Adding support for NEAT to iperf3

Fatemeh Ahmadi, Kim Dahlgren, Ari Karlsson Solaiman & Alexander Rabitsch

Faculty of Health, Science and Technology

Computer Science

Adding support for NEAT to iperf3

Fatemeh Ahmadi, Kim Dahlgren, Ari Karlsson Solaiman &
Alexander Rabitsch

WORKING PAPER | Januari 2018

Adding support for NEAT to iperf3

Fatemeh Ahmadi, Kim Dahlgren, Ari Karlsson Solaiman & Alexander Rabitsch

WORKING PAPER | Januari 2018

ISBN 978-91-7063-929-6 (pdf)

Distribution:
Karlstad University
Faculty of Health, Science and Technology
Department of Mathematics and Computer Science
SE-651 88 Karlstad, Sweden
+46 54 700 10 00

Print: Universitetstryckeriet, Karlstad 2018

WWW.KAU.SE

Adding support for NEAT to iperf3

Fatemeh Ahmadi
fatemeh.ahmadi058@gmail.com

Kim Dahlgren
kim.dahlgren93@gmail.com

Ari Karlsson Solaiman
ari.solaiman-@hotmail.com

Alexander Rabitsch
alexrabi100@student.kau.se

14th January 2018

Abstract

NEAT is an ongoing project which aims to re-enable the evolution of the transport layer of the Internet while at the same time providing an easy to use API for developers of networked software. Researchers behind the NEAT project has expressed an interest in a traffic generator which utilizes the NEAT framework in order to more easily conduct experiments, however, no such traffic generator has previously been available. The goal of this project was to deliver a working implementation of NEAT in iperf3. This report covers the overall design, implementation and design decisions made during the project.

Contents

1 Introduction

The NEAT project is a transport layer system currently in development that aims to provide a more flexible solution to how applications can interact with the Internet than what has previously been possible [8]. The standard BSD socket API has essentially left application developers with the choice between using the UDP and TCP transport protocols for transmitting their data, regardless of the actual needs of the application in question [3]. The BSD socket API has set a precedence for most network traffic to use either of the two previously mentioned transport protocols, which in turn has inhibited the development of new, experimental transport protocols [3]. The NEAT API allows applications to select network services such as security, low delay and reliability instead of necessarily specifying which transport protocol to use. The idea is to make networked applications easier to write while also allowing applications to dynamically take advantage of new network features as they become available without requiring rewriting of the code [8].

The NEAT project has so far been developed without having access to a proper traffic generator. While it would be possible to create a new traffic generator for this specific purpose, it would also be of interest to adapt an existing traffic generator. NEAT uses a single threaded approach, so for the sake of simplicity, it would be beneficial to adapt an existing traffic generator which follows the same principle as NEAT. While there may be many traffic generators that could fit this criteria, our project focuses on an implementation using iperf3 [1]. Our solution allows researchers such as those involved in the NEAT project at Karlstad University to conduct experiments using a familiar tool. The solution also aims to provide a future-proof version of iperf3 which can dynamically adapt to the evolution of the Internet. The iperf3-neat tool uses the NEAT system in order to generate traffic utilizing appropriate transport solutions based on the desired transport services.

2 Background

In this section we provide an introduction to the NEAT framework, as well as the iperf3 traffic generator tool. The NEAT framework is described in Section 2.1, whereas a description of iperf3 is provided in Section 2.2.

2.1 NEAT

NEAT is a transport layer system for network applications which is developed to address the problem of upgrading the internet transport layer as mentioned in Section 1. It is done in part by replacing the traditional socket API by a simpler and more flexible API. In order to achieve this, NEAT uses so called flows. Flows can, unlike sockets, be created without specifying which transport protocol to use when reading and writing data. Via the NEAT API, applications can instead specify the desired properties of the communication. NEAT is capable of polling flows internally, and will execute user-defined callbacks based on occurring events. Examples of such events include when a flow has successfully connected to a server, or when a flow may write data without blocking. The context is another new concept established in NEAT. A context is the essential environment for flows which provides services such as a DNS-resolver and an implementation of a Happy Eyeballs fallback algorithm for choosing transport protocol [4]. Because the same API is used regardless of operating system or which underlying protocol is used, writing networked applications can be considered easier in NEAT as compared to the socket API [7].

An application accesses the NEAT framework via the NEAT API. The different components of NEAT, such as the Policy Manager and Happy Eyeballs module, are connected via the NEAT Core (also known as the NEAT Logic). The NEAT Core is also responsible for handling the transport-layer communication once a connection has been established [6]. The NEAT architecture is depicted in Figure 1, where the different components are divided into four categories.

The *NEAT Framework components* are responsible for providing the most basic functionality required to run the NEAT transport system. This includes building blocks for flow creation and the ability to connect to a host name. These components are also able to translate the functionalities behind the NEAT User API into appropriate function calls. A NEAT flow contains a structure that keeps all of its relevant information. This structure provides access to additional flow information that extends the transport components and realises a transport independent interface. A NEAT flow can be assigned to one or more domain names as well as IP addresses. Both IPv4 and IPv6 addresses are supported. Another component in this category gathers statistics and information about the operation of the system for performance and diagnostic monitoring [2].

The *NEAT Transport components* provides the functionality to configure and manage transport protocols. This is done by ensuring that packets get through the network in the presence of unsupportive middleboxes, or for other types of challenging network paths. These components provide a path which can be used to introduce new transport protocols or transport protocol features. The NEAT transport components allow applications to use transport stacks implemented natively in operating systems as well as the transport stacks implemented in userspace. These components also implement the necessary functionalities to assign different priorities to each NEAT flow within a group of flows. In addition, the NEAT transport components handle end-to-end transport security, including encryption, integrity, and authentication for NEAT applications. The NEAT System provides secure connections using TLS and DTLS protocolos [2].

The *NEAT Selection components* are responsible for selecting an appropriate

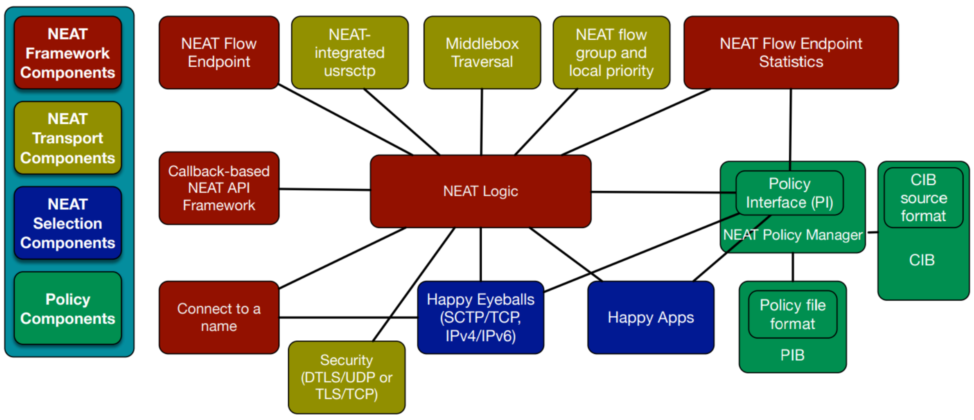
NEAT Framework Components
NEAT Transport Components
NEAT Selection Components
Policy Components
NEAT Flow Endpoint
NEAT-integrated usrsctp
Middlebox Traversal
NEAT flow group and local priority
NEAT Flow Endpoint Statistics
Callback-based NEAT API Framework
NEAT Logic
Policy Interface (PI)
NEAT Policy Manager
CIB source format
CIB
Connect to a name
Happy Eyeballs (SCTP/TCP, IPv4/IPv6)
Happy Apps
Policy file format
PIB
Security (DTLS/UDP or TLS/TCP)

Figure 1: The NEAT architecture. Each colour represent the different categories of components [2].

transport endpoint as well as components which can realise the required transport service. To select the required transport service the NEAT Selection components uses Happy Eyeballs in order to discover supported paths and an application layer feedback mechanism called Happy Apps. The NEAT selection process begins with the Policy Manager which combines the requirements from an application obtained through the NEAT User API with available transport protocols, transport-protocol parameters, feasible transport endpoints, IP addresses and port numbers. Together, they are used to create a list of candidates. Each candidate on the list has a priority, and the priorities are positive integers with zero being the highest priority. The Happy Eyeballs mechanism attempts to establish a connection for each of the candidates in the list and returns a handle to the first successfully established connection. Happy Apps offers selection mechanisms when the underlying transport protocol does not provide the signals required by the NEAT Core [2].

The *NEAT Policy components* are responsible for providing the possibility to manage and apply different policies. The NEAT System can seamlessly adapt to a wide range of scenarios using administrative policies. One important part of NEAT policy is the Policy Manager, which is responsible for generating a list of candidates that meet the requirements defined by the application for every new connection requested by the user. The Policy Manager will match the provided properties with information about the network stored in the Characteristics Information Base (CIB) as well as any applicable policies defined in the Policy Information Base (PIB). The CIB acts as a repository and stores information about available interfaces, supported protocols, network properties and current/previous connections between endpoints. The PIB acts as a repository that contains a collection of policies, where each policy consists of a set of rules which ties a set of matching requirements to a set of preferred or mandatory transport characteristics [2]. Figure 2 depicts the Policy components and how they interact.

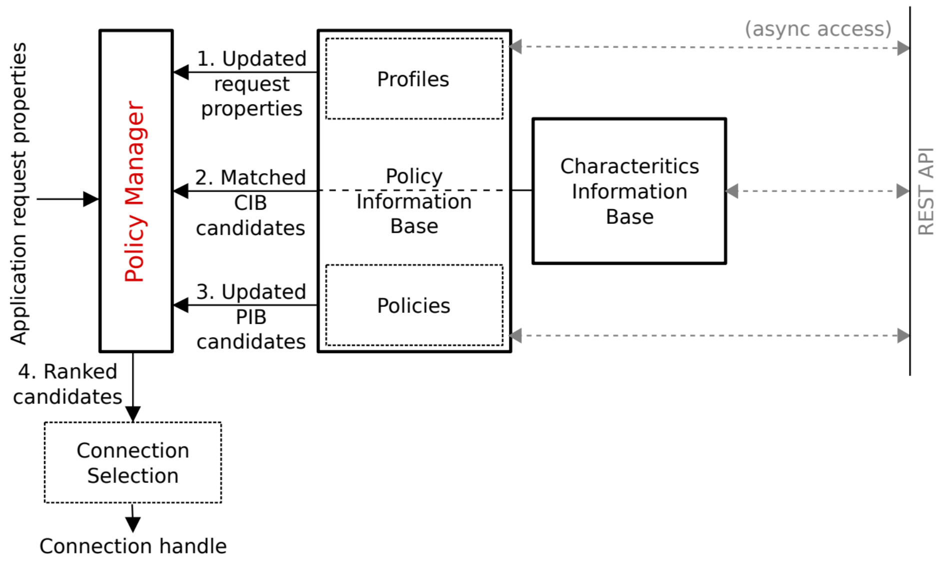
Application request properties
Policy Manager
1. Updated request properties
2. Matched CIB candidates
3. Updated PIB candidates
4. Ranked candidates
Profiles
Policy Information Base
Policies
Characteritics Information Base
(async access)
REST API
Connection Selection
Connection handle

Figure 2: NEAT Policy components and their interactions [2].

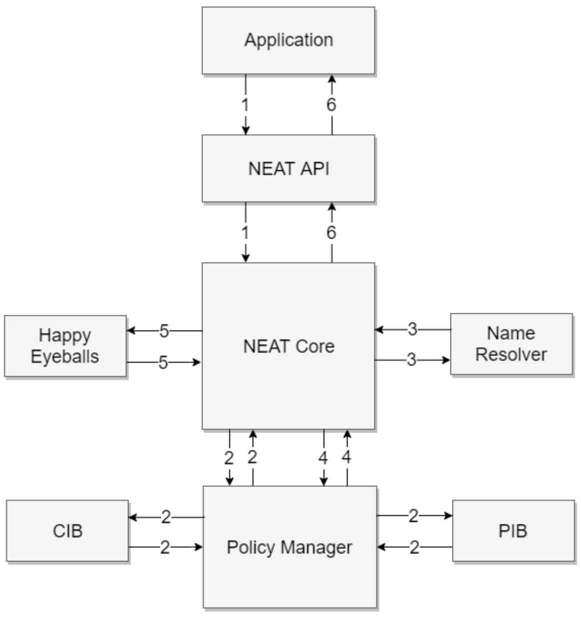

Figure 3: The NEAT workflow.

2.1.1 NEAT Workflow

The following sequence of events occurs when a NEAT connection is established [7]. The NEAT workflow is depicted in Figure 3.

1. First the application has to call the NEAT API, and provide the desired requirements (properties) for the connection to the server.
2. The NEAT Core transmits the specified properties to the Policy Manager, which has the responsibility of finding appropriate transport candidates. These candidates have to match the provided properties, the policies and the cached information. Next, the list of candidates is sent back to the NEAT Core.
3. The NEAT Core sends name resolution queries to the Name Resolver component to resolve the addresses for each of the candidates.
4. In order to determine a suitable candidate, the new list of candidates is returned to the Policy Manager. A candidate contains information such as transport protocol, interface, port, priority and the application properties.
5. The list of candidates is sent to the Happy Eyeballs module, which tries to establish a connection to each candidate in the order of their priorities. The first successful candidate is submitted to the NEAT Core.
6. NEAT checks the socket status and informs the application about the established connection and the state of the flow using callbacks.

2.2 iperf3

iperf3 is a tool developed by ESnet / Lawrence Berkeley National Laboratory for network traffic generation and network performance measurement tests [1]. Currently, iperf3 supports traffic generation using the TCP, UDP and SCTP transport protocols [5]. When running a test, iperf3 can report statistics such as periodic bandwidth, loss and jitter. The tool allows the user to tune the connection between an iperf3 server and client by specifying various parameters such as the window size, target bandwidth or how many streams to run in parallel [5].

An iperf3 test involves several steps. Figure 4 illustrates a scenario where a client-to-server test is run. The following steps provide a general description of how an iperf3 test is performed:

1. The user sets up a server by calling iperf3 with the appropriate flag, as well as any other parameters the user wishes to set, such as which port the server should use to listen for clients.
2. The server creates a TCP listener socket, which is used to listen for incoming clients as well as transmitting and receiving control messages during the test. Control messages include the parameter and result exchange, as well as communicating the current state of the test.

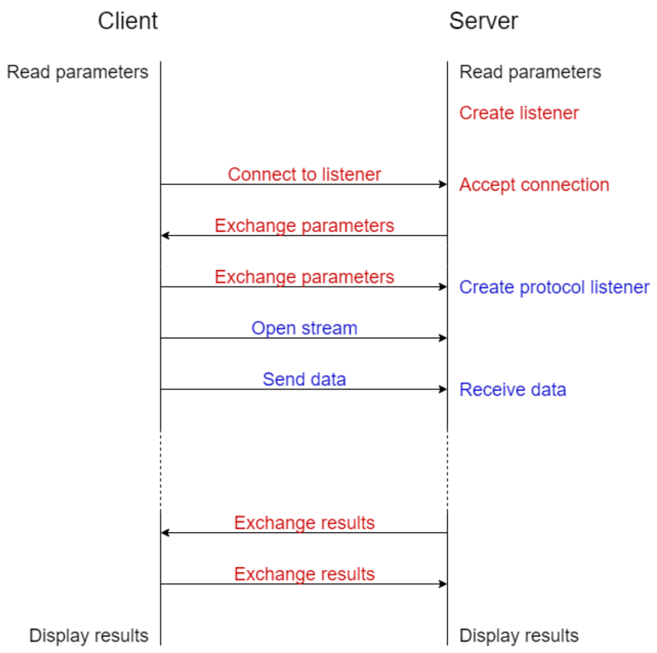

Figure 4: A Message Sequence Chart illustrating the different steps taken during an iperf3 test where the client sends data to the server. Events related to the control channel is shown in red, whereas events regarding streams are marked in blue.

3. The user sets up a client by calling iperf3 with the appropriate flags, as well as any other parameters the user wishes to set.
4. The client creates a socket to be used to connect the the server listener socket. If the server is not busy with another test, the server accepts the connection from the client.
5. Once the control channel between the server and client has been established, the client and server proceeds to exchange parameters over the control channel. The server creates a protocol listener socket of the type specified by the user. If no transport protocol has been specified by the user, the test defaults to TCP.
6. Once the parameters have been exchanged, the client attempts to establish the streams to be used during the test by connecting to the server protocol listener socket.
7. Once the server has accepted all streams as specified by the user, the test starts. Depending on the test mode, either the client transmits data and the server receives or vice versa.
8. Once the test has finished, the client and server exchange their results over the control channel which can then be displayed.

3 Implementation and design

In this section the design and implementation choices made during the development of the NEAT extension to iperf3 is detailed. Our intent was strictly to enable users to transmit data via iperf3 using the NEAT transport system instead of the standard socket API. As such, we decided that any other functionality in iperf3 that is not directly related to this purpose should be left intact. Our changes primarily affect the functionality outlined in Step 5 to 7 in Section 2.2.

In Section 3.1, key decisions based around the control channel used by iperf3 are described. The interaction between iperf3 and the NEAT API regarding properties, and how a user may provide properties to the system is described in Section 3.2. Section 3.3 outlines some important data structures used by iperf3 in order to create and transmit data through streams, and how these structures were adapted in order to make it possible to utilize streams via NEAT. Compatibility issues with other versions of iperf3 is detailed in Section 3.4

3.1 Control channel

iperf3 uses the control channel primarily during the start-up and ending phases of a test (see Section 2.2). It is used for the initial connection between the client and server, and is responsible for the communication of changes of the state of a test (including failures), as well as the exchanging of results after a test has completed. The control channel uses a TCP socket for communication. Once the initial connection between the client and server has been established, the control channel is regularly polled along with any other sockets used by the test. If a message has been received on the control channel, it is promptly read and the state of the test is changed accordingly.

While it would be possible to re-implement the control channel in order to take advantage of NEAT, the benefits of this endeavour would be limited. Using a NEAT-enabled control channel could make the it marginally more flexible, since the channel would be protocol agnostic. However, we expect TCP to be available on virtually any path.

When creating streams, NEAT may encounter problems connecting to the same IP address and port combination as the control channel. Thus, a NEAT-enabled stream will attempt to establish a connection to the previously used port number incremented by one.

```
$ iperf3 -c <ip-address-to-server> --neat <filepath-to-properties>
```

Figure 5: Example of how a to start an iperf3 client using the NEAT extension.

```
{
 "transport": [
   {
     "value": "SCTP",
     "precedence": 1
   },
   {
     "value": "SCTP/UDP",
     "precedence": 1
   },
   {
     "value": "TCP",
     "precedence": 1
   }
 ],
 "transport_type": {
   "value": "stream",
   "precedence": 1
 }
}
```

Figure 6: Example of NEAT properties.

3.2 NEAT Properties

In order to utilize the NEAT Policy Manager, a NEAT client is required to inform the NEAT Core about which properties are desired or required by the application. The Policy Manager will then attempt to match the given properties with user-defined policies provided in one or more JSON-formatted policy files. Properties are represented as JSON objects, and many properties may be represented by a single JSON object. The structure of a property is name, a value and a precedence. The name of a property is always a string, however, the value can also be a string, a Boolean, an integer, a float, an array or an interval. The precedence can be either be set to '1' for multiple protocols or '2' for one protocol. The user can chose to send the properties to the Policy Manager, which will attempt to match the properties against the specified policies and return a list of candidates which are ranked depending on the property. An example of a set of NEAT properties is shown in Figure 6.

NEAT can take properties as input to choose which protocol to use when transporting data between client and a server by using the Policy Manager. Since it is not possible to predict which NEAT properties and policies a user wishes to use at design time, it was decided that the NEAT extension to iperf3 should allow a user to define which properties should be used. Furthermore, since NEAT policies and properties offer a wide range of possibilities, we considered that the only plausible solution was to require the user to provide a file containing the desired properties in JSON format when calling the NEAT extension. Figure 5 provides an example of how to call the NEAT extension. If a property file with an invalid format, or if no property file was included during the call to the iperf3 extension, the user will get an error message. We initially used a default state if an invalid property file was included, which uses a default property specifying that TCP should be used when opening a connection.

3.3 NEAT-enabled Streams

In order to extend iperf3 to support the NEAT transport system, functionality to allow iperf3 to create and utilize streams through the NEAT API rather than the traditional BSD socket API needed to be implemented.

Each iperf3 test is represented by an *iperf_test* data structure, see Figure 7, which contains various settings and data structures used throughout the duration of the test. Most importantly, the structure contains a pointer to the *protocol* data structure, see Figure 9, to be used during the creation and usage of each stream in the test, as well as a singly-linked list of *iperf_stream* data structures, see Figure 8 representing the streams.

The *protocol* data structure, represents the transport protocol to be used during the test. This structure contains a set of callbacks that specify the behavior of the streams used in the test. These callbacks include:

- *listen*: this callback is used by the server after the server and client has exchanged parameter in order to create a listener socket for the transport protocol specified by the user. This socket is used to listen for incoming connections from the client during the stream creation phase.
- *connect*: this callback is used by the client in order to establish a connection between the client and server during the stream creation phase.
- *accept*: this callback is used by the server to accept incoming connections from the client during the stream creation phase. The channels created during the stream creation phase will be used to send or receive data.
- *send*: this callback is used by the sender in order to transmit data as well as to calculate how much data was sent on a stream. The *iperf_stream* data structure representing each stream inherits this callback during creation.
- *recv*: this callback is used by the receiver to read data as well as calculating how much data was received from a stream. As with the send callback, the *iperf_stream* data structure inherits this callback.

The *iperf_stream* data structure contains information about each stream, such as the local and remote ports and addresses, the underlying socket used by the stream as well as the callbacks used when sending and receiving data. The results from calling these callbacks are stored in the *iperf_stream_result* data structure.

To prevent excessive redesign of the internal structure of iperf3, we decided to reuse as many existing structures as possible. Thus, the NEAT extension was implemented as if it were a transport protocol. This was done by implementing equivalent versions of the protocol callbacks described previously, using the NEAT API instead of the standard socket API. These callbacks were contained in a new instance of the *protocol* structure set during the initiation of

```
struct iperf_test {
 ...
 struct protocol *protocol;
 ...
 int ctrl_sck;
 int listener;
 int prot_listener;
 ...
 int max_fd;
 fd_set read_set;
 fd_set write_set;
 ...
 SLIST_HEAD(slisthead, iperf_stream) streams;
 struct iperf_settings *settings;
 SLIST_HEAD(plisthead, protocol) protocols;
 ...
};
```

Figure 7: The *iperf_test* data structure.

```
struct iperf_stream {
 struct iperf_test *test;
 int local_port;
 int remote_port;
 int socket;
 int id;
 struct iperf_settings *settings;
 struct iperf_stream_result *result;
 ...
 char *buffer;
 ...
 struct neat_ctx *ctx;
 struct neat_flow *flow;
 int flow_number;
 ...
 struct sockaddr_storage local_addr;
 struct sockaddr_storage remote_addr;
 ...
 int (*snd)(struct iperf_stream *);
 int (*rcv)(struct iperf_stream *);
 ...
 SLIST_ENTRY(iperf_stream) streams;
 ...
};
```

Figure 8: The *iperf_stream* data structure.

```
struct protocol {
 int id;
 char *name;
 int (*accept)(struct iperf_test *, struct iperf_stream *);
 int (*listen)(struct iperf_test *);
 int (*connect)(struct iperf_test *, struct iperf_stream *);
 int (*send)(struct iperf_stream *);
 int (*recv)(struct iperf_stream *);
 int (*init)(struct iperf_test *);
 SLIST_ENTRY(protocol) protocols;
};
```

Figure 9: The *protocol* data structure.

the *iperf_test* structure. Pointers to the NEAT context and flow used by each stream were added to *iperf_stream* data structure.

When using the NEAT API, listening for and accepting a connection can be made with a single call. Thus, it was considered unnecessary to implement both the listen and accept callbacks. The accept callback handles the functionality of both the accept and listen callbacks, as the listen callback will not be called.

During a test, iperf3 regularly polls each stream and the control channel for activity before executing. Since the NEAT API is capable of internally polling flows for events, it was decided that iperf3 should skip polling streams manually by the application when the NEAT extension is used. Since the control channel was not changed during the implementation of the NEAT extension, it is polled manually.

3.4 Compatibility

The NEAT extension to iperf3 is not fully compatible with versions of iperf3 lacking support for NEAT - a test involving an iperf3 client using the NEAT extension cannot connect to an iperf3 server without the NEAT extension. This is mainly because of two reasons. The first reason is described in Section 3.1 - the NEAT extension is expecting different port numbers than a 'normal' iperf3 server accepts. The other reason is because of the design decisions described in Section 3.3 - the NEAT extension is implemented as if it were a transport protocol. The client will attempt to set the transport protocol of the current test to the NEAT protocol during the parameter exchange phase, however, no such transport protocol is implemented in the server.

4 Testing

As a requirement for this project, the principal functionalities of iperf3-neat should be tested. We chose to do black box testing using a shell script, which runs some command line options to tune the connections and transmit data.

This approach has been chosen because of its simplicity and maintainability aspects. Since there was already such a shell script for iperf3, it was considered appropriate to extend that script for NEAT. This approach enables running of tests for the standard iperf3 functionalities as well as new test commands for the NEAT extension.

4.1 Experimental setup

The primary setup we used for testing our implementation of NEAT in iperf3 was just using our laptops and running a test locally on the loopback interface. This worked pretty well for testing purposes but later on in the experiment we needed to actually test if our implementation would work in a real environment. At this time, we decided to setup a computer to act as the server and use one of our laptops as a client. The computer that we set up was an older computer that we received from the product owner for the purpose of testing. The computers used during these tests were running Ubuntu 16.04.

5 Problems

During the development of the NEAT extension for iperf3, we encountered a few different problems, which are covered in this section. The first major problem we encountered was to understand how iperf3 worked and what the internal structure looked like, since iperf3 lacks proper documentation. This was quite a time consuming process since we essentially had to go through the code line-by-line. In contrast, the NEAT API is fairly well documented, so the process to learn how to utilize the API was not a big hurdle to overcome in comparison.

During the implementation it was very hard to split up the work into tasks. Once we had understood the architecture of iperf3 and how to use the NEAT API, there was surprisingly little to code to write in order to get a working solution. Because of this, it became hard to coordinate different people working on this implementation. This was not helped by the fact that the code had to be tested often in order to verify its correctness.

6 Conclusion

The goal of this project was to extend the traffic generation and network measurement tool iperf3 with support for the NEAT transport system. Our solution could provide stakeholders of the NEAT Project, such as researchers at Karlstad University, an easy to use and familiar tool which is capable of generating and tuning network traffic utilizing the NEAT framework. In order to achieve this, we added support for creating and utilizing NEAT enabled streams in iperf3 by extending data structures already available in iperf3 as well as implementing the necessary callbacks needed by iperf3 and the NEAT API. This solution has worked well, since it has allowed us to implement these new features in iperf3 without having to rewrite large parts of the system. A drawback of this solution is limited backward compatibility with standard iperf3 instances. In the future, such compatibility issues could possibly be resolved.

Our solution has mainly focused on the traffic generation aspect of the extension. Another area of interest for future work involve more sophisticated support for measuring the traffic generated by our extension. In the future, support for calculating and displaying statistics such as loss rates, jitter etc. depending on which underlying transport protocol is chosen by the NEAT system could be implemented.

References

[1] ESnet. iperf3. Available at: `http://software.es.net/iperf/`, 2017. [Online; accessed: 2017-12-11].

[2] Naeem Khademi et al. Deliverable D2.2 - Core Transport System, with both Low-level and High-level Components. 2017.

[3] David Hayes Grinnemo, Per Hurtig, Tom Jones, Simone Mangiante, Michael Tüxen, and Felix Weinrank. NEAT: A Platform-and Protocol-Independent Internet Transport API.

[4] Karl-Johan Grinnemo, Anna Brunstrom, Per Hurtig, Naeem Khademi, and Zdravko Bozakov. Happy Eyeballs for Transport Selection. Internet-Draft draft-grinnemo-taps-he-03, Internet Engineering Task Force, July 2017. Work in Progress.

[5] Iperf.fr. iPerf - The ultimate speed test tool for TCP, UDP and SCTP. Available at: `https://iperf.fr/`. [Online; accessed: 2017-11-21].

[6] The NEAT Project. Architecture. Available at: `https://neat.readthedocs.io/en/latest/arch.html`. [Online; accessed: 2017-11-22].

[7] The NEAT Project. Welcome to the NEAT tutorial. Available at: `https://neat.readthedocs.io/en/latest/tutorial.html`. [Online; accessed: 2017-11-22].

[8] The NEAT Project. NEAT - A New, Evolutive API and Transport-Layer Architecture for the Internet. Available at: `https://www.neat-project.org/`, 2017. [Online; accessed: 2017-12-22].

A Using the iperf3 extension

The following is a description of the requirements for installing and running iperf3 with the NEAT extension (Linux).

A.1 Installing NEAT

Before installing the NEAT transport system, some dependencies are required to be installed. These are:

- cmake
- libuv (at least version 1.9 or later)
- ldns
- ljansson (at least version 2.7 or later)
- libmnl (for Linux users)
- libsctp-dev (for Linux users)
- swig

Once all dependencies are installed, open up a terminal and locate the NEAT source directory and execute the following commands:

```
$ mkdir build && cd build
$ cmake
$ cmake --build
$ sudo make install
```

More detailed instructions can be found at the official NEAT-project GitHub page: `https://github.com/NEAT-project/neat`

A.2 Installing iperf3 with the NEAT extension

Make sure that the NEAT transport system and API is installed before attempting to install iperf3 with the NEAT extension. To build and install iperf3, locate the iperf3 source directory and run the following commands:

```
$ ./configure
$ make
$ sudo make install
```

More information can be found at: `https://github.com/esnet/iperf`

A.3 NEAT Policy Manager

The Policy Manager is not a requirement for running NEAT but it is required if the user wants to utilize properties. The Policy Manager will give a list of candidates and choose the best protocol to use depending on the properties given. In order to run the Policy Manager, some dependencies needs to be installed:

- python3-pip
- netifaces
- aiohttp

To run the Policy Manager, locate the 'policy' directory located in the NEAT source directory and execute the following command:

```
$ python3.5 ./neatpmd --cib <location-to-cib> --pib <location-to-pib>
```

For more information, please visit: `https://github.com/NEAT-project/neat/tree/master/policy`

Adding support for NEAT to iperf3

NEAT is an ongoing project which aims to re-enable the evolution of the transport layer of the Internet while at the same time providing an easy to use API for developers of networked software. Researchers behind the NEAT project has expressed an interest in a trac generator which utilizes the NEAT framework in order to more easily conduct experiments, however, no such trac generator has previously been available. The goal of this project was to deliver a working implementation of NEAT in iperf3. This report covers the overall design, implementation and design decisions made during the project.

Faculty of Health, Science and Technology

WORKING PAPER | Januari 2018