.

`net.inet.tcp.hostcache.purge=1 (set by TEACUP)`

Tells the *host cache* to *expire* all the entries on the next *prune run*. Since we have set the option to prune often, all the *host cache* entries will be removed beween experiment runs.

`net.inet.tcp.cc.algorithm=newreno`

Sets the congestion control algorithm to *newreno* for TCP. We choose to use this algorithm because it is simple and works well to achieve our goal of emulating a bandwidth of 10 Mbps.

4.4.2 Server-side configurations

There are no specific configurations made only for the server-side. The default values can sufficiently handle up to 256 connections from the client applications and send/receive packets for those connections.

4.4.3 Client-side configurations

`net.inet.ip.portrange.randomized=0`

When connecting a socket to a remote peer, an *ephemeral port* is assigned to the connection. Normally this port number is chosen at random from a given interval, e.g. between 1024 and 65535. This option disables this *randomization* of port number assignment, and instead assigns the port numbers in *sequence*. We found that if we do not set this option, we would sporadically get a "`Address already in use`" error message when calling `connect`. We could not determine the source of this error, but found setting this option a good work-around.

Chapter 5

Evaluation

This chapter presents the results from the performance evaluation of the NEAT library. It considers the performance metrics listed in Table 3.1, and considers the experimental setup described in Chapter 4. The purpose of this chapter is to showcase the local resource overhead of using the NEAT library compared to other APIs in different scenarios, and identify the bottlenecks in the library code.

The two main scenarios considered in this chapter are:

1. **Connection establishment**: How connection setup influences the resource usage of an application.
2. **Data transfer**: How transferring web objects of different sizes influences the resource usage of an application.

This chapter illustrates how the local resource usage of an application is affected by the number of opened flows. In libuv and kqueue the number of opened flows is simply the number of times the BSD sockets API function `connect` is called, while in NEAT it is the number of opened NEAT flows. By considering different numbers of opened flows in different experiment scenarios, it can be observed how the APIs scale to different load.

Unless explicitly stated otherwise, the figures show the minimum, 10th percentile, median, 90th percentile, and maximum values of the sampled data. By including the distribution of the data in the figures, the stability of the data can be evaluated.

This chapter is split into two parts. The first part shows the delay, CPU usage, and memory usage during connection establishment for both client and server applications. The second part shows the CPU usage during data transfer for both client and server applications.

5.1 Connection establishment

This section presents the evaluation of the connection establishment scenario. This scenario considers initiating and accepting connection requests without transmitting any additional data. Table 4.3 in Section 4.3 describes how *connection establishment* is defined on both client- and server-side for the different networking APIs considered in the experiments.

Scenario	Memory usage overhead (kilobytes)	Meory usage overhead (%)
1 TCP flow	72.20	71
256 TCP flows	108.56	104
1 SCTP flow	70.62	34
256 SCTP flows	99.84	41

Table 5.1: The connection establishment delay overhead of using NEAT compared to libuv on the client-side.

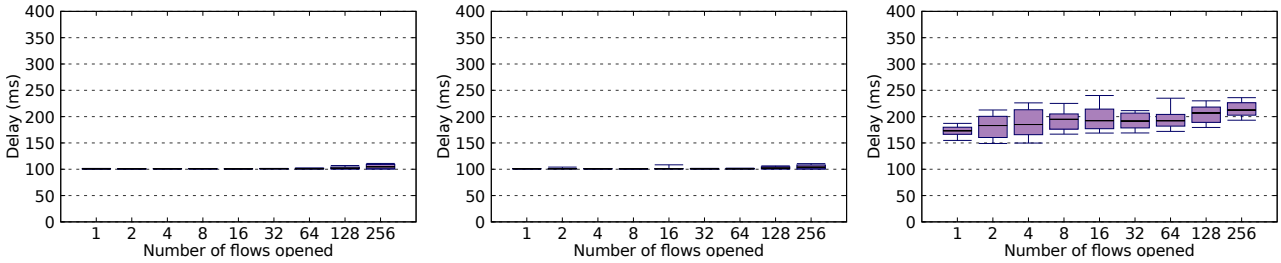
5.1.1 Connection establishment delay

Figures 5.1 and 5.2 show the per-flow connection establishment delay on the client-side when opening TCP and SCTP flows respectively. The server-side is listening to TCP and SCTP respectively. The figures show how the number of flows opened affects the per-flow connection establishment delay. All the flows are opened in a loop before starting the event loop.

It can be observed that SCTP flows use 100 ms more than TCP flows to connect. This is because SCTP performs a 4-way handshake while TCP performs a 3-way handshake, and the RTT of the experiments was set to 100 ms. The values of the per-flow SCTP connection delay data is deviating more from the median value compared to TCP. This is because it takes a longer time to establish SCTP connections compared to TCP connections because SCTP needs to generate a state cookie for the connections [RFC4960]. One of the reasons why the connection establishment delay values reported in Figures 5.1c and 5.2c are larger than the results for libuv and kqueue is that the NEAT function `neat_open` does not initiate the connection immediately. Instead, the connection is initiated once the application enters the NEAT event loop.

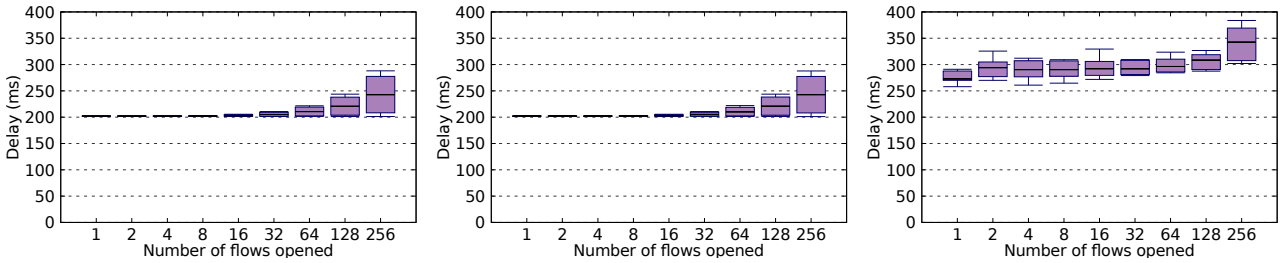
Table 5.1 shows the connection establishment overhead in NEAT compared to ilbuv. These overhead results show that the connection establishment delay is large in NEAT compared to libuv and kqueue. They also show that the initialization of the NEAT library introduces a large overhead since opening 1 NEAT flow takes 72 ms longer than in libuv and kqueue. However, as can be seen from the results when opening 256 flows, the 72 ms overhead is not introduced for every flow.

Figure 5.3 illustrates how much more time it takes to execute the loop that opens connections in the kqueue client application when opening SCTP flows compared to TCP flows. It can be observed that the delay overhead of initializing SCTP sockets is larger than for TCP. It takes about 2 times longer to initialize and connect with the SCTP sockets compared with the TCP sockets.



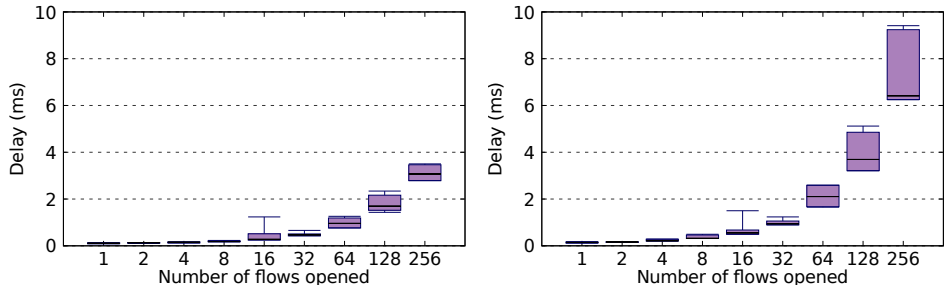
(a) kqueue (b) libuv (c) NEAT

Figure 5.1: Comparison of the connection establishment delay per flow between kqueue, libuv and NEAT when multiple flows are opened concurrently using TCP.



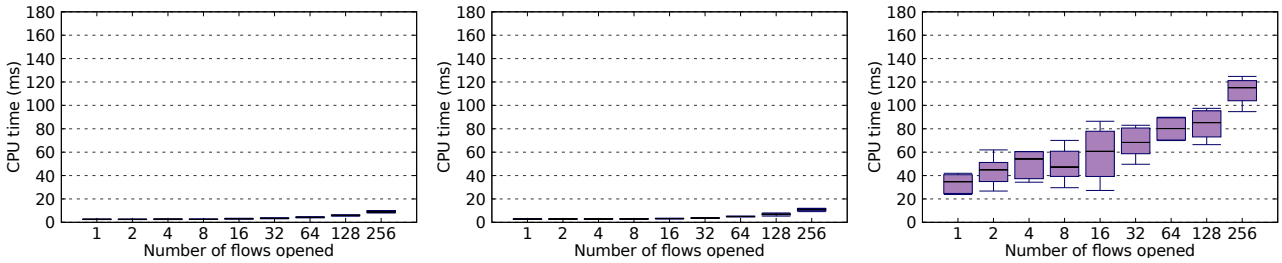
(a) kqueue (b) libuv (c) NEAT

Figure 5.2: Comparison of the connection establishment delay per flow between kqueue, libuv and NEAT when multiple flows are opened concurrently using SCTP.



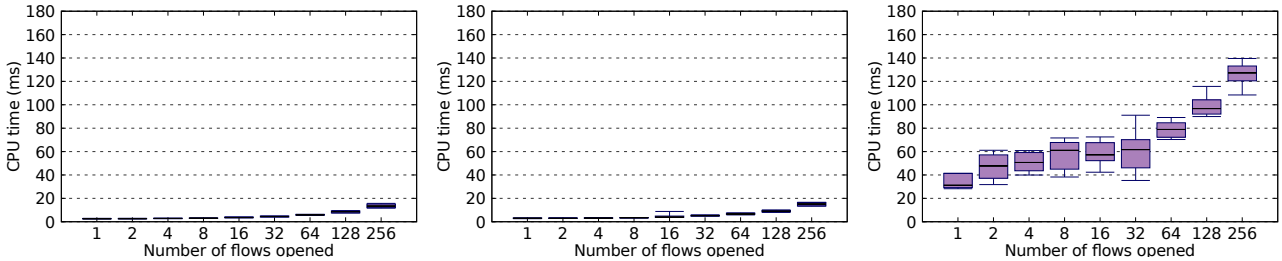
(a) TCP (b) SCTP

Figure 5.3: The delay overhead of using SCTP compared to TCP in a loop that creates and connects sockets. This data was sampled in the kqueue client application that also adds each socket to the kqueue.



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

Figure 5.4: The CPU time spent when establishing connections using TCP at the client-side.



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

Figure 5.5: The CPU time spent when establishing connections using SCTP at the client-side.

5.1.2 CPU usage

This subsection presents how connection establishment influences the CPU usage of NEAT, libuv, and kqueue applications. First it presents how the active connection establishment performed by the client-side influences the CPU usage. Then it presents the CPU usage of accepting new incoming connections at the server-side. This subsection also considers the CPU impact of using the *Happy Eyeballs* mechanism in NEAT.

Client-side CPU usage

Figures 5.4 and 5.5 show the client-side CPU time spent when establishing connections with TCP and SCTP respectively. They show that when the number of outgoing connection requests increases, the CPU time spent by the application also increases. It can be observed that kqueue and libuv have very similar CPU usage, while the overhead of NEAT is large in comparison. Table 5.2 shows the CPU time overhead of using NEAT compared to libuv. The CPU time overhead of the initialization of the NEAT library is large in comparison to libuv by observing that the CPU overhead of opening a single flow in NEAT is 31 ms. Also, the CPU time overhead of opening a single NEAT flow is large in comparison to libuv by observing that the CPU time difference in opening 1 TCP flow and 256 TCP flows in NEAT is 81 ms, while it is 8 ms for libuv. Thus,

Scenario	CPU time overhead (ms)	CPU time overhead (%)
1 TCP flow	31,95	1046
256 TCP flows	104,19	952
1 SCTP flow	28,20	949
256 SCTP flows	112,11	739

Table 5.2: CPU time overhead of using NEAT compared to libuv during connection establishment on the client-side.

the CPU time overhead of opening a single flow in NEAT compared to libuv is approximately 10 times larger. Finally, the use of SCTP introduces additional CPU usage for all the APIs (see Figure 5.5 compared to Figure 5.4) but the additional cost is greater in NEAT in comparison.

In order to understand the large resource overhead of the NEAT library on the client-side, we performed CPU profiling of the NEAT application using the Callgrind tool of the Valgrind tool suite (see Section 3.1.4 for more information). Table 5.3 presents the CPU instruction executed by an extract of functions called in the NEAT, libuv, and kqueue applications when opening 256 TCP flows at the client-side. It shows that the NEAT library imposes a large CPU instruction count overhead compared to libuv and kqueue. As can be observed from the table, about half of the overhead is introduced outside the NEAT event loop, while the other part is introduced within the event loop.

Table 5.4 presents the total number of CPU instructions executed by the NEAT functions called from outside the NEAT event loop when opening 256 TCP flows at the client-side. It shows that the functions that are called 256 times have the largest overhead (`neat_open`, `neat_set_property`, `neat_new_flow`). By investigating these functions individually, we find that much of the CPU usage is derived from calling various JSON operations from the libjansson library [38]. Table 5.5 shows the most CPU demanding operations that are executed within these NEAT functions. Notably, 6,300,672 CPU instructions are executed to convert the NEAT properties to JSON objects, and later 8,683,264 CPU instructions are executed to convert these JSON objects back to string representation to be sent to the *Policy Manager*. This happens although the *Policy Manager* is not running.

Table 5.6 presents an extract of the most CPU demanding internal functions of the NEAT library when opening 256 TCP flows at the client-side. The function `nt_send_result_connection_attempt_to_pm` is called every time a *Happy Eyeballs* connection attempt either suceeds or fails. In this case, the result of the connection attempt is sent to the *CIB* of the *Policy Manager* to be cached. This is done by first converting the JSON data structures containing the connection result to a string, and then sending that string to the *Policy Manager*. This JSON-to-string conversion occurs even when the *Policy Manager* is not running. In the case of opening 256 TCP flows, this operation uses 16,129,024 CPU instructions which

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

Table 5.3: Total number of CPU instructions executed by various functions in the kqueue, libuv and NEAT client applications when opening 256 TCP flows at the client-side. Note that we have enclosed the kqueue event loop in a separate function `start_event_loop` to make it more comparable to NEAT and libuv.

Function	CPU instructions	Number of times called
`neat_open`	22,889,216	256
`neat_set_property`	7,005,184	256
`neat_new_flow`	702,464	256
`neat_init_ctx`	260,664	1
`neat_set_operations`	39,936	256

Table 5.4: Total number of CPU instructions executed by NEAT functions that are called outside the NEAT event loop when opening 256 TCP flows on the client-side.

NEAT function	CPU demanding operation	CPU instructions
`neat_open`	`json_dumps`	8,683,264
	`getnameinfo`	3,449,600
	`json_pack`	2,913,280
	`json_delete`	1,474,560
`neat_set_property`	`json_loads`	6,300,672
`neat_new_flow`	`calloc`	573,696

Table 5.5: Extract of the most CPU demanding operations executed within the NEAT functions that are called outside the NEAT event loop when opening 256 TCP flows on the client-side.

NEAT internal function	CPU instructions
`nt_send_result_connection_attempt_to_pm`	16,129,024
`open_resolve_cb`	10,925,056
`uvpollable_cb`	1,251,584

Table 5.6: Extract of the most CPU demanding internal NEAT functions when opening 256 TCP flows on the client-side.

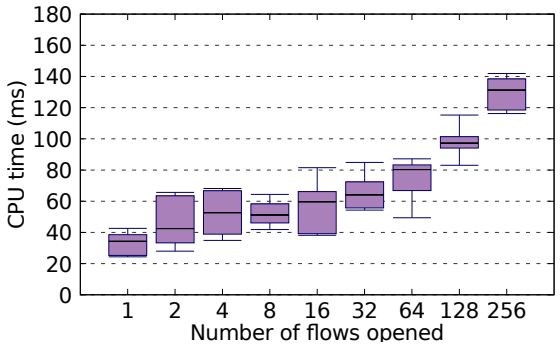
is much more than the CPU instructions executed by the kqueue and libuv applications all together. Another CPU demanding internal NEAT function is `open_resolve_cb` which is responsible for constructing a list of connection candidates based on all the available source-destination address pairs. The overhead of this function is related to address-to-name resolving and gathering of interface information.

We have investigated the CPU usage of the NEAT library when only a single TCP flow is opened. From Figure 5.4 it can be observed that the CPU time spent by opening a single flow in NEAT is greater than opening 256 flows in libuv and kqueue. However, *Callgrind* analysis shows that the total number of CPU instructions executed when opening a single flow in NEAT is approximately 600,000 for the `main` function. This number of CPU instructions is in fact smaller than the number of CPU instructions reported for libuv and kqueue in Table 5.3.

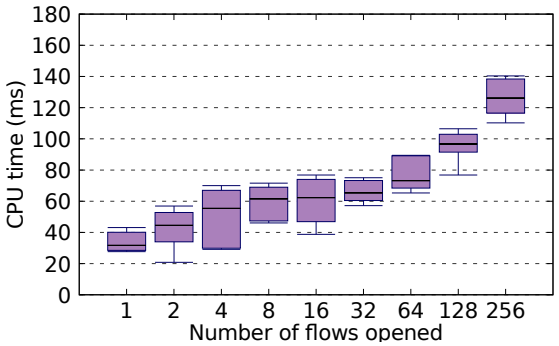
We also investigated why the use of SCTP instead of TCP introduces more CPU time when using NEAT compared to using libuv (Figure 5.4 vs Figure 5.5). *Callgrind* analysis shows that when SCTP is used as a transport protocol in NEAT, the NEAT library first reads SCTP control messages related to the association establishment process before the application is notified that the connection has been established. One of these SCTP control messages that signals to the application that the association has been established is `SCTP_ASSOC_CHANGE` [RFC6458]. In the kqueue and libuv applications that we have developed, SCTP control messages are not used. This is why the overhead of using SCTP compared to TCP is greater in the NEAT application compared to the kqueue and libuv applications.

Figures 5.6 and 5.7 show the CPU time spent when establishing connections by performing *Happy Eyeballs* between TCP and SCTP in the NEAT client application. Figure 5.6a considers establishing connections to a server listening only to TCP, while in Figure 5.6b the server is listening for both TCP and SCTP. In Figure 5.6b the default setting for *Happy Eyeballs* delay is used[1]. This causes TCP connections to always win over SCTP. Figure 5.7a considers establishing connections to a server listening only to SCTP, while in Figure 5.7b the server is listening to both TCP and SCTP. In Figure 5.7b we have changed the *Happy Eyeballs* delay value so that SCTP

[1]The NEAT library adds a delay of 10 ms between *Happy Eyeballs* candidates by default. We have added an option to the NEAT library that enables us to modify this delay.



(a) NEAT client performing HE between TCP and SCTP. The server only listens to TCP. TCP always wins.



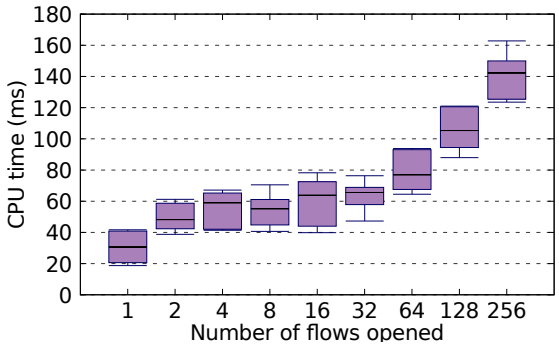
(b) NEAT client performing HE between TCP and SCTP. The server listens to both TCP and SCTP. TCP always wins.

Figure 5.6: The CPU time spent when establishing connections by performing HE between TCP and SCTP at the client-side. TCP connections are initiated before SCTP connections and always win.

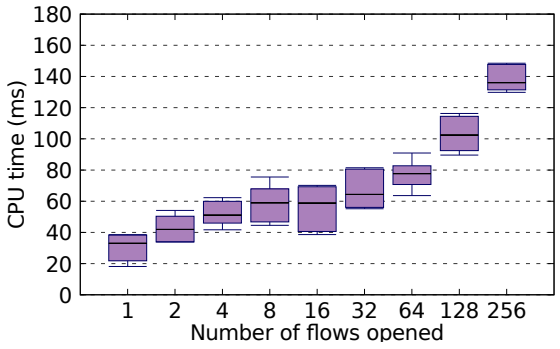
always wins over TCP[2].

Based on the results in Figure 5.6 and Figure 5.7 the CPU impact of the Happy Eyeballs mechanism on the client-side is independent of which protocols are listened for at the server-side. Based on the figures it can also be observed that the CPU time spent by the application is greater when using Happy Eyeballs compared to using a single transport protocol (see Figure 5.4 and Figure 5.5).

[2]We found that a *Happy Eyeballs* delay value of 260 ms would cause SCTP to always win over TCP in our testbed setup.



(a) NEAT client performing HE between TCP and SCTP. The server only listens to SCTP. SCTP always wins.



(b) NEAT client performing HE between TCP and SCTP. The server listens to both TCP and SCTP. SCTP always wins

Figure 5.7: The CPU time spent when establishing connections by performing HE between TCP and SCTP at the client-side. In the case of Figure 5.7b, the TCP connections are delayed long enough such that SCTP connections always win.

Server-side CPU usage

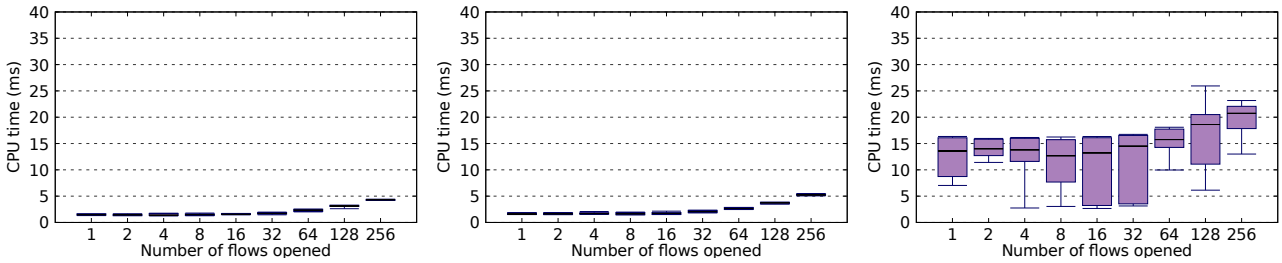
Figure 5.8 and Figure 5.9 show the CPU time spent when accepting TCP and SCTP connections respectively on the server-side. The figures show the results of running NEAT, libuv, and kqueue applications on the server-side, accepting connection requests sent from a remote client. In every experiment the same networking API is used for the server and client applications. The data presented in Figure 5.8 and Figure 5.9 were sampled in the same experiments as considered in Figure 5.4 and Figure 5.5 respectively. More generally, we sampled both client-side and server-side data in all experiment runs. In Figure 5.8 the server applications are listening for TCP and the client that connects is using TCP. In Figure 5.9 the server applications are listening for SCTP and the client that connects is using SCTP.

Compared to the large CPU time overhead of using NEAT compared to libuv and kqueue presented in Figure 5.4 and Figure 5.5, the CPU time overhead on the server-side presented in Figure 5.8 and Figure 5.9 is much smaller in comparison. This is because the NEAT server application does not need to perform Happy Eyeballs to determine the supported transport protocols in the end-to-end network path, and therefore there is no need to store the connection results in the *CIB* repository. Table 5.7 shows the CPU time overhead of using NEAT compared to libuv.

By performing CPU profiling with *Callgrind* we found that most of the CPU instructions are executed in the NEAT event loop calling the `do_accept` NEAT internal function to accept new incoming connections. The CPU time overhead of using SCTP in NEAT compared to using TCP, as can be seen by comparing Figure 5.8c and Figure 5.9c has the same explanation as described in the client-side scenario, namely that the NEAT library reads SCTP control messages before calling `do_accept` to accept the

Scenario	CPU time overhead (ms)	CPU time overhead (%)
1 TCP flow	11,96	744
256 TCP flows	15,50	295
1 SCTP flow	11,34	700
256 SCTP flows	21,71	403

Table 5.7: CPU time overhead of using NEAT compared to libuv during connection establishment on the server-side.

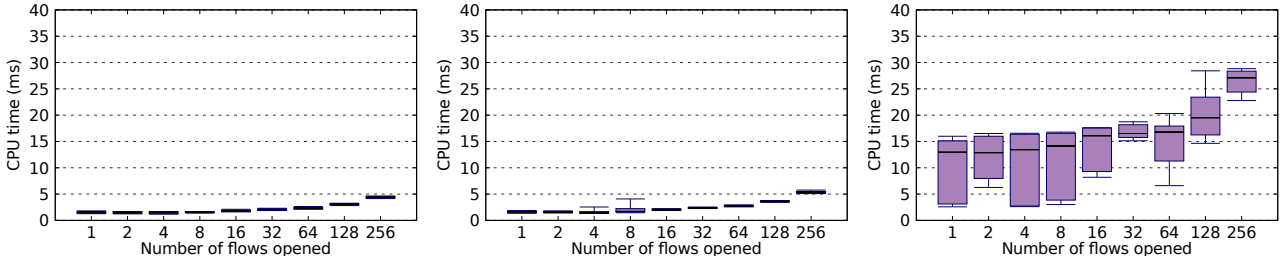


(a) kqueue (b) libuv (c) NEAT

Figure 5.8: The CPU time spent when establishing connections using TCP at the server-side.

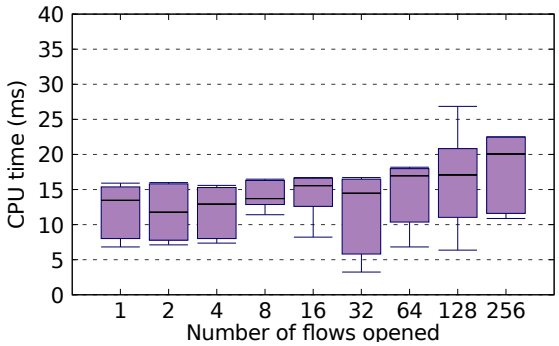
connection.

Figure 5.10 and Figure 5.11 show the CPU time spent when accepting connections in the NEAT server application while the remote client application performs Happy Eyeballs between TCP and SCTP. The figures consider the same scenarios as described for Figure 5.6 and Figure 5.7 but instead show the impact on the server-side. To recap, in Figure 5.10 only TCP connections are established (using the default HE delay of 10 ms), while in Figure 5.10 only SCTP connections are established (by delaying the TCP connections by 260 ms). In Figure 5.10a and Figure 5.11a the application listens to TCP and SCTP respectively. In Figure 5.10b and Figure 5.11b the application listens to both TCP and SCTP.

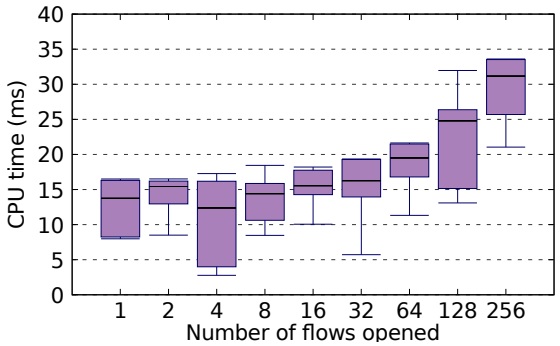


(a) kqueue (b) libuv (c) NEAT

Figure 5.9: The CPU time spent when establishing connections using TCP at the server-side.



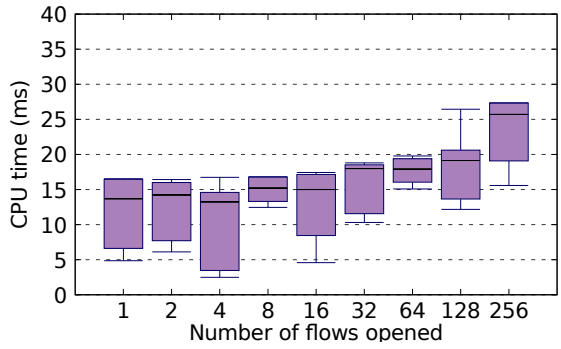
(a) The client performing HE between TCP and SCTP. The NEAT server only listens to TCP. TCP always wins.



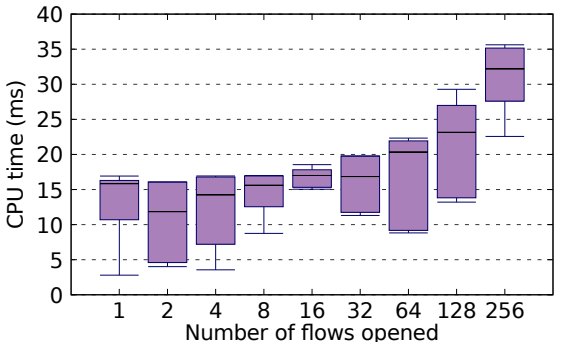
(b) The client performing HE between TCP and SCTP. The NEAT server listens to both TCP and SCTP. TCP always wins.

Figure 5.10: The CPU time spent when accepting incoming connection requests while the remote NEAT client performs Happy Eyeballs between TCP and SCTP. The TCP connections are initiated before the SCTP connections and always win.

In Figure 5.10 and Figure 5.11 it can be observed that when the NEAT server listens for both TCP and SCTP in comparison to only one of them, the CPU time spent by the application increases. This is because the NEAT server accepts twice the number of connections in this case. Also the CPU time impact presented in Figure 5.10b and Figure 5.11b seem to be very similar, which means that the CPU usage of the server application is independent of which transport protocol the client application deems as the winning candidate during connection establishment.



(a) The client performing HE between TCP and SCTP. The NEAT server only listens to SCTP. SCTP always wins.



(b) The client performing HE between TCP and SCTP. The NEAT server listens to both TCP and SCTP. SCTP always wins.

Figure 5.11: The CPU time spent when accepting incoming connection requests while the remote NEAT client performs Happy Eyeballs between TCP and SCTP. In the case of Figure 5.11b, the TCP connections are delayed long enough such that SCTP connections always win.

5.1.3 Memory usage

Client-side memory usage

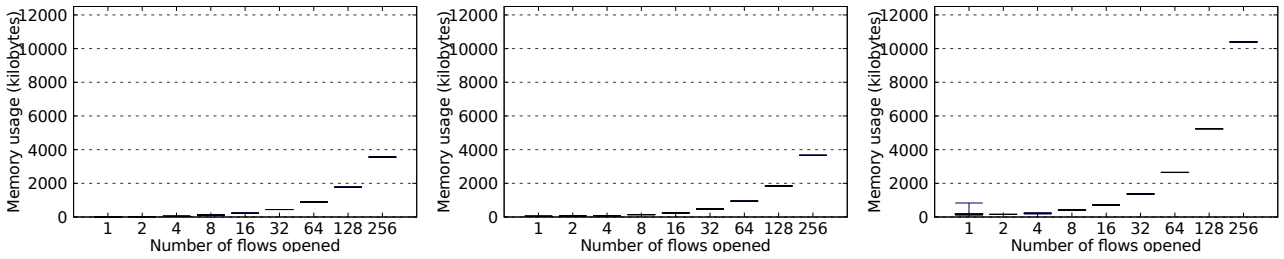
Figure 5.12 and Figure 5.13 show how much the memory usage *increases* when establishing connections with TCP and SCTP respectively. The memory usage considered in these figures is the Resident Set Size of the application. The figures show the results of running NEAT, libuv, and kqueue applications on the client-side, establishing connections to a server. The server application in the experiments uses the same networking API as the client application. In Figure 5.12 the client application is establishing TCP connections to a server that only listens for TCP. In Figure 5.13 the client application is establishing SCTP associations to a server that only listens for SCTP.

Table 5.8 shows the memory usage overhead of using NEAT compared to libuv. In Figure 5.12 and Figure 5.13 it can be observed that libuv adds a very small memory overhead compared to kqueue, and that NEAT has some memory overhead compared to libuv, but this overhead is proportionally not as large as in the CPU results. The memory usage of kqueue and libuv does not seem to be affected by the choice of TCP or SCTP, while the use of SCTP causes the NEAT application to use some additional memory.

In order to analyze what parts of the NEAT code is causing the memory overhead, we performed profiling on the applications using the *Massif* tool of the *Valgrind* tool suite (see Section 3.1.4 for more information). First we considered the scenario in Figure 5.12 with 256 opened TCP flows. In libuv, we find that about 75% of the memory is allocated in `on_connected` while about 25% of the memory is allocated in `main`. The total heap memory usage is approximately 3.4 MiB. In NEAT, we find that about 29% of the memory is allocated in `on_connected`, 9%

Scenario	Memory usage overhead (kilobytes)	Meory usage overhead (%)
1 TCP flow	94.20	230
256 TCP flows	6733.78	183
1 SCTP flow	98.304	240
256 SCTP flows	8272.18	225

Table 5.8: The memory usage overhead of using NEAT compared to libuv during connection establishment on the client-side.

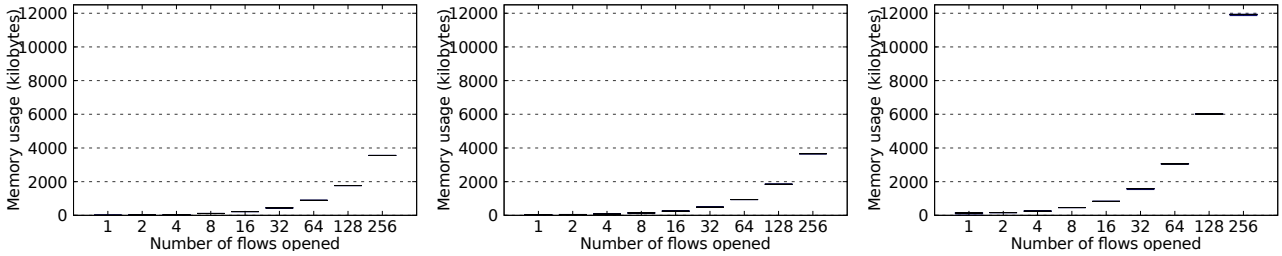


(a) kqueue (b) libuv (c) NEAT

Figure 5.12: The increase in application memory consumption when establishing connections using TCP at the client-side.

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

We also profiled the NEAT application when opening 256 SCTP flows to determine why the memory usage is greater than when using TCP. We find that the memory structures that are allocated for TCP is also allocated for SCTP. However, we also find that the NEAT internal function `resize_read_buffer` is allocating approximately 20 MiB. This is more memory than the reported Resident Set Size in Figure 5.13c. We suspect that this memory is allocated but not resident in RAM because it is not used until data is received.



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

Figure 5.13: The increase in application memory consumption when establishing connections using SCTP at the client-side.

Server-side memory usage

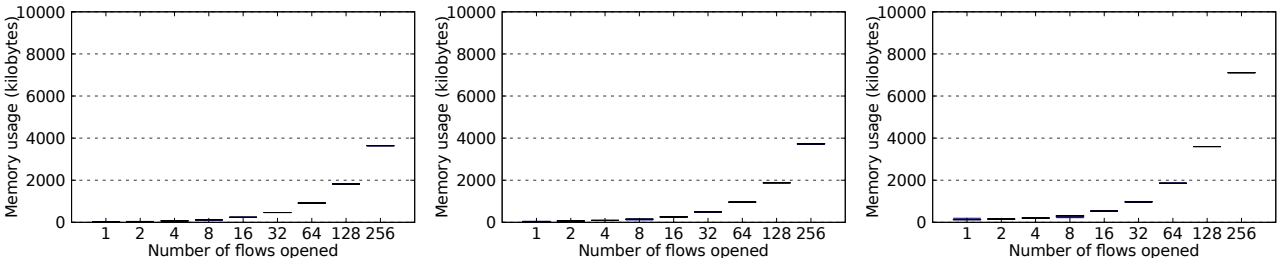
Figure 5.14 and Figure 5.15 show how much the memory usage increases when accepting TCP and SCTP connections respectively. The memory usage considered in these figures is the Resident Set Size of the application. The figures show the results of running NEAT, libuv, and kqueue applications on the server-side, accepting connection requests sent from the remote client. In every experiment the same networking API is used for the server and client applications. In Figure 5.14 the server applications are listening for TCP and the client that connects is using TCP. In Figure 5.15 the server applications are listening for SCTP and the client that connects is using SCTP.

Table 5.8 shows the memory usage overhead of using NEAT compared to libuv. It can be observed that the libuv and kqueue memory usage results for the client-side presented in Figure 5.12 and Figure 5.13 is about idnetical to the memory usage on the server-side, presented in Figure 5.14 and Figure 5.15. However, in the case of NEAT, the memory usage is smaller on the server-side than the client-side. Also, the use of SCTP introduces some overhead in NEAT compared to using TCP. This overhead of using SCTP is not present in the kqueue and libuv results.

By profiling the NEAT server, we find that the memory that is allocated by the `open_resolve_cb` NEAT internal function at the client-side is not present at the server-side because the server is not performing Happy

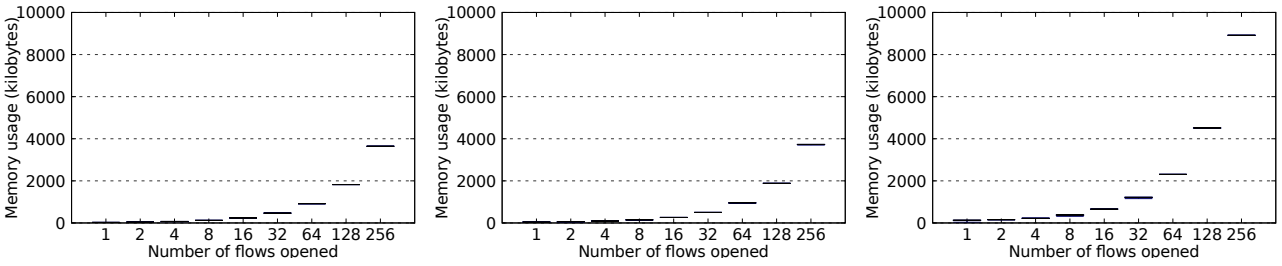
Scenario	Memory usage overhead (kilobytes)	Meory usage overhead (%)
1 TCP flow	86.01	210
256 TCP flows	3379.2	90
1 SCTP flow	86.01	210
256 SCTP flows	5181.44	138

Table 5.9: The memory usage overhead of using NEAT compared to libuv during connection establishment on the server-side.



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

Figure 5.14: The increase in application memory consumption when establishing connections using TCP at the server-side.



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

Figure 5.15: The increase in application memory consumption when establishing connections using SCTP at the server-side.

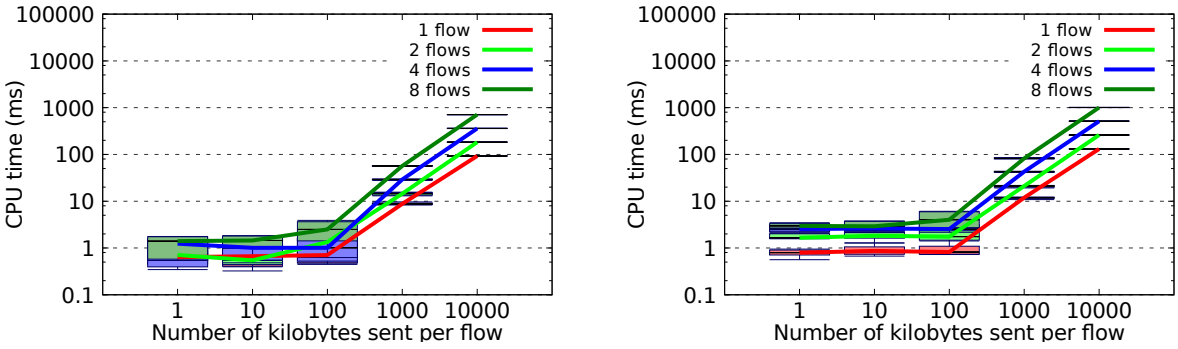
Eyeballs. This seems to be the reason why the NEAT client uses more memory than the NEAT server.

The memory usage overhead of the NEAT server compared to libuv and kqueue is the same as described for the client-side memory results, and is caused by allocating the internal platform- and protocol-independent representation of network sockets in NEAT (*struct neat_pollable_socket*). These structures are allocated when the NEAT function `neat_new_flow` is called.

The memory overhead in Figure 5.15c compared to Figure 5.14c is not completely clear as described for the client-side memory results, but we believe this overhead is caused by the `resize_read_buffer` which is called as a result of receiving SCTP control messages during connection establishment that negotiates the receive buffer size.

5.2 Data transfer

This section presents our evaluation of the data transfer experiment scenario. It considers data transfer in NEAT and libuv using either TCP or SCTP as the transport protocol. The purpose of this section is to evaluate the overhead of using NEAT for data transfer compared to libuv. We define the data transfer period when performance metrics are sampled as described in Table 4.4. For more information regarding the experiment



(a) libuv

(b) NEAT

Figure 5.16: CPU time spent when transferring data using TCP on the client-side for different numbers of flows and for different data object sizes.

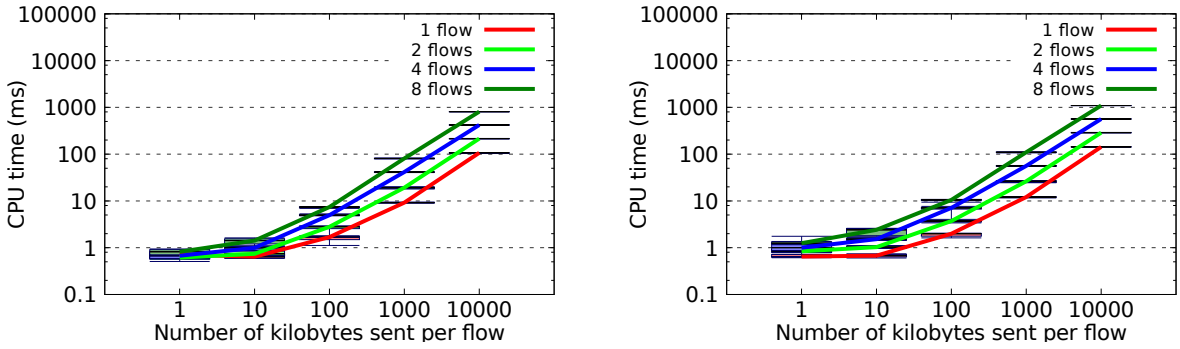
Scenario	CPU time overhead (ms)	CPU time overhead (%)
1 TCP flow, 10 MB	38,63	41
8 TCP flows, 10 MB	298,67	42

Table 5.10: CPU time overhead of using NEAT compared to libuv during data transfer with TCP on the client-side.

scenario, see Section 4.3.

Figure 5.16 shows the CPU time spent on the client-side (sender) during data transfer of different data object sizes and for different numbers of concurrent flows running TCP. Note that the axes are logscale. Table 5.10 shows the CPU time overhead of using NEAT compared to libuv. It can be observed that transferring data in NEAT has a slight CPU time overhead compared to libuv, but this overhead is small in comparison to the total CPU time usage. The reason why the CPU time usage is low until sending 1000 kB and more is most likely because we have set our socket buffer send space to 300 kB, and if less than 300 kB of data is sent the data will be added directly to the socket buffers without any delay.

Figure 5.17 shows the CPU time spent on the server-side (receiver) during data transfer of different data object sizes and for different numbers of concurrent flows running TCP. Note that the axes are logscale. Table 5.11 shows the CPU time overhead of using NEAT compared to libuv. It can be observed that receiving data in NEAT has a slight CPU time overhead compared to libuv, but this overhead is small in comparison to the total CPU time usage.



(a) libuv

(b) NEAT

Figure 5.17: CPU time spent when transferring data using TCP on the server-side for different numbers of flows and for different data object sizes.

Scenario	CPU time overhead (ms)	CPU time overhead (%)
1 TCP flow, 10 MB	36,12	33
8 TCP flows, 10 MB	284,26	35

Table 5.11: CPU time overhead of using NEAT compared to libuv during data transfer with TCP on the server-side.

Chapter 6

Discussion

In this chapter we discuss the evaluation results presented in Chapter 5. The goal is to give an overview of the results, how they can be interpreted in a macroscopic view, whether the NEAT library is scalable in different scenarios, and discuss possible improvements to the code that can mitigate some of the performance issues.

This thesis considers CPU usage, memory usage, and delay as metrics to evaluate the performance of NEAT in different scenarios. In general we find that the use of NEAT introduces a significant resource overhead compared to using libuv and kqueue. The fact that NEAT introduces resource overhead is not a problem by itself. It is expected that an advanced networking library and API like NEAT will be more resource intensive compared to simpler, less complex solutions. Also, NEAT is a prototype library where the development focus has been to provide rich functionality rather than developing efficient code. Even if the NEAT library is too resource intensive in the current version, the performance can be improved by optimizing the code. The resource overhead of a NEAT-like system becomes a problem if it is too high to meet the requirements of the end-users.

We found that the delay of establishing connections in NEAT is high compared to libuv and kqueue. We found that NEAT introduces a delay of about 70-100 ms based on the number of flows and transport protocol used. A small delay is important to real-time applications that have strict timing requirements. We find that the real-time delay during connection establishment in NEAT is too high to be useful for such applications. On the other hand, applications that uses long-lived connections may accept an increased delay during connection establishment in order to leverage the best available network services. In this case, the use of the NEAT library is feasible.

The CPU usage of NEAT was also found to be large in comparison to libuv and kqueue results. We found that the CPU usage when using NEAT was in the order of 500% to 1000% larger than libuv. The analysis we performed uncovered that JSON operations related to converting strings to JSON objects and vice versa constitute a large portion of the total CPU usage when opening many NEAT flows. String manipulation is CPU

intensive and should be avoided if possible by optimized code. The *Concise Binary Object Representation* (CBOR) [RFC7049] format is a solution to decrease the sizes of the JSON objects and thus improve efficiency. Optimally, the NEAT properties should only be converted from string format to some internal format once, instead of first converting to JSON format and then back to string as is done in the current implementation of NEAT. The current NEAT implementation requires that NEAT properties must be specified for every new NEAT flow. An optimization to this could be to enable the application developer to specify the property string once, convert the string to internal representation, and return an identifier which can be used later when opening new NEAT flows to specify the same services without having to convert the string again.

Other CPU intensive operations in the current NEAT version include gathering of interface information and address-to-name resolution, which is performed for every source-destination address pair for every opened NEAT flow. This can be improved by caching the gathered information, and reuse this information when creating new flows.

The memory overhead in NEAT is comparably smaller than the CPU overhead and delay overhead of NEAT. We found that the memory increase by using NEAT is around 100% to 200% larger than when using libuv. The memory usage in NEAT seems to closely relate to the number of open NEAT flows. We found that when the number of NEAT flows doubles, the memory usage doubles, just like the libuv and kqueue results. We argue that the memory usage of NEAT does not seem very large; especially when considering the abundant cheap memory in modern computers. The memory overhead is attributed to allocating the platform- and protocol-independent socket representation internal to NEAT. This is a data structure that aggregates data required by different transport protocols and features. A possible optimization would be to split this structure into several sub-structures related to specific protocols and features, and only allocate the memory that is required. The memory usage of NEAT may still be too large for some applications, like applications running in an embedded environment with very limited resources.

It is important to note that the metrics considered in this thesis are connected. When an application spends a lot of CPU time to execute an operation, the consequence is that the real-time delay of executing the same operation will be large. Also, when the memory usage for an application is large, there might be more cache misses which can lead to increased CPU time and real-time delay.

Chapter 7

Conclusive Remarks and Future Work

In this thesis we investigated the local resource usage of the NEAT library compared to other APIs. This investigation was conducted to give a clearer answer to whether NEAT-like systems can be deployed in the Internet. We were able to quantify the resource overhead of NEAT and find bottlenecks in the code. This answered our *Research Question* (What is the local resource overhead of using NEAT compared to other state-of-the-art APIs?)

7.1 Research findings

In this thesis we evaluated the performance of the NEAT library by comparing the resource usage of NEAT to networking APIs libuv and kqueue. This comparison made it possible to quantify the resource overhead of using NEAT compared to the other APIs and answer our *Research Question*.

In our evaluation we found that NEAT introduces a significant amount of CPU usage and delay when establishing connections compared to libuv and kqueue. The increase in memory overhead when opening/accepting new flows in NEAT is not as large in comparison, and the memory overhead can most likely be supported by the abundant memory available in modern computers. However, due to the general resource overhead of using NEAT, it might not meet the requirements for certain applications and systems. For instance, embedded systems may have limited computing, memory and power resources that cannot handle the overhead of NEAT.

Our evaluation has shed light on the possibility to deploy NEAT-like systems in the Internet. Before this can happen, an efficient implementation needs to be implemented that can compete with existing APIs.

7.2 Future Work

Happy Eyeballs between TCP and SCTP Because a client application uses 1 RTT to establish connections with TCP and 2 RTTs with SCTP,

the result of using Happy Eyeballs between TCP and SCTP is often that TCP wins; even when the SCTP candidate is initiated before the TCP candidate. There are several possible solutions to solve this problem[1]. One solution is to use a user-space implementation of SCTP that can call a user-specified callback function whenever the INIT-ACK message of the SCTP connection establishment process has arrived. In this way, the *Happy Eyeballs* mechanism will be notified that SCTP is supported end-to-end in the network after 1 RTT. However, this solution is not possible for a kernel implementation of SCTP because the user-specified callback can block the kernel process calling the callback. Another possibility is to use a similar API like *TCP Fast Open* (TFO) [RFC7413] which enables application data to be sent (and processed) during connection establishment. The specification of *SCTP Fast Open* is not available yet.

Evaluating flow grouping and priority In this thesis we have considered both TCP and SCTP in the experiments to investigate how the choice of transport protocol impacts performance. However, a seperate study can be performed on the flow grouping and priority functionality of NEAT, which enables individual TCP flows and individual SCTP streams to be prioritized so that they get a specific share of the available bandwidth. An interesting scenario would be to port an existing application to use NEAT, and leverage the priority functionality to improve application performance.

Leverage the Policy Manager to make multihoming decisions A seperate study can use the Policy Manager in NEAT to evaluate multiple scenarios. The Policy Manager enables transport services to be translated to specific protocols and options. Future work may include an investigation of the flexible services that can be offered by running the Policy Manager. For instance, the current RTT of all the available network links connected to an end-host can be sampled and updated continously into the NEAT Policy Manager. When doing this, the choice of which link to send data on for a multihomed host can be decided dynamically by the Policy Manager which can improve application performance.

In summary, we find that the resource usage of NEAT is too large in the current version to be scalable in the Internet. However, we have found many possible optimization solutions that can improve the performance of the library.

[1]The solutions to the problem of using Happy Eyeballs between TCP and SCTP were provided by Michael Tüxen (one of the implementors of SCTP).

Appendix A

TEACUP testbed

This appendix describes the physical network testbed that we used to run the experiments considered in this thesis, and describes how to set up this testbed and run TEACUP experiments on it. First, the topology of the testbed is presented, describing how the different components are connected and configured. Second, the extensions we have made to TEACUP is explained. Finally, an example of how to use TEACUP and write a TEACUP config file is described.

A.1 Testbed setup

Figure A.1 illustrates the physical network testbed in the CPS lab at the Department of Informatics, University of Oslo. The testbed consists of 5 experimental hosts (*E1-E5*), control server, router, and Cisco network switch. The Cisco switch is not shown in the figure to make the illustration of the topology more clean. Instead, the different *Virtual LANs* (VLANs) configured in the switch are included to illustrate the different networks in the testbed. All the hosts are connected to the switch with *Fast Ethernet* connections. The experimental host *E1* is placed in another VLAN than experimental hosts *E2-E5* so that experimental data sent from a host in one VLAN to the other will pass the router. The router can be configured with different traffic control rules to emulate various network conditions.

This appendix does not describe the specific hardware and software specifications of the testbed. The purpose of this appendix is to describe how a TEACUP testbed can be built and configured. For more information on the specific hardware and software related to the experiments considered in this thesis, see Section 4.1. For a detailed in-depth technical report on how a TEACUP testbed can be set up, see [102].

A.1.1 VLAN configuration

All the hosts in the testbed are connected to different ports on a Cisco switch. Based on which network interface in the hosts are connected to the switch, different VLANs are configured for the different switch ports (see Figure A.1 for an illustration of the VLANs the network interfaces are

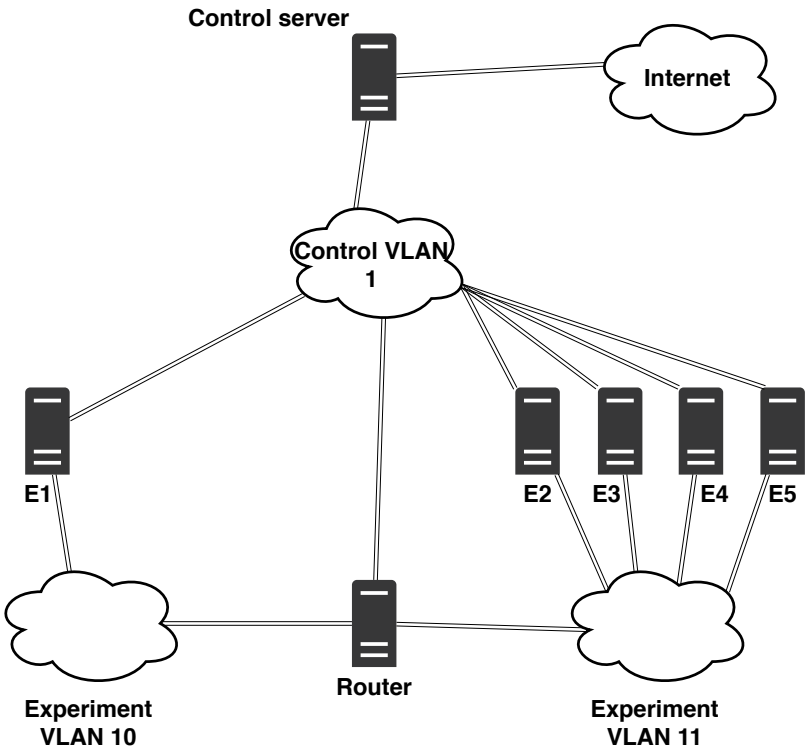


Figure A.1: The TEACUP testbed in the CPS lab at the Department of Informatics, University of Oslo.

connected to). In the switch configuration, the setting `switchport access vlan <VLAN-ID>` is set for the switch ports, where `<VLAN-ID>` is the VLAN number shown in the figure. This setting is not set for VLAN 1 ports because it is the default of the switch. The configuration of the VLANs enables control messages and experiment data to traverse the switch with no interference.

A.1.2 Addressing and routing settings

The following settings are set:

- A DHCP server is installed on the control server to configure the experimental hosts and the router with static IP addresses in the control VLAN network. This centralizes the information in the control server.

- The experimental hosts and router are given hostnames that map to the control network IP addresses configured via DHCP. This enables communicating with the different hosts by name, which is also a requirement to run TEACUP.

- All experimental hosts and routers are configured with static IP addresses in the experimental VLAN networks.

- The experimental hosts set routing rules to route packets destined to the remote experimental VLAN via the router.

A.1.3 Miscellaneous settings

The following settings are set:

- Disable that the hosts can sleep, go to screen lock, etc. because this may disrupt the experiments.
- Automatic updates should be disabled to have control over the software that is used during experiments.

A.2 TEACUP extensions

This section describes the extensions we added to TEACUP to support custom traffic[1] and custom loggers. This includes technical details about TEACUP code. See Section A.3 for an overview of how tu use TEACUP. See [103] for an in-depth description of TEACUP.

A.2.1 Custom traffic

TEACUP supports several types of traffic generators and sinks by default, which are started by calling *tasks* with different parameters that can control the experiment behaviour. Examples of supported tasks include *start_iperf* that starts an *iperf* server and client on the specified machines and send data for a specified time, *start_ping* that starts a ping client, and *start_httperf* that starts an *httperf* client. The tasks that will be run for a specific experiment is specified by the TEACUP config variable `TPCONF_traffic_gens`.

To extend TEACUP with the capability to run custom traffic, we added a new type of task called `start_custom_traffic`. We have specified a rich set of parameters for the task that enables the user to control how TEACUP should control custom applications. We have added the following parameters:

- **name**: The name of the application/command to run.
- **directory**: The directory where the application can be found. If not specified, the application must be available in the locations specified by the *PATH* environment variable.
- **hostname**: The name of the host to run the application.
- **copy_file**: If enabled (1), the application will be copied from the directory specified by *directory* on the control server to */usr/bin/* on the host specified by *hostname*. This is disabled by default.

[1] By *custom traffic* we mean any kind of application that can either send or receive data. The custom traffic extension is so general that any command or application can be controlled by TEACUP.

- **add_prefix**: If enabled (1), adds the current TEACUP experiment test ID[2] as the last argument to the application specified by *name*. This can allow the application to produce logs with filenames similar to those log files produced by TEACUP by default. This is disabled by default.

- **parameters**: The parameters that will be concatenated at the end of the application specified by *name*.

A.2.2 Custom loggers

TEACUP does not support specifying any loggers by default. We therefore added the new TEACUP config variable `TPCONF_custom_loggers` that enables the user to specify custom loggers. This variable expects the same format as `TPCONF_traffic_gens` except that a starting time is not needed for the specified tasks. When specifying traffic generators, there is an option to schedule how long after the experiment has started that the traffic generator will start. When specifying loggers, this option is not needed since all loggers are expected to be started before the experiments begin.

Also, when specifying custom loggers with `TPCONF_custom_loggers` it is required that the task `start_custom_logger` is used. This task has the same parameters available as the custom traffic task `start_custom_traffic`. In addition, it offers the parameter `logname` that enables the user to specify the name that will be appended to the TEACUP test ID when producing log files.

A.3 Example of using TEACUP

TEACUP depends on the Python package *Fabric* which is designed to simplify the process of executing shell commands remotely over *Secure Shell* (SSH) [19]. Once Fabric is installed[3] the command-line tool `fab` will be available. This tool is used to run TEACUP experiments. To run a TEACUP experiment the following should be in place:

1. The TEACUP code is placed in some directory on the control server.

2. A separate directory is created for every TEACUP experiment.

3. The experiment directory contains a TEACUP config file called `config.py` that defines the experiment and points to the TEACUP code.

4. The experiment directory contains a `fabfile.py` file that can be copied directly from the TEACUP code directory. This file is used to initiate the experiments when using the Fabric tool `fab`.

[2] All TEACUP experiments have a test ID that identifies the experiment and the parameters that are set.

[3] The Fabric package can be installed with the Python package manager `pip`.

Once all the above items are in place, the TEACUP experiment can be run by moving into the experiment directory and execute the command `fab run_experiment_multiple`. An example TEACUP config is given in Listing A.1. The example contains comments that describes the config file.

When the TEACUP experiment has completed, the experiment directory will contain a directory with the same name as the TEACUP experiment test ID. This directory contains a variety of log files from the experiment.

Listing A.1: Example TEACUP config file.

```
1  import sys
2  import datetime
3  from fabric.api import env
4
5  # Sets the username and password that Fabric will use when
6  # establishing SSH connections to experimental hosts and
7  # router.
8  env.user = 'root'
9  env.password = 'toor'
10
11 # Sets the shell in which all the TEACUP commands will be
12 # executed in.
13 env.shell = '/bin/sh -c'
14
15 # Sets the path where the TEACUP code can be found.
16 TPCONF_script_path = '/home/teacup/fredhau/git/teacup-code'
17 sys.path.append(TPCONF_script_path)
18
19 # Disables debug log messages
20 TPCONF_debug_level = 0
21
22 # Stores the first 400 bytes of the packets captured with
23 # tcpdump. This is enough to contain the protocol header
24 # information.
25 TPCONF_pcap_snaplen = 400
26
27 # Specifies the hostname of the router machine.
28 TPCONF_router = ['router', ]
29
30 # Specifies the hostnames of the experimental hosts to consider
31 # in the experiments.
32 TPCONF_hosts = [ 'testhost1', 'testhost5', ]
33
34 # Maps the hostname of the router and experimental hosts
35 # to IP addresses in the experimental VLAN networks.
36 TPCONF_host_internal_ip = {
37     'router': ['172.16.10.254', '172.16.11.254'],
38     'testhost1':  ['172.16.10.1'],
39     'testhost5':  ['172.16.11.4'],
40 }
41
42 # Sets the upper limit on the acceptable difference in time
43 # in the experimental hosts. Time synchronization is needed
44 # to analyze captured packets from several hosts. This is
45 # not relevant for this experiment.
46 TPCONF_max_time_diff = 1
47
48 # Sets the TEACUP test ID that identifies every TEACUP
```

```
49 # experiment.
50 now = datetime.datetime.today()
51 TPCONF_test_id = now + '_neatnomultconnectiontcp'
52
53 # Sets the directory on the experimental hosts and router where
54 # TEACUP log data will be stored during experiments.
55 TPCONF_remote_dir = '/tmp/'
56
57 # The number of experiment runs to perform for every
58 # combination of the experiment parameters.
59 TPCONF_runs = 10
60
61 # Sets the network conditions that will be emulated in
62 # the router.
63 # The V_* variables used are are the parameters that are varied
64 # in the experiments. They are described in the bottom of this
65 # config file.
66 TPCONF_router_queues = [
67     ('1', " source='172.16.10.0/24', dest='172.16.11.0/24', "
68           " delay=V_delay, loss=V_loss, rate=V_up_rate, "
69           " queue_disc=V_aqm, queue_size=V_bsize "),
70     ('2', " source='172.16.11.0/24', dest='172.16.10.0/24', "
71           " delay=V_delay, loss=V_loss, rate=V_down_rate, "
72           " queue_disc=V_aqm, queue_size=V_bsize "),
73 ]
74
75 # Specify the custom traffic applications that
76 # will be run in the TEACUP experiments.
77 #
78 # - In this case run a NEAT server and NEAT client.
79 # - The first argument of each tuple below specifies
80 #   the time at which the application will be started.
81 #   The client will be started 2 seconds after the
82 #   server to give the server time to initialize.
83 # - The second argument of the tuples are a
84 #   bookkeeping ID used by TEACUP internally to
85 #   identify the different traffic generators.
86 # - The task "start_custom_traffic" is specified
87 #   for both the NEAT server and NEAT client to
88 #   use the TEACUP extension for custom traffic.
89 # - The details of the options set in the "parameters"
90 #   task option are not relevant for the description of
91 #   the TEACUP config file. However, note that
92 #   V_* variables and literals can easily
93 #   be set as parameters, and Python code can be
94 #   used to manipulate the arguments.
95 traffic_custom = [
96     ('0.0', '1', " start_custom_traffic,"
97      " name='neat_server',"
98      " directory='/usr/home/fredhau/neat-test-suite/build',"
99      " hostname='testhost1',"
100     " duration=V_duration," # ignore this
101     " add_prefix='1',"
102     " parameters='-R %s -A -s -C %s -a %s -b %s -I %s -M %s -p
          %s -v %s -u %s' % ('fredhau/testhost1-freebsd/neat/no
          -mult/connection/tcp', str(int(V_connections) * int(
          V_flows)), V_flows, '0', '172.16.10.1', V_transports,
          12327, 1, 2)"),
103     ('2.0', '2', " start_custom_traffic,"
```

```
104         " name='neat_client',"
105         " directory='/usr/home/fredhau/neat-test-suite/build',"
106         " hostname='testhost5',"
107         " duration=V_duration," # ignore this
108         " add_prefix='1',"
109         " parameters='-R %s -A -s -C %s -a %s -b %s -i %s -l %s -M
                %s -n %s -v %s -u %s %s %s' % ('fredhau/testhost5-
               freebsd/neat/no-mult/connection/tcp', V_flows, V_flows
               , '0', '172.16.11.4', 10240, V_transports, V_flows, 1,
                2, '172.16.10.1', 12327)"),
110 ]
111
112 # This tells TEACUP which traffic generators to use.
113 TPCONF_traffic_gens = traffic_custom
114
115 # Run every experiment for 4 seconds. Since the
116 # experiment considered in this config file is
117 # about connection establishment, the experiments
118 # will not need to run long.
119 TPCONF_duration = 4
120
121 # Use the "newreno"  congestion control algorithm when
122 # using TCP.
123 TPCONF_TCP_algos = ['newreno', ]
124
125 # Enables setting per-host congestion control algorithm.
126 TPCONF_host_TCP_algos = {
127 }
128 TPCONF_host_TCP_algo_params = {
129 }
130
131 # Execute the specified commands in the specified hosts.
132 # These sysctls are described in greater detail in
133 # the "Experimental Setup" chapter.
134 TPCONF_host_init_custom_cmds = {
135 'testhost2' : [ 'sysctl net.inet.tcp.recvbuf_auto=0',
136                 'sysctl net.inet.tcp.sendbuf_auto=0',
137                 'sysctl net.inet.tcp.sendspace=300000',
138                 'sysctl net.inet.tcp.recvspace=300000',
139                 'sysctl net.inet.sctp.sendspace=300000',
140                 'sysctl net.inet.sctp.recvspace=300000',
141                 'sysctl net.inet.tcp.msl=5000' ],
142 'testhost5' : [ 'sysctl net.inet.tcp.recvbuf_auto=0',
143                 'sysctl net.inet.tcp.sendbuf_auto=0',
144                 'sysctl net.inet.tcp.sendspace=300000',
145                 'sysctl net.inet.tcp.recvspace=300000',
146                 'sysctl net.inet.sctp.sendspace=300000',
147                 'sysctl net.inet.sctp.recvspace=300000',
148                 'sysctl net.inet.ip.portrange.randomized=0',
149                 'sysctl net.inet.tcp.msl=5000' ]
150 }
151
152 # Delay packets for 50 ms in the router in both
153 # transfer directions. This gives a total
154 # Round-Trip Time (RTT) of 100 ms.
155 TPCONF_delays = [50]
156
157 # Do not introduce any emulated loss.
158 TPCONF_loss_rates = [0]
```

```
159
160 # Use the same emulated bandwidth for both of the outbound
161 # interfaces in the router.
162 TPCONF_bandwidths = [
163     ('10mbit', '10mbit'),
164 ]
165
166 # Use the "bfifo" Active Queue Management (AQM) algorithm
167 # with a queue size based on the current Bandwidth
168 # Delay Product (BDP) of the experiment. This is
169 # calculated by multiplying the currently set
170 # bandwidth with the Round-Trip Time (RTT) of
171 # the experiment, calculated by multiplying
172 # the value of "TPCONF_delays" with 2.
173 # This queue size ensures that the router queue
174 # is large enough to handle the specified bandwidth.
175 TPCONF_aqms = ['bfifo', ]
176 TPCONF_buffer_sizes = ['bdp']
177
178 # Ignore this, not used for connection experiments.
179 TPCONF_data_sizes = [123]
180
181 # Sets the different numbers of flows to be considered.
182 TPCONF_flows = [1, 2, 4, 8, 16, 32, 64, 128, 256]
183
184 # Sets the transport protocol to be considered.
185 # The second number specifies the number of
186 # connections per flow the server should expect.
187 # This is only relevant when the NEAT client
188 # performs Happy Eyeballs between TCP and SCTP
189 # and the NEAT server listens for both
190 # (in this case, 2 connections are expected
191 # for every flow).
192 TPCONF_transports = [('TCP', '1')]
193
194 # Here the different experiment parameters are
195 # defined. For each parameter, one or more
196 # V_ variables are defined. The V_ variables are
197 # used in the config file to specify a parameter
198 # that may potentially be varied between experiments.
199 # The V_ variable is varied between experiments if
200 # the parameter is specified in "TPCONF_vary_parameters"
201 # (see below).
202 TPCONF_parameter_list = {
203 #   Vary name           V_ variable       file name     values
                      extra vars
204     'delays'        :  (['V_delay'],      ['del'],
        TPCONF_delays,          {}),
205     'loss'          :  (['V_loss'],       ['loss'],
        TPCONF_loss_rates,      {}),
206     'tcpalgos'      :  (['V_tcp_cc_algo'],['tcp'],
        TPCONF_TCP_algos,       {}),
207     'aqms'          :  (['V_aqm'],        ['aqm'],
        TPCONF_aqms,            {}),
208     'bsizes'        :  (['V_bsize'],      ['bs'],
        TPCONF_buffer_sizes,    {}),
209     'runs'          :  (['V_runs'],       ['run'],      range(
        TPCONF_runs),      {}),
210     'bandwidths'    :  (['V_down_rate', 'V_up_rate'], ['down',
```

```
        'up'], TPCONF_bandwidths, {}),
211     'data_sizes'    : (['V_data_size'], ['dsize'],
            TPCONF_data_sizes, {}),
212     'flows'         : (['V_flows'], ['flows'], TPCONF_flows,
            {}),
213     'transports'    : (['V_transports', 'V_connections'], ['
            transports', 'connectionsperflow'], TPCONF_transports,
             {}),
214 }
215
216 # Sets the default values of the V_ variables if
217 # the associated parameter names are not specified
218 # in "TPCONF_vary_parameters"
219 TPCONF_variable_defaults = {
220 #   V_ variable                 value
221     'V_duration'        :       TPCONF_duration,
222     'V_delay'           :       TPCONF_delays[0],
223     'V_loss'            :       TPCONF_loss_rates[0],
224     'V_tcp_cc_algo'     :       TPCONF_TCP_algos[0],
225     'V_down_rate'       :       TPCONF_bandwidths[0][0],
226     'V_up_rate'         :       TPCONF_bandwidths[0][1],
227     'V_aqm'             :       TPCONF_aqms[0],
228     'V_bsize'           :       TPCONF_buffer_sizes[0],
229     'V_data_size'       :       TPCONF_data_sizes[0],
230     'V_flows'           :       TPCONF_flows[0],
231     'V_transports'      :       TPCONF_transports[0][0],
232     'V_connections'     :       TPCONF_transports[0][1],
233 }
234
235 # Specifies the V_ variables that will be varied between
236 # experiments. All combinations of the paramters are
237 # considered.
238 TPCONF_vary_parameters = ['flows', 'data_sizes', 'transports',
        'runs']
```

Appendix B

NEAT evaluation test suite

This appendix describes the structure of the NEAT evaluation test suite available on Github and how to use it [63]. This test suite was used when performing the experiments presented in this thesis. This enables anyone to clone the repository and run the same experiments. For more detailed information about source code, scripts, and config files, refer to the comments in the files or to the README files.

B.1 Download and installation

This section describes how the NEAT evaluation test suite can be downloaded and installed, and describes how to install the dependencies. As described in Chapter 4 the NEAT evaluation test suite source code is installed and tested on FreeBSD. However, we used Ubuntu 16.04.1 LTS when we parsed the results and produced graphs.

Before installing the NEAT evaluation test suite, the NEAT library must be installed:

```
1  # Install NEAT dependencies
2  pkg install cmake libuv ldns jansson swig30
3
4  # Get NEAT source code
5  git clone https://github.com/NEAT-project/neat.git
6
7  # Checkout the version of NEAT used in this thesis
8  cd neat
9  git checkout 2253d7464f33f149d58ba216e7dc99bf1140946f
10
11 # Install NEAT
12 mkdir build
13 cd build
14 cmake ..
15 make install
```

Then install the NEAT evaluation test suite:

```
1  # Get NEAT evaluation test suite source code
2  git clone https://github.com/fredhau/neat-test-suite.git
3
4  # Install it
```

Directory	Description
`teacup-code`	Contains the original TEACUP version 1.0 revision 1364 code with custom traffic and custom logger extensions.
`teacup-configs`	Contains the TEACUP configs used in the experiments.
`src`	Contains the source code for NEAT, libuv, and kqueue clients and servers, in addition to source code for utility functions.
`include`	Contains header files required by the source code in `src`.
`scripts`	Contains scripts to parse the sampled data and produce graphs.

Table B.1: Overview of the NEAT evaluation test suite repository.

```
5 cd neat-test-suite
6 mkdir build
7 cd build
8 cmake ..
9 make
```

To parse and produce graphs of the sampled experiment data both *R* and *gnuplot* are needed. To install them in Ubuntu, run:

```
1  # Install R
2  sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys
       E298A3A825C0D65DFD57CBB651716619E084DAB9
3  sudo add-apt-repository 'deb [arch=amd64,i386] https://cran.
       rstudio.com/bin/linux/ubuntu xenial/'
4  sudo apt-get update
5  sudo apt-get install r-base
6
7  # Install the R package ``gtools''
8  sudo R
9  install.packages(``gtools'')
10 q()
11
12 # Install gnuplot
13 sudo apt-get install gnuplot
```

B.2 Overview

Table B.1 lists the directory structure of the test suite and describes which files are contained in each directory.

B.3 Applications

After following the steps to install the NEAT evaluation test suite as described above, the result is the following applications:

- `neat_server`
- `neat_client`
- `libuv_server`
- `libuv_client`
- `kqueue_server`
- `kqueue_client`

Table B.2 lists the application options that are used in the TEACUP config files in `teacup-configs` and describes what each of the options mean.

The client-side applications have the following synopsis:

```
1 <client_application> <options> <server IP> <server port>
```

The server-side applications have the following synopsis:

```
1 <server_application> <options>
```

B.4 Parsing data

When using the config files specified in `teacup-configs`, all the application log files for all TEACUP experiments will be stored in a hierarchical directory structure on the experimental hosts. These log files are different from the log files that are created by different loggers started with TEACUP by default. These default log files are moved to the control server after every experiment, while this is not the case for the log files produced by the client and server applications.

The `scripts` directory contains different scripts for managing the sampled experiment data in the application log files.

To extract the relevant data from the application log files, run:

```
1 ./parse_all.sh <ROOT-DIR>
```

In this case, `<ROOT-DIR>` is the root directory in which all the application log files are stored in a specific directory hierarchy. The result of this parsing process will be put in the `data` subdirectory of every experiment directory in the directory hierarchy.

To calculate differences between the extracted data, aggregate results, and produce tables that can later be used to produce graphs, run:

```
1 ./calculate_diffs_all.sh <ROOT-DIR>
```

Option	Description
-R <dir>	Store log data produced by the application to directory <dir>.
-A	Expects the last argument when running the application to be the appended string TEACUP test ID + the parameters set in the current experiment.
-s	Enable sampling of resource usage data.
-C <flows>	If on the server-side, expect <flows> connections from the client to be established otherwise report error. If on the client-side, expect <flows> flows to connect (if Happy Eyeballs is used in NEAT, one flow can open several connections).
-D <flows>	Expect <flows> flows to send/receive all HTTP data. Otherwise, report an error.
-a <num>	The number of TCP connections expected to succeed.
-b <num>	The number of SCTP connections expected to suceed.
-h <bytes>	Client-side option. Send <bytes> bytes in a HTTP POST request to the server.
-H <bytes>	On the server-side allocate <bytes> bytes of random data that can be used as a data pool when responding to HTTP GET requests from the client. On the client-side request <bytes> bytes of data from the server by sending a HTTP GET request.
-i <IP>	Client-side option. Only initiate connections from the local network interface with IP address <IP>.
-I <IP>	Server-side option. Only listen to connections on the network interface with the IP address <IP>.
-l <size>	Sets the size of the application receive buffer.
-M <protocol>	Use the transport protocol <protocol>. This can be TCP, SCTP, or SCTP-TCP.
-n <num>	Client-side option. Initiate <num> flows.
-p <port>	Server-side option. The port number to listen to.
-v <level>	NEAT applications only. Sets the log level of the NEAT library.
-u <level>	Sets the log level of user-specified log messages.

Table B.2: Overview of the application options for the NEAT, libuv, and kqueue servers and clients.

This script creates tables that describes resource usage during connection establishment and data transfer. The tables will be put in the `tables` subdirectory of every experiment directory in the directory hierarchy.

From the tables created with `calculate_diffs_all.sh`, graphs can be produced. To produce graphs of the results, run:

```
1 ./produce_graphs_all.sh <ROOT-DIR>
```

The graphs will be put in the `graphs` subdirectory in the `scripts` directory.

Appendix C

Programming with the NEAT API and the BSD sockets API

This appendix serves as a guide and reference on how to develop applications with the BSD sockets API and the NEAT API. The purpose of this reference is that the reader can easily lookup specific API functions when they are mentioned in the thesis text.

C.1 Programming with the BSD sockets API

This section describes the BSD sockets API. First, a description of some of the core functions of the API relevant for this thesis is given. Finally, server and client examples are provided.

C.1.1 API

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

Creates a new socket descriptor that can be used for communication. `domain` specifies the network domain in which the socket should operate in: `AF_INET` creates an IPv4 socket, `AF_INET6` creates an IPv6 socket, `AF_UNIX` creates a UNIX domain socket used for inter-process communication, etc. `type` specifies the network service of the socket: `SOCK_STREAM` means a stream-oriented, reliable, connection-oriented service, while `SOCK_DGRAM` means a message-oriented, unreliable, connectionless service. `protocol` is the actual transport protocol that will be associated with the socket: `IPPROTO_TCP` means TCP, `IPPROTO_UDP` means UDP, `IPPROTO_SCTP` means SCTP, etc.

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen)
```

Binds the socket descriptor `sockfd` to the local address specified by the `addr` structure. For IPv4 this address consists of the IP address of the interface to bind to in addition to the port number that will be associated with the application so that the operating system can forward received data to the

correct application. It is also possible to bind to all interfaces by specifying the address `INADDR_ANY`.

`int listen(int sockfd, int backlog)`

Enable the socket `sockfd` to listen for incoming connections. `backlog` specifies the maximum length of the queue containing incoming connection requests.

`int accept(int sockfd, struct sockaddr *addr, socklen_t addrlen)`

Block until the socket `sockfd` has incoming connections to be processed. If `sockfd` is in non-blocking mode, `accept` will not block (see Section 2.2.3). The address of the remote peer will be populated into the given `addr` structure.

`int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen)`

Sends a connection request through socket `sockfd` to the remote peer with the address specified in the `addr` structure.

`ssize_t send(int sockfd, const void *buf, size_t len, int flags)`

Attempts to send `len` bytes of data found in buffer `buf` to the socket `sockfd`. If more advanced functionality is required when sending data, e.g. ancillary control data, `sendmsg` can be used instead. The return value is the number of bytes actually sent.

`ssize_t recv(int sockfd, void *buf, size_t count, int flags)`

Attempts to read `count` bytes from socket `sockfd` and put them into the `buf` buffer. If more advanced functionality is required when handling data, e.g. ancillary control data, `recvmsg` can be used instead. The return value is the number of bytes actually read.

`int close(int fd)`

Releases all resources associated with the socket `fd`. Can either perform a graceful or abortive close based on whether the `SO_LINGER` option is enabled.

C.1.2 Examples

Listing C.1 illustrates how the core functions of the BSD sockets API can be leveraged to implement a simple server. The specific details of the event loop is omitted as they are not related to the BSD sockets API. Details about non-blocking I/O and asynchronous I/O can be found in Section 2.2.3. Listing C.2 illustrates the code for a simple client that connects to a server.

Listing C.1: Code example showing the core functions in a server implemented with the BSD sockets API

```
1  struct sockaddr_in server_addr;
2  struct sockaddr_storage client_addr;
3  socklen_t socklen = sizeof (client_addr);
4  int server_socket;
5
6  /* Create new TCP socket that will be used over IPv4 */
7  server_socket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
8
9  /* Specify the local address and port information.
10    Here we bind to local address "192.168.10.1", port 12327 */
11 server_addr.sin_family = AF_INET;
12 server_addr.sin_port = htons(12327);
13 inet_pton(AF_INET, "192.168.10.1", &server_addr.sin_addr);
14
15 /* Bind the server socket to the local address */
16 bind(server_socket, (struct sockaddr *)&serv_addr, sizeof(serv_addr));
17
18 /* Start listening on incoming connections */
19 listen(server_socket, SOMAXCONN);
20
21 /* Some kind of event loop */
22 for (;;) {
23     ...
24     /* When an incoming connection request is available, call accept */
25     accept(server_socket, (struct sockaddr *)&client_addr, &socklen);
26     ...
27 }
```

The code also illustrates how a socket can be set non-blocking through the sockets API.

C.2 Programming with the NEAT API

This section describes the NEAT API. First, an overview of the NEAT library programming model is given. Second, a description of some of the core functions of the NEAT API relevant for this thesis is given. Finally, an example of a NEAT server is given, and it is shown how the server code can easily be modified to make the application a client.

C.2.1 Overview

As described in Section 2.2.3, NEAT provides a *callback-based* API administred by *libuv* internally. When setting callbacks in NEAT, the application must modify a pre-defined set of function variables that are associated with every *NEAT flow*, see Table C.1. These callbacks are called from within the *NEAT event loop* when specific events occur for the *NEAT flows*. This is almost similar to how *libuv* works, but when programming with *libuv*, callbacks must be set for a specific *event source*, e.g. a socket

Listing C.2: Code example showing the core functions in a client implemented with the BSD sockets API

```
1  struct sockaddr_in server_addr;
2  int client_socket;
3  int flags;
4
5  /* Create new TCP socket that will be used over IPv4 */
6  client_socket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
7
8  /* Specify the remote address and port information for the server */
9  server_addr.sin_family = AF_INET;
10 server_addr.sin_port = htons(12327);
11 inet_pton(AF_INET, "192.168.10.1", &server_addr.sin_addr);
12
13 /* Make socket non-blocking */
14 flags = fcntl(client_socket, F_GETFL, 0);
15 flags |= O_NONBLOCK;
16 fcntl(client_socket, F_SETFL, flags);
17
18 /* Send a connection request to the remote server */
19 connect(client_socket, (struct sockaddr *)&server_addr, sizeof (server_addr));
20
21 /* Some kind of event loop */
22 for (;;) {
23     ...
24 }
```

descriptor, timer, signal, etc., and only one callback can be set for every *event source*. It is the responsibility of the application to check exactly which event occured for the *event source*. NEAT will do this checking internally, and call the appropriate callback registered for each specific event. Note that the internal *libuv* loop handle is accessible through the NEAT function `neat_get_event_loop`. This enables the programmer to add additional events to the event loop, like signal handling. It also enables the programmer to link NEAT and other libraries in the same event loop.

A central entity when programming with the *NEAT API* is the concept of a *NEAT context*. This is a data structure that maintains all the global information about the current NEAT session, and for most situations, only a single *NEAT context* is needed for each application. Associated with each *NEAT context* is a *libuv* event loop handle, and a list of created *NEAT flows*. It also contains a list of available source addresses that is continuously updated. A *NEAT flow* must belong to a single *NEAT context*, and it cannot be moved to another *NEAT context* later.

Table C.2 lists the core functions of the *NEAT API*. Below the API is described in greater detail for all the functions. For a detailed description of the *NEAT API*, see Appendix B in [43].

Callback	Description
`on_connected`	The *NEAT Flow* successfully connects or a remote peer has connected.
`on_error`	An error has occured.
`on_readable`	The *NEAT Flow* is readable.
`on_writable`	The *NEAT Flow* is writable.
`on_all_written`	All the data that is buffered in the *NEAT Flow* is successfully sent.
`on_aborted`	The *NEAT Flow* is aborted.
`on_close`	The remote peer closes the connection or the application closes the connection associated with the *NEAT Flow*.

Table C.1: The core set of callback functions that can be set through the NEAT API

Function	Description
`neat_init_ctx`	Creates a new *NEAT context* (one per event loop).
`neat_new_flow`	Creates a new *NEAT Flow Endpoint*.
`neat_set_property`	Sets the user-specified *JSON properties* for a specified *NEAT Flow*.
`neat_set_operations`	Sets the user-specified *callbacks* for a specified *NEAT Flow*.
`neat_start_event_loop`	Starts the *NEAT event loop* running *libuv* internally.
`neat_get_event_loop`	Returns the *libuv* handle used internally in the *NEAT event loop*.
`neat_open`	Connects to the specified remote peer using the specified *NEAT Flow*.
`neat_accept`	Accepts connection requests using the specified *NEAT Flow*.
`neat_read`	Reads data from the specified *NEAT Flow*.
`neat_write`	Writes data to the specified *NEAT Flow*.
`neat_close`	Closes the specified *NEAT Flow*.
`neat_stop_event_loop`	Stops the *NEAT event loop*.
`neat_free_ctx`	Releases all the resources associated with the specified *NEAT context*.

Table C.2: The core functions of the NEAT API.

C.2.2 API

`struct neat_ctx *neat_init_ctx(void)`

Initializes the *libuv* event loop, the global data structures, and performs platform-specific operations related to maintaining the list of source addresses. The return value is the allocated and initialized *NEAT context*.

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

Allocates and initializes the data structures required to represent a *NEAT Flow Endpoint*. One of the elements of this data structure is the internal platform- and protocol-independent representation of a *network socket* in NEAT (`struct neat_pollable_socket`). Note that this function does not actually open any *socket descriptor*. The *flow* is added to the *NEAT context* and returned as return value.

`neat_error_code neat_set_property(struct neat_ctx *ctx, struct neat_flow *flow, const char *properties)`

Associates the *NEAT properties* specified in the `properties` string to the specified *NEAT flow* `flow`. An example of the format of the *NEAT properties* is given in Listing C.3. The properties is given in a *JSON* format, and the function converts the properties given in the string to actual *JSON objects* using the *libjansson* library [38]. Every property consists of a *key-value* pair, where the *value* consists of a list of other *key-value* pairs that defines the property. One of the fields is *value* which denotes the value of the property. Another field is *precedence* which denotes whether the property is *optional* (1) or *mandatory* (2). *Mandatory* properties constitute the strict requirements of the application that cannot be broken. *optional* properties consitute the *desired* network services, and will be more prioritized by the *Policy* components, but may not be used as the final end-to-end *transport solution*

Listing C.3: NEAT properties example showing the JSON format

```
1  {
2      "transport": {
3          "value": "reliable",
4          "precedence": 2
5      },
6      "multihoming": {
7          "value": true,
8          "precedence": 1
9      }
10 }
```

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

Sets the callbacks for the *NEAT flow* `flow`. The callbacks are specified in the `ops` structure. An example of how the callbacks can be set is given in

Listing C.5.

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

Starts the *NEAT event loop*. It works as an abstraction layer over `uv_run` that starts the *libuv* event loop. The *run_mode* specifies how the event loop shall operate. `NEAT_RUN_DEFAULT` means that the event loop will continue to run until there are no more active and referenced handles in the loop. The user will need to call `neat_stop_event_loop` in order to stop the execution of the loop. `NEAT_RUN_ONCE` means that the event loop will run only one iteration, but the event loop may block if there are no pending callbacks. `NEAT_RUN_NOWAIT` is equal to `NEAT_RUN_ONCE`, but the event loop will not block. This can be useful if the *NEAT event loop* should be integrated with another event loop. In this case, the event loops can run in tandem. An example of this is given in Listing C.4.

Listing C.4: Code example of a NEAT application running in tandem with another event loop

```
1 for (;;) {
2     if (event_loop_closed) {
3         break;
4     }
5
6     /* NEAT will run in tandem with another event loop */
7     neat_start_event_loop(ctx, NEAT_RUN_NOWAIT);
8     some_other_event_loop(...);
9 }
```

`uv_loop_t *neat_get_event_loop(struct neat_ctx *ctx)`

Returns a handle to the *libuv* loop run internally in the *NEAT event loop* associated with the *NEAT context* `ctx`.

`neat_error_code neat_open(struct neat_ctx *ctx, struct neat_flow *flow, const char *name, uint16_t port, struct neat_tlv optional[], unsigned int opt_count)`

Attempts to connect to the remote peer specified by the domain name or address `name` and port `port`. It is required that the properties for the flow is specified through `neat_set_property` before calling this function. This function does not actually initiate a connection request, but the request is added to the queue of requests that needs to be *resolved*. The *resolver* is running in the *NEAT event loop*. If `name` is a *literal* address, the address does not need to be *resolved*, and the connection request will be made as soon as possible within the event loop. Optional options can also be specified through the `optional` array. `opt_count` is the length of this array. Options include *priority*, *flow group*, *stream count*, etc.

```
neat_error_code neat_accept(struct neat_ctx *ctx, struct neat_flow
*flow, uint16_t port, struct neat_tlv optional[], unsigned int opt_count)
```

Signals to the *NEAT system* that the *NEAT flow* `flow` will be used as a listening flow listening on port `port`. Based on the properties specified for the flow, several *listening sockets* can be created for various transport protocols. Note that this function does not actually accept a new incoming connection like the BSD sockets API function `accept`. Instead, NEAT will accept new incoming connections internally whenever a *listening socket* is *readable*, create a new *NEAT flow*, and call the `on_connected` callback for that flow. This function supports optional options, including the possibility to specify on which local addresses to listen for incoming connections.

```
neat_error_code neat_read(struct neat_ctx *ctx, struct neat_flow
*flow, unsigned char *buffer, uint32_t amt, uint32_t *actualAmt,
struct neat_tlv optional[], unsigned int opt_count)
```

Attempts to read `amt` bytes from the *NEAT flow* `flow` and place them in the buffer `buffer`. The actual number of bytes read is stored in `actualAmt`. This function should be called within the `on_readable` callback that signals the application that the flow is *readable*. The optional options array that may optionally be passed as an argument will be filled with extra information that may be interesting to the application, e.g. the *stream number* for SCTP.

```
neat_error_code neat_write(struct neat_ctx *ctx, struct neat_flow
*flow, const unsigned char *buffer, uint32_t amt, struct neat_tlv
optional[], unsigned int opt_count)
```

Attempts to buffer `amt` number of bytes specified in the buffer `buffer` for transfer on the *NEAT flow* `flow`. Note that unlike the BSD sockets API function `send` that returns the number of bytes sent, this function simply buffers all of the specified data for potential later transmission. The rationale for this is to enable message-oriented transport protocols like SCTP and UDP to specify entire messages to be sent. The `on_writable` callback will only be called if there are no buffered data left; otherwise, NEAT will internally try to send the buffered data. Through the optional option array, the user may e.g. specify which *SCTP stream* that data should be sent on.

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

Closes the *NEAT flow*, closing all socket descriptors associated with it and releasing the resources. The `on_close` callback will be called as a result, where the user can free any application-allocated resources associated with the *NEAT flow*.

```
void neat_stop_event_loop(struct neat_ctx *ctx)
```

Provides an abstraction layer over the *libuv* function `uv_stop` that stops the event loop. The *NEAT event loop* will complete the current iteration of the event loop, and then the application will return from `neat_start_event_loop`.

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

Releases all the resources associated with the *NEAT context*. In particular, all *NEAT flows* that has not yet been closed are released. Deallocation callbacks for the remaining open handles in the event loop is queued for execution. The event loop will need to be run again so that these callbacks can be called, and all resources deallocated for the event loop.

C.2.3 Examples

Listing C.5 gives an example of a simple NEAT application that leverages the core functions of the *NEAT API* to implement a server that is listening for incoming connection requests. The example shows how the NEAT properties can be specified to *force* the use of specific transport protocols. The application will listen to both TCP and SCTP connections from the clients. If the function `neat_accept` is changed to `neat_open`, the example will instead show a simple NEAT client application that performs *Happy Eyeballs* between SCTP and TCP.

Listing C.5: Code example of a NEAT server application

```
/* Need to include this to access the NEAT User API */
#include <neat.h>

/* User-specified NEAT properties. In this example we
   force the application to listen to both SCTP and TCP */
static char *properties = "\
{\
    \"transport\": {\
        \"value\": [\"SCTP\", \"TCP\"],\
        \"precedence\": 2\
    }\
}";

/* Callback that is called when receiving incoming
   connection requests. */
static neat_error_code
on_connected(struct neat_flow_operations *ops)
{
    /* Set callbacks for the newly created NEAT Flow */
    ops->on_writable = on_writable;
    ops->on_readable = on_readable;
    neat_set_operations(ops->ctx, ops->flow, ops);
    return NEAT_OK;
}

int
main(int argc, char *argv[])
{
    struct neat_ctx *ctx;
    struct neat_flow *flow;
    struct neat_flow_operations ops;
    memset(&ops, 0, sizeof (ops));

    /* Create NEAT context. */
    ctx = neat_init_ctx();

    /* Create a NEAT flow (will be used for listening). */
    flow = neat_new_flow(ctx);

    /* Specify the properties to associate with the flow. */
    neat_set_property(ctx, flow, properties);

    /* Set some callback functions for the flow. */
    ops.on_connected = on_connected;
    ops.on_error = on_error;
    neat_set_operations(ctx, flow, &ops);

    /* Listen on port 8080 for incoming connections */
    neat_accept(ctx, flow, 8080, NULL, 0);

    /* Start the NEAT event loop */
    neat_start_event_loop(ctx, NEAT_RUN_DEFAULT);

    /* Release resources associated with the NEAT context */
    neat_free_ctx(ctx);
}
```

Bibliography

[1] *A portable SCTP userland stack (Github repository)*. URL: https://github.com/sctplab/usrsctp.

[2] Emile Aben. *Hampering Eyeballs - Observations on Two "Happy Eyeballs" Implementations*. RIPE NCC. Nov. 2011. URL: https://labs.ripe.net/Members/emileaben/hampered-eyeballs.

[3] *An Introduction to libuv*. URL: https://nikhilm.github.io/uvbook/introduction.html (visited on 22/04/2018).

[4] *Analyzing UDP usage in Internet traffic*. URL: https://www.caida.org/research/traffic-analysis/tcpudpratio (visited on 26/02/2018).

[5] *Architecture, interface, and implementation drafts for the definition of an abstract API for IETF TAPS (github public repository)*. URL: https://github.com/taps-api/drafts (visited on 08/05/2018).

[6] Vaggelis Atlidakis et al. 'POSIX Abstractions in Modern Operating Systems: The Old, the New, and the Missing'. In: *Proceedings of the Eleventh European Conference on Computer Systems*. EuroSys '16. London, United Kingdom: ACM, 2016, 19:1–19:17. ISBN: 978-1-4503-4240-7. DOI: 10.1145/2901318.2901350. URL: http://doi.acm.org/10.1145/2901318.2901350.

[7] *Augmented socket interface for an application to express knowledge about its communication*. URL: https://github.com/fg-inet/socket-intents (visited on 08/05/2018).

[8] Vaibhav Bajpai and Jürgen Schönwälder. 'Measuring the Effects of Happy Eyeballs'. In: *Proceedings of the 2016 Applied Networking Research Workshop*. ANRW '16. Berlin, Germany: ACM, 2016, pp. 38–44. ISBN: 978-1-4503-4443-2. DOI: 10.1145/2959424.2959429. URL: http://doi.acm.org/10.1145/2959424.2959429.

[RFC6556] Fred Baker. *Testing Eyeball Happiness*. RFC 6556. RFC Editor, Apr. 2012, pp. 1–10. URL: https://rfc-editor.org/rfc/rfc6556.txt.

[9] A. Bergkvist et al. *WebRTC 1.0: Real-time Communication Between Browsers*. W3C Working Draft. Nov. 2017. URL: http://www.w3.org/TR/webrtc/.

[10] *Blocking I/O, Nonblocking I/O, And Epoll*. URL: https://eklitzke.org/blocking-io-nonblocking-io-and-epoll (visited on 25/04/2018).

[RFC7323] David Borman et al. *TCP Extensions for High Performance*. RFC 7323. RFC Editor, Sept. 2014, pp. 1–49. URL: https://rfc-editor.org/rfc/rfc7323.txt.

[RFC7049] Carsten Bormann and Paul E. Hoffman. *Concise Binary Object Representation (CBOR)*. RFC 7049. RFC Editor, Oct. 2013, pp. 1–54. URL: https://rfc-editor.org/rfc/rfc7049.txt.

[11] Z. Bozakov et al. *Validation and evaluation results*. Deliverable D4.3. The NEAT Project (H2020-ICT-05-2014), Apr. 2018.

[12] *Callgrind: a call-graph generating cache and branch prediction profiler*. URL: http://valgrind.org/docs/manual/cl-manual.html (visited on 21/05/2018).

[RFC1958] Brian E. Carpenter. *Architectural Principles of the Internet*. RFC 1958. RFC Editor, June 1996, pp. 1–8. URL: https://rfc-editor.org/rfc/rfc1958.txt.

[RFC675] V. Cerf, Y. Dalal and C. Sunshine. *Specification of Internet Transmission Control Program*. RFC 675 (Historic). Obsoleted by RFC 7805. Internet Engineering Task Force, Dec. 1974. URL: http://www.ietf.org/rfc/rfc675.txt.

[RFC7413] Yuchung Cheng et al. *TCP Fast Open*. RFC 7413. RFC Editor, Dec. 2014, pp. 1–26. URL: https://rfc-editor.org/rfc/rfc7413.txt.

[13] S. Cheshire, J. Graessley and R. McGuire. *Encapsulation of TCP and other Transport Protocols over UDP*. Internet-Draft draft-cheshire-tcp-over-udp-00. Internet Engineering Task Force, Jan. 2014. URL: https://tools.ietf.org/html/draft-cheshire-tcp-over-udp-00.

[14] D. Clark. 'The Design Philosophy of the DARPA Internet Protocols'. In: *SIGCOMM Comput. Commun. Rev.* 18.4 (Aug. 1988), pp. 106–114. ISSN: 0146-4833. DOI: 10.1145/52325.52336. URL: http://doi.acm.org/10.1145/52325.52336.

[RFC8200] Dr. Steve E. Deering and Robert M. Hinden. *Internet Protocol, Version 6 (IPv6) Specification*. RFC 8200. RFC Editor, July 2017, pp. 1–42. URL: https://rfc-editor.org/rfc/rfc8200.txt.

[15] ECMA. *The JSON Data Interchange Format*. 2013. URL: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf.

[16] Korian Edeline et al. 'Using UDP for Internet Transport Evolution'. In: *CoRR* abs/1612.07816 (2016). arXiv: 1612.07816. URL: http://arxiv.org/abs/1612.07816.

[RFC793] Jon Postel (editor). *Transmission Control Protocol*. RFC 793. RFC Editor, Sept. 1981, pp. 1–91. URL: https://rfc-editor.org/rfc/rfc793.txt.

[RFC8085] Lars Eggert, Gorry Fairhurst and Greg Shepherd. *UDP Usage Guidelines*. RFC 8085. RFC Editor, Mar. 2017, pp. 1–55. URL: https://rfc-editor.org/rfc/rfc8085.txt.

[17] *epoll(7) – Linux manpage*. URL: http://man7.org/linux/man-pages/man7/epoll.7.html (visited on 19/04/2018).

[18] *Event Ports – Solaris event handling*. URL: https://docs.oracle.com/cd/E36784_01/html/E36874/port-create-3c.html (visited on 20/04/2018).

[19] *Fabric – Pythonic remote execution*. URL: http://www.fabfile.org/ (visited on 03/06/2018).

[RFC7231] Roy T. Fielding and Julian Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. RFC 7231. RFC Editor, June 2014, pp. 1–101. URL: https://rfc-editor.org/rfc/rfc7231.txt.

[RFC4340] Sally Floyd, Mark J. Handley and Eddie Kohler. *Datagram Congestion Control Protocol (DCCP)*. RFC 4340. RFC Editor, Mar. 2006, pp. 1–129. URL: https://rfc-editor.org/rfc/rfc4340.txt.

[RFC6824] Alan Ford et al. *TCP Extensions for Multipath Operation with Multiple Addresses*. RFC 6824. RFC Editor, Jan. 2013, pp. 1–64. URL: https://rfc-editor.org/rfc/rfc6824.txt.

[20] Bryan Ford and Janardhan R. Iyengar. 'Efficient Cross-Layer Negotiation'. In: *Eight ACM Workshop on Hot Topics in Networks (HotNets-VIII), HOTNETS '09, New York City, NY, USA, October 22-23, 2009*. 2009. URL: http://conferences.sigcomm.org/hotnets/2009/papers/hotnets2009-final123.pdf.

[21] *FreeBSD implementation of Coupled Congestion Control for TCP (gitlab public repository)*. URL: https://gitlab.com/kristahi/freebsd (visited on 06/05/2018).

[RFC3493] Robert E. Gilligan et al. *Basic Socket Interface Extensions for IPv6*. RFC 3493. RFC Editor, Mar. 2003, pp. 1–39. URL: https://rfc-editor.org/rfc/rfc3493.txt.

[22] *gnuplot (homepage)*. URL: http://www.gnuplot.info/ (visited on 22/05/2018).

[23] K. J. Grinnemo, T. Andersson and A. Brunstrom. 'Performance Benefits of Avoiding Head-of-Line Blocking in SCTP'. In: *Joint International Conference on Autonomic and Autonomous Systems and International Conference on Networking and Services - (icas-isns'05)*. Oct. 2005, pp. 44–44. DOI: 10.1109/ICAS-ICNS.2005.73.

[24] Karl-Johan Grinnemo et al. *Final Report on Transport Protocol Enhancements*. Deliverable D3.2. NEAT Project (H2020-ICT-05-2014), Feb. 2017.

[25] Karl-Johan Grinnemo et al. *Happy Eyeballs for Transport Selection*. Internet-Draft draft-grinnemo-taps-he-03. Work in Progress. Internet Engineering Task Force, July 2017. 10 pp. URL: https://datatracker.ietf.org/doc/html/draft-grinnemo-taps-he-03.

[RFC2782] Arnt Gulbrandsen and Dr. Levon Esibov. *A DNS RR for specifying the location of services (DNS SRV)*. RFC 2782. RFC Editor, Feb. 2000, pp. 1–12. URL: https://rfc-editor.org/rfc/rfc2782.txt.

[26] Sangtae Ha, Injong Rhee and Lisong Xu. 'CUBIC: A New TCP-friendly High-speed TCP Variant'. In: *SIGOPS Oper. Syst. Rev.* 42.5 (July 2008), pp. 64–74. ISSN: 0163-5980. DOI: 10.1145/1400097.1400105. URL: http://doi.acm.org/10.1145/1400097.1400105.

[27] M. Handley. 'Why the Internet Only Just Works'. In: *BT Technology Journal* 24.3 (July 2006), pp. 119–129. ISSN: 1358-3948. DOI: 10.1007/s10550-006-0084-z. URL: http://dx.doi.org/10.1007/s10550-006-0084-z.

[28] David A. Hayes, Jason But and Grenville Armitage. 'Issues with Network Address Translation for SCTP'. In: *SIGCOMM Comput. Commun. Rev.* 39.1 (Dec. 2008), pp. 23–33. ISSN: 0146-4833. DOI: 10.1145/1496091.1496095. URL: http://doi.acm.org/10.1145/1496091.1496095.

[29] D. Henrici and B. Reuther. 'Service-oriented Protocol Interfaces and Dynamic Intermediation of Communication Services'. In: *Proceedings of the 2nd IASTED International Conference on Communications, Internet and Information Technology (CIIT)*. Scottsdale (AZ), USA, Nov. 2003.

[30] *How fast should HZ be?* URL: https://lwn.net/Articles/145973/ (visited on 17/05/2018).

[31] *httperf (SourceForge)*. URL: https://sourceforge.net/projects/httperf/ (visited on 11/05/2018).

[32] Geoff Huston. *Bemused Eyeballs: Tailoring Dual Stack Applications for a CGN Environment*. May 2012. URL: https://labs.apnic.net/?p=188.

[33] 'IEEE Standard for Information Technology- Portable Operating System Interface (POSIX) Base Specifications, Issue 7'. In: *IEEE Std 1003.1-2008 (Revision of IEEE Std 1003.1-2004)* (Dec. 2008), pp. c1–3826. DOI: 10.1109/IEEESTD.2008.4694976.

[34] 'IEEE Standard for Information Technology - Portable Operating System Interface (POSIX(R))'. In: *IEEE Std 1003.1, 2004 Edition The Open Group Technical Standard. Base Specifications, Issue 6. Includes IEEE Std 1003.1-2001, IEEE Std 1003.1-2001/Cor 1-2002 and IEEE Std 1003.1-2001/Cor 2-2004. Shell* (Dec. 2008), pp. 1–3874. DOI: 10.1109/IEEESTD.2008.7394902.

[35] *IETF. Transport Services (taps) Working Group*. URL: https://datatracker.ietf.org/wg/taps/about/ (visited on 15/04/2018).

[36] *IOCP – I/O Completion Ports (Windows)*. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/aa365198(v=vs.85).aspx (visited on 20/04/2018).

[37] Jana Iyengar and Martin Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. Internet-Draft draft-ietf-quic-transport-10. Work in Progress. Internet Engineering Task Force, Mar. 2018. 101 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-quic-transport-10.

[38] *Jansson – C library for working with JSON data*. URL: www.digip.org/jansson/ (visited on 07/05/2018).

[RFC3828] Lars-Erik Jonsson et al. *The Lightweight User Datagram Protocol (UDP-Lite)*. RFC 3828. RFC Editor, July 2004, pp. 1–12. URL: https://rfc-editor.org/rfc/rfc3828.txt.

[39] Stefan Jorer. 'A Protocol-Independent Internet Transport API'. MA thesis. University of Innsbruck, Dec. 2010.

[40] *Julia – Dynamic programming language for numerical computing*. URL: https://julialang.org/ (visited on 19/04/2018).

[41] Rishi Kapoor et al. 'Bullet Trains: A Study of NIC Burst Behavior at Microsecond Timescales'. In: *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*. CoNEXT '13. Santa Barbara, California, USA: ACM, 2013, pp. 133–138. ISBN: 978-1-4503-2101-3. DOI: 10.1145/2535372.2535407. URL: http://doi.acm.org/10.1145/2535372.2535407.

[42] Dragana Damjanovic Kashif Munir Michael Welzl. 'Linux beats Windows! - or the Worrying Evolution of TCP in Common Operating Systems'. In: *Proceedings of the International Workshop on Protocols for Fast Long-Distance Networks (PFLDnet '07)*. Marina Del Rey (Los Angeles), California, USA: ENS Lyon, Feb. 2007, pp. 43–48. URL: http://www.welzl.at/research/publications/pfldnet2007.pdf.

[43] Naeem Khademi et al. *Final Version of Core Transport System*. Deliverable D2.3. NEAT Project (H2020-ICT-05-2014), Aug. 2017. URL: https://www.neat-project.org/publications/.

[44] N. Khademi et al. 'NEAT: A Platform- and Protocol-Independent Internet Transport API'. In: *IEEE Communications Magazine* 55.6 (2017), pp. 46–54. ISSN: 0163-6804. DOI: 10.1109/MCOM.2017.1601052.

[45] *kqueue(2) – FreeBSD manpage*. URL: https://www.freebsd.org/cgi/man.cgi?query=kqueue&sektion=2 (visited on 20/04/2018).

[46] D. Kreutz et al. 'Software-Defined Networking: A Comprehensive Survey'. In: *Proceedings of the IEEE* 103.1 (Jan. 2015), pp. 14–76. ISSN: 0018-9219. DOI: 10.1109/JPROC.2014.2371999.

[47] Jonathan Kua and Grenville Armitage. *Generating Dynamic Adaptive Streaming over HTTP Traffic Flows with TEACUP Testbed*. Tech. rep. 161216A. Melbourne, Australia: Centre for Advanced Internet Architectures, Swinburne University of Technology, 16 December 2016. URL: http://caia.swin.edu.au/reports/161216A/CAIA-TR-161216A.pdf.

[48] *libev (homepage)*. URL: http://software.schmorp.de/pkg/libev.html (visited on 22/04/2018).

[49] *libevent – an event notification library*. URL: http://libevent.org/ (visited on 22/04/2018).

[50] *libuv – Cross-platform asynchronous I/O*. URL: https://libuv.org/ (visited on 19/04/2018).

[51] *libuv (Github public repository)*. URL: https://github.com/libuv/libuv (visited on 19/04/2018).

[52] *Lighttpd – fly light*. URL: https://www.lighttpd.net/ (visited on 11/05/2018).

[53] *Linux Performance*. URL: http://www.brendangregg.com/linuxperf.html (visited on 17/05/2018).

[54] Igor Lubashev. *Partially Reliable Streams for QUIC*. Internet-Draft draft-lubashev-quic-partial-reliability-01. Work in Progress. Internet Engineering Task Force, Jan. 2018. 8 pp. URL: https://datatracker.ietf.org/doc/html/draft-lubashev-quic-partial-reliability-01.

[55] *Luvit – Asynchronous I/O for Lua*. URL: https://luvit.io/ (visited on 19/04/2018).

[56] D. Martin, H. Wippel and H. Backhaus. 'A future-proof application-to-network interface'. In: *2011 International Conference on the Network of the Future*. Nov. 2011, pp. 20–24. DOI: 10.1109/NOF.2011.6126676.

[RFC5905] Jim Martin et al. *Network Time Protocol Version 4: Protocol and Algorithms Specification*. RFC 5905. RFC Editor, June 2010, pp. 1–110. URL: https://rfc-editor.org/rfc/rfc5905.txt.

[57] *Massif: a heap profiler*. URL: http://valgrind.org/docs/manual/ms-manual.html (visited on 22/05/2018).

[RFC5766] Philip Matthews, Jonathan Rosenberg and Rohan Mahy. *Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)*. RFC 5766. RFC Editor, Apr. 2010, pp. 1–67. URL: https://rfc-editor.org/rfc/rfc5766.txt.

[RFC5389] Philip Matthews et al. *Session Traversal Utilities for NAT (STUN)*. RFC 5389. RFC Editor, Oct. 2008, pp. 1–51. URL: https://rfc-editor.org/rfc/rfc5389.txt.

[58] Microsoft. *Windows Sockets 2*. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/ms740673(v=vs.85).aspx (visited on 26/02/2018).

[RFC1035] P. Mockapetris. *Domain names - implementation and specification*. RFC 1035. RFC Editor, Nov. 1987, pp. 1–55. URL: https://rfc-editor.org/rfc/rfc1035.txt.

[RFC896] John Nagle. *Congestion Control in IP/TCP Internetworks*. RFC 896. RFC Editor, Jan. 1984, pp. 1–9. URL: https://rfc-editor.org/rfc/rfc896.txt.

[59] P. Natarajan. *Leveraging Innovative Transport Layer Services for Improved Application Performance*. University of Delaware, 2009. URL: https://books.google.no/books?id=MT1tswEACAAJ.

[60] Preethi Natarajan, Paul D. Amer and Randall Stewart. 'Multistreamed Web Transport for Developing Regions'. In: *Proceedings of the Second ACM SIGCOMM Workshop on Networked Systems for Developing Regions*. NSDR '08. Seattle, WA, USA: ACM, 2008, pp. 43–48. ISBN: 978-1-60558-180-4. DOI: 10.1145/1397705.1397717. URL: http://doi.acm.org/10.1145/1397705.1397717.

[61] *NEAT – A New, Evolutive API and Transport-Layer Architecture for the Internet*. URL: https://neat-project.org (visited on 15/04/2018).

[62] *NEAT documentation (readthedocs)*. URL: https://neat.readthedocs.io/en/latest/ (visited on 08/05/2018).

[63] *NEAT evaluation test suite (Github public repository)*. URL: https://github.com/fredhau/neat-test-suite (visited on 03/06/2018).

[64] *NEAT Library (Github repository)*. URL: https://github.com/NEAT-project/neat (visited on 15/04/2018).

[65] *neat-performance (test suite for testing the performance of NEAT)*. URL: https://github.com/fredhau/neat-performance.

[66] *NetEm - Network Emulator*. URL: http://man7.org/linux/man-pages/man8/tc-netem.8.html.

[67] George V. Neville-Neil. 'Whither Sockets?' In: *Queue* 7.4 (May 2009), 35:34–35:35. ISSN: 1542-7730. DOI: 10.1145/1538947.1538949. URL: http://doi.acm.org/10.1145/1538947.1538949.

[68] *Node.js*. URL: https://nodejs.org/en/ (visited on 19/04/2018).

[69] *OpenSSL – Cryptography and SSL/TLS Toolkit*. URL: https://www.openssl.org (visited on 06/05/2018).

[70] Giorgos Papastergiou et al. 'On the Cost of Using Happy Eyeballs for Transport Protocol Selection'. In: *Proceedings of the 2016 Applied Networking Research Workshop*. ANRW '16. Berlin, Germany: ACM, 2016, pp. 45–51. ISBN: 978-1-4503-4443-2. DOI: 10.1145/2959424.2959437. URL: http://doi.acm.org/10.1145/2959424.2959437.

[71] G. Papastergiou et al. 'De-Ossifying the Internet Transport Layer: A Survey and Future Perspectives'. In: *IEEE Communications Surveys Tutorials* 19.1 (Firstquarter 2017), pp. 619–639. ISSN: 1553-877X. DOI: 10.1109/COMST.2016.2626780.

[72] B. Penoff et al. 'Portable and Performant Userspace SCTP Stack'. In: *2012 21st International Conference on Computer Communications and Networks (ICCCN)*. July 2012, pp. 1–9. DOI: 10.1109/ICCCN.2012.6289222.

[73] *Performance Analysis of BSD*. URL: http://www.brendangregg.com/blog/2015-03-06/performance-analysis-bsd.html (visited on 17/05/2018).

[74] *picohttpparser – tiny HTTP parser written in C (github public repository)*. URL: https://github.com/h2o/picohttpparser (visited on 26/05/2018).

[75] *poll(2) – Linux manpage*. URL: http://man7.org/linux/man-pages/man2/poll.2.html (visited on 19/04/2018).

[RFC768] Jon Postel. *User Datagram Protocol*. RFC 768. RFC Editor, Aug. 1980, pp. 1–3. URL: https://rfc-editor.org/rfc/rfc768.txt.

[76] *R: The R Project for Statistical Computing*. URL: https://www.r-project.org/ (visited on 22/05/2018).

[RFC5246] Eric Rescorla and Tim Dierks. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246. RFC Editor, Aug. 2008, pp. 1–104. URL: https://rfc-editor.org/rfc/rfc5246.txt.

[RFC6347] Eric Rescorla and Nagendra Modadugu. *Datagram Transport Layer Security Version 1.2*. RFC 6347. RFC Editor, Jan. 2012, pp. 1–32. URL: https://rfc-editor.org/rfc/rfc6347.txt.

[77] B. Reuther, D. Henrici and M. Hillenbrand. 'DANCE: Dynamic Application Oriented Network Services'. In: *Proceedings. 30th Euromicro Conference, 2004.* Aug. 2004, pp. 298–305. DOI: 10.1109/EURMIC.2004.1333384.

[78] *Scalable Event Multiplexing: epoll vs. kqueue*. URL: http://people.eecs.berkeley.edu/~sangjin/2012/12/21/epoll-vs-kqueue.html (visited on 20/05/2018).

[79] M. Scharf and S. Kiesel. 'NXG03-5: Head-of-line Blocking in TCP and SCTP: Analysis and Measurements'. In: *IEEE Globecom 2006*. Nov. 2006, pp. 1–5. DOI: 10.1109/GLOCOM.2006.333.

[RFC3261] Eve Schooler et al. *SIP: Session Initiation Protocol*. RFC 3261. RFC Editor, July 2002, pp. 1–269. URL: https://rfc-editor.org/rfc/rfc3261.txt.

[RFC3263] Henning Schulzrinne and Jonathan Rosenberg. *Session Initiation Protocol (SIP): Locating SIP Servers*. RFC 3263. RFC Editor, July 2002, pp. 1–17. URL: https://rfc-editor.org/rfc/rfc3263.txt.

[80] *select(2) – Linux manpage*. URL: http://man7.org/linux/man-pages/man2/select.2.html (visited on 19/04/2018).

[81] T. Seth et al. *Performance requirements for signaling in internet telephony*. Internet-Draft draft-seth-sigtran-req-00.txt. IETF, Nov. 1998. URL: https://tools.ietf.org/html/draft-seth-sigtran-req-00.

[82] David Shinazi. *Apple and IPv6 – Happy Eyeballs*. Email to the IETF v6ops mailing list. July 2015. URL: https://www.ietf.org/mail-archive/web/v6ops/current/msg22455.html.

[83] W. Richard Stevens, Bill Fenner and Andrew M. Rudoff. *UNIX Network Programming, Vol. 1*. 3rd ed. Pearson Education, 2003. ISBN: 0131411551.

[84] Lawrence Stewart and James Healy. *Tuning and Testing the FreeBSD 6 TCP Stack*. Tech. Rep. 070717B. CAIA, July 2007. URL: http://caia.swin.edu.au/reports/070717B/CAIA-TR-070717B.pdf.

[85] R. Stewart, M. Tüxen and I. Ruengeler. *Stream Control Transmission Protocol (SCTP) Network Address Translation Support*. Internet-Draft draft-ietf-tsvwg-natsupp-11. Internet Engineering Task Force, June 2017. URL: https://tools.ietf.org/html/draft-ietf-tsvwg-natsupp-11.

[RFC4960] Randall R. Stewart. *Stream Control Transmission Protocol*. RFC 4960. RFC Editor, Sept. 2007, pp. 1–152. URL: https://rfc-editor.org/rfc/rfc4960.txt.

[RFC6525] Randall R. Stewart, Michael Tüxen and Peter Lei. *Stream Control Transmission Protocol (SCTP) Stream Reconfiguration*. RFC 6525. RFC Editor, Mar. 2012, pp. 1–34. URL: https://rfc-editor.org/rfc/rfc6525.txt.

[RFC8260] Randall R. Stewart et al. *Stream Schedulers and User Message Interleaving for the Stream Control Transmission Protocol*. RFC 8260. RFC Editor, Nov. 2017, pp. 1–23. URL: https://rfc-editor.org/rfc/rfc8260.txt.

[86] *TCP Experiment Automation Controlled Using Python (TEACUP)*. URL: http://caia.swin.edu.au/tools/teacup/.

[87] Philipp S. Tiesel, Theresa Enghardt and Anja Feldmann. *Socket Intents*. Internet-Draft draft-tiesel-taps-socketintents-01. Work in Progress. Internet Engineering Task Force, Oct. 2017. 15 pp. URL: https://datatracker.ietf.org/doc/html/draft-tiesel-taps-socketintents-01.

[88] Brian Trammell et al. *Post Sockets, An Abstract Programming Interface for the Transport Layer*. Internet-Draft draft-trammell-taps-post-sockets-03. Work in Progress. Internet Engineering Task Force, Oct. 2017. 31 pp. URL: https://datatracker.ietf.org/doc/html/draft-trammell-taps-post-sockets-03.

[RFC6951] Michael Tüxen and Randall R. Stewart. *UDP Encapsulation of Stream Control Transmission Protocol (SCTP) Packets for End-Host to End-Host Communication*. RFC 6951. RFC Editor, May 2013, pp. 1–12. URL: https://rfc-editor.org/rfc/rfc6951.txt.

[RFC6458] Michael Tüxen et al. *Sockets API Extensions for the Stream Control Transmission Protocol (SCTP)*. RFC 6458. RFC Editor, Dec. 2011, pp. 1–115. URL: https://rfc-editor.org/rfc/rfc6458.txt.

[89] Geir Ola Vaagland. 'Improvements of the Linux SCTP API'. MA thesis. University of Oslo, May 2014.

[90] *Valgrind (home page)*. URL: http://valgrind.org/ (visited on 21/05/2018).

[91] *Web10G (homepage)*. URL: https://www.web10g.org/ (visited on 27/05/2018).

[92] F. Weinrank and M. Tüxen. 'Transparent flow mapping for NEAT'. In: *2017 IFIP Networking Conference (IFIP Networking) and Workshops*. June 2017, pp. 1–6. DOI: 10.23919/IFIPNetworking.2017.8264876.

[93] M. Welzl, S. Jorer and S. Gjessing. 'Towards a Protocol-Independent Internet Transport API'. In: *2011 IEEE International Conference on Communications Workshops (ICC)*. June 2011, pp. 1–6. DOI: 10.1109/iccw.2011.5963568.

[94] Michael Welzl and Stein Gjessing. *A Minimal Set of Transport Services for TAPS Systems*. Internet-Draft draft-ietf-taps-minset-03. Work in Progress. Internet Engineering Task Force, Mar. 2018. 46 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-taps-minset-03.

[95] M. Welzl et al. *TCP-CCC: single-path TCP congestion control coupling*. Internet Draft draft-welzl-tcp-ccc, work in progress. Oct. 2016. URL: https://tools.ietf.org/html/draft-welzl-tcp-ccc.

[96] *What's the difference between libev and libevent?* Sept. 2017. URL: https://stackoverflow.com/questions/9433864/whats-the-difference-between-libev-and-libevent (visited on 22/04/2018).

[97] Dan Wing and Preethi Natarajan. *Happy Eyeballs: Trending Towards Success with SCTP*. Internet-Draft draft-wing-tsvwg-happy-eyeballs-sctp-02. Work in Progress. Internet Engineering Task Force, Oct. 2010. 8 pp. URL: https://datatracker.ietf.org/doc/html/draft-wing-tsvwg-happy-eyeballs-sctp-02.

[RFC6555] Dan Wing and Andrew Yourtchenko. *Happy Eyeballs: Success with Dual-Stack Hosts*. RFC 6555. RFC Editor, Apr. 2012, pp. 1–15. URL: https://rfc-editor.org/rfc/rfc6555.txt.

[98] Dan Wing and Andrew Yourtchenko. 'Improving User Experience with IPv6 and SCTP'. In: *The Internet Protocol Journal* 13.3 (Sept. 2010). URL: https://www.cisco.com/c/en/us/about/press/internet-protocol-journal/back-issues/table-contents-49/133-he.html.

[99] Dan Wing, Andrew Yourtchenko and Preethi Natarajan. *Happy Eyeballs: Trending Towards Success (IPv6 and SCTP)*. Internet-Draft draft-wing-http-new-tech-01. Work in Progress. Internet Engineering Task Force, Aug. 2010. 13 pp. URL: https://datatracker.ietf.org/doc/html/draft-wing-http-new-tech-01.

[100] L. Wood. *Specifying transport mechanisms in Uniform Resource Identifiers*. Internet-Draft draft-wood-tae-specifying-uri-transports-08. Internet Engineering Task Force, May 2010. URL: https://tools.ietf.org/html/draft-wood-tae-specifying-uri-transports-08.

[101] Xipeng Xiao and L. M. Ni. 'Internet QoS: A Big Picture'. In: *Netwrk. Mag. of Global Internetwkg.* 13.2 (Mar. 1999), pp. 8–18. ISSN: 0890-8044. DOI: 10.1109/65.768484. URL: http://dx.doi.org/10.1109/65.768484.

[RFC3267] Qiaobing Xie et al. *Real-Time Transport Protocol (RTP) Payload Format and File Storage Format for the Adaptive Multi-Rate (AMR) and Adaptive Multi-Rate Wideband (AMR-WB) Audio Codecs*. RFC 3267. RFC Editor, July 2002, pp. 1–49. URL: https://rfc-editor.org/rfc/rfc3267.txt.

[RFC3286] John Yoakum and Lyndon Ong. *An Introduction to the Stream Control Transmission Protocol (SCTP)*. RFC 3286. RFC Editor, May 2002, pp. 1–10. URL: https://rfc-editor.org/rfc/rfc3286.txt.

[102] Sebastian Zander and Grenville Armitage. *CAIA Testbed for TEACUP Experiments Version 2*. Tech. Rep. 150210C. Melbourne, Australia: Centre for Advanced Internet Architectures, Swinburne University of Technology, 2015. URL: http://caia.swin.edu.au/reports/150210C/CAIA-TR-150210C.pdf.

[103] Sebastian Zander and Grenville Armitage. *TEACUP v0.8 – A System for Automated TCP Testbed Experiments*. Tech. Rep. 150210a. Melbourne, Australia: Centre for Advanced Internet Architectures, Swinburne University of Technology,

2015. URL: https : / / pdfs . semanticscholar . org / e4ab / 37857299025c1b48f57c76bc144c81788381.pdf.

[104] H. Zimmermann. 'OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection'. In: *IEEE Transactions on Communications* 28.4 (Apr. 1980), pp. 425–432. ISSN: 0090-6778. DOI: 10.1109/TCOM.1980.1094702.