NEATPy

A Standards-Conformant Protocol-Independent Transport System

Michael Gundersen

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

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2020

NEATPy

A Standards-Conformant Protocol-Independent Transport System

Michael Gundersen

NEATPy

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

Printed: Reprosentralen, University of Oslo

Abstract

The BSD sockets API has been the common interface at the transport layer for more than thirty years and hinders evolution by forcing programmers to commit to a transport protocol at design time. Currently, standardization efforts are made by the Transport Services Working Group in the IETF to create a transport API that replaces Sockets. The choice of transport is dynamically made at run time, based on abstract transport requirements specified by applications.

NEAT (A New, Evolutive API and Transport-Layer Architecture for the Internet) is a transport API written in C, and it was the first open-source implementation of TAPS. Since the NEAT project ended in 2018, the interface in TAPS has undergone several changes. In this thesis, we present NEATPy: a TAPS-conforming transport API written in Python, realized with the help of language bindings utilizing the protocol machinery in NEAT.

Our findings denote that the choice of target language represents a significant part of the natural overhead generated by a transport layer API using language interoperability. We argue that NEATPy's separation of concerns makes for a future-proof transport system that can easily adapt to future changes in the standardization efforts and, ultimately, assists in de-ossifying the Internet transport layer.

Acknowledgements

I would like to acknowledge my supervisor, Professor Michael Welzl, for his guidance and feedback. Throughout the process, he has given his honest opinion, which has helped to shape and inform the final product of this research. Lastly, I want to thank him for making it an enjoyable experience through his sense of humour and genuine interest in the project.

I am truly grateful for having a wonderful group of fellow students, all of which I consider close friends: Felix, Henning, Kai, Mattis, and Vegar. Through conversations, both technical and mundane, they have brightened my days in this especially challenging time, particularly when face to face contact was limited. I truly appreciate you all.

I would like to say a special thank you to Bridgette Vanderwolf and Trevor Adams for taking the time to proofread my thesis.

To my dearest friend, Kevin. I cannot thank you enough for your many years of friendship. Thank you for always being there through thick and thin, your friendship truly made a difference.

Finally, I would like to express my sincere gratitude to my parents for their unconditional encouragement and support. It is with thanks to them that I have had such extraordinary learning opportunities and have found a field I love.

Michael Gundersen
Oslo, May, 2020

Contents

List of Figures

List of Tables

Listings

Part I

Introduction and Background

Chapter 1

Introduction

1.1 Motivation

Ossification at the Transport Layer

In the process of creating an application a programmer, to some extent, needs to think about the dataflow and how the data is transported. For more than thirty years, the Berkeley Sockets Application Programming Interface (API) and its two transport options, TCP [35] and UDP [36], have been ubiquitous. The socket API forces the programmer to commit to a single transport protocol when designing programs. This static and inflexible mechanism makes it cumbersome for the programmer to opt-in for new transport services because the programs source code needs change. The transport needs of today's applications goes beyond the services provided by TCP and UDP. These needs have resulted in new transport alternatives, many of which are implemented in user space, layered over UDP. These alternatives enrich the choices of transport services, but complexity for the programmers remains. One must implement the logic for a possible fall-back in case a certain protocol is not supported along the network path, resulting in per-application protocol stacks.

Revitalization through IETF Standardization efforts

The Transport Services Working Group (TAPS WG) under the Internet Engineering Task Force (IETF) was created with the goal of re-enabling evolution at the transport layer. The end goal of its standardization efforts is a transport system that replaces the sockets API as the lowest common denominator interface to the transport layer. This transport system features an API that is protocol-agnostic, moving the decision of transport protocol from design time to run time.

1.2 Problem Statement

In parallel with the creation of TAPS, NEAT (A New, Evolutive API and Transport-Layer Architecture for the Internet) was conceived. NEAT is a user-space middleware and API written in C, developed by a European research project[1] and was the first open-source implementation of TAPS. The NEAT project

[1] Funded by the European Union's Horizon 2020 research and innovation programme under grant agreement no. 644334.

finished in 2018, while the TAPS API design has moved ahead. A development effort is therefore needed to making this transport system fully conformant to TAPS. At the end of NEAT's development period, experimental language bindings for Python were created. These bindings function as the groundwork for the implementation presented in this thesis.

1.3 Research Questions

The chosen implementation strategy poses some interesting research questions (RQ), which we would like to answer as a result of this thesis:

- **RQ1**: What are the main challenges in terms of language interoperability, when representing functions of a transport layer API written in C in Python?
- **RQ2**: Can the design of a wrapped, low-level transport layer library limit the implementation of a high-level transport layer interface in any way?
- **RQ3**: What is the performance penalty of a transport layer interface making use of language bindings?

1.4 Contributions

The main contribution of this thesis is **NEATPy**: an up-to-date, TAPS-conforming transport system implemented in Python, leveraging the protocol machinery provided by NEAT.

As part of the process as a whole we have:

- Contributed with various bug fixes to the NEAT language bindings, removing their experimental status.[2]
- Utilized said bindings to create NEATPy.
- Created an extensive documentation solution, presenting the API to developers in a lucid manner. It is presented in Appendix A.[3]

These contributions have enabled us to develop an API that is developer-friendly, providing a high level of ease-of-use.

1.5 Chapter Overview

Chapter 2 starts with a brief introduction to the transport layer in general: from the very beginning with ARPANET to contemporary transports as QUIC and SCTP. This concise section describing the origins makes it easier to grasp the present-day situation. The remainder of this chapter features a more detailed description of TAPS and NEAT, as a fundamental understanding of both are needed.

Chapter 3 presents related work, PyTAPS and Network.framework, which are the known implementations of a transport system conforming to the ongoing standardization effort. Both are briefly presented

[2] Outlined in section 4.2.4.
[3] The documentation is hosted online at: https://neatpy.readthedocs.io

with significant implementation details. At the end of this chapter, a comparison of PyTAPS, Network.framework and NEATPy is presented.

Chapter 4 motivates choice of Python as the implementation language, followed by an introduction to SWIG, which is the tool to realize language bindings from C (NEAT) to Python. These bindings are heavily used by the implementation and therefore they are explained in detail.

Chapter 5 presents the implemented, conformant API in full. Each section corresponds to a phase in the transport system's life cycle in TAPS. Furthermore, each section starts with a mapping from constructs featured in the interface draft to a concept in NEAT. Implementation details follow the initial comparison. Throughout the chapter, several different techniques are used to shed light on implementations details.

Chapters 6 and 7 report on tests and benchmarks of NEATPy, focusing on the overhead generated by the new transport system. This aids us in answering the research questions outlined.

Chapter 8 concludes the thesis by answering our research questions and comparing the result against our listed goals. Additionally, we propose some future work.

1.6 Project Source Code

The source code in full is available and hosted on GitHub.[4]

[4] https://github.com/theagilepadawan/NEATPy

Chapter 2

Background

2.1 The Transport Layer

2.1.1 Genesis

Connected computers and communication between them have existed for a long time. The need for systematic transportation of data between nodes arguably has its beginnings with the introduction of packet switching. Advanced Research Projects Agency Network (ARPANET) and CYCLADES both utilized the packet switching technique.

ARPANET

Every host in ARPANET was accompanied by additional nodes called Interface Message Processors (IMP), which made interconnection possible by packet switching. The very first Request for Comments (RFC) describes connection management and creation [20]. This first approach is connection-oriented with primitives for both connection creation and termination. Even at this early stage, we have the concepts of messages (information divided into one or more packets) and checksums.

There were several iterations of ARPANET's host-host software and eventually got the name Network Control Program (NCP) [8] [25]. NCP was running on all hosts connected to the network. This, however, is not end-to-end as known today, mainly due to two reasons. Firstly, because of the intermediary IMPs, as each host passed its message to the IMP which carried out the network functionality. Secondly, the NCP managed connections more than the network itself. It utilized a stop-and-wait technique which meant the hosts could only receive one message at a time.

CYCLADES

CYCLADES introduced a cleaner separation of concerns with the notion of sending a "datagram" between packet switches and on top of this, a transport protocol implemented at host level, acting as a virtual circuit between two endpoints. This transport protocol used many techniques ahead of its time, such as sliding windows and end-to-end acknowledgements. The separation of a unreliable service at the network layer and a reliable service between the hosts themselves is, of course, the precursor to the

renowned TCP/IP model. Several of the ideas conceived in CYCLADES continued to be developed into the Transmission Control Protocol (TCP).

2.1.2 Transmission Control Protocol

As time passed networking moved from research- and military-based networks that were isolated to interconnected networks globally; the conception of the "Internet Transmission Control Program" took place [7]. Some iterations later the Internet Transmission Control Program as a whole was divided into the Transmission Control Protocol and the Internet Protocol (IP).

With TCP, a great deal of complexity was moved from the packet switching nodes to the hosts themselves, thanks to advances in both central processing units and memory. The result was a reliable, ordered and connection-oriented protocol that operates on a byte stream. To this day, forty years later, TCP is widely used in major applications such as email and the Hypertext Transfer Protocol (HTTP). Using TCP, the application is guaranteed error-free transportation of data. This guarantee is mainly enabled by re-transmission of packets that are lost, and the use of checksums.

Equilibrium between TCP and the Application Layer

For optimal performance, there iis a delicate balance between how the applications utilize the transport protocol stack and how the transport handles the ever-changing application landscape. Two notorious examples of this are the "congestion collapse" in the late 80s and the "World Wide Wait" in the 90s [1].

The World Wide Wait is an example of how an application's performance could significantly decrease when not utilizing the underlying transport protocol stack the way it should be used. In this specific case, HTTP/1.0 initiated a new TCP connection for every URI in an HTML document, which caused major, unnecessary overhead. The next iteration of HTTP, HTTP/1.1, patched this problem by a policy called persistent connections. The deployment of HTTP/1.1 reduced the amount of web-based traffic on the internet by 40%, and load times of webpages by up to 40% in some cases, as reported in [31].

A fixed window size caused the congestion collapse in TCP. When designing TCP, the engineers had not taken into account how major growth in network size would affect transportation. This oversight resulted in topped up queues, heavy retransmissions and consequently a significant decline in goodput. Over the years, congestion control in TCP has been reiterated several times – from the very beginning with TCP Tahoe's reactive approach implemented with timers to TCP Vegas's proactive approach with calculated round trip times. The reliability service provided by TCP is guaranteed by retransmission, and therefore congestion control is a crucial part of its performance.

2.1.3 User Datagram Protocol

The User Datagram Protocol was brought to life to complement the reliable, in-order service provided by TCP. The 1980 RFC [36] describes it as "a procedure for application programs to send messages to other programs with a minimum of protocol mechanism.", which is really what it is, a header put on top of IP. UDP provides a unreliable service. Communication is unidirectional, and data is sent within units called datagrams.[1] UDP is a connectionless protocol; once a packet is sent, no further operations

[1] A combination of *data* and *telegram* - coined by Louis Pouzin, the inventor of CYCLADES.

are carried out, and no state is saved by UDP. Due to its unreliable nature, UDP is used by applications and services where timeliness trumps reliability, e.g. a real-time, online multiplayer game where latency is crucial or service discovery like the Simple Service Discovery Protocol (SSDP) which broadcasts information.

UDP as a Substrate

Due to its simplicity, UDP has been used as a substrate by applications implementing custom, user space transports. The reason is straightforward; the application requires transport services beyond what UDP (and TCP) offers. This use of UDP is a mixed blessing; albeit providing a lot of flexibility due to its lightness, applications using UDP need to implement vital mechanisms such as congestion control. Given this use of UDP is quite common, the IETF have developed standardization and guidance for applications utilizing UDP in this way [12].

2.1.4 The Socket API

A socket API is the way a programmer takes use of different protocol stacks supported by the operating system. The original, and well renowned implementation is that of Berkeley sockets. This implementatoin provided the programmer with an interface utilizing the operating system's implementation of either UDP or TCP. The API was written in C and part of the UNIX operating system. Considered a de facto standard, most of the different socket APIs written are based upon Berkeley sockets. However, the socket API has not changed fundamentally since its origins, while the application's needs have exceeded the services of UDP and TCP.

2.2 Contemporary Transport Protocols

Since the 1980s, computer technology has evolved rapidly. Improved, and more complex hardware empowers programmers to create more sophisticated applications. These may have different requirements for transportation of data, than offered by the socket API's TCP and UDP – e.g., an application with multiple subflows needing only partial reliability. The following is a brief introduction to two contemporary transport protocols that offer services beyond the two giants, UDP and TCP.

2.2.1 Stream Control Transmission Protocol

The Stream Control Transmission Protocol was originally designed to handle telephone signaling over IP. With the aforementioned need for a transport that provides functionality beyond TCP and UDP, the IETF recognized SCTP as a candidate for such a transport protocol, and this ultimately led to the standardization of SCTP as a general-purpose protocol [42].

Compared to TCP and UDP, SCTP is an extensive protocol with a plethora of options and operation modes. Below is a list of the essential information:

- **Data unit and operation modes**: SCTP is *message oriented* and operates on units coined *chunks*. Each chunk has a type, which is either a data chunk or one of many control chunks (e.g. abort, error etc.). SCTP uses the term *association*, which is semantically like a connection,

but associations can be multihomed and multistreamed. SCTP supports both a one-to-many and one-to-one mode. This support makes it possible to have several associations maintained within one socket. Due to this and other features in SCTP, the Socket API was changed to support this [45].

- **Multistreaming and optional ordering**: SCTP has the concept of multiple logical flows within an association. Each stream within an association has a *stream sequence number*. This (partial ordering), and the option of immediately passing packets to the application upon arrival (order-of arrival delivery) solves the head-of-line (HOL) delay which occurs when TCP needs to handle packets that arrive out of order.
- **Multihoming**: SCTP supports multihoming at the transport layer. A host with more than one interface is labeled as a multihomed host [4]. This feature makes it possible for an application to bind multiple addresses to a single endpoint. Also, it is mitigation against network failure, being able to use alternate paths when detecting failures.
- **Comparison to TCP**: In many ways, SCTP exceeds the capabilities of TCP. As described, out-of-order arrival delivery is possible. Furthermore, SCTP provides a reliable service with strict order-of-transmission, just like TCP. Additionally, SCTP can be configured to offer partial reliability [43]. Regarding congestion control, SCTP uses the same techniques as TCP, including slow-start and fast retransmit.

As described in the list above, SCTP uses constructs from both UDP and TCP and, in some ways, functions as a hybrid of the two. What is interesting is that many of the features provided by SCTP are made possible when we simply step away from a byte stream to a data unit with known size. Examples of this could be the ability to deliver messages in the wrong order or providing partial reliability[2].

Despite its being a standardized protocol, SCTP, with all its features, has not gained traction and deployment, neither in the Internet nor major operating systems due to the infamous "vicious cycle" [17] of deploying new protocols. Work and research have been put in to encourage adaptation and increase deployment:

- The authors of [2] transparently utilize multihoming with the help of a shim layer implemented in the kernel. Applications required no modification and gained the benefit of SCTP's fault tolerance when hosts where multihomed.
- In a similar fashion, [50] presents an implementation which exploits SCTP's multistreaming, again transparently, without modifications to applications. This implementation proved to work especially well with parallel file transfers, additional streams taking advantage of a large congestion window already established with a previous transfer.

Additionally, to assist in gaining SCTP traction, the examples above use elements of SCTP that CAN be offered without leaking message orientation. Such observations are useful in the creation of a transport system, and particularly when dealing with a group of connections, which we address in 5.3.3.

2.2.2 QUIC

QUIC [38], like SCTP, is a reliable, general-purpose transport protocol that embraces multistreaming within its connections. Originally designed and implemented by Google, it's now undergoing

[2] Both require signaling by SCTP, but could trivially be implemented in any protocol.

standardization efforts by the IETF [24]. As described in Section 2.1.3, QUIC uses UDP as a substrate, and consequently, any implementation needs to implement its own congestion controller. Albeit operating over UDP, QUIC can fall back to using TCP when either RTTs exceed a certain limit[3], or UDP is blocked along the network path. QUIC has two features that make it stand out:

- **0-RTT connection establishment**: This enables an application to send data immediately and is made possible by clients caching information about server endpoints[4]. Together with 0-RTT establishment, QUIC is always encrypted. This is a very desirable combination, and with that, was included in *TLS 1.3* [39].
- **Improved connection migration**: A connection in QUIC is identified by a 64-bit connection ID. This eases the migration of a connection, as opposed to TCP, where connections are identified by a 4-tuple of [source address/port + destination address/port] and a client e.g., changing its interface would deem a connection invalid.

Originally developed by Google, QUIC aims to break out of the previously mentioned, vicious deployment cycle. This is achieved, in that QUIC has been used within the Chrome web browser since 2014 and subsequently tested by millions of users. Measurements presented by Google show a very high success rate for QUIC when used in their browser [21]. Given these results, and the features presented, QUIC is replacing TCP as the transport protocol for the next iteration of HTTP, HTTP/3 (H3) [3]. In the same way that QUIC is incorporated and tailored for H3, using it as a general-purpose transport protocol calls for an API that utilizes its features. This need is the very reason for the transport system API presented by TAPS, which we describe next.

[3] Currently set to 2.5 seconds.
[4] As a consequence a first-ever QUIC connection attempt requires 1-RTT due to caching.

2.3 TAPS

There is consensus that the Internet's transport layer is at a standstill and ossified, hampering further evolution. The authors of [33] present two principal sources for the ossification:

1. The ubiquitous deployment of middleboxes
2. **Limited flexibility of the current transport API**

Likewise, [33] presents a set of solutions:

1. Designing middlebox-proof transports
2. Signaling for facilitating middlebox traversal
3. **Enhancing the API between the applications and the transport layer**
4. Discovering and exploiting end-to-end capabilities
5. Enabling user-space protocol stacks

As highlighted, the Transport Services (TAPS) Working Group of the Internet Engineering Task Force (IETF) was created to deal with the inflexibility of ubiquitous Socket API. The goal is to standardize an architecture and API for a transport system that is protocol agnostic, focusing on transport features.

The main output of the Working Group consists of three documents:[5]

1. **Architecture**: Presents an overall architecture for a transport system [34]. It presents design principles and defines concepts and terminology used throughout the API, altogether forming an architecture for exposing transport protocol features to applications.
2. **Interface**: This document [46] provides an abstract API to the transport layer, following the aforementioned architecture. With this, the TAPS WG aims to replace the BSD sockets API as the lowest common denominator interface to the transport layer.
3. **Implementation**: The implementation document [5] sits logically between the architecture and interface document. Providing implementation guidance for the abstract API, it follows the design outlined in the architecture document.

2.3.1 API

The abstract application programming interface (API) provided by the TAPS Working Group (TAPS WG) is a product of (1) analyses of existing IETF transport protocols to determine common transport services to support [49] [13] and (2) additional analyses of security properties [52].

Concepts and Abstractions

The abstract interface [46] is *object-oriented* and has the concept of *actions* and *events*. The application will interact with the objects created by invoking actions on said objects. The interface is phase-oriented and has three main phases: pre-establishment, establishment, and after-establishment. The pre-

[5] At the time of writing all documents are work in progress and labelled *Internet-drafts*.

establishment phase enables the user to specify desired Connection Properties and Security Parameters. In the Establishment-phase, the information gathered from the pre-establishment phase is used along with information from the network itself to select a path and protocol stack for the connection. This can be done actively (initiate), passively (listen), or in a peer-to-peer fashion doing both (rendezvous). The after-establishment phase contains every event after a connection is established: data is transferred, events handled, and the connection is closed.

Another important design decision is that of message orientation. This enables applications to communicate boundaries for the data being sent. If a connection is backed by a protocol that does not preserve message boundaries (i.e. a protocol that is stream-oriented, like TCP), the API helps with the concept of Message Framers. This mechanism makes it possible for applications to create custom framers, providing Application Level Framing (ALF). The framer implements abstract functions, which are appropriately called by the transport system for framing and deframing.

Lastly, the API is asynchronously designed, where objects, i.e. a connection, are firing events asynchronously. This event-driven nature modernizes the API and enables multiple operations.

Transport Features in a Transport System

As RFC 8303 [51] defines, a *transport feature* as “a specific end-to-end feature that the transport layer provides to an application”. Subsequently, it defines a *transport service* as “a set of transport features, without an association to any given framing protocol, which provides a complete service to an application.” An example could be transmission of real time multimedia streaming.

The authors of [49] expand on the concept of a transport feature by distinguishing between "functional", "optimizing", and "automatable" transport features. Functional transport features are features that offer functionality that the application programmer has to take into consideration, otherwise resulting in unexpected behavior and ultimately a failure. Optimizing features are features that would not result in a failure if customized, but would require application-specific knowledge to be used efficiently. Lastly, "automatable" features are transport features that do not need application-specific knowledge. When constructing the minimal set of transport features, the "automatable" features were removed, as well as features that, if they were implemented over TCP or UDP, would behave incorrectly. The conclusion of deriving a minimal set of transport features was that a transport system must implement all functional and optimizing features.

Transport Properties are TAPS’s conceptualization of said transport features and are branched out into three subcategories of properties: *Selection Properties*, *Connection Properties* and *Message Properties*.

Selection Properties are used in the selection of transport protocol and network path, and they are provided to the Preconnection object. Most Selection Properties may be set with one of five preferences: *Require*, *Prefer*, *Ignore*, *Avoid*, or *Prohibit*. Reliability and ordering of the data are two examples of Selection Properties that can be set on a Transport Property object.

The Connection Properties are either generic protocol-specific or a simple primitive like the connection state. They have three major use cases: (1.) configuration of the connection, (2.) being queried by the application, and (3.) being passed along during a Soft Error event. Writable Connection Properties should be set as early as possible, i.e., during pre-establishment and on a Preconnection object. This is beneficial, as the API could use this additional information during the selection process.

Message Properties are a way for the application to customize the behavior of the data being sent. The effect of this mechanism is once again more flexibility for the application programmer in contrast with the Socket API. Message Properties are passed along with a MessageContext object. Examples of a Message Property could be *singular transmission* or *lifetime*.

2.3.2 TAPS Client and Server Example

TAPS Server

Firstly, a new local endpoint object is created and initialized a listening port. Secondly, objects are created for Transport Properties and Security Parameters for the Preconnection.

```
// Create and initialize a new local endpoint object
   new_endpoint = newLocalEndpoint();
   new.endpoint.port(5000);

// Transport and Security objects
   transport_properties = newTransportProperties();
   security_parameters = newSecurityParameters();

// Create a Preconnection and start listening, handle incoming requests
   pre_connection = newPreConnection(new_endpoint,
                             transport_properties,
                             security_parameters);
   pre_connection.listen();
// Event ConnectionReceived occured
   connection.receive();
// Received event occured
   connection.send(Response);
// Close connection
   connection.close();
// Kill server / stop listening
   pre_connection.stop();
```

TAPS Client

The client follows the same pattern as the server. An object is created for the endpoint, Transport Properties, and Security Parameters. A Preconnection object is created, and, as this is a client, `initiate()` is invoked on the object. This is known as the *establishment phase*. Following the initiate-call, two events can occur: (1.) the ready event when a connection has been established, or (2.) the initiate error event.

```
// Create an endpoint object and set properties.
   remote = new_end_Point();
   remote.host("localhost")
   remote.port(5000);

// Create an object for transport properties and security parameters.
   t_props = new_transport_properties();
   security_params = new_security_parameters();

// Create Preconnection object
```

```
11    pre_con = new_preconnection(remote, t_props, security_params);
12
13  // Initiate connection, handle ready-event and send data
14    connection = pre_con.initiate()
15    connection.send(msg_req) // ready event fires ---> send
16
17  // Call receive, wait for an event and close connection
18    connection.receive()
19    // received events are handled (asynchronously)
20
21    // i.e final msg received ---> close connection
22    connection.close()
```

2.4 NEAT

NEAT is one of two known open-source implementations of the efforts done by the IETF. The NEAT project was initiated briefly after the TAPS working group, and its participants contributed both to NEAT and TAPS. Consequently, NEAT influenced TAPS and vice versa.

2.4.1 Architecture and Design

NEAT uses a component design, and with this, emphasizes separation of concerns, with each component providing one core service. An overview of the overall architecture can be seen in Figure 2.1. Following is a short description of the various components:

- **Policy component**: Is the mechanism in NEAT that is responsible for generating transport candidates based around so-called *NEAT properties*. A property is a key/value pair and is used both for user requirements passed to the NEAT library and within NEAT policies themselves. NEAT uses JavaScript Object Notation (JSON) as the format to represent these properties. Additionally, the Policy Manager can cache and monitor system and network characteristics, which could be used for subsequent connection attempts. This is coined the *Characteristics Information Base (CIB)*.

- **Selection component**: Is used to select a transport protocol for a given connection. The selection component gets its input from the Policy Manager, and its task is to test and select a transport protocol for the connection.

- **Transport component**: Has the responsibility to create and instantiate the transport stack chosen by the selection component. This includes both a user- and kernel-space API.

- **Signaling and handover component**: Complements the transport component when the connection needs additional setup, e.g. signaling for middlebox traversal.

- **Framework component**: Works as an intermediary between the user API and the other components mentioned. In addition to this, the framework components handle core library tasks, such as address resolution.

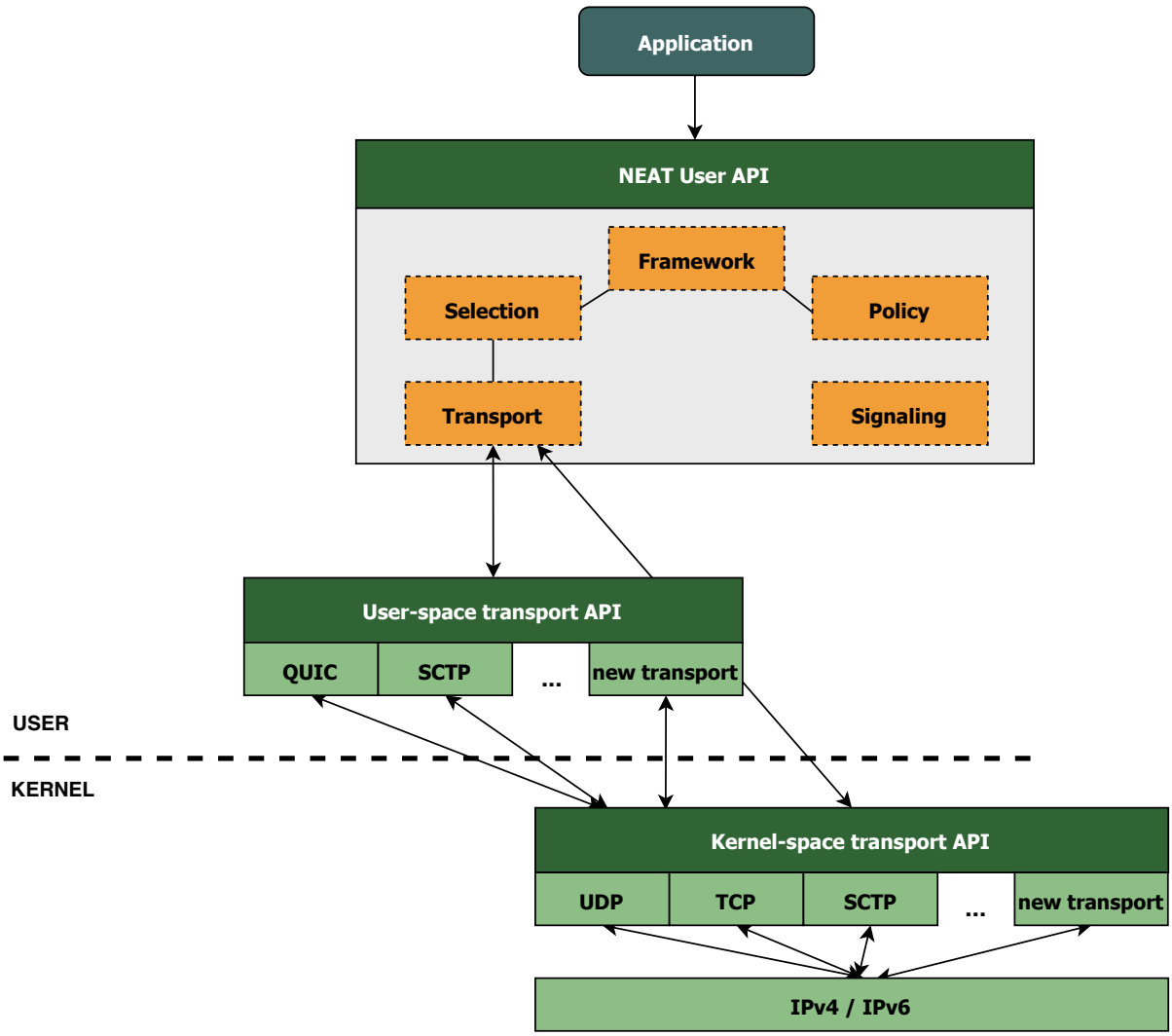


Figure 2.1: NEAT's architecture [27]

Asynchronous Design with the Help of libuv

Contemporary applications are designed with an asynchronous approach when handling operations like calls to databases, file, and network I/O. Similarly, NEAT provides an API that is asynchronous with the help of *libuv* [29]. Libuv is an asynchronous, event-driven I/O library, written in C++ and available cross-platform. The library provides two powerful abstractions for dealing with network I/O; these are *handles* and *requests*.

Handles are libuv's concept of long-lived objects. These objects are registered within the loop, which we will introduce in the following. NEAT utilizes mainly a type of handle called *poll handles* which are used to monitor file descriptors. After NEAT has resolved what type of protocol(s) needs to be created, it does the appropriate kernel calls to create the socket and uses the file descriptor given to initialize a poll handle. Lastly, it will call *uv_poll_start* together with a callback and bitmask for events to poll for (readability, writability, or disconnection). This registers the handle to the main event loop. Figure 2.2 presents the stages of the event loops iteration. NEAT's file descriptors are polled at the stage "Poll for I/O", and the callback that was registered with the handle is fired. The callback registered

with the file descriptors within the event loop is a function within the NEAT library, which functions as a dispatcher for further callbacks registered by the application for various events. Said events are a set which is specified by the NEAT User API, presented in Table 2.1. The application are able to register custom functions as callback for each of these events. This is realized through a NEAT struct, *neat_flow_operations*. Each member in this struct corresponds to an event, and an application registers a callback by assigning it to the desired member and calling `neat_set_operations()`.

Another important aspect of the libuv event loop is that network I/O is tied to a single thread (the very thread of the event loop). This tie was one of the more important aspects to have in mind during the implementation of NEATPy.

Figure 2.2: The internal loop of libuv

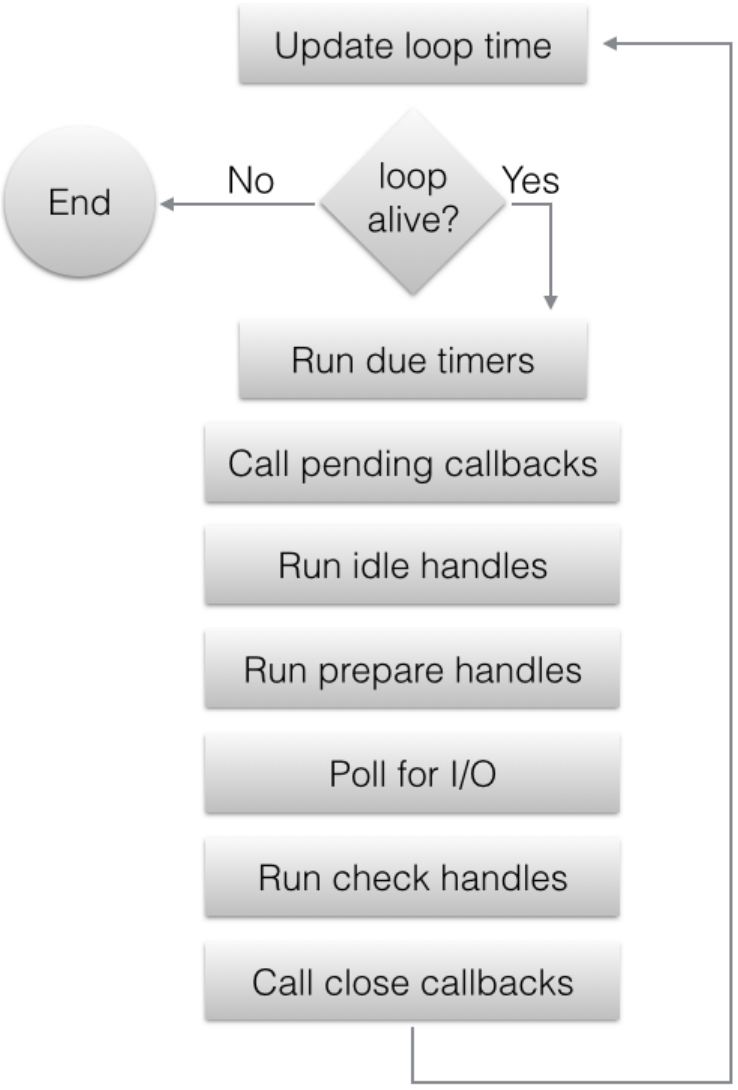


Table 2.1: The set of callbacks available for applications using NEAT

Callback	Description
on_connected	A connection has been established for the flow.
on_readable	The flow can be read from without blocking.
on_writable	The flow can be written to without blocking.
on_all_written	All data sent with `neat_write` has been completely written.
on_aborted	The remote end aborts the flow[1]
on_close	Called when graceful connection shutdown has completed.[1]

[1] Available for flows using TCP or SCTP.

2.4.2 Essential Implementation Details

Abstractions

NEAT brings two major abstractions to the table:

- **Flow**: To move away from the protocol binding in the Berkeley Socket API, NEAT introduces the concept of a flow. In the same manner, like a socket, a flow is a bidirectional link but differs in being a dynamic concept: to create a flow, one does not need to specify a transport protocol.
- **Context**: A context is closely tied to the application's host and operating system. Initialization, will, therefore differ between operating systems. For example, on Linux, Netlink sockets are used to gather address information.

NEAT follows the specification of Selection Properties in TAPS, and its API gives the application programmer the possibility to specify properties for the communication. These properties are used by the policy component in the NEAT User Module. Here the properties given by the user are mapped to policies. This mapping, combined with the Happy Eyeballs (HE) mechanism provided by the API, enables racing between candidate protocols, and also the possibility for the application to gracefully fall back in case a protocol fails for any reason. This eliminates the general architectural problem [27] with the Socket API. NEAT aims to revolutionize the transport layer, but at the same time, hide complexity and make life easier for the programmer. This is largely due to transport selection and fallback being handled internally by NEAT; the abstraction of properties is the only thing that is presented to the application programmer.

Racing

If more than one transport stack is deemed a candidate for the service needed, NEAT features a Happy Eyeballs (HE) mechanism, which enables racing between candidate protocols. This is a great benefit as it allows for the application to gracefully fall back in case a protocol fails for any reason. This eliminates the general architectural problem [27] with the Socket API. If one candidate protocol is preferred over

another, the HE mechanism delays initiations of the other protocols. Furthermore, the connection that successfully establishes first is selected for the flow. For optimization purposes, the results of the aborted connection attempts are cached within the aforementioned CIB, for which the duration of the cached entry is configurable.

Multistreaming

To be able to support multistreaming protocols such as SCTP [42], TAPS has the concept of Connection Groups and the Clone Action. NEAT's implementation differs in that both explicit and implicit multistreaming is possible. Regarding transport features, previously mentioned in Section 2.3.1, multistreaming falls in the category of features being "automatable". This is the reasoning behind implicit multistreaming in NEAT; if applicable, NEAT can internally utilize multistreaming for performance optimization. NEAT features a rather sophisticated, transparent flow mechanism [48]. This removes the additional efforts for developers when utilizing multiplexing protocols.[6]

Nevertheless, the API exposes the possibility of specifying the numbers of streams, for applications that are aware of this, and want to specify this explicitly. With this, the application developer can specify the number of streams to create during initiation, and subsequently, specify which stream data should be written to during sending.

2.4.3 Surrounding Environment and Life Cycle

A connection is realized in NEAT by either passively listening with `neat_accept()` or actively calling `neat_open()`. As aforementioned, in NEAT, we have the concept of flow and context. In this way, the life cycle of a connection is realized through these two concepts. A context is an environment where the flow resides. Everything before either neat_open or neat_accept can be considered as the equivalent of a pre-establishment phase in TAPS. This includes the creation of a context, a flow, as well as specifying properties and setting them. A flow is terminated with either `neat_close()` or `neat_abort()`, and a context is terminated with `neat_free_ctx()`.

TAPS and NEAT also apply a stricter semantic meaning to the opening and termination of a connection than common with, e.g., TCP. TCP's feature of half-closed connections is therefore not supported or part of the API. This choice is made to support all protocols.

2.4.4 Sending Side Buffering

Control of the sending side buffer has proven to make a significant difference for applications as there is an extensive delay in the hosts themselves. The TCP_NOTSENT_LOWAT option was proposed as a remedy when overstuffing in the kernel would result in unnecessary delay before a blocking write occurs [28]. In this way, the socket becomes writable only when a certain threshold is reached. TAPS outlines the Sent event but leaves it up to an actual implementation to specify whether the data still resides in a buffer at the host itself or not. NEAT is specifying this in the on_writable callback and is only called when the flow can be read non-blocking. The same applies to the on_readable event.

[6] At the time of writing this feature is limited to the FreeBSD operating system and SCTP.

2.4.5 NEAT Client and Server Example

NEAT Server

The basic building blocks for any application using NEAT for communication are a Flow, Context and Operations. First, a Flow and Context are initialized. What follows is the registration of callbacks to the Operations struct, so that the appropriate function is called when an event is fired. The call to neat_start_event_loop will get NEAT running.

```
// Create a new context
   ctx = new_neat_context();
   if (!ctx) {
      return EXIT_FAILURE;
   }
// Create a new flow within the context
   flow = new_flow(ctx);
   if (!flow) {
      return EXIT_FAILURE;
   }
// Create flow_operation struct and set up call backs
   flow_ops_struct = new_flow_operation_struct();
   flow_ops_struct.callback_X = x; // and so on...
// Set up is done, start listening / wait for flows to close
   neat_start_event_loop(ctx, NEAT_RUN_DEFAULT);

         /* Internally in NEAT */
   while (1) {
      // Callbacks are handled...

      // Either local or remote host ends the flow
      if (on_close || on_aborted) {
         break;
      }
   }
// Event loop returns, app (server) --> exits
   EXIT_SUCCESS
```

NEAT Client

The blueprint for the client is quite similar to the server. Being the side that initiates communication, the client must open the flow and also has the possibility to specify properties mentioned earlier.

```
// Create a new context
   ctx = new_neat_context();
   if (!ctx) {
      return EXIT_FAILURE;
   }
// Create a new flow within the context
   flow = new_flow(ctx);
   if (!flow) {
      return EXIT_FAILURE;
   }
// Create flow_operation struct and set up call backs
   flow_ops_struct = new_flow_operation_struct();
```

```
   flow_ops_struct.callback_X = x;
// Set properties for the flow
   neat_set_property(ctx, flow, json_props);
// Open the flow and connect to the endpoint provided
   neat_open(ctx, flow, "uio.no", 80, NULL, 0)
// Start event loop
   neat_start_event_loop(ctx, NEAT_RUN_DEFAULT);
// on_connected call back invoked

      /* Internally in NEAT */
   while (/* app has data to send */) {
      // on_writable invoked --> write data
      // on_all_written --> shutdown or re-enable on_writable
      // on_readable invoked --> read response

      if (on_close || on_aborted) {
         break;
      }
   }
// Event loop returns, app (server) --> exits
   EXIT_SUCCESS
```

Chapter 3

Related Work

3.1 PyTAPS

PyTAPS [14] is a transport system implemented in Python. The implementation has used the specification from the abstract interface described by the TAPS WG as a guide from the very start, and all implementors contribute to the working group.

Asynchrony

To present an asynchronous transport system, PyTAPS uses asyncio, a Python Standard Library for writing concurrent code. There are three main aspects of asyncio which assist PyTAPS in achieving asynchrony:

- **The event loop**: In a similar manner to libuv used in NEAT, asyncio provides an event loop that operates on tasks.
- **Co-operative routines (coroutines)**: An asynchronous block of code with the ability to yield (that is, to pause its execution and give control to the event loop scheduler) at any time during its execution, and, maintain its internal state.
- **Tasks**: These enable concurrent execution of coroutines, by wrapping said coroutines. Tasks are then registered with event loops queue.

A major design decision of PyTAPS is that all API functions, as well as all callback functions registered by the applications, is defined as coroutines. The developers backs up these decisions with two reasons: (1.) the TAPS API is heavily callback based. Coroutines ease the task of not exiting until every task added to the loop has finished. (2.) coroutines provide the developers with additional flexibility, e.g., a developer could specify coroutine dependencies [14]. Coroutines, in conjunction with the event loop, make for a powerful construct. In addition, to yield at any time during execution, a coroutine is able to schedule an additional coroutine to be executed by the event loop next:

```
1 await self.loop.create_datagram_endpoint(lambda: DatagramHandler(self),
2                 local_addr=(self.local_endpoint.address[0], self.local_endpoint.port))
```

Listing 3.1: Usage of a coroutine in PyTAPS

Supported Protocols and Features

At the time of writing, PyTAPS supports TCP, UDP, and the use of TLS on top of TCP. PyTAPS also uses convenience functions provided by asyncio to create sockets. An example can be seen in Listing 3.1, where PyTAPS uses the *await* keyword, stating that it wants the ongoing subroutine to yield and schedules another coroutine (the asyncio coroutine that creates a UDP socket) to run immediately.

Furthermore, the implementation supports framers, in that users of the transport system implement an abstract framer class in the API.

A Suitable Candidate for Testing

PyTAPS makes for an ideal implementation to test the API implemented in this thesis. Both APIs are written in Python and follow the interface draft as closely as possible. A result of this is that tests of both implementations could use almost identical source code, and test the difference in performance solely on the back end machinery.

3.2 Network.framework

Network.framework [22] is Apple's implementation of a modern transport system. An API is available in both Objective-C and Swift, used to transport data by applications across Apple's many platforms. Apple contributes to the Working Group extensively, and the API provided by Network.framework follows the interface closely.

Being a platform for thousands of developers, the transport API presented by Apple has the advantage of being tested thoroughly. In extent, this provides value for the TAPS WG, as the ongoing standardization efforts could make good use of Apple's experiences.

There are also some more sophisticated and advanced options presented to the developers, like Optimistic DNS and Service Classes. Setting a service class makes it possible for the application to indicate what type of traffic it is sending on a given connection, e.g. it could set the Service Class to ".background" to get a "scavenger like" connection, where packets will be queued with a lower priority on an interface. Optimistic DNS will try to establish a connection with an expired DNS answer, and, simultaneously execute a new DNS query. This strategy makes for an optimal connection establishment.

Lastly, there is interesting work put in by Apple regarding User-Space Networking.[1] The objective with User-Space Networking is to eliminate the usual overhead of context switching between the kernel and application when making syscalls to interact with the socket. What enables this is a memory-mapped channel directly from the interface to the transport stacks, which is moved within the applications in user space.[2]

[1] Presented as part of [16].
[2] At the time of writing TCP and UDP are supported.

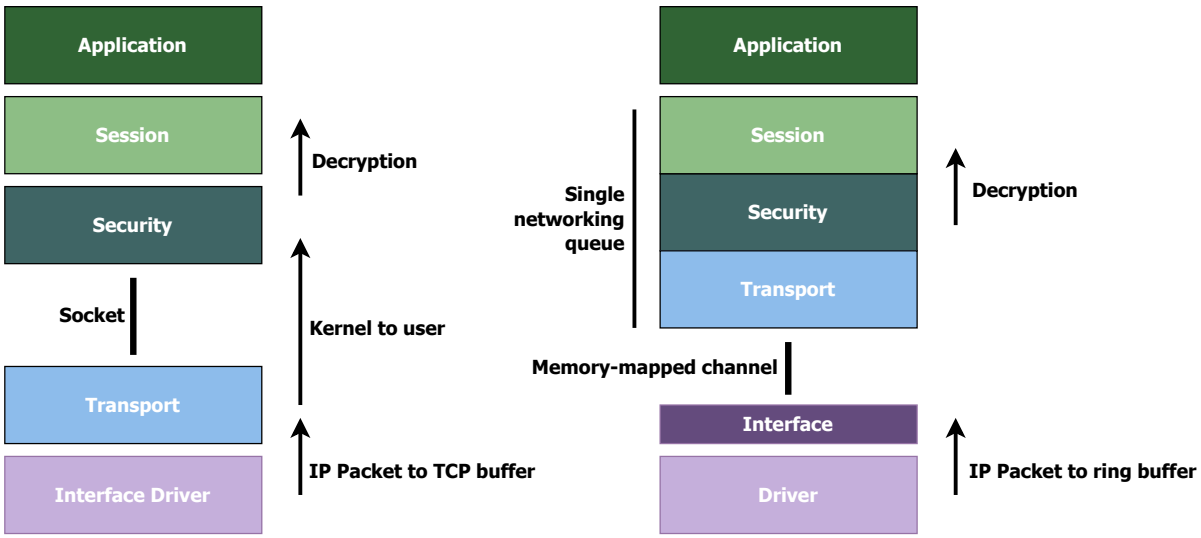


Figure 3.1: User-Space Networking in Network.framework [16]

3.3 Comparison of TAPS-conforming Implementations

Table 3.1: Conformity of TAPS implemented interfaces

What	PyTAPS	Network.Framework	NEATPy
API language	Python	Swift | Objective-C	Python
Supported transport protocols	UDP | TCP	UDP | TCP	UDP | TCP | SCTP
TLS/TCP/IP	✓	✓	✓
DTLS/UDP/IP	✘	✓	✓
DTLS/SCTP/UDP/IP	✘	✘	✓
Protocol selection by Selection Properties	✓	✘	✓
Transport protocol racing	✓	✗	✓
Message framers	✓	✓	✓
Message Context / Properties	✗	✓	✓
Cloning	✗	✗	✓
Rendezvous	✗	✓	✘
Connection Properties	✗	✓	✓

The two implementations described above, together with NEATPy, are all the known implementations of the interface worked out by the TAPS WG. Figure 3.1 displays a comparison of the three implementations. Core mechanics like listening/initiating connections or simple sending/receiving of messages is omitted, being present all implementations. Below is a rundown of each of the implementations:

PyTAPS

PyTAPS provides TCP and UDP, as well as TLS on top of TCP. Beyond the interface specification, PyTAPS offers Source-Specific Multicast (SSM) [19], which differs from the "original" multicast service model defined in RFC 1112 [10] in that the recipient specifies which host it wants to receive datagrams from. This has the benefit of relieving the network of both source discovery and routing. At the time of writing PyTAPS lacks the following core interface mechanics:

- Message Properties and Message Context
- Connection Properties
- Connection cloning
- Rendezvous
- Cloning

PyTAPS is, however, a work in progress, and aims to be fully TAPS-conformant.

Network.framework

Apple's implementation does not have the opportunity to specify abstract requirements needed for the transportation of data, which could be used for selection of a protocol satisfying said requirements. However, the user has some ways to indicate preferences, though, not in a protocol-agnostic way, but rather preferences tied to a specific protocol. E.g. UDP is modeled as a class, which has the option `preferNoChecksum`. In TAPS, this is an abstract, protocol-agnostic preference among the Selection Properties, explained earlier. Network.Framework does not provide the opportunity to clone a connection. Connection Properties, in some way, like the Selection Properties, are tied to a protocol in Network.Framework, e.g. the TCP class has the option `connectionTimeout`, which is among the Connection Properties in TAPS. One could argue that this is fine, as this property is only applicable to TCP, but it also defeats the purpose of an interface that is agnostic to network protocols and therefore hinders the evolution of new transports. Nevertheless, some properties can be queried on a connection and function just like Connection Properties in TAPS, just not explicitly labeled as Connection Properties. Examples of this are the ability to call `maximumDatagramSize` or `metadata` on a connection. This is partly what is returned by `connection.GetProperties` in TAPS.

Network.Framework is different compared to the other two implementations in that it is an interface that is used by thousands of developers and in terms of share size and services. Examples of additional services provided by the API include the possibility for developers to get highly detailed connection metrics and the ability to create connections using WebSocket. A reason why Network.Framework isn't fully up to date with TAPS is likely due to the nature of its being; major changes to the API are not trivial when it is being used in production all over the world.

NEATPy

There are two reasons behind why our implementation is supporting all the major mechanics in TAPS: (1.) NEAT was developed, from scratch, at the same time TAPS was created and worked on. Many of NEAT's developers were actively contributing to TAPS. (2.) One goal of this thesis was to bridge the

gap between NEAT and the updated TAPS interface. Consequently this is much of the reason why this implementation features all objects and core mechanics specified in TAPS.

In the next part, we present the NEATPy in full.

Part II

NEATPy Development

Chapter 4

From C to Python

So far, we have taken a brief journey through the transport layer's history; from the beginning with the conception of TCP, followed by heterogeneous transportation demands due to advancements in technology and contemporary transport protocols which aim to meet said demands. Lastly, we have presented the idea of a transport system, with the ultimate goal of making the transport layer interface protocol-independent. This brings us to the main task of this thesis: to create a TAPS-conforming API by extending the NEAT library with an API written in Python. In this chapter, we justify the choice of Python as the implementation language and describe the language bindings created by Simplified Wrapper and Interface Generator (SWIG) [40].

4.1 On the Choice of Python

As already stated, the reasoning behind the efforts of both the TAPS WG and NEAT is to break free from the static state the transport layer has been in for years, and transition to a dynamic and flexible transport layer. However, this transition is to benefit the developers. To ignite the spark of revolutionizing the transport layer, it is, therefore, an incentive to reach as many developers as possible. Achieving this goal requires making the system usable with at least one of the most popular programming languages.

There are several reasons which make Python a suitable choice for implementing the API:

- **Existing Python bindings**: During the NEAT project's lifetime, experimental work on developing Python bindings was carried out. This makes it possible to create a conformant API with Python, and at the same time, utilize all the machinery and power of the NEAT library.
- **Object orientation**: The API described by the TAPS WG uses an object-oriented approach. Python itself supports multiple programming paradigms but has an object-oriented approach in its core. To implement the API in such a language makes it easier to map from the abstract interface described.
- **Availability and popularity**: Python as a programming language is ubiquitous and is the major programming-language with the fastest-growing rate among developers in the world [32]. This ubiquity is beneficial, as it makes the transport system available to many more developers, compared to the NEAT library written in C programming language. Being implemented in Python, it could even provide some educational value, as Python is often used as an introductory

programming language to students. Students could be introduced to the transport layer earlier than before, which could spark an interest in the transport layer.

- **Portability**: The Python interpreter is available for every major operating system and enables the developer to only write an application once. As of today, NEAT supports Linux, OS X, NetBSD, and FreeBSD. Python supports all of these systems.

4.2 Simplified Wrapper and Interface Generator (SWIG)

4.2.1 Introduction

SWIG is a tool that is used to generate wrapper code for interfaces written in C and C++ to several high-level programming languages, including Python. It is open-source, and with its initial release in 1996, it is considered a highly stable tool for interconnection between low-level code and scripting languages.

4.2.2 The Value of higher-level Bindings

NEAT is written in C due to obvious reasons: the raw performance of low-level code and interfacing with the kernel/socket API (known as system programming). There are, however, some things that are cumbersome and tedious when writing programs in the monolithic model of the C programming language. A good example is the creation of solid user interfaces, especially graphical ones. The steps of (1.) making changes to the code, (2.) recompiling the program, and (3.) testing the program make for an inflexible developing experience. SWIG was created to deal with this very problem. The main idea is separations of concerns: with the C codebase providing high-performance code, and at the other end, a solid, flexible interface written in a scripting language.

With the interface generated by SWIG, we are able to leverage the sizeable code base that is NEAT and use it to create a standard-conformant interface written in Python. In this way, the transport system could logically be divided into two logical partitions; the *front-end* being the Python interface, and NEAT functioning as a *back-end*.

4.2.3 How are the Bindings Created?

The SWIG Interface File

As input SWIG accepts an interface file, which mainly contains the two following components:

- **C-declarations and prototypes**: i.e. all the functions and variables to generate wrapper code for. The most common way to provide this is to pass header file(s), but one could also provide the C-declarations within the interface file itself.
- **SWIG directives**: These are extensions to the C/C++ standards that are processed by the SWIG preprocessor (which itself is an enhanced version of the C preprocessor) when generating the wrapper module. These directives range from very simple directives, like providing a name for the module, to more complex, like the inclusion of SWIG library modules. Library modules are

support files providing several different utility functions. An example is the *cpointer* module, providing macros to generate wrappers around C pointers and arrays.

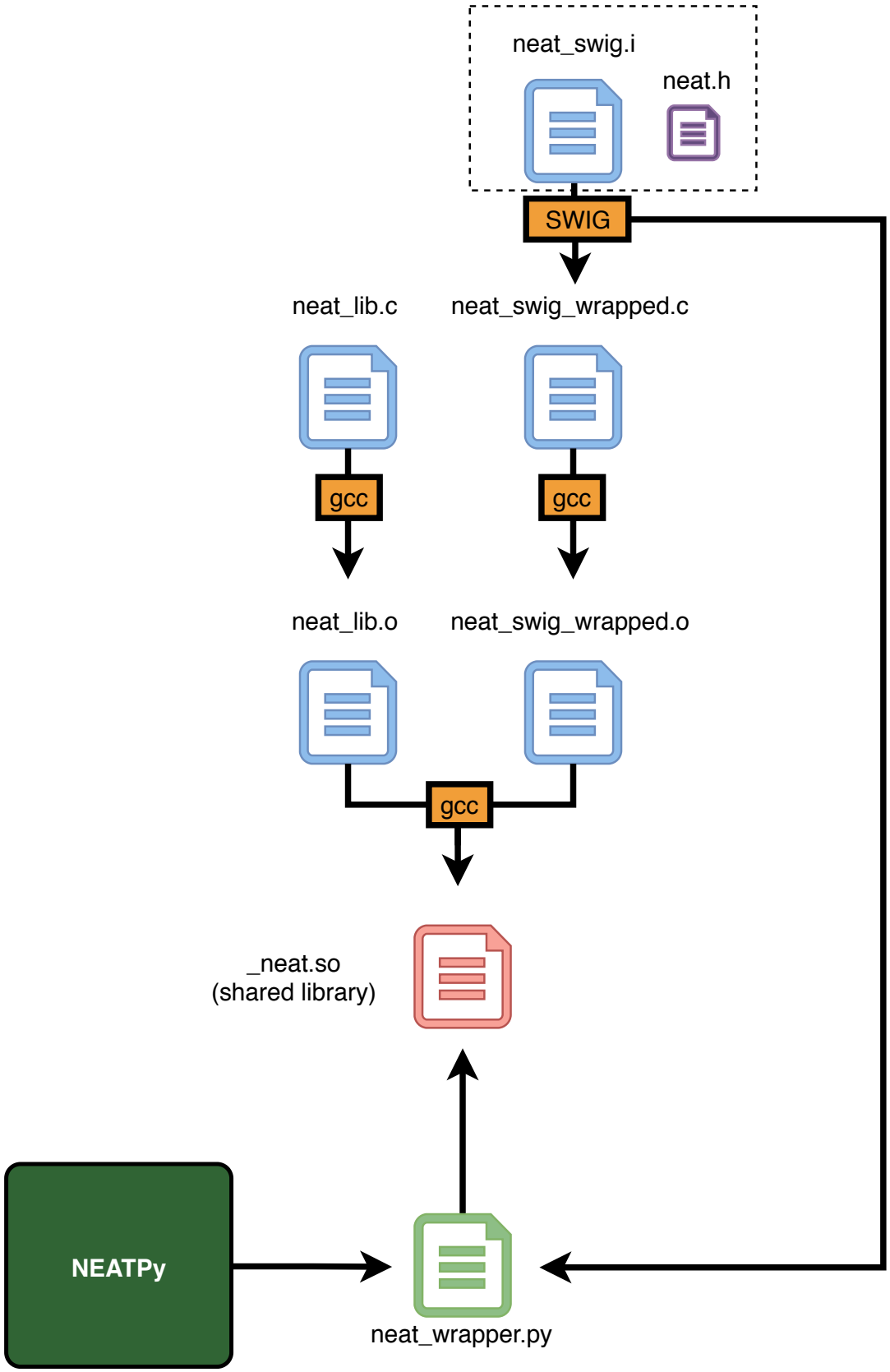
Compilation and Wrapper Code Generation

Once the interface file is written and contains the necessary information, it is passed to SWIG for compilation. In addition to the interface file, the desired target language is provided:

```
swig -python neat_swig.i
```

The output of the compilation consists of two artifacts: (1.) a C-file containing wrapper code for the C-declarations provided, and (2.) a Python module. This is followed by a second compilation, where the generated C wrapper is compiled together with the original C codebase, resulting in a shared library. The previously generated Python module provides access to the shared object. Lastly, the generated module is imported by the shim. The process as a whole is presented in Figure 4.1.

Figure 4.1: The process of wrapping a C library with SWIG



Example

Let us get a better understanding of the convenience SWIG provides by looking at an example of how a single function in NEAT is wrapped. Take the following NEAT declaration, which declares the function of creating a new flow:

```
1  // Function declaration in the NEAT header file
2  struct neat_flow *neat_new_flow(struct neat_ctx *ctx);
```

This function declaration is then provided to SWIG, together with the rest of the header file. The generated wrapper code for the function above is the following:

```
1  // Description of methods (used by Python module when accessing the shared library)
2  static PyMethodDef SwigMethods[] = {
3     {"neat_new_flow", _wrap_neat_new_flow, METH_O, NULL},
4     // ... rest of the methods wrapped
5  }
6
7  // Generated C wrapper code
8  SWIGINTERN PyObject *_wrap_neat_new_flow(PyObject *SWIGUNUSEDPARM(self), PyObject *args) {
9    PyObject *resultobj = 0;
10   struct neat_ctx *arg1 = (struct neat_ctx *) 0;
11   void *argp1 = 0;
12   int res1 = 0;
13   PyObject *swig_obj[1] ;
14   struct neat_flow *result = 0;
15
16   if (!args) SWIG_fail;
17   swig_obj[0] = args;
18   res1 = SWIG_ConvertPtr(swig_obj[0], &argp1,SWIGTYPE_p_neat_ctx, 0| 0);
19   if (!SWIG_IsOK(res1)) {
20     SWIG_exception_fail(SWIG_ArgError(res1), "in method '" "neat_new_flow" "', argument "
         "1"" of type '" "struct neat_ctx *""'");
21   }
22   arg1 = (struct neat_ctx *)(argp1);
23   result = (struct neat_flow *)neat_new_flow(arg1);
24   resultobj = SWIG_NewPointerObj(SWIG_as_voidptr(result), SWIGTYPE_p_neat_flow, 0| 0);
25   return resultobj;
26 fail:
27   return NULL;
28 }
```

Note that this is just wrapper code necessary to translate passed Python arguments to proper C code. The call to the canonical C function is made at line 23 above. To be able to call the NEAT function, the wrapper code is compiled together with the NEAT source code. The generated Python module contains the following code for the given function:

```
1  # Generated Python module
2  import _neat
3  def neat_new_flow(ctx):
4     return _neat.neat_new_flow(ctx) # calling the shared library
```

The Python module is pretty simple and is carries out two tasks: (1.) import the low-level C wrapper module, and (2.) act as an intermediary between the application using the bindings and the low-level module. The last step is the one of the application, in this case NEATPy, importing the module, utilizing

it in the creation of its interface:

```
1 import neat # importing the SWIG generated Python module
2 def neatpy_example_function():
3     # calling the module and storing the result in Python variables
4     python_context_variable = neat.neat_init_ctx()
5     python_flow_variable = neat.neat_new_flow(python_context_variable)
6     # and so on ...
```

The example described above is only for a single function declaration in the NEAT library. The result of wrapping the entire NEAT header file, containing 315 lines of declarations, is 8043 lines of low-level wrapper code. As seen, SWIG parses input arguments, converts them to native types, and in the end, calls the C function. Similarly, it converts output values to Python objects, or, like in the example above, special SWIG pointer objects, which hold references to variables/memory allocated by the low-level library.

4.2.4 Challenges / Contributions / Fixes

Detection and fix of breakable bug

When the development of NEAT ended mid 2018, the Python bindings were still considered experimental. With the NEAT codebase, Python-translated examples of a simple client and server are provided. When running the examples, however, we experienced that they did not run correctly. What we experienced was that the internal event loop in NEAT got stuck in an infinite loop. Even with NEAT's option of printing debugging information, there was no clear reason for the infinite loop. In an effort to solve the issue, the NEAT developer who worked out the bindings was consulted. Unfortunately, after running the Python examples, the developer experienced the same issue.[1] This was resolved as part of the development work underlying this thesis by simultaneously debugging in Python and C, with respectively the ipdb [23] and the GNU Project Debugger (GDB) [15]. The problem was due to a *TypeError* in Python. The reason for which this error got "under the radar" is that the TypeError occurred during a NEAT callback (`on_readable`), resulting in the following endless loop:

1. NEAT / libuv fires the `on_readable` callback, which is a Python function.
2. The Python function returns None due to a TypeError.
3. The socket is not read.

As the error occurs during a NEAT callback, the function simply returns `None` and control is returned to the low-level code that initiated the function call. During a normal Python program, execution would be terminated, and the error would be much easier to find. The actual reason for the error was tied to the variable named "buffer" in the following NEAT function:

```
int neat_read(struct neat_flow *flow, unsigned char *buffer...);
```

The converted Python client used a function called `create_string_buffer` in the well-known, foreign function library for Python, ctypes [9]. This however, creates a buffer of type `char *`, while the function requires a buffer of type `unsigned char *`. When fixing this, we decided to get rid of the use of ctypes, and instead used one of the previously mentioned SWIG directives, called `array_class`.

[1] Huge thanks to Andreas Fischer for his time testing this!

This a convenience function provided by SWIG, wrapping a pointer of chosen type inside a class-based interface. The utilization of the convenience function is done in two steps: (1.) in the SWIG interface file declare the use of the directive, together with the desired type and name for the function to later call in Python:

```
array_class(unsigned char, new_unsigned_char_buffer)
```

After compilation of the interface file, the function can be called and used in Python:

```
1 import neat # The generated SWIG module
2 def python_readable_callback():
3    buffer = neat.new_unsigned_char_buffer(20)
4    neat.neat_read(flow, buffer...)
```

The lesson taken from working out this error is that SWIG, in certain scenarios, will fail to translate types between the different languages. In this particular case, there are no objects in Python that can automatically be translated into the type unsigned char * in C. A different way of creating a representation of this type is then needed. One way to do this with is with the SWIG convenience function we ended up using. The lesson to be taken from this is to deploy proper exception handling by surrounding code calling SWIG with a try and except block. With this we are able to detect an `TypeError` and halt execution.

Migration to Python 3

During the implementation of the bindings, the developer ran into a bug that resulted in the bindings only supporting Python 2. The bug was a problem regarding conflicting versions of Python, with one version being used for linking and another during execution. Due to this, it was decided to hard code support for Python 2. However, as Python 2 reached end of life (EOL) on January 1st, 2020, the decision was made to consider this a bug, and make the code compliant with Python 3. Luckily, the solution was simple and could be solved by minor changes to the CMakeList-file of NEAT. The first change is the use of a CMake-module called `FindPython3`, which finds the Python 3 *libraries* and *include directories*. The second and last change is to specify in the CMakeList-file that only Python 3 is supported. The main reason for this choice is to follow best practice by restricting the use of software that no longer receives bug reports or fixes of any kind.

4.2.5 Summary

In this chapter, we have presented SWIG, a software tool that generates the language bindings used by NEATPy. Furthermore, we have looked at how the bindings are created and used, as well as additional fixes that were needed to use the bindings for the implementation of NEATPy. In the following chapter, the implementation is presented in detail.

Chapter 5

Implementation of a Conformant Interface

In this chapter, we present the main task of this thesis: the creation of a TAPS conforming application programming interface. Furthermore, an additional important aspect is to test the feasibility of making such an interface with the help of language interoperability provided by SWIG.

The implementation of the shim has mainly been worked out in four phases:

1. **Modeling**: This corresponds to the creation of Python classes that model TAPS objects, actions, and events.
2. **Mappings from TAPS to NEAT**: This involves using the language bindings to implement the aforementioned classes.
3. **New functionality**: This is functionality that is not possible to implement by simply calling functions in NEAT. The majority of new functionality is written in the Python front-end, but there are also some minor ancillary functions and variables added to the NEAT C-library where this has been favorable to implement the given functionality in Python.
4. **Proper documentation**: As the task of this thesis is to implement an API, the decision was made to construct proper documentation. The documentation is created with Sphinx [41], a documentation generator for various languages, including Python. The public API documentation is included in Appendix A.

This chapter will first go through general implementation details for the implementation, before the API is thoroughly presented in four sections:

1. **Pre-Establishment**
2. **Establishing Connections**
3. **Sending and Receiving of data**
4. **Connection Management and Connection Termination**

Each section presents a table with the TAPS objects, actions, and events that belong to that specific part. The table also displays whether or not the corresponding element maps to a NEAT construct. After the

initial table, what follows is an implementation specification of the section, going in-depth with listings and figures demonstrating logic, focusing on additional, new logic implemented.

5.1 General Implementation Details

5.1.1 A safer API with Enumerations

The interface specification has several concepts where it is mentioned that API elements are implemented as strings. One example is the representation of Transport Properties. In [46], it is stated: "the Transport Properties are referred to by property names. These names are lower case strings". As the interface states, the reason for this is to make code of different TAPS implementations look similar and allow for different components of the implementations to pass Transport Properties, or retrieve them from persistent storage.

Passing around raw strings is generally not a good solution. A function that has a string as a parameter has infinite possible values. Surely, the user has some sort of control of the provided string, but strings are easily mistyped. Strings, being an unbounded type, could make it difficult (or impossible) to limit state representations. Therefore some thought has been put into bounding these unbounded types when designing the API. The solution is quite simple; lift these unbounded types into enumerated types (enums).

5.1.2 Use of Type Hints

Python, being a dynamically typed programming language, has some implications. Variables not having a type makes room for possible errors by the application programmer, e.g., providing a function with a string, when it takes an integer. This backs up the decision to make use of enumerations.

However, as of Python 3.6, there is the possibility of declaring types, with the `typing` module:

```
def function_name(parameter: type)-> return_type:
```

Type hints does not change the behavior of the Python interpreter during run-time, however. The advantage lies in that, type-hints works as mitigation against errors done by the application programmer at design time. There are two major ways a Python application programmer can take advantage of type hints: (1.) By using them in an Integrated Developer Environment (IDE) that provides various checks, including inspecting calls to functions annotated by type hints; (2.) Using Mypy [30], which is a static type checker for Python. Mypy takes a Python script/program as input. It will then do static analyses of the program, and as output specify whether or not the given program co. Even though type hints require one of the methods above, we believe that contemporary developers are using these tools, especially IDEs. This is the basis for annotating functions in NEATPy with the help of type hints. In functions where this is deemed necessary, type checking is explicitly carried out. In these critical functions, the termination of the transport system is hindered by explicit type checking, and the application is notified about the error.

5.1.3 Avoidance of Leaky Abstractions

The fact that this API takes utilizes the NEAT library has some implications. One of them is leaky abstractions. Therefore, when implementing this interface, it was decided to make the abstractions "airtight", and to provide a clean, conformant interface that is mostly opaque to the use of NEAT behind the scenes. Most of the functions that call functions in NEAT are in the file `"backend.py"`. Even though this code is only used internally by the shim, it is considered best practice to avoid leaky abstractions when designing an API. Additionally, by having the abstraction of a "backend", it makes the API more modular. One example could be a more sophisticated transport system, where calls like `backend.read()` could choose a "transport backend", like NEAT or PyTAPS, based on transport characteristics. As David Wheeler once said: “All problems in computer science can be solved by another level of indirection”.

5.1.4 Settling on the Nature of Event Handlers

Event handlers are a critical part of the API, as they enable applications to react to completion of the various events as well as errors. However, to provide a flexible API, it is crucial to allow applications to ignore some events. The specific mechanism for how events handlers are registered and called is left out for the implementations. The interface draft [46] adds, however, that "applications do not need to handle all events, and that some events *may* have implementation-specific default handlers". This adds quite a lot of freedom for implementors of the transport system.

Table 5.1 presents an overview of the enumeration of events, how their respective handlers are registered within this implementation, and if there is any default handling if no event handler is registered. There are several possible ways to implement this. One way could be to simply use one variable/member for each event handler and a simple setter mechanism for the application to register a handler. This would have to be repeated for every object/class that has events. Another approach is to categorize the events in two groups: (1. state events) events that signal a state change, and (2. completion events) events that are a response to an action initiated by the application. This is the approach used in Network.Framework. Apple solves this elegantly with `stateUpdateHandlers`[1] for both Connections and Listeners, and optional completion handlers for actions that would either result in an error or completion. The implementation of the registration of event handlers in this interface is influenced by how this is done in Network.Framework. Given the vast number of events in the interface, the separation of concerns fits well. Secondly, it is more natural for a programmer to pass a (completion) handler when initiating an action than registering.

[1] https://developer.apple.com/documentation/network/nwconnection/2998577-stateupdatehandler

Table 5.1: Event handlers and how they are registered

Event	How is the handler registered?
Preconnection events	
Rendezvous done	Function argument to Rendezvous
Rendezvous error	Function argument to Rendezvous
Listen events	
Connection received	Member of Listener class
Stopped	Member of Listener class
Listen error	Member of Listener class
Connection events	
Ready	Member of Connection class
Connection error	Member of Connection class
Initiate error	Member of Connection class
Closed	Member of Connection class
Sent	Function argument to Send
Send error	Function argument to Send
Received	Function argument to Receive
Received partial	Function argument to Receive
Receive error	Function argument to Receive
Clone ready	Function argument to Clone
Clone error	Function argument to Clone

5.1.5 Overall Control Flow & Logic Handling NEAT Callbacks

Given the nature of NEAT's event loop, the control flow in NEATPy can logically be divided in two phases:

1. **Pre event loop**: This fits well with the pre-establishment phase in TAPS: the application specifies requirements for the transport, and the phase is concluded when the application either initiates or listens for a Connection.

Table 5.2: Python methods handling NEAT callbacks

NEAT callback	NEATPy method	NEATPy class
on_connected	handle_connected	`Preconnection / Listener`
on_readable	handle_readable	`Connection`
on_writable	handle_writable	`Connection`
on_all_written	handle_all_written	`Connection`
on_aborted	handle_aborted	`Connection`
on_close	handle_closed	`Connection`
on_error	handle_error	All of the above

2. **Post event loop**: The call `neat_start_event_loop` will start the libuv event loop, and further interfacing with NEAT is achieved through its callbacks.

As a direct consequence of this, a considerable amount of critical logic in NEATPy is implemented around the NEAT callbacks and the Python methods that handle them.[2] Table 5.2 presents the various class methods that handle the corresponding callbacks. The implementation details for these methods are outlined throughout this chapter. In the same way as NEATPy communicates with NEAT thorough its callback, applications must interface with NEATPy thorough the various events, following the start of the event loop. The libuv event loop in NEAT has the possibility of only being run once, and then returning. When designing NEATPy, this way of using the event loop was discarded, as an additional event loop in Python would be needed. We argue that this makes for a cleaner API, and accommodates the event-based TAPS API well.

5.2 Pre-establishment

Table 5.3: Pre-establishment mapping

Type	TAPS	NEAT
Object	Endpoints	✗
Object	Preconnection	✗
Object	Transport Properties	✗
Object	Security Parameters	Security Property

[2] The concept of a method in Python is functions belonging to a class or instances of it.

5.2.1 Endpoints

The Endpoint classes consist of the `LocalEndpoint` and `RemoteEndpoint`. NEAT does not model an endpoint, instead it takes strings containing addresses or hostnames as arguments to its functions. For the implementation of Endpoints in NEATPy, only a simple Python class was needed. As seen in Figure 5.1 the classes are small, only holding a few variables. Endpoints are used when creating a Preconnection, presented next.

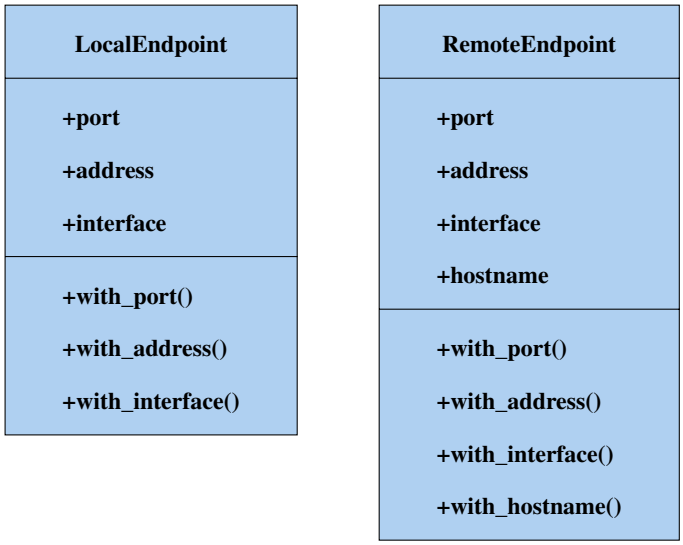


Figure 5.1: The Endpoint classes

5.2.2 Transport Properties

As described back in Section 2.3.1, Transport Properties is the way an application declares its preferences for operation. A Transport Properties object acts as an overall wrapper object for Selection Properties, Message Properties and Connection Properties. In combination, these properties are essential in the whole lifespan of the transport system. The implemented shim follows the interface specification closely. Transport Properties are implemented as a single class; the same goes for the three sub-categories, all implemented as separate classes. This composition is shown in Figure 5.2.

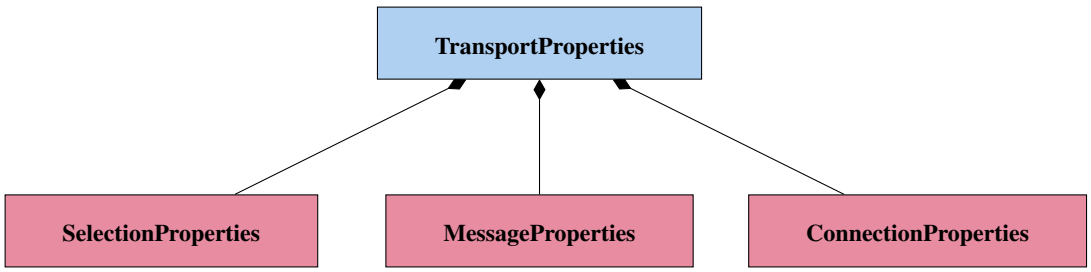


Figure 5.2: Composition of the Transport Properties

The three sub-categories of properties are implemented as enumerations. Each member of the enum has its name as specified in the implementation draft as value. The motivation behind this is to make the code in different implementations of a transport system similar. In Listing 5.1, part of the Selection

Properties class is shown. The Message Properties and Connection Properties class are implemented with the same approach.

```
class SelectionProperties(Enum):
   RELIABILITY = 'reliability'
   PRESERVE_MSG_BOUNDARIES = 'preserve-msg-boundaries'
   PER_MSG_RELIABILITY = 'per-msg-reliability'
   # ... rest of the properties
```

Listing 5.1: The SelectionProperties class

The three property classes share another implementation detail: the mechanism for obtaining their default property value. This is implemented by a simple function within each enum class. This function takes an optional parameter, `property`. If the property is specified, its default value is returned; otherwise, a dictionary of all the defaults is returned. The latter is used for convenience when initializing a new TransportProperties object. Listing 5.2 shows the function for retrieving a default value for the `MessageProperties` class. The example is shortened due to repetitiveness, but a full specification of all defaults and properties are presented in the aforementioned public API documentation.

```
def get_default(property: MessageProperties = None) -> None:
      defaults = {
         MessageProperties.LIFETIME: math.inf,
         MessageProperties.PRIORITY: 100,
         MessageProperties.ORDERED: True,
         MessageProperties.IDEMPOTENT: False,
         # ... rest of the message properties
      }

      if prop:
         return defaults[property]
      else:
         return defaults
```

Listing 5.2: Retrieving property defaults

The NEAT Policy Manager

When selecting candidate protocol stacks, NEAT uses the policy manager heavily. As stated before, the policy manager operates on policies that match on one or more properties. The downside with this approach is that the policy manager operates on profiles and policies, which need to be configured and populated. This configuration is done through separate files and requires additional set-up by potential users. Additionally, the Policy Manager needs to be started as a separate process. Additionally, NEAT does not offer the same properties as the Selection Properties in TAPS, nor the same requirement levels as specified in Listing 5.3. A goal with the API, in general, is ease of use. A choice was therefore made to implement the ranking decision (outlined in Algorithm 1 on page 47) in NEATPy. NEAT offers a direct choice of protocol(s) with the "transport" property. This enables us to carry out the candidate ranking in NEATPy, and send the results to NEAT which will perform HE on the candidates, delaying each candidate according to the priority it are sent with by NEATPy.

The Policy Manager could, however, be a powerful tool, using well thought, pre-defined policies and profiles. This could serve as auxiliary support for the Python front-end. The shim is designed so that the Policy Manager is not a requirement for using the API, but could be used by aware users.

Candidate Gathering

The `SelectionProperties` class is the mechanism for which the application communicates transport requirements. Selection Properties in NEATPy are configured with the same default value, as in the interface specification, which represents a configuration that can be implemented over TCP.

As explained Section in 2.3.1, each Selection Property is specified with a preference level. This is another class that is trivially implemented as an enumeration in NEATPy:

```
1 class PreferenceLevel(Enum):
2    REQUIRE = 2
3    PREFER = 1
4    IGNORE = 0
5    AVOID = -1
6    PROHIBIT = -2
```

Listing 5.3: The PreferenceLevel class

Again, enumeration provides us with more safety, in that the types are bound and is limited to one of the members in the enumeration. A Selection Property can then trivially be added by an application using the two enumerations like so:

```
transport_properties.add(SelectionProperties.RELIABILITY, PreferenceLevel.REQUIRE)
```

To decide on valid candidates NEATPy introduces the `PreferenceLevel` enum, together with an enumeration representing the degree of support a protocol provides for a given Selection Property. The enumeration is coined `ServiceLevel`:

```
1 class ServiceLevel(Enum):
2    INTRINSIC_SERVICE = 2
3    OPTIONAL = 0
4    NOT_PROVIDED = -2
```

Listing 5.4: The ServiceLevel class

Each of the supported protocols is modelled as an enumeration as well:

```
1 class SupportedProtocols(Enum):
2    UDP = 1
3    UDPLITE = 2
4    TCP = 3
5    MPTCP = 4
6    SCTP = 5
```

Listing 5.5: The SupportedProtocols class

Each of the protocols is mapped with respect to its "service level" within a dictionary:

```
1 protocols_mapped = {
2          SupportedProtocolStacks.TCP: {
3             SelectionProperties.RELIABILITY: ServiceLevel.INTRINSIC_SERVICE,
4             SelectionProperties.PRESERVE_MSG_BOUNDARIES: ServiceLevel.NOT_PROVIDED,
5             # and so on ...
6          }
7           SupportedProtocolStacks.SCTP: {
8             SelectionProperties.RELIABILITY: ServiceLevel.INTRINSIC_SERVICE,
```

```
9               SelectionProperties.PRESERVE_MSG_BOUNDARIES: ServiceLevel.INTRINSIC_SERVICE
10          }
11          # and so on ...
12  }
```

Listing 5.6: Mapped protocols with regard to service level

With the help of these enumerations the algorithm for generating candidates is implemented as follows:

Algorithm 1 Generating candidate protocols

```
1:  for all SelectionProperties do
2:      filter_protocols(ServiceLevel.INTRINSIC_SERVICE, PreferenceLevel.PROHIBIT)
3:      filter_protocols(ServiceLevel.NOT_PROVIDED, PreferenceLevel.REQUIRE)
4:  end for
5:  if candidates == None then
6:      return None
7:  else
8:      if candidates > 1 then
9:          rank_protocols(PreferenceLevel.PREFER)
10:         if same rank then
11:             rank_protocols(PreferenceLevel.AVOID)
12:             return candidates
13:         end if
14:     else
15:         return candidates
16:     end if
17: end if
```

This follows the description in the interface draft closely. First, we iterate through Selection Properties that are specified with a `PROHIBIT` preference and filter out the protocols which have this property's feature as an intrinsic service. The same applies when filtering for Selection Properties set with a preference of `REQUIRE`. The function `filter_protocols()` is a helper function that works on the current list of candidates, and takes as arguments the `PreferenceLevel` to check for, and the `ServiceLevel` that would lead to a candidate removal. The result is a ranked list of instances of `SupportedProtocols`. Following the generated candidate list, the final step is to translate this list into the NEAT property, `transport`, which is represented by a JSON-string. This is handled by the function `pass_candidates_to_back_end` in the backend module:

```
1  def pass_candidates_to_back_end(candidates, context, flow):
2      if len(candidates) is 1:
3          candiates_to_backend = json.dumps({"transport": {"value": candidates.pop(0).name,
               "precedence": 1}})
4      else:
5          candiates_to_backend = json.dumps(
6              {"transport": {"value": [candidate.name for candidate in candidates],
                   "precedence": 2}})
7      neat_set_property(context, flow, candiates_to_backend)
```

Listing 5.7: Passing protocols candidates to NEAT

Implemented Convenience Functions

NEATPy implements both convenience functions specified in the interface draft's [46]. The first being a series of simple helper functions for adding preferences to `Selection Properties`. The following one-liner functions are added:

- `TransportProperties.Require(property)`
- `TransportProperties.Prefer(property)`
- `TransportProperties.Ignore(property)`
- `TransportProperties.Avoid(property)`
- `TransportProperties.Prohibit(property)`
- `TransportProperties.Default(property)`

This is simply a shorthand. The example provided earlier:

```
transport_properties.add(SelectionProperties.RELIABILITY, PreferenceLevel.REQUIRE)
```

Translates to:

```
transport_properties.require(SelectionProperties.RELIABILITY)
```

The second and last convenience functions is that of `Transport Property Profiles`. These are meant as a construct to ease the use of the interface, representing a set of frequently used properties. They are implemented as an enumeration:

```
1  class TransportPropertyProfiles(Enum):
2     """ Transport property profiles are used as a mechanism to pre-configure
          :py:class:‘transport_properties‘ objects, with frequently used sets of properties.
3     """
4
5     # This profile provides reliable, in-order transport service with congestion control.
          An example of a protocol that provides this service is TCP.
6     RELIABLE_INORDER_STREAM = auto()
7     # This profile provides message-preserving, reliable, in-order transport service with
          congestion control. An example of a protocol that provides this service is SCTP.
8     RELIABLE_MESSAGE = auto()
9     # This profile provides an unreliable datagram transport service. An example of a
          protocol that provides this service is UDP.
10    UNRELIABLE_DATAGRAM = auto()
```

Listing 5.8: The TransportPropertyProfiles class

The application can simply optionally specify a `TransportPropertyProfile` during creation of a `TransportProperties` object. If specified, this object' properties are updated:

```
1  class TransportProperties:
2
3     def __init__(self, property_profile: TransportPropertyProfiles = None):
4        """ Transport properties is the collection of :py:class:‘message_properties‘,
              :py:class:‘selection_properties‘ and :py:class:‘connection_properties‘.
5
6        :param property_profile: Transport property profile to use
7        """
```

```
8       self.selection_properties = SelectionProperties.get_default()
9       self.message_properties = MessageProperties.get_default()
10      self.connection_properties = ConnectionProperties.get_default()
11
12      # Updates the selection properties dict with values from the transport profile
13      if property_profile:
14         if property_profile is TransportPropertyProfiles.RELIABLE_INORDER_STREAM:
15            self.selection_properties.update(
16               {SelectionProperties.RELIABILITY: PreferenceLevel.REQUIRE,
17                SelectionProperties.PRESERVE_ORDER: PreferenceLevel.REQUIRE,
18                SelectionProperties.CONGESTION_CONTROL: PreferenceLevel.REQUIRE,
19                SelectionProperties.PRESERVE_MSG_BOUNDARIES: PreferenceLevel.PROHIBIT
20                })
21         # and so on ...
```

Listing 5.9: TransportPropertyProfiles used during initialization

5.2.3 Preconnection

The Preconnection is an object holding information needed to select both a transport protocol and path. Its signature is the following:

```
class Preconnection(local_endpoint: LocalEnpoint = None, remote_endpoint: RemoteEndpoint =
  None, transport_properties: TransportProperties = None, secure_connection: bool = None,
                      unfulfilled_handler: Callable[[], None] = None)
```

An overview of the class member and functions are given in Figure 5.3. Both local and -remote endpoints are optional parameters, as a Preconnection is used for both listen() and initiate(). However, an error will occur if either listen or initiate are being called without the needed endpoint type. The actions of initiate(), listen() and rendezvous() are described in Section 5.3, covering connection establishment.

5.2.4 Security

NEAT has implemented end-to-end transport security, but there are little to no possibilities for applications to specify security further than toggling it on and off. Applications using NEAT have only two boolean properties, `security` and `verification`, to tell NEAT if security is needed and whether or not verification is required. As the NEAT authors document in [26], there was a lot of work left in implementation of the Security component of the NEAT system, e.g., a list of Certification Authorities (CA) is hardcoded within a string. During the implementation of NEATPy we experienced quite some trouble with the Security Component, specifically with certificate verification. However, we wanted the transport system to provide the most basic of security services. This need is realized with a simple call to NEAT:

```
1  def register_security() -> None:
2     sec = json.dumps({"security": {"value": True, "precedence": 2}})
3     neat_set_property(self.__context, self.__flow, sec)
4     ver = json.dumps({"verification": {"value": False, "precedence": 2}})
5     neat_set_property(self.__context, self.__flow, ver)
```

Listing 5.10: Passing Security Properties to NEAT

With this NEATPy is able to provide the following security stacks:

- TLS/TCP/IP
- DTLS/UDP/IP
- DTLS/SCTP/IP[3]

NEATPy models the `SecurityParameteres` class as specified, but the functions, as described in the interface draft, remain unimplemented due to the limitations within NEAT. Due to this, the way an application is signaling it wants a secure connection is simply a boolean when creating a `Preconnection`. This is an area of the transport system that would need further work, most likely by extending the Security component within NEAT. We will touch upon this again in Section 8.2, where we discuss future work.

NEAT Logic

During the creation of a Preconnection, the required NEAT calls are made to create a Context, Flow and Operations struct. The Preconnection conceals this with the call `backend.bootstrap_backend()`, which contains the ceremonial NEAT setup:

```
1 def bootstrap_backend() -> None:
2     ctx = neat_init_ctx()
3     flow = neat_new_flow(ctx)
4     ops = neat_flow_operations()
5     neat_log_level(ctx, NEAT_LOG_DEBUG)
6     neat_set_operations(ctx, flow, ops)
7
8     return ctx, flow, ops
```

Listing 5.11: Bootstrapping NEAT in Python

The NEAT-specific variables are private, as they only are used for logic within the transport system itself. The same goes for the private methods listed in Figure 5.3. These are functions that are registered as NEAT callbacks, as specified in Section 5.1.5.

[3] SCTP is only supported on Linux and FreeBSD.

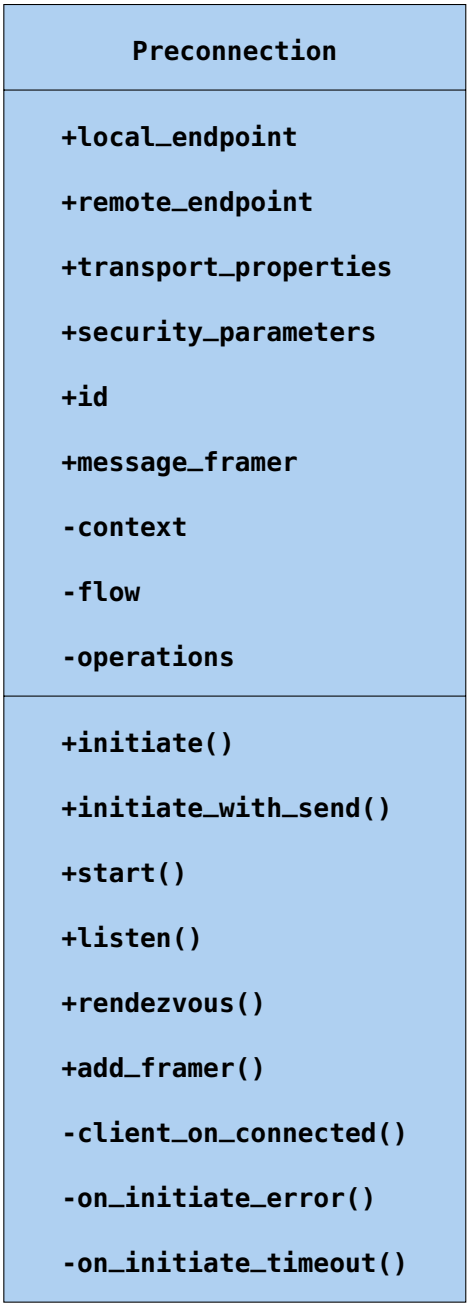


Figure 5.3: The Preconnection class

5.3 Establishing Connections

Table 5.4: Establishment mapping

Type	TAPS	NEAT
Action	Initiate	neat_open
Action	Initiate_with_send	neat_open
Action	Listen	neat_accept
Action	Rendezvous	✗
Action	Cloning	✗
Event	Ready	on_connection_ready

5.3.1 Passive Open with Listen

A Listener object is created when calling `preconnection.listen()`. It follows the interface specification, and consist of the following implementation details:

- **NEAT logic**: During initialization of a Listener object, the call to `neat_accept()` is done. As briefly explained in Section 2.4.3, `neat_accept()` bears the same semantic meaning, as it starts listening for connections.

- **Deep copy of properties**: NEATPy follows the implementation draft [5] by making sure the Transport Properties passed to the Listener (and Connection) object are immutable. This is realized by the function `copy()` in the Python Standard Library module `copy`. A deep copy makes sure to create a new object and recursively copy the original object's values. This ensures immutability in that *values* are copied and not simply references.

- **Rate-limiting the number of connections**: This is realized through the variables `number_of_connections` and `connection_limit`. By default, the connection limit is set to the infinite value `math.inf`, but can be set with the function `set_new_connection_limit()`, or reset by `reset_connection_limit()`. The variable `number_of_connections` is then used to monitor the number of active inbound connections, denying additional incoming connections, if the limit is reached.

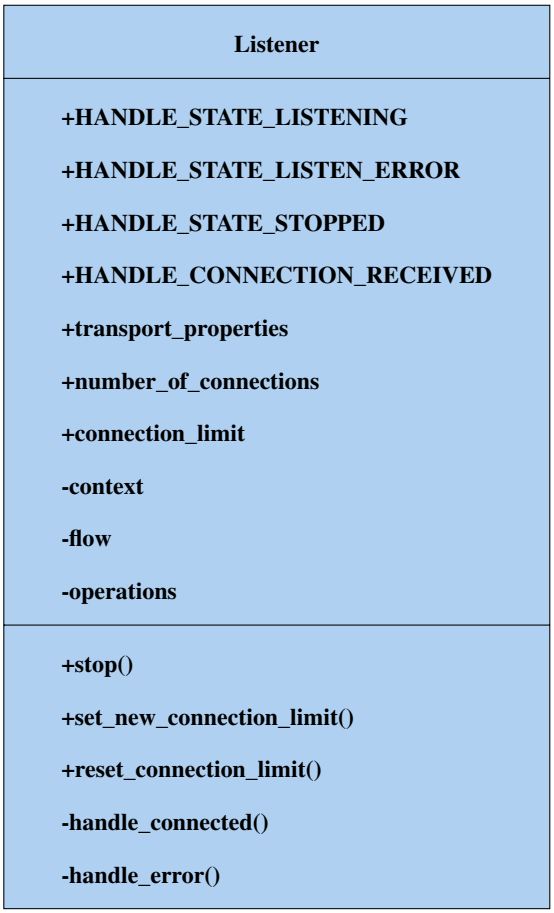


Figure 5.4: The Listener class

5.3.2 Active Open with Initiate

The action of actively initiating a connection is part of the Preconnection class. The following logic is carried out in `initiate`:

1. Check to validate that a `RemoteEndpoint` is registered with the `Preconnection`. If not, an error is returned.
2. The candidate generation function carried out by the `TransportProperties` class, explained at the beginning of this section, is carried out. If this results in an empty candidate list, the `unfulfilled handler` is fired. If the application has not provided a handler, the transport system is shut down.
3. The candidates are passed to NEAT, with the call `neat_set_property`. What follows is the call `backend.initiate()`, which again calls the NEAT function `neat_open()` (described in Section 2.4.3).
4. If the optional `timeout` parameter is provided, a timer is started. The implementation details regarding this timer are provided below.
5. A `Connection` object is created and returned to the application.

6. An active open is fulfilled after the application has called `preconnection.start()` and NEAT fires its `on_connected()` callback, which is handled by the function `preconnection.client_on_connected()`.

Initiate Timeout

As seen in the table above, initiate (active open) in TAPS corresponds to `neat_open` in NEAT. Three scenarios can lead to an *initiate error*:

1. The properties set on the Preconnection make for no suitable connection candidate (this could mean that there are no suitable protocol stacks and/or no paths available)
2. The remote specifier can not be resolved, e.g. an incorrect URL was passed.
3. The optional timeout passed with initiate() has expired.

(1.) is handled solely in the Python front-end as Selection Properties are processed before starting the NEAT event loop, and therefore, if the properties resolve to no supported protocol stack, an initiate error occurs. When the remote specifier cannot be resolved, NEAT will call the 'on_error' callback. To translate this to an initiate error, we need to create a function in Python handling the callback, fire off the initiate error event, and register said function in the neat operations struct. For the last scenario, however, NEAT does not provide an option to specify a timeout for the connection attempt. A mechanism to translate the optional timeout parameter for the initiate-call in TAPS was therefore needed.

A timer implemented purely in Python is not possible due to the event loop running in NEAT, which is called from Python, and therefore holding control of the Python Global Interpreter Lock (GIL). A better approach is to hook into the libuv event loop running in NEAT. In addition to using the event loop for pollable sockets, NEAT uses it for various asynchronous tasks such as DNS resolution.

Initiate With Send

The function `initiate_with_send()` works as a convenience function for scenarios where the client is sending the first message. The shim implementation simply calls `initiate()`, followed by `send()` called on the `Connection`. This function could also make use of 0-RTT establishment techniques as `TCP Fast Open` (TFO) or SCTP's `COOKIE ECHO`. NEATPy does not support this, as they are not implemented in the low-level NEAT library.

The Start Method

NEATPy features the method `start()` as part of the Preconnection's interface. This is a deviation from the interface specified by TAPS. The method is added to implement the functionality of the methods `initiate()` and `listen()`, which respectively return a Connection and Listener. To start NEAT and its event-loop, we need to call `neat_start_event_loop()`. This function does not return, which is the reason for this addition to NEATPy's interface. We thought of two ways for NEATPy to hand over control to NEAT:

1. Calling `initiate()` or `listen()` would call appropriate calls in NEAT and *immediately* start the NEAT event-loop, not returning a Connection or Listener object, like specified in the interface draft.

2. Calling `initiate()` or `listen()` would call appropriate calls in NEAT, and return one of the respective objects. The application would then be free to operate on the returned object. To start the transport system the application would have to call `preconnection.start()`.

We chose the latter, the reason being the addition of one method trumps the removal of functionality outlined by TAPS. By not returning said objects, applications would not have the opportunity to carry out operations like registering event handlers. We observe that Apple's Network.framework also features a start method for its Connection and Listener objects.[4]

5.3.3 The Connection Class

Given the nature and responsibility of a `Connection`, it makes for the most complex class of the implementation. The class specification given in Figure 5.5 presents the essential class members and functions. Some of the more "routine" helper functions are omitted, but touched upon, where necessary, in the following list supplementing the class diagram:

- The variables `messages_passed_to_back_end`, `receive_request_queue`, `message_sent_with_initiate` and `message_framer` are all used in sending and reception of messages, and described in detail in Section 5.4.
- The same applies to the members, `connection_group` and `parent`, which are used in the implementation of connection cloning, described in Section 5.3.3.
- Similar to the Listener class, a deep copy is done for the `TransportProperties` object, and the NEAT variables holding references to Context, Flow and Operations are passed to the Connection for use in further logic in communication with the NEAT backend.

Connection methods handling NEAT callbacks

Ideally, the methods inside the Connection class handling NEAT callbacks would be implemented as class methods; everything needed would be to register the methods as callbacks for that specific NEAT flow, and when a callback is fired, it would be fired for the Connection the flow belongs to. This is however not possible, as all class methods in Python are called with the instance whose method was called as the first parameter:[5]

```
def class_method(self, param1...)
```

Every callback in NEAT is fired with the operations struct for the given flow as the only parameter. During the transition from NEAT to NEATPy, the Python interpreter would overwrite the operations struct with the Connection class instance. One would think that this could easily be solved by having the operations struct as a member of the Connection class. During implementation, this strategy was indeed tested, but this approach was unsuccessful. After debugging the issue, it turned that the low-level variables created, like an operations struct, are moved around in memory during execution, leaving this approach unfeasible.

To get the callback flow with NEAT right, static methods are used. Static methods are bound to a class, rather than instances for a class. This removes the problem above, but leaves us with another; we need

[4] https://developer.apple.com/documentation/network/nwconnection/2998575-start
[5] The parameter name `self` is used as a convention, but any legal identifier may be used.

a mechanism to infer which Connection the given callback is for, as static methods are shared by every instance of the class. The solution is quite simple: add a variable in the low-level operations struct identifying the connection it belongs to. During initialization a Connection is assigned an id, using a static counter. Consequently, it is added to a static dictionary, mapping connection ids to connection objects:

```
1 def __init__():
2     # ...
3     # Map connection for later callbacks fired
4     Connection.static_counter += 1
5     self.connection_id = Connection.static_counter
6     Connection.connection_list[self.connection_id] = self
7     # ...
```

Figure 5.5: The Connection class

Connection Cloning

The action of cloning, and with it, using Connection Groups, is specified in the interface draft. As the goal of a transport system is to provide a more flexible API, the action of cloning is convenient in contrast to creating a new connection with the same properties as an existing connection.

Cloning a connection with an underlying protocol stack that supports multi-streaming is probably the most natural use case, as each call to clone would normally create a new stream. In these scenarios, connections are multiplexed both end-to-end within an association, and at the endpoints, in their Connection Group. In NEATPy, when cloning a connection with a protocol stack that does not provide multi-streaming, a clone call will create a new connection with identical properties as the parent (the connection on which clone is called). As the interface draft [46] states, after calling `clone()` on a Connection, these two becomes "entangled". This has two implications: (1.) calling clone on either Connection again adds a third Connection to the Connection Group; (2.) The Connections part of a Connection Group share Connection Properties; a property configuration change is mirrored onto the other Connections. This is described further in Section 5.5.1.

The following two NEAT concepts are relevant for connection cloning:

- **`NEAT_FLOW_GROUP`**: As the name suggests this forms a logical group of NEAT flows. Every flow in NEAT belongs to a flow group, which defaults to 0.
- **`NEAT_PRIORITY`**: This is the NEAT equivalent of the Connection Property, `priority` in TAPS (Connection Properties are outlined in Section 5.5.1). It specifies the priority of a given flow compared to the other flows in the same flow group. It has a value between 0.1 - 1.0.

The use of priority in NEAT was never fully implemented. In the source code, coupled congestion control is only implemented for the FreeBSD kernel, requiring a kernel patch.[6] Additionally, the priority in NEAT can only be set when opening a flow with `neat_open()`. In [26] the authors also state that prioritized transfer for SCTP was an ongoing project. Unfortunately, this was not finalized.

This makes for the following overall logic in NEATPy during the creation of a new flow:

1. **Active and Passive open**: NEATPy will create a new flow and create a new flow group in NEAT. This is done by using a counter, which is incremented for each new flow created in this way.
2. **Cloning**: The flow created for the clone is opened with passing the same flow group as the parent. If NEATPy is run on FreeBSD, the Connection Property `priority` is used and translated to the priority value scheme used in NEAT.

The implementation of cloning is described below, accompanied by simplified code in Listing 5.12:

1. Clone is called on an existing connection along with an optional event handler to handle potential clone errors.
2. Necessary set-up for NEAT is handled. This includes creating a new flow and operations struct, register callbacks - `on_connected()` for when a new connection is successfully created, and `on_error` to handle errors. Lastly, the protocol that is backing the parent connection is sent as a property to NEAT.
3. A new Connection object is created for the new clone.

[6] Available at: http://safiquli.at.ifi.uio.no/tcp-ccc/

4. The connection id is set for the cloned connection. This is used when handling the `on_connected()` callback, which is mapped to a static method in the Connection class called, `handle_clone_ready()`. The `on_connected()` callback is isolated to a dedicated Python method, as there is special logic carried out when a cloned connection has established successfully.

5. The Connection object for the clone is then returned to the application. The application is then able to interact with the Connection object, e.g., call `send()` or register a function as event handler for when the Connection is ready with the `HANDLE_STATE_READY` member of the Connection class.

6. The `handle_clone_ready()` method is called, signaling that the cloned connection has established a connection successfully. Both the Connection Group for the parent and clone are updated.

```
def clone(self, clone_error_handler: Callable[[Connection], None]) -> None
   # NEAT boilerplate
   flow = neat_new_flow(self.__context)
   ops = neat_flow_operations()

   new_connection = Connection(self.preconnection, 'active', parent=self)

   # Set connection id for the new clone in the operations struct
   ops.connection_id = new_connection.connection_id

   # Set callbacks to properly handle clone establishment / error
   ops.on_error = on_clone_error
   ops.on_connected = handle_clone_ready
   neat_set_operations(self.__context, flow, ops)

   # Asynchronously call NEAT to create a new connection
   neat_open()
   return new_connection

def handle_clone_ready(ops):
   # Retrieve parent and child connection
   child_connection = Connection.connection_list[ops.clone_id]
   parent_connection = child_connection.parent

   # Add child to parents connection group and call established routine for child
   parent_connection.add_child(child_connection)
   child_connection.established_routine(ops)

   return NEAT_OK

### Application code ###
cloned_connection = connection.clone()
cloned_connection.send("Hello from clone")
```

Listing 5.12: Cloning carried out in NEATPy

5.4 Sending and Receiving

Table 5.5: Send / Receive mapping

Type	TAPS	NEAT
Action	Send	neat_write
Event	Sent	on_all_written
Event	Expired	✗
Event	SendError	on_error
Object	Message Properties	✗
Action	Partial Sends	✗
Action	Batching Sends	✗
Action	Receive	on_readable
Event	Received	neat_read
Event	ReceivedPartial	✗
Event	ReceiveError	on_error
Object	Message Context	✗
Object	Message Framers	✗

5.4.1 Message Context

The `MessageContext` object is a small, but important object. It is used for various tasks:

1. **Identification of messages**: This is useful in scenarios like partial sends or when an error occurs.

2. **Set/get Message Properties and framing meta-data**: Examples could be to set a custom lifetime for a message or add a header for an HTTP-framer.

3. **Provide information about endpoints**: The variables `local_endpoint` and `remote_endpoint` can be queried during send and receive events with `get_remote_endpoint()` and `get_local_endpoint()`.

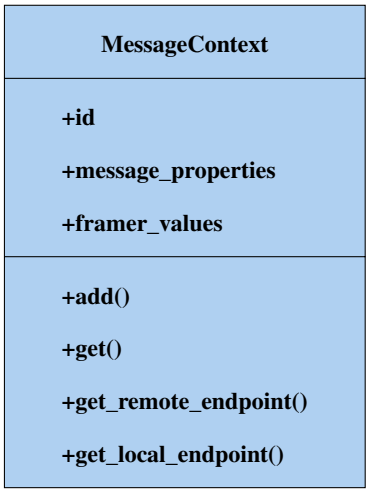


Figure 5.6: The MessageContext class

The functions for adding and querying Message Properties and framer meta-data are examples of functions where explicit type checking is done. The reasoning behind this is to hinder termination of the transport system. Blindly using duck typing in this method could lead to an error when the properties are used down the road. Checking this is ensure that we fail as early as possible, and the type assertion is just a few additional lines of code:

```
1  def add(self, key, value, scope: Framer = None) -> None:
2        # If scope is not set, try to add message property
3        if not scope:
4           if not isinstance(key, MessageProperties):
5              shim_print("Invalid message property provided - ignoring", level='error')
6           else:
7              self.props[key] = value
8        # Else a framer is given, try to add meta-data
9        else:
10          framer = scope
11          if not isinstance(framer, Framer):
12             shim_print("Provided argument is not a valid framer - ignoring",
                   level='error')
13             self.framer_values[key] = (framer, value)
```

5.4.2 Sending

The action of sending a message has the following signature in NEATPy:

```
    def send(self, message_data: bytearray, sent_handler: Callable[['Connection',
'SendErrorReason'], None] = None, message_context: MessageContext = None, end_of_message:
                                    bool = True)-> None:
```

Data Flow - Sending

Semantically, the action of sending data in the interface draft corresponds to writing data to a flow in NEAT. NEAT, however, does not have a concept of enqueueing messages passed by an application;

instead, the application' on_writable callback is fired when the flow can be written without blocking. This makes for a good fit: the additional logic of enqueuing/dequeuing messages is implemented in NEATPy, while the callback flow of NEAT is used as-is. This makes for the following flow:

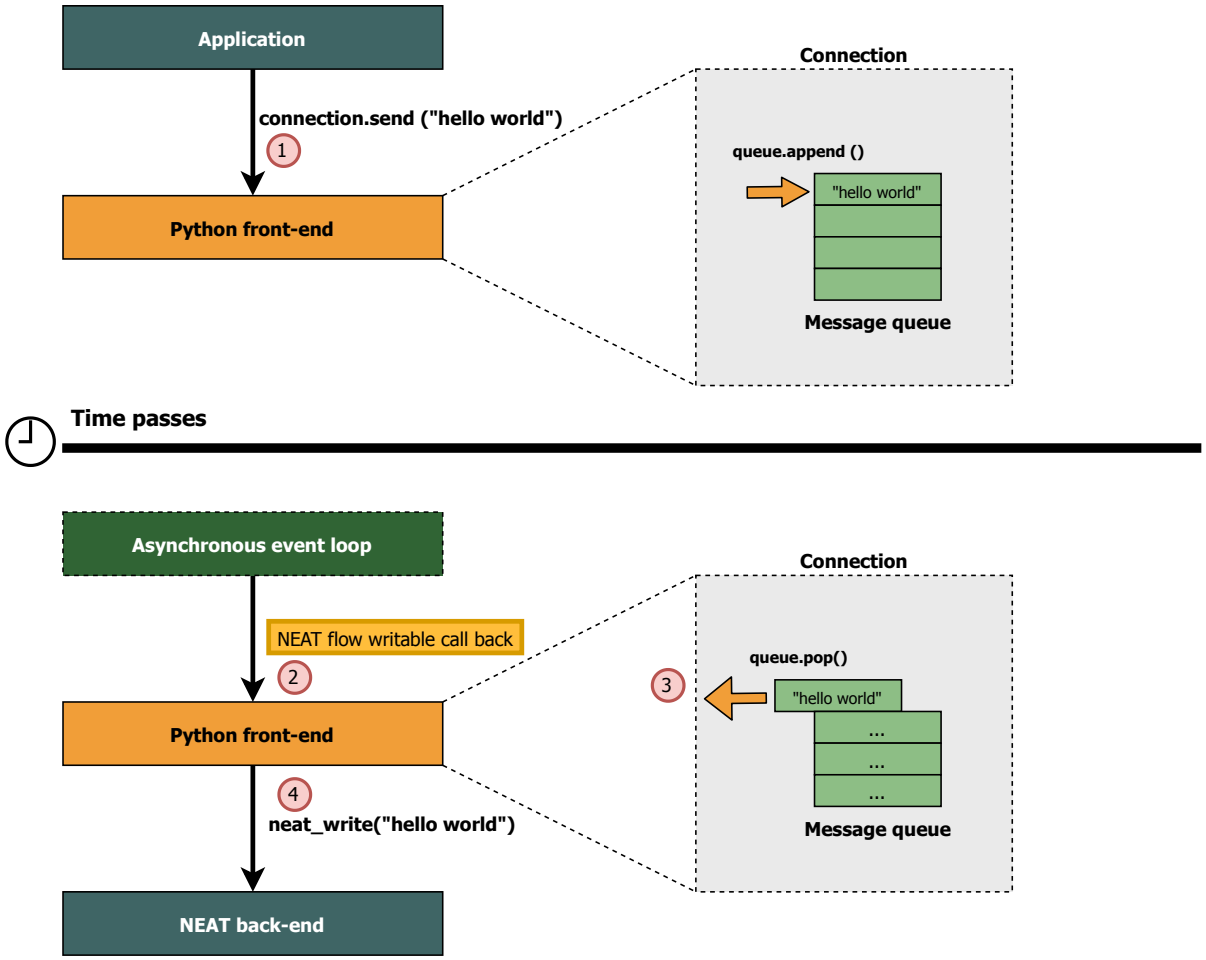


Figure 5.7: How a message gets passed to the NEAT back end

1. When calling the Send action the message data along with the Message Context and optional completion handler is stored together in a wrapper object named `MessageQueueObject` in the queue.

2. The socket is writable, and NEAT fires the `on_writable()` callback.

3. The `handle_writable()` class method in the Connection class fetches the connection matching the connection id and checks its message queue.

4. If present, the first message in the queue is dispatched with neat_write().

The implemented logic makes the send action in the NEATPy indirectly asynchronous; the send call itself will just enqueue a message in a data structure and will asynchronously be dispatched during the on_writable callback handled in Python front-end, which calls neat_write.

Additional Implementation Details

1. When the application does not provide a `MessageContext`, one is created with `MessageProperties` having their default values.
2. The mechanism of partial sends is implemented as described in the interface draft; if `end_of_message` is set to false, the bytes sent will be buffered in a dictionary with the `MessageContext` id as key. Each time send is called, this dictionary is checked.
3. An essential implementation detail is the queue in which the messages are queued. This is implemented as a priority queue. The reason behind this choice is to accommodate the Message Properties, `FINAL`, and `PRIORITY`:
   (a) If a message is marked final it is given the value `math.inf`, which effectively will place it at the end of the queue.
   (b) If the priority property is set, the priority value for the property is used as a priority value for the queue.
   (c) In all other cases, messages will be queued with the default priority value, which is 100, as specified in the interface draft.

Batch Sending

In the TAPS specification, batch sending is a convenience and -efficiency construct that allows `send()` to be called multiple times. This construct reduces overhead and context switches, which occur when the transport system is passing messages down the stack. In NEATPy, this is implemented in a very basic way:

```
1 def batch(self, batch_block: Callable[[], None]) -> None:
2       """Used to send multiple messages without the transport system dispatching messages
            further down the stack. Used to minimize overhead, and as a mechanism for the
            application to indicate that messages may be coalesced when possible.
3
4       :param batch_block: A function/block of code which calls send multiple times
5       """
6       self.batch_in_session = True
7       bacth_block()
8       self.batch_in_session = False
```

The boolean variable `batch_in_session` is used within the regular `send()` function; if a batch operation is in session, all messages sent during this period are grouped in a list. When the flag is turned off, an internal function iterates through the "batch list" and, as the specification states, groups messages into bigger messages where possible. Apple[7] presented an example during WWDC 2018, used batch sending to send frames captured by a camera, sent over UDP. In the same way, the implementation will try to group UDP messages sent during a batch into larger ones, by conforming to the Connection Property `"Maximum Message Size on Send"`.

[7] Presented by Tommy Pauly at *WWDC18*: https://developer.apple.com/videos/play/wwdc2018/715/

Expired Event

An expired event occurs if the transport system is unable to dispatch an enqueued message before its lifetime property has expired. The default lifetime value for a message is infinite, which indicates that the application has no time constraints regarding the transport of that particular message.
The implementation of the expired event in the interface is a continuation of the previously mentioned queuing logic. The following steps describe the implementation:

1. A check to see if the Message Property `LIFETIME` is less than infinity (default value):

```
if message_context.props[MessageProperties.LIFETIME] < math.inf:
```

2. If this is the case, set a variable holding the expiration time:

```
expired = int(time.time())+ message_context.props[MessageProperties.LIFETIME]
```
[8]

3. At the time of dispatching the message further down to NEAT, check to see if it is expired; if so, discard the message:

```
if expired_epoch and expired_epoch < int(time.time())
```

Error Handling

Several situations could lead to a `SendError`. Listing 5.13 presents a simplified view of the many checks in the implementation of `send` and `handle_writable` in the interface. The implementation tries to detect SendErrors as early as possible. This include checking message properties immediately after `send()` is called, as well as a check prior to dispatching the message to the NEAT back-end.

The potential send errors in the send function itself are errors that could be labeled as application errors. Sending is prohibited when either the application already has sent a message marked final, or the connection is closing (either by a signal from the peer or the application itself). Lastly, in the send function, there is a function call that checks whether the Message Properties provided with the Message Context are consistent with the features of the protocol backing the established Connection. The following table list all the inconsistencies that are checked.

Table 5.6: Inconsistencies between Message Properties and Selection Properties backing the Connection

Message Property	Protocol property
Lifetime: infinite	Reliability: not provided
Idempotent: false	Protection against duplicated messages: not provided
Reliable data transfer: true	Reliability: not provided
Ordered: true	Preserve order: not provided

[8] The time function in the Python time module returns the current time in seconds since Epoch.

Secondly, NEATPy must handle potential send errors returned by the NEAT back-end. These are handled in the function dispatching data to NEAT, `handle_writable`:

1. `NEAT_ERROR_MESSAGE_TOO_BIG` is returned when the message handled to NEAT is larger than the combined constraint of the underlying protocol and write buffer. This would be the case for the message-based protocols, UDP and SCTP.[9]
2. `NEAT_ERROR_IO` is returned by NEAT when there is an error writing to the socket, or as the interface draft specifies, *"some failure of the underlying Protocol Stack"*

```
def send(...):
   # If the connection is closed, further sending will result in an SendError
   if self.close_called:
      shim_print(f"SendError - {SendErrorReason.CONNECTION_CLOSING.value}")
      sent_handler(self, SendErrorReason.CONNECTION_CLOSING)
      return

   # If another Message is sent after a Message marked as Final has already been sent on
       a Connection the Send Action for the new Message will cause a SendError Event
   if self.final_message_passed:
      shim_print(f"Send error - {SendErrorReason.FINAL_MESSAGE_PASSED.value}",
          level='error')
      sent_handler(self, SendErrorReason.FINAL_MESSAGE_PASSED)
      return

   # Check for inconsistency between message properties and the connection's transport
       properties
   inconsistencies = self.check_message_properties(message_context.props)
   if inconsistencies:
      shim_print(f"SendError - {SendErrorReason.INCONSISTENT_PROPERTIES_PASSED.value}:",
          level='error', additional_msg=inconsistencies)
      sent_handler(self, SendErrorReason.INCONSISTENT_PROPERTIES_PASSED)

   #----------------------------------------#

def handle_writable(ops):
   neat_result = backend.write(ops, message_to_be_sent)
   # NEAT error -> call send handler with the error
   if neat_result:
      if neat_result == NEAT_ERROR_MESSAGE_TOO_BIG:
         reason = SendErrorReason.MESSAGE_TOO_LARGE
      elif neat_result == NEAT_ERROR_IO:
         reason = SendErrorReason.FAILURE_UNDERLYING_STACK
         shim_print(f"SendError - {reason.value}")
         handler(context, reason)
      return
```

Listing 5.13: Assertions during the send call

5.4.3 Receiving

To provide the application with a mechanism for flow control is of importance for a transport system. TAPS and NEAT enable the application to provide backpressure with different approaches. In NEAT,

[9] SCTP could bypass this by enabling explicit EOR: https://tools.ietf.org/html/rfc6458#section-8.1.26

the equivalent of enqueueing a receive in TAPS is to set the `on_readable` callback; both methods signal that the application is ready to receive data. Similarly, as each receive-call in TAPS ends either with an error or success event, in NEAT the application gets the on_readable callback fired when the socket is readable without blocking and calls `neat_read`. If an application in NEAT wants to apply backpressure, it could then set on_readable to NULL. In TAPS this type of backpressure is achieved by simply not calling `receive`.

The following list contains key implementation details for the receive action:

- **Interaction with NEAT on_readable callback**: Each call to receive gets enqueued into a Python list, which operates in a regular First In, First Out (FIFO) manner. NEATPy will set the `on_readable` callback to the local method `handle_readable` **if** the receive queue was empty prior to the incoming request. Similarly, if the Python list is empty after a receive request is completed, on_readable is set to `None`.
- **Usage of Message data object**: This object is specified in the interface draft and contains the data received, and the length of this byte array. It is shown in Listing 5.14.
- **Helper functions for buffer management**: Additionally, this implementation sports two convenience functions (presented in Listing 5.15), that are used for buffer management. NEATPy calls `combine_message_data_objects()` when the application has set `min_incomplete_length` to anything other than the default value, `None`. This buffers data of two `MessageDataObjects` into one. The function `partition` is used to get the `received_partial` event right. As the interface specification states, this event is fired in several scenarios:
  1. The incoming message is larger than the `max_length` specified by the application.
  2. The connection is backed by a protocol supporting message boundaries, but the message is larger than the buffers holding a single message.
  3. A message framer is used, but it cannot determine the end of the message with the buffer space it has available.
  4. The connection is backed by a protocol that does not support message boundaries (e.g. TCP), and no framer is used. This, of course, implies that all bytes received on the given Connection will be represented as one full message.

```
@dataclass()
class MessageDataObject:
    """
    The MessageDataObject provides access to the bytes that were received for a Message,
        along with the length of the byte array.
    """
    data: bytearray
    length: int
```

Listing 5.14: The MessageObject class

```
def combine_message_data_objects(self, other_object):
    self.data += other_object.data
    self.length += other_object.length

def partition(self, partition_size):
    other_partition = MessageDataObject()
```

```
7     other_partition.data = self.data[partition_size::]
8     other_partition.length = len(other_partition.data)
9
10    self.data = self.data[0:partition_size]
11    self.length = len(self.data)
12
13    return other_partition
```

Listing 5.15: Convenience functions operating on MessageDataObjects

With this in mind, the following describes the algorithm implemented, handling data that is received from the NEAT backend with the call `neat_read()`, and passed on to fulfill queued receive calls by the application:

Algorithm 2 Handling data from NEAT

```
1:  if message Framer present then
2:      framer.handle_received_data()
3:  else
4:      if connection provides message preservation then
5:          if message length > max length then
6:              partition()
7:              ➔ « received_partial event »
8:          else
9:              ➔ « received event »
10:         end if
11:     else
12:         if buffered data then
13:             combine_message_data_objects()
14:         end if
15:         if message length < min length then
16:             buffer_data()
17:         else if message length > max length then
18:             partition()
19:             ➔ « received_partial event »
20:         else
21:             ➔ « received event »
22:         end if
23:     end if
24: end if
```

5.4.4 Message Framers

The implementation of framers in NEATPy is true to the suggested approach specified by the implementation document [5]. The abstract class, `Framer`, is provided for applications to extend:

```
1  class Framer(ABC):
2
3      @abstractmethod
4      def start(self, connection):
```

```
5       """When a Message Framer generates a Start event, the framer implementation has the
            opportunity to start writing some data prior to the Connection delivering its
            Ready event. This allows the implementation to communicate control data to the
            remote endpoint that can be used to parse Messages.
6       :param connection:
7       """
8       pass
9
10  @abstractmethod
11  def stop(self, connection):
12      pass
13
14  @abstractmethod
15  def new_sent_message(self, connection, message_data, message_context, sent_handler,
        is_end_of_message):
16      """Upon receiving this event, a framer implementation is responsible for performing
            any necessary transformations and sending the resulting data back to the
            Message Framer, which will in turn send
17      """
18      pass
19
20  @abstractmethod
21  def handle_received_data(self, connection):
22      """Upon receiving this event, the framer implementation can inspect the inbound
            data. The data is parsed from a particular cursor representing the unprocessed
            data. The application requests a specific amount of data it needs to have
            available to parse. If the data is not available, the parse fails.
23      :param connection:
24      """
25      pass
```

Listing 5.16: The abstract Framer class

Additionally, NEATPy has modeled and implemented the `MessageFramer` object. This object acts as an intermediary between implemented Framers and the transport system. It carries the responsibility of delivering events, that is, both when data is to be framed with outgoing messages, or parsed, with incoming messages. The class consist of the following variables and methods:

Some details:

- An ancillary object, `FramerHelperObject`, is introduced to the `Connection` class. This variable is used for all events regarding parsing inbound data to be deframed. The implementation is presented in Listing 5.17.
- To support the use of more than one framer, the variables `framer_list` and `ongoing_transformations`, as well as the functions `append_framer()` and `prepend_framer()`, are introduced.
- The methods `dispatch_handle_received_data()` and `dispatch_new_sent_message()` are used by the `Connection` object when either data is received, or to be sent.
- The remaining methods are used by the custom Framer as an interface when parsing and framing messages:
  - The `send` method: The custom framer calls this method with the sent data successfully transformed.
  - The `parse` method: When the custom framer gets notified about received data, `parse`

Figure 5.8: The MessageFramer class

can be called with the desired number of bytes to inspect. The data is parsed from the aforementioned `buffered_framer_data_object`.

- The `deliver` method: is called by the custom framer together with a complete message.
- The `advance_receive_cursor` method: is called with the number of bytes to advance. This method is used by the custom Framer after e.g. it has parsed a header, implying that these bytes can be discarded by the transport system.
- The `deliver_and_advance_receive_cursor` method: This is a convenience method suggested by the TAPS implementation document, which is implemented by NEATPy. With this method, the custom Framer implementation can signal that the next complete message consists of `X` bytes. This eases the delivery of large messages, in that the custom framer only have to inspect, and parse the initial bytes. The result is an overall performance boost. In these scenarios, during the reception of incoming data, the `Connection` will not engage the `MessageFramer`, but instead call the method `fill_earmarked_bytes()` for the `FramerHelperObject`, as shown in Listing 5.17.

```
1 class FramerHelperObject:
2     connection: Connection
3     inbound_data: bytearray = bytearray()
4     buffered_data: bytearray = bytearray()
5     cursor: int = 0
6     earmarked_bytes_missing = 0
7     saved_message_context = None
```

```
8     saved_is_end_of_message = None
9
10    def advance(self, length):
11       self.inbound_data = self.inbound_data[length::]
12
13    def fill_earmarked_bytes(self, bytes):
14       # Are we still missing bytes for a complete message?
15       if len(bytes) < self.earmarked_bytes_missing:
16          self.buffered_data += bytes
17          self.earmarked_bytes_missing -= len(bytes)
18       else:
19          # Have we received more bytes than needed to complete the message?
20          if len(bytes) > self.earmarked_bytes_missing:
21             msg = self.buffered_data + bytes[0:self.earmarked_bytes_missing]
22             self.inbound_data += bytes
23          # ... or have we received the exact number of bytes to complete the message?
24          else:
25             msg = self.buffered_data + bytes
26          # Call the application's receive event handler
27          handler, min_length, max_length = self.connection.receive_request_queue.pop(0)
28          message_data_object = MessageDataObject(msg, len(msg))
29          handler(self.connection, message_data_object, self.saved_message_context,
                self.saved_is_end_of_message, None)
30          # ... and engage the framer in the case we received more bytes than needed -->
                new message
31          if len(self.inbound_data) > 0:
32             self.connection.message_framer.dispatch_handle_received_data(self.connection)
```

Listing 5.17: The FramerHelperObject

Example Framer

NEATPy ships with an example framer to showcase how a simple framer is implemented. The example sports a trivial Type-Length-Value (TLV) framer simply framing TCP messages:

```
1  class TestFramer(Framer):
2
3     def new_sent_message(self, connection, message_data, message_context, sent_handler,
          is_end_of_message):
4        """
5        To provide an example, a simple protocol that adds a length as a header would
             receive the NewSentMessage event, create a data representation of the length of
             the Message data, and then send a block of data that is the concatenation of
             the length header and the original Message data.
6        """
7        shim_print(f"Framer got message: {message_data}")
8        new_block = (len(message_data)).to_bytes(4, byteorder='big') + message_data
9        connection.message_framer.send(connection, new_block, message_context,
             sent_handler, is_end_of_message)
10
11    def handle_received_data(self, connection):
12       """
13       To provide an example, a simple protocol that parses a length as a header value
             would receive the HandleReceivedData event, and call Parse with a minimum and
             maximum set to the length of the header field. Once the parse succeeded, it
             would call AdvanceReceiveCursor with the length of the header field, and then
```

```
        call DeliverAndAdvanceReceiveCursor with the length of the body that was parsed
        from the header, marking the new Message as complete.
    """
    shim_print("Framer handles received data")
    header, context, is_end = connection.message_framer.parse(connection, 4, 4)
    length = int.from_bytes(header, byteorder='big')
    shim_print(f"Header is {length}")
    connection.message_framer.advance_receive_cursor(connection, 4)
    connection.message_framer.deliver_and_advance_receive_cursor(connection, context,
        length, True)
```

Listing 5.18: Example Framer provided by NEATPy

Figure 5.9 and 5.10, presents the flow for both framing and parsing:[10]

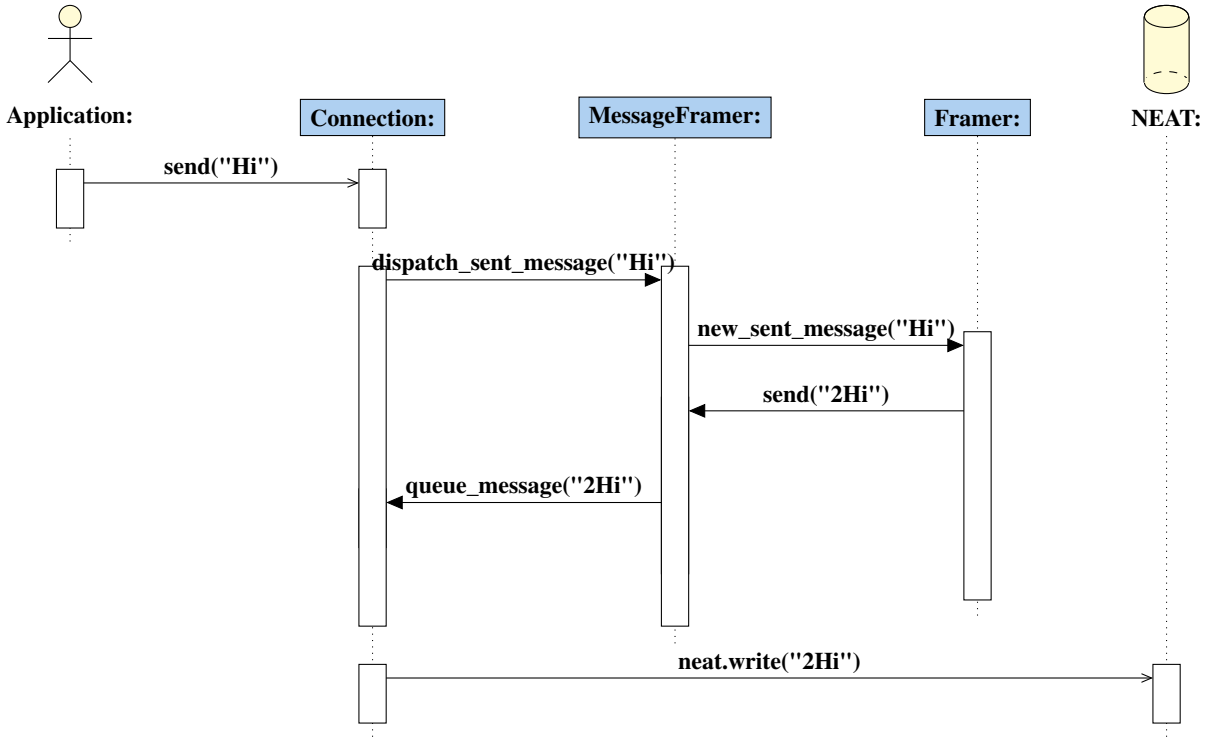


Figure 5.9: Framer flow when framing

[10] Notice how the Framer parses four bytes, as the message length is represented by four bytes (32-bits) by the framer implementation.

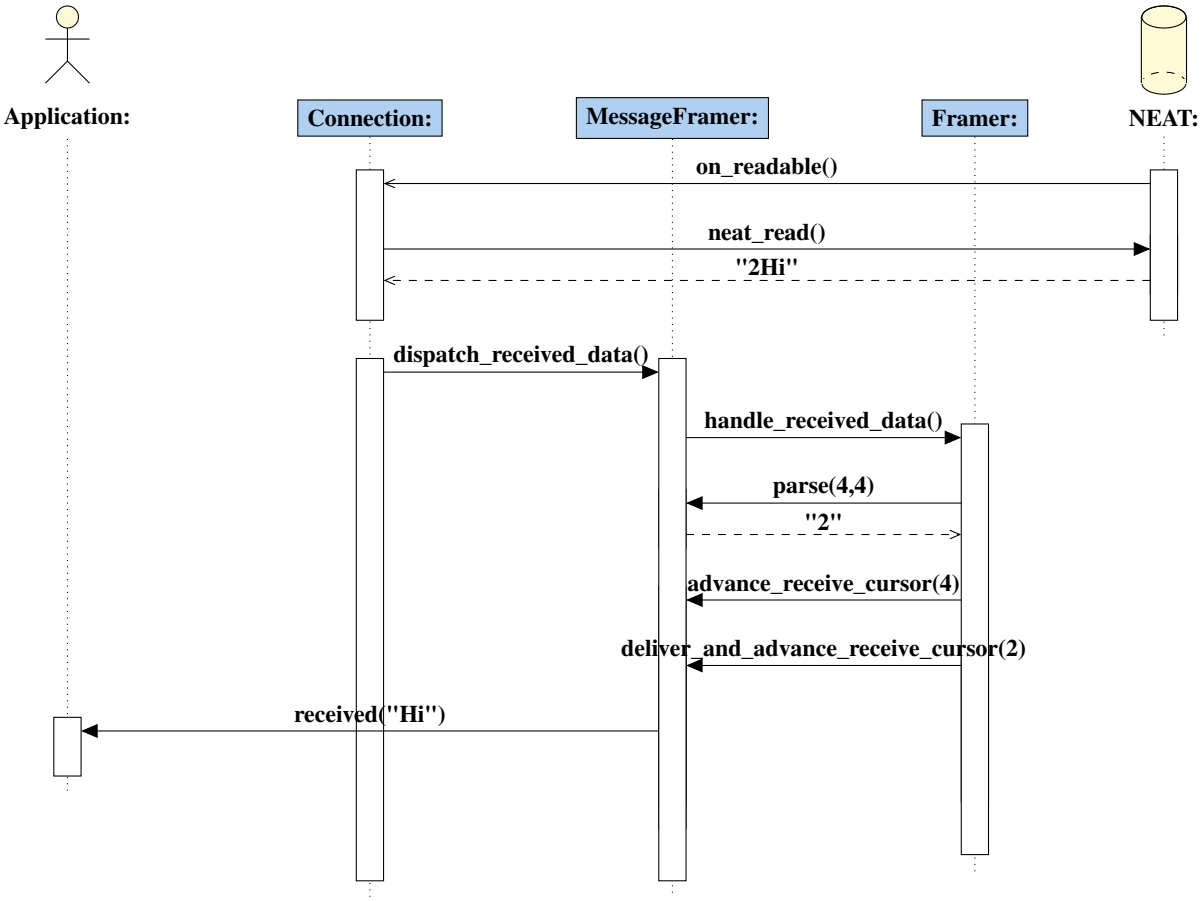


Figure 5.10: Framer flow when parsing

5.5 Connection Management and Termination

Table 5.7: Send / Receive mapping

Type	TAPS	NEAT
Object	Connection Properties	✗
Event	Soft Errors	✗
Event	Excessive retransmissions	✗
Action	Close	neat_close
Action	Abort	neat_abort
Event	ConnectionError	on_error
Event	Closed	on_close

5.5.1 Connection Management

Connection management, and with it, Connection configuration is carried out by using `Connection Properties`. They can both be configured pre and -post-establishment. In NEATPy, a set of `Connection Properties` are created with their default values when a `TransportProperties` object is created.

Connection Properties are configured with the `set_property()` method of the Connection class. True to the specification, a change in a property of a Connection that is part of a `Connection Group` is applied to the other members of the group. Some simple checks are also carried out:

```
def set_property(self, connection_property: ConnectionProperties, value):
      if not ConnectionProperties.is_valid_property(prop_to_set):
         shim_print("Given property not valid - Ignoring...", level='error')
         return
      elif ConnectionProperties.is_read_only(prop_to_set):
         shim_print("Given property is read only - Ignoring...", level='error')
         return

      # If the connection is part of a connection group set the property for all of the
          entangled connections
      if self.connection_group:
         for connection in self.connection_group:
            ConnectionProperties.set_property(
            connection.transport_properties.connection_properties, connection_property,
                value)
      # Then set the property for the given connection
      ConnectionProperties.set_property(self.transport_properties.connection_properties,
          connection_property, value)
```

Listing 5.19: How NEATPy sets Connection Properties

NEATPy follows the interface by implementing the method `get_properties`, which is called on the `Connection` class:

```
def get_properties(self):
      return {'state': self.state,
            'send': self.can_be_used_for_sending(),
            'receive': self.can_be_used_for_receiving_data(),
            'props': self.transport_properties}
```

Listing 5.20: The get_properties method

The method returns a dictionary to the application. The application is then able to inspect the dictionary. E.g., if one wants to check if the Connection can be used to receive data, one would access the dictionary with:

```
props_dict['receive']
```

The function has a rather lengthy documentation string, which specifies the different keys and return-types. This docstring is therefore omitted in the example but is part of the public API presented in Appendix A.

Capacity Profile

The Connection Property `Capacity Profile` is used by an application to specify a desired network treatment for its transported data. The TAPS interface draft [46] specifies six profiles and each one has recommended Differentiated Services Code Point (DSCP) values. NEATPy implements the profiles as an enumeration, with each codepoint's recommend value as corresponding enumeration value:

```
class CapacityProfile(Enum):
   DEFAULT = int("0x00", 0)
   SCAVENGER =int("0x01", 0)
   LOW_LATENCY_INTERACTIVE = int("0x24", 0)
   LOW_LATENCY_NON_INTERACTIVE = int("0x12", 0)
   CONSTANT_RATE_STREAMING = int("0x1C", 0)
   CAPACITY_SEEKING = int("0x0A", 0)
```

Listing 5.21: The CapacityProfile enumeration

The mechanism for setting these value to sockets is provided by NEAT with the call `neat_set_qos()`. As a result, when the application changes the Capacity Profile value, NEATPy will simply call NEAT with a new profile value.

5.5.2 Connection Termination

In general, there are two ways an application can shut down a connection: gracefully via the `close()` function call (usually signaling EOF), or through the `abort()` function call (signaling an error from the peer). TCP features the possibility of half-closed connections, which enables the closing application to receive data after the execution of close. SCTP has a more refined and clear semantic; after close is called, no further reception is possible.

As an interface for a transport system that is protocol agnostic, the interface description exposes a close call that follows the protocol with the strictest semantic meaning; messages that were passed to the transport system before calling `close()` are guaranteed delivery, but further reception of data is not possible.

Mapping `close()` from the description in TAPS to NEAT is not completely straightforward, as NEAT's implementation semantically differs. The difference lies in that NEAT does not guarantee delivery of data passed by the application, unless it has been passed down to the network layer; data still present in "NEAT buffers" will be discarded. When `close()` is called, NEATPy checks whether there are any messages left, passed over by the application. This applies to both messages not yet passed down to NEAT and messages passed to NEAT, but not handed over to the transport layer (i.e. not confirmed by the `on_all_written` event):

```
def close(self) -> None:
      # Check if there are any messages left to pass to NEAT or messages that are not
          given to the network layer
      if self.msg_list.empty() or self.messages_passed_to_back_end:
         self.close_called = True
         self.state = ConnectionState.CLOSING
      else:
         neat_close(self.ops.ctx, self.ops.flow)
         self.state = ConnectionState.CLOSED
```

Listing 5.22: Check done in the close method

If NEATPy has messages left to send, additional logic is implemented in the function that handles NEAT write completions, `handle_all_written()`:

```
def handle_all_written(ops):
   # Get the message that is confirmed sent.
   message, message_context, handler = connection.messages_passed_to_back_end.pop(0)

   # Check if close has been called; if so, check if there are any messages left to be
       confirmed sent
   close = False
   if connection.close_called and len(connection.messages_passed_to_back_end) == 0:
      shim_print("All messages passed down to the network layer - calling close")
      close = True
   # A message marked final has been successfullt sent with NEAT, initiate close.
   elif message_context.props[MessageProperties.FINAL] is True:
      shim_print("Message marked final has been completely sent, closing connection /
          sending FIN")
      close = True
   if close:
      neat_close(connection.__ops.ctx, connection.__ops.flow)
   return NEAT_OK
```

Listing 5.23: Checks done after a message has been confirmed sent by NEAT

The last bit of logic is implemented in the function, `handle_closed()`, handling the event when NEAT signals that a graceful shutdown has completed. NEATPy will fire the application's event handler handling the close event if present. Additionally, NEATPy delivers any data associated with unfulfilled receive requests:

```
def handle_closed(ops):
   if connection.HANDLE_STATE_CLOSED:
      connection.HANDLE_STATE_CLOSED(connection)

   # If a Connection closes before a requested Receive action can be satisfied, the
       implementation should deliver any partial Message content outstanding..."
   if connection.receive_request_queue:
      if connection.buffered_message_data_object:
         handler, min_length, max_length = connection.receive_request_queue.pop(0)
         shim_print("Dispatching received partial as connection is closing and there is
             buffered data")
         message_context = MessageContext()
         handler(connection, connection.buffered_message_data_object, message_context,
             True, None)
   # ...or if none is available, an indication that there will be no more received
       Messages.
   else:
      shim_print("Connection closed, there will be no more received messages")
   return NEAT_OK
```

Listing 5.24: Handling Connection shutdown

The semantic meaning of aborting a connection is identical in both TAPS and NEAT; the connection is closed, and any data present in buffers will be discarded. This makes for the simple mapping:

`connection.abort() -> neat_abort().`

5.6 Summary / Conformity of NEATPy

In this chapter, we have presented NEATPy and its implementation details. The full, public API, available to applications, is provided in Appendix A.

At the time of writing, the interface draft is at version 6 [47], while most of the implementation during this thesis has used version 5 [46]. NEATPy sports an implementation featuring all major objects, actions and events. Nevertheless, some implementation details of version 5 are left out:

- **Actions**:
  - `Rendezvous`: Currently, the TAPS description is not very clear, and the full specification for the action is yet to be finalized.
- **Events**:
  - `Soft Errors`
  - `Excessive retransmissions`

  NEAT does not send events related to Internet Control Message Protocol (ICMP) error messages or excessive retransmissions. Additional NEAT logic would therefore be needed.
- **Selection Properties**:
  - `Interface Instance or Type`
  - `Provisioning Domain Instance or Type`
  - `Use Temporary Local Address`

  Successful implementation of the Selection Properties listed above would all require additions to the NEAT codebase.
- **Connection Properties**:
  - `Retransmission Threshold Before Excessive Retransmission Notification`: NEAT does not provide any mechanism to specify a retransmission threshold. This Connection Property was therefore left out, as it would not be very useful without a configurable threshold.
  - `Connection Group Transmission Scheduler`: Implementing any function here would have to interface with SCTP, but NEAT does not offer this — so, accessing this SCTP functionality would have been a change to NEAT, but changing NEAT was not the goal of this thesis.
- **Security Parameters**: As described in Section 5.2.4, NEATPy can provide secure connections, but further customization, as specified in the interface draft, is not implemented due to constraints in NEAT. For the transport system to be fully security-conformant, further implementation of security within NEAT is needed.

In the next part we test and benchmark NEATPy in various ways.

Part III

Evaluation and Conclusions

Chapter 6

Experimental Setup

This chapter presents the setup for the various experiments carried out to test NEATPy. Even though NEAT is a major part of the implemented transport system as a whole, the goal with the test performed is to evaluate the contributions of this thesis; the "Py" in "NEATPy". In-depth performance analysis of NEAT already exists [18].

6.1 Testbed

The testing topology is streamlined with simply having the transport system running on a client and server:

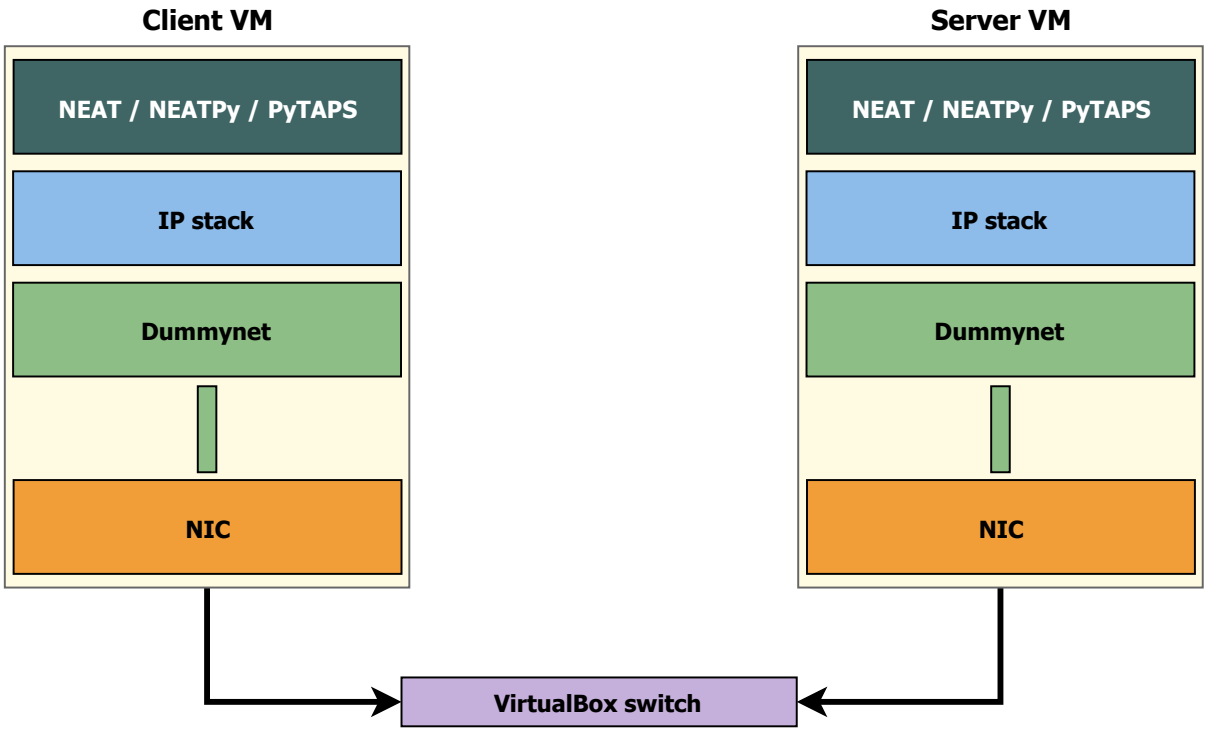


Figure 6.1: The overall structure of the testbed

6.1.1 Tools

VirtualBox

VirtualBox is a well-known virtual machine monitor (VMM) developed by Oracle and is used for x86 virtualization. The client and server are created as virtual machines and are running FreeBSD 12.1 (64-bit) as the operating system. FreeBSD is chosen as this is the only operating system in which NEAT supports SCTP multistreaming. Table 6.1 presents the specifications of the host the virtual machines are running on, while Table 6.2 lists the specifications of the guests themselves. The client and server are attached to an internal network configured in VirtualBox. By specifying the same network name in the settings for both virtual machines, VirtualBox configures the virtualized network interface cards accordingly and functions as a switch to the nodes connected to the internal network. Virtualization could affect the results to some degree. We will touch upon this in our evaluation.

Table 6.1: Specification of the host machine

Type	Value
Machine model	MacBook Pro (15-inch, 2018)
CPU	2,2 GHz Intel Core i7 - 6 cores / 12 threads
RAM	16 GB 2400 MHz DDR4

Table 6.2: Specification of the guests

Type	Value
Number of CPUs	1
Amount of base memory	2048 MB
Network adapter	Intel PRO/1000 MT Desktop (82540EM)

Dummynet

Dummynet [6] is used as the network emulator for the experiments. It is part of the FreeBSD OS, and with its lifespan of over 25 years, it is a reliable and well-tested tool for network emulation. Its traffic shaping capabilities include configuration of delay, packet loss, bandwidth, and queues. Dummynet is used in conjunction with `ipfw`, a firewall for FreeBSD, which functions as the user interface for Dummynet. The emulation is configured in two main steps: (1.) traffic selection is realized by specifying *rules* within ipfw. (2.) the desired traffic is then passed through objects called *pipes*. Pipes in Dummynet are implemented with their own scheduler, queues, and link. It is here the configuration of features like bandwidth and loss rate is done.

6.2 Experiments

This section describes the experiments. For each experiment we start with the motivation, followed by configurations and parameters.

6.2.1 Overhead Measurements

This experiment measures the overhead generated by calling the compiled NEAT library written in C from the Python shim with the help of language bindings generated by SWIG. We measure the execution time for all of the transport systems when transferring a certain number of bytes. With this experiment, we seek to answer one of the main research questions: what is the performance penalty of having a transport system that makes use of language bindings? Besides, comparing NEATPy against NEAT, we also include PyTAPS in the experiment both for reference and measurements of PyTAPS itself.

Configuration

- **Payload**: To measure the overhead, we repeat the experiment with three different payload sizes:

  12 Bytes - 70 KB - 1 MB

- **Fixed parameters**: For simplicity the following parameters are configured with the same value throughout this experiment:
  - Latency: 15 ms, consequently a Round trip time (RTT) of 30 ms.
  - Queue size: 50 packets.
- **Configuration of the Bandwidth delay product(BDP)**: In these experiments we manipulate the BDP by the configuring the bandwidth. Each payload transfer is measured with the following enumeration of link speeds:

  1 - 5 - 10 - 15 - 20 - 25 - 30 - 50 - 100 (Mbit/s)

- **Repetitions**: Each combination of payload size and link speed is repeated ten times. This is to ensure that the results are reliable, as factors like context switching could affect the result.

6.2.2 NEATPy Analyses Using a Python Profiler

This experiment focuses on NEATPy itself and its goal is to investigate how time is distributed within the Python implementation. By measuring the duration of NEATPy's various functions, we get a better overall picture of the runtime, and this could assist in better pinpoint where the overhead is compared to NEAT. The tool used to assist in this is a Python profiler.

Python Profiling and Choice of Profiler

Profiling is often referred to as benchmarking and is, in essence, a dynamic analysis of a program achieved by instrumentation. By using a profiler, a programmer gets an overview of execution times, which aids in detecting bottlenecks, and ultimately, code optimization.

There are mainly two types of profilers: deterministic and statistical profilers. A deterministic profiler monitors every function call executed, while a statistical profiler records the call stack at a given sample rate, and based on this information extrapolates the result. There are trade-offs when using one or the other. Deterministic profilers offer better precision, but the fact that the profiler is called on every single function could distort the result, in that functions that invoke the profiler often may appear slower, or in this use case, may appear to generate significantly more overhead. With statistical profilers, one trades some precision against significantly less overhead.

We chose *pyinstrument* [37], an open-source, statistical profiler to be used for this experiment. This choice is due to the following reasons:

- While most deterministic profilers record the duration of a function in terms of CPU time, pyinstrument uses wall-clock time. With this, we can measure the duration of time a function in NEATPy uses, including calls to the low-level code.
- pyinstrument performs what its developers refers to as "full-stack recording". This is of great convenience for those who use the profiler, as it by default hides library frames and presents a tree of calls focusing on the application code. Deterministic profilers will simply measure every single function call and preset a broad list of functions, which is ordered by time spent in each function. This list could sport many functions unknown to the programmer, like deep library functions, for which it could be hard why were called. By recording the entire stack, pyinstrument is able to answer why these functions are called.

Configuration

- **Payload**: We use the larger payload size of 1 MB in this type of experiment. The reason for this is to let pyinstrument get as many samples as possible (being a statistical profiler).
- **Static parameteres**: We use the same static parameters as in the first experiment. Furthermore, we use a reasonable link speed of 15 Mbit/s for each payload transfer. Again, the choice is made to get better measurements from pyinstrument.
- **Repetitions**: We chose to repeat all experiments three times, the reason being pyinstrument provides a lot of data, and there is no comparison, unlike the overall overhead measurements.

6.2.3 Multiple Connections

While NEAT supports SCTP on both Linux and FreeBSD, multistreaming support is only implemented for FreeBSD. With this experiment we investigate the possibilities NEATPy has in offering SCTP multistreaming. Two messages are sent: one large, measuring 15MB and a smaller one of 1 MB. Additionally we run the same experiment with PyTAPS for comparison. With both implementations following the ongoing standardization efforts, we are able to run almost identical source code for both NEATPy and PyTAPS.

- **Payload**: One large payload of 15 MB, followed by a smaller one of 1 MB.
- **Static parameteres**: To better see the effect of the streams, we use a moderate bandwidth of 5 Mbit/s and a total RTT of 30 ms between the hosts.
- **Repetitions**: The experiments are repeated 10 times to ensure reliable results.

6.2.4 Framer Test

Lastly, we want to test whether the introduction of framers to the API brings any non-negligible overhead. As far as we know this kind of comparison has not been done before. The simple TLV framer presented in Section 5.4.4 is used. We will simply measure the runtime of a client sending and receiving messages, with and without the Framer.

Configuration

- **Payload**: We use a payload size of 70 KB throughout this experiment.
- **Static parameters**: We continue using a delay of 15 ms and a queue size of 50 packets. Additionally, we keep the bandwidth static, with a link speed of 15 Mbit/s.
- **Number of messages**: To investigate what effect it has to frame TCP messages, we will do measurements for three different numbers of messages:

  1 message - 50 messages - 100 messages

- **Repetitions**: We repeat each test ten times.

Chapter 7

Evaluation

7.1 Overhead Measurements

The results are shown in Figures 7.1, 7.2 and 7.3, supplemented by Tables 7.1, 7.2 and 7.3. In this section, we will first comment on the overall results and the most interesting findings. The second part of the section discusses the results for NEATPy, and tries to shed light on origins for the overhead. Throughout this experiment, the standard deviation σ was between 2 and 4 %. Given its low value, it is omitted from the graphs, as it would be too small to properly display.

7.1.1 Comments on the Results

The tests showed the following key findings:

- NEATPy has the highest runtime in all of the test scenarios. Being a transport system that uses language interoperability from Python to C, this was expected. Additionally, we observe that the overhead, compared to NEAT, is by far the highest for the experiment sending 12 bytes. With this transfer, NEATPy reaches almost three times (200 %) the duration of NEAT. There is reason to suspect that Python's startup delay is the source of this high number. We will discuss this in detail in Section 7.1.2.
- Another interesting finding is that of PyTAPS being the transport system with the lowest runtime in the middle-size transfer. In fact, both Python implementations total overhead, compared to NEAT, is smaller in percent than in the lowest size transfer. As this experiment transfers a lot more data, Python's startup time makes up a much smaller portion of the total runtime of the program. This confirms that the startup time of a Python program can become a bottleneck, compared to a C program, which is compiled into binary machine code.
- With the last experiment NEAT and PyTAPS have switched places once again. Measuring two Python featured implementations provides us with a benefit when interpreting results; NEATPy's overhead in percent, compared to NEAT, is higher than that of the 70 KB transfer. This is also the case for PyTAPS. This points towards some inherent feature of Python being the reason for this performance decrease. Further investigation has indicated that the reason for this is Python's automatic memory management, and with it, garbage collection. We elaborate on this next, discussing NEATPy's overhead.

7.1.2 Discussion - NEATPy's Overhead

When discussing NEATPy's overhead compared to NEAT it is important to remember that both implementations register callbacks in NEAT. Naturally, this implies that callback methods in NEATPy do not account for all of the overhead, as applications written for NEAT have similar callback functions. This leaves us with two major factors that are likely to be the source of the net overhead:

SWIG

As we described in Section 4.2.3, the usage of the language bindings will result in the following chain of calls:

NEATPy ➔ SWIG-Python-module ➔ SWIG-low-level-wrapper ➔ NEAT

This chain of calls naturally results in additional stack frames. A part of this process is the type-translation done by SWIG, enabling the interoperability between the languages. The overhead is constant for each function call in the application's lifetime. To further explain the variation in overhead percent between the three test scenarios, we will have to look closer at the Python machinery.

Python

As touched upon earlier, Python as a programming language has some inner mechanics that seem to play a major part in the measurements:

- **Python's startup time**: As already stated, there is a significant difference between running a Python program and a C executable. The startup of a Python program can mainly be divided into two parts:

  1. **Starting up the Python interpreter**: The interpreter itself is written in C, while most of the standard library is written in Python itself.
  2. **Module import**: The way module code is made available to the interpreter (and applications) is through the import statement. The overhead of importing modules can logically be divided into two elements:

     (a) Localize the module and read in its data.

     (b) Execution of top-level code, i.e. code that is outside functions and classes.

     We believe that all this put together, in combination with the low runtime when transferring 12 bytes, is the reason for the very high overhead, as displayed in Table 7.1. The NEAT executable is faster than its Python counterparts in this scenario. In fact, the author in [11] states that "Python startup time is worse than some other scripting languages and more recent versions of the language are taking more than twice as long to start up when compared to earlier versions (e.g. 3.7 versus 2.7)". This is acknowledged and backed up by the creator of Python, Guido van Rossum, stating that "the cumulative startup time of large Python programs is a serious problem"[1].

[1] As part of a discussion on the *Python-Dev* mailing list: https://mail.python.org/archives/list/python-dev@python.org/message/2SPHFS4EBKQIYCDBOBHWRMZ35JZTN5ZS/

- **Dynamic memory management and garbage collection**: In the last test scenario of this experiment, we see that the two Python implementations have a performance decrease compared to NEAT and the 70 KB transfer. We believe that this is a result of dynamic memory management and garbage collection. Simplified, this mechanism of the Python interpreter can be explained as such:

  1. The Python interpreter sees every value as an object.
  2. Every new object created is placed on the heap and managed by the interpreter. This entry includes a reference count and type of object.
  3. When an object's reference count is 0, garbage collection is carried out, i.e. the object's memory is deallocated.

  This means that every new object adds additional overhead, in that reference counting together with garbage collection must be maintained. With the longer running example, the number of function calls, and with it objects created, grows rapidly, which results in the interpreter performing garbage collection. We believe this is the reason for the measurement results in the case of the largest transfer.

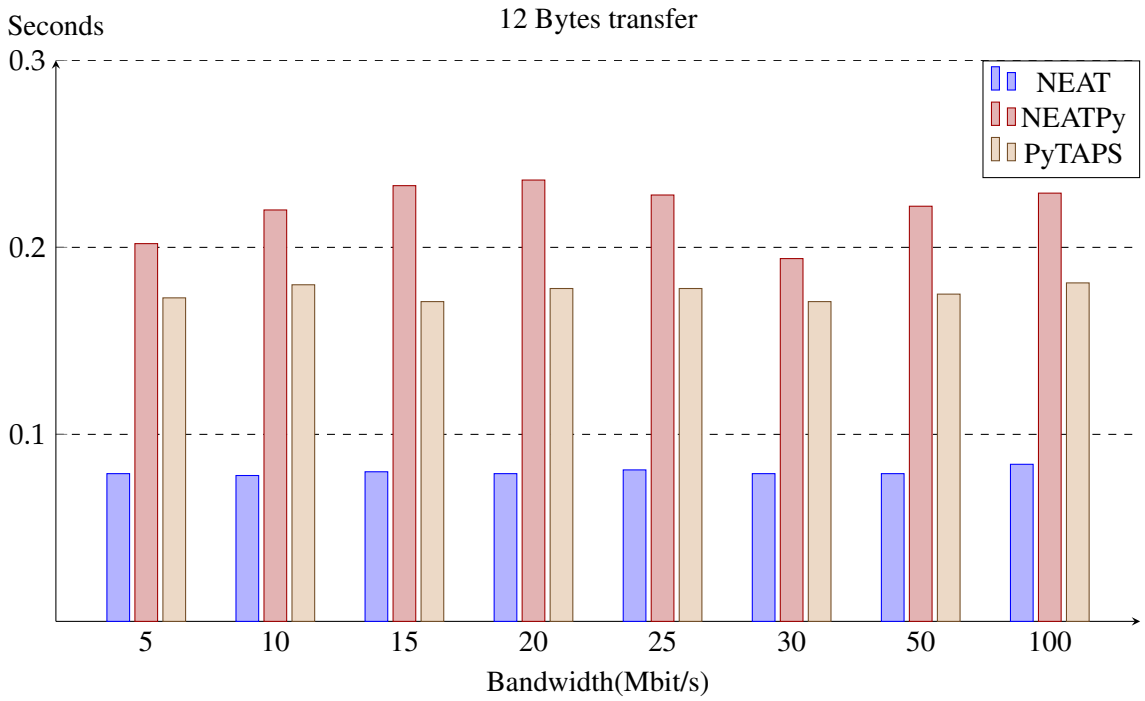


Figure 7.1: Graph for the 12 Byte transfer. 1 Mbit results were removed to better display the graphs. The data in full is presented in the corresponding tables.

Bandwidth	NEAT	NEATPy	PyTAPS	NEATPy compared to NEAT (in percent)	NEATPy compared to NEAT (in seconds)
1 Mbit/s	0,077	0,218	0,177	183,117	0,141
5 Mbit/s	0,079	0,202	0,173	155,513	0,123
10 Mbit/s	0,078	0,220	0,180	183,012	0,142
15 Mbit/s	0,080	0,233	0,171	191,136	0,153
20 Mbit/s	0,079	0,236	0,178	198,986	0,157
25 Mbit/s	0,081	0,228	0,178	180,788	0,147
30 Mbit/s	0,079	0,194	0,171	144,823	0,115
50 Mbit/s	0,079	0,222	0,175	179,219	0,142
100 Mbit/s	0,084	0,229	0,181	172,262	0,145

Table 7.1: Measurements - 12 Byte transfer

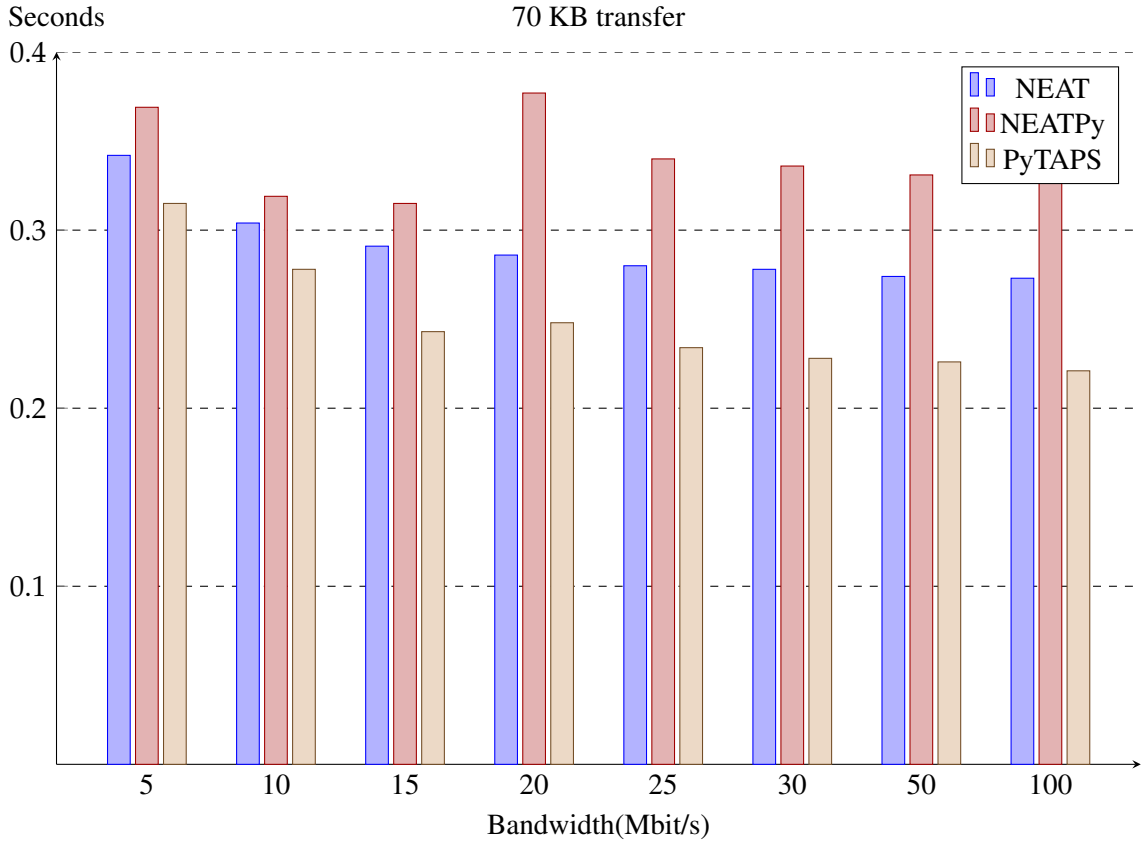


Figure 7.2: Graph for the 70 KB transfer

Bandwidth	NEAT	NEATPy	PyTAPS	NEATPy compared to NEAT (in percent)	NEATPy compared to NEAT (in seconds)
1 Mbit/s	0,808	0,891	0,773	10,341	0,084
5 Mbit/s	0,342	0,369	0,315	7,901	0,027
10 Mbit/s	0,304	0,319	0,278	4,852	0,015
15 Mbit/s	0,291	0,315	0,243	8,520	0,025
20 Mbit/s	0,286	0,377	0,248	31,759	0,091
25 Mbit/s	0,280	0,340	0,234	21,409	0,060
30 Mbit/s	0,278	0,336	0,228	20,755	0,058
50 Mbit/s	0,274	0,331	0,226	21,024	0,058
100 Mbit/s	0,273	0,350	0,221	28,532	0,078

Table 7.2: Measurements - 70 KB transfer

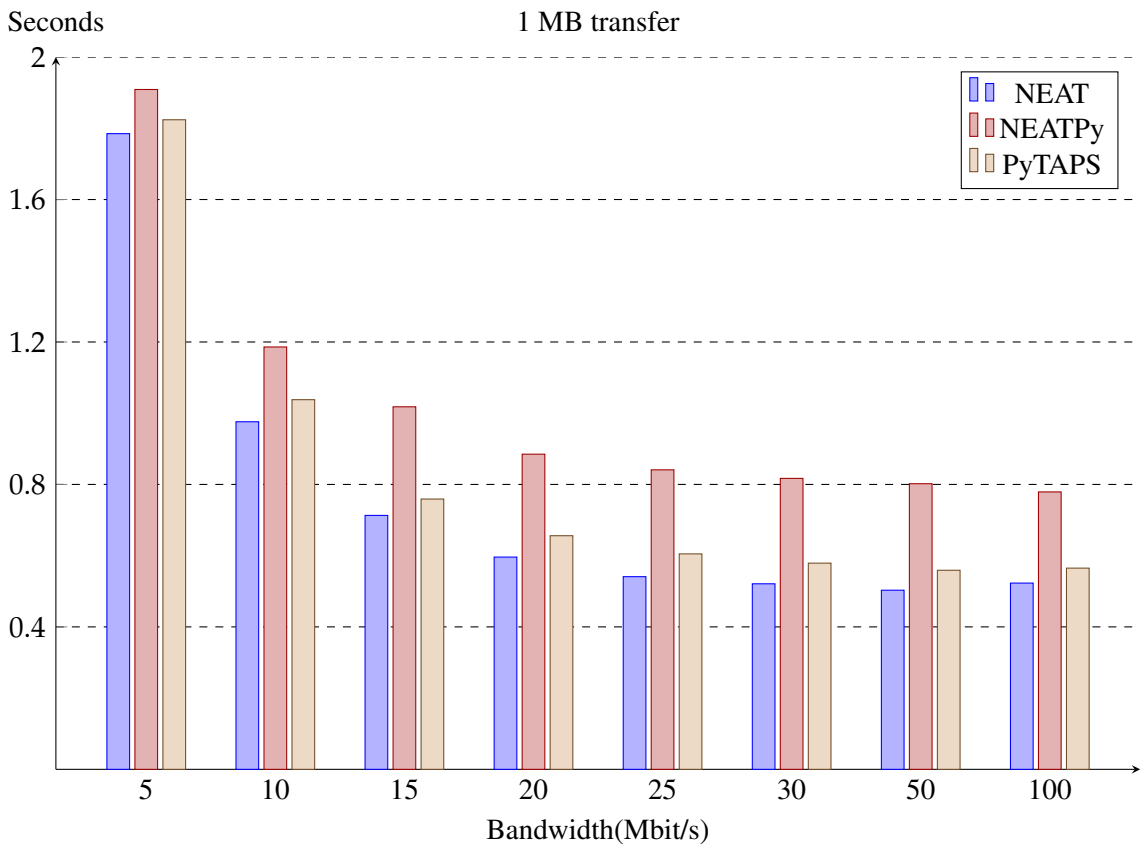


Figure 7.3: Graph for the 1 MB transfer

Bandwidth	NEAT	NEATPy	PyTAPS	NEATPy compared to NEAT (in percent)	NEATPy compared to NEAT (in seconds)
1 Mbit/s	8,410	8,511	8,458	1,204	0,101
5 Mbit/s	1,785	1,909	1,824	6,961	0,124
10 Mbit/s	0,976	1,186	1,038	21,579	0,211
15 Mbit/s	0,713	1,018	0,759	42,912	0,306
20 Mbit/s	0,596	0,885	0,656	48,532	0,289
25 Mbit/s	0,541	0,841	0,605	55,597	0,301
30 Mbit/s	0,521	0,817	0,579	56,888	0,296
50 Mbit/s	0,503	0,802	0,559	59,450	0,299
100 Mbit/s	0,523	0,779	0,565	48,910	0,256

Table 7.3: Measurements - 1 MB transfer

7.2 NEATPy Analyses Using a Python Profiler

During the prior experiments, we experienced that pyinstrument did not provide us with all the information that we wanted. More specifically, some functions were not timed. This is due to pysintruments's sampling rate, being a statistical profiler. Therefore, we decided to employ a deterministic profiler. We chose to use *cProfile*, which is included with Python. As the functions we want to time are only called once, the additional overhead of a deterministic profiler won't affect the results. To verify this we compared several functions that pyinstrument managed to time and the results were identical down to millisecond precision.

Distribution of Time

We will start with a broad view, and gradually focus on more fine-grain measurements. Let us start with the time spent before starting the NEAT event-loop and start of the event loop to termination (i.e. the Pre-establishment phase and from establishing to connection termination). This is shown in Figure 7.4.

The way we measure this is by looking at the duration for the function call `preconnection.start()`. As previously mentioned, this call will not return until the event loop is stopped.

Next, we investigated how the time spent in the pre-establishment phase was divided. The results are shown in Figure 7.5.

We can make the following observations:

- As shown, the creation of a `TransportProperties` object makes for a little more than 30 % of the time spent. The majority of the time is spent importing the modules for the Message Properties, Selection Properties and Connection Properties. Again, these are implemented as enumerations. The import of the modules combined with the fact that Python treats every variable as an object, described in Section 7.1.2, is likely the reason for the amount of time spent here.
- Outlined in Section 5.2.4, the call `bootstrap_backend()` creates a NEAT Context, Flow and Operations struct. This is the call that uses the most time in this phase. Unlike most other usages of the language bindings, this call creates and initializes objects in the NEAT back-end, hence

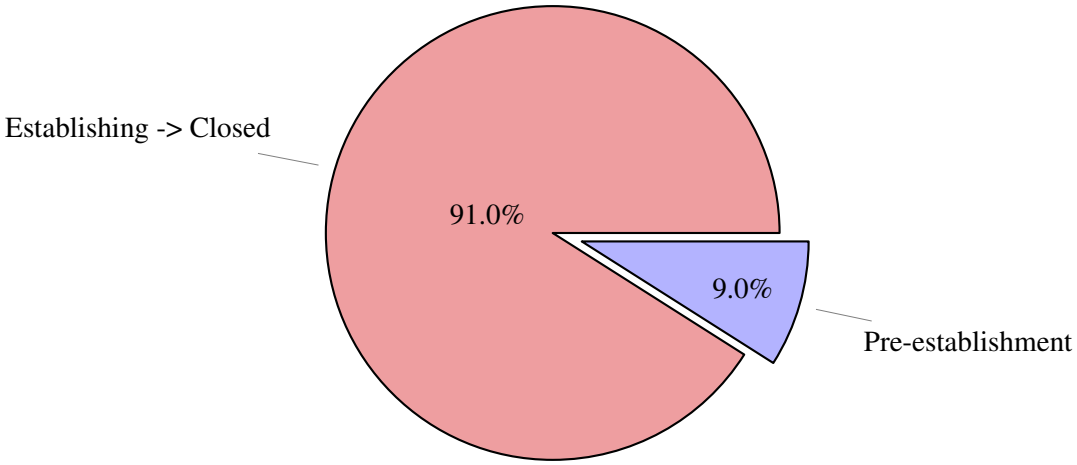


Figure 7.4: Distribution of time spent in Pre-establishment vs. Establishing to Closed

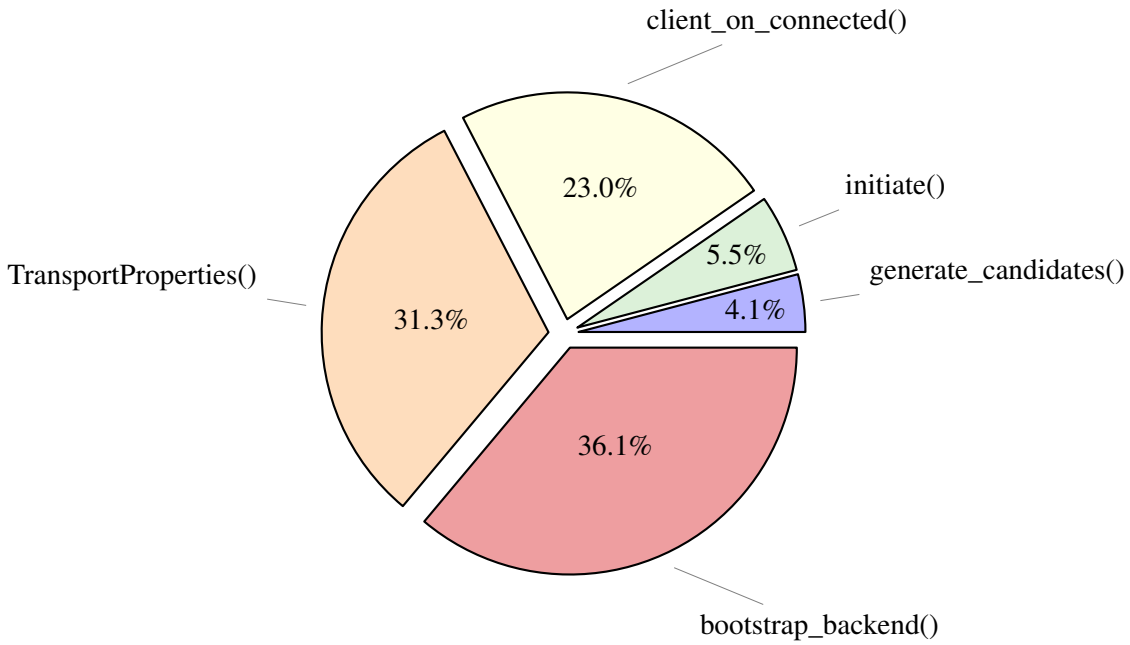


Figure 7.5: Distribution of Time within the Pre-establishment phase

we see a rather large amount of time spent here. Besides, the time spent during Preconnection initialization is so low that it is listed as 0 by the profiler, which indicates a time less than 1 ms.

- The function `client_on_connected()` is the function that handles the `on_connected` callback from NEAT. It finalizes a Connection for further use. This includes making a deep copy of the Transport Properties. Secondly, it calls the event handler, `HANDLE_CONNECTION_STATE_READY`, if the applications have registered a handler within the Preconnection class.

- Lastly we have two functions with a lower percentage of the time used. These functions have both a smaller footprint and simpler logic, hence less time elapsed. The `initiate()` method calls `neat_open()` and creates a Connection object, which is returned to the application. Lastly, `generate_candidates` generates candidates to be used for racing in the NEAT back-end. Most of the logic when generating candidates consist of conditional statements and yields little additional overhead.

After the event loop in NEAT is started, the program flow and communication with NEAT is mainly realized through the NEAT callbacks `on_readable()` and `on_writable()`. These are handled by the methods `handle_on_readable()` and `on_writable()` in NEATPy. Measurements by the profiler show that these method calls have a very low execution time. The `handle_on_readable()` clocks in only 1 ms, and `handle_on_writable()` is listed as 0, which means it has an execution time of less than 1 ms.

During the measurements, we observed two modules with an execution time too low for the profiler to measure. However, they were called between 1000 and 3000 times. This is what they do:

- **importlib**: The documentation states that this is the library module that provides the implementation of the import statement, and provides functions to locate and load Python modules. As mentioned in the previous section, Python, relative to other scripting languages, uses quite some time "feeding" code to the interpreter with imports.

- **built-in methods**: These are methods that, as the name suggests, are built into the interpreter itself. The function `isinstance` is one of the functions we observed called many thousand times. The interpreter itself could be using this "under the hood", albeit, we could not verify this suspicion.

With this we have iterated through the NEATPy functions we were able to measure with the profiler. Sadly, the profiler cannot measure the time spent in the various low-level functions. Therefore, to further get a more detailed view of the greater part in Figure 7.4, one would need to profile the NEAT library and event loop. Such an investigation has been undertaken in prior work [18].

7.3 Multiple Connections

As mentioned, we are able run source code that is quite similar for both implementations measured. However, in PyTAPS the experiment will result in two TCP connections, while NEATPy is able to create an additional stream within the same SCTP association.[2] This is a scenario that demonstrates how a transport system specified by TAPS really is protocol-agnostic and dynamic in the choice of transport.

During our testing we encountered some difficulties with NEAT's SCTP implementation:

- Initially, we experienced that an SCTP association created by NEAT immediately got aborted by the client. Luckily this bug had previously been described in an issue on NEAT's GitHub page.[3] The problem was specific to running FreeBSD in a virtual environment, while at the same time using both a NAT adapter and an internal network adapter. The reset was sent by the NAT service, and the solution was to simply disable the NAT adapter in VirtualBox.
- Following this, we encountered an error within NEAT when creating a new logical stream within the already established association. NEAT failed to create a new stream when previously sent messages on the association (stream 0) were larger than around 30 KB. We were unable to locate this error on our own, but fortunately, one of the main implementors of SCTP on FreeBSD was able to locate the error and provide a fix for the bug in NEAT.[4] The error can be summarized as follows:

  1. A signaling message (for stream 0) and a data message (for stream 1) are received with only a short time difference. The signaling message is used by NEAT to transparently map new SCTP streams by creating new NEAT flows, as described in [48].[5]
  2. When NEAT calls `recvmsg()` both messages have already been processed by the kernel.
  3. Within the NEAT logic, requests are made to receive information about the current and next packet - `SCTP_RECVRCVINFO` and `SCTP_RECVNXTINFO`.
  4. There is however only space provided for RCVINFO in the code:

```
char cmsgbuf[CMSG_SPACE(sizeof(struct sctp_rcvinfo))];
```

[2] Multistreaming would not even be expected in PyTAPS, as it does not support the TAPS "clone" call. However, even if "clone" were available, PyTAPS could only implement it by opening two TCP connections as it is only written on top of TCP and UDP.

[3] As described in: https://github.com/NEAT-project/neat/issues/432

[4] A massive thanks to Michael Tüxen. We would not have been able to perform this experiment without his help!

[5] The signaling message has a 1 byte payload and is simply sent prior to any application data, to accommodate for said mapping.

5. The result is that the kernel is not able to deliver the information available; as a result the signaling message of a new stream is discarded.

The suggested fix for this error in NEAT was to disable the request of information regarding the next message, as the information was only used for debug output.

- In the process of helping out with the previously outlined error, two additional errors were discovered:
  - An error in NEAT's write function that resulted in termination of the program running. The error occurred when sending messages larger than around 5 MB. The reason for this was that an an error number used in a NEAT debug statement was not cleared and then incorrectly mapped to `NEAT_ERROR_IO` in subsequent logic, which results in termination of the application. The fix was simply to comment out the log statement.
  - The last bug occurred when transferring fairly large messages, exceeding 40 MB. FreeBSD would output a warning reporting that a chunk larger than the Maximum Transfer Unit (MTU) was present and the transportation would simply cease. This was classified as a bug in the FreeBSD kernel, and we will assist the implementors of FreeBSD in reproducing the bug and ultimately fixing it.

Initially, our desired approach was to start with the bigger transfer, then commence with the smaller payload after a given number of seconds. However, to start a new stream after a given amount of time proved difficult when interfacing with NEAT and its event loop.

Our attempt to facilitate this was to solve this in an approach similar to how we implemented NEATPy's initiate timer (described in Section 5.3.2 on page 53): register a simple C function with the NEAT event loop and provide libuv with the number of seconds for when to carry out the callback. Our C function would then simply call a Python function registered with the NEAT operations struct. When this function is called, `clone()` would simply be called by the client in NEATPy. Unfortunately, when using this approach NEAT would not register the new flow as a multistream flow. Efforts were made to investigate what could cause this, but we were not able to locate the error.

Due to this, our approach of creating a new stream and transferring both payloads in NEATPy is the following:

1. Call `send()` and start the bigger payload in NEATPy. NEATPy's send function takes an optional event handler for when the bytes are successfully sent down to the transport layer. We create a simple sent handler that carries out a call to `clone()`.
2. NEAT will fire off the `on_all_written` callback, which calls our handler in the Connection class. This handler will then call our sent handler described.
3. The Connection gets cloned, and as a result a new stream is created within NEAT.

This is not optimal, as the new stream is created and starts its transfer when the first one is about to end. With PyTAPS we start the second TCP connection 10 seconds after the first one is started.

Figure 7.6 presents the measurements for the transfer of the two payloads with NEATPy and PyTAPS. Despite the less than ideal approach of using streams with NEATPy, we can observe the advantages SCTP offers in parallel transfers: a new SCTP stream is able to take advantage of the congestion window obtained by the association. The TCP connections in PyTAPS does not have the same advantage and must "compete" for bandwidth.

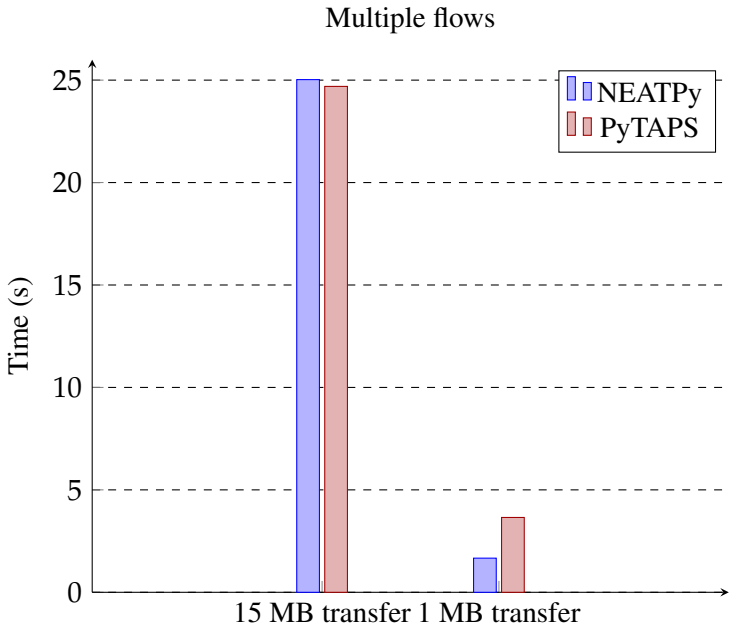


Figure 7.6: Multiple transfers with NEATPy and PyTAPS. The experiment was repeated 10 times, and the standard deviation σ was between 0,49 and 1.53 %

7.4 Framer Test

For this test, we used the framer that ships with the API. As previously explained, this framer simply prepends a TCP message with the length of said message. The experiment is carried out with one, fifty and one hundred messages. Each message sent corresponds to one call to `send()`, both when using a framer and not. The results are presented in Figure 7.7, complemented by Table 7.4.

The action of framing messages intuitively suggests a possible overhead. Interestingly, our findings show using a framer with the API can be slightly more effective.

Before further elaborating, let us quickly consider the program flow for both when using a framer and not:

- **A Connection backed by TCP, without a framer**:
  - **Sending**: A call to send will simply add the message to the outgoing message queue.
  - **Receive**: Without a framer, a TCP stream will be interpreted as one big message, resulting in one or more `receivePartial` events. To further receive more of the same message the application has to call `receive()`. The message is deemed complete when the remote end signals this with a FIN.
- **A Connection backed by TCP, using a framer**:
  - **Sending**: A call to send will result in the internal `MessageFramer` signals the Framer implementation to frame the message, as explained in Section 5.4.4. This corresponds to a total of four function calls.
  - **Receive**: The Framer implementation is signaled an incoming message, and parses it. Initially, this corresponds to four function calls, with the addition of the convenience method,

Messages	No framer	With framer
1	0,2878	0,2939
50	3,0042	2,9425
100	5,6519	5,4611

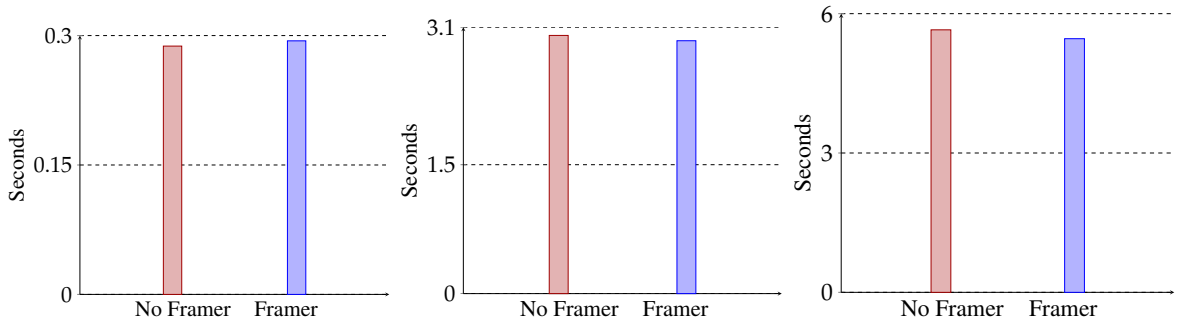
Table 7.4: Framer measurements

`deliver_and_advance_receive_cursor`, which enables the Framer to earmark bytes that have not been received by the transport system yet.

We can observe that with a single message being framed, framing causes a miniscule amount of overhead. The reason for this is simply the overhead of additional function calls needed for framing and parsing exceeds that of a Connection without a Framer.

However, when the number of messages grows, the cumulative overhead of not using a Framer actually exceeds that of using a Framer. The ability for the Framer to earmark bytes is most likely the reason for this. Following the first chunk of data received, the call flow, and overhead for the Connection not using a Framer, will over time result in more overhead:

- Having parsed the length, the Connection using a Framer will only need to check whether the amount of data received completes the message. If this is not the case, the data is simply buffered, and control handed over to the NEAT event loop.
- The Connection not using a Framer will have to handle a `receivePartial` event, buffer the data, then call `receive()` for the remaining part of the message.
- One can therefore also conclude that it is at the "parsing end" where the benefit of the Framer is obtained.



(a) 1 Message **(b)** 50 Messages **(c)** 100 Messages

Figure 7.7: Measurements with/without framers - Each test was repeated 10 times and the standard deviation σ was between 2.3 and 2.9 %.

Chapter 8

Conclusions and Future Work

Having discussed the implementation of NEATPy with all its challenges and evaluated its overhead, we can now return to the initial research questions and see if we can derive general answers from the lessons that we have learned in the course of NEATPy development. Then, we conclude this work after a discussion of possible future work.

8.1 Addressing Research Questions

- RQ1: What are the main challenges in terms of language interoperability, when representing functions of a transport layer API written in C in Python?

As outlined in Section 4.2.4, we find that type conversion is one of the most challenging aspects when wrapping a low-level codebase, especially when that codebase is as large as NEAT. Generated wrappers from a tool such as SWIG are not guaranteed to work "out of the box".

Takeaway #1: **Exception handling is vital when working with language interoperability.**

Following the discovery that our initial problem with the bindings was tied to a camouflaged TypeError, we made sure to use Python's *try statement*, handling TypeErrors. During development, this assisted us several times, enabling us to locate errors swiftly.

- RQ2: Can the design of a wrapped, low-level transport layer library limit the implementation of a high-level transport layer interface in any way?

Yes, the design of the low-level library could influence how a high-level interface is developed. In this thesis, we have shown that this is especially the case when there is an event loop running within the wrapped code. The two most notable design details in NEATPy would be:

- The deviation from the TAPS interface with the introduction of the `preconnection.start()` method - outlined in Section 5.3.2.
- Minor additions to the low-level code, to map flows to Connections during callbacks - presented in Section 5.3.3.

Takeaway #2: **Writing an interface with language bindings could be limited by the design of the low-level code. Ideally, such code should be written with the possibility of a future shim layer**

like NEATPy in mind. Sometimes, flexibility must be achieved through additions to the low-level code.

At the time of NEAT's design, it was impossible to know what kinds of properties that would be offered by TAPS. An example of this is that the requirement levels of TAPS are different from the ones in. Luckily, NEAT offers a direct choice of protocol, which allowed us to implement the ranking decision within NEATPy (Algorithm 1 on page 47). It's important for future TAPS implementations to give this kind of control to the system above, even when "normal" applications won't need it.

Our experience is that an implementation like NEAT is a suitable candidate for employing language bindings. We believe that the combination of having NEAT, dealing with protocol machinery and interfacing with the kernel, with NEATPy, providing a clean and easy to use interface, is an ideal solution. Furthermore, we believe it holds the flexibility for future growth, both for changes in TAPS's interface and supporting future transport protocols by additions to the NEAT codebase.

RQ3: What is the performance penalty of an interface making use of languages bindings?

In Chapter 7, we present our measurements that show that NEATPy's overhead varies from 30 to 200 %. With our findings, we can also conclude that the majority of NEATPy's overhead derives from inherent mechanisms of the Python programming language.

Takeaway #3: **Concerning the performance of wrapped libraries, the choice of target languages has great significance.**

8.2 Future Work

For future work we propose the following:

- **Security**: As stated in Section 5.2.4, NEATPy provides secure connections, but further development in NEAT is needed. Currently NEAT is only able to create secure connections, but not able to verify certificates. Additionally, trusted certification authorities (CAs) are hardcoded within a file, while in TAPS there are methods for adding identities, keys, and supported algorithms.
- **Coupled congestion control and Connection Group Transmission Scheduler**: Both are tied to Connection Groups and cloning. Again, additions to NEAT are needed:
  - NEAT only has support for coupled congestion control on FreeBSD with a patch, implementation efforts for other platforms are therefore needed. Furthermore, support for prioritised transfer for SCTP would need to be implemented. With this functionality added, the Connection Property `Priority` in TAPS would be implemented successfully.
  - To support the Connection Property `Connection Group Transmission Scheduler` one would have to interface with SCTP as described in RFC 8260 [44]. Currently NEAT does not offer this, so changes to NEAT are needed.

8.3 Conclusion

In this thesis, we have created a high-level interface with the help of language bindings that wrap around a large, low-level library. The result is NEATPy: a standards conformant, protocol-independent transport system. At the time of writing it is the implementation that most closely follows the interface specified by the TAPS WG. This is shown in Table 3.1 on page 25.

Being developer-friendly should be a core value of any implementation offering an API. We argue that this has been achieved with NEATPy: having the transport system implemented in Python provides developers with powerful abstractions and at the same time offers a decent level of ease-of-use. A good example of this is the difference in a minimal client. In NEAT, the client consists of 101 lines of code, while its Python counterpart measures 19 lines of code. In addition to this, we have invested time in creating thorough documentation, available for developers online.

Listings 8.1 and 8.2 shows a minimal client in NEATPy and PyTAPS, respectively. The code is very similar; with only minimal changes, this network code can run over two entirely different systems. This truly shows the protocol-independence manifested in the transport system. Given the Selection Properties in the examples, the client in PyTAPS is going to run over TCP, with lower overhead (as measured in Chapter 7). NEATPy's client will run over SCTP, if supported along the path and the hosts themselves, and run over TCP otherwise. This could provide benefits, like multihoming in the network fails, or if the application later calls `clone()`, the cloned connection could be backed with a stream within the SCTP association.

We believe that NEATPy holds great promise for being future-proof because it realizes separation of concerns: core networking functionalities can be implemented by amending the logical backend (NEAT), while higher-level API changes can be developed purely within NEATPy. With this, we have a transport system that could adapt as the standardization efforts progress, and ultimately, assists in de-ossifying the Internet transport layer.

```
def ready_handler(connection: Connection):
   connection.send("Hello server", None)

def main():
   ep = RemoteEndpoint()
   ep.with_address("127.0.0.1")
   ep.with_port(5000)

   tp = TransportProperties()
   tp.require(SelectionProperties.RELIABILITY)
   tp.prefer(SelectionProperties.PRESERVE_MSG_BOUNDARIES)

   preconnection = Preconnection(remote_endpoint=ep, transport_properties=tp)
   connection = preconnection.initiate()
   connection.HANDLE_STATE_READY = ready_handler
   preconnection.start()

if __name__ == "__main__":
   main()
```

Listing 8.1: A minimal client in NEATPy

```
class TestClient():
   async def handle_ready(self, connection):
      await self.connection.send_message("Hello server")

   async def main(self):
      ep = taps.RemoteEndpoint()
      ep.with_hostname("127.0.0.1")
      ep.with_port(5000)
      tp = taps.TransportProperties()

      tp.require("reliability")
      tp.prefer("preserve-msg-boundaries")

      self.preconnection = taps.Preconnection(remote_endpoint=ep,transport_properties=tp)
      self.preconnection.on_ready(self.handle_ready)
      self.connection = await self.preconnection.initiate()

if __name__ == "__main__":
   client = TestClient()
   asyncio.get_event_loop().create_task(client.main())
   asyncio.get_event_loop().run_forever()
```

Listing 8.2: A minimal client in PyTAPS

Appendix A

API documentation

NEATPy

Release 1.0.0

Michael Gundersen

May 15, 2020

ii

API DOCUMENTATION

This is the documentation for NEATPy: a transport system conforming to the specification of a transport system specified by the TAPS WG. While written in Python, it utilizes the NEAT codebase with the help of language bindings created by SWIG. In this way this transport system is logically divided in a front-end and back-end; the Python front-end presents a standards conforming API to the end user, while under the hood, it uses NEAT to handle all protocol machinery.

Missing implementation details:

The implementation of NEATPy is mostly based on version 4 and 5 of the interface draft by the TAPS. It implements all major objects, actions and events. However some implementations details are left out. The reason for this, is that successful implementation of these would require changes to NEAT, which was outside the scope of the thesis:

- *Selection Properties:*

Interface Instance or Type
Provisioning Domain Instance or Type
Use Temporary Local Address

- *Connection Properties:*

Retransmission Threshold Before Excessive Retransmission Notification
Connection Group Transmission Scheduler

- *Events:*

Soft Errors
Excessive retransmissions

- *Security Parameters:*

  NEATPy provides secure connections, with **TLS/TCP | DTLS/UDP | DTLS/SCTP**, but further customization, as specified in the interface draft is not implemented, due to constraints in NEAT. For the transport system to be fully security-conformant, further implementation of security within NEAT is needed.

- *Rendezvous Action:*

  Currently, the TAPS description is not very clear, and the full specification for the action is yet to be finalized.

Deviations:

- `Preconnection.start()`:

  The method is added to fulfill the methods `Preconnection.initiate()` and `Preconnection.listen()`, which respectively returns a Connection and Listener. The reason for this addition to NEATPy's interface is to facilitate the start of the event loop running in the within NEAT. This function starting the event loop does not return. To be able to return Connection and Listener objects this method is needed to start the transport system and with it the event loop.

CHAPTER

ONE

API

1.1 Preconnection

class Preconnection(*local_endpoint=None*, *remote_endpoint=None*, *transport_properties=None*, *security_needed=False*, *unfulfilled_handler=None*)

A Preconnection represents a set of properties and constraints on the selection and configuration of paths and protocols to establish a Connection with a remote Endpoint.

Parameters

- **local_endpoint** (Optional[LocalEndpoint]) – Optional local endpoint to be used with the Preconnection
- **remote_endpoint** (Optional[RemoteEndpoint]) – Optional remote endpoint to be used with the Preconnection
- **transport_properties** (Optional[TransportProperties]) – A transport property object with desired Selection Properties.
- **security_needed** (bool) – Indicated whether or not a secure connection is needed.
- **unfulfilled_handler** (Optional[Callable[[], None]]) – A function handling an unfulfilled error

add_framer(*framer*)

Adds a framer to the Preconnection to run on top of transport protocols. Multiple Framers may be added. If multiple Framers are added, the last one added runs first when framing outbound messages, and last when parsing inbound data.

Parameters **framer** – The framer to be added. Must inherit from the framer class and implement its abstract functions.

Return type None

initiate(*timeout=None*)

Initiate (Active open) is the Action of establishing a Connection to a Remote Endpoint presumed to be listening for incoming Connection requests. Active open is used by clients in client-server interactions. Note that start() must be called on the Preconnection.

Parameters **timeout** – The timeout parameter specifies how long to wait before aborting Active open.

Return type Connection

initiate_with_send(*message_data*, *sent_handler*, *message_context=None*, *timeout=None*)

For application-layer protocols where the Connection initiator also sends the first mes-

sage, the InitiateWithSend() action combines Connection initiation with a first Message sent. Returns a Connection object in the `establishing` state.

Parameters

- **message_data** (`bytearray`) – The message to be sent
- **sent_handler** (`Callable[[Connection, SendErrorReason], None]`) – A function / completion handler, handling both a successful completion and errors.
- **message_context** (`Optional[MessageContext]`) – Optional, used to indicate the message is idempotent, so it possibly can be used with 0-RTT establishment, if supported by the transport stack and system.
- **timeout** (`Optional[int]`) – The timeout parameter specifies how long to wait before aborting Active open.

Return type `Connection`

listen()

Listen (Passive open) is the Action of waiting for Connections from remote Endpoints. Before listening the transport system will resolve transport properties for candidate protocol stacks. A local endpoint must be passed to the Preconnection prior to listen.

Return type `Listener`

Returns A listener object.

start()

Starts the transport systems. Must be called after initiate / listen.

Return type `None`

Returns This function does not return.

1.2 Connection

class ConnectionState

An enumeration of the different states for a connection.

CLOSED = 4

CLOSING = 3

ESTABLISHED = 2

ESTABLISHING = 1

class MessageDataObject(*data*, *length*)

The messageData object provides access to the bytes that were received for a Message, along with the length of the byte array. It is passed to applications during the *receive* event, signaling a completion of a `receive()` call.

data = None

The raw bytes of the message

length = None

The message length

class Connection

A Connection represents a transport Protocol Stack on which data can be sent to and/or received from a remote Endpoint (i.e., depending on the kind of transport, connections can be bi-directional or unidirectional).

A Connection is created from a `preconnection` with active or passive open, or cloning, i.e it cannot be instantiated directly.

HANDLE_STATE_CLOSED = None
Handler for when the connection transitions to closed state

HANDLE_STATE_CONNECTION_ERROR = None
Handler for when the connection gets experiences a connection error

HANDLE_STATE_READY = None
Handler for when the connection transitions to ready state

abort()
Abort terminates a Connection without delivering remaining data.

Return type None

batch(*batch_block*)
Used to send multiple messages without the transport system dispatching messages further down the stack. Used to minimize overhead, and as a mechanism for the application to indicate that messages could be coalesced when possible.

Parameters **batch_block** (Callable[[], None]) – A function / block of code which calls send multiple times

Return type None

clone(*clone_error_handler*)
Calling Clone on a Connection yields a group of two Connections: the parent Connection on which Clone was called, and the resulting cloned Connection. These connections are "entangled" with each other, and become part of a Connection Group. Calling Clone on any of these two Connections adds a third Connection to the Connection Group, and so on. Connections in a Connection Group generally share connection_properties. However, there are exceptions, such as the priority property, which obviously will not trigger a change for all connections in the connection group. As with all other properties, priority is copied to the new Connection when calling Clone().

Parameters **clone_error_handler** (Callable[[Connection], None]) – A function to handle the event which fires when the cloning operation fails. The connection which clone was called on is sent with the handler.

Return type None

close()
Close terminates a Connection after satisfying all the requirements that were specified regarding the delivery of Messages that the application has already given to the transport system. For example, if reliable delivery was requested for a Message handed over before calling Close, the transport system will ensure that this Message is indeed delivered. If the Remote Endpoint still has data to send, it cannot be received after this call.

Return type None

get_properties()
Returns a dictionary consisting of the connections properties, which include the following:

- connection_state - key 'state'
- A boolean which holds the value for whether the connection can be used for sending - key 'send'
- A boolean which holds the value for whether the connection can be used for receiving - key 'receive'
- **A transport_properties object, which will differ with the connection's state - key 'props'**

- A connection in an establishing phase will hold transport properties that the application specified with the preconnection.
- A connection in either an established, closing or closed state will hold the selection_properties and connection_properties of the actual protocols that were selected and instantiated.

An example showing an application checking if the connection can be used for sending:

```
returned_props_dict = connection.get_properties()
can_be_used_for_sending = returned_props_dict['send']
if can_be_used_for_sending:
    ...
```

Return type None

receive(*handler*, *min_incomplete_length=None*, *max_length=inf*)

As with sending, data is received in terms of Messages. Receiving is an asynchronous operation, in which each call to Receive enqueues a request to receive new data from the connection. Once data has been received, or an error is encountered, an event will be delivered to complete the Receive request.

Parameters

- **handler** (Callable[[Connection, MessageDataObject, MessageContext, bool, bool], None]) – The function to handle the event delivered during completion, which includes both potential errors and successfully received data.
- **min_incomplete_length** (Optional[int]) – The default None value indicates that only complete messages should be delivered. Setting it to anything other than this will trigger a receive event only when at least that many bytes are available.
- **max_length** (int) – Indicates the maximum size of a message in bytes the application is prepared to receive. Incoming messages larger than this will be delivered in received partial events. To determine whether the received event is a partial event the application is able to check whether the variable is_end_of_message holds the boolean value *False*, which indicates a partial event, while a None value indicates a complete message being delivered.

Return type None

send(*message_data*, *sent_handler=None*, *message_context=None*, *end_of_message=True*)

Data is sent as Messages, which allow the application to communicate the boundaries of the data being transferred. By default, Send enqueues a complete Message, and takes optional per-message_properties. Applications are able to handle events with the :param sent_handler. This handles completion in form of an either an error or a successfully sent message.

Parameters

- **message_data** (bytearray) – The data to send.
- **sent_handler** (Optional[Callable[[Connection, SendErrorReason], None]]) – A function that is called after completion / error.
- **message_context** (Optional[MessageContext]) – Additional message_properties can be sent by adding them to a Message Context object *Optinoal*.
- **end_of_message** (bool) – When set to false indicates a partial send. All data sent with the same MessageContext object will be treated as belonging to the same Message, and will constitute an in-order series until the endOfMessage is marked.

Return type None

set_property(*connection_property*, *value*)
The application can set and query Connection Properties on a per-Connection basis. Connection Properties that are not read-only can be set during pre-establishment (see connection_properties) as well as on connections directly using the SetProperty action:

Parameters

- **connection_property** (ConnectionProperties) – The property to assign a value.
- **value** – The value to assign the property.

Return type None

1.3 Transport Properties

class TransportProperties(*property_profile=None*)
Transport properties is the collection of message_properties, selection_properties and connection_properties.

Parameters **property_profile** (Optional[TransportPropertyProfiles]) – Transport property profile to use

add(*prop*, *value*)
Add a property to the transport property object.

Parameters

- **prop** (Union[SelectionProperties, MessageProperties, ConnectionProperties]) – Property to add
- **value** – Value for given property

Return type None

avoid(*prop*)
Set the preference level to avoid for the passed selection property.

Parameters **prop** (SelectionProperties) – Selection property to set avoid as preference level for.

Return type None

default(*prop*)
Set the default preference level for the given selection property.

Parameters **prop** (SelectionProperties) – Selection property to reset to default preference level for.

ignore(*prop*)
Set the preference level to ignore for the passed selection property.

Parameters **prop** (SelectionProperties) – Selection property to set ignore as preference level for.

Return type None

prefer(*prop*)
Set the preference level to prefer for the passed selection property.

Parameters **prop** (SelectionProperties) – Selection property to set prefer as preference level for.

Return type None

prohibit(*prop*)
Set the preference level to prohibit for the passed selection property.

Parameters **prop** (SelectionProperties) – Selection property to set prohibit as preference level for.

Return type None

require(*prop*)
Set the preference level to require for the passed selection property.

Parameters **prop** (SelectionProperties) – Selection property to set require as preference level for.

Return type None

class TransportPropertyProfiles
Transport property profiles are used as a mechanism to pre-configure transport_properties objects, with frequently used sets of properties.

RELIABLE_INORDER_STREAM = 1
This profile provides a reliable, in-order transport service with congestion control. An example of a protocol that provides this service is TCP.

RELIABLE_MESSAGE = 2
This profile provides message-preserving, reliable, in-order transport service with congestion control. An example of a protocol that provides this service is SCTP.

UNRELIABLE_DATAGRAM = 3
This profile provides an unreliable datagram transport service. An example of a protocol that provides this service is UDP.

1.4 Selection Properties

class PreferenceLevel
An enumeration. Used when specifying a preference for selection_properties.

E.g an application specifying that a reliable transport is required would do this the following way:

```
transport_properties.add(SelectionProperties.RELIABILITY, PreferenceLevel.REQUIRE)
```

AVOID = -1

IGNORE = 0

PREFER = 1

PROHIBIT = -2

REQUIRE = 2

class SelectionProperties
An enumeration. Selection properties are used for application to specify applications requirements for transport and used for path and protocol stack selections. Selection properties are added to a transport_properties object.

CONGESTION_CONTROL = 'congestion-control'
Default preference_level.REQUIRE,

DIRECTION = 'direction'
Default communication_directions

INTERFACE = 'interface'
Default preference_level

LOCAL_ADDRESS_PREFERENCE = 'local-address-preference'
Default preference_level

MULTIPATH = 'multipath'
Default preference_level.PREFER

MULTISTREAMING = 'multistreaming'
Default preference_level.PREFER,

PER_MSG_CHECKSUM_LEN_RECV = 'per-msg-checksum-len-recv'
Default preference_level.IGNORE,

PER_MSG_CHECKSUM_LEN_SEND = 'per-msg-checksum-len-send'
Default preference_level.IGNORE,

PER_MSG_RELIABILITY = 'per-msg-reliability'
Default preference_level.IGNORE,

PRESERVE_MSG_BOUNDARIES = 'preserve-msg-boundaries'
Default preference_level.PREFER,

PRESERVE_ORDER = 'preserve-order'
Default preference_level.REQUIRE,

PVD = 'pvd'
Default preference_level

RELIABILITY = 'reliability'
Default preference_level.REQUIRE,

RETRANSMIT_NOTIFY = 'retransmit-notify'
Default preference_level.IGNORE

SOFT_ERROR_NOTIFY = 'soft-error-notify'
Default preference_level.IGNORE

ZERO_RTT_MSG = 'zero-rtt-msg'
Default preference_level.IGNORE,

1.5 Connection Properties

class CapacityProfiles
By specifying a Capacity Profile, an application is able to signal what kind of network treatment it desires. Under the hood the transport system will map each profile to different DSCP values for the given Connection.

CAPACITY_SEEKING = 10
Sending and receiving at the maximum rate allowed by the Connection's congestion controller.

CONSTANT_RATE_STREAMING = 28
Sending and receiving at a constant rate is desired. Minimal delay is wanted.

DEFAULT = 0
No explicit information for expected capacity profile is given.

LOW_LATENCY_INTERACTIVE = 36
An interactive Connection. Loss is preferred over latency.

LOW_LATENCY_NON_INTERACTIVE = 18
Loss is preferred to latency, but the Connection is non-interactive.

SCAVENGER = 1
A non-interactive Connection. The data is sent without any urgency for either sending or receiving.

class ConnectionProperties
Connection Properties represent the configuration and state of the selected Protocol Stack(s) backing a Connection. The application can set and query Connection Properties on a per-Connection basis. Connection Properties that are not read-only can be set prior to a call to either `initiate()` or `listen()`.

BOUNDS_ON_SEND_OR_RECEIVE_RATE = 'max-send-rate / max-recv-rate'
Default value is (-1, -1)

CAPACITY_PROFILE = 'conn-capacity-profile'
Default value is `CapacityProfiles`

CONNECTION_GROUP_TRANSMISSION_SCHEDULER = 'conn-scheduler'
Default value is "Weighted fair queueing"

MAXIMUM_MESSAGE_SIZE_BEFORE_FRAGMENTATION_OR_SEGMENTATION = 'singular-transmission-msg-max-len'
Default value is -1

MAXIMUM_MESSAGE_SIZE_CONCURRENT_WITH_CONNECTION_ESTABLISHMENT = 'zero-rtt-msg-max-len'
Default value is -1

MAXIMUM_MESSAGE_SIZE_ON_RECEIVE = 'recv-msg-max-len'
Default value is -1

MAXIMUM_MESSAGE_SIZE_ON_SEND = 'send-msg-max-len'
Default value is -1

PRIORITY = 'conn-prio'
Default value is 100

REQUIRED_MINIMUM_CORRUPTION_PROTECTION_COVERAGE_FOR_RECEIVING = 'recv-checksum-len'
Default value is -1

RETRANSMISSION_THRESHOLD_BEFORE_EXCESSIVE_RETRANSMISSION_NOTIFICATION = 'retransmit-notify-threshold'
Default value is -1

TIMEOUT_FOR_ABORTING_CONNECTION = 'conn-timeout'
Default value is -1

USER_TIMEOUT_TCP = 'tcp-uto'
An enumeration of three values, see `TCPUserTimeout`.

class TCPUserTimeout
These properties specify configurations for the User Timeout Option (UTO), in case TCP becomes the chosen transport protocol.

To set a property of TCP UTO, pass a dictionary with one or more properties:

```
tcp_uto_dict = {TCPUserTimeout.ADVERTISED_USER_TIMEOUT: 150}
transport_properties_object.add(ConnectionProperties.USER_TIMEOUT_TCP, tcp_uto_dict)
```

ADVERTISED_USER_TIMEOUT = 'tcp.user-timeout-value'
This time value is advertised via the TCP User Timeout Option (UTO) - Default 300 seconds.

CHANGEABLE = 'tcp.user-timeout-recv'
This property controls whether the Timeout for aborting Connection may be changed based on a UTO option received from the remote peer - Default *True*

USER_TIMEOUT_ENABLED = 'tcp.user-timeout'
This property controls whether the UTO option is enabled for a connection - Default False‘

1.6 Message Properties

class MessageProperties
Message Properties are used by the application to annotate the Messages they send with extra information to control how data is scheduled and processed by the transport protocols in the `connection`. Message Properties are sent by adding them to a `message_context`, and pass said context to the `send()` call.

CORRUPTION_PROTECTION_LENGTH = 'msg-checksum-len'
Default value is -1

EARLY_DATA = 'early-data'
Read only property

ECN = 'ecn'
Read only property

FINAL = 'final'
Default value is False

IDEMPOTENT = 'idempotent'
Default value is False

LIFETIME = 'msg-lifetime'
Default value is math.inf

MESSAGE_CAPACITY_PROFILE_OVERRIDE = 'msg-capacity-profile'
Default value is CapacityProfiles.DEFAULT

ORDERED = 'msg-ordered'
Default value is True

PRIORITY = 'msg-prio'
Default value is 100

RECEIVING_FINAL_MESSAGE = 'receiving-final-messages'
Read only property

RELIABLE_DATA_TRANSFER = 'msg-reliable'
Default value is True

SINGULAR_TRANSMISSION = 'singular-transmission'
Default value is False

1.7 Endpoints

class LocalEndpoint
This class holds information about a local endpoint. It could be passed when initiating a `preconnection`. Furthermore it is required when trying to establish a connection with a remote endpoint with `listen()`.

with_address(*address*)
This function sets the address desired to use with the local endpoint.

Parameters **address** (str) – The address to set.

Return type None

with_interface(*interface*)
This function sets the interface desired to with the local endpoint.

Parameters **interface** (str) – The endpoint to set.

Return type None

with_port(*port_number*)
This function sets the port desired to use with the local endpoint.

Parameters **port_number** (int) – The port to set.

Return type None

class RemoteEndpoint
This class holds information about a remote endpoint. It could be passed when initiating a preconnection. Furthermore it is required when trying to establish a connection with a remote endpoint with initiate().

with_address(*address*)
This function sets the address desired to use with the local endpoint.

Parameters **address** (str) – The address to set.

Return type None

with_hostname(*hostname*)
This function sets the hostname for the remote endpoint.

Parameters **hostname** (str) – The hostname to set.

Return type None

with_interface(*interface*)
This function sets the interface desired to with the local endpoint.

Parameters **interface** (str) – The endpoint to set.

Return type None

with_port(*port_number*)
This function sets the port desired to use with the local endpoint.

Parameters **port_number** (int) – The port to set.

Return type None

1.8 Framer

class Framer

abstract handle_received_data(*connection*)
Upon receiving this event, the framer implementation can inspect the inbound data. The data is parsed from a particular cursor representing the unprocessed data. The application requests a

specific amount of data it needs to have available in order to parse. If the data is not available, the parse fails.

Parameters **connection** – The connection the framer is registered with.

abstract new_sent_message(*connection*, *message_data*, *message_context*, *sent_handler*, *is_end_of_message*)

Upon receiving this event, a framer implementation is responsible for performing any necessary transformations and sending the resulting data back to the Message Framer, which will in turn send it to the next protocol.

:param connection:The connection the framer is registered with. :type message_data: `bytearray` :param message_data: The data to send. :param message_context: Additional `message_properties` can be sent by adding them to a Message Context object *Optinoal*. :type sent_handler: `Callable[[Connection, SendErrorReason], None]` :param sent_handler: A function that is called after completion / error. :param end_of_message: When set to false indicates a partial send.

abstract start(*connection*)

When a Message Framer generates a Start event, the framer implementation has the opportunity to start writing some data prior to the Connection delivering its Ready event. This allows the implementation to communicate control data to the remote endpoint that can be used to parse Messages.

Parameters **connection** – The connection that the framer is registered with.

class ExampleFramer

To provide an example this class implements the abstract interface in `framer` class. It's a simple TLV framer that frame TCP messages by prepending message size. This is then parsed by the same framer at the destination.

To use the framer, simply create an instance, and pass it to a `preconnection` like so:

```
new_preconnection = Preconnection(remote_endpoint=ep)
preconnection.add_framer(framer.ExampleFramer())
```

CHAPTER

TWO

CLIENT-SERVER EXAMPLE

Let us create a simple client and server with NEATPy!

Our server is going to reply *"Hello from server"* to all incoming messages, while our client will simply send a message, wait for a reply, then terminate the connection.

Let us start with the server!

2.1 Server

First we need to create a `local_endpoint` and specify which port we want to listen to:

```
local_specifier = LocalEndpoint()
local_specifier.with_port(5000)
```

We want a transport that is reliable and stream-oriented. To specify this we need to create a `transport_properties` object and set a `preference_level` for a couple of `selection_properties`:

```
transport_properties = TransportProperties()
# Selection properties can be set with the add call...
transport_properties.add(SelectionProperties.RELIABILITY, PreferenceLevel.REQUIRE)
# Or one of the convenient functions:
transport_properties.prohibit(SelectionProperties.PRESERVE_MSG_BOUNDARIES)
```

The next step is to create a `preconnection`, passing our local endpoint and transport properties as arguments. Next, we call `listen()`

```
new_preconnection = Preconnection(local_endpoint=local_specifier, transport_properties=tp)
new_listener: Listener = new_preconnection.listen()
```

To reply *Hello from server* to new incoming messages, and then terminate the connection we need to register two event handlers:

- One event handler that is registered for the listener, called when a new connection is established. This is registered with the member *HANDLE_CONNECTION_RECEIVED* of the listener class.
- We pass our second event handler with the send call for our reply, being fired with the 'sent' event.

The signatures of these event is listed in the documentation. The event handlers could be either full fledged functions or anonymous functions (in essence all objects that are callable), let us create one of each for demonstration:

```
def simple_connection_received_handler(connection, message, context, is_end, error):
    anon_func = lambda connection: connection.close()
    connection.send(b"Hello from server", anon_func)
```

The last step will be to register the event handler and call `preconnection.Preconnection.start()`.

```
new_listener.HANDLE_CONNECTION_RECEIVED = new_connection_received
new_preconnection.start()
```

Note: Calling start on the Preconnection starts the inner event loop of the transport system and does not return. Further interaction is achieved through the various events, e.g. the event signaling a Connection is received, manifested in the `HANDLE_CONNECTION_RECEIVED` member of the `listener` class.

That is it! Assuming we are running our program from the command line and using a main function, the typed out server looks like the following:

```
import neatpy

def simple_connection_received_handler(connection, message, context, is_end, error):
    anon_func = lambda connection: connection.close()
    connection.send(b"Hello from server", anon_func)

def main():
    local_specifier = LocalEndpoint()
    local_specifier.with_port(5000)

    transport_properties = TransportProperties()
    transport_properties.add(SelectionProperties.RELIABILITY, PreferenceLevel.REQUIRE)
    transport_properties.prohibit(SelectionProperties.PRESERVE_MSG_BOUNDARIES)

    new_preconnection = Preconnection(local_endpoint=local_specifier, transport_properties=tp)
    new_listener: Listener = new_preconnection.listen()

    new_listener.HANDLE_CONNECTION_RECEIVED = new_connection_received
    new_preconnection.start()

if __name__ == "__main__":
    main()
```

2.2 Client

To establish a connection to our server, we will first need to create a Remote Endpoint and specify the remote port and address:

```
remote_specifier = RemoteEndpoint()
remote_specifier.with_address("127.0.0.1")
remote_specifier.with_port(5000)
```

Following we create a `transport_properties` object, but this time we will use one of the `transport_profiles`. These functions as a convenience objects, pre-configured with frequently used sets of properties, and are passed on when initializing a `transport_properties` object:

```
transport_properties = TransportProperties(TransportPropertyProfiles.RELIABLE_INORDER_STREAM)
```

Next, just like with the server, we create a `preconnection` and pass out Remote Endpoint and Transport Properties:

```
new_preconnection = Preconnection(remote_endpoint=remote_specifier, transport_properties=transport_
↪properties)
new_connection = new_preconnection.initiate()
```

The last thing we need to do is to register our event handler for when the initiated connection is successfully established, and then start the transport system with

```
new_connection.HANDLE_STATE_READY = ready_handler
new_preconnection.start()
```

With our client we have two event handlers. One for handling when the Connection is successfully established while the last one is passed when calling `receive()`, handling a receive event:

```
def receive_handler(connection, message, message_context, is_end_of_message, error):
    print(f"Got message {len(message.data)}: {message.data.decode()}")
    connection.stop()

def ready_handler(connection: Connection):
    connection.send(b"Hello server", None)
    connection.receive(receive_handler)
```

Our client in full looks like the following:

```
def receive_handler(connection, message, message_context, is_end_of_message, error):
    print(f"Got message {len(message.data)}: {message.data.decode()}")
    connection.stop()

def ready_handler(connection: Connection):
    connection.send(b"Hello server", None)
    connection.receive(receive_handler)

def main():
    remote_specifier = RemoteEndpoint()
    remote_specifier.with_address("127.0.0.1")
    remote_specifier.with_port(5000)

    transport_properties = TransportProperties(TransportPropertyProfiles.RELIABLE_INORDER_STREAM)

    new_preconnection = Preconnection(remote_endpoint=remote_specifier, transport_properties=transport_
↪properties)
    new_connection = new_preconnection.initiate()
    new_connection.HANDLE_STATE_READY = ready_handler
    new_preconnection.start()

if __name__ == "__main__":
    main()
```

PYTHON MODULE INDEX

INDEX

P

R

S

T

U

W

Z

Bibliography

[1] Colin Allison, Martin Bramley and Jose Serrano. 'The World Wide Wait: Where Does the Time Go?' In: *24th EUROMICRO '98 Conference, Engineering Systems and Software for the Next Decade, 25-27 August 1998, Vesteras, Sweden*. IEEE Computer Society, 1998, pp. 20932–20938. DOI: 10.1109/EURMIC.1998.708124. URL: https://doi.org/10.1109/EURMIC.1998.708124.

[2] Ryan W. Bickhart. 'Transparent TCP-to-SCTP Translation Shim Layer'. MA thesis. University of Delaware, 2005. URL: http://www.cis.udel.edu/~amer/PEL/poc/pdf/BickhartMSthesis.pdf.

[3] Mike Bishop. *Hypertext Transfer Protocol Version 3 (HTTP/3)*. Internet-Draft draft-ietf-quic-http-27. Work in Progress. Internet Engineering Task Force, Feb. 2020. 61 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-quic-http-27.

[4] Robert T. Braden. 'Requirements for Internet Hosts - Communication Layers'. In: *RFC* 1122 (1989), pp. 1–116. DOI: 10.17487/RFC1122. URL: https://doi.org/10.17487/RFC1122.

[5] Anna Brunstrom, Tommy Pauly, Theresa Enghardt, Karl-Johan Grinnemo, Tom Jones, Philipp S. Tiesel, Colin Perkins and Michael Welzl. *Implementing Interfaces to Transport Services*. Internet-Draft draft-ietf-taps-impl-06. Work in Progress. Internet Engineering Task Force, Mar. 2020. 51 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-taps-impl-06.

[6] Marta Carbone and Luigi Rizzo. 'Dummynet revisited'. In: *Computer Communication Review* 40.2 (2010), pp. 12–20. DOI: 10.1145/1764873.1764876. URL: https://doi.org/10.1145/1764873.1764876.

[7] Vint Cerf, Yogen K. Dalal and Carl A. Sunshine. 'Specification of Internet Transmission Control Program'. In: *RFC* 675 (1974), pp. 1–70. DOI: 10.17487/RFC0675. URL: https://doi.org/10.17487/RFC0675.

[8] Stephen D. Crocker. 'New Host-Host Protocol'. In: *RFC* 33 (1970), pp. 1–19. DOI: 10.17487/RFC0033. URL: https://doi.org/10.17487/RFC0033.

[9] *ctypes — A foreign function library for Python*. URL: https://docs.python.org/3/library/ctypes.html. (last accessed: 15.04.2020).

[10] Stephen E. Deering. 'Host extensions for IP multicasting'. In: *RFC* 1112 (1989), pp. 1–17. DOI: 10.17487/RFC1112. URL: https://doi.org/10.17487/RFC1112.

[11] Jake Edge. 'Reducing Python's startup time'. In: (2017). URL: https://lwn.net/Articles/730915/. (last accessed: 29.04.2020).

[12] Lars Eggert and Godred Fairhurst. 'Unicast UDP Usage Guidelines for Application Designers'. In: *RFC* 5405 (2008), pp. 1–27. DOI: 10.17487/RFC5405. URL: https://doi.org/10.17487/RFC5405.

[13] Godred Fairhurst, Brian Trammell and Mirja Kühlewind. 'Services Provided by IETF Transport Protocols and Congestion Control Mechanisms'. In: *RFC* 8095 (2017), pp. 1–54. DOI: 10.17487/RFC8095. URL: https://doi.org/10.17487/RFC8095.

[14] Max Franke, Theresa Enghardt, Philipp S. Tiesel and Jake Holland. *PyTAPS - A TAPS implementation based on Python asyncio*. 2020. URL: pytaps.readthedocs.io/.

[15] *GDB: The GNU Project Debugger*. URL: https://www.gnu.org/software/gdb/. (last accessed: 15.04.2020).

[16] Josh Graessley, Tommy Pauly and Eric Kinnear. *Introducing Network.framework: A modern alternative to Sockets*. 2018. URL: https://developer.apple.com/videos/play/wwdc2018/715/. (last accessed: 31.03.2020).

[17] M. Handley. 'Why the Internet only just works'. In: *BT Technology Journal* 24 (2006), pp. 119–129.

[18] Fredrik Haugseth. 'Performance Evaluation of NEAT Internet Transport Layer API and Library'. MA thesis. University of Oslo, 2018. URL: http://urn.nb.no/URN:NBN:no-67127.

[19] Hugh Holbrook and Brad Cain. 'Source-Specific Multicast for IP'. In: *RFC* 4607 (2006), pp. 1–19. DOI: 10.17487/RFC4607. URL: https://doi.org/10.17487/RFC4607.

[20] *Host Software*. RFC 1. Apr. 1969. DOI: 10.17487/RFC0001. URL: https://rfc-editor.org/rfc/rfc1.txt.

[21] *IETF93 QUIC BarBoF: Protocol Overview*. URL: https://docs.google.com/presentation/d/15e1bLKYeN56GL1oTJSF9OZiUsI-rcxisLo9dEyDkWQs.

[22] Apple Inc. *Network.framework - Create network connections to send and receive data using transport and security protocols.* 2020. URL: developer.apple.com/documentation/network.

[23] *IPython pdb*. URL: https://github.com/gotcha/ipdb. (last accessed: 15.04.2020).

[24] Jana Iyengar and Martin Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. Internet-Draft draft-ietf-quic-transport-27. Work in Progress. Internet Engineering Task Force, Feb. 2020. 174 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-quic-transport-27.

[25] Richard B. Kalin. 'A Simplified NCP Protocol'. In: *RFC* 60 (1970), pp. 1–8. DOI: 10.17487/RFC0060. URL: https://doi.org/10.17487/RFC0060.

[26] Naeem Khademi, Zdravko Bozakov, Anna Brunstrom, Øystein Dale, Dragana Damjanovic, Kristian Riktor Evensen, Gorry Fairhurst, Andreas Fischer, Karl-Johan Grinnemo, Tom Jones, Simone Mangiante, Andreas Petlund, David Ros, Irene Rüngeler, Daniel Stenberg, Michael Tüxen, Felix Weinrank and Michael Welzl. 'Deliverable D2.3 - Final Version of Core Transport System'. In: (2017). URL: https://www.neat-project.org/wp-content/uploads/2017/10/D2.3.pdf.

[27] Naeem Khademi, David Ros, Michael Welzl, Zdravko Bozakov, Anna Brunström, Gorry Fairhurst, Karl-Johan Grinnemo, David A. Hayes, Per Hurtig, Tom Jones, Simone Mangiante, Michael Tüxen and Felix Weinrank. 'NEAT: A Platform- and Protocol-Independent Internet

Transport API'. In: *IEEE Communications Magazine* 55.6 (2017), pp. 46–54. DOI: 10.1109/MCOM.2017.1601052. URL: https://doi.org/10.1109/MCOM.2017.1601052.

[28] P. Lakhera and S. Cheshire. *Your App and Next Generation Networks*. 2015. URL: https://developer.apple.com/videos/wwdc/2015/?id=719.

[29] *libuv: Cross-platform asynchronous I/O*. URL: https://libuv.org/. (last accessed: 08.05.2020).

[30] *mypy - Optional Static Typing for Python*. URL: http://mypy-lang.org/. (last accessed: 21.04.2020).

[31] Henrik Frystyk Nielsen, James Gettys, Anselm Baird-Smith, Eric Prud'hommeaux, Håkon Wium Lie and Chris Lilley. 'Network Performance Effects of HTTP/1.1, CSS1, and PNG'. In: *Proceedings of the ACM SIGCOMM 1997 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, September 14-18, 1997, Cannes, France*. Ed. by Christophe Diot, Christian Huitema, Scott Shenker and Martha Steenstrup. ACM, 1997, pp. 155–166. DOI: 10.1145/263105.263157. URL: https://doi.org/10.1145/263105.263157.

[32] Stack Overflow. *Developer Survey Results 2019*. 2019. URL: https://insights.stackoverflow.com/survey/2019#key-results.

[33] Giorgos Papastergiou, Gorry Fairhurst, David Ros, Anna Brunström, Karl-Johan Grinnemo, Per Hurtig, Naeem Khademi, Michael Tüxen, Michael Welzl, Dragana Damjanovic and Simone Mangiante. 'De-Ossifying the Internet Transport Layer: A Survey and Future Perspectives'. In: *IEEE Communications Surveys and Tutorials* 19.1 (2017), pp. 619–639. DOI: 10.1109/COMST.2016.2626780. URL: https://doi.org/10.1109/COMST.2016.2626780.

[34] Tommy Pauly, Brian Trammell, Anna Brunstrom, Gorry Fairhurst, Colin Perkins, Philipp S. Tiesel and Christopher A. Wood. *An Architecture for Transport Services*. Internet-Draft draft-ietf-taps-arch-06. Work in Progress. Internet Engineering Task Force, Dec. 2019. 27 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-taps-arch-06.

[35] Jon Postel. 'Transmission Control Protocol'. In: *RFC* 793 (1981), pp. 1–91. DOI: 10.17487/RFC0793. URL: https://doi.org/10.17487/RFC0793.

[36] Jon Postel. 'User Datagram Protocol'. In: *RFC* 768 (1980), pp. 1–3. DOI: 10.17487/RFC0768. URL: https://doi.org/10.17487/RFC0768.

[37] *pyinstrument - Call stack profiler for Python.* URL: https://github.com/joerick/pyinstrument. (last accessed: 24.04.2020).

[38] *QUIC, a multiplexed stream transport over UDP*. URL: https://www.chromium.org/quic.

[39] Eric Rescorla. 'The Transport Layer Security (TLS) Protocol Version 1.3'. In: *RFC* 8446 (2018), pp. 1–160. DOI: 10.17487/RFC8446. URL: https://doi.org/10.17487/RFC8446.

[40] *Simplified Wrapper and Interface Generator (SWIG)*. URL: http://www.swig.org/. (last accessed: 14.04.2020).

[41] *Sphinx - Python Documentation Generator*. URL: https://www.sphinx-doc.org/en/master/. (last accessed: 17.04.2020).

[42] Randall R. Stewart. 'Stream Control Transmission Protocol'. In: *RFC* 4960 (2007), pp. 1–152. DOI: 10.17487/RFC4960. URL: https://doi.org/10.17487/RFC4960.

[43] Randall R. Stewart, Michael A. Ramalho, Qiaobing Xie, Michael Tüxen and Phillip T. Conrad. 'Stream Control Transmission Protocol (SCTP) Partial Reliability Extension'. In: *RFC* 3758 (2004), pp. 1–22. DOI: 10.17487/RFC3758. URL: https://doi.org/10.17487/RFC3758.

[44] Randall R. Stewart, Michael Tüxen, Salvatore Loreto and Robin Seggelmann. 'Stream Schedulers and User Message Interleaving for the Stream Control Transmission Protocol'. In: *RFC* 8260 (2017), pp. 1–23. DOI: 10.17487/RFC8260. URL: https://doi.org/10.17487/RFC8260.

[45] Randall R. Stewart, Michael Tüxen, Kacheong Poon, Peter Lei and Vladislav Yasevich. 'Sockets API Extensions for the Stream Control Transmission Protocol (SCTP)'. In: *RFC* 6458 (2011), pp. 1–115. DOI: 10.17487/RFC6458. URL: https://doi.org/10.17487/RFC6458.

[46] Brian Trammell, Michael Welzl, Theresa Enghardt, Gorry Fairhurst, Mirja Kühlewind, Colin Perkins, Philipp S. Tiesel, Christopher A. Wood and Tommy Pauly. *An Abstract Application Layer Interface to Transport Services*. Internet-Draft draft-ietf-taps-interface-05. Work in Progress. Internet Engineering Task Force, Nov. 2019. 64 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-taps-interface-05.

[47] Brian Trammell, Michael Welzl, Theresa Enghardt, Gorry Fairhurst, Mirja Kühlewind, Colin Perkins, Philipp S. Tiesel, Christopher A. Wood and Tommy Pauly. *An Abstract Application Layer Interface to Transport Services*. Internet-Draft draft-ietf-taps-interface-06. Work in Progress. Internet Engineering Task Force, Mar. 2020. 69 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-taps-interface-06.

[48] Felix Weinrank and Michael Tüxen. 'Transparent flow mapping for NEAT'. In: *2017 IFIP Networking Conference, IFIP Networking 2017 and Workshops, Stockholm, Sweden, June 12-16, 2017*. IEEE Computer Society, 2017, pp. 1–6. DOI: 10.23919/IFIPNetworking.2017.8264876. URL: https://doi.org/10.23919/IFIPNetworking.2017.8264876.

[49] Michael Welzl and Stein Gjessing. *A Minimal Set of Transport Services for End Systems*. Internet-Draft draft-ietf-taps-minset-11. Work in Progress. Internet Engineering Task Force, Sept. 2018. 50 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-taps-minset-11.

[50] Michael Welzl, Florian Niederbacher and Stein Gjessing. 'Beneficial Transparent Deployment of SCTP: The Missing Pieces'. In: *Proceedings of the Global Communications Conference, GLOBECOM 2011, 5-9 December 2011, Houston, Texas, USA*. IEEE, 2011, pp. 1–5. DOI: 10.1109/GLOCOM.2011.6133554. URL: https://doi.org/10.1109/GLOCOM.2011.6133554.

[51] Michael Welzl, Michael Tüxen and Naeem Khademi. 'On the Usage of Transport Features Provided by IETF Transport Protocols'. In: *RFC* 8303 (2018), pp. 1–56. DOI: 10.17487/RFC8303. URL: https://doi.org/10.17487/RFC8303.

[52] Christopher A. Wood, Theresa Enghardt, Tommy Pauly, Colin Perkins and Kyle Rose. *A Survey of Transport Security Protocols*. Internet-Draft draft-ietf-taps-transport-security-06. Work in Progress. Internet Engineering Task Force, Mar. 2019. 37 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-taps-transport-security-06.